**Lexical specifications**

**Notes on the regular expressions and the DFA**

* All letters (a..z, A..Z) are represented as “L”.
* Nonzero digits (1..9) are represented as “D”.
* Zero (0) is considered distinct from the other digits
  + Reason: it makes a difference when parsing integers and floats. Thus, I always gave it a separate symbol and transition in the DFA.
* The DFA will recognize a start or an end of a comment (// /\* \*/), but won’t remove the commented text
  + Comment removal will happen inside the code
  + Reason: it’s impossible to implement nested comments in a DFA (need some form of memory to keep track of the comment levels, which a DFA doesn’t have).
* At the regex/DFA level I do not make the distinction between IDs and reserved words, they are both scanned as “words”
  + A function later in the code takes care of determining if it’s a reserved word or not
  + Reason: it is much simpler to do in code than in a DFA

**Regular expressions**

word: L (L|D|0|\_)\*

int: (D (D|0)\*) | 0

float: ( (D (D|0)\*) | 0) . ( ( (D|0)\* D) | 0 )

semicolon: ;

comma: ,

plus: +

minus: -

openpar: (

closepar: )

opencurly: {

closecurly: }

opensquare: [

closesquare: ]

assignment: =

equals: ==

divide: /

line comment: //

open comment: /\*

multiply: \*

close comment: \*/

less than: <

less than or equals: <=

greater than: >

greater than or equals: >=

not equals: <>

dot: .

**Other specifications and notes**

1. Reserved words are: and, not, or, if, then, else, for, class, int, float, get, put, return, program
2. The language is case-sensitive
3. // is a line comment
4. /\* … \*/ is a block comment
5. Nested comments are supported

