Alex Sachkov
Hugo Cheng
Fedora Furtado
Sara Beatriz Martinez

Assignment #4

1. Global Variables (Scalar and Vector)

Decaf support for Array types and global variables was accomplished using LLVM GlobalVariable class.

FieldDecl Codegen consisted of checking for a pre-existing global var before initializing a new one. If the array size is 0 then throw an error, if array size is -1 then declare a simple global var otherwise declare a new global with a new array type and initializer. Return the resulting global var.

Codegen for ArrayLocExprAST, used for referencing array values, was done by fetching the global var associated with the Name, generating llvm for array location & index and returning the value, thereby that loading values by de-referencing an array location.

2. Control-Flow and Loops

If Statements

There are two possible cases: If with and without else block. For if with else, I created 4 blocks: one for the entry that does expression evaluation, one for the default behavior on expression success, one for the else condition, and one for the branch merging. The entry block evaluates the given expression and branches to either the default or the else conditions. Each condition block contains the code for its execution, and a conditionless branch to the merging block. The merging block contains all code located after the if statement. For the case without else block, the condition evaluator branches to the merge block instead in case the condition fails.

While Loop

To implement while loop, three blocks are created: one for the expression evaluation, one for the loop body, and one for the exit. Upon entry, the loop block evaluates the expression, and if true, it branches to the body block. Otherwise it branches to the exit block. The body block executes all the code inside the while statement, and branches back to the loop entry. Both loop entry and exit are entered into the symbol table for loop control statements.

For Loop

The for loop implementation consists of 4 block: entry block, body block, next block, and exit block. The entry block evaluates the expression and branches to the body block on success. Otherwise, it branches to the exit block. The body block contains all its inner code and branches to the next block which executes the for loop expressions. Afterwards, the next block branches back to the loop block. The next block and the exit blocks are entered into the symbol table for loop control statements.

Break/Continue

Continue statement looks up the most inner loop start block and branches to it. Break statement branches to the most inner loop exit statement.

3. Semantic Checks
    a. A method called main has to exist in the Decaf program.
        i. Check decafstatementlist's list of statements to see if main has been declared as one of the functions, throw error if not.
    b. Find all cases where there is a type mismatch between the definition of the type of a variable and a value assigned to that variable. e.g. bool x; x = 10; is an example of a type mismatch.
        i. Proposed Implementation: Check binary expressions (BinaryExprAST) to see if the left value and right values are of the same type using getType(). This includes integer, boolean and comparison expressions. Also check unary expressions (UnaryExprAST) of negation and minus, along with mismatches in variable assignment (AssignVarAST).
        ii. Implementation: Same as proposed implementation however it did not initially consider type mismatches in method calls (MethodCallAST) or return statements (ReturnStmtAST), if function was to return void, int, or bool etc.

    c. Find all cases where an expression is well-formed, where binary and unary operators are distinguished from relational and equality operators. e.g. true +false is an example of a mismatch but true != true is not a mismatch.
        i. Done in checking binary expressions previously.
    d. Check that all variables are defined in the proper scope before they are used as an lvalue or rvalue in a Decaf program (see below for hints on how to do this).
        i. Done in VariableExprAST, MethodCallAST, AssignVarAST and AssignArrayLocAST - check to see if the variable, function or array is in the symbol table/declared.
    e. Check that the return statement in a method matches the return type in the method definition. e.g. bool foo() { return(10); } is an example of a mismatch.
        i. Done from b) where ReturnStmtAST had its return value checked against the parent function, and if that function was to return void, int, or bool etc.

4. Code Optimization using LLVM

The following headers were added to the expr-codegen.yfile:

```
#include "llvm/IR/DataLayout.h"
#include "llvm/PassManager.h"
#include "llvm/ExecutionEngine/ExecutionEngine.h"
#include "llvm/ExecutionEngine/JIT.h"
#include "llvm/IR/DerivedTypes.h"
#include "llvm/Analysis/Verifier.h"
#include "llvm/Analysis/Passes.h"
#include "llvm/Transforms/Scalar.h"
#include "llvm/Support/TargetSelect.h"
```

When attempting to do optimization:
We noticed that if q6-failure passed 1/1
the rest of the test cases failed. Except for two of the semantic checks.
For those that did fail The following message was being displayed: code generation exit status failure message.

Most of the errors appeared due to this line added in the expr-codegen.y:

```
std::string ErrorStr;
  TheExecutionEngine = llvm::EngineBuilder(TheModule).setErrorStr(&ErrorStr).create();
  if (!TheExecutionEngine) {
    fprintf(stderr, "Could not create ExecutionEngine: %s\n", ErrorStr.c_str());
    exit(1);   //Exit Faiulre
  }
```

This is supposed to take ownership of the module being used.

As well as the following line of code: This one in particular is supposed set up the optimizer pipeline.

```
  OurFPM.add(new llvm::DataLayout(*TheExecutionEngine->getDataLayout()));
```

The code above was critical for the next steps:

```
 // Reassociate expressions
     // OpFPM.add(llvm::createReassociatePass());
  // Eliminate Common SubExpressions.
      //OpFPM.add(llvm::createGVNPass());
  //control flow graph simplification
      //OpFPM.add(llvm::createGVNPass());
```

```
OpFPM.doInitialization();
```

Lastly we want to simplify the control flow such as deleting any of the unreachable blocks.

Once those steps are done then set the global so codegen can have access to it.
TheFPM=&OpFPM;

All these is done in the main () function in the expr-codegen.y file.