SANKLAK: A python Compiler

Divyansh 210355   Rajeev Kumar 210815   Sandeep Nitharwal 210921

April 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` for dot file, `dump_symbol_table.csv` and `final.s` for the assembly file.

```
$ cd src
$ make
$ ./sanklak test.py
>>> <program_output>
```

2 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.
- `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
- `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 `A_method_m`

3 Features Supported

Following are the features supported by our compiler.

- Primitive data types: int, bool, and str
- 1D list: list of primitive data types and objects
- Operators:
  - Arithmetic operators: +, -, *, /, //, %, **
  - Relational operators: ==, !=, >, <, >=, <=
  - Logical operators: `and`, `or`, `not`
  - Bitwise operators: &, |, ˆ, ˜, <<, ¿¿
  - Assignment operators: =, +=, -=, *=, /=, //=, %=, **=, &=, |=, ˆ=, <<=, >>=
- Control flow via if-elif-else, for, while, break, and continue
- Iterating over ranges specified using the range() function: range(end), range(start, end) where start and may be variable or constant
- Function (including recursion)
- Library function print() for only printing the primitive data types, one at a time
- Classes and objects, including multilevel inheritance and constructors.
- Methods and method calls, including overloading of the `__init__` constructor function.

4 Command lines and test cases

We have provided the following testcases as a demo for our compiler implementation:

1. `test1.py`: This implements the algorithm to find maximum sum of adjacent elements in an array.
2. `test2.py`: Applies merge sort on a list of strings.
3. `test3.py`: Checks the inheritence and other properties for the class
4. `test4.py`: Implements a class which checks if a list is reverse/palindrome or not and also finds the maximum value in it.
5. `test5.py`: Implements a list of class objects and iterate over them to find the object with maximum and minimum field.

5 Manual support & Remarks

- No need to manually do anything. Just run the instructions provided above, and you will have your executable.
- We have not missed any feature listed in the milestone specification.

6 Effort sheet

Members	**Efforts**
Divyansh	`33.3%`
Rajeev Kumar	`33.3%`
Sandeep Nitharwal	`33.3%`

7 Miscellaneous

This project was a beautiful journey. Although it was frustrating sometimes, we didn't stop, and the result is in front of you, our `Sankalak`. All these sleepless nights are worth it now. All of us have contributed equally and this is something that we all are very proud of. This project taught us that building something as big as a compiler actually requires the effort of each and every team member. From building the lexer to the code generator, we have evolved much as programmers. Not only did we learned a lot of things related to writing code and debugging, but we also learned code management. As our codebase expanded, finally reaching $\sim$ 5000 lines of code, managing codes became very difficult. We realized a lot of things, like why the notion of `.h` and `.c` was introduced, why header guard was introduced, and so on. There are a lot of things that we want to write about at this moment, but let's wrap it up here.

`Ever tried. Ever failed. No matter. Try again. Fail again. Fail better. - Samuel Beckett`