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

- This assignment is **due by 11:59 p.m. on Friday, April 5, 2019.**
- This assignment will be worth **6%** of your course grade.
- You are to work on this assignment **BY YOURSELF!**
- Before you submit your assignment for grading, you should test it on the sample input files posted on the Canvas website. The same **bash script** posted for HW #4a can be used to run all of the sample input files for this assignment so that you can see which ones produce output different from what we are expecting.

Basic Instructions

For this assignment you are to modify your HW #4a to make it finish the necessary **semantic analysis** for Mini-R. 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 #4a, **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 **all of the semantic errors** that your program will need to be able to detect for Mini-R programs:

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

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

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. Only listed are **additions (or changes)** to what was specified for HW #4a.

N_ASSIGNMENT_EXPR → T_IDENT N_INDEX = N_EXPR

If a **T_IDENT** is a **parameter** in a function definition, it should be in the symbol table as type **INT**. Its **type cannot be changed** with an assignment statement (like non-parameter variables can be); it can only be assigned a value that is "compatible" with an **INT**.

N_INPUT_EXPR → T_READ (~~N_VAR~~)

IMPORTANT: To make things easier, we're changing the production for **N_INPUT_EXPR** to no longer have **N_VAR**.

Input can either be a string, an integer, or a float (we won't know until runtime). So, 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 types **INT**, **STR**, and **FLOAT** (and any combinations thereof).

N_FOR_EXPR → T_FOR (T_IDENT T_IN N_EXPR) N_EXPR

To keep things simple, we will stipulate that the first **N_EXPR** of a *for-expression* can only be type **LIST**. The resulting type of **N_FOR_EXPR** is the type of the second **N_EXPR**; that can be of any type.

If the **T_IDENT** already exists **in the most recently created** symbol table prior to processing this *for-expression*, you need to check that its type is compatible with **INT_OR_STR_OR_FLOAT_OR_BOOL**; otherwise, it is an error. If the **T_IDENT** doesn't already exist **in the most recently created** symbol table (i.e., you're making a symbol table entry for it), then its type should be assigned as **INT_OR_STR_OR_FLOAT_OR_BOOL**.

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

IMPORTANT: If using the above productions for an *if-expression*, we would need to intersperse code blocks into the productions in order to type-check the **N_EXPR**s and output the error messages on the correct line numbers. Doing this would likely introduce ambiguity, and hence conflicts, into the grammar because *bison* treats the code blocks kind of like grammar symbols and it would have trouble determining which production to apply. Therefore, we are changing the grammar for how *if-expressions* are to be parsed as indicated below:

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

The very first **N_EXPR** of an *if-expression* (which is now in **N_COND_IF**) can be any type except **FUNCTION, LIST, NULL**, or **STR.** The "then" and "else" expressions can be any type except **FUNCTION**, and they do not have to be the same type.

If there isn't an "else" expression, then the type of the **N_IF_EXPR** is the type of the "then" expression. If there is an "else" expression, we don't know (at this time) whether the "then" or "else" expression actually will be evaluated. Therefore, the resulting type of an **N_IF_EXPR** should be assigned based on an "OR" type combination of the types of the "then" and "else" expressions.

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, in HW #4a we had you assign the type of each **T_IDENT** in **N_PARAMS** as **INT** (i.e., we're assuming 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**; by the way, that's also the number of entries in the function's symbol table right after you process **N_PARAM_LIST** in **N_FUNCTION_DEF**.

Note: Recursive function calls are not supported in Mini-R because we don't know a function's return type until *after* we are finished processing **N_COMPOUND_EXPR** (which we wouldn't know yet if we're making a recursive call to the function). A recursive function call should get flagged as an undefined identifier.

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 compatible with type **INT** since we have restricted function parameters to only being integers.

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** if **N_EXPR_LIST** was just the epsilon production; otherwise; it's the type of **N_EXPR_LIST**. The same goes for setting the type of **N_EXPR_LIST** based on its productions. Note: This is slightly different than what was said in the directions for HW #4a.

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**). 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 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!**