CST320 Lab 3

Symbol Table

**Purpose**

This lab serves the following purposes:

1. Allows you to create and test your symbol table code

**This lab should be completed in the same working directory you used for Lab 1.** I suggest you unpack the tar file in a different directory so that you do not overwrite any files you edited in Lab 1. If you mistakenly overwrite a file, don’t panic, it can be recovered from git.

NOTE: I will take points off if your git repository does not include both Lab 1 and Lab 3.

**Symbol Tables**

A compiler stores symbol information (information about identifiers) in a table. In this lab, you will write the symbol table code. You can use the data structure of your choice, but I highly recommend a hash table because hash tables have the performance characteristics that are appropriate for our use. You can use the STL[[1]](#footnote-1) (or some other ready-made data structure) or create your own. In previous years, people who used the STL had the best success.

Languages that have nested scopes must have some way to keep track of multiple different symbols with the same name. For example, in the following code snippet, there are two variables by the name “var”, and the symbol table definitions must be distinct.

{

int var;

{

double var;

}

}

To do this, your symbol table should have two methods: IncreaseScope() and DecreaseScope(). IncreaseScope() should create a new, empty, symbol table and place it on a stack of symbol tables. This new table becomes the default table and new symbols will be added to this table. DecreaseScope() will pop the top table off the stack and the next one down will become the default table.

Note: when I talk about a stack of tables, it is NOT required that you implement a stack. However, you must implement the behavior that a stack would provide. The implementation you choose is entirely up to you. Some of the access patterns for the symbol table are inconsistent with stack access patterns, so an actual stack is probably not the best choice of implementation.

Note 2: When you DecreaseScope(), you should NOT free the memory for the symbols stored in the table you removed from the stack. In general, nodes in the AST will still have references to the symbols for use in later stages of compilation. If you were to free all the memory for these symbols your AST would have invalid references.

Your symbol table must have methods to insert and lookup symbols. Inserted symbols must always go into the current default table (the one at the top of the stack). Lookup should first search the top-of-stack table, but if the symbol is not found there, it must continue to search the other tables on the stack until either the symbol is found or the last table is searched (representing the outer most scope).

Note: Because of a combination of the implementation of the parser generator we will use starting next week and a limitation of the C++ language, you must store pointers to the symbols in your table not the actual symbol objects. More on this next week, but be sure to implement your table as storing pointers to symbols or you will have to re-implement your table next week.

You will also need a lookup method that only searches the default table (the inner most scope).

A symbol tables holds symbols. For now, a symbol must have a text field for storing the symbol name (the identifier). Your symbol must also have a ToString() method that returns a string representation of the symbol. The test suite defines the required format of your string (also described later in this document). You must use the following prototype:

virtual std::string ToString();

We will use this method to print symbols. We could have used an ostream << operator for doing this, but later on we will need ToString methods, so we might as well start out with them now.

As a final requirement for your symbols, each symbol must have a unique integer identifier. This requirement is not necessary for the compiler, but it will allow us to test your symbol table. You may also find it useful when debugging later stages of your compiler. The following code snippet illustrates one method to implement this requirement.

class cSymbol

{

public:

cSymbol()

{

mSequence = ++symbolCount;

}

protected:

int mSequence;

static int symbolCount = 0;

}

Note: the tar file supplied with this lab contains a cSymbol.h file that contains a definition for a symbol that meets these requirements. Feel free to use that implementation.

**Putting it together:**

Your scanner will control the scope of your symbol table, and it will insert new symbols when they are found. The lang.l in the.tar file gives a hint as to how to do this. The idea is that every time a '{' is found, the scope should be increased. Every time a '}' is found, the scope should be decreased.

In addition, every time an identifier is found, you should search the innermost scope and if the symbol is not found there, insert it. If the symbol is already in the innermost scope, simply return that symbol. The symbol value is “returned” through the global variable yylval. Sample code in the lang.l in the tar file gives a hint as to how to do this.

Finally, you must create the root symbol table (the global scope). I’d suggest you do it in main() as a global variable. If you choose an implementation other than a global variable, that is fine, but main() and yylex() need access to the symbol table. In later labs, the parser will also need access to it. Finally, main() must have access to the global variable yylval so that it can print the values of symbols that are found by the scanner. The main.cpp in the.tar file illustrates how to do this.

At this point, you can test your symbol table using the regression tests. The output is an XML file. The cSymbol::ToString method must return strings with a format consistent with the following example:

<sym id="1" name="a"/>

The quotations are required as is the “/”, and the attributes id and name must occur in this order.

There are three tests that contain progressively more difficult situations.

NOTE: the output of the “compiler” is an XML file. The utility xmllint is used to pretty-print the XML file to make it easier to diff your output with the correct output. The test scripts will take care of this for you.

**Turning it in**

I want your code to output your name so that as I’m grading I can see whose code it is by running your program. To do this, you can add a line such as the following to the **beginning** of main(). This should not affect the output used for testing.

std::cout << "Philip Howard" << std::endl;

What I need from you is the location of your git repository and the tag you attached to the version you want me to grade. If you do not supply me with a tag, I will try “Lab2”. If that does not work, I will pull the most recent version from your repository. It is far safer to supply me with a label, so don’t forget.

Remember to document your code using the department’s coding standard or some other reasonable standard. I will be looking at your code, and if I can’t understand it, I will mark you down.

1. Note: I still use the old name “STL” to refer to the container classes that are now part of the std namespace. I’m referring to things like <list> <map>, etc. [↑](#footnote-ref-1)