Programming Assignment 3

(Based on Material By: R. Heckendorn)

Due: Sunday, Oct. 8 @ 11:59pm

The Problem

In this assignment, we will type our abstract syntax tree (AST) and start to do semantic analysis and semantic error generation. We will also add some command line options and an option to print out the tree.

New compiler options:

This time we add two new options to the **`-d`** option from the last assignment. The first option is **`-p`** (lower case p) which prints the abstract syntax tree. That is, it prints the syntax tree we did for assignment 2. The **`-p`** option only works if there are no syntax errors.

The second option is **`-P`** (cap p) which prints the syntax tree as above plus prints type of all blocks in your AST. Many of these, like if will be undefined. But many will have values.

IMPORTANT: you will be graded on the output of both tree prints. If you didn't get your tree perfect in assignment 2, now you can! The **`-P`** option only works if there are no syntax errors. Semantic errors are allowed.

c- should still accept a single input file either from a filename given on the command line or redirected as standard input.

We will continue adding options throughout the semester. Using getopt will make your life easier and let you focus on the compiler stuff. Here is a useful link for getopt:

`https://www.gnu.org/software/libc/manual/html_node/Getopt.html`

Reorganize your code

To help us with grading, please put your tree printing code in a file called printtree.c (or printtree.cpp) and printtree.h. Put your semantic analysis code in semantic.c (or semantic.cpp) and semantic.h.

Update your makefile to build these by putting their .o's in the dependency list for building c- and the g++ line.

Semantic errors

We want to generate errors that are tagged with useful line numbers. So, we will need to be sure each node is tagged with a useful line number. Remember, to do this effectively we need to grab the line number as soon as possible (in flex) and associate it with the token. This can be done nicely (portably) by passing back a struct/class for each token (as you have probably already done) in the `yylval` which has all the information about the token such as its line number, lexeme (what the user typed), constant value, even token class. (A struct/class allows you to return more than a single value in yylval.) You should avoid using global variables for token information when possible.

Scope and type checking
After checking if you should print the abstract syntax tree, you will now traverse the tree looking for typing and program structure errors. So your `main()` might look something like this:

```
numErrors=0;
numWarnings=0;

yyparse();

if (numErrors==0) {
if (printSyntaxTree) printTree(syntaxTree, NOTYPES);

scopeAndType(syntaxTree);   // semantic analysis (may have errors)

if (printAnnotatedSyntaxTree) printTree(syntaxTree, TYPES);

// code generation will go here

// report the number of errors and warnings
}

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

Your **`main`** may look quite different. The routine **`scopeAndType`** will process the tree by calling a **`treeTraverse`** routine that starts at the root node for the tree and recursively calls itself for children and siblings until it gets to the leaves. Declarations will make entries in the symbol table (see below).

Your job in writing the **`treeTraverse`** routine is to catch a variety of warnings and errors and duplicate my output for any input given. You should keep count of the number of warnings and errors and report that at the end of a run. Here is the list of errors right out of my version in printf format.

