COMP 442
Compiler Design

Assignment 1: Lexical Analyzer

Writer:
Théo Bary (ID: 40348931)

Instructor: Dr. Joey Paquet
joey.paquet@concordia.ca

Date Due: Sunday, 25 January 2026, 11:59 PM

Section 1 - Lexical Specifications:

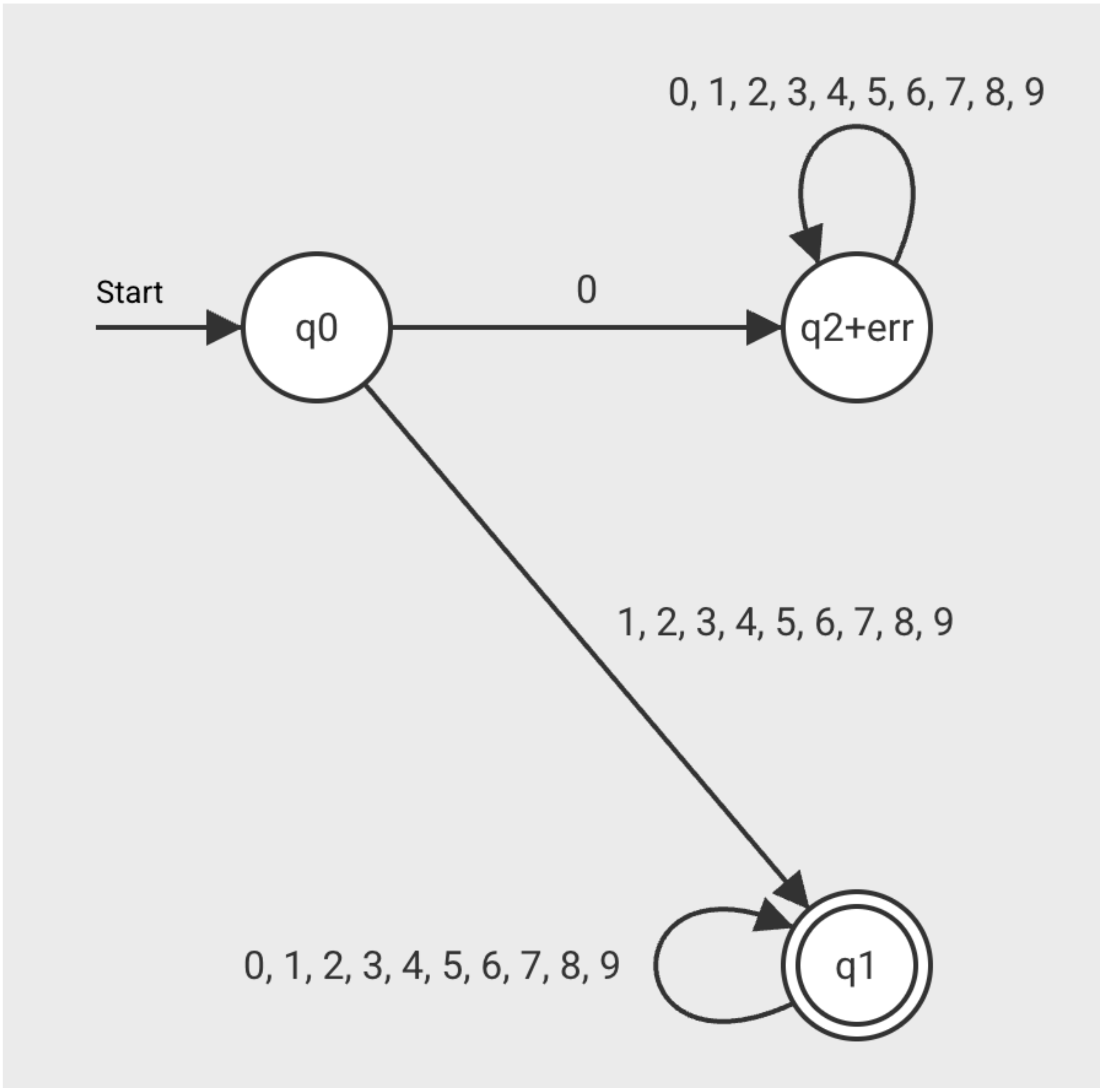
Block comment	`\/\*[\s\S]*?\*\/`
Inline comment	`\/\/.*$`
Float number	`([1-9][0-9]*|0)\.([0-9]*[1-9]|0)(e(\+|\-)?([1-9][0-9]*|0))?`
Integer number	`([1-9][0-9]*)|0`
ID	`[a-zA-Z]([a-zA-Z]|[0-9]|_)*`
Keyword	`if|then|else|while|class|integer|float|do|end|public|private|or|and|not|read|write|return|inherits|local|void|main`
Operator	`==|<>|<=|>=|::|<|>|\+|-|\*|\/|=|\(|\)|{|}|;|,|.|:|\[|\]`

