Milestone 3

Indian Institute of Technology Kanpur

Semester II, 2023-24

Course: CS335
Instructor: Swarnendu Biswas

Students:

- Paras Sikarwar - 210699
- Ravija Chandel - 210835
- Siddheshwari Ramesh Madavi - 211036

1 Tools Used

- **Flex** is used for writing the lexer.
- **Bison** is used for writing the parser.
- **Dot** is used for creating the Abstract Syntax Tree.

2 Features implemented

The following language features are supported end-to-end in our implmentation of the compilation toolchain to generate x86_64 assembly from Python code:

- Primitive data types - int, float, string, boolean
- 1D list
- Basic operators:
  - Arithmetic operators: +, -, *, /, //,%, **
  - Relational operators: ==, !=, > , >, >=, <=
  - Logical operators: and, or, not
  - Bitwise operators: &, ||, ∼, ˆ, <<, >>
  - Assignment operators: =, +=, -=, *=, /=, % =, **=, & =, || =, ˆ=, <<=, >>=
- Control flow via if-elif-else, for, while, break and continue
- Iterating over ranges specified using the range() function.
- Recursion
- The library function print() for only printing the primitive Python types, one at a time
- class constructor

3 Functionalities

The following functionalities are implemented in addition to Milestone 1 and Milestone 2:

- a translator to generate x86_64 assembly from the three-address code output in Milestone 2

4 Testcases

We have provided 8 testcases in the tests folder which encompass all the language features implemented by us.

To run the testcases together:

$ make

$.\run_tests.sh

This will give all dot files in outputDOT folder, all tree files in outputAST folder, all 3AC files in output3AC folder and all assembly files in outputASM folder.

5 Compilation and Execution Instructions

```
$ make
```

```
$ ./m3.out -input inputfilename.py -output outputdigraphfilename.dot -output3AC output3ACfilename.txt -outputSymTab outputSymTabfilename.csv -outputASM outputASM.s -verbose
```

```
$ dot -Tpdf outputfilename.dot -o graph.pdf
```

After running these instructions, the user will get the digraph representation of the input program in outputdigraphfilename.dot, the abstract syntax tree of the input program in graph.pdf, the dump of the 3AC of the functions in text format in the file output3ACfilename.txt, a dump of the symbol table of each function in the CSV file outputSymTabfilename.csv and the x86_64 code of the input program in outputASM.s.

6 Command Line Options

The following command line options have been implemented:

- **-help**: Gives instructions on how to use the parser and generate the parse tree and get the 3AC intermediate representation of the code
- **-input**: Used to pass the path to the input file containing the Python code to be parsed.

  If path to input file is not passed, a warning message will be displayed and input will be parsed from a default file ”input.py”.
- **-output**: Used to pass the path to the file where the digraph output will be displayed.

  If path to output file is not passed, a warning message will be displayed and the output will be displayed in the default file ”output.dot”.
- **(NEW) -output3AC**: Used to pass the path to the file where the 3AC output will be displayed.

  If path to output file is not passed, a warning message will be displayed and the output will be displayed in the default file ”3AC.txt”.

- **(NEW) -outputSymTab**: Used to pass the path to the file where the symbol table output will be displayed.

  If path to output file is not passed, a warning message will be displayed and the output will be displayed in the default file "symtab.csv".

- **(NEW) -outputASM**: Used to pass the path to the file where the x86_64 code for the input program will be displayed.

  If path to output file is not passed, a warning message will be displayed and the output will be displayed in the default file "asm.s".

- **-verbose**: Generates a file "verbose-output.pdf" which contains detailed information about the parser's output stored in "parser.output"

  Usage: ./m3.out -help -input input.py -output output.dot -output3AC 3AC.txt -outputSymTab symtab.csv -outputASM asm.s -verbose

  The tags -help and -verbose are optional and can be used as and when required.

7 Generating x86_64 assembly code

The x86_64 assembly code is created by using the struct asm_ins to store an assembly instruction along with all other relevant information, the struct prog_elem for storing the name and the offset of the base pointer for the various elements in the input program and the struct prog_elem_info for storing the size and parameters of the element and other relevant functions for checking the important components in the program. We use a function gen_asm_code to finally integrate the generation of all assembly code of all the program elements.

We have added the files "make_x86.hpp" and "make_x86.cpp" containing the defintions of the created structs and the functions and the functionalities to create assembly code from the generated quads. We have integrated this in the main function in the file "parser.y"

We have added the following additional asm files to generate properly formatted assembly code: "print_func.s", "print_str.s", "allocmem.s", "len_func.s"

8 Major Changes to 3AC Code

In Milestone 2, we had added relational jumps in the 3AC form, we have changed that to absolute jumps in Milestone 3 for convenient x86_64 assembly generation.

9 Manual Changes to the Assembly Code

No manual changes are required.

10 Final Effort Sheet

Name	Contribution
Paras Sikarwar	35%
Ravija Chandel	30%
Siddheshwari Ramesh Madhavi	35%