CS 335 Semester 2022–2023-II: Project Milestone 4

Group members:
Priyanka Jalan (190649)
Piyush Agarwal (190600)
Gargi Naladkar (200371)

April 20, 2023

Using Lex and Bison, we developed a scanner and parser that accepts any valid Java program with the features listed below as input and outputs a DOT script representing the AST(abstract syntax tree) of the input program. The DOT script, when processed by the Graphviz tool called dot, will produce a postscript file with the diagram of the parse tree. We incorporated the scanner and parser specifications that are stated in the issue statement for milestone 1.

In milestone 2, we implemented support for symbol table data structure to the parser we have developed in milestone 1. We also performed semantic analysis and did type checking including type casting. And finally we converted our input into 3-address code (3AC) code (intermediate representation).

In Milestone 3 we added runtime support for procedure calls. Our implementation of activation records include the following fields -

- Space for actual parameters
- Space for return value
- Space for old stack pointers to pop an activation
- Space for saved registers
- Space for locals

In Milestone 4, we implemented a translator to generate x86_64 instructions from the 3AC code we generated in milestone 3.

So this project as a whole implements a compilation toolchain where the input is in Java language and the output is x86_64 code.

Compilation and Execution:

1. Download the cs335 project zip file. Extract the contents.
2. Move to the src directory by using the following command in your terminal.

```
$ cd milestone4/src
```

3. To generate executable and compile the lexer and parser files, use the following command:

```
$ make
```

The above command generates the following files:

(a) milestone4 : an executable file.

(b) parser.output : descriptions of various states of LR.

(c) parser.tab.c : parsing program in C

(d) lex.yy.c : lexer program in C

(e) parser.tab.h : header file

Makefile contains the following commands:

(a) For parsing using "make" command:

```
bison -dtv parser.y
flex lexer.l
g++ -w lex.yy.c parser.tab.c -o milestone4
```

(b) For compiling the x86 code generated and further run the executable file using "make run" command:

```
gcc code_gen.s -o code_gen
./code_gen
```

(c) For testing using "make test1" command:

```
./milestone4 ../tests/test_1.java
```

(d) For testing using "make test2" command:

```
./milestone4 ../tests/test_2.java
```

(e) For testing using "make test3" command:

```
./milestone4 ../tests/test_3.java
```

(f) For testing using "make test4" command:

```
./milestone4 ../tests/test_4.java
```

(g) For testing using "make test5" command:

```
./milestone4 ../tests/test_5.java
```

(h) For testing using "make test6" command:

```
./milestone4 ../tests/test_6.java
```

(i) For testing using "make test7" command:

```
./milestone4 ../tests/test_7.java
```

(j) For testing using "make test8" command:

```
./milestone4 ../tests/test_8.java
```

(k) For testing using “make test9” command:

```
./milestone4 ../tests/test_9.java
```

(l) For testing using “make test10” command:

```
./milestone4 ../tests/test_10.java
```

(m) For cleaning using ”make clean” command:

```
rm output/*
rm milestone4
rm parser.tab.c parser.tab.h parser.output
rm lex.yy.c
rm symbol_table.txt output.dot
```

(n) For testing using ”make test” command:

```
./milestone4 ../tests/test_1.java
./milestone4 ../tests/test_2.java
./milestone4 ../tests/test_3.java
./milestone4 ../tests/test_4.java
./milestone4 ../tests/test_5.java
./milestone4 ../tests/test_6.java
./milestone4 ../tests/test_7.java
./milestone4 ../tests/test_8.java
./milestone4 ../tests/test_9.java
./milestone4 ../tests/test_10.java
```

4. A typical session for running the test file with address ”../tests/test_1.java” which creates files with the name ”code_gen.s” containing the x86 generated code, ”code_gen” the final executable file, ”output.3ac” containing the 3AC code for the input code, ”output.dot” containing the AST and a directory with name ”output” containing symbol tables with the file name < Class Name > . <Function Name> .csv, use the following command:

```
make
make test1
make run
```

5. The following options are supported for running the executable file:

```
usage: ./milestone4 [options] file
Options:
--help                  To print this message.
--input=<file name>     You can use it to specify the input file name.
                        By default, the name of the input file can be
                                                              specified
                                                              directly.
--output=<file name>    You can use it to specify the output file name.
                        By default, the name of the output file is 'output.
                                                              3ac'
Do not use --input=<file name> flag if you specify the input file name
                                      directly.
--verbose               It outputs the stack trace
```

6. To Render the AST generated, use the following command:

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

Features:

We have implemented following features upto Milestone 3 and our code would be able to generate 3AC code for it as well. We have created java programs incorporating these features and tested them in Milestone 3.

