Lab 5a

Semantic Errors

**Purpose:**

In this lab, you will add semantic error checking to your compiler.

Semantic processing has been split into two pieces: this week you will add code that needs to be done during the parsing process. Next will, you will add code that can be done in a second pass using a visitor pattern.

**Preliminaries:**

There should be a function defined in lang.y that gets called any time a semantic error is detected. You need to add a prototype to an include file so that your AST classes can call the function when necessary. The code is as follows:

// Function that gets called when a semantic error happens

void SemanticError(std::string error)

{

std::cout << "ERROR: " << error << " on line "

<< yylineno << "\n";

yynerrs++;

}

Some semantic error checks can be included in the constructors and other code you wrote for Lab 4. In other cases, you may need to write additional routines. Some of these could be called from the code snippets in lang.y, but be sure to keep lang.y as clean as possible.

**Declarations**

In order to do semantic processing, each symbol needs to know what kind of symbol it is. In particular, we need to know where the symbol was declared. Add a field to cSymbol that is a cDeclNode\* with a getter and setter. Every time you add a symbol to the symbol table, you need to set its declaration. This is typically done in the constructor for the decl with a line such as:

id->SetDecl(this);

NOTE: This violates the “all children must be in m\_children” principle from the previous lab. The reason for doing this is so that we don’t have circular children: a decl contains a cSymbol that contains a decl that….

To make sure the decl is set correctly for each symbol, change the AttributesToString in cSymbol to print the id of the declaration attached to the symbol. The number should be the same as the symbol’s number, but this will allow us to check that the declarations are set appropriately.

The cSymbol.h in the tar file has these changes. You **MUST USE** the implementation of cSymbol::AttributesToString contained in the tar file.

Note that char, int, float are symbols, so they need a declaration attached to them. I would suggest you create a cBaseTypeNode class that contains the following attributes: name, size, isFloat. This class should inherit from cDeclNode. When you insert the symbols for char, int, float into the symbol table, attach a cBaseTypeNode to them as follows:

type = new cSymbol("char");

type->SetDecl(new cBaseTypeNode("char", 1, false));

Insert(type);

You also need to add the following pure virtual functions to cDeclNode:

virtual cSymbol\* GetName() = 0;

virtual cDeclNode \*GetType() = 0;

You will then have to override these function in every class that inherits from cDeclNode. GetName() returns the symbol for the name of the decl, GetType() returns the type of the item. For example, for a cVarDecl, GetType() should return the decl associated with the TYPE\_ID symbol.

Having made these changes, you no longer need the isType field in cSymbol. Instead, you can determine if a symbol is a type using something along the following:

if (yylval.symbol->GetDecl()->IsType()) return TYPE\_ID;

test0a is intended to test proper behavior of the features described in this section, however, it does not fully check the GetType() behavior. Be sure to check this on your own or you may have trouble later.

test0 is a concatenation of all tests from Lab 4. It is intended to check that your work this week doesn’t break anything you had working last week. The output is not the concatenation of last week’s output because it includes the decl information attached to each symbol.

**Suggestions:**

You may find it convenient to add the following virtual functions to cDeclNode:

virtual bool IsArray() { return false; }

virtual bool IsStruct() { return false; }

virtual bool IsType() { return false; }

virtual bool IsFunc() { return false; }

virtual bool IsVar() { return false; }

virtual bool IsFloat() { return false; }

virtual bool IsInt() { return false; }

virtual bool IsChar() { return false; }

You can then override the functions to return true where appropriate. For example, cArrayDeclNode could override IsArray() and IsType() to return true, but it would not need to override the others. By placing these functions in the base class, you can always figure out what kind of decl you have.

**Semantic Processing:**

Your semantic processing must be such that a semantic error at some point in the source code that you are compiling should not cause a seg-fault in later processing.

You must handle the following errors.

Note: the test numbers roughly follow the numbering of this list.

1. Duplicate variable definitions: If an identifier is defined multiple times in the same scope, the following error should be emitted:

Symbol <id> already defined in current scope

1. Function declarations and definitions must match in terms of return type and parameter types. In addition, every instance of a function declaration, definition, or call must refer to the same symbol. If there are violations, one or more of the following errors should be emitted:

<func> previously declared as other than a function

<func> previously declared with different return type

<func> previously declared with a different number of parameters

<func> previously declared with different parameters

1. The identifier for a function call must be defined as a function. There must be only one definition, but multiple declarations are allowed. If there are violations, the following error should be emitted.

<func> already has a definition

1. If there are multiple declarations of the same function, the symbols for the function name must be the same. The decl associated with that symbol must be the most complete decl. In other words, if you have a function prototype followed by a function definition followed by a function prototype (all for the same function), the same symbol ID must be used for all instances of that function name, and the decl associated with that symbol must include the full definition of the function.