**Semantics of the language:**

**Do we allow in an expression or an assignment statement a mixture of integer and float variables?**

No, Each expression has to be of one type.

Examples:

1 + x + y.z \* a.b() must all evaluate to an integer. This is a valid expression

1 + 2.1 is not a valid expression. It mixes ints and floats

**How are parameters passed to procedures (by value, by reference, etc.)?**

Passed by value

**Is there a limit on the number of function parameters?**

This depends on how many registers we have. We have 16 registers, but only 13 are usable. Since 1 is r0 is can’t be used, and r14,r15 are used for return and jump registers.

**Is function overloading allowed, i.e. functions with the same name but with varied number/type of**

**parameters in the same scope?**

No, function overloading is not allowed. The implementation only allows for a function to have one signature.

**Are recursive function calls allowed, or can two instances of the same function be active at the same time?**

No. Everything is allocated statically.

*Semantic Rules*

// Rules for creating 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

#addFuncDefParameter# 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

#startFuncCall# 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

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

#assignmentStatementEnd#

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

#forStatementEnd#

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

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

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

#getStatementEnd#

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

#putStatementEnd#

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

#returnStatementEnd#

Updated Grammar can be seen in the **grammar\_with\_sdt.txt**

Compiler output such as errors or important info can be seen **compilerOutput.txt**

Compiler output is also being output to the console.

**There are output files for every part of the compiler. Outputting is done for everything.**

Output code generation file is in **sourcefile.m**

To see example input/output files see the **/Tests/FullDemo\_TestFiles/…** directory.This will show the output of semantic/syntax errors and will have the code generation for some test input files

The overall structure of the solution goes by building each individual parts of our language. Parts such as statements, expressions, function definitions all get created by the semantic actions.

Expressions are built by reading in the individual parts by the semantic actions and are turned into a postfix expression. When the expression is done being built, it is converted to an expression tree. This expression tree is then type checked by comparing each sub tree has the same type. If they are all the same type, then the expression is that type. If there is a mismatch in type, then the expression is typed mismatched.

Statements are also built by using the semantic actions. There are semantic actions for each type of statement telling what specifically that statement is. For example in our semantic actions we have a action such as #ifelseStatementStart# and a corresponding #ifelseStatementEnd#. When we hit a start statement, this tells the attribute migrater to migrate the next expression to the if condition, and any following statement as if/else statements. This technique is applied for all types of statements.

We use the semantic stack to do attribute migration. For example if we have the assignment statement

x = 32;

this will do the following semantic actions

pushStatement

assignmentStatementStart

pushVar

popVar (check for variable errors) then this will migrate it to the assignment statement’s lhs variable

pushExpression

addNumberToExpr

popExpr(type checking) then this will migrate it to the assignment statement’s rhs expression

assignmentStatementEnd

popStatement

**What is implemented for code generation**

Memory allocation for int/floats

Memory allocation for vars/objects

Memory allocation for arrays and array of objects

Loop statements with jumps

If/else statement with jumps

Input output

Expressions with arithmetic and relational operator

Function declarations jump on call/return value

Parameter passing

Array offset calculation with n dimensional arrays

**What is not implemented**

Offsets for objects and method calls

Again you can see the outputs for all of this in the **/Tests/FullDemo\_TestFiles outputs/codegen/…**

**Some techniques**

All of our statements and expressions are done using intermediate representations. This is done to simply our generation by abstracting way statements and expressions. Also Since we need to generate function definitions before the actual code, we need to generate the code before the program code. If we didn’t have representations, then this would be a lot trickier to do.

The code for turning an expression into instructions is done well and abstracted. Basically you create an Expression Instruction, give it an expression, then it gives you back the output register of the expression’s value and you can save it or use it for something else.

**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 the unit testing framework provided inside visual studio to create unit tests.
* The moon documentation