CS 3500 – Programming Languages & Translators
Homework Assignment #4

- This assignment is **due by 11:59 p.m. on Friday, March 22, 2019.**
- This assignment will be worth **12%** 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 HW #3 to make it perform **semantic analysis**. 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 wanted to create an output file (named *outputFileName*) of the results of running your program on *inputFileName*, you could use:

```
minir_parser < inputFileName > outputFileName
```

As in HW #3, **no attempt should be made to recover from 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 list
Arg *n* must be integer or float or bool
Arg n cannot be null
Arg n cannot be list
Arg *n* cannot be function
Arg *n* cannot be function or list

Arg *n* cannot be function or null

Arg *n* cannot be function or null or list

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 Mini-R is a functional language. Every expression (i.e., statement) can be thought of as a function with arguments. For example, *if (x) y else z* can be thought of as a function named '*if*' that has 3 arguments: *x, y,* and *z*. Therefore, it there is a problem with one "part" of a expression/statement, we will word the error message in terms of which argument contains the error.

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 input until it processes *quit()* or encounters an error or detects end of input.

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 Mini-R. This should serve as a guide for your type-checking. These rules assume that you have defined integer constants to represent the following types:

NULL, **INT, STR, BOOL, FLOAT, LIST, FUNCTION, INT_OR_STR, INT_OR_BOOL, INT_OR_FLOAT, STR_OR_BOOL, STR_OR_FLOAT, BOOL_OR_FLOAT, INT_OR_STR_OR_BOOL, INT_OR_STR_OR_FLOAT, INT_OR_BOOL_OR_FLOAT, STR_OR_BOOL_OR_FLOAT,** and **INT_OR_BOOL_STR_OR_FLOAT** **Need to add combos for LIST**

Important notes about these constant names:

- **FUNCTION** means a function **definition**, NOT a function call!
- **NULL** does not mean 0 or FALSE or undefined! Instead it's kind of like *void* in C++.

Almost every nonterminal in the grammar will have a type associated with it. You will have to write code in your bison file to assign the nonterminals a value for their type, which will be dependent upon which production gets applied.

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

N_CONST → T_INTCONST | T_STRCONST | T_FLOATCONST | T_TRUE | T_FALSE

The resulting type of **N_CONST** is **INT** if the **T_INTCONST** rule is applied, **STR** if the **T_STRCONST** rule is applied, **FLOAT** if the **T_FLOATCONST** rule is applied, or **BOOL** if the **T_TRUE** or **T_FALSE** rules are applied.

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

The resulting type of **N_COMPOUND_EXPR** is the resulting type of **N_EXPR**. The resulting type of **N_EXPR_LIST** is also the type of its **N_EXPR**.

N_ARITHLOGIC_EXPR → N_SIMPLE_ARITHLOGIC |
N_SIMPLE_ARITHLOGIC N_REL_OP N_SIMPLE_ARITHLOGIC
N_SIMPLE_ARITHLOGIC → N_TERM N_ADD_OP_LIST
N_ADD_OP_LIST → N_ADD_OP N_TERM N_ADD_OP_LIST | ε
N_TERM → N_FACTOR N_MULT_OP_LIST
N_MULT_OP_LIST → N_MULT_OP N_FACTOR N_MULT_OP_LIST | ε
N_FACTOR → N_VAR | N_CONST | (N_EXPR) | T_NOT N_FACTOR
N_ADD_OP → T_ADD | T_SUB | T_OR
N_MULT_OP → T_MULT | T_DIV | T_AND | T_MOD | T_POW
N_REL_OP → T_LT | T_GT | T_LE | T_GE | T_EQ | T_NE

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. The following rules must be enforced:

- An operand can never be of type **FUNCTION** or **NULL** or **LIST** for any operator.
- Relational operators can have operands of any types (besides those listed above), and they don't necessarily have to be the same type.
- Arithmetic operators can only have operands of type **INT** or **FLOAT** or **BOOL.**

- Logical operators can only have operands of type **INT** or **FLOAT** or **BOOL.**

The resulting type of an **N_ARITHLOGIC_EXPR** will be **INT** if the operator is arithmetic and neither of the operands is **FLOAT**; if one of those operands is **FLOAT**, the resulting type will be **FLOAT**. If the operator is relational or logical, the resulting type will be **BOOL**.

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

All three operand expressions (i.e., *arg1, arg2,* and *arg2,* respectively) of an *if-expression* can be any type except **FUNCTION**. An exception to that is the first expression cannot be type **LIST**. The expressions do not all have to be the same type. At this time, we don't know whether the second or third expression actually will be evaluated (i.e., the 'then' or the 'else'). Therefore, 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). A table showing only some of these combinations is shown below:

	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 a big table in your program or have a hideously complex if/else statement!

N_WHILE_EXPR → T_WHILE (N_EXPR) N_LOOP_EXPR

The **N_EXPR** of a *while-expression* can be any type except **FUNCTION, LIST,** or **NULL**. The resulting type of **N_WHILE_EXPR** is the type of **N_LOOP_EXPR**.

N_FOR_EXPR → T_FOR (T_IDENT T_IN N_EXPR) N_LOOP_EXPR

The **N_EXPR** of a *for-expression* can be any type except **FUNCTION** or **NULL**. The resulting type of **N_FOR_EXPR** is the type of the **N_LOOP_EXPR**.

The type of the **T_IDENT** has to be compatible with the type of the **N_EXPR**. If the **N_EXPR**'s type is **LIST**, then the **T_IDENT**'s type must be compatible with **INT**, **STRING**, **BOOL**, or **FLOAT** (since lists can contain different types of values; we won't worry about exactly what a particular list contains for now). But if the type of the **N_EXPR** is **not LIST**, then the type of the **T_IDENT** must be compatible with the **N_EXPR**'s type (e.g., they must both be **STRING**).

