CS 3500 – Programming Languages & Translators Homework Assignment #5

- This assignment is **due by 11:59 p.m. on Tuesday, April 16, 2019**.
- This assignment will be worth **10%** of your course grade.
- You may work on this assignment **with at most one other person enrolled in CS 3500 this semester (either section)**.
- Before you submit your assignment for grading, you should test it on the sample input files posted on the Canvas website. A new **bash script** has been posted that can be used to run all of the sample input files for this assignment **EXCEPT for the files with "read" in the file name (which require keyboard input)** so that you can see which ones produce output different from what we are expecting.

Basic Instructions

For this assignment you are to finish developing the Mini-R interpreter by modifying your HW #4b program to make it also **evaluate** expressions.

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_eval
minir_eval  inputFileName
```

Note that **we are no longer redirecting input using < in the command line**. This is necessary in order to be able to get input from the keyboard for the Mini-R *read()* statement at the time that it actually is evaluated. Consequently, you need to change your *main( )* function to get the input file from the command line:

```
int main(int argc, char** argv)
{
 if (argc < 2)
 {
  printf("You must specify a file in the command line!\n");
  exit(1);
 }
 yyin = fopen(argv[1], "r");
 do
 {
  yyparse();
```

```
   } while (!feof(yyin));
   return 0;
}
```

As in HW #4b, **no attempt should be made to recover from errors**; if your program encounters an error, then it should simply output a meaningful message (that includes the line number) and terminate execution. There are **2 new errors** that you need to detect for this assignment:

Subscript out of bounds

Attempted division by zero

Your program should <u>NOT</u> **output the tokens and lexemes, or symbol table management messages.**

As before, your program should process input until it processes *quit()* or encounters an error or detects end of input. After processing an expression from the input file (starting from the start nonterminal of the grammar), your program should evaluate and **output the resulting value of the expression**; this is known as the *read-eval-print* process of interpreters. Specifically, the N_START rule in your *bison* file should include the following output statements:

```
printRule("START", "EXPR");
printf("\n---- Completed parsing ----\n");
printf("\nValue of the expression is: ");
```

Note that because your program is now **evaluating** expressions in the input program, you also will need to **record an identifier's <u>value</u> in the symbol table**, which could be a string, integer, float, Boolean, list, or a special value representing "null." Lists can contain any combination of strings, integers, floats, or Booleans; they cannot contain other lists.

<u>Programming Language (Dynamic) Semantics</u>

What follows is a brief description about the evaluation rules for the various expressions in Mini-R.

N_EXPR → N_IF_EXPR | N_WHILE_EXPR | N_FOR_EXPR |
N_COMPOUND_EXPR | N_ARITHLOGIC_EXPR |
N_ASSIGNMENT_EXPR | N_OUTPUT_EXPR | N_INPUT_EXPR |
N_LIST_EXPR |
N_FUNCTION_DEF | N_FUNCTION_CALL |
N_QUIT_EXPR

The resulting value of an **N_EXPR** is the resulting value of the nonterminal on the right-hand side of the production that is applied. For example, if **N_EXPR → N_IF_EXPR**, then the value of **N_EXPR** is **N_IF_EXPR**'s value.

N_CONST → T_INTCONST | T_STRCONST | T_FLOATCONST | T_TRUE | T_FALSE
The resulting value of an **N_CONST** is the value of an integer constant if the **T_INTCONST** rule is applied, the value is a string constant if the **T_STRCONST** rule is applied, or a Boolean value if the **T_TRUE** or **T_FALSE** rules are applied. You'll have to associate the value of the lexeme with the token in your .l file (similar to what you did for a **T_IDENT** token). You might find the C functions *atoi* and *atof* helpful for taking the char* lexeme (i.e., *yytext*) and changing it to an integer or a float; see the internet for documentation on those functions.

N_COMPOUND_EXPR → { N_EXPR N_EXPR_LIST }
N_EXPR_LIST → ; N_EXPR N_EXPR_LIST | ε

The resulting value of **N_COMPOUND_EXPR** is the resulting value of **N_EXPR** if **N_EXPR_LIST** was just the epsilon production; otherwise; it's the value of **N_EXPR_LIST**. The same goes for setting the value of **N_EXPR_LIST** based on its productions.

N_IF_EXPR → N_COND_IF) N_THEN_EXPR | N_COND_IF) N_THEN_EXPR T_ELSE N_EXPR
N_COND_IF → T_IF (N_EXPR
N_THEN_EXPR → N_EXPR

If the very first **N_EXPR** of an *if-expression* (which is in **N_COND_IF**) evaluates to a non-zero value, then the resulting value of the **N_IF_EXPR** is the value of the "then" expression; otherwise, the resulting value of the **N_IF_EXPR** is the value of the "else" expression. If there is no "else" expression and the very first **N_EXPR** evaluates to a zero value, then the value of the **N_IF_EXPR** is **NULL**.

Note that we can now definitively determine the type of the **N_IF_EXPR** (i.e., there no longer should be designation of "combo" types like **INT_OR_STR**, **INT_OR_STR_OR_BOOL**, etc.); therefore, you are expected to assign the type of the **N_IF_EXPR** accordingly.

N_QUIT_EXPR → T_QUIT( )

The resulting value of **N_QUIT_EXPR** is your special value that represents "null."

N_LIST_EXPR → T_LIST (N_CONST_LIST)
N_CONST_LIST → N_CONST , N_CONST_LIST | N_CONST

The resulting value of **N_LIST_EXPR** is a list made up of the constants (values) in **N_CONST_LIST**.

N_ASSIGNMENT_EXPR → T_IDENT N_INDEX = N_EXPR

The resulting value of **N_ASSIGNMENT_EXPR** is the value of **N_EXPR**. Note that Mini-R lists use **1-based indexing**.

N_OUTPUT_EXPR → T_PRINT (N_EXPR) | T_CAT (N_EXPR)

When an **N_OUTPUT_EXPR** is processed, it should **output the value** of the **N_EXPR** **followed by a newline** to standard output. The resulting value of a **N_OUTPUT_EXPR** is the value of the **N_EXPR** if the **T_PRINT** rule is applied; otherwise, the resulting value is **NULL**. Note: When you output a list, output each item of the list separated by one space, all on one line and between parentheses (e.g., (1 2 3)).

N_INPUT_EXPR → T_READ ( )

When an **N_INPUT_EXPR** is processed, it should perform a C or C++ *getline* call into a string (or char array); do not use *cin* >> since **valid input can contain spaces**. If the first character that is input is **NOT** a digit or a '+' or a '–', assume the entire input is a string (**STR**). Otherwise, if the input contains a period, assume the input is a float (**FLOAT**); otherwise, assume it is an integer (**INT**).

Note that you should now set the type of an **N_INPUT_EXPR** to either **INT** or **STR** or **FLOAT** (it's no longer the "combo" type **INT_or_STR_or_FLOAT**), and **dynamically type-check its use in other expressions**.

N_VAR → N_ENTIRE_VAR | N_SINGLE_ELEMENT

N_SINGLE_ELEMENT → T_IDENT [[N_EXPR]]

N_ENTIRE_VAR → T_IDENT

In general, an **N_VAR**'s value is determined by looking up the **T_IDENT** in the symbol table(s); you should be recording its value there. If it is an indexed variable (i.e., **N_SINGLE_ELEMENT**), its value in the symbol table should be some kind of "list" data structure, and you will have to evaluate the **N_EXPR** in **N_SINGLE_ELEMENT** to determine which value in the "list" to return as the value of this **N_VAR**.

N_ARITHLOGIC_EXPR → N_SIMPLE_ARITHLOGIC |
N_SIMPLE_ARITHLOGIC N_REL_OP N_SIMPLE_ARITHLOGIC

…and its friends **N_SIMPLE_ARITHLOGIC, N_ADD_OP_LIST, N_TERM, N_MULT_OP_LIST, N_FACTOR**

The values for these nonterminals will be determined by the values of the operands, when there are operators, performing the specified operations on those operands. For arithmetic operators, perform the calculations as they would be done in C++. For example, division between two integers produces an integer result with no rounding, whereas division between an integer and a float produces a float result. Arithmetic with a Boolean operand is treated like an integer.

