CS 335 Semester 2023–2024-II: Milestone 3

A. Atulya Sundaram, 210001
Anuj Singhal, 210166
Goutam Das, 210394

Compilation and Execution Instructions

The `milestone3` directory has 4 subdirectories

- `src`: containing the project's source files, i.e. the scanner(`lex.l`), parser(`parser.y`), `symtab.cpp`, `makefile`.
  To compile the code, you can simply use the given `makefile`, make sure that `flex`, `bison` and `g++` are installed on the system

  ```
  $ make all
  ```

- `doc`: Includes documentation of the project (i.e. this file)
- `tests`: Contains all the test python files that to be evaluated
- `out`: Contains all the output files including the symbol table (`symtab.csv`),the 3AC text file (`3AC.txt`), the x86 assembly file (`x86.s`) and the final executable generated by the assembler (`<test_name>`)

The implementation has been tested to successfully compile and execute on

- `flex 2.6.4`
- `bison 3.8.2`
- `g++ 11.4`

`make` command creates an executable named `parser`, to parse a file using parser, the usage instructions are as follows:

```
USAGE:
[-input <path-to-inputfile>]: specify the file path to parse, if not given, takes
input from stdin

[-csv <path-to-outputfile>]: specify the file path to write the symbol table csv,
if not given, defaults to symtab.csv

[-tac <path-to-outputfile>]: specify the file path to write the 3AC code, if not
given, defaults to 3AC.txt
```

```
[-asm <path-to-output-asm>]: specify the file path to write the assemble code,
if not given defaults to x86.s

[-verbose]: prints verbose messages to stdout, error messages are anyways printed

[-help]: print usage instructions
```

The parser, generates the required assembly file.

Once you have the assembly file, you can execute the following to generate the executable:

```
$ gcc -c <path-to-assembly-file> -o <path-to-[.o]-file>
$ gcc <path-to-[.o]-file> -o <path-to-executable>
```

Finally, once, we have the executable, to execute the binary:

```
$ ./<relative-path-to-binary>
```

Thus, the end-to-end commands to be executed, from downloading the compiler to using it to compile a program in a file named test.py that is present in the tests folder mentioned above is the following.

```
$ git clone https://git.cse.iitk.ac.in/anuj1/cs335_course_project.git
$ cd cs335_course_project
$ git checkout milestone3
$ cd milestone3/src
```

Now, to build the compiler and compile and execute the file test.py:

```
$ make all
$ ./parser -input ./../tests/test.py -tac ./../out/tac.txt -asm ./../out/x86.s
$ gcc -c ./../out/x86.s -o ./../out/x86.o
$ gcc ./../out/x86.o -o ./../out/exec
$ ./../out/exec
```

Features handled

The list of features we have handled are the following:

- Primitive Data types : int,bool,str
- 1D List
- Basic Operators:
  - Arithmetic operators: +, -, *, /, //, %, **
  - Relational operators: ==, !=, >, <, >=, <=
  - Logical operators: and, or, not
  - Bitwise operators: &, |, ^, ~, <<, >>
  - Assignment operators: =, +=, -=, *=, /=, %=, **=, &=, |=, ^=, <<=, >>=

- Control flow via `if-elif-else, for, while, break` and `continue` and supporting `range`
- Recursion
- The library function `print()` for printing the primitive types, `int, str, bool` one at a time
- We have explicitly handled the primitive type `bool` while printing, so as to enable greated code usability.
- Support for classes, attributes access and method calls and constructors
- Support for type checking, while initialization and function calls
- There are no manual changes that need to be done to run the generated assembly file with `as` or `gcc`

Features not supported

- Support for inheritance: A major challenge with inheritance was to maintain the size, attributes and methods of parent class, which was giving a lot of errors we could not correct

3AC IR Generation (Modified)

We have handled the generation of the 3AC along with the parsing and thus generate it in the first scan. The 3AC generated is dumped in a text file called `3AC.txt` for each program, overwriting previous content of the file in case it already exists.

Note that we have added an extra `#` in front of some of the instructions, as it can be an identifier in the Python program.

The `3AC` code contains the following forms, note that `x,y and z` can be either variables used in the program or temporaries of the form `#t_i`:

- `beginfunc <function_name>` : Beginning of the function with `<function_name>` following the format prescribe above.
- `x = #popparam` : Pop a parameter entry in a function from the stack and assign it to a `x`.
- `x = y <op> z` : Operating on `y` and `z` with the variable `<op>`, which can be an arithmetic operator or a relational one.
- `param x` : Push the parameter `x` onto the stack.
- `call <func_name> <no_of_param>` : call the function `<func_name>` which contains the number of paramters, which have been pushed into the stack by `param`.
- `*(x) = y` : assigning the value of `y` to the address of `x`.
- `x = *(y)` : dereferencing `y` which is a pointer.
- `if_false x goto @Labeli` : Go to the label with number `i` if `x` is false
- `goto @Labeli` : Unconditionally go to the label with number `i`

- `label:@Labeli` : Form for the label with number `i`
- `shiftpointer +x`: Shift the pointer of the stack up by x bytes (before calling the function and pushing in parameters.
- `shiftpointer -y` : Shift the pointer of the stack down by y bytes (used after calling the function).

The newly added 3AC codes are:

- `x = #space <space-size` : In order to support the shifting the stack to accommodate lists and classes
- `x = #retval` : To store the return value of a function in to the variable/register `x`.

Effort Sheet

Table 1: Effort Sheet

Contributions	**Anuj**	**Atulya**	**Goutam**
Contributions	33.33	33.33	33.33

Each person worked in each area, along with testing and debugging for all the milestones. All the code work was done concurrently on a shared system.