CASE WESTERN RESERVE UNIVERSITY
Case School of Engineering
Department of Electrical Engineering and Computer Science
EECS 337 Systems Programming (Compiler Design)
Fall 2013
Assignment #3 Due: September 17, 2013
30 points + 5 extra credit

Part 1: Reading and Exercises
Compilers - Principles, Techniques, & Tools, 2nd Edition, Sections 2.7, 2.8, 3.1, 3.3
Homework Exercises – problems 3.3.7, laboratory assignment and answer the questions at the end of the assignment.

Introduction
In this assignment, you download a tar file and create the basic components of an ANSI C compiler (main, scanner, parser). This compiler version shall produce the equivalent output of the previous Echo2.c program. The code is based off a lex and yacc source code contribution from Jeff Lee from 1985. Coding standards have changed since that time so an updated version is provided for you as a working base.

The updated version adds an include file (yystype.h) to share common definitions between the three program files (main.c, scan.l, gram.y). You are provided with the original source files (c-grammar) to compare the differences between the versions. You are welcome to use the original source or the updated version as your starting point. Part of the laboratory instructions includes how to start working with the original source code. If you decide to start with the updated version than you can skip over that section of the assignment.

The code starts as a skeleton scanner and parser for an ANSI C compiler and with each assignment you will build upon the code base. Over each assignment you add compiler features by implementing a lexical scanner, symbol table, parser actions and code generator functions.

Part 2: Laboratory

From a console window, make a directory on your computer in your EECS337 directory under your Case ID and call it hw03. Then change directory to the hw03 directory.
mkdir EECS337/caseid/hw03/ ; where caseid is YOUR Case ID, enter all in lower case
cd EECS337/caseid/hw03/

Download a copy of the hw03_caseid.tar file from http://blackboard.case.edu/ in the EECS337 homework assignment area to the hw03 directory.

To untar the tar file type the command.
tar xvf hw03_caseid.tar

Your directory structure should now contain the files shown below:
ls -lsag
Makefile
gram.y
hw03_caseid.tar
hw03_test.sh
main.c
original/c-grammar
scan.l
tokens2.c
yystype.h

Part 2 A: Original Source Code
Change directory into the original sub-directory and execute the self-extracting shell script.
cd original/
./c-grammar

The shell script is self-extracting and will create the following files:
main.c
scan.l
gram.y
Makefile
README

Look at the README file to acknowledge the contribution from Jeff Lee for providing this sample code. This example code will be the foundation of the ANSI C compiler for this semester.

(Optional, GOTO Part 3) Edit the Makefile to change the mistake in creating the output file. Change the line with `$(OBJ)/ansi_c` to `$(OBJ) -o ansi_c`. Then add into the `rm -f y.tab.h y.txtput *.o` line the file names `scan.c` and `gram.c`. You are now ready to run ***make***. The files should build and create an executable file (ansi_c). To clean the build type ***make clean*** and the old objects will be removed. Then type **make** to build everything again.

There are times when you may want to look at the output files created by lex and yacc. To create the C version of the scanner or the parser type:
lex -t scan.l > scan.c ; to create the C version of the lex file

yacc -dv gram.y ; to create the C version of the yacc file (y.tab.c, y.tab.h y.output)
mv -f y.tab.c gram.c ; rename it gram.c

This version of the compiler uses the standard input (stdin) for reading inputs, writes data to the standard output (stdout) and the error messages to the standard error (stderr). In the updated version the program opens the files directly from the command line. Notice on the build of scan.l there are three warning messages.
"scan.l", line 54: warning, rule cannot be matched
"scan.l", line 56: warning, rule cannot be matched
"scan.l", line 58: warning, rule cannot be matched

Edit the scan.l file and notice the warning messages are due to duplicate patterns being in the lex file. Go to each line number (largest first) and remove the three duplicate lines. Build the files again to show the warnings are now gone.

