CS 3500 – Programming Languages & Translators
Homework Assignment #4

- This assignment is **due by 8 a.m. on Thursday, October 19, 2017**.
- This assignment will be worth **10%** of your course grade.
- You may work on this assignment **with at most one other person in the class**.
- 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 HW #3 to make it perform **semantic analysis**. 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 wanted to create an output file (named *outputFileName*) of the results of running your program on *inputFileName*, you could use:

```
mfpl_parser < inputFileName > outputFileName
```

As in HW #3, **no attempt should be made to recover from non-lexical errors**; if your program encounters an error, it should simply output a meaningful message containing the line number where the error was found, and terminate execution. Listed below are **new errors** that your program also will need to be able to detect for MFPL programs:

Arg *n* must be integer
Arg *n* must be string
Arg *n* must be integer or string
Arg *n* cannot be function
Too many parameters in function call
Too few parameters in function call

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

Note that every parenthesized expression in MFPL (Mini Functional Programming Language) is of the form ***(function arg1 arg2 arg3 ...)***. For example, in the arithmetic expression *(+ x y)*, *x* is the first argument for the *+* function, and *y* is the second argument. If *x* is not an integer, then we should output the message "Arg 1 must be integer."

Your program should **still output the tokens, lexemes**, **productions being processed, open/close scope messages, and symbol table insertion messages.**

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; we'll do that in the next assignment! Consequently, you **do not have to record an identifier's value in the symbol table** – storing an identifier's name, type, and number of parameters and return type (if it is a function) should be sufficient for now.

Programming Language Semantics

What follows is a brief description about the semantic rules that we want to enforce for the various expressions in MFPL. This should serve as a guide for your type-checking.

The following descriptions assume that you have defined integer constants to represent the following types: **BOOL**, **INT, STR, FUNCTION, INT_OR_STR, INT_OR_BOOL, STR_OR_BOOL,** and **INT_OR_STR_OR_BOOL.**

N_EXPR → N_CONST | T_IDENT |
T_LPAREN N_PARENTHESIZED_EXPR T_RPAREN

The resulting type of an **N_EXPR** is the resulting type of the **N_CONST** if that rule is applied, or the actual type 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 type), or the resulting type of the **N_PARENTHESIZED_EXPR** if that rule is applied.

N_CONST → T_INTCONST | T_STRCONST | T_T | T_NIL

The resulting type of an **N_CONST** is **INT** if the **T_INTCONST** rule is applied, **STR** if the **T_STRCONST** rule is applied, or **BOOL** if the **T_T** or **T_NIL** rules are applied.

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

The resulting type of an **N_PARENTHESIZED_EXPR** is the resulting type of whichever rule is applied.

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 operand expressions of an **N_ARITHLOGIC_EXPR** (i.e., *arg1* and *arg2*) will need to be checked to see if they are appropriate for the operator being used. For **N_UN_OP**, the **N_EXPR** can be any type except **FUNCTION**. For **N_BIN_OP**, valid **N_EXPR** type combinations (regardless of order) are shown in the following table:

	relational ops (<, >, <=, >=, =, /=)	**logical ops (and, or)**	**arithmetic ops (*, +, -, /)**
INT, INT	legal	legal	legal
INT, STR	illegal	legal	illegal
INT, BOOL	illegal	legal	illegal
INT, FUNCTION	illegal	illegal	illegal
STR, STR	legal	legal	illegal
STR, BOOL	illegal	legal	illegal
STR, FUNCTION	illegal	illegal	illegal
BOOL, BOOL	illegal	legal	illegal
BOOL, FUNCTION	illegal	illegal	illegal
FUNCTION, FUNCTION	illegal	illegal	illegal

The resulting type of an **N_ARITHLOGIC_EXPR** will be **INT** if the operator is ***, +, -,** or **/**; otherwise, it will be **BOOL**.

N_IF_EXPR → T_IF N_EXPR N_EXPR N_EXPR

All three operand expressions (i.e., *arg1, arg2,* and *arg2,* respectively) of an if-expression can be any type except **FUNCTION** (and they don't all have to be the same type). At this time we don't know whether the second or third expression actually will be evaluated. So the resulting type of an **N_IF_EXPR** will be assigned based on the type combinations of the second and third expressions (regardless of their order) as shown in the following table:

	INT	**STR**	**BOOL**
INT	INT	INT_OR_STR	INT_OR_BOOL
STR	INT_OR_STR	STR	STR_OR_BOOL
BOOL	INT_OR_BOOL	STR_OR_BOOL	BOOL
INT_OR_STR	INT_OR_STR	INT_OR_STR	INT_OR_STR_OR_BOOL
INT_OR_BOOL	INT_OR_BOOL	INT_OR_STR_OR_BOOL	INT_OR_BOOL
STR_OR_BOOL	INT_OR_STR_OR_BOOL	STR_OR_BOOL	STR_OR_BOOL
INT_OR_STR_OR_BOOL	INT_OR_STR_OR_BOOL	INT_OR_STR_OR_BOOL	INT_OR_STR_OR_BOOL

