**Identify the lexical specification you use in the design of the lexical analyzer, as well as any changes that you may have applied to the original lexical specifications**

No significant changes were made. This implantation stays true to the specification given in the assignment. From the picture of the State Machine, it looks like there is no reserved words, integer values, or floats, but the implementation updates the token type before giving the token to the consumer of Lexer::***nextToken().*** A num token will changed to either an **int\_value** or **float\_value** token when calling nextToken() as well. However a num element can still be made if needed by editing ***Specification::updateTokenType().***

This Lexer strips all comments in the source. It will not return any comment tokens to the consumer.

All lexical elements provided in the specification are implemented the same. There are no medications here. These can be seen in the Specification class.

**Specification**

id ::= letter alphanum\*

alphanum ::= letter | digit | \_

*num ::= integer | float* ***Num tokens can be created manually, but the lexer will not give num tokens***

***They will updated to either integer or float***

integer ::= nonzero digit\* | 0

float ::= integer fraction

fraction ::= .digit\* nonzero | .0

letter ::= a..z |A..Z

digit ::= 0..9

nonzero ::= 1..9

**Note:** alphanum/fraction/letter/digit/nonzero are not tokens, but are parts used to create of id/integer/float tokens.

**Tokens**

== + ( if

<> - ) then

< \* { else

> / } for

<= = [ class

>= and ] int

; not float

, or get

. put

return

program

**Note:** There are comment tokens, but these are used by the lexer internally. Comment tokens will not be outputted by the lexer.

**Finite State Machine**

(Look at **finite\_state\_machine** image in folder for higher resolution)

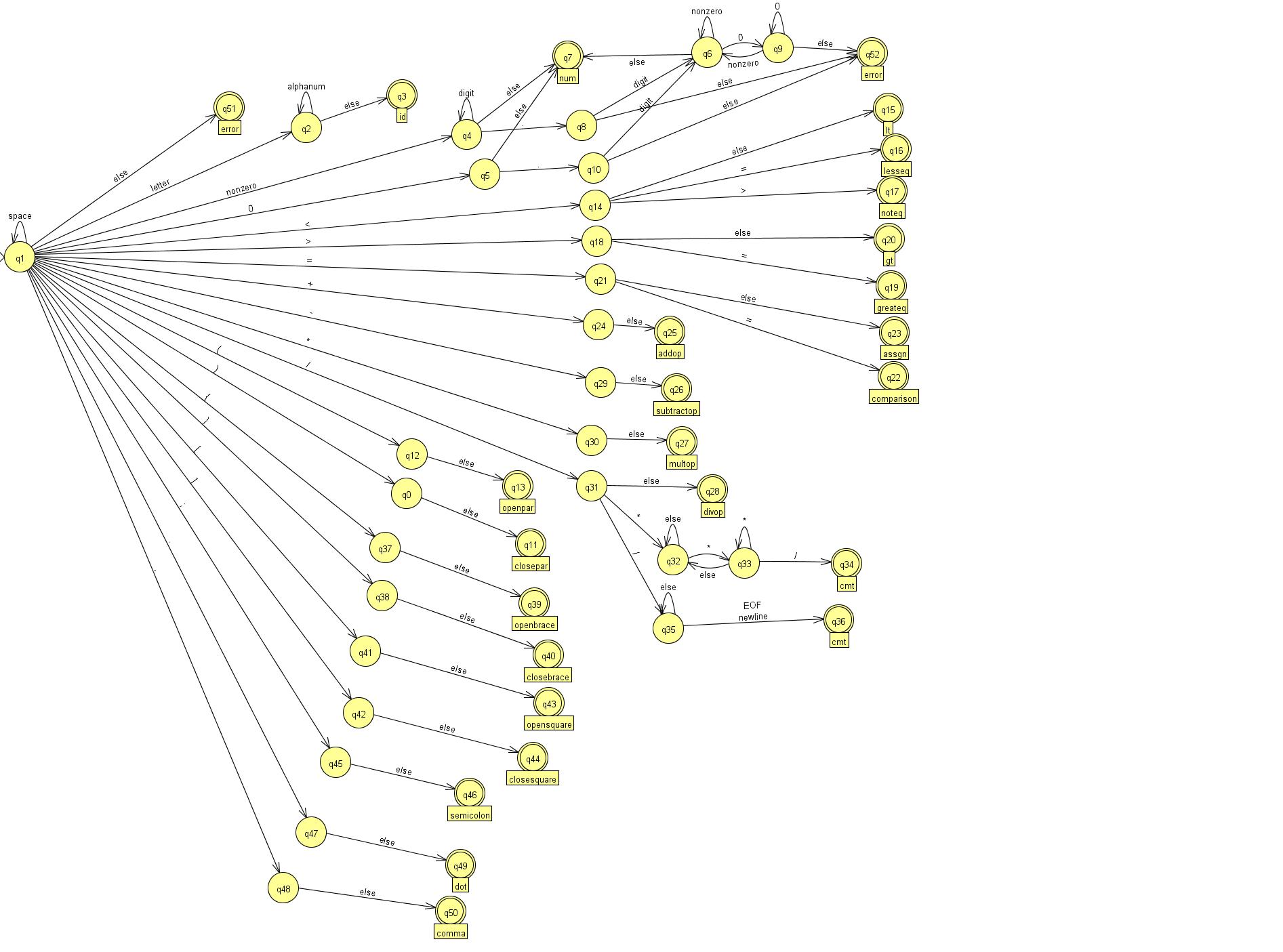
Starting State q1. The space transition includes all whitespace characters.

