1. **Lexical specifications:**

Alphabet : {'L', 'e', '0', 'N', '\_', '"', '/', '\*', '=', '<', '>', ':', '+', '-', '&', '.', ' ', '\n', '\r', '~', '\u0000'}

In order to **reduce the number of transitions in the table**, I used some **mappings** between lookup\_char to input\_char. Lookup\_char is the character that the lexer reads from the file. Input\_char is the one that used in the transition table. Since there is 'e' in the regular expression for id and '0' is also special for many cases, I separated them from other letters or numbers to keep the automaton is deterministic.

lookup\_char input\_char

[abcdf..z|A..Z] -> L (letter except e)

e -> e

0 -> 0

[1..9] -> N(nonzero)

|&!?(){}[];, -> &

space, \t, null -> ' '

: . \* = < > \_ + - \* / " -> : . \* = < > \_ + - \* / " (keep the same)

-1 -> \u0000 (end of file)

invalid character -> ~

regular expressions:

**id::= (L|e)(L|e|0|N|\_*)\****

***integer::= N(0|N)\*|0***

***float::= (N(0|N)\*|0).(((0|N)\* N)|0)(e(+|-)?(N(0|N)\*|0))?***

***String::= "(L|e|0|N|\_*|S)\*"**

1. **Finite state automaton:**

First I generated the DFA for id, integer, float, and string using the online tool.

A picture containing radar chart

Description automatically generated

DFA for id, integer, float, and string

Then I combined all the DFA for required lexical specifications.

Diagram, schematic

Description automatically generated

DFA for lexical analyzer

1. **Design:**

A picture containing timeline

Description automatically generatedClass diagram

I followed the approach of **Table-driven scanner**, reading character by character from the file(.src), and judging the next state according to the transition table. If the state is a final state, it will create a valid token or a error token. Repeat the process until the end of the file.

The **Token class** contains information such as the token type, the lexeme, the location, etc.

The **State class** represents the state in DFA, including attributes such as the ID of the state(an integer), the name of the state, etc. The rows of the transition table is recorded by the Map(Character, Integer). For each state(an integer), there is a map with the key as input character and the value as the next state(Integer).

The **class of LexicalAnalyzer** is responsible for scanning every file, tokenizing and writing to output files. It contains an array of States, store the whole transition table. Since the state is represented as integer, it is efficient to use an array which can utilise the index as integer.

The **LecicalAnalyzerDriver** opens every file(.src) in a folder or a single file(.src), creates a LexicalAnalyzer object to handle the task of tokenizing.

More detailed explanation can be seen in the source code.

1. **Use of tools:**

I used [CyberZHG’s](https://cyberzhg.github.io/toolbox/min_dfa?regex=KGF8Yikq) Toolbox to generate the DFAs from regular expressions of id, integer, float, and string. This tool is very easy to use, without overhead of the installation of any application. Although it does not accept some characters such as '+' and '-', using some other signs to replace them is a quick method to have the DFA for key tokens.

Then, with the draft DFA, I used [diagrams.net (formerly draw.io)](https://app.diagrams.net) to draw the whole DFA for this assignment(the lecture slide gave me detailed insights). It provides great flexibility with which I can label final states(backtrack) as red.

Following the DFA, I create a transition table using Excel, which is good as recording large amount of data in a form of table.

As for writing codes, I used [IntelliJ IDEA](https://www.jetbrains.com/idea/), which is great at coding auto completion, code debugging, etc. The class diagram is generated by [PlantUML](https://plantuml.com/), which can be integrated in the IDE conveniently.