PHASE 1: LEXICAL ANLYSIS

Definition:

Given a piece of code, this phase should analyze it and extracts a list of tokens found. This phase relies on a pre-made DFA transition table, which should be well-designed for reliable output.

Design:

THE CLASS:

We made a class and called it `Lexical`. The constructor of the class takes only the path of the DFA table (as plain-text file).

A lexical analyzer requires a transition table to use. We wanted to make that table code-independent, which will make future additions and edits much easier. To solve this problem, we designed our system to deal with a specific-formatted plain-text file containing the transitions.

The system initially takes the path of the table once, and analyze any piece of code afterwards with `analyze` function, using the same constructed table.

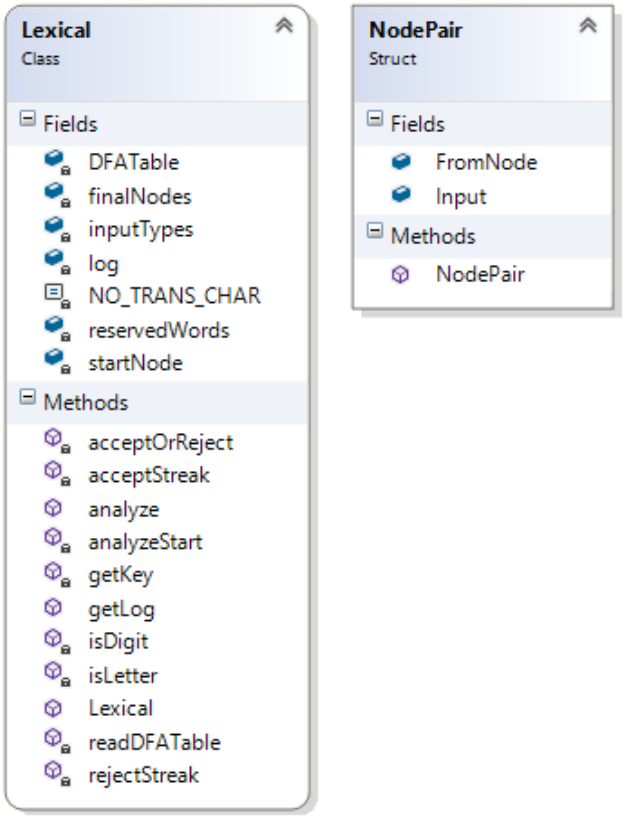
`Lexical` main functions are:

1. `readDFATable`: A private function that reads the content of the plain-text DFA transition table and store it in a private structured data object. This function is automatically called when a new object of `Lexical` is instantiated.

