Lab 5

Semantic Errors

**Purpose:**

In this lab, you will add semantic more semantic processing to your compiler.

**Preliminaries:**

The semantic processing in this lab will be done using a Visitor class. In order to report line numbers (since the visitor will run after ALL scanning and parsing is completed), we need to attach a line number to AST nodes. This can be done by adding the following protected data member to cAstNode:

int m\_lineNumber;

This data member can be initialized in the constructor as follows:

m\_lineNumber = yylineno;

To prevent cascading errors, we will also want to be able to mark nodes as having an error. Add the following protected data member to cAstNode along with a getter and setter. The value should be initialized to false in the constructor, and the setter only needs to set the value, never clear the value.

bool m\_hasSemanticError;

The tar file contains a cAstNode.h with these changes.

You need to make sure the cVisitor base class has a Visit function for Every node type. I would suggest keeping them in alphabetic order so that it is easy to compare the list of functions against a directory listing of your development directory. Be sure to declare the functions in the .h file and give a default implementation in the .cpp file. Follow the examples already in those files.

Create your visitor class as a sub-class of cVisitor. Your class should have an instance variable that keeps track of the number of errors found. There should be a getter to retrieve this value. You also need a function for printing errors. The error function is analogous to the SemanticError function we added to lang.y in the last lab. The following implementation will do (you may need to change the names of a few things)

void SemanticError(cAstNode \*node, string msg)

{

m\_numErrors++;

node->SetHasError();

std::cout << "ERROR: " << msg <<

" on line " << node->LineNumber() << "\n";

}

You also need a VisitAllNodes method. The following will work:

void VisitAllNodes(cAstNode \*node)

{

node->Visit(this);

}

Finally, you need to call your visitor from main(). The main.cpp included in the tar file does this.

**Rules on type casting**

You are not allowed to use any C-style type-casts. That is, the following is not allowed:

myNodePtr = (cIntExprNode\*)otherNodePtr;

You ARE allowed to use static\_cast and dynamic\_cast with the following restrictions:

1. No casting of any kind is allowed in your semantic visitor.
2. Nodes can use casting to retrieve values from the m\_children vector. For example, the following implementation is acceptable for the GetName method in cVarDeclNode:

cSymbol \*GetName()

{

return static\_cast<cSymbol\*>(GetChild(1));

}

Note that the index specified in GetChild is hard-coded. This is acceptable because, given the constructor, every cVarDeclNode will have two children and the second one will always be the symbol for the name of the variable. Because of this, I consider the static\_cast safe – a dynamic\_cast is not required.

Another Note: if you are getting strange behavior in your compiler, you might want to use dynamic\_cast instead of static\_cast. With dynamic\_cast if you attempt an illegal cast, you will get a nullptr and then a segfault when you dereference it. If you use static\_cast to attempt an illegal cast, the conversion will be performed, but the data at the address pointed to will not have valid data. This is a much more difficult error to track down.

1. You can use static\_cast provided you know ahead of time that it is safe (see example above). If you use dynamic\_cast, you must check the returned value against nullptr to make sure the cast was successful, and provide appropriate error handling on failure. Note: often the failure means a catastrophic bug in your code. “Appropriate error handling” in this case is to print an error message and exit.

As a hint: if you are using casting outside your GetChild() getters, that implies a potential design flaw in your code. I will check your use of casting when I grade these labs. If I find ANY C-style casting, I will take up to **10 points off** for each occurrence. If your C++ style casting reveals design flaws, I will also take points off for that.

**Types**

We need to be able to determine the type of every expression. To do this, add the following pure virtual function the cExprNode:

virtual cDeclNode \*GetType() = 0;

This function needs to return a pointer to the declaration that identifies the type of the expression. The declaration must be a node that inherits from cDeclNode. For this to work, we need declarations for the base types: char, int, float. You should have added these in the last lab.

The classes that inherit from cExprNode should return the type as follows:

cIntExprNode Lookup the symbol for “int” or “char” (based on the value of the node) and return the decl attached to that symbol.

cFloatExprNode Lookup the symbol for “float” and return the decl attached to that symbol.

cVarExprNode Return the decl attached to the symbol for the type of the variable.

cFuncExprNode Return the decl attached to the symbol for the return type of the function.

cBinaryExprNode Compare the types of the two operands and return the correct type based on the normal C promotion rules.

cBaseTypeNode Should return this

**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 may need to check the HasError() of the children and skip some processing if the children have errors.

You must handle the following errors.

1. Undefined variable: If a variable is referenced that is not defined, the following error should be emitted. Note, you should be able to check this by checking to see if the symbol has a decl attached to it.

Symbol <id> not defined

1. The type of the lval in an assignment must be compatible with the type of the expression. The C compatibility rules apply. Note: many C compilers issue warnings when these rules are violated. We will treat these violations as errors. Function parameters are passed by value, so these rules also apply to the actual parameters in a function call. For integer constants, values from -128..127 shall be considered char. All other values shall be considered int. Structs and arrays shall use name equivalence for determining type compatibility. Violations should emit the following error message:

Cannot assign <type> to <type>

1. For an array dereference, the item being dereferenced must be an array. For example:

int ii;

ii[5] = 7;

should emit the following error:

<id> is not an array

1. The index to an array must be an integer (int or char). If not, the following error must be emitted:

Index of <var> is not an int

1. The identifier for a function call must be defined as a function. If not, the following error should be emitted. Note: this error means the identifier was defined as something other than a function.

<id> is not a function

1. The identifier for a function call must be defined as a function. If there is no declaration for the identifier, the first error should be emitted. If the function is declared but not defined, the second error should be emitted:

Function <id> is not declared

Function <id> is not fully defined

1. Function calls must have the correct number of parameters. If not, the following error should be emitted:

<id> called with wrong number of arguments

1. Function calls must have the correct type of parameters. If not, the following error should be emitted:

<id> called with incompatible argument

**Note on test files:**

I only included one test file with this lab. The purpose of that test file is to ensure that your SemanticError function is formatting errors correctly. You will have to generate your own test cases for this lab.