SANKLAK: A python Compiler

Divyansh	Rajeev Kumar	Sandeep Nitharwal
210355	210815	210921

March 2024

1 Running Instructions

This compiler employs `Flex` and `Bison` for the implementation of Lexer and Parser, respectively, while semantic actions, helper functions, and linkings are done in `C++`. The following script will install all the required packages:

```
sudo apt-get update
sudo apt-get install bison
sudo apt-get install flex
sudo apt-get install graphviz
```

Move inside the `src` directory, and then run the `Makefile` using the command `make` to generate the executable `python`. The command line provided by our program is as follows:

1. `-h, --help` : Prints the manual page for supported arguments
2. `-i, --input` : Takes the input program file
3. `-o, --output` : Takes the output file to redirect the `DOT` code file
4. `-v, --verbose` : Prints the grammar production logs

All the arguments mentioned above are optional. The input program can also be passed directly without `-i` or `--input`, the only condition in this case is that the input file must have the `.py` extension. By default, the output file generated is `file.dot` and `verbose` is false. After the `DOT` file has been generated, we can draw the graph using the `dot` utility.
The csv of symbol is under the file named `dump_symbol_table.csv` and the generated `3AC` is in the file named `3ac.txt`;

```
$ cd src
$ make
$ ./python test.py
$ ls
>>> ... file.dot dump_symbol_table.csv 3ac.txt
```

2 Grammar Augmentation

We have followed the `version 3.8` of the `python` grammar mentioned in the python documentation. The documentation here uses a few symbols/operators in their specifications that are not supported by `bison`, for eg: `+/*` operators. These were tackled by providing additional productions.

After pruning those productions that were not required by our milestone specification, the grammar on compilation creates multiple conflicts. The primary reason for those conflicts was that many non-terminals were going to single other non-terminals since our productions of that non-terminal were removed.

3 AST Specification

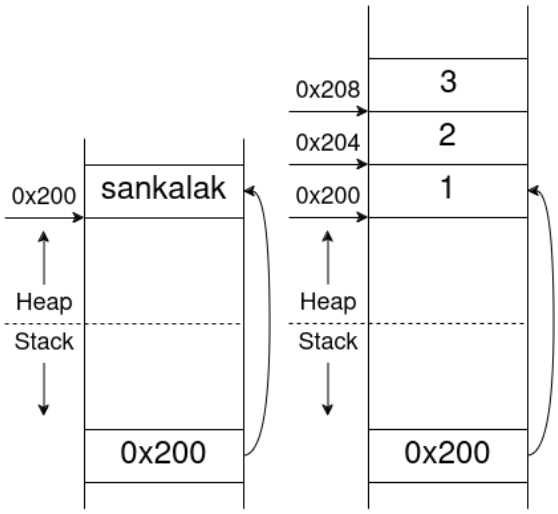
After the creation of `parse tree` from the parser, we compress the parse tree in order to get the `AST` using the following rules:

1. Removing the occurrence of terminals that were not needed for deterministically deciding the actual code of the grammar.
2. The parse tree has nodes for all the non-terminals that occurred during the grammar reduction but doesn't have any significance in the ast tree since to produce other expressions eventually.
3. We have preserved certain key non-terminals for eg: `funcdef, if_stmt,etc`, in order to make the ast tree more readable.

4 Storing variable-sized data types

The approach we have used to store variable-sized data is as follows. Consider the example of a string declared as

```
$ name: str = "sankalak"
```



(a) Storing string in heap (b) Storing list in heap

Figure 1: Storing variable sized data types

We have stored the string in heap and the pointer to the address in the heap in the stack. As in the figure 1(a), the actual string "sankalak" has been stored at address 0x200 in heap. And, the address 0x200 has been stored in the stack. Similarly, we have stored the list as shown in Figure 1(b).

At the start of each function, we have performed the following actions related to the prologue.

```
$ push %rbp
$ mov %rsp, %rbp
$ sub %rsp, N
```

where N is the number of bytes (4-aligned) required for storing local variables.

Similarly, at the end of each function, we have performed the following actions related to the epilogue.

```
$ mov %rbp, %rsp
$ pop %rbp
```

5 3AC instructions

Following are the 3AC instructions that we have used in our compiler.

- `t_1, t_2`, and so on are registers.
- `L_1, L_2` and so on are labels.
- `value at t_1` means the value at the address pointed by `t_1`. In C, it is equivalent to `*t_1`.
- `begin_func` signifies the starting of a function definition and `end_func` the end
- `begin_class` signifies the starting of a class definition and end_class the end
- `call f` means call function `'f'`
- `push_param` pushes parameter onto the stack of callee function
- `pop_param` pops out the parameter from the stack and saves it into a register
- `push_obj_param` and `pop_obj_param` are nothing different than `push_param` and `pop_param` except that they represent push and pop operations for an object
- `Goto L_1` means go to `label L_1`
- `ifZ t_1 Goto L_1` means if `t_1` is not `true` then go to label `L_1`
- `mem_alloc n` means allocate n bytes of memory on the heap and return the start address of the allocated space
- `str_alloc "hello world"` means allocate the required amount of memory (`12` in this case) on heap and store the string
- return at the end of a function makes the instruction pointer and stack pointer restored so that the caller function continues its execution
- The method (say `method_m`) of a class (say `class_A`) have been labelled as `%% class_A method_m`