CS335 Project Milestone 3

Rishabh Kothary
180608
rishk@iitk.ac.in

Rikesh Sharma
180606
rikesh@iitk.ac.in

Kartavya
180343
kartavya@iitk.ac.in

1 Tools Used

We did lexical analysis using *flex* and parsing using *bison*. To automate compiling we used *Makefile* and to make graphs we used *graphviz*. Our codebase is written in C++ programming language.

2 Flags Available

The flags available are :

- `--input=` : Used to specify path to input file
- `--output=` : Used to specify path to output file
- `--verbose` : Used to switch on the debugger of bison
- `--help` : Used to load the help page

3 Compilation

To compile the parser for the first time, on the Milestone_2 directory run the following command on the command line.

```
make parse
```

If you want to rebuild the parser then run the following commands from the Milestone_2 directory on the command line:

```
make clean
make parse
```

If you want to run our test suites in an automated manner run the following command from the Milestone_2 directory on the command line:

```
make test
```

then you would be asked for the test suite number. Is suppose you picked test suit x, then the files test_x.3ac is created in the output folder. In order to run multiple test suites at the same time run the following command on the command line from the Milestone_2 directory :

```
make test_many
```

and then you will be asked number n, which will tun $test_1.java$ to $test_n.java$ and return the output in the output folder.

If you want to run your own test cases then use the following commands from the Milestone_2 folder on the command line :

```
cd bin
./parser --input = < Path to Input File > --output = < Path to output file >
```

This would generate the 3ac output file.

4 Activation Record Structure

The Activation record structure is the following :

- **Parameters**
- **Return Values**
- **Saved Registers** : We are saving $\$rbp$ (Base Pointer Register) and the Program counter till now. We will expand it according to the x86 architecture in the next Milestone.
- **Local Variables** : We are allocating the space for Local Variables in the stack but are not showing the loading instructions of the local variables as the register allocation is not yet clear. We are assuming that we have infinite registers and thus are not allocating any space for the temporaries.

5 Caller Callee Convention

5.1 Caller Convention

Suppose the *caller* wants to invoke a function $callee(arg_1, ..., arg_n)$ then caller does the following tasks in order to call a method :

- Pushes arg_n, then arg_{n-1}, ... , then arg_1 on the stack conceptually.
- Saves space for the output value
- Pushes the Program Counter

After returning from the callee function, the caller does the following :

- Pops the saved Program Counter from the stack
- Loads the returned value and pops it from the stack
- Pops all the arguments from the stack

5.2 Callee Convention

Extending the previous example the callee upon receiving the call would do the following :

- Saves $\$rbp$ (Base Pointer Register) of the caller on the stack.
- Saves the current $\$rsp$ (Stack Pointer Register) in $\$rbp$.
- Loads the arguments passed in the stack into temporary variables.

The callee function just before terminating does the following :

- Saves the returned value on the allocated space in the activation frame.
- Restores the $\$rsp$ and $\$rbp$ to the values caller had left it at.
- Jumps back to the saved Program counter of the caller.

6 Three-Address Code

A few conventions we used in our Three Address Code is the following :

- *push* $a\ (i)\$rsp$: This instruction pushes the value of a on the stack with an offset of i bytes from $\$rsp$.
- *load* $a\ (i)\$rsp\ j$: This instruction loads the first j bytes from the stack with an offset of i bytes from $\$rsp$.
- *ret* : This instruction simply changes the current program counter to the saved program counter.