CS335: Milestone 1

Aman	Aniket Suhas Borkar	Siddharth Kalra
210104	210135	211002

3rd March 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.
- `node.h` and `node.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 <filename>`: This option is used to specify the file to which the DOT specification for the AST is to be stored. By default, the output is saved to a file named `"trial.out"`.
- `-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.

In short, to run the parser on a file `"trial.py"` and create the AST, the following commands must be executed:

```
make
./parser -input trial.py -output trial.out
dot -Tpdf trial.out -o trial.pdf
```

2 AST from Parse Tree

The parser is constructing the parse tree. Certain adjustments are made to transform it into an Abstract Syntax Tree (AST) based on the following principles:

1. **Elimination of Redundant Nodes:** Nodes corresponding to NEWLINE, INDENT, and DEDENT tokens are omitted to enhance the conciseness of the parse tree.

2. **Simplification of Single Productions:** Nodes resulting from single productions, such as $(A \to B \to C)$, are condensed to $(A \to C)$ as needed, streamlining the structure.

3. **Operator Restructuring:** Operators are elevated one level above the operands, a modification aimed at facilitating the subsequent evaluation of expressions.

Terminals are enclosed in rectangular boxes to distinguish them from non-terminals. A color-coding scheme is also employed for an improved visualization of the AST shown below:

Operator Keyword Delimiter Name String Literal Number Type Name

3 Wrapper Script

The submission also includes a wrapper script named `test.sh` in the `milestone1/scripts` directory that will create DOT files for the AST of the sample python codes present in `milestone1/tests` and create PDF files containing the ASTs. All the outputs would be generated in the `milestone1/out` directory. Run the script using the following command from the `milestone1/scripts` directory:

```
cd milestone1/scripts
chmod +x test.sh
./test.sh
```

4 References

The references for the extra testcases which have been taken from the internet are as follows:

1. Editorial of problem: "Maximum subset XOR" in python for `test6.py`

2. Editorial of problem: "Robots" in python for `test7.py`

3. Editorial of problem: "Solve the Sudoku" in python for `test8.py`

4. Editorial of problem: "Maximum Rectangular Area in a Histogram" in python for `test9.py`