CS 3500 – Programming Languages & Translators
Homework Assignment #5

- This assignment is **due by 11:59 p.m. on Thursday, November 1, 2018**.
- This assignment will be worth **7%** 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 **<u>before</u>** 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.

<u>Basic Instructions</u>

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

As before, your program **<u>must</u>** compile and execute on one of the campus Linux machines. 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_eval
mfpl_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 MFPL *(input)* statement at the time that it actually is evaluated. Consequently, you need to change your *main( )* function to process 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 #4, **no attempt should be made to recover from errors**; if your program encounters a syntactical or semantic error, or **attempted division by zero (a new error you need to catch!),** then it should simply output a meaningful message (that includes the line number) and terminate execution.

Your program should **still output the tokens and lexemes** that it encounters in the input file, and still should **output an appropriate message whenever it begins or ends a scope, or whenever it makes an entry in the symbol table**.

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. After reading in the expression from the input file, 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\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 value in the symbol table**, which could be a string, integer, or Boolean value.

Programming Language (Dynamic) Semantics

What follows is a brief description about the evaluation rules for the various expressions in MFPL (Mini Functional Programming Language).

N_EXPR → N_CONST | T_IDENT |
T_LPAREN N_PARENTHESIZED_EXPR T_RPAREN

The resulting value of an **N_EXPR** is the value of the constant if the **N_CONST** rule is applied, or the value of the identifier if the **T_IDENT** rule is applied (i.e., you'll have to look up its name in the symbol table to find out its value), or the value of the **N_PARENTHESIZED_EXPR** if that rule is applied.

N_CONST → T_INTCONST | T_STRCONST | T_T | T_NIL

The resulting value of an **N_CONST** is the value of an integer constant if the **T_INTCONST** rule is applied, or the value of a string constant if the **T_STRCONST** rule is applied. Be careful what value you choose to internally represent the MFPL constants t and nil; logically, MFPL should treat nil as *false* and **everything** else (including 0) as *true*.

N_PARENTHESIZED_EXPR → N_ARITHLOGIC_EXPR | N_IF_EXPR |
N_LET_EXPR | N_PRINT_EXPR |
N_INPUT_EXPR | N_EXPR_LIST

The resulting value of an **N_PARENTHESIZED_EXPR** is the value returned by whichever rule is applied. Note that **you are no longer required to process N_LAMBDA_EXPR.**

N_ARITHLOGIC_EXPR → N_UN_OP N_EXPR | N_BIN_OP N_EXPR N_EXPR
N_BIN_OP → N_ARITH_OP | N_LOG_OP | N_REL_OP
N_ARITH_OP → T_MULT | T_SUB | T_DIV | T_ADD
N_LOG_OP → T_AND | T_OR
N_REL_OP → T_LT | T_GT | T_LE | T_GE | T_EQ | T_NE
N_UN_OP → T_NOT

The value of the **N_ARITHLOGIC_EXPR** depends on what the operator is; basically, you have to perform the operation on the values of the **N_EXPR**'s and assign the resulting value to **N_ARITHLOGIC_EXPR**. If the operator is division, you must check for **attempted division by zero**. For the logical operators (*and*, *or*, not), you should treat nil as *false* and **everything** else (including 0) as *true*.

N_IF_EXPR → T_IF N_EXPR$_1$ N_EXPR$_2$ N_EXPR$_3$

If **N_EXPR$_1$** evaluates to **nil**, then the resulting value of the **N_IF_EXPR** is the value of **N_EXPR$_3$**; otherwise, the resulting value of the **N_IF_EXPR** is the value of **N_EXPR$_2$**. Note that we can now definitively determine the type of the **N_IF_EXPR** (i.e., there no longer should be designation of types **INT_OR_STR**, **INT_OR_STR_OR_BOOL**, etc.); thus you are expected to assign the type of the **N_IF_EXPR** accordingly.

N_LET_EXPR → T_LETSTAR T_LPAREN N_ID_EXPR_LIST T_RPAREN N_EXPR
N_ID_EXPR_LIST → ε | N_ID_EXPR_LIST T_LPAREN T_IDENT N_EXPR T_RPAREN

The value of each **T_IDENT** in **N_ID_EXPR_LIST** will be the value of its **N_EXPR**. **This will need to be recorded in the symbol table!** Then the resulting value of the **N_LET_EXPR** will be the value of its **N_EXPR**.

N_LAMBDA_EXPR → T_LAMBDA T_LPAREN N_ID_LIST T_RPAREN N_EXPR
N_ID_LIST → ε | N_ID_LIST T_IDENT

For this assignment, your program will NOT have to handle functions.

N_PRINT_EXPR → T_PRINT N_EXPR

When an **N_PRINT_EXPR** is processed, it should **output the value** of the **N_EXPR** followed by a newline to standard output. Also note that the resulting value of a **N_PRINT_EXPR** is the value of the **N_EXPR**.

N_INPUT_EXPR → T_INPUT

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 a digit or a '+' or a '–', treat the entire input as an integer; otherwise, treat the input as a string. Note that you should now set the type of an **N_INPUT_EXPR** to either **INT** or **STR** (it's no longer **INT_OR_STR**), and **dynamically type-check its use in other expressions**.

N_EXPR_LIST → N_EXPR N_EXPR_LIST | N_EXPR

The resulting value of an **N_EXPR_LIST** is the value of the last **N_EXPR**.

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 someone, name your files using the combination of your last names; only submit make one 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 NOT be graded!!!

The grading rubric is given below 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).

Filename	Functionality	Points Possible	Mostly or completely incorrect (0% of points possible)	Needs improvement (50% of points possible)	Mostly or completely correct (100% of points possible)
addGood	Arithmetic expression	2			
arithBad	Arithmetic expression with type error	2			
arithGood	Arithmetic expression	2			
bigLet	Let*, if, arithmetic expressions	10			
divBad	Arithmetic expression with division by zero	5			
divGood	Arithmetic expression	2			
ifGood1	If expression	1			
ifGood2	If expression	1			
ifGood3	If expression	1			
ifGood4	If expression	1			
intconst	Simple constant	1			
letPrint	Let*, arithmetic, and print expressions	5			
letPrintIf	Let*, if, and print expressions	5			
logop_bool_int	Logical expression	2			
lopop_bool_str	Logical expression	2			
lopop_int_bool	Logical expression	2			
lopop_int_int	Logical expression	2			
lopop_int_str	Logical expression	2			
lopop_str_bool	Logical expression	2			
logop_str_str	Logical expression	2			
nestedLet	Let* (deeply nested), arithmetic expressions	7			
nil	Simple constant	1			
not_int	Logical expression	2			

not_nil	Logical expression	2			
not_relexpr	Logical expression	2			
not_str	Logical expression	2			
not_t	Logical expression	2			
printExprList	Print expression	4			
relop_int_int1	Relational expression	2			
relop_int_int2	Relational expression	2			
relop_str_str1	Relational expression	2			
relop_str_str2	Relational expression	2			
strconst	Simple constant	1			
subGood	Arithmetic expression	2			
t	Simple constant	1			
input_arith	Input in arithmetic expression works with variety of integer inputs	5			
input_arith	Input in arithmetic expression correctly detects error for non-numeric inputs	5			
input_relop	Input in relational op expression works with variety of inputs	1			
All files	Error messages reported for correct line in input file	3			