	INT_OR_STR	**INT_OR_BOOL**	**STR_OR_BOOL**	**INT_OR_STR_OR_BOOL**
INT	INT_OR_STR	INT_OR_BOOL	INT_OR_STR_OR_BOOL	INT_OR_STR_OR_BOOL
STR	INT_OR_STR	INT_OR_STR_OR_BOOL	STR_OR_BOOL	INT_OR_STR_OR_BOOL
BOOL	INT_OR_STR_OR_BOOL	INT_OR_BOOL	STR_OR_BOOL	INT_OR_STR_OR_BOOL
INT_OR_STR	INT_OR_STR	INT_OR_STR_OR_BOOL	INT_OR_STR_OR_BOOL	INT_OR_STR_OR_BOOL
INT_OR_BOOL	INT_OR_STR_OR_BOOL	INT_OR_BOOL	INT_OR_STR_OR_BOOL	INT_OR_STR_OR_BOOL
STR_OR_BOOL	INT_OR_STR_OR_BOOL	INT_OR_STR_OR_BOOL	STR_OR_BOOL	INT_OR_STR_OR_BOOL
INT_OR_STR_OR_BOOL	INT_OR_STR_OR_BOOL	INT_OR_STR_OR_BOOL	INT_OR_STR_OR_BOOL	INT_OR_STR_OR_BOOL

Note: If you're clever about your choice of values for constants like INT, STR, INT_OR_STR, etc., and the use of C/C++ bitwise operators, you won't have to store this table in your program or have a hideous if/else statement!

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

For each **T_IDENT N_EXPR** pair within parentheses in **N_ID_EXPR_LIST**, the type of the **N_EXPR** establishes the type of the **T_IDENT**.

The type of **N_LET_EXPR** (i.e., this should be considered the *let** function's *arg2*) is the type of the **N_EXPR** at the very end of **N_LET_EXPR**. This can be any type except **FUNCTION**; if someone tried to make that be a function, it should be considered an error.

N_LAMBDA_EXPR → T_LAMBDA T_LPAREN N_ID_LIST T_RPAREN
N_EXPR

N_ID_LIST → ε | N_ID_LIST T_IDENT

To simplify things, assign the type of each **T_IDENT** in **N_ID_LIST** as **INT** (i.e., we'll assume that functions can only have integer parameters). The **N_EXPR** in **N_LAMBDA_EXPR** (i.e., what should be considered the *lambda* function's *arg2*) can be any type except **FUNCTION**; if someone tried to make that be a function, it should be considered an error. The overall type of an **N_LAMBDA_EXPR** is **FUNCTION**. The return type of the function is whatever type **N_EXPR** is, and the number of parameters for the function is the length of **N_ID_LIST**. **Note**: Recursive function calls are **not** supported in MFPL; the way we're managing the symbol table and doing the parsing should automatically catch a recursive function call as an undeclared identifier.

N_PRINT_EXPR → T_PRINT N_EXPR

The expression **N_EXPR** (i.e., *arg1*) can be any type except **FUNCTION**; if someone tried to make that be a function, it should be considered an error. The resulting type of an **N_PRINT_EXPR** is whatever is the type of the **N_EXPR**.

N_INPUT_EXPR → T_INPUT

Input can either be a string or an integer (we won't know until runtime). So for now the resulting type of a **N_INPUT_EXPR** should be considered **INT_or_STR**. **Note**: An expression that is of type **INT_or_STR** should be considered type-compatible with both type **INT** and type **STR**.

N_EXPR_LIST → N_EXPR N_EXPR_LIST | N_EXPR

The resulting type of an **N_EXPR_LIST** can be any type except **FUNCTION**; if someone tried to make that be a function, it should be considered an error. Make sure that if the very first **N_EXPR** is a function name or function definition (i.e., a lambda expression), that: (1) you check that the subsequent number of **N_EXPR**'s in the **N_EXPR_LIST** matches the number of parameters expected by that function, and (2) you assign the resulting type of the **N_EXPR_LIST** to be the function's return type.

Symbol Table Management

You will need to make some changes to the symbol table data structures to accommodate the 'type' information that we need to assign and check in this project. For example, you might find it useful to define the following to store pertinent information:

```
#define UNDEFINED           -1   // Type codes
#define FUNCTION            0
#define INT                 1
#define STR                 2
#define INT_OR_STR          3
#define BOOL                4
#define INT_OR_BOOL         5
#define STR_OR_BOOL         6
#define INT_OR_STR_OR_BOOL  7

#define NOT_APPLICABLE      -1

typedef struct
{
  int type;          // one of the above type codes
  int numParams;  // numParams and returnType only applicable if type == FUNCTION
  int returnType;
} TYPE_INFO;
```

Additional Tips on Semantic Error Detection

In order to do type checking in this assignment, you will need to specify what 'type' of information will be associated with various symbols in the grammar. As in HW #3, one way to do this is to define the following *union* data structure right after the **%}** in your mfpl.y file:

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

As in HW #3, the *char** type (which is aliased as ***text***) can be used to associate an identifier's name with an identifier token. The TYPE_INFO type (which is aliased as ***typeInfo***) can be used to associate a *struct* of type information with a nonterminal (or terminal). You'll need to define what type is to be associated with what grammar symbol by using %type declarations in your mfpl.y file such as the following:

```
%type  <text> T_IDENT
%type <typeInfo> N_CONST  N_EXPR  N_PARENTHESIZED_EXPR  N_IF_EXPR
```

Note that **not every symbol in the grammar has to be associated with a %type**; some symbols may not need any such information at this time (e.g., N_START, N_UN_OP, etc.). When you process a nonterminal during the parse, you can assign its 'type' (encoded as ***typeInfo***) as in the examples below:

```
N_CONST       : T_INTCONST
                {
                  printRule("CONST", "INTCONST");
                  $$.type = INT;
                  $$.numParams = NOT_APPLICABLE;
                  $$.returnType = NOT_APPLICABLE;
                }
…
N_EXPR        : T_LPAREN N_PARENTHESIZED_EXPR T_RPAREN
                {
                  printRule("EXPR", "( PARENTHESIZED_EXPR )");
                  $$.type = $2.type;
                  $$.numParams = $2.numParams;
                  $$.returnType = $2.returnType;
                }
```

You can then check the type of an expression within a particular context as in the following:

```
N_ARITHLOGIC_EXPR       : N_UN_OP N_EXPR
                        {
                        printRule("ARITHLOGIC_EXPR", "UN_OP EXPR");
                        if ($2.type == FUNCTION)
                        {
                        yyerror("Arg 1 cannot be function");
                        return(1);
                        }
                        $$.type = BOOL;
                        $$.numParams = NOT_APPLICABLE;
                        $$.returnType = NOT_APPLICABLE;
                        }
…
```

Note that the HW #3 output requirements for using particular names, error messages, etc. are still in effect for this assignment.

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 (broken down by what is being tested for in each sample input file). Note that the next assignment builds upon this one, so **it is critical that this assignment works properly in all respects!**

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)
arith_good	input as part of an arithmetic expression	1			
arith_nonIntArg1	non-int arg in arithmetic expression	1			
arith_nonIntArg2	non-int arg in arithmetic expression	1			
arith_nonIntArg3	non-int arg in arithmetic expression	1			
arith_nonIntArg4	non-int arg in arithmetic expression	1			
arith_nonIntArg5	non-int arg in arithmetic expression	1			
if_int_int_bool	if with different (legal) types of args	1			
if_int_int_input	if with different (legal) types of args	1			
if_int_int_int	if with different (legal) types of args	1			
if_int_int_str	if with different (legal) types of args	1			
if_nonIntArg1	if with illegal type of arg	1			
if_nonIntArg2	if with illegal type of arg	1			
if_nonIntArg3	if with illegal type of arg	1			
intconst_good	expression that is just an intconst	1			
lambda_good1	lambda with no params handled correctly	2			
lambda_good2	lambda with multiple params handled correctly	2			
lambda_good3	lambda with no params handled correctly	2			
lambda_nonIntOrStr	lambda expression can't be a lambda	4			
lambda_tooFewArgs	Too many/few params detected in lambda	3			
lambda_tooManyArgs	Too many/few params detected in lambda	3			
lambda_tooManyArgs2	Too many/few params detected in lambda	3			
let_good1	let* with no variables, type is int	1			
let_good2	let* with no variables, type is string	1			
let_good3	let* with no variables, type is int_or_str	1			

let_good4	let* with variables, type is function return type	4			
let_good4a	let* with variables, type is function return type	4			
let_good4b	let* with variables, type is function return type	4			
let_good4c	let* with variables, type is function return type	4			
let_good5	let* with variables, type is ident type	4			
let_nonIntOrStr	let* with no variables, type is lambda return type	4			
logop_bool_bool	logical operation between 2 bool's	1			
logop_functionArg1	logical operation between int and bool	2			
logop_functionArg2	logical operation between int and bool	2			
logop_input_input	logical operation between 2 input's	1			
logop_int_bool	logical operation between int and bool	1			
logop_int_int	logical operation between 2 int's	1			
logop_int_str	logical operation between int and str	1			
logop_str_bool	logical operation between bool and str	1			
logop_str_str	logical operation between 2 str's	1			
not_function	logical operation on a function return value (int)	2			
not_goodOnes	logical operations on several types	2			
print_good1	print string	1			
print_good2	print int	1			
print_good3	print string	2			
print_good4	print bool	1			
print_good5	print bool	1			
print_good6	print bool	1			
print_good7	print bool	2			
print_nonIntArg	print lambda	2			
relop_boolArg1	relational operation between bool and int	1			

relop_boolArg2	relational operation between bool and int	1			
relop_functionArg1	relational operation between int and function result (int)	2			
relop_functionArg2	relational operation between string and function result (int)	2			
relop_goodInts	relational operations on int's	2			
relop_goodStrs	relational operations on strings	2			
strconst_good	expression that is just an strconst	1			
All files	Error messages reported for correct line in input file	3			