**letter transition:** is short hand for all lowercase and uppercase letters. Internally this creates a transition for each letter in the alphabet. [a-z]|[A-Z] = 26 \* 2 transitions

**nonzero transition:** is short hand for numbers [1-9]. Internally this creates a transition for each number. 9 transitions

**digit transition:** is short hand for numbers [0-9]. Internally this creates a transition for each number. 10 transitions

**alphanum:** is short hand for **letter|digit|\_** . Internally this creates a transition for each. 10 + 26 + 26 + 1 = 63 transitions

**Errors and Recovery**

**Types of errors**

A simple data structure holding the error message and type

struct TokenError {

std::string errorMessage;

ErrorType type;

};

enum ErrorType {

unkown\_symbol,

invalid\_float,

incomplete\_float,

};

**unkown\_symbol:** When the tokenizer reads an input that is not in our alphabet. Ex: {!,@,#,$,%,^,&,`,~,?,\,|, …etc}

When the tokenizer reads symbols that are not transitions on the first state, it will produce this error. However, when encountering this error, it still continues to read the rest of the file.

**invalid\_float:** Error for when floats don’t follow the specification. Ex: 1.00

1.00 Is considered an error because it doesn’t follow the specifications. There was no way to recover from this without creating weird issues. It shouldn’t be split into {“1”, “.”, “0”, “0”}, or {“1.0”, “0”} because it splits 1 number into multiple numbers. This may cause more confusion than it solves. However, when encountering this error, it still continues to read the rest of the file.

**incomplete\_float:** Error for when floats are in the form of 3.a. This doesn’t create an error, but it recovers by converting it into 3 tokens -> “3”, “.”, and “a”

The reasoning behind this recovery is that 3 might be an integer followed by a dot then an identifier. Maybe the programmer didn’t intend to write a float but instead wanted to use the dot operator on an integer.

In languages such as C++ or Java, this would be an error, but in other languages such as Ruby, integers have functions such as 1.next, or 2.lcm(2). Since I don’t anything about the semantics of the language, this was done to allow for more flexibility. However, this can easily be fixed if needed.

Example: <https://ruby-doc.org/core-2.2.0/Integer.html>

**Overview of implementation and Components**

Language used: C++ with visual studio 2015 (v140).

*To compile on visual studio 2013 (v120) in labs, see README for instructions.*

*Though I am not a C++ expert, I wanted to use C++ to develop my C++ and programming skills further.*

**Token:** A simple data structure to hold the token’s lexeme, token type, and source line. The source line indicates where in the source file this token is located.

struct Token {

std::string lexeme;

TokenType type;

int tokenLine;

TokenError error; // The error data structure in case this token is an error token

};

**token.h:**

* Holds data structure for a token
* Keeps an enumeration of token types. Example: (Id token, operator tokens, reserved word tokens, etc)
* Holds all error types
* Holds data structure for a Token Error

**Dfa:** Holdsthe state transition table and structure for the state.

It handles creating the states, and adding transitions between states.

**State:** A simple data structure to hold the state’s identifier, token type, errorType if needed, and boolean flags that hold whether that state is a starting state and final state. Also holds a boolean flag indicating whether the lexer should bracktack if it accepts at that state

struct State {

int stateIdentifier;

bool isStartState;

bool isFinalState;

bool needsToBacktrack;

TokenType tokenType;

ErrorType errorType;

};

**Specification:** Holds all the required specification data given by the assignment. Internally it creates the tokenizer finite state machine. The specification must be passed to the lexer in order for the lexer to give tokens.

**Lexer:** The actual lexer that tokenizes the source file. The specification must be created and passed to the lexer. Also must set the source file, for the lexer to tokenize.

**main.cpp:** Driver class that outputs the token to console or file. Set **writeToFile = true** to write the output to a file.

**Steps:**

* Create specification
* Pass specification into lexer
* Set lexer source file
* Keep calling nextToken() on the lexer, to get the tokens

Output file: **output.txt** outputs of the tokens from that source file

Error file: **errors.txt** outputs errors that occurred from that source file

**Tools/Libraries**

* I didn’t use any external tools/libs/dlls to write any code. Only used the internal libs provided with C++.
* Used JFLAP to draw and convert simple non deterministic state machines to deterministic state machines. <http://www.jflap.org/jflaptmp/>
* Used Bitbucket/Git for version control.
* Used STL containers such as <vector> and <unordered\_map> to hold data over simple arrays.
* Used C++11 smart pointers for easier pointer management.
* Used the unit testing framework provided inside visual studio to create unit tests.

**Techniques**

* Used nested <unordered\_map> to hold state transition table. I used maps instead of creating a list. This allows for quick lookups.
* Elements such as id, num, int, float are built up using smaller components. The code maps very closely 1 to 1 with the specification provided.
* When doing multi-line comments, I used a counter to hold how many nested multi-lines there are. If the nest count = 0, then the multi-line comments are properly closed. I looked into using a stack implementation to solve this, similar to how a push down automata would do it, but I found this more complex than it needed to be.
* Loaded entire source file into vector instead of trying to read it into a buffer. Initially I tried a double buffer technique, but this was more complex than it needed to be. It would cause issues where if we needed to backtrack, and that backtrack wasn’t in the current buffer, we would need to go into the previous buffer. It is doable but offers no advantages to how a vector would do it.