Programming Assignment 2
(Based on Material By: R. Heckendorn)

Due: Sep. 24 @ 11:59pm

The Problem
In this programming assignment, you will work on the following tasks:

- Tweak-in some improvements to the interface.
- Use Flex and Bison to build a *parser* for the C- language.
- While doing the parsing, you will construct the syntax tree. This is not the annotated syntax tree but just the initial abstract syntax tree.
- You will write the procedure **`printTree`** which will print out the tree exactly as I have in the examples I give below. Almost the same format is not good enough. The more we have to work to show your output is the same the fewer points you get. Your main program will contain:

```
yyparse();          // parse and build the tree in the global var
syntaxTree.
printTree(stdout, syntaxTree);
```

The first line calls the parser which will store the tree in the global called **`syntaxTree`**.

The detailed description of your tasks in this assignments is as follows:

Improving the interface
When done with this assignment, you will have created code that will recognize legal C- programs and generate the first pass at the tree.

The parser will be named c- just like last time. It will read and process a stream of tokens from a filename given as the first argument to the c- command OR from standard input if the filename argument is not present. Just like before.

It will now also take the **`-d`** option as an argument. The **`-d`** option turns on the **`yydebug`** flag by setting it to 1.
For example: **`c- -d sort.c-`** should run the c- compiler on the program sort.c- and give details of the parsing that is going on. While **`c- sort.c-`** should simply run the c- compiler. In both cases the abstract syntax tree will be printed. I recommend using the **`getopt`** routine since this will handle UNIX arguments in a uniform and standard way. Be sure to include the **`getopt`** code in your tar.

The Parser

For the parsing part of the assignment, replace your Bison grammar to parse C- code. A good approach is to initially forget about the syntax tree part of the assignment. If you get the right grammar into your compiler it will successfully parse any C- program. A program that simply recognizes whether a program is legal or not is called a recognizer. When you build your bison grammar directly from the grammar supplied you will find that you have the dangling else problem. There are several ways to fix this problem. We have discussed similar situations in class.

Coding restriction:

Do not attempt to fix dangling else or any other grammar issue with associativity declarations such as %left, %right, or %assoc. Do not fix any other problem with your grammar by using the %expect feature of Bison. This causes Bison to ignore some number of parsing errors. Really, you now have the knowledge to do this without this "feature".

IMPORTANT: I expect your parser to compile without any parser errors.
Now that your recognizer is working. Let's look at the syntax tree I want you to produce. The tree is an abbreviated portion of the parse tree containing the parts we are interested in.

Here is a sample **`TreeNode`**. You can use what you want.

```
typedef struct treeNode
{
    // connectivity in the tree
    struct treeNode *child[MAXCHILDREN];   // children of the node
    struct treeNode *sibling;              // siblings for the node

    // what kind of node
    int lineno;                         // linenum relevant to this node
    NodeKind nodekind;                  // type of node
    union                               // subtype of type
    {
     DeclKind decl;                        // used when DeclK
     StmtKind stmt;                        // used when StmtK
     ExpKind exp;                          // used when ExpK
    } kind;

    // extra properties about the node depending on type of the node
    union                               // relevant data to type -> attr
    {
     OpKind op;                    // type of token (same as in bison)
     int value;                // used when an integer constant or boolean
     unsigned char cvalue                 // used when a character
     char *string;                         // used when a string constant
     char *name;                           // used when IdK
    } attr;
```

```
    ExpType expType;          // used when ExpK for type checking
    bool isArray;             // is this an array
    bool isRecord;            // is staticly allocated?
    bool isStatic;            // is staticly allocated?

    // even more semantic stuff will go here in later assignments.
} TreeNode;
```

Feel free to use the modern **struct** rather than using **typedefs**. You can actually use any data structure you like as long as the output matches mine and it builds a tree of the same shape as mine.

To encode the program as a tree you need to make the right nodes at the right steps in the parsing. When you need to make a node you will use routines you write similar to the **newStmtNode** function in **util.c** for the Tiny language (see code on Canvas). These nodes will be passed up the tree as pointers and then assembled. This means that many of the nonterminals will be of type pointer to a node.

Coding restriction: Do not use YYSTYPE.

The Parser and printTree

So what should the tree look like for a given program? This is essentially described in the Bison code.

