CS445 Introduction to Compilers

Assignment 3 (Symbol Table and Type Checking) DUE: Sat Nov 2 at 5 PM PST

350 points

In this assignment we will do semantic analysis and semantic error generation. We will also add some more command line options.

Do not be fooled. This is a nontrivial homework. Do not put off this assignment. It is very complicated.

A New Compiler Option

This time we add **two new options**. The first option is **-p** which prints the syntax tree. That is, it prints the syntax tree we did for assignment 2.

The second option is **-s** and turns on symbol table tracing. See the section on the symbol table below.

It 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. **Note: the options need not come before the file name.** (HINT: Getopt code that is in this **getopt.cpp** file and is free for your use. Check out: **a getopt example** that is a silly example of how to do a variety of things with getopt. This will make your life easier and let you focus on the compiler stuff. I also have the **getopt man page** available.)

Semantic Errors

We want to generate errors that are tagged with useful line numbers. So we will need to tag each node in the abstract syntax tree with a useful line number. 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 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 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 mine:

```
    // compile the code
    int numErrors;
    int numWarnings;

    yyparse();

    if (printSyntaxTree) printTree(syntaxTree);

    if (numErrors==0) scopeAndType(syntaxTree);

    // 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 are the errors right out of my version Here is the **list of error messages** sorted by type of error message.

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 IO routines output, outputb, input, inputb, outnl, you will need to add prototype equivalents into the abstract syntax tree after the tree print routine. They have the C- prototypes:

  ```
  void output(int)
  void outputb(bool)
  void outputc(char)
  int input()
  bool inputb()
  char inputc()
  void outnl()
  ```

  Errors that need to address the defining location for the I/O routines can report the linenumber -1.
- The linenumber for the WARNING of missing return statement is the line number of the function declaration.
- For **declaration nodes** check for duplicate definitions in the symbol table.
- For **compound statements** a newScope needs to be handled.
- **While** and **If** should check they have boolean tests.
- **Foreach** should check that the types match the description in the c-Grammar semantics section.
- **Assignments** and **operators** should check that they have the proper type. Types of expressions will have to be passed up. Beware of **Cascading Errors** as discussed in class.
- For **return** check for return type matches. Note that it is easy to check that the return is of the same type. It is hard to check that a return is made out of every exit so we will not issue a error for that. However, **I have added a warning** for a much simplified check. Note that routines that do not return a value use the return without an argument. A routine whose code falls out the bottom will perform a return of the right type by default. (More on the value returned in the next assignment.)

- For **break** check that the break is inside a loop.
- 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 that type. This is because assignment is an expression and can be used in cascaded assignment like: a = b = c = 314159265
    - ++ takes in Integer and returns and 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.
- 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 name indicating 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. Note: You may have to guard against UndefinedType in many places.
- 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. Check for indexing of nonarrays and using unindexed arrays where they can't be used.
- 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.
- The **function call** is the trickiest of all. You must use the symbol table lookup to find the definition node and from there compare the types in the parameter list with the types of the arguments given. This means you must compare the type of the supplied argument to the type of the formal parameters for a type match. Also if you run out of parameter list items before you run out of argument list items you need to issue an error. Same for running out of arguments. See the error messages for an appropriate error.

Symbol Table

Here is a useful C++ symbol table object you can use:

symtab.cpp
symtab.h

Here is a brute-force translation to C of the the above files (for a single symbol table).

symtab.c
symtab.h

and a tar file of all symtab files:

symtab tests
symtab expected output
tar of all symtab stuff

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 symtab.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).

One feature of the symbol table is the debug method and the two DEBUG flags. At construction time the SymTab object is in nondebugging mode. But by setting the flags with the debug method you can get the object to spew out info. Use the **-s** flag to set the debug flags to DEBUG_TABLE. This will announce entry into every scope and prints the symbol table on exit from a scope along with the scope names.

Finally the constructor takes a print routine of the type void blah(void *). So if you define something to print a node given a TreeNode * then you can supply that name to the constructor to print out your symbol table stack. That way I don't have to know what you TreeNode looks like internally. For instance in my code:

```
    symtab = new SymTab(nodePrint);
```

creates the symbol table and nodePrint is defined as in this prototype:

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

Example Runs

Here is **basicAll.c-** and **basicAll.out**. Here is a **tar of a bunch of tests** **These are under construction**

Submission

Homework will be submitted as an **uncompressed** tar file to the Homework submission page. You can submit as many times as you like. **The LAST file you submit BEFORE the deadline will be the one graded.** For all submissions you will receive email showing how your file performed on the pre-grade tests. The grading program will use more extensive tests and those results will be mailed to when they are run.

As always, 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.

Robert Heckendorn   Up One Level   Last updated: Mar 5, 2007 23:23