CS 445 – Spring 2022

Homework 3

Due on or before 27 Feb at 11:59 PM Pacific Time.

WARNING

Because the output of your program for this assignment will first be processed by an automatic comparison program before being examined by a human being, please follow formatting instructions/examples very carefully. The results your program produces will need to **look exactly like the examples**. Do not embellish with extra titles or other text such as "run complete" or "CS445 output" or even any extra spaces. Points will be deducted for this. (This is realistic. Most companies run test suites on their products and breaking the test suites is not looked upon well in industry.) The testing facility of the submit script will help you refine your software to the point that it is completely correct if you use it as intended.

1. The problem

In this assignment we will type expressions in the abstract syntax tree (AST) and start to perform semantic analysis and error generation. We will also check to ensure that variables are declared before being used and be able to warn when a variable might not be initialized before being used.

1.1 New Compiler Options

For this assignment your compiler will handle five command-line options. Some of these are repeats from the last assignment. The following are the command-line options that your compiler will handle:

1. The -d option which sets the value of the yydebug variable to 1. This is a repeat from the last assignment.

2. The -D option which turns on symbol table debugging. Enabling this option will cause a line of information to be printed for each action that your compiler performs with the symbol table. If you are using the provided symbolTable.cpp code, just enable debugging in the SymbolTable instance that your compiler creates.

3. The -p option which is a repeat from the last assignment.

4. The -P option which prints the AST with type information added. This information is printed for symbols at their declaration and their use. The symbolTable.cpp code will assist with this option.

5. The -h option which will print the following usage message:

```
usage: -c [options] [sourcefile]
options:
-d          - turn on parser debugging
-D         - turn on symbol table debugging
-h          - print this usage message
-p          - print the abstract syntax tree
-P          - print the abstract syntax tree plus type information
```

Your c- must continue to accept a single input file from either a file name provided on the command-line or redirected from standard input.

1.2 Reorganize Your Code

Put your semantic analysis code in `semantic.cpp` and `semantic.h`. If you use the ourgetopt code, place that in `ourgetopt.cpp` and `ourgetopt.h`. Update your makefile to build these targets by putting their .o files in the dependency list for building c- and the line used to build c- with g++. You may find it useful and educational to put your `main()` and associated routines in their own `main.cpp` file if you haven't done so already. This is not required.

1.3 Semantic Errors

We want to generate errors that are tagged with line numbers that are useful to the user. In order to accomplish this, you will need to ensure that each node is tagged with the correct line number. To do this effectively you will need to grab the line number as soon as possible (in flex) and associate it with a given token. This can be done nicely (and portably) by passing a struct/class for each token from the scanner to the parser in the `yylval`. This should contain all the information about the token such as the line number, lexeme, constant value, and token class. You are probably already doing this. Recall that you should avoid using pointers to global yy variables for token information because the parser generated by yacc looks ahead and will possibly overwrite the variable that you are pointing to with new information.

Once the information is passed in some form of struct/class, this information can then be stored in nodes within your AST. This information can then used when traversing the AST to perform semantic analysis.

1.4 Scope and Type Checking

After checking if you should print the AST (just like in the last assignment) you will now traverse the AST and search for typing and program structure errors. Your new main() might look something like this:

```
numErrors = 0;
numWarnings = 0;

yyparse();

if (numErrors == 0)
{
    // check -p option
```

```
    if (printSyntaxTree)
        // print only info for declarations
        printTree(syntaxTree, NOTYPES);

    symbolTable = new SymbolTable; // instantiate the symbol table

    // perform semantic analysis (may find errors when doing this)
    semanticAnalysis(syntaxTree, symbolTable);

    // check -P option
    if (printAnnotatedSyntaxTree)
        // print type info for all types
        printTree(syntaxTree, TYPES);

    // code generation will eventually go here…
}


// report number of errors and warnings
printf("Number of errors: %d\n", numErrors);
printf("Number of warnings: %d\n", numWarnings);
```

Or your main() might look quite different. You will likely want to perform more initialization before you call yyparse(). The `semanticAnalysis` routine will process the AST by calling a `treeTraverse` routine that will start at the root node of the AST and recursively call itself for children and siblings until it reaches the leaf nodes in the tree. Declarations encountered during this process will make entries in the symbol table (the symbol table is is discussed below) and references to symbols will be checked by looking up the symbol name in the symbol table. Your goal in writing the `treeTraverse` routine is to catch a variety of warnings and errors that are indicated in the example output files for this assignment. You will need to duplicate the output presented in these example output files exactly for each corresponding input file.

