CS 3500 – Programming Languages & Translators
Homework Assignment #3

- This assignment is **due by 5 p.m. on Wednesday, October 4, 2017**.
- This assignment will be worth **5%** of your course grade.
- You are to work on this assignment **by yourself**.
- You should **take a look at the sample input and output files** posted on the Canvas website **before** you actually submit your assignment for grading. In particular, you should **compare your output with the posted sample output using the *diff* command**, as was recommended for the previous homework assignments; this can be done very easily using the bash script ***tester.sh*** that is posted for HW #2.

Basic Instructions:

For this assignment you are to modify your lexical and syntactical analyzer from HW #2 to make it also do **symbol table management**, which is a prerequisite to doing full-blown **semantic analysis** (which you'll do in HW #4).

As before, your program **must** compile and execute on one of the campus Linux machines (such as rc*nn*ucs213.managed.mst.edu where ***nn*** is **01-08**). If your *flex* file was named **mfpl.l** and your *bison* file was named **mfpl.y**, we should be able to compile and execute them using the following commands (where *inputFileName* is the name of some input file):

```
flex mfpl.l
bison mfpl.y
g++ mfpl.tab.c -o mfpl_parser
mfpl_parser < inputFileName
```

If you want to create an output file (named *outputFileName*) of the results of running your program on *inputFileName*, you can use:

```
mfpl_parser < inputFileName > outputFileName
```

As in HW #2, **no attempt should be made to recover from errors**; if your program encounters an error, it should simply output a meaningful message (including the line number where the error occurred) and terminate execution. Listed below are the **new errors** that your program also will need to be able to detect for MFPL programs:

Undefined identifier
Multiply defined identifier

Note: Since we will use a script to automate the grading of your programs, you must use these exact error messages!!!

Your program should **still output the tokens and lexemes** that it encounters in the input file, **and the productions being processed** in the derivation[1]. Your program also is still expected to detect and report syntax errors.

In order to check whether your symbol table management is working correctly, your program should **output the message "___Entering new scope" whenever it begins a scope, "___Exiting scope" when it ends a scope, and "___Adding … to symbol table" whenever it makes an entry in the symbol table** (where … is the name of the identifier it is adding to the symbol table); again, you must **use exactly those messages** in order for the grading script to correctly grade your program! Use the *diff* command given in the HW #1 project description to compare your output to the sample output files posted on Canvas.

Below is an example of input with a multiply defined identifier:

```
(let* ( (x 5)
        (y "hello")
        (x 1)
       )
 (print x)
)
```

Given below is the output that should be produced for the example with the multiply defined identifier:

```
TOKEN: LPAREN       LEXEME: (
TOKEN: LETSTAR      LEXEME: let*

___Entering new scope...

TOKEN: LPAREN       LEXEME: (
ID_EXPR_LIST -> epsilon
TOKEN: LPAREN       LEXEME: (
TOKEN: IDENT        LEXEME: x
TOKEN: INTCONST     LEXEME: 5
CONST -> INTCONST
EXPR -> CONST
TOKEN: RPAREN       LEXEME: )
ID_EXPR_LIST -> ID_EXPR_LIST ( IDENT EXPR )
___Adding x to symbol table
TOKEN: LPAREN       LEXEME: (
TOKEN: IDENT        LEXEME: y
TOKEN: STRCONST     LEXEME: "hello"
CONST -> STRCONST
```

[1] The HW #1 and HW #2 output requirements for using **particular TOKEN and NONTERMINAL names, etc.** are still in effect for this assignment.

EXPR -> CONST
TOKEN: RPAREN LEXEME:)
ID_EXPR_LIST -> ID_EXPR_LIST (IDENT EXPR)
___Adding y to symbol table
TOKEN: LPAREN LEXEME: (
TOKEN: IDENT LEXEME: x
TOKEN: INTCONST LEXEME: 1
CONST -> INTCONST
EXPR -> CONST
TOKEN: RPAREN LEXEME:)
ID_EXPR_LIST -> ID_EXPR_LIST (IDENT EXPR)
___Adding x to symbol table
Line 3: Multiply defined identifier

As before, your program should process a single expression from the input file, but remember that a **single expression** could be a **list of expressions (i.e., an expression list)**, terminating when it completes processing the expression or encounters an error.

Note that your program should **NOT** evaluate any statements in the input program (that will be done later in HW #5), or check for things like too many or too few parameters in a function call (which we'll do next in HW #4).

Symbol Table Management:

Posted on the Canvas website are files for **SYMBOL_TABLE and SYMBOL_TABLE_ENTRY C++ classes;** you are welcome to use these files, or you may create your own. You should only need to **#include "SymbolTable.h"** in your **mfpl.y** file to be able to reference these classes.

SYMBOL_TABLE contains one member variable: a hash table (implemented as an STL *map*) of **SYMBOL_TABLE_ENTRY**s; the hash key is a string (i.e., an identifier's name). Defined are class methods for finding an entry in the hash table based on the (string) key, and for adding a **SYMBOL_TABLE_ENTRY** into the hash table.

SYMBOL_TABLE_ENTRY contains two member variables: a string variable for an identifier's name, and an integer variable to represent its type. For now, **assume that every identifier is of type UNDEFINED** (where **UNDEFINED** is an integer constant that you need to define in your program); we will determine and record the actual types of identifiers in the next assignment.

