CS335 Semester 2022–2023-II
Project Milestone 2

Aryan Vora (200204) **S Pradeep (200826)** **Yash Gupta (201144)**

1 Usage Instructions

1.1 Compilation and Executing Instructions

To compile the compiler, go into the src directory and run the following make command:

```
make
```

1.2 Executing Options

For the given compiler, we have to run the file using the following command:

```
./a.out <command-linearguments>
```

- -help option is passed as the **first** argument to know how to pass command line arguments to compiler. In this case, no other command line argument is take
- The **first** command line argument is always the input file path if "-help" option is not used.
- -verbose option is used for verbose error displaying
- -output option is used to set the name of the output dot file. An assign symbol is used after -output before mentioning the filename. This is done using the following syntax:

  ```
  -output=<filename.dot>
  ```

  For example "-output=hi.dot"

1.3 Assembly Code Options

The x86 code is generated in the 'assembly.s' file. To run the code, follow the following instructions:

```
gcc -c assembly.s -o assembly.o
gcc assembly.o -o assembly
./assembly
```

2 Assumptions and Explanations

2.1 Milestone 1

We have assumed that and AST for a complex program means that we display the biggest tree without any ambiguity. An AST can have ambiguity in creation especially in cases where there are more than 3 nonterminals/terminals on the right side of a production. In this case, there is a confusion of whether to incorporate the second or the third child of the node in the parse tree into the value label of the given node.
A structure for the AST node is defined in ast.h in src directory.
We have modified the Java grammar so that it is acceptable by a LALR(1) parser. We have used flex and bison in C++ for this task.

Preprint. Under review.

2.2 Milestone 2

The symbol table is stored in the file 'symbolTable.csv' and the three address code is stored in the file 'threeAC.csv'.
Symbol table functionalities are defined in 'symtab.h' and three address code functionalities are defined in 'threeAC.h'.
We have not computed the values of expressions, neither have we implemented a dimensionality check for array accesses. We do not have support for function and operator overloading. We also do not have support for imported libraries and files. We have not added support for inheritance.
We have not provided support for the evaluation of expressions and invocation of methods inside the println function.
We have not supported the declaration of variables inside the for loop header.

2.3 Milestone 3

We add functionalites included in the calling sequence and the return sequence for function calls. This includes pushing and retrieving the function arguments from the stack. We also include functionalites for allocating and deallocating memory space in the stack for local data and temporary variables. We store offsets for the arguments in the symbol table and reference them when needed. We also store the return address (PC+4) in the stack so that it can be used to set the value of the RAX register for proper return. We properly update the Base Pointer Register and Stack Pointer Register when we transfer control from the caller to the callee.
We define the following:

- BP = Address of Base of Stack
- SP = Address of Top of Stack
- PC = Program Counter
- obj_ref = Reference of object
- return_addr = Address to be returned to after return
- beginfunc marks the beginning of a function
- endfunc marks the end of a function
- allocate allocates space in the stack by an amount which is specified
- deallocate deallocates space in the stack by an amount which is specified
- popparam is used to pop the object reference parameter from the stack. This is a default last parameter for all functions.

Our tasks broadly included the following:

1. Implemented a flag in the symbol table to specify whether the variable is an argument or not
2. Added offsets from BP of each variable in the symbol table
3. Calculated and stored the cumulative size of all arguments to a function
4. Calculated and stored the amount of memory that needs to be allocated and deallocated during a function call
5. Correctly identify the return address of a function
6. The callee must push the value of the caller's base pointer on the stack
7. The callee must update the value of the base pointer to it's own activation record

2.4 Milestone 4

Our solution has two main parts, one is to get the registers for variables, the other is to convert the three address code to x86.
To get the registers for variables, we implement two ideas, one is to get the location of the variables in the stack. This is done by incorporating the offsets from the base pointer. The second idea is of a map between variables and registers they are stored in.

To convert the TAC to x86, we analyse the tokens in the TAC and appropriately generate x86 code for the same, while using the aforementioned methods to handle variables.
We have support for the following features:

- Arithmetic operations (*,/,+,-,=)
- Relational operations (<,>,==,!=,<=,>=)
- Logical operations (, ||, , |,)
- Shift operations («, »)
- Loops (While, For)
- Branch (If, Else)

We have not been able to implement array references. We have also implemented basic recursive programs. We have added the imported libraries to the symbol table. We were not familiar on how to link these libraries and so haven't incorporated it in the TAC and x86 codes. Nested for and if loops also work in our implementation. As a manual change, rarely the push statement for the argument before the System.out.println causes a segmentation fault and has to be commented out.

3 Contribution

Roll Number	Member	Contribution
202004	Aryan Vora	33%
200826	S Pradeep	33%
201144	Yash Gupta	33%