CS335: Project

Milestone-4

Kajal Deep (200483),
Kuldeep Singh (200530),
Sandeep Kumar (200856)

April 20, 2023

Tools and Utilities used

1. **Utilities**
   - *FLEX*: lexical Ananlysis
   - *BISON*: syntax Ananlysis
   - *GRAPHVIZ*: graphical analysis
   - *MAKEFILE*: compilation of multiple files
   - *ARGPARSE*: to implement command line arguments
2. **Sources**
   - Java Language Specification, ORACLE

How to run

1. Go to the src directory. Run the following command to remove previous generated files and compile the required ones:

   ```
   make clean
   make
   ```

   final is the executable.

2. For the command line executions, write

   ```
   ./final -h
   ```

   This will show you all the commands which can be given along with ./final

   - **-i**: For getting the input file, as -i="FileName"
   - **-o**: For obtaining the output file, as -o="FileName"

- **-h**: will show all the commands and there uses to the user
- **-v**: will show the version of the program
- **−−verbose**: it will tell, what all has happened through the execution of **final**.

3. To run the program, use the following command, where Source File is **test.java** and the output dot file is **out.dot**

```
./final -i="test.java" -o="out.dot"
```

4. To print the graph, use following command, where the dot file is **out.dot** and the graph will be in **graph.ps**

```
dot -Tps -o graph.ps out.dot
```

5. The symbol table, 3AC dump and the x86_64 assembly code can be found in file SymbolTable.csv, 3ac.txt and out.s respectively in src directory

6. Running the below command will test the **out.s**

```
gcc -no-pie out.s && ./a.out
```

Type Checking

1. checks for valid variable declaration with allowed data types
2. allows float int char typecasting
3. checks whether the correct type and number of arguments are passed into the function/ method
4. checks for the correct dimensions of array and its index is not out of the bound

Symbol Table

1. The symbol table stores the following informations :
   - Variable : variable name or method name
   - Type : type of the variable name or the return type of the method for method name
   - Line : line in which the variable or the method is declared
   - Size : size of the variable. In case of a method, size is given as 4 assuming it to be a pointer
   - Offset : the offset of the variable

The csv for the symbol Table has been output in the "SymbolTable.csv".

3-Address Code

3-Address Code has been output in the text file "3ac.txt". For this abstraction of assembly code we will take some assumptions and those functions would work as follows.

1. When the function call begins, space is allocated in the stack according to the size of the function. This space will be used for storing the local variables in that function.

2. For storing the local variables, on to the stack, we will move them as they are defined to the respective position in the stack given by

   $$basePointer - offset(a)$$

   where a, is the name of the variable. We will differentiate it from a temporary variable in Milestone 4, by the naming convention of temporary variables.

3. For a function call **LCall** is used. Before that, the arguments required for the function are pushed into the stack. The previous base pointer is pushed into the stack and then the base pointer is updated for the new scope. Then space is allocated for the arguments again along with the local variables in that function. The stackPointer is moved down according to the offset. The arguments are first added in that space, using
   getparam i, which does the following:

   $$mov \quad (basePointer - 8 - offset(i)) \quad (basePointer + (offset(i) + size(i)))$$

   While returning the calculated value **Return a**, stores a value in return-Register, which can be accessed in other functions.

4. After Return, the stackPointer is updated to the basePointer and the base-Pointer is restored to the previous one through the **RestoreMachineS-tate**. The pushed arguments are then removed by moving the stackPointer up the stack according to the total size of the arguments.

5. For objects and array, new keyword is used, which will allocate the required memory on the heap using **allocmem** and store its reference pointer on a temporary variable.

x86_64 Assembly Code

x86_64 Assembly Code has been output in the text file "out.s".

1. For printing integer variables, we have used the **call printf**, from gcc.

2. Some of the lines are given along with comments, for better understanding of the code.

3. 64 bit registers are used in our implementation.

Limitations and Assumptions

1. For printing the variable's value, instead of *system.out.println()*, we will be using $println\_ < variable\_name > ()$. For example, int x = 3, $println\_x()$, will print 3 on the terminal.

2. For printing an array value or value obtained from method invocation, we will first create a temporary variable, say x, then store that value in x and then use println_x().

3. Integer operations, loops, function calls works properly. Objects and constructors are also working.

4. Temporary variables are not stored, so before calling function expression or while accessing/ storing array at an index, store that function call or expression in a variable and then provide it later.

5. We are not creating any class object by default, So to use variables of class(i.e global variables) a object needs to be created by constructor. Also to access these variables "this" keyword is must. E.g this.x for global variable x. Whereas these points are optional in actual java compiler

Contribution

Name	Roll No.	Email ID	Contribution
Kajal Deep	200483	kajald20@iitk.ac.in	33.33%
Kuldeep Singh Chouhan	200530	kuldeepsc20@iitk.ac.in	33.33%
Sandeep Kumar Bijarnia	200856	sandeepb20@iitk.ac.in	33.33%