# Codegen : CS3423

## Code Documentation

Saksham Mittal CS16BTECH110032
Shubham Kumar ES16BTECH11028

November 6, 2018

# Contents


**1 Code Overview** **3**

**2 Class structure** **3**

2.1 Codegen.java . . . 3
2.2 ClassInfoCG.java . . . 4
2.3 TypeMapping.java . . . 5
2.4 BasicClassBlockCG.java . . . 6
2.5 InstructionInfo.java . . . 6
2.6 OpClass.java . . . 6
2.7 GenerateLlvm.java . . . 6
2.8 VisitNodeClass.java . . . 9
2.8.1 AST.assign : . . . 9
2.8.2 AST.static_dispatch : . . . 10
2.8.3 AST.cond(IF-ELSE) : . . . 10
2.8.4 AST.loop(while) : . . . 10
2.8.5 AST.block : . . . 10
2.8.6 AST.new : . . . 10
2.8.7 Arithmetic Expressions(+, -, *, /) : . . . 10
2.8.8 Unary Operations( , not) : . . . 11
2.8.9 AST.object : . . . 11
2.8.10 Constants(int_const, string_const, bool_const) : . . . 11
2.9 PrintUtility.java . . . 11
2.10 ConstValClass.java . . . 12
2.11 IntValClass.java . . . 12
2.12 BoolValClass.java . . . 12

**3 File hierarchy** **13**

**4 Test Cases** **13**

# 1 Code Overview

The code takes in an annotated AST of a COOL program produced by the semantic analyzer and generates an equivalent representation of it in LLVM-IR. The LLVM-IR generated can be compiled with clang to get the output that the COOL program was expected to give. The working of the code can be divided into the following parts:

- First we generate various declarations that would be present in the IR generated for any COOL program.
- Then we perform some operations to store information that the classes in the COOL program contain including the default classes i.e. Object, IO, Int, String and Bool.
- We then generate IR code for the entrypoint of the code i.e. main() method of IR. The code for the definitions of methods of String, IO and Object classes is also emitted.
- Then IR code for string constants present in the class as assigned values to String attributes and IR code for the constructor of the class is generated.
- Finally, the IR code for the methods of a class is generated along with the string constants present inside the methods. This process also invokes the AST visitor pattern to emit out code for all the expressions inside the method.

# 2 Class structure

## 2.1 Codegen.java

This is the driver class for our program. The member variables include :

- **Object of PrintUtility class :** The class of this object(printUtil) contains methods for printing the IR statements which are used often(like alloca, load, and store, etc.)
- **Object of GenerateLlvm class :** The class of this object(gl) contains methods for generating llvm-IR code for C methods, methods of predefined classes(abort for IO, out_string for IO, etc.), constructor of classes, and generating IR for all classes.
- **Static Object of ClassInfoCG class :** The class of this object(clsInfoCG) iterates over all classes and adds their information like methods and attributes to a Hashmap.
- **Static Hashmap<String, Integer> stringLineToMapping :** This maps the string constants to their line numbers so that the llvm-IR code can be generated for those constants

The Codegen class performs the following actions in its constructor :

- Instantiate the printUtil object.
- Instantiate the gl object.
- Call the declareStandardCFunctions() method from GenerateLlvm class. This method generates the code for C methods like strcat(), strcmp(), printf(), scanf() which will be called by the COOL methods. **[For more info, see the GenerateLlvm class]**.
- Next, it calls the parameterized constructor of ClassInfoCG class, which iterates over all classes and adds their information like methods and attributes to a Hashmap.
- Then, generateLlvmIrForClasses() method of GenerateLlvm class is called. This generates the IR for all classes by iterating over them. **[For more info, see the GenerateLlvm class]**.

## 2.2 ClassInfoCG.java

This class stores information of all classes in form of Hashmap, which maps the class names to their BasicClassBlockCG objects. It has only one member variable : Hashmap<String, BasicClassBlockCG> cls. This maps class names to all the class related attributes and methods.