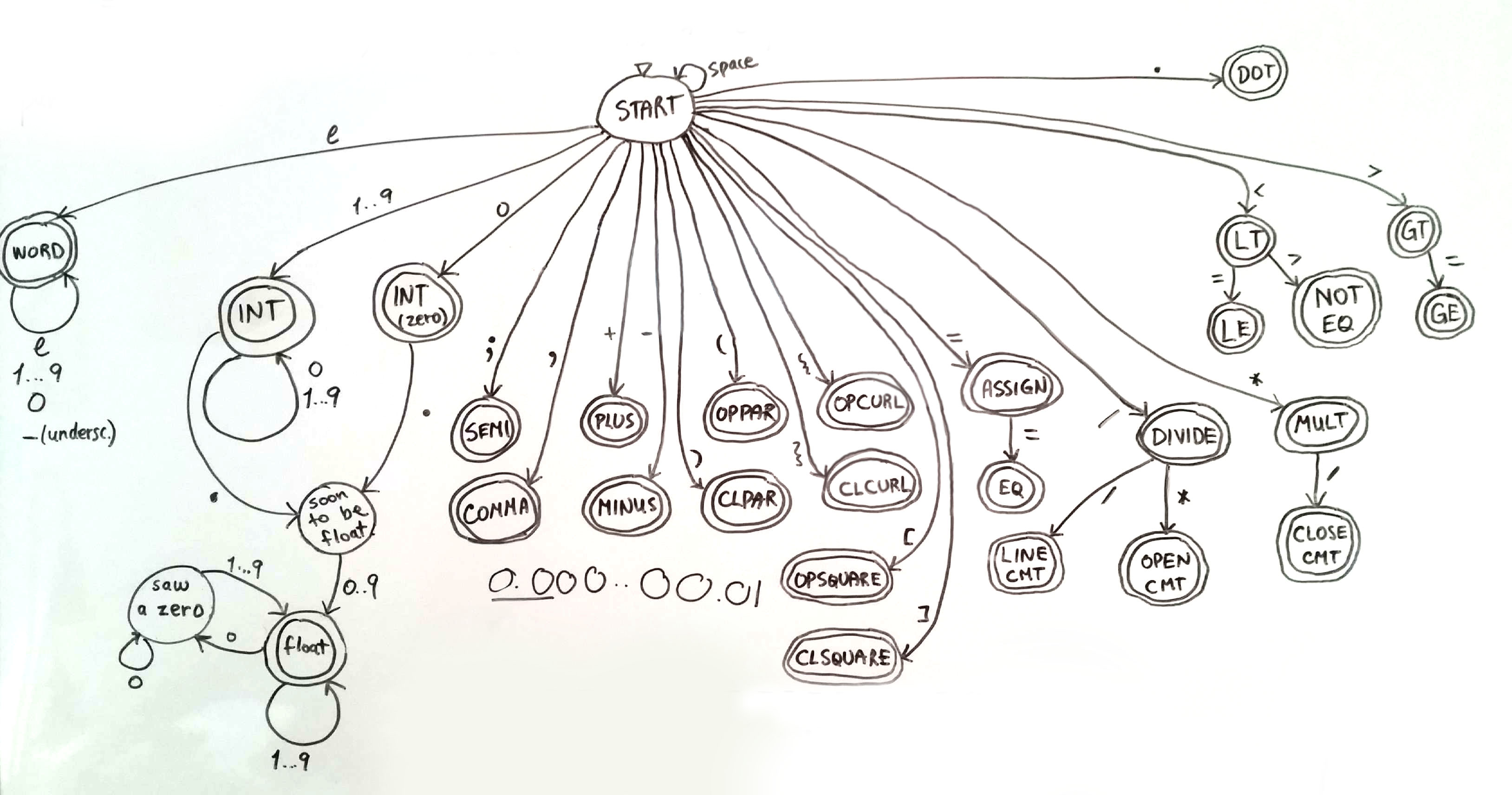
**Edge case choices**

I encountered many edge cases that presented ambiguity. I chose to consider most of them not to be errors at the scanning phase, even the resulting tokens might not make a lot of sense. This was done in order to simplify the DFA and the scanner logic.

Here is a description of all the choices that were made for each edge case.

1. Decimal dot
   1. If seen outside a float number, will be read as the dot operator regardless of what is next to it
   2. Exception: if it is seen after an integer, for example “0.banana”, I will be read as “0.” (invalid token) and a valid identifier token “banana”
2. “-“ is always read as an operator token
   1. The syntax analyzer will later differentiate between the minus sign and the minus operator
3. Spaces or lack of spaces:
   1. If there is no space between two IDs or two RW (reserved word) or an ID and a RW that are one after another, they are considered as one ID
      1. Example: “ifBob” will be parsed as ID with value “ifBob”
      2. Example: “if Bob” will be parsed as a RW “if” and an ID with value “Bob”
   2. A space is not mandatory
      1. between a RW (reserved word) and an operator or punctuation
      2. between an ID and an operator or punctuation
      3. Example: “for(“ becomes a reserved word “for” and punctuation “(“
   3. If an integer is **after** an ID or a RW, without a space between them, they will be recognized together as one ID. If an integer is **before** an ID or RW, it will be recognized separately.
      1. Example: “Bob1” will become an ID token “Bob1”
      2. Example: “1Bob” will become an int token “1” and an ID token “Bob”
   4. If an float is **after** an ID or RW, without a space between them, it will be recognized as an ID up to the decimal dot, then the dot and the remaining part of the float will be scanned as separate tokens. If on the other hand, the float is **before** an ID or RW, it will be recognized as a full float.
      1. Example: “Bob1.54” will become an ID token with value “Bob1”, an operator “.” And an int “54”.
      2. Example “1.54Bob” will become a float “1.54” and an id “Bob”

**DFA**



**Token Data Structure**

Each token is an instance of class Token. Each token contains:

* An enum type describing its type (int, float, operator, punctuation, identifier, reserved word)
* A string containing its value
* A pair of integers containing its position (line, column)

**Implementation**

The solution has 2 main components

1. **StateTable.cpp, which contains all the DFA logic**
   1. State transition table which represents exactly the DFA above in a 2D int array
   2. Static getState method, which takes the current state and a character as a parameter, looks them up in the state transition table and outputs the next state
   3. Final state information, conversion from final state to token type
   4. Static helper members and methods:
      1. enum type which maps states to integers
      2. method to map input characters to input indices for state transition table
2. **Scanner.cpp, a table-driven scanner**
   1. Reads input file character by character, saving them in a buffer
   2. Uses StateTable methods to navigate the DFA until it falls out of the DFA
      1. Note: this is different from the implementation we saw in class.
      2. Class implementation: we sometimes need to backtrack (if there is ambiguity) and sometimes not (when there is no ambiguity).
      3. My implementation: I always read until I reach a point where I can’t progress in the DFA anymore, this happens when I’ve read the first character of the next token. So I always backtrack. This simplified the ambiguity handling logic for me.
   3. Checks if it was an opening or closing of a comment and adjusts appropriate flags
   4. Checks these comment flags, and if we are not inside a comment, looks if it can create a token
   5. When we fell out of the DFA it’s either because we found a token or because we encountered an error while looking for it. We can know this by asking the StateTable if the **last** state we were in was final or not
   6. If it was a final state, then we just saw a valid token, so we create and return a Token object
      1. If it was a WORD state, it could be either an identifier or a reserved word. So the scanner checks if it can find our token in the array of reserved words, and creates the token accordingly
   7. If it was not a final state, we report the error
   8. Upon creating a token or finding an error, get the scanner ready for processing the next token:
      1. Clear the buffer
      2. Backtrack
      3. Reset DFA to the Start state

