CS335 Project Milestone 3 Report

Group Members: Divyansh Kankariya (200352), Nishi Mehta (200645), Mohd. Umam (200595)

April 20, 2023

1 Tools Used

In this project, we used the following tools:

1.1 Flex

Flex is a tool for generating scanners (also known as lexers or tokenizers) that can recognize regular expressions in the text. We used Flex to generate a lexer for the Java 17 language.

1.2 Bison

Bison is a tool for generating parsers that can recognize context-free grammar. We used Bison to generate a parser for the Java 17 language.

1.3 GAS x86_64

Assembler used to run the assembly language.

2 Compilation and Execution Instructions

To compile and execute our lexer and parser, follow the steps below:

1. Unzip the file `cs335_project_200645_200595_200352`
2. Navigate to the project directory
3. Compile the lexer and parser using the makefile provided.

```
// Your code goes here
make
make st
make gcc
```

- On executing the 'make st' command, we will have to enter the filename of the test file. After the execution of the parser, all the symbol tables are stored in the folder 'outputSymTabs' with the filename as 'Scope_SCN.csv' where SCN is the scope number. For example, In our representation, the scope number is represented as '1_2a_3a' for three scopes.
- The type-checking errors corresponding to the line of code are saved into a new text file log.txt. In the same file, complete symbol table and 3AC code have been printed to make debugging easier.
- The 3AC code generated has been stored as output3AC/3AC.txt file.
- The x86_64 instructions file is saved in the same location as all the other source files. Name of this code is "assembly.s"

3 Features we worked on during this milestone

We have updated following features in this milestone which were implemented in previous milestone:

- Our work in last milestone majorly fulfilled all the requirements except one or 2.
- We tried to correct the parsing, symbol table entry and typechecking in the statements where "this" keyword is used.
- We improvised our 3AC code for ArrayCreationExpression and ArrayAccess, so as to become compatible with further implementation

In the final project milestone we generated x86_64 assembly instructions using the 3AC code and symbol table generated in previous milestones. Assembling this file using gcc, we get correct output of the code for huge range of functionalities.

- Basic arithmetic operations, works fine. The output is printed on the console which can be used to check the correctness.
- If-Else statements, For, While, Do-While Loops are functional, along with the break and continue statements.
- MethodInvocation is also functional. Recursive implementation runs without any error and generates the correct output.
- ArrayCreation and ArrayAccess is handled properly. The ArrayAccess includes both, assigning value to an array entry as well as using an entry of the array for assignment of other variables.
- **Optional**: We also have addded the functionality of handling multidimensional arrays, where the ArrayCreationExpression can have both Java-type and C-type syntax.
  - 3AC code for array access was implemented as given in the slides
  - 3AC code for array creation expression included "new" keyword and size of the array to be created
  - **call malloc** is used to allocate memory on heap, where we give size in bytes as argument and receive address on heap of array as return value
  - following the 3AC code handling the array access was not major issue
  - type of both declaration and array creation expression is stored in same way eg:- "int[][]" for 2 dimensional int array so type checking becomes easy
- Object creation and ConstructorInvocation, whenever an object of particular class has been created, has been taken care of.
- Public fields of that class of which the object is created, can be accessed from outside the class, both for assigning its value and its value may be used in RHS of any assignment expression.
- **Optional**: Modifiers like Private, Public and Static are functional as per the Java rules.
  - Public and Private are end-to-end fully functional, if a field is public only then it can be accessed not if it is private
  - Static was functional until milestone 3, that is typechecking was being done correctly, but could not extend it in milestone 4
  - We implemented these modifiers using an integer flag which was used to translate the modifiers to symbol table entry
  - each time accesing a field, it checks the modifier field of the entry in symbol table
- TypeChecking in each and every place where it is required is ensure including -
  - types of all the operands in arithmetic operators ,

- number, type and order of arguments in MethodInvocation,
- dimensionality of arrays,
- redeclaration of variable with same name

etc.

In addition, we have created 9 test cases to ensure the correctness of our compiler. These test cases are located in the `tests` folder

4 Table for Contribution

Sr No	**Name**	**Roll No**	**Email**	**Contr.**
1	Nishi	200645	nishim20@iitk.ac.in	30%
1	Divyansh	200352	divyanshmk20@iitk.ac.in	40%
1	Umam	200595	umam20@iitk.ac.in	30%