1 Recursive Descent – LL(1) Grammar

The grammar given on INGInious needs to be modified in order to not be left-recursive. The following presents an equivalent grammar which is LL(1) (its first-sets and follow-sets, as well as the parsing table are given in the code file):

E ::= TE'

E' ::= or TE' | ε

T ::= FT'

T' ::= and FT' | ε

F ::= not F | (E) | id

2 Programming directly in Java bytecode

Using `CLEmitter`, it is quite simple to create a "hand-written" class file. In order to specify the output directory for the generated file, one can use the `destinationDir` method. A `CLEmitter` uses an `ArrayList` to store the various instructions that are needed. Similarly to how this was done in the provided examples of `CLEmitter` usage on the lecture slides and in the `tests/clemitter` folder, the class is created using the `addClass` method, and an implicit no-arg constructor is added. For the `gcd` method, `ClassToGenerate` mainly uses `iload_0`, `iload_1` (to load the first and second arguments, resp.), `istore_0`, `istore_1` (to (over)write the first and second arguments, resp.), as well as the `isub` command to subtract two numbers. These are all called using `CLEmitters`'s `addNoArgInstruction` method.

In order to implement the various control structures (the `while` loop and `if` block), `CLEmitter` has an `addBranchInstruction` method, which takes a first argument with the branch instruction to execute (`goto` or `if_icmple`), and a second argument with the name of the label to which one needs to branch. Labels can be specified using `CLEmitter`'s `addLabel` instruction, which takes a `String` as argument.

To test the correctness of the `Generator` class, one can use the Fernflower (or any other) decompiler, which translates a compiled class file into a regular java file. This output can then be compared with the original java file.

3 Lexical Analysis

3.1 Hand-written compiler

In the `Scanner`, skipping over block comments is done by first detecting the start of a block comment in the `getNextToken` method. Once this has been detected, one enters a `while` loop which exits as soon as the end of the block comment has been detected.

3.2 JavaCC compiler

Using JavaCC and lexical states, a neat solution is the following:

```
 MORE: {"/*":  IN_BLOCK_COMMENT}
< IN_BLOCK_COMMENT > MORE: {< ~[] >}
< IN_BLOCK_COMMENT > SKIP: {"*/":  DEFAULT}
```

It works as follows: when the start of a block comment is matched in the `DEFAULT` state, it is skipped and the state is swtiched to `IN_BLOCK_COMMENT`. Any other character seen in this new state is skipped, unless the JavaCC sees the end of a block comment, in which case this is skipped and the state is switched back to `DEFAULT`.