```
"ERROR(%d): '%s' is a simple variable and cannot be called.\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 index nonarray.\n"
"ERROR(%d): Cannot return an array.\n"
"ERROR(%d): Cannot use function '%s' as a variable.\n"
"ERROR(%d): Symbol '%s' is already defined at line %d.\n"
"ERROR(%d): Symbol '%s' is not defined.\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 type %s but was given %s.\n"
```

To get an easy match to the expected output, it helps if you use exactly these formats. Note that the type string that you put into the message is often something like "type int" or "unknown type". These are exactly the errors you must catch for this assignment. There are another 15 or so error messages for the next assignment.

Here are some details by node type but this list is not exhaustive. You are in control of the design as long as it duplicates my output.

- For declaration nodes, check for duplicate definitions in the symbol table. A special case happens in the case of the first compound statement in a function which will NOT open a new scope:

  `fred(int x) { int x; }`
  is a duplicate definition of x while:

  `fred(int x) { { int x; } }`
  is not.

  This is what C++ does. Try it.

- For compound statements, a newScope needs to be handled. You can put a label of your choice in the enter method for the SymbolTable object. For instance: enter("Compound Statement"); to label the scope. If you turn on debugging a good label for each scope might help.

- Assignments and operators should check that they have the proper type. Types of expressions will have to be passed up.

  Hint: it might be useful to have an undefined type that is used when variables are undefined or the type is undefined.

- Consider using an array or clever function rather than a switch or *cascading if* to know what types operators require for the operand and use the same strategy for remembering what type is returned. Some examples are:

    - > takes Integers and returns a Boolean.
    - + takes Integers and returns an Integer.
    - or takes Booleans and returns a Boolean.
    - The operators == and != take arguments that are of the same type (both Boolean or both Integer) and return a Boolean.
    - = take arguments that are of the same type and returns the type of the lhs. This means if there is an undefined operand, the lhs operand even if undefined is the type of the assignment. This is because assignment is an expression and can be used in cascaded assignment like: a = b = c = 314159265
    - ++ and -- take in Integer and immediately returns an Integer.

    See the error messages above to find an appropriate error message. Note that in the error messages above lhs means left hand side and rhs means right hand side.

    All operators work only on integers and return integers except the following. The table shows the operator, LHS, and RHS of the OP (if applicable) and the result of the operator:

OP	LHS	RHS	result
AND	b	b	b
OR	b	b	b
'='	abci	abci	aL
EQ	abci	abci	b
NOTEQ	abci	abci	b
'<'	ci	ci	b
'>'	ci	ci	b
GRTEQ	ci	ci	b
LESSEQ	ci	ci	b
'['	Abci	i	L
'*'	Abci	N/A	i

    a is array allowed, b is Boolean, c is character, i is integer, L type is the lhs operand, and A is requires an Array.

- For Ids, you have to see if the variable has been defined or not and set the type of the Id node to the type of the declaration. If the id is undefined then set the type of the id to UndefinedType (or some other indicator that the type information is missing). This is a type that when it does not match the required type of parent nodes it does not generate an error!!! (to prevent cascading errors.) Note: You may have to guard against UndefinedType in many places. It is helpful in this assignment to encapsulate redundant code as functions.

- Note that for this assignment each undefined reference must generate an error message. We'll fix this later.

- For Ids you can have arrays that are indexed. Once they are indexed, their type becomes nonarray. That is the type of the '[' is the type of the lhs. Check for indexing of nonarrays and using unindexed arrays where they can't be used.

- void is the type of a function that returns no value. It is possible for a type to be of type void in an error message.

- Ids that are arrays can also be prefixed with '*' operator. That lets you get at the size of the array. Every array stores not only the values in the array but its size. This means that an array of size 10 (e.g. frog[10]) needs 11 spaces allocated to it. More about this in the code generation assignment. For this assignment, you only need to know that '*' works on arrays and returns an int.

- For the return statement, you must make sure that the user does not try to return a whole array.

- Finally, after processing the whole tree, **`main`** should be in the global symbol table. If the procedure main is not defined, then you must print out an error.

Symbol table
For those of who are using C++, a useful C++ symbol table object can be found in the file `symbolTable.tar` (posted with this assignment.)

This version uses STL (Standard Template Library) and is cheap. Feel free to augment it or build your own. Though it is written with std::string type as the argument type in many places, C++ will coerce char * to std::string if you supply a char *.

It provides a symbol table object with insert and lookup methods for symbols and a pointer (you can use the pointer to point to a `TreeNode`. It also has **enter** and **leave** methods of managing the scope stack. Read the **`symbolTable.h`** for more information on how to use it. You might want to just play with it to see how it works before you put it into your compiler (see test routines commented out in the supplied code.). Inserting a symbol that is already defined returns false, success is true. Looking up a symbol that is not there will return a NULL pointer.

One feature of the symbol table is the debug method and the two DEBUG flags. At construction time the SymbolTable object is in nondebugging mode. But by setting the flags with the debug method you can get the object to spew out info. You might consider starting out by printing the symbol table on exit from a scope using the debug flag.

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

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

creates the symbol table. To print the symbol table:

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

will print each void * in the symbol table using your supplied function:

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

Submission
You will submit a single uncompressed tar file through Canvas. You can submit as many times as you like. The LAST file you submit BEFORE the deadline will be the one graded.

We will use extensive tests, so thoroughly test your program with inputs of your own.

If you have tests you really think are important or just cool please send them to me and I will consider adding them to the test suite.