CS335: Milestone 3

Aman	Aniket Suhas Borkar	Siddharth Kalra
210104	210135	211032

April 19, 2024

1 Compilation and Execution Instructions

The lexical analyser and parser have been written in `C++` using flex and bison respectively. In order to automate the compilation process, a Makefile has been created. The code is structured as follows:

- `lexer.l` contains the flex specification.
- `parser.y` contains the bison grammar rules.
- `3ac.h`, `symbol_table.h`, `x86.h` and `symbol_table.cpp`, `3ac.cpp`, `x86.cpp` contain helper function declarations and definitions respectively, and are included in both the flex and bison specifications.
- `Makefile`

The code makes use of the `C++` STL extensively by including the `<bits/stdc++.h>` header file, and must be compiled using `g++`. The parser supports the following execution options:

- `-input <filename>`: This option is used to give the input `.py` file to be compiled. By default, the input is read from `stdin`.
- `-output_tac <filename>`: This option is used to specify the file to which the 3AC is to be stored. By default, the output is saved to a file named `"tac.txt"`.
- `-output_x86 <filename>`: This option is used to specify the file to which the x86 code is to be stored. By default, the output is saved to a file named `"x86.S"`.
- `-verbose`: This option prints all the steps of the parsing process to `stderr` and provides debugging information.
- `-help`: This option prints out the usage instructions.

To run a program written in python in file named `trial.py`, the following steps need to be followed in order:

- Running the makefile:
  `make`
- Genetating the tac, x86 code and .csv files (for symbol tables):
  `./parser -input trial.py -output_tac trial.tac -output_x86 trial.S`

- Using gcc to assemble the generated assembly code:
  `gcc -o trial trial.S`
- Running the binary:
  `./trial`

2 Required Features supported

Our implementation supports all the required features for the project. They are also listed below:

- Primitive data types (e.g., int, float, str, and bool)
- 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 . Iterating over ranges specified using the range() function.
- 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 constructors.
- Methods and method calls

3 Optional features

Following optional features are also supported:

- Multi-dimensional lists
- General atom expressions like `myObj.fn().myList[i].myVar`.
- There is no restriction of declaring any variable, object, list or string before using them. Hence, the following constructs are valid:
  - `myList:  list[A]=[A()] # Where A is a class`
  - `print(([1,2,3])[1]) # prints '2'`
  - `myFunc([1,2,3], "abc", 1+2, B()) # myFunc is a function with arguments: list[int], string, int, obj of B. B is a class`

4 Other specifications

- Changes made in 3AC:
  - Temporaries are now named as `#t{n}` as opposed to `t{n}` in milestone 2.
  - Stackpointer is now manipulated before `'param'` instruction.
  - Strings are now given labels at the top of 3AC and labels are used as placeholders.
- No manual change is required in the generated assembly to run it using gcc.
- Function names cannot be `L{n}` or certain other labels used to support library routines like print.
- The compiler checks that control does not reach end of function without a return statement in functions returning non-None values. However control flow analysis is not implemented, and this check is simply performed by forcing a return statement at the outermost level in a function.

5 Implementation details

Following are some of the implementation details for generating x86.

- Stack is used for passing the function arguments
- Stack is used for storing the temporaries generated for 3AC.
- `rax` register is used for storing the return value.
- Lists and objects are allocated in heap, whereas strings are stored in data segment.
- Stack pointer is aligned to 16 byte boundary before every function call to meet ABI requirements.

6 Wrapper Script

The submission also includes a wrapper script named `test.sh` in the `milestone3/scripts` directory that will do the following tasks for sample python codes present in `milestone3/tests`:

- Create 3AC
- Create symbol table CSV dumps
- Create assembly code
- Assemble the assembly code using gcc to create executable
- Execute the binary and store the output.

All these outputs would be generated in the `milestone3/out` directory. Run the script using the following command from the `milestone3/scripts` directory:

```
cd milestone3/scripts
chmod +x test.sh
./test.sh
```