Lexical specifications

The lexical specifications are exactly same as the assignment handout, like the following

Atomic lexical elements of the language

```
      id ::=  letter alphanum*
alphanum ::=  letter | digit | _
 integer ::=  nonzero digit* | 0
   float ::=  integer fraction [e[+|−] integer]
fraction ::=  .digit* nonzero | .0
  letter ::=  a..z |A..Z
   digit ::=  0..9
 nonzero ::=  1..9
```

Operators, punctuation and reserved words

```
==      +       (       if
<>      -       )       then
<       *       {       else
>       /       }       for
<=      =       [       class
>=      &&      ]       integer
;       !       /*      float
,       ||      */      read
.               //      write
:                       return
::                      main
```

Finite State Automaton

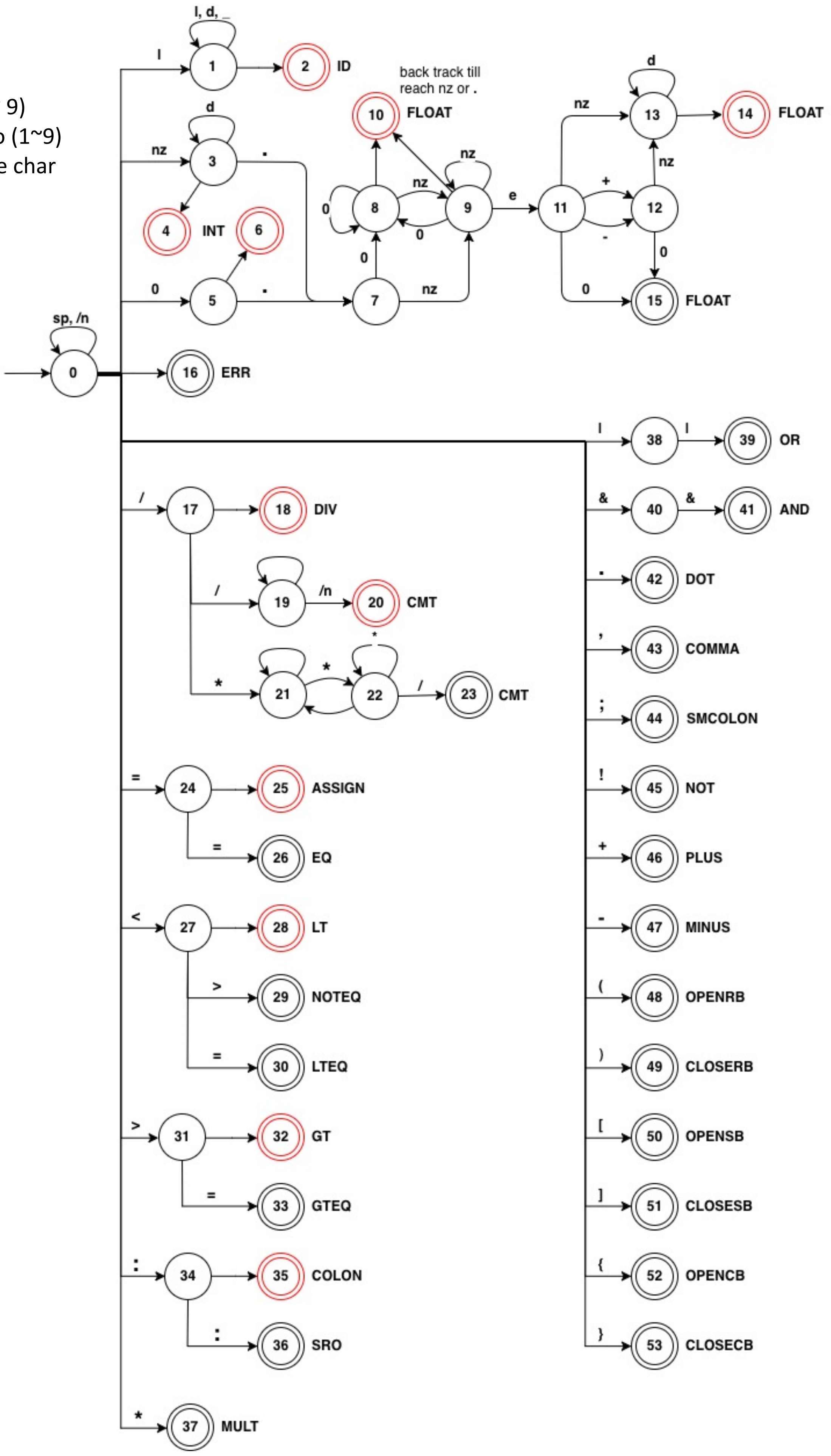
Special notation

l means letter

d means digit (0 ~ 9)

nz means nonzero (1~9)

/n means new line char



Error reporting and recovery

Encounter an error

1) Encounter a character with unknown behavior at state 0, for example, char '$' is not defined, so it is considered as unknown type error. Also, this kind of errors are able to accumulate, for example, there is a line $$$a in the input file, when first char '$' is read, the error message is not generated right away, keep reading the next char till find a valid one, in our case, it's char 'a'. So, $$$ is considered together as unknown type error, not separate them and report error three times.
2) Encounter a character with inappropriate position in the middle of the state flow, for example 1.3ea, the value 1.3e is considered as invalid float, the reason is that before reading char 'a', the state is on the float-state line, and after reading char 'a', there is no 'a' as input for current state, so invalid message is generated. After output the invalid float message, the current state back to 0 and re consider the input char 'a'.
    a. Float error
        i. After reading "1.", at state 7, the next char is neither '0' or nonzero, for example, 1.a. The "1." is reported as float error.
        ii. After reading "1.1e", at state 11, the next char is not one of the following: '+', '-', '0' or nonzero, for example 1.1ea. The "1.1e" is reported as float error
