CS 3500 – Programming Languages & Translators
Homework Assignment #3

- This assignment is **due by 11:59 p.m. on Friday, March 8, 2019**
- This assignment will be worth **6%** of your course grade.
- You may work on this assignment **with at most one other person enrolled in CS 3500 this semester (either section)**.
- 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.

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. If your *flex* file was named **minir.l** and your *bison* file was named **minir.y**, we should be able to compile and execute them using the following commands (where *inputFileName* is the name of some input file):

```
flex minir.l
bison minir.y
g++ minir.tab.c -o minir_parser
minir_parser < inputFileName
```

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

```
minir_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 Mini-R 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.

There are only two times that a new scope begins (and hence a new symbol table should be created) in Mini-R: (1) at the very beginning of the program a "global" symbol table should be created, and (2) every time you process a function definition (i.e., **N_FUNCTION_DEF**) a new symbol table should be created. When the program session is finished, the global symbol table goes away. And whenever you finish processing a function definition, its symbol table goes away.

In this assignment you also are responsible for **making entries in the symbol table for identifiers**. These entries **only have to record the identifier's name**; you do **NOT** have to record any other information such as the identifier's type or its value! Listed below are the places in the grammar where you'll **need to add code to make an entry in a symbol table**:

N_ASSIGNMENT_EXPR → T_IDENT N_INDEX T_ASSIGN N_EXPR
For N_ASSIGNMENT_EXPR, if the T_IDENT does not already exist in the most recently opened symbol table, you need to add it to the most recently opened symbol table. **Do this processing immediately after you process N_INDEX in this production!**

N_FOR_EXPR → T_FOR T_LPAREN T_IDENT T_IN N_EXPR T_RPAREN N_LOOP_EXPR
For N_FOR_EXPR, if the T_IDENT does not already exist in the most recently opened symbol table, you need to add it to the most recently opened symbol table. **Do this processing immediately after you process the T_IDENT in this production!**

N_FUNCTION_DEF → T_FUNCTION (N_PARAM_LIST) N_COMPOUND_EXPR

N_PARAM_LIST → N_PARAMS | N_NO_PARAMS

N_PARAMS → T_IDENT | T_IDENT , N_PARAMS

For N_FUNCTION_DEF, you need to make an entry in the function's symbol table for each T_IDENT in the N_PARAM_LIST. (Note: This should be the most recently created

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

symbol table because **you need to create a symbol table for the function immediately after you process the T_FUNCTION token in N_FUNCTION_DEF!**)

As mentioned on the first page of this assignment, there are **two new errors that you must be able to detect** in this assignment: (1) multiply defined identifier and (2) undefined identifier. Listed below are the places in the grammar where you'll **need to add code to be able to detect these errors**:

N_FUNCTION_DEF → T_FUNCTION (N_PARAM_LIST) N_COMPOUND_EXPR

N_PARAM_LIST → N_PARAMS | N_NO_PARAMS

N_PARAMS → T_IDENT | T_IDENT , N_PARAMS

For N_FUNCTION_DEF, you need to make an entry in the function's symbol table for each T_IDENT in the N_PARAM_LIST. If a parameter name is used more than once in a parameter list (e.g., *function(**x**, y, **x**)*), it should be flagged as a **multiply defined identifier**.

N_FUNCTION_CALL → T_IDENT T_LPAREN N_ARG_LIST T_RPAREN

In N_FUNCTION_CALL, the T_IDENT must already exist in some open symbol table; otherwise, it should be flagged as an **undefined identifier**.

N_ENTIRE_VAR → T_IDENT

In the context of N_ENTIRE_VAR, the T_IDENT must already exist in some open symbol table; otherwise, it should be flagged as an **undefined identifier**.

N_SINGLE_ELEMENT → T_IDENT T_LBRACKET T_LBRACKET N_EXPR
T_RBRACKET T_RBRACKET

In the context of N_SINGLE_ELEMENT, the T_IDENT must already exist in some open symbol table; otherwise, it should be flagged as an **undefined identifier**.

As before, your program should process expressions from an input file until it processes the expression *quit()* or it encounters an error or it reaches the end of input.

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 **.y** file to be able to reference both of 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 we *#define*); we will determine and record the actual types of identifiers in the next assignment.

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

stack<SYMBOL_TABLE> scopeStack;

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

A **new scope should begin** (i.e., **open**) at the beginning of *main()* for the global scope and right after you process **T_FUNCTION** in **N_FUNCTION_DEF**.

In contrast, a **scope should be closed** at the end of *main()* and at the end of the processing for **N_FUNCTION_DEF**. Given below are functions you can add to your **.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");
 }
```

As explained previously, identifiers will need to be added to a symbol table when processing **N_ASSIGNMENT_EXPR, N_FOR_EXPR,** and **N_PARAMS**. For **N_PARAMS**, a

"multiply defined identifier" error should be generated if you are trying to add an identifier to the current symbol table and it is already there.

When you process **N_FUNCTION_CALL, N_ENTIRE_VAR**, and **N_SINGLE_ELEMENT**, you will need to look up a **T_IDENT** 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 **.y** file and called from the code for those productions 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 open 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 **.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 must specify the following in your **.l** file:

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

Then in your **.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 your **.y** file):

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

You'll then be able to reference that information in your *bison* code anyplace **T_IDENT** is used in a production, as shown in the following example:

```
N_ENTIRE_VAR : T_IDENT
             {
               printRule("ENTIRE_VAR", "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**). If you work with another person, name your files using the combination of your last name and your programming partner's last name (e.g., if Bugs Bunny worked with Daffy Duck, they would make a **SINGLE** submission in Canvas under **one or the other's username (NOT BOTH!!!)**, naming their files bunnyduck.l, bunnyduck.y, bunnyduck.zip, etc.; be consistent in your naming scheme – do **NOT** name one file bunnyduck and the other file duckbunny!). 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!**

Test File	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)
assignmentUndef	5				
compoundExprErrors	8				
compoundExprNoErrors	6				
functionDefErrors	9				
functionDefNoErrors	7				
functionUndef	9				
listErrors	8				
listNoErrors	6				
multipleDef	12				
multipleFunctions	18				
undef	12				
	100				