**Utility component**

There is also a small singleton Logger class which provides utility for logging tokens, debug messages and errors in their respective log (.txt) files.

**UML**



**Possible errors**

* Invalid characters, for example “$”
* Trailing zeroes in floats, for example “4.50”
* Malformed floats, for example “ 4.”
* Underscore not inside an identifier
* Close comment without having opened one
* Open a comment without closing it

Leading zeroes are not an error, they are scanned as separate integers with value “0”. So “06.3” becomes two tokens: and integer with value “0” and a float with value “6.3”.

**How errors are handled and recovered from**

Upon seeing one of the errors above, the scanner reports the error, its description and its location in the logError.txt file. Then it goes in “panic mode”: resets the DFA to its start state, backtracks if needed and attempts to resume scanning and finding tokens.

**Testing**

Testing is done using the Visual Studio Unit Test Framework. Test cases are subdivided in categories, with one input file per category:

* Correct numbers
* Numbers with errors
* Correct words (ids and reserved words)
* Words with errors
* Correct comments
* Comments with errors
* Operators & punctuation (correct and incorrect, plus invalid characters)

For correct input files, assert statements verify that we obtained the expected number of tokens of each type and no errors. For input files with errors, we verity that we got the expected number of errors, and the expected number of tokens of each type too.

**Test cases**

|  |  |
| --- | --- |
| Correct numbers | **Integers**: single digit, multiple digits, with and without zeros, very long number  **Floats**: single digit before / after decimal dot, multiple digits before / after decimal dot, zeroes after decimal dot (but not trailing zeroes),  **Leading zeroes** (leading zero becomes a separate int token):zero before int, zero before float  **Mixing numbers with other things:** letter before/after an int, operator before/after an int, punctuation before/after int, letter before/after a float, operator before/after a float, punctuation before/after a float, decimal dot followed by an int (ex. “.1”) |
| Numbers with errors | **Mixing numbers with other things:** underscore before/after an int, underscore before/after a float, invalid character before/after an int,  **Trailing zero** in a float  **Malformed float** (ex. “1.a” or “43.”) |
| Correct words | **Identifiers:** single letter, letters mixed with digits, letters mixed with digits/underscores  **Reserved words:** all reserved words spelled correctly  **Misspelled reserved words** (insert a letter before/after/ inside, insert a digit or underscore inside or after, it becomes an identifier)  **Integer/float before reserved word/identifier**: scanned as 2 tokens  **Float after an identifier:** scanned as an id as far as possible, then dot is a separate token and the remaining part is an int  **Operators/punctuation next to identifiers/reserved words** |
| Words with errors | **Identifier starts with underscore** (error on underscore, the rest becomes a valid identifier)  **Invalid float next to an identifier or reserved word** (error on float, but word still becomes a correct token)  **Illegal characters next to identifier or reserved word** (error on illegal character, but word still becomes a correct token) |
| Correct comments | **Simple line and block comments**  **Nested block comments**  **Line comment inside block comment**  **Block comment inside line comment**  **Mixing line/block comments** |
| Comments with errors | **Open comment without closing**  **Close comment without opening** |
| Operators, punctuation, invalid characters | **Valid operations and punctuation mixed and matched in different ways**  **Invalid characters mixed with operations and punctuation**  **Underscores mixed with operations and punctuation** |

**Tools used**

* Visual Studio: a widely used IDE for C++. It is convenient because of its GUI debug functionality for setting breakpoints, stepping through the code and watching variables. Also intellisense helps while writing code by autocompleting and finding compile errors.
* Microsoft Visual Studio Unit Test Framework: used for testing. Used it because it was already built in Visual Studio.
* No libraries used, other than the standard C++ library