Edit main.c and add a standard header to the top of the file and change the main() function to properly include the normal C language command line arguments. Then save the file.

```
/*****************************************************************************
*
*
* FILE:       main.c
*
* DESC:       EECS 337 Assignment 3
*
* AUTHOR:     caseid
*
* DATE:       September 10, 2013
*
* EDIT HISTORY:
*
*****************************************************************************/
int    main( int argc, char *argv[])
```

Edit the gram.y file and add a standard header to the top of the file along with the missing declarations section. The YYDEBUG feature includes verbose debug information.

```
/*****************************************************************************
*
*
* FILE:       gram.y
*
* DESC:       EECS 337 Assignment 3
*
* AUTHOR:     caseid
*
* DATE:       September 10, 2013
*
* EDIT HISTORY:
*
*****************************************************************************/
%{
#include "y.tab.h"
#define       YYDEBUG       1
%}
```

Edit the scan.l file and add a standard header to the top of the file. Then save the file.

```
/*****************************************************************************
*
*
* FILE:       scan.l
*
* DESC:       EECS 337 Assignment 3
*
* AUTHOR:     caseid
*
* DATE:       September 10, 2013
*
* EDIT HISTORY:
*
*****************************************************************************/
```

Edit the Makefile and add a standard header to the top of the file. Notice the comments are defined as # to end of the line. Add the .KEEP_STATE: line below the header and save the file.

```
#-----------------------------------------------------------------------------
#
# FILE:             Makefile
#
# DESC:             EECS 337 Assignment 3
#
# AUTHOR:           caseid
#
# DATE:             September 10, 2013
#
# EDIT HISTORY:
#
#-----------------------------------------------------------------------------
.KEEP_STATE:
```

Build the ansi_c compiler again and test with the main.c function. The main source code should echo back to the screen with no syntax errors and notice part of the comments are missing. You can compare this version of the source code to the updated version using the diff command. If you decide to use this version as your starting point, copy the files up a directory and overwrite the updated version. You are now ready to solve the laboratory assignment in Part 4.

./ansi_c < main.c

diff main.c ../main.c

Part 3: Laboratory Introduction – Command Line Flags
From a console window, change directory into the hw03 directory.
cd ~/EECS337/caseid/hw03/

In this assignment, you build and test the "ansi_c" compiler and start to examine the capabilities and limitations of the scanner and parser. This updated version uses the same file names as the original ansi_c compiler and corrects some old coding standards. The updated version of the compiler uses an include file (yystype.h) and adds initialization and exit routines before and after the call to the compiler or yyparse(). The main routine also adds command line flags and handles input file names for processing. The usage command is:

Usage: ***ansi_c [[+|-]echo] [[+|-]debug] [[+|-]yydebug] [filename] [...]***

key: [] is an optional parameter and | is either or.
Where +echo turns on the input source code echo function and -echo turns it off.
Where +debug turns on embedded debug statements and -debug turns them off.
Where +yydebug turns on the yacc parser debugging function and –yydebug turns it off.
Where filename is a source file name for the compiler.
Where [...] flags and file names can be entered in any order.

By default all flags are initially off and the compiler uses the standard input (stdin). The flags do not effect redirecting input source files or output (stdout) and error (stderr) results. Example:

./ansi_c +echo +debug +yydebug < original/main.c > main.txt

In this case the echo, debug and yydebug flags are turned on, the main.c file is sent to the standard input and the output results (stdout) are sent to the main.txt file. Any error messages generated by the compiler are sent to the standard error (stderr), which is the console. If an input command is not a valid flag or the input file name does not exist, then the usage message is displayed and the software exits at that point. The +flag turns the flag on and -flag turns the flag off. Using the command line parameters, it is now possible to compile multiple input files and change the flags between each compile operation. Example:

The +yydebug flag provides a verbose output showing all the yacc stack values from the current compiler operation. The +debug flag provides a verbose output of all the debug statements. The y.output file is created by the yacc -dv gram.y operation and provides a complete verbose listing of all the compiler states. We will be looking at the y.output file in a future assignment. The +debug flag turns on the debug statements inside the parser and are normally used while crafting a compiler. Once all the stages of the compiler are completed then the #define YYDEBUG statement is removed from the yystype.h file to allow for an optimal build of the software.