To understand the examples you must understand the output format of the printTree function. The printTree function prints the important information contained in the node pointed to by the second argument. It then applies the printTree function to all the nonnull children and prints them out numbered starting with child[0] being numbered "Child 0" etc. It then recursively follows the sibling pointer if it is nonnull and applies the printTree function to that. The first sibling found is numbered 0. Reading the syntax tree printed for sample input programs shows you what to do in each case. For example, given this program:

```
bool a;
int b[100];

int max(int x, y)
{
    int z, zz;
    char c;

    if (x>y) z=x;
        else z=y;

    while (666>665) break;
    ;
    b[41] = max(42, 43)+44*55;

    return;
}
```

you should get the following output from your c-.

```
Var a of type bool [line: 1]
Sibling: 0  Var b is array of type int [line: 2]
Sibling: 1  Func max returns type int [line: 4]
!   Child: 0  Param x of type int [line: 4]
!   Sibling: 0  Param y of type int [line: 4]
!   Child: 1  Compound [line: 5]
!   !   Child: 0  Var z of type int [line: 6]
!   !   Sibling: 0  Var zz of type int [line: 6]
!   !   Sibling: 1  Var c of type char [line: 7]
!   !   Child: 1  If [line: 9]
!   !   !   Child: 0  Op: > [line: 9]
!   !   !   !   Child: 0  Id: x [line: 9]
!   !   !   !   Child: 1  Id: y [line: 9]
!   !   !   Child: 1  Assign: = [line: 9]
!   !   !   !   Child: 0  Id: z [line: 9]
!   !   !   !   Child: 1  Id: x [line: 9]
!   !   !   Child: 2  Assign: = [line: 10]
!   !   !   !   Child: 0  Id: z [line: 10]
!   !   !   !   Child: 1  Id: y [line: 10]
!   !   Sibling: 0  While [line: 12]
!   !   !   Child: 0  Op: > [line: 12]
!   !   !   !   Child: 0  Const: 666 [line: 12]
!   !   !   !   Child: 1  Const: 665 [line: 12]
!   !   !   Child: 1  Break [line: 12]
!   !   Sibling: 1  Assign: = [line: 14]
!   !   !   Child: 0  Op: [ [line: 14]
!   !   !   !   Child: 0  Id: b [line: 14]
!   !   !   !   Child: 1  Const: 41 [line: 14]
!   !   !   Child: 1  Op: + [line: 14]
!   !   !   !   Child: 0  Call: max [line: 14]
!   !   !   !   !   Child: 0  Const: 42 [line: 14]
!   !   !   !   !   Sibling: 0  Const: 43 [line: 14]
!   !   !   !   Child: 1  Op: * [line: 14]
!   !   !   !   !   Child: 0  Const: 44 [line: 14]
!   !   !   !   !   Child: 1  Const: 55 [line: 14]
!   !   Sibling: 2  Return [line: 16]
Number of warnings: 0
Number of errors: 0
```

In the cases where there is an optional expression or statement the corresponding child pointer is set to NULL (i.e. 0). For example compound statements might not have any declarations so child[0] pointer would be set to NULL. Return optionally takes an expression. If there isn't an expression then the Child[0] pointer is NULL. The while statement might not have a body: for example while (searching()); in which case child[1] is NULL. The default for unneeded children and siblings is always the NULL pointer. This means your code has to check for NULL pointers!

HINT: The code for the Tiny compiler (on Canvas) is a good model for how to create nodes and print a tree.

Sizes of arrays, locations in memory, etc will all be handled in the next assignment.
Be sure to initialize a variable for counting the number of errors and warnings and print out the results at the end. I will only give you legal C- code in this assignment so both will always be zero, but we are getting ready for the much harder next assignment.

Examples
A great example file is the **everything06.c-** file which produces the everything06.out file posted on Canvas when the command:

```
c- everything06.c- > everything06.out or c- < everything06.c- > everything06.out
```

is used.

Submission
You will submit a single uncompressed tar file through Canvas. You can submit as many times as you like. The LAST file you submit BEFORE the deadline will be the one graded.
We will use extensive tests, so thoroughly test your program with inputs of your own.
If you have tests you really think are important or just cool please send them to me and I will consider adding them to the test suite.