Computer Science 426 --- Fall 2014
Project 5 --- Symbol Table Management

1. Details and What to Turn In: This project is due at 4pm on Wednesday November 12th. You should make a copy of your stage4 folder and call it stage5<lastname>. Do your work for this project in the stage5 folder.

Video Resource: There is a video in the DyKnow class named "CompilersHomework" that will be helpful to watch and listen to. The video makes some suggestions about dealing with parameters in the symbol table. The video is called "InsertingParametersWithAudio."

2. The Project: Implement a symbol table class for your compiler and incorporate its routines into your recursive descent and operator precedence parsers. The full symbol table should be implemented as a stack of (possibly self organizing) lists --- one for each scope level. Each of the lists should be comprised of:

i. The name of the scoping level (for example, main.foo.fee).

ii. The nesting level of the table (level zero for program body, level one for everything defined within the program body, level two for those things defined inside of level 1 scopes, etc.)

iii. The next offset to be used when a new entry is added to this level of the stack. This offset should start at 0 when the stack level is created. The offset should be incremented by 1 each time a variable identifier **(not a function identifier – trust me for later!)** is added to this level of the table.

iv. A pointer to a self-organizing list of nodes. Each node contains information about a symbol. You may omit the self-organizing feature of the list for a maximum grade of 90% (**you should save this feature till the end -- consider turning in two versions in two directories if you take this feature on).** The information stored for each symbol includes:

a) The lexeme value.

b) The kind (func-id, var-id, ref-param, value-param) – you need new consts for this.

c) The type (func-id-type, int-type, boolean-type) you need new consts for this.

d) The offset of this identifier from the start of the current scope.

e) The nesting level of the table that this entry is contained in – this may seem redundant, but it will be helpful in project 7. This value is the same as (ii) above, it is simply repeated for each node in the list.

f) If (kind == func-id) then store a pointer to a linked list of the formal parameters for the function. Each node in the list should contain items (a) - (e) above. NOTE: You do not need to try to self organize this list, in fact, **you should not do so.**

You will need to implement the following routines:

Constructor --	Initializes the table to be empty.
TableAddLevel --	Add a new level to the table. The new level will contain an empty (self-organizing) list. This routine is used when you enter the scope of a new function.
TableDelLevel --	Delete the top-most level of the table. **Free all memory.** Used when you leave the scope of a new function. Note… be careful about when you call this method. You will want to call it after the last token in a function but you need to be sure this is the end of a function and not the end of a loop, if statement, or compound statement.
TableAddEntry --	Add an entry to the table (the entry is added to the **front** of the self-organizing list which is the top-most table on the stack of tables**.**) **Report an error if a duplicate identifier is encountered. The error should consist of the message "Duplicate identifier <name of identifier>", and the print the offending line and exit.**
TableLookup --	Given a lexeme as a parameter, determine if it appears in the table (searching multiple levels may be needed.) You may want to write this routine in terms of a helping function that searches an individual list. Tablelookup should return NULL if the specified lexeme is not in the table, otherwise it should **return a pointer** to the node of the list that contains information about this lexeme. If you are writing your list as a self-organizing list, this routine should also move the item to the front of the list.

Add new files to your project named table.h and table.cc and define the symbol table there. Add private data of class tableclass to your scanner class. The scanner class will need member functions which correspond to each of the routines shown above. When the scanner's member function is called it simply calls the corresponding symbol table routine. Thus we think of the scanner as managing information about the various lexemes in the program. The scanner contains a symbol table to help it do this. The parser gives the scanner instructions about how to manage the symbol table of lexemes based on the syntactic layout of the program.

Incorporate the routines into your parsers (both recursive descent and operator precedence) as appropriate so that the parser can detect three types of errors which it did not before: (1) duplicate identifier, (2) undeclared identifiers, and (3) incorrect number of parameters. When these errors are detected print an appropriate message, print the current line, and exit the program. There are other errors that will eventually be detected with help from your symbol table – but not yet. For example, we are not yet ready to handle type mismatch errors.

Include a compiler option {$s+} which causes the symbol table to be dumped as soon as $s+ is encountered. When this directive is encountered the current symbol table (all levels) should be printed in a format similar to what is displayed below. Display the name of each list printed as well as the nesting level. Also print each identifier in the list along with all of its attributes. Label your output neatly. The order in which items are printed will depend on whether or not your list is self organizing.

For example if the input file is:

```
program demo;
        int abc;
        int cde;

        function fee;
        begin
        end;

        function foo (int &ppp, boolean qqq);
                int  abc;
                boolean  bb5;
        begin
                ppp = 6;
                qqq = true
                {$s+}
        end;

begin
  abc = 7;
  foo (abc, abc)  {note the type error will not be detected yet}
  {$s+}
end.
```

```
**********
Name = main.foo  Nesting level = 1
lexeme=bb5, kind=var-id, type=boolean-type, offset=3, nesting level 1
lexeme=abc, kind=var-id, type=int-type, offset=2, nesting level 1
lexeme=qqq, kind-val-param, type=boolean-type, offset=1, nesting level 1
lexeme=ppp, kind=ref-param, type=int-type, offset=0, nesting level 1

Name = main  Nesting level = 0
lexeme=foo, kind-func-id, type=func-id, offset=2, nesting level 0
        param-list:
                lexeme=ppp, kind=ref-param, type=int-type, offset=0
                lexeme=qqq, kind=val-param, type=boolean-type, offset=1
lexeme=fee, kind=func-id, type=func-id, offset=2, nesting level 0
        param-list:
lexeme=cde, kind=var-id, type-int-type, offset=1, nesting level 0
lexeme=abc, kind=var-id, type=int-type, offset=0, nesting level 0

**********
Name = main  Nesting level = 0
lexeme=foo, kind-func-id, type=func-id, offset=2, nesting level 0
        param-list:
                lexeme=ppp, kind=ref-param, type=int-type, offset=0
                lexeme=qqq, kind=val-param, type=boolean-type, offset=1
lexeme=fee, kind=func-id, type=func-id, offset=2, nesting level 0
        param-list:
lexeme=cde, kind=var-id, type-int-type, offset=1, nesting level 0
lexeme=abc, kind=var-id, type=int-type, offset=0, nesting level 0

**********
```