You should keep track of the number of errors and warnings and print them at the end of a run. The list of errors and warnings that your c- should emit are shown below as format specifiers in printf format.

```
"ERROR(%d): '%s' is a simple variable and cannot be called.\n"
"ERROR(%d): '%s' requires both operands be arrays or not but lhs is%s an array and
            rhs is%s an array.\n"
"ERROR(%d): '%s' requires operands of %s but lhs is of %s.\n"
"ERROR(%d): '%s' requires operands of %s but rhs is of %s.\n"
"ERROR(%d): '%s' requires operands of the same type but lhs is %s and rhs is %s.\n"
"ERROR(%d): Array '%s' should be indexed by type int but got %s.\n"
"ERROR(%d): Array index is the unindexed array '%s'.\n"
"ERROR(%d): Cannot index nonarray '%s'.\n"
"ERROR(%d): Cannot return an array.\n"
"ERROR(%d): Cannot use function '%s' as a variable.\n"
"ERROR(%d): Symbol '%s' is already declared at line %d.\n"
"ERROR(%d): Symbol '%s' is not declared.\n"
"ERROR(%d): The operation '%s' does not work with arrays.\n"
"ERROR(%d): The operation '%s' only works with arrays.\n"
"ERROR(%d): Unary '%s' requires an operand of %s but was given %s.\n"
"ERROR(LINKER): A function named 'main()' must be defined.\n");
"WARNING(%d): The variable '%s' seems not to be used.\n"
"WARNING(%d): Variable '%s' may be uninitialized when used here.\n
```

To get a perfect match to the expected output it helps immensely if you just copy these format strings and use them in your code. The type string that you replace in the message is often something like "type int" or "unknown type". The messages listed above are exactly the errors/warnings that you must catch for this assignment. There are 16 error messages and 2 warning messages for this assignment. In later assignments we will add more.

1.5 Type Checking for Specific AST Nodes

This section contains details regarding some errors that you should be checking for. This list is not exhaustive. You can do whatever you like as long as your output matches the example output file corresponding to a given input.

1.5.1 Declarations

For declarations, check to ensure that duplicate declarations do not exist. You can use the symbol table to help with this. Recall that the arguments to a function in C- are contained in the outermost scope of the function in which they appear (the first compound statement in the function definition). For example, this C- code

```
int func(int x) begin int x; end
```

contains a duplicate declaration of the symbol x, whereas this C- code

```
int func(int x) begin begin int x; end end
```

does not contain a duplicate declaration since the innermost compound statement is a different scope than the outermost compound statement. This is the way that C++ works.

1.5.2 Compound Statements

A new scope needs to be created for each compound statement. The `symbolTable.cpp` code that is provided will permit you to created and destroy scopes, enter symbols in the symbol table, and check to see if a symbol is already contained in the table. For documentation purposes, you can ascribe a label of your choice to a given scope using the `SymbolTable::enter` method. For example, enter("Compound Statement") will create a new scope with the label "Compound Statement" associated with it. If you enable debugging in the symbol table, using a distinct identifier for each scope name may make the debugging information more readable.

1.5.3 Assignments and Other Operators

Your code should check to ensure that types provided for assignments and other operators are correct. The type information associated with an expression will have to be passed up the tree (synthetic attributes) so that they can be checked by the operators that use it. Note that many operators have an expected type that they produce. The + operation, for example, will produce a type int. Be sure to set this sort of information as you traverse the tree. It might be useful to have an "undefined type" or "placeholder type" for use when the type of an entity is undefined or is not yet known.

Consider using an array or clever function to determine what types are required for operands to specific operators rather than using nested if statements. Use this same strategy to determine what type the

result of an operation will be. The number of cases required for type checking has been intentionally limited, making it easier to design functions for this purpose. The types of all operands and return types for all C- operators are contained in the C- grammar document on the course website. Note the following:

1. The = and != operators take operands that are the same type and produce a value of type Boolean. The operands to both of these operators can be arrays.
2. The <- operator takes operands that are of the same type and produces the type of the lhs. This means that the <- operator will produce the type of the lhs regardless of whether or not it is undefined. This is because assignment in C- is an expression and can be used in cascaded assignment like: a <- b <- c <- 314159265; (An intentional and interesting side note is that an assignment is NOT a simpleExp! How does that effect what you can put in say... an if statement predicate? Python does this sort of thing.)
3. The ++ and - - operators in C- take an integer operand and *immediately* produce an integer result. This is different than the way that they behave in C/C++.

