**Semantic Rules**

// Rules for building the symbol tables

#createGlobalTable# creates the global symbol table

#endGlobalTable# jumps out of the scope of the global table

#createClassEntryAndTable# creates an entry in the global table and creates the class table

#endClassEntryAndTable# jumps out of the class table scope

#createProgramTable# creates the program table

#endProgramTable# jumps out of the program scope

#createVariableEntry# create a variable record in the current scope

#addParameter# adds a parameter to the current function

#createFuncEntryAndTable# creates a function record in the current scope and goes to the scope

#endFuncEntryAndTable# jumps out of the function scope

#startFuncDef# tells the next semantic action that we are at the start of a function

#checkTypeGlobal# checks to see if the identifier given is in the global scope

#checkCircular# checks to see if the class var defined in a class is circularly defined

#storeId# stores the id/lexeme of the token into the semantic stack

#storeType# stores the type/lexeme of the token into the semantic stack

#storeArraySize# stores the arraysize/lexeme of the token into the semantic stack

// Rules for building expressions

#addNumericExprFragment# adds a numeric value to our expression building

#operatorExprFragment# adds a operator value to our expression building

#addSignExprFragment# adds a sign +/- value to our expression building

#checkExpr# checks to see if our expression is valid

#pushExpr# pushes an expression onto the semantic stack

#popExpr# pops an expression onto the semantic stack

// Rules for building variables

#pushVar# pushes a variable onto the semantic stack

#popVar# pops a variable onto the semantic stack

#addToVar# adds a variable fragment to the current variable ex a.bc.de, will add a, bc, de fragment parts

#setFunc# tells that this identifier is now a function call

// Rules for building statements

#pushStatement# pushes a statement onto the semantic stack

#popStatement# pops a statement onto the semantic stack

#assignmentStatement# updates the current statement to be an assignment statement

#forStatement# updates the current statement to be an for statement

#ifelseStatement# updates the current statement to be an if/else statement

#getStatement# updates the current statement to be a get statement

#putStatment# updates the current statement to be a put statement

#returnStatment# updates the current statement to be a return statement

Not all rules are implemented, some are just stubs for the next assignment, and some may be changed/removed in the next part

Grammar with actions file -> ***See grammar\_with\_sdt.txt file***

Symbol Table file -> ***See symbolTables.txt file***

Semantic errors file -> ***See semanticErrors.txt file***

**Design/Implementation**

**Overall structure and reasoning on implementation**

I have a **SymbolTable** class which is essentially a map of names to **SymbolTableRecord.**

A **SymbolTableRecord** is a data structure holding data such as:

1. Name of the identifier
2. The kind of identifier (var,param,function,class)
3. The type of identifier (types:int/float/class)
4. The structure of the identifier (simple/array), and any array data with dimensions
5. Function data if needed such as return type, and parameters with types
6. the scope which is a pointer to the linked symbol table if used
7. The defined location of this identifier (used in error handling)
8. The address which will be used in the next assignment
9. whether or not this record has been properly declared
10. Attribute structure which holds necessary data for when we are building vars/expression/statements. This data gets pushed and popped on the semantic stack. This is used to help build the record

Attribute Migration happens when we pop vars/expressions/statements.

The data gets passed to the next attribute on top of the stack, either up or sideways depending on the context

A lot of the record is built by a bunch of sub structures; you can see most of these structures in the **SymbolTableData.h** file

**Error Handling**

**Two phase parser**

First phase builds the symbol table and turns off error reporting. However if it finds an error during this phase, it will skip those errors and move on. This allows for classes/functions to be used even if they are defined after.

The second phase turns on the error reporting and will give the reason of the error, the value/token and on which line the error happened.

Some example of errors:

* if a identifier is not defined
* if identifiers have been defined multiple times,
* if there is circular declarations in classes,
* if an variable has been used before being defined
* if a variable is being used as a function and it isn’t supposed to be a function and vice versa
* if a variable is an array but is being used as an array and vice versa
* if an array is being used with the wrong dimensions
* if a variable is being used as an object but it isn’t an object

**Some techniques**

To output the symbol tables in order, I used a breadth first traversal. This allows all top level tables to be outputted first, then the next level, and so on

To find the circular dependencies, I used a depth first Traversal. As I visited a symbol table, I marked that I visited it. If I visit a table that has already been marked, then there is a circular dependency.

The semantic actions are implemented in the grammar as a #name# pattern. When it sees this pattern it will create a sub production rule. So each production has two rules. One without the semantic actions and one with the semantic actions. This was to avoid writing checking code in the first/follow set code and parsing table generation code. This allows the productions to be generated as if there were no actions, and it by passes all the parser generation code. The semantic productions rules only get used when we start doing the parsing.

The semantic actions are implemented in the **SemanticActions.cpp** file as functions with the same name. These are stored in function pointer map.

The push/pop var/expr/statement allows for easy nesting. For example, an assignment statement of

**j = j \* 2;**

would

* push an assignment statement
* push variable j
* pop j, then when popping it would check to see if j is defined or if there is any errors
* push an expression
* push variable j
* pop j, again do error checking
* add the \* operator to the current expression
* add 2 to the current expression
* pop the expression, and add the expression as the expression to the assignment statement
* pop the statement, then carry on to the next piece of code

**Use of tools**

**Tools / Libraries / Techniques**

* I didn’t use any external tools/libs/dlls to write any code. Only used the internal libs provided with C++.
* Used Bitbucket/Git for version control.
* Used STL containers such as <vector> and <unordered\_map> to hold data over simple arrays.
* Used C++11 smart pointers for easier pointer management.
* Used the unit testing framework provided inside visual studio to create unit tests.