Decaf LLVM Code Generator Report

Ashish Kumar
Mentor: Suresh Purini

Aim: Given a decaf program, parse the input and generate llvm code, after determining the syntactical correctness of the code. This project requires the building of the front end of a compiler, i.e. to generate intermediate representation from the source input.
The project was divided into 3 phases:
1. To build a lexer and a parser.
2. To generate the AST IR
3. To generate llvm intermediate representation

The project is to generate llvm IR fromt the source input. The project requires clear understanding of context free grammars, and working of the front end of the compiler. Also, the programmer must have knowledge of c++ programming language, and basic regular expressions. The design of the code is modular and holds good design practices. We have made sure that variable and class declaration follow camelcasing, and descriptive names.

The first phase involved writing the lexical analyzer. We had to define all token types and parse them using lex. The tokenizer can be found in “tokens.l”. The parser “parser.y” contains the grammar which determines the syntactic correctness of the code. This phase was done using lex and bison as the main programming tools. A makefile was written to compile the program.

The second phase involved required generating AST IR. This was done by declaring classes in “codegen.cpp” and using the concept of inheritence. Hence, the reuquirement of an object oriented language. The code was written with modularity and all classes were given a “print()” function which allows the AST to be printed. All parent nodes call the print function of their child nodes and this has been done recursively.

AST Classes and inheritence:

ASTStart node

```
          |                  |                            |                  |                            |                  |
ASTExpr       ASTStatement              ASTDecl     ASTProgram     ASTOp        Variable
```

ASTDecl

```
                             |                                      |
          ASTMethodDecl                 ASTVarDecl
```

ASTStatement

```
|                              |            |            |                  |                            |              |
ASTAssignment   ASTIf  ASTFor   ASTReturn   ASTBlock  ASTKeyword     ASTCaller
```

ASTCaller

```
                             |                                               |
                    ASTCallout                       ASTMethodCall
```

ASTExpr

```
                             |                                               |
                    ASTLocation                ASTInt other constant classes
```

The third phase requried the generation of llvm IR. This was done using llvm module and required online reference beacause of the typical syntax. In this phase, we used an online template for generating llvm code and then wrote llvm code corresponding to each class in the AST.

Problems Faced:

1. 27 Shift/Reduce conflicts. The modulus operator was made left associative.

3. Initially, the grammar that we wrote had separate 'statement' grammar for the program body and the method block. Our program was running incorrectly because of this.

4. Installling LLVM 3.2 did not work. So, we upgraded to 3.4.

5. Initially, we were storing the variables as string and had not specified a separate class for them. This caused us problems during the llvm code generation part. The new Variable class is generate at terminal Id. A single non-terminal node Variable which holds the identifier.

Conclusion and Future Work:

1. The AST part of the project works for all decaf code.
2. The llvm code generation part of the project is not done for FOR, IF, CALLOUT statements. Hence, the code does not handle recursions, and can not print any output.

LLVM Resource: ageofblue.blogspot.in/2012/01/writing-your-own-toy-compiler-using.html

The future aim will be be to complete the llvm code generation for the rest of the statements.