- Support for primitive data type(e.g. **byte, short, int, long, float, double, boolean, char**, etc.)
- Variables and Evaluation of Expressions with **correct type**
- Support for **For** and **While loops**
- Support for **Primitive data type Array** with many dimensions (applicable for greater than 3 dimensions too) declaration using new keyword :
  int arr[] = new int[10];
- Support for **System.out.print()** and **System.out.println()**
- Support for **Strings** (addition of strings with strings, int, boolean, char, float) and **Text block**
- Supported for **this** keyword
- **Variable Declaration** supported format:
  int a, b = 10, c;
- Support for **operators** $(+, -, *, /, !, \sim, <, >, \leq, \geq, <<, >>, >>>, ||, \&\&, |, \&, ==, !=, ?:)$

  1. Complement Operators: ($\sim$, !
  2. Additive operators: (+ , -)
  3. String concatenation operator: (+)
  4. Shift Operators: $(<<, >>, >>>)$
  5. Conditional Operators:$(\&\&, ||)$
  6. Relational Operators: $(<, >, <=, >=)$ for Numeric comparison.
  7. Equality Operators: (==, !=)
  8. Ternary Operator: (? :)
  9. Multiplicative Operators: (*, /(only int division supported in java), %(int and float))
  10. Assignment Operators:$(=, *=, /=, \%=, +=, -=, <<=, >>=, >>>=, \&=, \wedge=, |=)$
  11. Bitwise and logical Operators: $(\&, |, \wedge)$

- Support for **Unary Operation**:
  ++ and - - on numeric types(both post and pre)
  +, - as unary operator on numeric types.
  ! works on boolean type only.
  $\sim$ works on numeric integral types.
- Support for access modifiers **private** and **public** for variables and methods inside the class.
- Support for Class object declaration using new keyword. e.g.
  class tree { }
  public static void main(){
  tree obj = new tree();
  }
- Support for **Method Declaration**(Method Declaration after the main function and called inside the main method.)
- Support for **Object Method Invocation, field access**

Generation of 3-Address code:

- On execution the compiler generates files containing the 3AC code for each declarations within the class body.
- Each file corresponding to that declaration contains the 3AC code for every statement in that declaration. Each statement is represented using addresses and operators.
- For conditional statements jumps are defined using "goto" and "IfFalse" and "IfTrue" are used correspoding to false and true values of the statements. .For "for" and "while" loops also jumps to different states defined.
- For code x + y, if x is an int and y is a float, "cast_to_float" is used to represent type casting.
- "alloc_memory" used to represent allocation of memory of any array.
- Allocating memory for the array creation.
- **recursion, return, break, continue** statements working in 3ac
- Conditional statements working fine in 3ac
- Stack operations push, pop, call supported.

Runtime support for procedure calls:

- We have a base pointer and a stack pointer in the stack. The base pointer is used to mark the start of a function's stack frame. It marks the area of the stack managed by the function. The stack pointer points to the last value pushed onto the stack.
- The stack is divided into regions, one for each function, called stack frames.

  Callee Function :

- At the beginning, we load the object address in a temporary. We then load the function parameters from the caller saved stack address.
- We then allocate space in stack for base pointer. We then update the base pointer to the current stack pointer.
- We also allocate space for local variables and then the saved registers.
- After the setup, the 3AC code is then specified for each instruction of that function.
- We then store the return value present at the space allocated during calling of the function in a temporary.
- We then restore the stack pointer to callee base pointer and we pop the base pointer that was pushed at the beginning.
- We return from the function via a "return" instruction. This transfers execution to return address mentioned during the calling of the function.

Caller Function :

- During method invocation, we first push the formal parameters onto the stack. For more than one arguments, the first argument is pushed last. We then push the address of the current class object specified using this pointer.
- At the end of the function, we allocate space for the return value and push the return address onto the stack. We then allocate space for the return address pointer.
- After the setup the function is called using "CALL function-name".
- After the execution of the function we load the return value of the function in a temporary.
- We then restore the stack to that before the function call by incrementing the stack pointer by the total space occupied by the arguments passed in the function and the return address pointer.

Object Creation :

- During object creation we first store the size of object in a temporary.
- We then push formal parameter containing the size of object onto the stack, allocate space in stack for return address and push the return address onto the stack.
- We then allocate memory to the object using "CALL allocmemory 1".
- We store the object address in a temporary and restore the stack after function call.
- The object address stored is pushed onto the stack, space is allocated in stack for return address and the return address is pushed onto the stack.
- Finally the object is created by calling the constructor using "call constructor-name".
- We then store the object address in a temporary and restore the stack.

Generation of x86_64:

- On execution the compiler generates files containing the x86_64 code in code_gen.s
- Instructions used movl, addl, subl, imull, idivl, sall, sarl, cmpl,
- Each statement is represented using addresses and operators.
- For conditional statements, unconditional jumps are defined using "jmp" and conditional jumps using "jge", "jg", "jl", and "jle".
- Support for "for" and "while" loops
- Support for "if-else" statements
- Support for "print() and println()" statements
- Support for Recursion
- Support for arithmetic operations like Subtraction, division, multiplication, addition incorporated(all expressions too.)
- Support for preincrement, predecrement, postincrement, and postdecrement
- Support for relational operators: $(==, !=, >, <, >=, <=)$
- Support for bitwise operators: $(\sim, \&, |, <<, >>, >>>)$
- Support for assignment operators $(=, +=, -=, *=, /=)$

Basic features we could not implement in Milestone 4:

The main reason was due to lack of time.

- Arrays
- Boolean and char data type
- Support for class and objects
- Ternary operators
- Logical operators $(\&\&, ||, !)$

Contribution :

Member Name	Roll Number	Member Email	Contribution
Priyanka Jalan	190649	prianka@iitk.ac.in	50%
Piyush Agarwal	190600	piyuagr@iitk.ac.in	35%
Gargi Naladkar	200371	gargin20@iitk.ac.in	15%