The constructor adds the default classes and their methods to the AST.program.classes list. The default classes include :

- **Object**
- **IO**
- **Int**
- **Bool**
- **String**

Object has methods:

- **abort() : Object**
- **type_name(): String**
- **copy() : Object**

IO has methods:

- **out_string(x : String) : IO**
- **out_int(x : Int) : IO**

- **in_string() : String**
- **in_int() : Int**

String has methods :

- **length() : Int**
- **concat(s : String) : String**
- **substr(i : Int, l : Int) : String**

Furthur, the class has the method insertClasses(). This method iterates over all classes of AST.program.classes and inserts their attributes and methods to the Hashmap cls.

## 2.3 TypeMapping.java

This class is used almost everywhere as it maps the types specified by COOL language to the types we are going to use in the llvm-IR. The class includes the following members :

- **TypeID enum :** The members of this enum are mapped to a String name.
- **Parameterized constructor 1 :** This constructor takes an Object of class TypeID and assigns the name corresponding to the enum. For eg. INT1PTR is mapped to i8*, and so on.
- **Parameterized constructor 2 :** This constructor takes a String and assigns the id to be of Object type and adds '%' to the beginning of the name. This is for used because user defined types/classes are mapped to Object type.
- **Parameterized constructor 3 :** This is same as Parameterized constructor 2 but it also takes an int which is for the number of pointers to be added to the string name. This is also for user defined types.
- **A boolean method isPtr() :** This method checks if the id of the TypeMapping object has single pointer or not.
- **A boolean method isDoublePtr() :** This method checks if the id of the TypeMapping object has double pointer or not.
- **correspondingPtrType() method :** This method returns a TypeMapping object corresponding to the pointer type of the current id. For eg. if id is INT32, we return TypeMapping object of id INT32PTR.
- **dereferencedPtrType() method :** This does the reverse of the above method, i.e. if we id is INT32PTR, we return TypeMapping object of id INT32.

## 2.4 BasicClassBlockCG.java

This is a helper class which contains basic info of a class like :

- Class name
- Parent class name
- Attribute list of class
- Method list of class

## 2.5 InstructionInfo.java

This is a helper class containing the following information for an IR instruction :

- Register value used in that instruction
- Type of last instruction which is an object of TypeMapping class
- Name of last basic block

## 2.6 OpClass.java

This is a helper class for storing operand information. Its constructor takes an object of class TypeMapping and a string and assigns it to Operand's type and name. The name is appended with '%' in the beginning. There is also an empty constructor which initializes the type with empty and name as empty string.

## 2.7 GenerateLlvm.java

This is one of the important class since all of the functionality of a program is inside its methods. We are able to generate the IR code for all types of expressions inside a method with the help of a utility class called PrintUtility. It has following methods :

- **declareStandardCFunctions() :** This method generates code for C format specifiers(%d, %s), C String methods(strcat, strcpy, strcmp, strncpy, strlen), C IO methods(printf, scanf), C malloc and exit methods. For each of these methods generateDeclUtil() is called which generates the required llvm-IR. **[For more info, see the generateDeclUtil() method of PrintUtil class]**.
- **generateStringMethods() :** This method generates the code for COOL defined string methods, by calling the corresponding C string methods.Code for length, concat, substr, and strcmp methods are generated appropriately.

- **generateObjectMethods() :** This method generates the llvm-IR code for methods of COOL predefined class Object. The name is mangled in the following manner : Object_<method-name>. As per writing of this report, we have generated code for only abort method of Object.

- **generateIOMethods() :** This method generates the llvm-IR code for methods of COOL predefined class IO. The name is mangled in the following manner : IO_<method-name>.

  The code is generated for following methods :

  - **out_string**
  - **out_int**
  - **in_string**
  - **in_int**

- **operandType() :** This method an object of class TypeMapping corresponding to the typeid and the number of pointers in the parameters. For non-predefined classes it appends 'class.' to the object id **[For more info of Typemapping constructors, see the TypeMapping class]**.

