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.

```
$ cd src
$ make
$ ./python -i test.py -o output.dot
$ dot -Tpdf output.dot -o ast.pdf
```

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.