CS 3500 – Programming Languages & Translators
Homework Assignment #4a

- This assignment is **due by 11:59 p.m. on Friday, March 22, 2019.**
- This assignment will be worth **9%** 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 **bash script** has been posted that will run your program on all of the input files, compare your output to our expected output, and tell you which files differed in the output. Instructions for running that script can be found by opening the script file with a simple text editor (like WordPad) and reading the comments at the top. If you open it with a Windows editor, it will add \r characters to the script file; you'll then have to run *dos2unix* on the script file to remove those characters before executing the script with bash.

Basic Instructions

For this assignment you are to modify your HW #3 to make it perform **part** of the necessary **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 will need to be able to detect for Mini-R programs (although **not all of these errors** will **actually** need to be detected in **this part** of your semantic analyzer):

Arg *n* must be integer
Arg *n* must be list
Arg *n* must be function
Arg *n* must be integer or float or bool

Arg *n* cannot be function
Arg *n* cannot be list
Arg *n* cannot be function or null
Arg *n* cannot be function or null or list or string
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 an expression/statement, we will word the error message in terms of which argument contains the error.

IMPORTANT: We no longer want your program to output the tokens, lexemes, productions being processed, the open/close scope messages, and the symbol table insertion messages. However, that is valuable output for debugging your program. It is recommended that you add a variable or constant to your program that you can set to turn on/off that output.

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

IMPORTANT: We have decided to make some changes to the Mini-R grammar to simplify the processing for semantic analysis and evaluation (a future assignment). We are eliminating the *break* and *next* statements for loops. You no longer need to have productions for **N_LOOP_EXPR, N_BREAK_EXPR,** or **N_NEXT_EXPR**. And the productions for **N_WHILE_EXPR** and **N_FOR_EXPR** should now be:

N_WHILE_EXPR → T_WHILE (N_EXPR) N_EXPR
N_FOR_EXPR → T_FOR (T_IDENT T_IN N_EXPR) N_EXPR

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 in this part of the semantic analyzer. 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**

Only one "combination" type will be required for **this part** of your semantic analyzer; that type is **INT_OR_STR_OR_BOOL_OR_FLOAT**.

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** also is the type of its **N_EXPR**; for the epsilon production of **N_EXPR_LIST** you won't need to set any type information for **N_EXPR_LIST** because there is nothing being processed.

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

If-expressions will not be processed in this part of the semantic analyzer. They will not appear in any of the test files.

N_WHILE_EXPR → T_WHILE (N_EXPR) N_EXPR

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

N_FOR_EXPR → T_FOR (T_IDENT T_IN N_EXPR) N_EXPR

For-expressions will not be processed in this part of the semantic analyzer. They will not appear in any of the test files.

N_QUIT_EXPR → T_QUIT( )

The resulting type of **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 **N_INDEX** was **not** the epsilon production, the **T_IDENT** must have already existed before this statement and be type **LIST**, and the **N_EXPR** will not be allowed to be type **LIST**.

Note: If the T_IDENT doesn't already exist in the most recently created symbol table, then add it to the most recently created symbol table.

N_OUTPUT_EXPR → T_PRINT (N_EXPR) | T_CAT (N_EXPR)

The expression **N_EXPR** 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-expressions will not be processed in this part of the semantic analyzer. They will not appear in any of the test files.

N_FUNCTION_CALL → T_IDENT (N_ARG_LIST)

N_ARG_LIST → N_ARGS | N_NO_ARGS

N_NO_ARGS → ε

Function calls do not need to be processed in this part of the semantic analyzer. They will not appear in any of the test files.

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

The overall type of an **N_FUNCTION_DEF** is **FUNCTION**. Function parameters should go into the symbol table as type **INT**. Nothing else needs to be done with function definitions for this part of the semantic analyzer (e.g., you don't need to record the number of parameters a function has or its return type because we won't be type-checking function calls yet).

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 is determined 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**. Since this part of the semantic analyzer is not dealing with *for-loops*, you don't have to worry about the third case.

If the **N_VAR** is an **N_SINGLE_ELEMENT**, you need to check whether the **T_IDENT**'s type was **LIST**; if not, it is an error to be indexing it! Also, since lists can contain values of different types, we don't know what type an indexed variable could be. Therefore, the type of **N_SINGLE_ELEMENT** should be **INT_OR_STR_OR_BOOL_OR_FLOAT**. This is the only "combination" type you will have in this part of the semantic analyzer; this special type should be considered "compatible" with any of its constituent types.

N_ARITHLOGIC_EXPR → N_SIMPLE_ARITHLOGIC |
N_SIMPLE_ARITHLOGIC N_REL_OP N_SIMPLE_ARITHLOGIC

In general, the operand expressions of a binary arithmetic/logical expression (i.e., *arg1* and *arg2*) will need to be checked to see if they are appropriate for the operator being used. The following rule must be enforced:

- Relational, arithmetic, and 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**.

