CS-335

README

MILESTONE 3

Group 12

Aditya Ajmera (210056)

Maurya Aryan Swaminath (210595)

Depanshu Sahu (210316)

1 Output Files

After executing the `./main`, there will be an output folder. It will contain the `".CSV"` of the symbol table for each function in a different file with the same name as the function, and the **3AC** instruction is stored as **output.txt** in the same output folder (file name is specified by the user in the input). The dot file produced will also be stored in output folder by default as `output.dot` or as specified by user in the input command line. The folder is `milestone3/output`. The **x86 assembly code** will be stored in the output folder as **output.s** (name specified by the user in the command line). The folder is automatically created on compilation and files are saved after execution.

The Symbol tables are saved as : **function_name.csv**
The 3AC is saved as : **output.txt**
The assembly code is saved as: **output.s**
The dot code is saved as: **output.dot**

Note: You need to delete the whole `output` folder each time before running a new input.

Note: `make clean` also deletes the output folder.

Each csv file stores the name of function, the number of parameters in that function and the return type of that function. It further stores the function parameters and the function body variables directly declared inside the scope of that function. If **n** is the number of parameters of that function, then the first **n** entries are the parameters of the function. The variables have their types, lexeme, token and line number stored in the table.

The submission contains 5 test cases in the `./milestone3/testcases` folder to test different aspects of the python language.

2 Required Features

- Primitive data types (e.g., int, str, and bool)
- 1D list of primitive types

- Basic operators:
  - Arithmetic operators: +, -, *, /, //, %, **
  - Relational operators: ==, !=, >, <, >=, <=
  - Logical operators: and, or, not
  - Bitwise operators: &, |, $\wedge, \sim, \ll, \gg$.
  - Assignment operators: =, +=, -=, *=, /=, //=, %=, **=, =, —=, $\wedge$=, $\gg$=, $\ll$=
- Support for recursion
- Support the library function print() for only printing the primitive Python types, one at a time
- Support for classes and objects, including multilevel inheritance and constructor
- Control flow via if-elif-else, for, while, break and continue
- Support iterating over ranges specified using the range() function
- Associativity of operator

3 Optional Features

- Iterating over lists. (`for i in arr: ...`)
- Passing string and lists as function argument.
- Returning string and list from function.
- Implemented `len(array)`.
- String Comparison (<, >, ==, ! =)

4 How to Run

4.1 Compilation Instructions

After extracting the zip, open the terminal in the folder and execute:

```
cd milestone3/src
```

Now we are in the same directory as the make, lexer and parser files. Now run

```
make clean
```

The above command cleans up the previously created files by lexer and parser.

```
make compile
```

This command compiles the lexer and parser and makes the `main.exe` file.

4.2 Command Line Options

- **- -input** Add this flag to specify an input file to the parser. This is a required flag.
  Example:

  ```
  ./main --input=input.py
  ```

- **- -output** Add this flag to specify the output name of the dot, assembly and 3AC file. This is optional and the default output is `output.dot`,`output.s` and `output.txt`.
  Example:

  ```
  ./main --input=input.py --output=result
  ```

- **- -help** Use this flag to know about the rules to run the commands.
  Example:

  ```
  ./main --help
  ```

- **- -verbose** Use this flag to apply the debug mode of bison parser. This is an optional flag.
  Example:

  ```
  ./main --input=input.py --output=result --verbose
  ```

4.3 Execution Instructions

To execute, run the following commands:

```
./main --input=<input_file_name> --output=<output_file_name>
```

For example:

```
./main --input=input.py --output=result
```

There will be a file produced named `result.s` (for the above example), to compile the assembly code, run the command

```
gcc -g -no-pie <input_file_name> -o <output_file_name>
```

For example:

```
gcc -g -no-pie result.s -o result
```

To run the executable file produced simply run:

```
<output_file_produced>
```

For example:

```
./result
```

Now generate the AST pdf from dot file using:

```
dot -Tpdf <output_file_name> -o AST.pdf
```

In our example case `<output_file_name>` is replaced by `result.dot`:

```
dot -Tpdf result.dot -o AST.pdf
```

5 Changes in 3AC

- Modified the 3AC of prologue part before a function call.
- Modified the self keyword 3AC to implement correctly.
- Created 3AC instructions for string support.
- Created 3AC instructions for comparison operators support.
- Modified 3AC instructions for print.
- Modified declare_array to alloc_mem.

6 Changes in Assembly File

No changes were made in the assembly file.

7 Required Features Not Supported

All Required features are supported.

8 Effort Sheet

Name	Roll Number	Contribution
Aditya Ajmera	210056	33.33%
Maurya Aryan Swaminath	210595	33.33%
Depanshu Sahu	210316	33.33%

9 Tools Used

- We have used the lexical specifications and the grammar for Python 3.8.
- We have used the GraphViz tool to visualise AST for our grammar.
- We have written the lexer part in Flex and the grammar part in Bison.
- Make utility is used for automatic tracking for files and compilation.