**Analysis**

Description of semantic rules:

In my compiler, I don’t allow mixing ints and floats, so if you add an int to a float, it will give an error.

For each expression, arithmetic expression, relational expression, and assignment statement, I check the types of each member, to make sure that there is no type mismatch.

The following semantic actions have been added to my grammar:

* createProgTable = 0,
* startCollecting = 1,
* stopCollecting = 2,
* createSemanticClassAndTable = 3,
* calculateClassSize = 4,
* scopeIn = 5,
* scopeOut = 6,
* createSemanticVariable = 7,
* createSemanticFunctionAndTable = 8,
* processAssignment = 9,
* processVariableUse = 10,
* processExpression = 11,
* processTerm = 12,
* processIdNestListIdThenIndiceListOrAParams = 13,
* processNum = 14,
* processIndiceList = 15,
* processArithExpr = 16,
* processRelExpr = 17,

Augmented grammar:

I augmented my rules file (which is derived from the grammar). It can be found in GrammarFiles/rules.txt.

Each semantic rule is preceded by an exclamation mark (!), for example “!startCollecting”, so that the parser knows that it’s a semantic action and not a regular symbol in the grammar.

**Design/Implementation**

Solution structure:

SemanticAction: inherits from GSymbol, which is the parent class for all grammar symbols. This allows me to put objects of this class into the parsingStack, which is std::stack<GSymbol\*>. It simply contains the information about what kind of action needs to be executed. Inside the parse() method, each time a semantic action is popped from the parsing stack, it gets executed.

SemanticRecord: a parent class for 3 different types of semantic records: SemanticVariable, SemanticFunction and SemanticClass. Each SemanticRecord has the following member variables:

* SemanticRecordType semanticType;
* std::string identifier;
* bool declared;
* SemanticStructure semanticStructure;
* int arrayDimension;
* int address;

Semantic Variable: in addition to inherited members, has VariableKind kind; (normal or parameter)

SemanticFunction: in addition to inherited members, has:

* std::list<SemanticVariable\*>\* params;
* SymbolTable\* localSymbolTable;

SemanticClass: in addition to inherited members, has: SymbolTable\* localSymbolTable;

SymbolTable: a class for the symbol tables. Member variables:

* SymbolTable\* parent;
* std::list<SemanticRecord\*>\* records;
* std::string tableName;

Member functions: create, search, insert, delete and print.

**Error reporting**

I was unable to make expressions work (assignment, arithmetic, relational), therefore I wasn’t able to have my test cases actually output the proper errors. However, here are some of the code lines that output full error descriptions and locations. They just never gets triggered in my program.

Logger::getLogger()->log(Logger::SEMANTIC\_ERROR, "Type mismatch: right side of assign statement does not match the type of left side, on line " + idTerm->getPosition().first);

Logger::getLogger()->log(Logger::SEMANTIC\_ERROR, "\nIdentifier '"+ functionIdToken->getValue() + " at line " + std::to\_string(functionIdToken->getPosition().first) + "' is not defined in current scope (" + currentScope->getTableName() + ")");

Logger::getLogger()->log(Logger::SEMANTIC\_ERROR, "\nIdentifier " + term->getValue() + " at line " + std::to\_string(term->getPosition().first) + " is not defined in the current scope(" + currentScope->getTableName() + ")");

**Tools and techniques used**

* <http://hackingoff.com/>: an online tool that takes a LL1 grammar as an input and outputs FIRST set, FOLLOW set, rules and parsing table.
* Git, GitHub for version control
* VisualStudio for creating the project solution and debugging
* Semantic analysis technique described in class (table-driven)

**Test cases**

* All test cases can be found at **TestFiles/Semantic**

1. Variable used without being declared
2. Function used without being declared
3. Class used without being declared
4. Scope
   1. Same variable defined twice in same scope (bad)
   2. Same variable defined in one scope, and then in another scope (good)
   3. Variable used outside its scope (bad)
5. Circular class dependencies
6. Full valid program from Assignment3 handout
7. Full valid program from Assignment2 handout
8. Type
   1. Assign float value to int variable
   2. Assign int value to class variable
   3. Assign class value to float variable
   4. Add int to float