./ansi_c +echo original/main.c -echo +yydebug original/main.c
In this case the echo function is turned on, main.c is compiled, the echo function is turned off, the yydebug flag is turned on and main.c is compiled. The command line flag routines compare the input command line with the flag names based on the length of the input string. In this case +e will set the echo function the same as +echo. The shortest strings to set the flags are: + for echo, +d for debug and +y for yydebug.

Part 3 A: Laboratory Source Code
Look at the file yystype.h and examine the DATA structure definition. The DATA structure defines all the global data for the compiler and the structure is declared in main.c. All the global variables are "extern" so each variable is visible from the main.c, scan.l and gram.y files. Using one data structure to hold all the compiler variables allows for changing the compiler from using a static memory (data.member) model to using a dynamic memory (data->member) model. This feature allows multiple versions of the compiler to run in parallel on the same processor.

Each source code file includes all the external definitions with `#include "yystype.h"`. All C function prototypes from main.c, scan.l and gram.y have an external reference in this file to allow function calls across the three files. The tokens defined in the gram.y file are defined in the y.tab.h file and included inside this file.

Look at the main.c file. Because all global data is defined in the DATA structure the values must be initialized before starting the compiler (java constructor). In this case we use `memset` on the `sizeof` the structure and clear all the data fields. This would be the same as initializing each field (`data.flags = 0; data.column = 0;`). It is possible for the initialization routine to encounter an error and in that case would return a value other than zero. For example there could be a statement to allocated dynamic memory or open a file, which could return an error. We use the address of the structure and type cast it to a void pointer for the `memset` function. The main routine processes the status of the initialization function and on an error, reports the error and exit at that time.

```
int    main_init( void)
{
       memset((void*)&data, 0, sizeof( DATA));
       return 0;
}
```

There are normally things that need to be cleaned up after the compiler completes (calculate total statistics, de-allocate dynamic memory, close files, close devices) so there is an exit routine after the main control loop (java destructor). At this time there is nothing to clean up so the exit function just returns a zero. We will add statements in a future assignment.

```
int    ansi_c_exit( void)
{
       return 0;
}
```

The command line parameters are now processed by the `main_process_flags()` routine. The `argc` parameter is an integer and represents the number of `argv` string parameters. A value of 1 means only the name of the program was typed at the command line (`./ansi_c`) and the string is normally the name of the executable program. A value of 2 means a command line string was entered along with the program name. The strings are contained in the `argv` parameter. Because `argv` is a character pointer, by incrementing the pointer, the software knows how to increment to the next string pointer. Using the `--` or `++ operators` before the variable name processes the variable before using it. So the routine to process and consume the input parameters becomes:

```
       while( --argc)
              main_process_flags( *++argv);
```

In the ansi_c compiler the main routine now prints the Case ID, the start time, calls the initialize routine, processes the command line arguments, parses an input file in the main control loop and exits the software.

Look at the main.c file and find the two calls to `yyparse()`. The original call in the body of the main routine is still present but is conditioned by a flag. The flag is defined as `IS_FLAGS_PARSE` and indicates if `yyparse();` has already been called from processing the command line arguments. For each flag there are three macos `SET_FLAGS_XXXX`, `CLR_FLAGS_XXXX` and `IS_FLAGS_XXXX`. This is the normal way to define a state machine (one-hot) in software. Use `SET_FLAGS_XXXX` to set the flag, `CLR_FLAGS_XXXX` to clear the flag and `IS_FLAGS_XXXX` to check the flag. The flags represent states for the compiler and are declared in the `DATA` memory structure. In a future assignment you may want to add new flags so add a `#define` to define a new bit (not used by another flag), copy the three macro functions and edit to replace the flag names.