- **generateConstructorOfClass() :** This method generates the llvm-IR code corresponding to the definition of constructor of classes. It starts with storing the mangled name of constructor in the form : <class-name>_Cons<class-name>. The we declare the attrOperandList, which contains the attributes of this class. 'this' attribute is added first, because it is present by default. Then we generate the declaration of this constructor by calling generateDeclUtil() method. Then code for alloca, load, and store instructions is called for 'this' attribute. Then, we loop over the attributes of the class to generate alloca, store, and load instructions for them. If the attribute is a predefined class(Bool, Int, String) then they are handled separately.
  For each attribute, we check if it is of AST.no_expr class. If that is the case, we invoke the store instruction there itself, else we call it after calling the VistorPattern on the attribute's value.**[For more info of VisitorPattern, see the VisitNodeClass.java]**.In the end, the return instruction for the class is generated.

- **stringProcess() :** This method takes an object of AST.expression class and an object of Integer class which is a counter of the the number of string constants before the current string constant as its arguments. This method then compares the incoming expression with all the possible types of expressions in COOL that can have a string constant involved. It then recurses on the subexpressions involved in the current expression until it reaches an expression of AST.string_const class. Then it emits the IR code for the string constant with its name mangled with the integer passed

to so that each string constant has a unique name. It also takes care of escape characters which may appear in the string.

- **allocateMethodAttributes() :** This method generates IR code for alloca and store instructions on all method attributes. It takes an attribute list as one of its parameters.

- **generateIrForClasses() :** This is one of the most important methods of this class, which is called from Codegen class. It iterates over all classes of AST.program.classes and generates the llvm-IR code for them.

The class 'Main' is handled separately. Firstly, the code for C main method is generated which is the entry point of IR. This main method then invokes the COOL defined main method and Main class constructor in IR.

For other predefined classes of COOL, the appropriate method is called(For eg. generateStringMethods() method for String class). This takes care of generating code for constructor for IO and Object classes also.

Now, we handle the rest of the classes. We start off by iterating over the attributes of the class, and calling the stringProcess() method for capturing the string constants. Then we generate code for the attributes themselves, by calling the classTypeUtil() method of printUtility class. Then, we invoke the generateConstructorOfClass() method which generates the code for Constructor of class and also initialization of attributes.

Next part involves iterating over all methods of this class, and performing the following steps:

- The first step in generating code for a method is to emit out the IR code for all the string constants present in the given method. This is done with the help of a stringProcess() method.

- The second step in this part is to emit the definition of the method. Firstly, we emit out the define instruction in IR using PrintUtility::generateDefUtil() method. Then we emit out the alloca instructions for the formals of the method using PrintUtility::allocaInstUtil() method. We then go on to check if the names of any of the formals clash with the name of any of the attributes of that class.

- The third step is to visit all the expressions inside a method. Here we initialise an object(registerCounter) of class InstructionInfo. This object keeps track of the register number that was used in the last instruction and updates it accordingly. It also has an attribute which keeps track of the type of last instruction which is an object of class TypeMapping. Also, this object has an member variable to keep track of the last basic block we entered in the IR. We then call the visitor pattern VisitNodeClass::VisitorPattern() method on the body of the current method.

- After the visitor mechanism is complete we perform store operation to store the return value of the function which was being operated. Then the code for run time error message generating labels is generated. These labels are dispatch_on_void_basic_block and func_div_by_zero_abort. An error message is printed for both the cases using out_string() function of IO class and the program is aborted by invoking abort() function of Object class.

- **attributeAddressOfObj() :** This method returns a string containing the address writing format of IR for a AST.formal list of a method.

## 2.8 VisitNodeClass.java

This class is implementing the visitor mechanism on the generated AST. It has the following methods :

- **operandType() :** This method an object of class TypeMapping corresponding to the typeid and the number of pointers in the parameters. For non-predefined classes it appends 'class.' to the object id **[For more info of Typemapping constructors, see the TypeMapping class]**.