N_WHILE_EXPR → T_WHILE (N_EXPR) N_EXPR
N_FOR_EXPR → T_FOR (T_IDENT T_IN N_EXPR) N_EXPR
N_FUNCTION_CALL → T_IDENT (N_ARG_LIST)
N_FUNCTION_DEF → T_FUNCTION (N_PARAM_LIST) N_COMPOUND_EXPR
For this assignment, your program will <u>**NOT**</u> have to handle loops or functions; they will not appear in any of the test files. Evaluation of these constructs can be done for extra credit; see the end of this homework description for more information.

<u>What to Submit for Grading</u>

Via Canvas you should submit <u>only</u> 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 <u>not</u> 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 someone, name your files using the combination of your last names; only submit make <u>one</u> submission using either person's login. 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 for submitting this assignment, the automated grading script will reject your submission, in which case it will <u>NOT</u> be graded!!! Submissions that do not flex, bison, and/or compile will receive a grade of zero!

The grading rubric is posted on Canvas so that you can see how many points each part of this assignment is worth (broken down by what is being tested for in each sample input file).

<u>Helpful Hints</u>

Following are some helpful hints for doing this assignment:

- C++ has modulus functions both for integer operands (i.e., %) and float operands; for float operands, see ***fmod***.
- C++ has a function for doing exponentiation; see ***pow***.
- Whenever you output a float value, format it to use only **2 decimal places** (e.g., 3.125 should be output as 3.13 with rounding).
- You are likely to get into **BIG TROUBLE if you use *char**** to store string values! Instead, use *string* (outside of the *%union*) or a null-terminated character array. Note that you could *typedef CString char[256];* and then use the type *CString*. It's OK to assume that strings will not be longer than 255 characters.

- As a follow-up to what is said above, if you use pointers, do NOT set the value of one pointer variable to the value of another pointer variable using =. That's just begging for trouble! And don't do "shallow copying" of pointer-based data structures! If you don't know what that means, look it up on the internet - you should have learned about that in your data structures course ☹
- In general, **expression evaluation should be done as it is done in C++**. For example, an operation between 2 integers produces an integer result, whereas an operation involving a float produces a float result. A Boolean operand can be treated like an integer (TRUE = 1, FALSE = 0); a non-zero value (integer or float) can be treated like TRUE and a zero value can be treated as FALSE.
- There is no STL class that can store values of different types (which you need for the Mini-R list); you will have to define your own data structure. If you statically declare memory, you can assume that **a Mini-R list will not contain more than 255 items**.
- In general, ***%union* types** must be simple C types (e.g., *char*, *int*, etc.); they can't be classes. *%union* types can be *struct*'s and those *struct*'s can have member variables that are *struct*'s or arrays of simple types; but the *struct* member variables cannot be arrays of other *struct*'s. You can, however, use *struct*'s with member variables that are **pointers** to other *struct*'s (which contain arrays of non-simple data types) or even classes. If you specify a *%union* type that disobeys these rules, you might not get a *bison* or *g++* error; you might get a **stack dump** when you execute your file (no matter what input you give it)!

Extra Credit

You can earn extra credit by making your program **evaluate loops (for-loops and while-loops) and functions**. This is not trivial! You basically will have to store in the symbol table a representation of the code (i.e., expression) that is in the body of the loop/function so that you can repeatedly evaluate it. Plus, you will need to write an "evaluation" function to handle every one of those possible expression data structures that you store in the symbol table.

These constructs would be evaluated as follows:

N_WHILE_EXPR → T_WHILE (N_EXPR) N_EXPR
N_FOR_EXPR → T_FOR (T_IDENT T_IN N_EXPR) N_EXPR
The value of an **N_WHILE_EXPR** or an **N_FOR_EXPR** is NULL.

N_FUNCTION_CALL → T_IDENT (N_ARG_LIST)
N_FUNCTION_DEF → T_FUNCTION (N_PARAM_LIST) N_COMPOUND_EXPR
The value of an **N_FUNCTION_CALL** is the value of the **N_COMPOUND_EXPR** of its **N_FUNCTION_DEF** using the values of parameters supplied in the function call.

If you do this extra credit, **YOU are responsible for creating a set of test files** that demonstrate that your program works (i.e., similar to the sample input files given to you for each homework assignment); otherwise, your extra credit program will **NOT** be graded (**WE are NOT going to spend the time making up all of those test files!**).

If you correctly implement all facets of loops and functions (i.e., they work for all possible Mini-R expressions), you will earn **10% extra credit on your course grade**; that's basically an entire letter grade increase! If you only partially implement the extra credit, the amount of credit you earn will depend on how much you have done. If you work with another person, the two of you will split the extra credit (e.g., each person will only get at most 5% extra).