Look at the scan.l file and find the major sections. Before the first %{ the lex macros for character definitions are normally defined. The next section is between the %{ and %} lines. This is where C pre-code is normally located for include files, global variables and C support functions. The next section is located between the %% and %% lines and has the lex regular expressions and actions. The scanner actions are C code statements enclosed between curly brackets.

For example, the action function `count()` keeps track of the current character location (`data.column`) of the line. The attribute variable (`YYSTYPE yylval`) is set to the current attribute value and the TOKEN is returned. The third section follows the lex expression section (%%) to the end of the file. The last section is where C post-code for the lex actions is normally located.

Look in the gram.y file and find the major sections (`%{` to `%}`, `%}` to `%%`, `%%` to `%%`, `%%` to `EOF`). In this case the first and last sections are the same as in the lex file and normally contains the pre and post C source code in support of the compiler. The second section defines the TOKENS (`%token`), the start symbol (`%start`) and sets precedence (`%left, %right`). The section between the %%s are where the yacc productions are coded. The parser actions are C code statements enclosed between curly brackets. The last section is where the yyerror() function is defined to print the syntax error message.

Part 4: Laboratory Test and Modify
Edit main.c and find the line that prints caseid and the current time. Change caseid to your Case ID in all lower case. Save the file. Clean the directory and build the ansi_c compiler using the following commands ***make clean*** *and* ***make.***

Test the ANSI C compiler by using the edited example programs from assignment 2. Start with the first example file.
./ansi_c +echo < ../hw02/Code_1_6_1.c

Notice the first example stops echo of the file output with a syntax error message.

```
for caseid start time: Sun Sep  8 17:43:21 2013
/*****************************************************************************
*
*
* FILE:             Code_1_6_1.c
*
* DESC:             EECS 337 Homework Assignment 1
*
* AUTHOR:     caseid
*
* DATE:             August 27, 2013
*
* EDIT HISTORY:
*
*****************************************************************************/
/*
 *     main program
 */
int    main( int argc, char *argv[])
{
/
^
syntax error
Error: yyparse 1
```

This syntax error is due to the comments defined using the // operator. This type comment was not defined in the original ANSI C compiler. Edit the scan.l file and add a pattern and action to the scanner to remove this type of comment. Add a line just below the `"/*" { comment(); }` line.

```
"//"                { comment2(); }
```

And below the comment() function add a function to read slash-slash C comments to EOL.

```
/*
 *     c slash-slash comment function
 */
void   comment2( void)
{
       char c;
       if( IS_FLAGS_ECHO( data.flags))
           ECHO;
       while(( c = input()) != '\n' && c != 0)
             if( c && IS_FLAGS_ECHO( data.flags))
               putchar(c);
       if( c != 0)
             if( IS_FLAGS_ECHO( data.flags))
               putchar(c);
}
```

Edit the yystype.h file and add an `extern void comment2( void);` line below the comment() line. Build the compiler again (make clean, make) and test using the following commands.
./ansi_c < ../hw02/Code_1_6_1.c
./ansi_c < ../hw02/Code_1_6_2.c
./ansi_c +echo < ../hw02/Code_1_6_4.cpp

Notice the last example stops echo of the source output with another syntax error message:

```
 for caseid start time: Sun Sep  8 17:59:45 2013
 1
 ^
syntax error
Error: yyparse 1
```

This is due to the preprocessor # operations in the C program which are created by the C precompiler (cpp). The #define macro is changed to substitute the final expression but the C preprocessor also adds # characters into the source code which are not handled by the scanner. Add a pattern and action to the scanner to allow # to end of line (EOL) to be treated the same as the slash-slash (//) comment. Add a line just below the `"//" { comment2(); }` line in scan.l

```
"#"                 { comment2(); }
```

Build the compiler again (make clean, make) and run the test command:
./ansi_c +echo < ../hw02/Code_1_6_4.cpp &> Code_1_6_4_cpp.txt