1.5.4 Identifiers

All C- identifiers must be unique, and variables must be declared before they are used. Your code must check to see if a variable has been declared. If a referenced identifier has been used in a declaration, set the type of the identifier node to the type of the declaration and associate a pointer to the declaration node with the symbol table information for that identifier. This is **very** useful because it permits you to quickly find your way to the declaration node of a given variable and allows you to store all of the information about a given variable with its declaration. If the declaration associated with a given identifier cannot be found, set the type of the identifier to an “unknown type” or some other indicator that the type information is missing. To prevent cascading errors, undefined types do not provoke an error when being compared with an expected type (e.g., int produced by the + operator). We assume that an error was generated when a given identifier was marked as being of an unknown or undefined type. For this assignment, each reference to an undeclared identifier must generate an error message. This will likely be changed later.

For this assignment you must issue a warning for the possibility that a variable is being used before it was initialized. This applies strictly to locally declared variables; not globals, statics, or parameters. If the rhs value of a variable is used before (appears in the tree traversal before) it has been initialized or appears on the lhs of an assignment, a warning must be issued. This means that an identifier being on the lhs in a binary assignment or being initialized on declaration will cause the variable to be marked as initialized.

For identifiers, C- permits arrays that are indexed. An identifier used in this form becomes a non-array type because it has been indexed. This means that type type of the AST ‘[‘ node is the type of the lhs of the operation. Be sure to check for attempting to index an identifier that is a non-array type and using identifiers of an array type in a non-indexed manner where this is illegal. Identifiers that are arrays can be prefixed with the * operator to determine the number of items in the array. This operator produces an int.

1.5.5 Functions

Functions in C- cannot return an entire array. Your c- must check for this and ensure that is not permitted. Functions in C- can have a return type of `void`. This means that the specified function cannot return a value. An implication of this is that it is possible for the type `void` to appear in an error message. All C- programs must contain a function named `main`. If, after processing the entire tree representing a program, there is no function named `main` in the symbol table, you must print an error message of the form indicated earlier in this assignment.

1.6 The Symbol Table

A symbol table that you can use in your program is available on the course website. The version provided uses the C++ standard template library. Feel free to augment it or to build your own. Though the version provided uses the `std::string` type as the argument type in many places, you can convert a `const char *` to and from a `std::string`. The `SymbolTable::insert` method allows you to associate a pointer with a symbol in the table. This pointer could be used to associate a symbol name with the tree node where the symbol was declared. The `SymbolTable::enter` and `SymbolTable::leave` methods allow you to manage the stack of scopes associated with a program. The scopes within a SymbolTable are always managed via these methods. The `SymbolTable::insert` method will return false if you attempt to insert a symbol that has already been defined. The `SymbolTable::lookup` method will return `NULL` if you attempt to look for a symbol that is not defined.

One feature of the symbol table is the debug flag. At construction time the SymbolTable object is in non-debugging mode. But by setting the flag with the `SymbolTable::debug` method you can get the symbol table to print out information. You can also just print the symbol table if using the provided `SymbolTable::print` method. You might consider starting out by printing the symbol table when exiting from a scope using the debug flag.

The `SymbolTable::print` method takes a print function that will print a tree node. So if you define something to print the data in a node given a `TreeNode*` then you can supply the name of that print function to the `SymbolTable::print` method to print out your symbol table stack. That way the code doesn't have to know what your TreeNode looks like internally. For instance

```
symtab = new SymbolTable();
```

creates the symbol table. If you have defined a function to print a tree node with a prototype like this:

```
void nodePrint(void *p)
```

You can print the symbol table by just doing this:

```
symtab->print(nodePrint);
```

which will print each void * in the symbol table using your supplied `nodePrint` function.

2. Deliverables and Submission

Your homework will be submitted as an *uncompressed* .tar file that contains no subdirectories. Your .tar file must contain all files and code necessary to build and test your compiler. If you use ourgetopt, *be sure to include the ourgetopt files in your .tar archive***.**

The .tar file is submitted to the class submission page. You can submit as many times as you like. **The last file you submit before the deadline will be the only one graded.** No late submissions are accepted for this assignment. For all submissions you will receive email at your uidaho address showing how your file performed on the pre-grade tests. The grading program will use more extensive tests, so thoroughly test your program with inputs of your own. The error messages that your c- emits will be sorted before comparison with the expected output so that the order in which your messages are printed is not as important as it otherwise would be.

Your code must compile successfully and have no shift/reduce or reduce/reduce conflicts from bison. Your program must run with no runtime errors such as segmentation violations (SIGSEGVs).