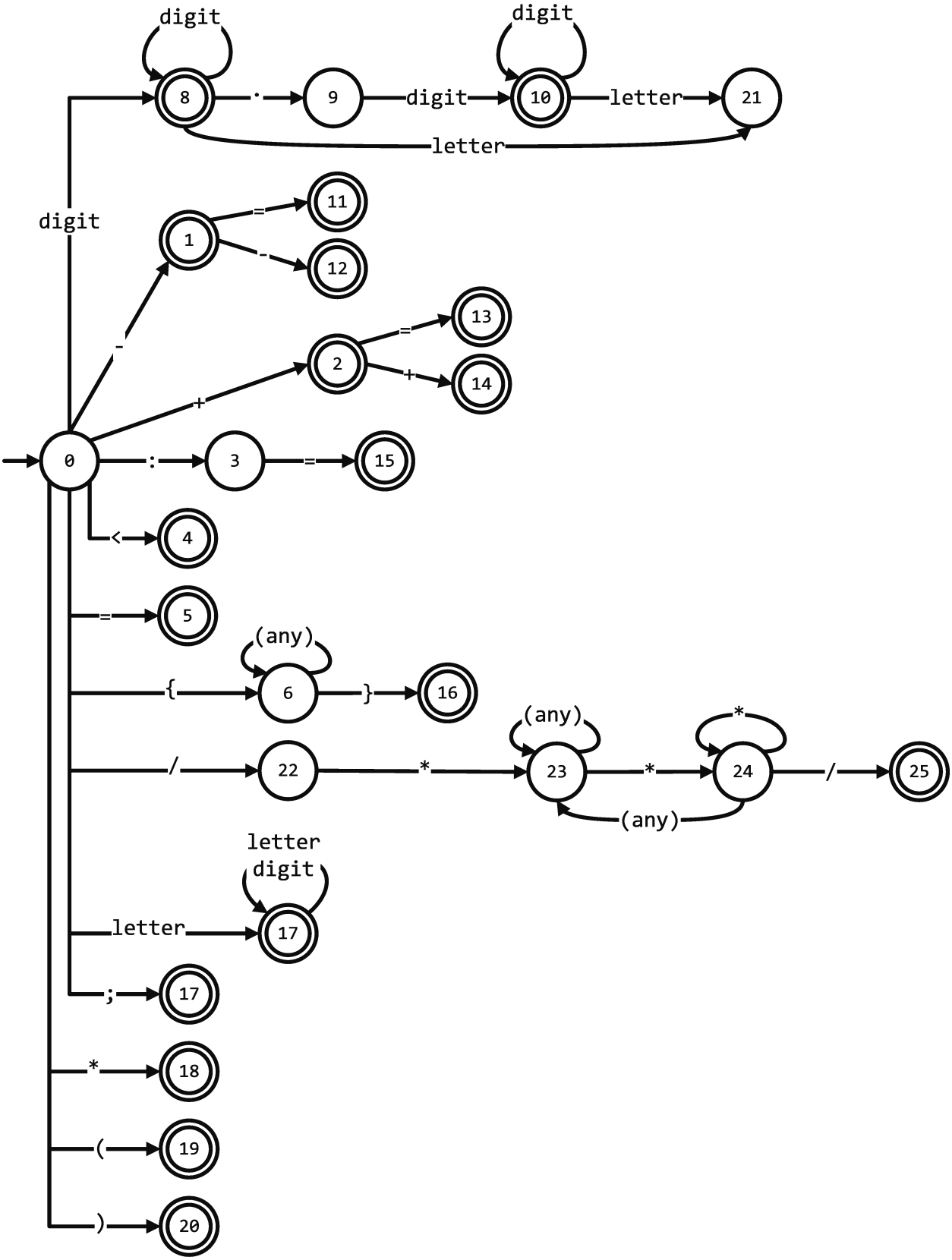
2. `analyze`: A public function that do the work of analyzing the code (converting code into tokens). The code is a `string` object, and is given as the only input parameter. Returns a `List` object of accepted tokens.

3. `getLog`: A public function that returns a summary (log history) of the last analyzed code by `analyze`. This is usually useful for testing the system. The log contains both accepted and rejected tokens along with their lexeme names.

CLASS DIAGRAM:



DFA DIAGRAM:


digit
digit
8
.
9
digit
10
letter
21
letter
digit
1
=
11
-
12
-
2
=
13
+
14
+
0
:
3
=
15
<
4
=
5
(any)
{
6
}
16
(any)
*
/
22
*
23
*
24
/
25
(any)
letter
digit
letter
17
;
17
*
18
(
19
)
20

Output:

The expected output from this phase is a list of tokens found in the code. The `Lexical` class also offers log history showing the accepted tokens along with any error/unidentified tokens for testing purposes.

The system supports the following tokens:

- ID's
- Numbers
- Fraction numbers
- Minus operator (`-`)
- Plus operator (`+`)
- Less-than comparison (`<`)
- Equality comparison (`=`)
- Assign operator (`:=`)
- Augmented minus operation (`-=`)
- Augmented plus operation (`+=`)
- Decrement operator (`--`)
- Increment operator (`++`)
- C-like comments (`/*comment*/`)
- Curley braces comments (`{comment}`)
- End of statement sign (`;`)
- Multiplication operator (*)
- Left parenthesis
- Right parenthesis

And the following reserved words (as defined by Tiny-C language description):

`If`, `Then`, `Else`, `End`, `Write`, `Read`, `Repeat`, `Until`.