In your **mfpl.y** file, you should define a variable that is an STL **stack of symbol tables**; that is:

stack<SYMBOL_TABLE> scopeStack;

Make sure you also **#include <stack>** in your **mfpl.y** file.

A **new scope should begin** (i.e., **open**) whenever a ***let**** or ***lambda*** token is processed in **mfpl.l**; if you wait to open a new scope when you process **LET_EXPR** or **LAMBDA_EXPR** in **mfpl.y**, it will be too late!

In contrast, a **scope should be closed** when the parser actually processes **LET_EXPR** or **LAMBDA_EXPR** as specified in your **mfpl.y** file. Given below are functions you can add to your **mfpl.y** file (and call as appropriate) to begin and end a scope, respectively:

```
void beginScope( )
{
 scopeStack.push(SYMBOL_TABLE( ));
 printf("\n___Entering new scope...\n\n");
 }
void endScope( )
{
 scopeStack.pop( );
 printf("\n___Exiting scope...\n\n");
 }
```

Identifiers will need to be added to a symbol table when processing **ID_EXPR_LIST** for a **LET_EXPR**, and when processing **ID_LIST** for a **LAMBDA_EXPR**. A **"multiply defined identifier" error** should be generated if you are ever adding an identifier to a symbol table and it is already there.

When you process the **EXPR → IDENT** production in **mfpl.y**, you will need to look up the identifier in the symbol tables that are open at that time, starting with the most recently created symbol table and working backwards from there. The following function can be added to your **mfpl.y** file and called from the code for the **EXPR → IDENT** production to do this:

```
bool findEntryInAnyScope(const string theName)
{
 if (scopeStack.empty( )) return(false);
 bool found = scopeStack.top( ).findEntry(theName);
 if (found)
  return(true);
 else {   // check in "next higher" scope
          SYMBOL_TABLE symbolTable = scopeStack.top( );
          scopeStack.pop( );
          found = findEntryInAnyScope(theName);
          scopeStack.push(symbolTable);   // restore the stack
          return(found);
 }
}
```

If the entry you are looking for cannot be found in any valid scope, an **"undefined identifier" error** should be generated.

How *bison* Will Know an Identifier's Name:

As discussed above, there are places in your *bison* code where you will need to add an identifier to the symbol table. That name is automatically put in the predefined variable *yytext* when you process an **IDENT** token in your *flex* file (i.e., it is the lexeme for the **IDENT** token). In order to communicate that information to the *bison* code, here's what you need to do.

Define the following *union* data structure right after the **%}** in your **mfpl.y** file:

```
%union {
  char* text;
};
```

The *char** type will be used to associate an identifier's name with an identifier token. To do this, you should specify the following in your **mfpl.l** file:

```
{IDENT}      {
             yylval.text = strdup(yytext);
             …
             return T_IDENT;
             }
```

And then in your **mfpl.y** file you'll need to declare that the **T_IDENT** token will be associated with the *char** type using the following line (which goes right after your ***%token*** lines in **mfpl.y**):

```
%type  <text> T_IDENT
```

You'll then be able to reference that information in your *bison* code as shown in the following example:

```
N_EXPR : T_IDENT
         {
           printRule("EXPR", "IDENT");
           bool found = findEntryInAnyScope(string($1));
           …
```

Here **$1** is the *bison* convention for referring to the information that has been associated with the first symbol on the right hand side of the production (which in this case is the **T_IDENT** token).

What to Submit for Grading:

Via Canvas you should submit only your *flex* and *bison* files as well as any .h files necessary for your symbol table, **archived as a *zip* file**. Note that a *make* file will not be accepted (since that is not what the automated grading script is expecting). **Your *bison* file must *#include* your .h files as necessary.** Name your *flex* and *bison* files using **your last name followed by your first initial** with the correct *.l and .y* file extensions (e.g., Homer Simpson would name his files **simpsonh.l** and **simpsonh.y**). Your zip file should be similarly named (e.g., **simpsonh.zip**). You can submit multiple times before the deadline; only your last submission will be graded.

WARNING: If you fail to follow all of the instructions in this assignment, the automated grading script will reject your submission, in which case it will NOT be graded!!!

The grading rubric is given below so that you can see how many points each part of this assignment is worth. Note that the next assignment builds upon this one, so **it is critical that this assignment works properly in all respects!**

Functionality	Points Possible	Mostly or completely incorrect (0% of points possible)	Needs improvement (70% of points possible)	Adequate, but still some errors (80% of points possible)	Mostly or completely correct (100% of points possible)
Data structures for symbol table, symbol table entry, and collection of symbol tables are appropriately implemented.	10				
Program correctly opens symbol table for new scope (*let** and *lambda)* and outputs message.	10				

Program correctly closes symbol table for terminated scope (*let** and *lambda)* and outputs message.	10				
Program correctly makes symbol table entries for *let** identifiers and outputs message.	15				
Program correctly makes symbol table entries for *lambda* identifiers and outputs message.	15				
Multiply defined identifier correctly identified and error message printed.	15				
Undeclared identifier correctly identified and error message printed.	15				
Error messages correctly include line number where error occurred.	5				
Program properly terminates upon end-of-file or detection of error.	5				