If the **T_IDENT** already exists in some symbol table prior to this *for-expression*, you need to check that its type works for the type of the **N_EXPR** (as explained above). If the **T_IDENT** didn't already exist (i.e., you're making a symbol table entry for it), then its type be assigned as the type of the **N_EXPR** if **N_EXPR's** type is **not LIST,** or **INT_OR_STR_OR_FLOAT_OR_BOOL** if **N_EXPR's** type is **LIST**.

N_BREAK_EXPR → T_BREAK

N_NEXT_EXPR → T_NEXT

N_QUIT_EXPR → T_QUIT( )

The resulting type of **N_BREAK_EXPR**, **N_NEXT_EXPR,** and **N_QUIT_EXPR** is **NULL**.

N_LIST_EXPR → T_LIST (N_CONST_LIST)

The resulting type of **N_LIST_EXPR** is **LIST**.

N_ASSIGNMENT_EXPR → T_IDENT N_INDEX = N_EXPR

The type of **N_EXPR** can be any type, including **FUNCTION** and **NULL**. The resulting type of **N_ASSIGNMENT_EXPR** is the type of **N_EXPR**. If the **T_IDENT** already exists in some symbol table prior to this *assignment-*

expression, you need to check that its type is compatible with the type of the **N_EXPR** (otherwise, that's an error!); if the **T_IDENT** didn't already exist (i.e., you're making a symbol table entry for it), then its type should be the type of the **N_EXPR**. Also note that an **N_INDEX** that is **not epsilon** is only valid for a **T_IDENT** that is of type **LIST**!

N_OUTPUT_EXPR → T_PRINT (N_EXPR) | T_CAT (N_EXPR)

The expression **N_EXPR** (i.e., *arg1*) can be any type except **FUNCTION** or **NULL**. The resulting type of **N_PRINT_EXPR** is whatever is the type of the **N_EXPR** if the production with **T_PRINT** is used; otherwise, the type of **N_PRINT_EXPR** is **NULL**.

N_INPUT_EXPR → T_READ (N_VAR)

Input can either be a string, an integer, or a float (we won't know until runtime). For now, the resulting type of a **N_INPUT_EXPR** should be considered **INT_or_STR_or_FLOAT**. **Note**: An expression that is of type **INT_or_STR_or_FLOAT** should be considered type-compatible with type **INT** and type **STR** and type **FLOAT** (and any combinations thereof).

N_FUNCTION_DEF → T_FUNCTION (N_PARAM_LIST)
N_COMPOUND_EXPR

N_PARAM_LIST → N_PARAMS | N_NO_PARAMS

N_NO_PARAMS → ε

N_PARAMS → T_IDENT | T_IDENT , N_PARAMS

To simplify things, assign the type of each **T_IDENT** in **N_PARAMS** as **INT** (i.e., we'll assume that functions can only have integer parameters). The **N_COMPOUND_EXPR** in **N_FUNCTION_DEF** (i.e., what should be considered it's *arg2*) can be any type except **FUNCTION**. The overall type of an **N_FUNCTION_DEF** is **FUNCTION**. The return type of the function is whatever type **N_COMPOUND_EXPR** is, and the number of parameters for the function is the length of **N_PARAM_LIST**. **Note**: Recursive function calls are not supported in Mini-R; the way we're managing the symbol table and doing the parsing doesn't handle this.

N_FUNCTION_CALL → T_IDENT (N_ARG_LIST)

N_ARG_LIST → N_ARGS | N_NO_ARGS

N_NO_ARGS → ε

T_IDENT must be of type **FUNCTION**. The length of **N_ARG_LIST** must be the same as what was declared for this function; you'll need to look up **T_IDENT** in the symbol table to see what you recorded for it when you saw it declared as a function. The resulting type of **N_FUNCTION_CALL** is the return type of this function; again, you should have recorded that for **T_IDENT** in the symbol table when you saw its definition as a function.

N_ARGS → N_EXPR | N_EXPR , N_ARGS

Each **N_EXPR** must be of type **INT** since we have restricted function parameters to only being integers.

N_VAR → N_ENTIRE_VAR | N_SINGLE_ELEMENT
N_SINGLE_ELEMENT → T_IDENT [[N_EXPR]]

N_ENTIRE_VAR → T_IDENT

The **N_VAR**'s type comes by looking up **T_IDENT** in the symbol table(s). Remember that at the time that **N_VAR** is referenced (i.e., used) in an expression, it should be in some symbol table; otherwise, it should get flagged as undefined. The only ways it could have gotten into a symbol table were: (1) it was used on the left-hand side of an assignment statement, (2) it was a parameter in a function definition, or (3) it is the loop variable in **N_FOR_EXPR**. In the first two cases, you definitely know its type. However, in the third case, it could be iterating through a list that is made up of different types of constants; in that case, you must consider its type to be **INT_OR_STR_OR_BOOL_OR_FLOAT**. So, here, **N_VAR** gets its type from either **N_ENTIRE_VAR** (i.e., look up **T_IDENT** in the open symbol table(s)), or from **N_SINGLE_ELEMENT** (in which case **T_IDENT** must be looked up in the open symbol table(s), checked to make sure it is of type **LIST**, and then considered type **INT_OR_STR_OR_BOOL_OR_FLOAT**).

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_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_REL_OP, etc.). When you process a nonterminal during the parse, you can assign its 'type' (encoded as ***typeInfo***) as in the examples below:

Give some sample code here

```
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!**