CS335A: Compiler Design

Milestone 3

Harshit Raj · 200433
Kumar Arpit · 200532
Akshan Agrawal · 180061

April 21, 2023

---

1 Dependencies

- python 3
- pip
- Tools used
  - Graphviz
  - ply (python-lex-yacc)
  - argparse
- gcc 11.3.0+

2 How to Run

```
cd $DIR/milestone3
pip install -r requirements.txt
python3 src/parse.py --input /path/to/file.java [--output
 ↪  /path/to/output] [--all] [--verbose]
dot -Tps src/output -o src/output.ps
xdg-open src/output.ps
```

PS: Verbose will pretty print the symbol table, TAC and x86 code on the terminal. Also, the last two lines are for the the graph.
Alternatively, you can also use the Makefile

```
make ARG=x
```

this will run `tests/test_x.java`, optionally followed by

```
make graph
```

to open a postscript file of the tree.
Also, you may use

```
make dev ARG=x
make graph
```

To get the verbose output and parse tree in place of ast.

3 Command Line Arguments:

- input: Specify Input file name
- output: Specify output file, defaults to *javao*
- all: Generates parse tree in dot file instead of AST
- verbose: Prints symbol table, 3ac and the x86 code to the terminal
- help: Shows help

4 Features Implemented

- All basic required except for loop
  - All the basic features such as various integer types and (multidimensional) arrays, all operators, loops (except for loops), if-else, function calls, recursion, object creation, field access and printing have been added and seem to work flawlessly.
  - **For loop**: We were unable to implement for loop as parsing the parse tree for a for statement proved rather challenging. This is because we were doing a DFS, traversing our tree to generate the code however finding were to end our traversal in a node was difficult due to multiple nodes (initialization, condition checking, updation and for body) being present in a same logical block. However, this is not completely game breaking as one may convert any given for loop to while loop easily, either manually or even through scripts.
- **Invoking functions on objects**: We allow invoking functions on objects. The functions invoked recieve the 'this' pointer which allow them to access and modify the fields of objects. Thus, a.foo() calls are supported where a is any object and foo() is its non static function.
- **Do while statement**: Do while statement works as expected, entering the loop at least once even if the condition is false.
- **Break** and **Continue**: Both, break and continue statements work as expected and have been highlighted in our test cases.
- **String**: String printing works. Moreover, string concatenation also works while printing
- **System.out.print()** and **System.out.println()**: Both these calls work, almost fully featured. Printing variables, strings, expressions and return values all work. There is, however, one special requirement while using it. One must wrap expressions and function calls in parentheses "()" and must not wrap a String or variable name in it while printing. For example System.out.println("a"+var+5+(5+5)+(foo(3))).
- **File output**: One may perform file outputting (to an external file) using our fopen, fclose, and fprintf implementations (invoking C calls) by declaring there prototype as in test case 11.

- **Multidimensional arrays**: We support multidimensional arrays of any number of dimensions instead of just 3 required.
- **boolean** and **char**: There is some support for operations on boolean and char types in expressions.
- **Constructor calls**: We support invoking user defined constructors for object creation.

5 Relevant Code Files

- lexer.py: Responsible for lexer specification or lexer definition, contains instructions for a lexer program to recognize and categorize the lexical elements or tokens in a source code file.
- parse.py: Responsible for parser specification or parser definition, contains instructions for a parser program to analyze the structure of a source code file based on the sequence of tokens produced by the lexer program.
- dot.py: Responsible for representation of the structure of a program's source code in the form of a parse tree or syntax tree.
- symbol_table.py: Responsible for providing template for symbol table (data structure) for keeping track of various symbols.
- symbol_tac.py: Responsible for generation of symbol table with the help of DFS traversal of the parse tree governed under the template supported by symbol_table.py file.
- tac.py: Responsible for generation of 3AC code representation of the program.
- utils.py: Responsible for providing certain helper functions (built by us) to the symbol_tac.py file.
- codegen.py: Responsible for converting tac code to x86 code (64 bit system) and thereby generating a low level language code so that desired outputs can be obtained.
- main.py: Responsible for driving all the necessary functions.

6 CSV Header Format

- class: className, symbolType(class), symbolTableName, modifiers, parentClass, interfaces, size
- method: methodName, methodSignature, symbolType(method), symbolTableName, methodParameters, returnType, modifiers, throwsList
- variable: variableName, symbolType(variable), dataType, size, offset, modifiers, numberOfDimensions, arrayOfDimensions
- block: blockName, symbolType(block), symbolTableName

7 Memory Management

- The stack is managed by the caller and callee functions on method invocations.
- On function calls, we need to manually align the stack to 16 bytes as a requirement for gcc verions <12.
- Arrays and objects are allocated on the heap using malloc() calls.

8 Register Allocation

We are using Round Robin (based on time) like algorithm for register allocation.

9 Project Contribution

Name	Roll Number	Email	Contribution
Harshit Raj	200433	harshitr20@iitk.ac.in	100%
Kumar Arpit	200532	arpit20@iitk.ac.in	100%
Akshan Agrawal	180061	akshan@iitk.ac.in	100%