Compilers Project
Report

Mohit Aggarwal
201101164

The aim of the project was to write a compiler for the Decaf language. Decaf is a simple imperative language similar to C or Pascal.

The compiler construction was done in 3 phases :

- **Phase 1 -** Writing a parser for parsing the source code, using flex and bison, and detecting any errors.
- **Phase 2 -** Constructing an AST of the given source code, using bison, defining a custom class for each type of node.
- **Phase 3 -** Generating IR code from each of the nodes in the AST, using LLVM.

Phase 1

This phase involved writing a parser in flex and writing corresponding grammar rules using bison, (specified in the handout).

Challenges faced

- In this phase, it took us time to understand the code structure of the lex and yacc files, and how they work together.
- Specifying the correct regular expressions for different tokens was a challenging task. However after much practice, we were able to write the regular expressions for different tokens that would be present in Decaf language. While working on this phase, we got a clear understanding of the Decaf language.
- We also faced a lot of shift reduce errors while parsing the source code. It took us a lot of time and effort in coming up with a solution to tackle them. We used precedence rules to solve these errors, and in some cases left factored the grammar rules.

Phase 2

This phase involved the construction of Abstract Syntax Tree, using bison. This required creating a ***class hierarchy*** for different type of nonterminals in the grammar. Our hierarchy consists of:

- *class_node* at top, having *field_decl_node* list and *method_decl_node* list as its children.
- Each *field_decl_node* has *idListVar* (list of variables) and *idListArray* (list of Arrays) as its children.
- Each *method_decl_node* has a type, id, list of parameters and a *block_node* as its child.
- Each *block_node* in turn has *var_decl_node* and *statement_node* as its children.

- Assignment, for, if-then-else, method call, break, continue, etc nodes inherit from *statement_node*.
- Inherited classes of *statement_node* have *expr_node* as its children.
- method_call, literal, binary expression, unary expression, *operator_node* all inherit from *expr_node*.

Challenges faced

- This phase required going into more detail and therefore required more effort. Coming up with a proper class hierarchy for different types of non terminals in the grammar was a tough task.
- Specific proper actions for each grammar rule so that we construct the correct AST was a challenging task. It took considerable amount of time and effort to come up with the correct set of actions for each rule.
- While working on this part, we got all types of runtime errors ranging from segmentation faults to OOPS related errors. We got our concepts of OOPs clearer while working on this phase.
- We also spent time on deciding how to display the AST. We had to print it in such a way, so that one could easily understand the program flow just by looking at the AST.

Phase 3

This phase required generating the LLVM IR for the AST. For this we had to specify the Codegen() method in each class, and then call the codegen() method of the root node. This would result in code generation of the entire AST(by way of recursion).

Understanding the LLVM API took us a lot of time and effort, as it was completely new. This was the *most challenging part* of the project. However we progressed incrementally, from generating IR for constants to generating IR for for-loops and so on.

The **process** we followed was

- First generate the IR assuming there are no mutable variables. In this case, the only variables in a method were its parameters.
- Assuming this, IR for Binary operations, method declarations, method calls, if-else-then expression, and for-loop expression were added in the given order.
- After that, it was modified to allow the declaration of local mutable variables and also, assignment of already declared variables. Mutable variables were dealt with using location binding of the variables.
- Then, we tried to add the global variables support, which includes arrays also. It was pretty difficult as there was no documentation available for it. We were able to successfully generate the IR for the declaration of the global variables, but due to the time constraint, could not progress further in this.