    b. Operator OR (||) and AND (&&) error
        i. After reading char '|', the next char is not '|', for example |a. The "|" is reported as unknown type error.
        ii. Similar for operator &&.

Invalid Type

1) INVALID ID:
    a. only for char '_', report [INVALID ID: _]
2) INVALID FLOAT:
    a. 1., 1.a, 1.=, report [INVALID FLOAT: 1.]
    b. 1.1ea, since char 'a' is defined, so just report [INVALID FLOAT: 1.1e]
    c. 1.1e$a, since char '$' is not defines, so accumulate the error, when reach char 'a', like the case above, report [INVALID FLOAT: 1.1e$]
3) UNKNOWN TYPE:
    a. $, report [UNKNOWN TYPE: $]
4) UNCLOSED CMT:
    a. Unclosed comment like /* abc… (end of file), report the error line where /* located

Design

Class

1) KeyWords: stores all reserved words, able to check whether an id is reserved word of not.
2) Token: stores token info such as type, value, and line number.
3) Table: The analyzer is table-driven, so all the DFA is translated and stored in it. It is able to match a coming char with current state, check final state, get next state, check backtrack info and create token.
4) LexicalAnalyzer: it's table driven, nextToken() function keep calling nextChar(), and determine when the token reading is over and report it. Also, the backtrack function is included (explained later).
5) Driver: repeatedly calls the lexical analysis function nextToken() and prints each token until the end of the source program is reached.
6) Main: creates a Driver and let it run.

Backtrack

For cases like 1.100a, 1.000a, when char 'a' or any other invalid char coming when current state is 8, the backtrack is needed. The backtrack function keep going backward till reach either nonzero or '.'.

a. 1.100a, seeing 'a', go backward, reach '1', report [FLOAT: 1.1]
b. 1.000a, seeing 'a', go backward, reach '.', move one position forward, report [FLOAT: 1.0]

Note: Clear explanations for each class method is written in .java source file with JavaDoc format.

Use of tools

```
java.util.Arrays          // setup table
java.util.Hashsets       // store final state, backtrack state info
java.util.ArrayList      // set up table
java.lang.Character   // dealing with characters
java.io.*                    // for reading and writing source program files, also for the exception
```