- **attributeAddressOfObj() :** This method returns a string containing the address writing format of IR for a AST.formal list of a method.

- **returnTypeOfMethod() :** This method returns an object of TypeMapping class corresponding to the return type of method. It takes the typeid of method and method name as one of its parameter. For Object method type, we return TypeMapping object of VOID type, else we call the operandType() method

- **VisitorPattern() :** This is the driver method of this class, which takes in an expression of and calls the corresponding overloaded VisitNode() method on the typecasted expression, depending on the type of the expression.

- **VisitNode() :** This method is overloaded for different types of AST.expr.

The working of VisitNode() method for different types of expressions can be described as written below(Note that there is a separate VisitNode method for each of these expressions):

### 2.8.1 AST.assign :

This first calls recursively invokes VisitNode() on the expression that is to be assigned. Then it invokes a method called attributeAddressOfObj() on the attribute or formal to be assigned which returns a string containing the address writing format of IR for a variable. Finally it terminates after invoking PrintUtility::storeInstUtil() to print the store operation in IR.

#### 2.8.2 AST.static_dispatch :

Here we first recurse on the caller expression if the type of static dispatch is not any of Int, Bool or String. Then, we check for dispatch on void condition and branch the control flow of the IR accordingly. Then we recurse for all the actuals expressions in the dispatch in a loop. After that the call instruction is emitted with the help of PrintUtility::callInstUtil() function. Register count is increased at return operation in case of the return type of the method is not void.

#### 2.8.3 AST.cond(IF-ELSE) :

First of all for handling the nesting of if-else statements we have a static variable that keeps count of the value of nesting to mangle the if-else labels accordingly. In this function we start by recursion on the predicate expression followed by emitting the branching instruction according to the predicate with the help of PrintUtility::brConditionUtil() function. Then we emit label for if.then followed by recursion on the ifbody expression and branching to the if.end label. Same is done for if.else and elsebody. For handling nested if-else we use phi nodes which assigns the register value of else block as either then block minus one or else block minus one depending upon the branching.

#### 2.8.4 AST.loop(while) :

Just like for if-else we keep a variable to keep track of nesting. Whenever this expression is visited we branch to for.cond. Then we visit the predicate expression of the loop and branch accordingly to either for.body or for.end. This is followed by visiting the body expression which is then branched to for.cond to complete the loop. Last label for the loop is for.end which when branched to, executes the IR in its original order after the loop has has ended.

#### 2.8.5 AST.block :

For this we visit all of the expressions in the list in a loop.

#### 2.8.6 AST.new:

For this expression we first check if the object being created is of type String, Bool, Int. If it is then we just simply call PrintUtility::allocaInstUtil(), PrintUtility::storeInstUtil() and PrintUtility::loadInstUtil(). Int object is initialized with a value of zero, false for Bool and empty string for String type object. For object of other classes, the constructor of that class is called after performing alloca operation.

#### 2.8.7 Arithmetic Expressions(+, -, *, /) :

The code generating mechanism for all the arithmetic operations is same except for division which is explained. For AST.plus, AST.sub and AST.mul first both

the expressions to be added are visited and then PrintUtility::arithmeticUtil() is invoked which emits add, sub, mul operation statements to IR. The only thing different in case of division is that we also have to check whether the divisor is zero or not. If it is zero then we branch to func_div_by_zero_abort which would print an error message. If it is not zero then we branch to our normal execution.

#### 2.8.8 Unary Operations( , not) :

For compliment operation( ) we perform xor operation in IR between the expression to be complimented and true with invoking PrintUtility::arithmeticUtil() after visiting the expression to be complimented. For negation(not) we just perform mul operation in IR between the expression to be negated and -1 with invoking PrintUtility::arithmeticUtil() after visiting the expression to be negated.

#### 2.8.9 AST.object :

In this case load operation is performed in the IR which is emitted with the help of PrintUtility::loadInstUtil() for the corresponding type of object.

