Compilers Project
Report

ANANYA VAKA
201401249

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 4 phases :

- **Phase 1**- Make a code to perform lexical analysis and generate Tokens (using flex).
- **Phase 2**- Writing a parser using bison which can work together with lexical analyser to do semantic analysis
- **Phase 3**- Constructing an AST of the given source code, using bison, defining a custom class for each type of node.
- **Phase 4-** Generating IR code from each of the nodes in the AST, using LLVM.

Phase 1

This phase involved writing a scanner using flex

Challenges faced:
Specifying the correct regular expressions for different tokens was a challenging task while defining the tokens in “scanner.l”.
This helped us understand and get a better hold of the Compiler structure and how the RE are useful in compilation process.

Phase 2

This phase involved writing a parser using bison.The file “decaf.l”, is the scanner that is used to tokenize the input file. The parser “decaf.y” contains the grammar rules for a subset of the Decaf Language which determines the syntactic correctness of the code.
A makefile was written to compile the code.

Challenges faced:
This phase involved spending a lot time in understanding the structure of the lex and yacc files and figuring out how they work together.
The major area we had problems in and the errors we faced were - Shift Reduce Conflicts and Reduce Reduce Conflicts - due to errors in the grammar, while parsing the source code. In order to remove Shift-Reduce errors, we had to introduce the use of precedence rules and also left-factored some grammar rules. The Reduce-Reduce errors occurred due

to ambiguity in the grammar rules. These were removed by re-writing the rules, so as to make them unambiguous

Phase 3

This phase involved the construction of Abstract Syntax Tree.This required creating a ***class hierarchy*** for different type of nonterminals in the grammar.

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 4

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.