LLVM IR Generation

```
/* Sample AST class */
class Element {
        //Fields
        //Constructors
         return_type accept(Visitor v) {
                return v.visit(this);    //optional return based on requirement in IR generation
        }
}

/*
 * Sample Visitor Design class methods implementation
 * Class: VisitorImpl
 * Method Return Type: any valid return type
*/
class VisitorImpl {
        return_type visit(Element e)
        {
            //Code generation logic for elements e comes here.
         }
         return_type  visit(Element1 e1, Element2 e2)
        {
            //Code generation logic for elements e1 and e2 comes here.
        }
}
```

Code generation:

Its starts with llvm module and IR Builder objects, which are declared as public parameters.

1. Module and IR Builder:
   a. Module stores all the information of the IR being generated.
   b. Module Contains
      i. Globals
      ii. Functions
   c. IR Builder constructs the LLVM IR. It is used for creating LLVM Instructions(e.g., arithmetic, shift, method calls, storing and loading data from memory, etc..), creating basic blocks, functions, branches, setting insertion points (used in conditional statements), etc .,

```
#include <llvm/Module.h>
…….
static IRBuilder<> Builder(getGlobalContext());
class VisitorImpl {
```

```
public:
    Module *module;
    …….
    VisitorImpl() {
        module = new Module("main", getGlobalContext());
    }
    …….
}
```

2. Function
   a. It is used for two purposes - creating a function & basic blocks which are to be declared in the scope of function.
   b. Creating a function
      ```
      FunctionType *FT  =  FunctionType::get(Type::getDoubleTy(getGlobalContext()), Doubles, false);
      Function *F =  Function::Create(FT, Function::ExternalLinkage, "function name", module);
      ```
   c. Inserting basic blocks (e.g., creating basic block if condition)
      ```
      Function *f = Builder.GetInsertBlock()->getParent();
      BasicBlock *if = BasicBlock::Create(getGlobalContext(), "if", f);
      // explore how to create basic blocks for else/for statements.
      ```
3. Globals
   a. All the global parameters must be declared as llvm global variables. In decaf all the array elements are to be declared as global along with other other globals.

Basic Blocks

   b. LLVM basic blocks are used for functions, conditionals, nested blocks, etc,
   c. Setting start point and end points is important for a basic block, you will notice the importance in conditional statements.
      ```
      BasicBlock *bb = BasicBlock::Create(getGlobalContext(), "sample basic block", f);
      ```

Instructions

1. By now we got some information about the hierarchy in llvm ir generation. Instructions are the elements of basic blocks. All the elements of AST are to be converted to instructions in the order of the semantical execution of program.
2. Example:

   Rule: Statement1 -> Statement2 '+' Statement3
   AST Implementation

```
class Statement1 {
    public:
        Statement2 s2=null;
        Statement3 s3=null;
        Statement1(Statement2 s2, Statement3 s3){
            this.s2=s2;
```

```
                this.s3=s3;
        }
        void accept(Visitor v) {
            v.visit(this);
        }
    }
```

Code Generation

```
        Value *VisitorImpl::visit(Statement s) {
                Value *s1 = Statement2->accept();
                Value *s2 = Statement3->accept();
                Value res = Builder.CreateFAdd(s1, s2, "sample instr");
                return res
        }
```

Store instruction to store value of statement1 in memory

```
        BasicBlock *currBlock =
        idLoc = /* stack location; explore stackallocation instr */
        Value *st = Statement1.visit();
        new StoreInst(st,idLoc,currBlock);
//explore constructs for each instruction.
```

Method calls & Callout:

1. All the function objects created in code generations are stored by module. Function objects are retrieved from the module and function call is made using call instruction along with the required argument values.

```
Function *func = module->getFunction("Function Name");
std::vector<Value *> args;
Builder.CreateCall(func, args, "call instr"); //there are other constructs please look into them.
```

2. Figure out how "callout" works, look for the definition of callout.

LLVM Types and Constants:

1. Explore llvm types and llvm constants, use appropriate ones. For example boolean requires an integer type of width '1' bit.

Specifying start point of program execution:

1. Module object created in the code generation process must be analyzed using llvm verify module. This notifies the errors in code generated.
2. After finishing the code generation process. Explicit call (llvm method call) is to be made to invoke main function, if no main function is defined, throw an error.
3. If there are no errors, extract IR generated to a file from the module object and execute the IR with clang/llvm.

Note: This step can be automated by directly passing the module to LLVM JIT, but prefer two step process for demonstrating your work.

Validating generated IR:

1. Clang can be used to generate LLVM IR for c/c++ source file. As Decaf follows C++ structure, it can be compiled with Clang.
   clang –S –emit-llvm s.c
2. Validate the output of your compiler with Clang output

Reference Links:
http://www.cs.sfu.ca/~anoop/teaching/CMPT-379-Fall-2013/decafLLVM.pdf
http://llvm.org/docs/tutorial/LangImpl3.html
http://llvm.org/docs/doxygen/html/classllvm_1_1IRBuilder.html