CS 256 – Programming Languages and Translators
Assignment 3

- This assignment is due by 1 p.m. on Friday, February 28, 2014
- This assignment will be worth 5% of your grade
- You are to work on this assignment by yourself

Basic Instructions

For this assignment you are to modify your lexical and syntactical analyzer from HW2 to make it also do symbol table management, which is a prerequisite to doing full-blown semantic analysis (which you will do in HW4).

As before, your program must compile and execute on one of the campus Linux machines (such as rc*nn*xcs213.managed.mst.edu where *nn* is 01-32). 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 HW2, 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. 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!

Sample Input with Multiply Defined Identifier

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

Sample Output for 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
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 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 HW5), or check for things like too many or too few parameters in a function call (which we will do next in HW4).

Symbol Table Management

Posted with this assignment are files for a symbol table and a symbol table entry written in C++; you may use these files or 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 that is a hash table (implemented as an STL map) of `SYMBOL_TABLE_ENTRYs`; the hash key is a string (i.e., an identifiers 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 identifiers 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 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(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 Knows 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 is 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 identifiers 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 will 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 will 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).

Submission

You will submit this assignment using `cssubmit`. Your ***single*** lexer file ***must*** have a `.l` extension, and your single yacc/bison file must have a `.y` extension. If it does not, you will receive a zero for this assignment. From the directory containing the `.l` file, `.y`, and symbol table files, you will run
`cssubmit 256 a 3`
on the cs213 Linux machines. This will collect your submission and submit it to me. You may submit as many times as you desire; only your last submission will be graded (previous submissions are overwritten). **READ** the output of cssubmit; it may have changed since you last used it.

When grading, I will run the following commands:

```
flex *.l
bison *.y
g++ *.tab.c
```

If you cannot generate an executable by following EXACTLY those steps from the directory you are submitting from, your submission will not receive full credit. If you find yourself having example .l and .y files, or some other file with a `.tab.c` extension, remove them from your submission directory.