#### 2.8.10 Constants(int_const, string_const, bool_const) :

In this case we just perform alloca, store and load operations for the corresponding constant on register level. For String constants store operation is performed by using getelementptr inbounds instruction with the corresponding size of the string.

### 2.9 PrintUtility.java

This is a helper class containing methods to print the various possible llvm-IR instructions. Every method prints an IR line for corresponding operations happening in the IR.

This prevents us from writing code for various operations again and again and helps implement code reusability.

This has the following methods :

- **returnInstUtil() :** Generates IR instruction for return operation in a method.
- **allocaInstUtil() :** Generates IR instruction for alloca operation for a variable.
- **loadInstUtil() :** Generates IR instruction for load operation for a variable.
- **storeInstUtil() :** Generates IR instruction for store operation for a variable.

- **stringEscUtil() :** Used by stringProcess() method to handle escaped characters in string constants.
- **generateDefUtil() :** Generates IR instruction for definition of a method.
- **generateDeclUtil() :** Generates IR instruction for declaration of a method.
- **arithemeticUtil() :** Generates IR instruction for performing the corresponding arithmetic operation.
- **cmpInstUtil() :** Generates IR instruction for performing the corresponding comparison instruction.
- **arithemeticUtil() :** Generates IR instruction for performing the corresponding arithmetic operation.
- **cmpInstUtil() :** Generates IR instruction for performing the corresponding comparison instruction.
- **getElemPtrInstUtil() :** Generates IR instruction for getelementptr inbounds operation.
- **brConditionUtil() :** Generates IR instruction for branching according to the trueness of a given condition.
- **brUncoditionUtil() :** Generates IR instruction for branching without any condition.
- **callInstUtil() :** Generates IR instruction for performing the calling of a method.
- **classTypeUtil() :** Generates IR instruction for declaration of a class.

## 2.10 ConstValClass.java

This class is a helper class which acts as a superclass for IntValClass and BoolValClass.

## 2.11 IntValClass.java

Utility class for representing operands of type 'Int' inheriting from ConstValClass. It contains a Integer value as member to store the integer value of the operand.

## 2.12 BoolValClass.java

Utility class for representing operands of type 'Bool' inheriting from ConstValClass. It contains a boolean value as member to store the boolean value of the operand.

# 3 File hierarchy

```
codegen/
|___src/
|    |___java/
|    |    |___cool/
|    |         |___AST.java
|    |         |___BasicClassBlock.java
|    |         |___BuildInheritanceGraph.java
|    |         |___BuildScopeTable.java
|    |         |___ClassInfo.java
|    |         |___ReportError.java
|    |         |___ScopeTabble.java
|    |         |___Semantic.java
|    |         |___VisitorMechanism.java
|    |         |___CodeGen.java
|    |         |___ClassInfoCG.java
|    |         |___TypeMapping.java
|    |         |___BasicClassBlockCG.java
|    |         |___InstructionInfo.java
|    |         |___OpClass.java
|    |         |___GenerateLlvm.java
|    |         |___VisitNodeClass.java
|    |         |___PrintUtility.java
|    |         |___ConstValClass.java
|    |         |___BoolValClass.java
|    |         |___IntValClass.java
|    |
|    |___test_cases/
|         |___Trivial/
|              |___test1.cl
|              |___test2.cl
|              |___test3.cl
|
|         |___NonTrivial/
|              |___test1.cl
|              |___test2.cl
|              |___test3.cl
|
|___Readme.pdf
```

# 4 Test Cases

We have divided the test cases into two categories :

- **Trivial:** These are trivial test cases which check for basic expressions like

arithmetic, unary and block.

- **NonTrivial:** Their description is as follows :
  - **test1.cl:** Testing for arithmetic operations along with static dispatch and new expressions.
  - **test2.cl:** Testing for unary operations, string and int constants along with if-else statements, while statement and static dispatch.
  - **test3.cl:** Testing for nested if-else and nested while and string functions.