An arithmetic/logical expression may be derived from several nonterminals and their productions. Given below is some guidance in what you will need to consider for assigning type information. **Most of the following nonterminals will require some type checking in their productions.**

N_SIMPLE_ARITHLOGIC → N_TERM N_ADD_OP_LIST

The type of **N_SIMPLE_ARITHLOGIC** is the determined by the types of **N_TERM** and **N_ADD_OP_LIST**, the latter of which could have just been an epsilon production.

N_ADD_OP_LIST → N_ADD_OP N_TERM N_ADD_OP_LIST | ε

The type of **N_ADD_OP_LIST** depends on whether the **N_ADD_OP** is an arithmetic or logical operator. If it's the former, the type could be **INT** or **FLOAT**; it depends on **N_TERM**'s type and **N_ADD_OP_LIST**'s type. If it's the latter (i.e. a logical operator), then the type is **BOOL**.

N_TERM → N_FACTOR N_MULT_OP_LIST

The type of **N_TERM** is the determined by the types of **N_FACTOR** and **N_MULT_OP_LIST** (although the latter may have just been an epsilon production).

N_MULT_OP_LIST → N_MULT_OP N_FACTOR N_MULT_OP_LIST | ε

The type of **N_MULT_OP_LIST** depends on whether the **N_MULT_OP** is an arithmetic or logical operator. If it's the former, the type could be **INT** or **FLOAT**; it depends on **N_FACTOR**'s type and **N_MULT_OP_LIST**'s type. If it's the latter (i.e. a logical operator), then the type is **BOOL**. If the epsilon production is applied, the **N_MULT_OP_LIST**'s type is not applicable.

N_FACTOR → N_VAR | N_CONST | (N_EXPR) | T_NOT N_FACTOR

The type of **N_FACTOR** is the type of **N_VAR, N_CONST, N_EXPR,** or **N_FACTOR** depending upon which respective production is applied.

N_ADD_OP → T_ADD | T_SUB | T_OR

You may find it useful in **N_ADD_OP_LIST** if here you associate an attribute with **N_ADD_OP** that indicates whether this is an arithmetic operator (i.e., **T_ADD** or **T_SUB**) or a logical operator (i.e., **T_OR**).

N_MULT_OP → T_MULT | T_DIV | T_AND | T_MOD | T_POW

You may find it useful in **N_MULT_OP_LIST** if here you associate an attribute with **N_MULT_OP** that indicates whether this is an arithmetic operator (i.e., **T_MULT, T_DIV, T_MOD,** or **T_POW**) or a logical operator (i.e., **T_AND**).

N_REL_OP → T_LT | T_GT | T_LE | T_GE | T_EQ | T_NE

There really isn't anything you need to do for **N_REL_OP**.

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 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;
```

This will require that you make changes to some of the **SYMBOL_TABLE** and **SYMBOL_TABLE_ENTRY** functions. For example, it will no longer be sufficient for *findEntry* to just return a bool indicating whether or not it found a name in the symbol table; now it must return the **TYPE_INFO** associated with the name if it found it.

Hint: In the symbolTable.h file given to you for HW 3, *addEntry* was inserting into the hash table a data structure called a ***pair*** (see http://www.cplusplus.com/reference/utility/pair/pair/), which has two member variables: *first* and *second*. The *pair* it is inserting has a *first* that is a *string* name and a *second* that is a *TYPE_INFO* struct. The function *findEntry* is using something called an *iterator* to go through the hash table and look for a name. If it finds the name, the variable *itr* will be pointing at a *pair*.

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 .y file:

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

In HW #3, the *char** type (which is aliased as ***text***) was 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 .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 example below:

```
N_CONST       : T_INTCONST
                {
                 printRule("CONST", "INTCONST");
                 $$.type = INT;
                 $$.numParams = NOT_APPLICABLE;
                 $$.returnType = NOT_APPLICABLE;
                }
```

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

```
N_OUTPUT_EXPR   : T_PRINT T_LPAREN N_EXPR T_RPAREN
                  {
                   printRule("OUTPUT_EXPR", "PRINT ( EXPR )");
                   if (($3.type == FUNCTION) || ($3.type == NULL_TYPE))
                  {
                    yyerror("Arg 1 cannot be function or null");
                  }
                  $$.type = $3.type;
                  $$.numParams = $3.numParams;
                  $$.returnType = $3.returnType;
                  }
```

Additional Tips on Bison

If you intersperse a C/C++ code block (i.e., { }) between grammar symbols on the righthand side of a production, there are some important things you need to be aware of:

- It will affect the "numbering" (i.e., indexing) of the symbols on the righthand side (i.e., $1, $2, etc.) because each code block counts as a number.
- A code block cannot access information for a symbol that comes after it on the righthand side of the production (e.g., if you put { } after symbol $1, then that code can only access attribute information about $1; it can't access information about $3, $4, etc. because those haven't been processed yet).

- The only code block that can assign any information to $$ is the one that comes at the very end of the righthand side symbols of the production.

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!!!

A grading rubric will be 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). Note that the next assignment builds upon this one, so **it is critical that this assignment works properly in all respects! You also will not have a partner on the next assignment, so it is important that you understand what is going on in all aspects of this assignment if you work with a partner!**