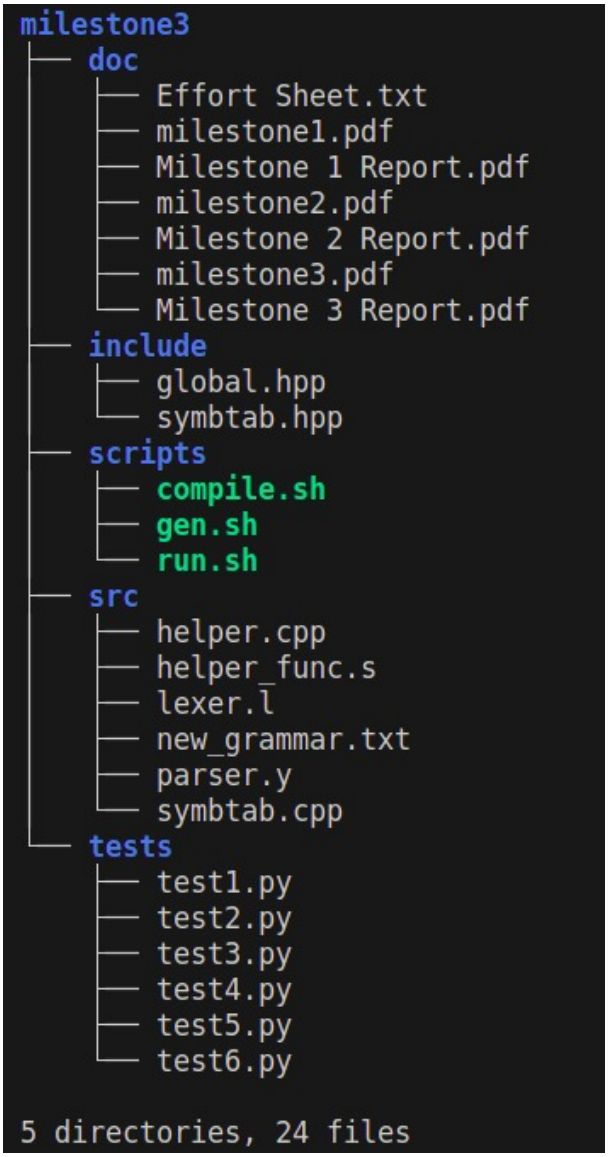
Pythoneers: Milestone 3 Report

Chitwan Goel	Shrey Bansal	Talin Gupta
Roll No. 210295	Roll No. 210997	Roll No. 211095

Code Base Structure



Requirements

To run the program, you will need the latest versions of `bison` and `flex`.

Usage

We have provided a script `run.sh` to run the program in the `scripts/` directory. The output `csv` files (for symbol tables), `txt` file (for 3AC) and `s` file (for x86) will be generated in the `src/` directory. If no output file name is mentioned, the default output file is `code.s`. Run the following commands:

```
cd scripts
./run.sh --input=<input_file_path> [options]

The following options can be used
--output=<output_file_path>     : Output assembly file specification
                                  (extension should always be .s,
                                  default is code.s)
--execute                       : Run the generated executable
--symbtab                       : Print the symbol tables in separate
                                  .csv files
--no-save-temps                 : To remove temporary files after
                                  execution (by default they are
                                  saved)
--help                          : Instruction regarding usage
                                  instructions and options
--verbose1                      : Prints our custom debug statements
                                  while reducing a production
--verbose2                      : Prints the complete stack trace of the
                                  parser execution
```

Providing the input file is necessary via the `--input` flag. The input file must have `.py` extension. The input file path should be relative to the `src/` directory. The program will terminate and throw an error if the input file is not specified.
If an output file is provided via the `--output` flag, then the extension of the file must be `.s`.
If you pass the flag `--execute` then the generated executable will be executed. Otherwise, the executable will be generated in the `scripts/` directory as well as the `src/` directory. Alternatively, you can use the script `compile.sh` to generate an executable `parser` and run it as follows:

```
cd scripts
./compile.sh
./parser --input=<input_file_path> [options]
gcc -c code.s -o code.o
gcc code.o -o code
./code
```

Optional Features Included

We have supported the following extra features:

- Chaining of object attributes (eg. `obj.a.b`, `obj.a.b()` and `obj.a().b`)

- Implicit and Explicit Line Joining
- Multiple function calls in a line, for example: `a(b(), 1)` where a and b are global functions. This has also been supported for class functions.
- List index out of range check (if an index is accessed that is greater than the size of a list)

Printing Symbol Tables

Use the flag `--symbtab` to generate symbol tables. For each scope, the symbol table has its own CSV. The CSVs appear in `src/` directory.

- Global symbol table (`st.csv`): Has sections for classes, functions, and variables. The token specifies whether it's a class, function, or variable. The return types of functions are also mentioned here. Parent classes are also mentioned in the enclosed parenthesis.
- In the global symbol table, by default, there are 4 functions for print, each taking different arguments. There is also the len function. Range has not been implemented as a function; we have treated it as a keyword.
- Class symbol table (`st_<class_name>.csv`): Has entries for functions and variables within the class (names, types (or return type in case of functions))
- In the class symbol tables, we don't store the attributes and methods of the parent symbol table. Instead, we store only the pointer to the parent class, and at the time of usage of an attribute or invocation of a method of the parent class, we do a lookup in the class inheritance hierarchy.
- We have assumed that the attributes of a child class are (implicitly) laid after the attributes of the parent class. Hence, the offsets of the attributes in the child class start from the size of the parent class.
- Function symbol table (`st_<class_name>_<func_name>.csv`): Has entries for variables and formal parameters (name, type, and line number).

Three Address Code Generation

- The instruction number indexes each instruction/statement in the 3AC.
- Temporaries start with `__t`.
- Name mangling has been used, @ is used as the separator.
- `call strcmp` in the 3AC refers to the GNU implementation of strcmp, and `call print` refers to the syscall to print to stdout.
- For `jump` / `goto`, we use the instruction number as labels.

Changes to 3AC from Milestone 2

- Earlier, we had 3AC like `__t1 = "Hello"` for strings. Now, we store the address of the string from the global segment into a temporary. For this, we have maintained a map for labels of string, and using leaq instruction for getting addresses.
- Some changes for printing strings. Earlier we were simply calling print, now there is a separate quad.
- Added an extra quad to detect when to pop registers when calling functions with None return type.

Steps followed for making x86 from 3AC

- We had the 3AC stored in form of vector of quads.
- All the variables(including temporaries) in a function were assigned an offset for the stack. That is, we pushed all variables and temporaries on the stack. And for using them, we do load and store.
- Basic blocks were made. Targets of goto instructions set up by making them as labels in x86.
- We then do template matching, and for each kind of quad emit the corresponding x86 instructions.
- There is a file by the name of helper_func.s which contains implementation of allocmem, strcmp and print functions.

Supporting strings

- We store strings in the data section, with labels as `LC0`, `LC1`, ...
- Using `leaq`, we get address of strings and assign it to corresponding temporaries.
- This address is the one used for all purposes, including relational operators on strings.