The file should now correctly echo to the screen, without syntax errors, showing the #define macro in the final expression format and the # lines skipped over as comments. Notice this output is not the same as the output generated without the C preprocessor and no syntax error is reported. This means the #define macro is missing and the error is not detected.
./ansi_c +echo < ../hw02/Code_1_6_4.c &> Code_1_6_4_c.txt
diff Code_1_6_4_cpp.txt Code_1_6_4_c.txt

Part 4 A: Laboratory Print Token Function
Edit scan.l and after the first include line add the tokens2.c file.

```
#include      "tokens2.c"
```

Then decide how you are going to add the print_token function to each lex expression. You want to pass the TOKEN and the yytext strings as the parameters and condition the call based on the +debug flag. For example you could pass the token as a parameter with the count function and add the call to print_token from inside the count function. Change the count function argument.

```
       void count( int token)

       if( IS_FLAGS_DEBUG( data.flags))
           print_token( token, yytext);
```

Change **all** the lex expressions to pass the token into count, including the white spaces.

```
"auto"              { count(AUTO); return(AUTO); }
"break"             { count(BREAK); return(BREAK); }
… // all of them!
[ \t\v\n\f]         { count( yytext[0]); /* skip white space characters */ }
```

Then change the extern in the yystype.h file to:
`extern void count( int token);.`

Build the compiler again (make clean, make) and run the test commands below. Your output should match the output from assignment 02 as shown below.

./ansi_c +debug < ../hw02/Code_1_6_4.cpp &> Code_1_6_4.txt
diff Code_1_6_4.txt ../hw02/hw02_test.txt

< for caseid start time: Sun Sep 8 18:54:17 2013

> rm -f *.o lex.c yacc.c y.output
> gcc -g   -c -o Echo2.o Echo2.c
> gcc Echo2.o -lm -g -o Echo2
> for caseid start time: Wed Sep 4 06:46:15 2013

Part 5: Laboratory Output Generation
When all your lab assignments have been completed execute the homework script file "./hw03_test.sh". You may need to change the mode of the file to an executable using the command "chmod 755 *.sh". The script file shall remove the old object files, make all source programs and run the tests. To redirect the standard error (stderr) and standard output (stdout) to a file use the following command. Check the output with the second command.
./hw03_test.sh &> hw03_test.txt
diff hw03_test.txt ../hw02/hw02_test.txt

Print out your hw03_test.txt file. Put your name, assignment number, date on it and turn it in with your homework exercises and the laboratory questions at the end of this assignment.

Your final directory should look like (using your CaseID):
EECS337/caseid/hw03/Makefile
EECS337/caseid/hw03/ansi_c
EECS337/caseid/hw03/gram.y
EECS337/caseid/hw03/hw03_caseid.tar
EECS337/caseid/hw03/hw03_test.sh
EECS337/caseid/hw03/hw03_test.txt
EECS337/caseid/hw03/main.c
EECS337/caseid/hw03/original/c-grammar
EECS337/caseid/hw03/scan.l
EECS337/caseid/hw03/tokens2.c
EECS337/caseid/hw03/yystype.h

Part 5 A: Extra Credit (5 points)
Edit scan.l and rewrite the comment function to remove the GOTO statement. Use what you learned about writing a state machine from the previous assignments. Use the input, putchar and unput functions in your design. Your program shall work the same for the +echo flag as the current comment function. When done, upload **ONLY** your scan.l file to blackboard in the assignment area. Submit it with the comments **hw03** with your **CaseID** on the title line. The extra credit will be graded on-line and your score will be added to your assignment score.

Part 5: Laboratory Questions

1) What is the definition for ECHO? (hint: lex scan.l and edit the lex.yy.c file to find the macro.)

2) What is the type of yylval?

3) What is the type of yytext?

4) What is the command line to create the y.tab.h file?

5) In what file is the y.tab.h file included?

6) Why are there two calls to yyparse() in main.c?

7) What is the definition of unput(c) found in the scan.l file?

8) Why is the token passed as a parameter to the count function?

9) What programming method should be avoided, which is found inside the comment function?

10) What is the name of the lex variable for reading input files (fopen) instead of using the redirect operation?