All matches are attempted at each character, the longest match is then selected.

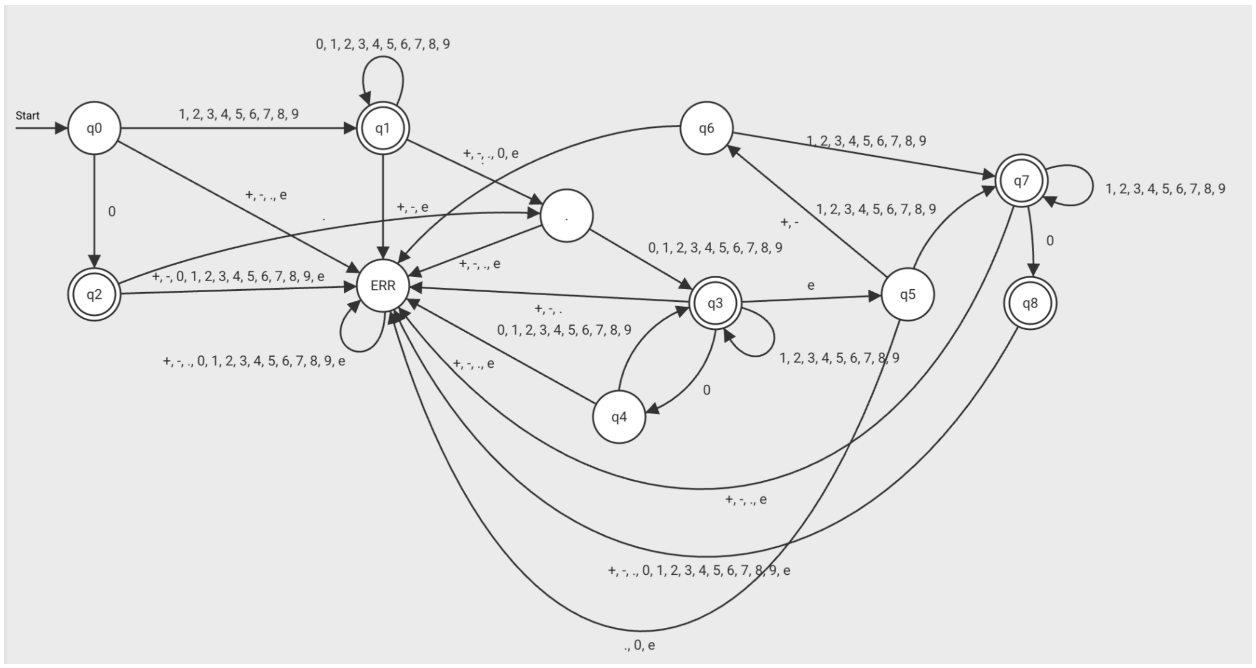
No changes are made to the original specification.

Section 2 - Final State Automaton:

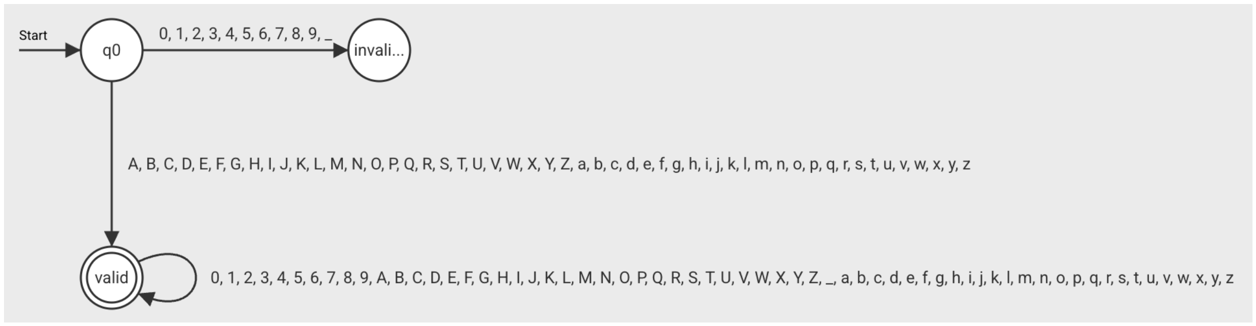
Integer numbers:



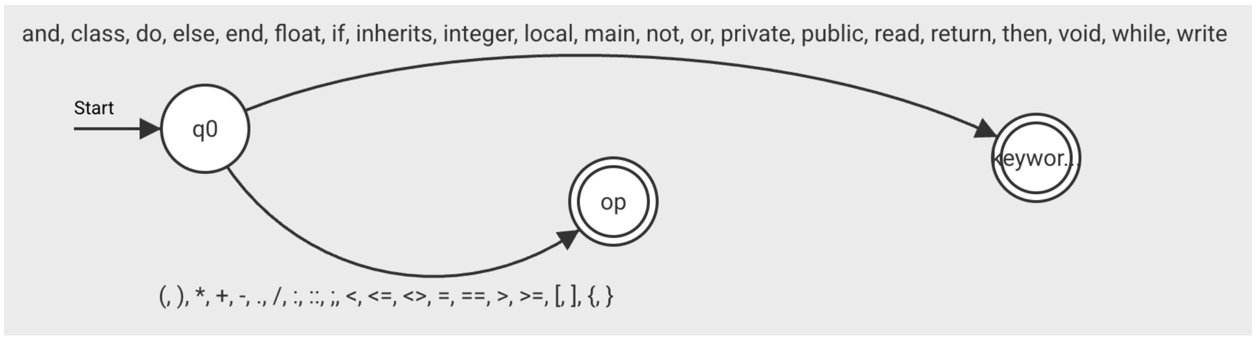
Float numbers:



IDs:


Start
q0
invali...
valid
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, _
A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z, a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z, _, a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z

Keywords and Operators:


and, class, do, else, end, float, if, inherits, integer, local, main, not, or, private, public, read, return, then, void, while, write
Start
q0
keywor...
op
(,), *, +, -, ., /, :, ::, ;, <, <=, <>, =, ==, >, >=, [,], {, }

Keywords and operators are checked as whole strings instead of character per character, this leads to faster processing. The longest match is always prioritized.

Section 3 - Design:

All classes, types, structures and methods are inside a “lang” namespace to ensure no conflicts are caused with the standard library or any imported 3rd party library.

The **Lexical Analyzer** is a class. It has the following methods:

```
std::uint64_t lang::LexicalAnalyzer::readFile(std::string_view path);
lang::Token lang::LexicalAnalyzer::next();
float lang::LexicalAnalyzergetProgress() const;
```

2 custom data structures/types are present:

```
enum class TokenType { … };
```

And

```
using Token = struct {
    TokenType type;
    std::string lexeme;
    std::uint64_t line;
    std::uint64_t pos;
};
```

To correctly use the **Lexical Analyzer** class, it must be instantiated and the method LexicalAnalyzer::readFile() must be called while giving it the path to the file to be “Lexed”. This method will read the contents of the file and store it in memory.

The LexicalAnalyzer::next() function can then be called repeatedly until a Token with its TokenType type == TokenType::END_OF_FILE is returned to analyze and get all tokens in the file corresponding to the **Lexical Specifications** given in Section 1.

The std::string lexeme field of the data structure represents the characters that were used to match the specific TokenType.

Errors such as unknown/unexpected characters will have their TokenType type set as TokenType::UNKNOWN and what caused the error will be in the std::string lexeme.

When calling LexicalAnalyzer::next(), here is the sequence of checks that are ran:

1. Are we at the end of the file? If yes -> return TokenType::END_OF_FILE
2. Regular Expression matches (see Section 1) are attempted for:
    a. Block comment
    b. Inline comment
    c. Float number
    d. Integer number
    e. ID
        i. If type ID is matched, we check against the reserved keyword list.
    f. If none of the above are matched, we check against the operator list.
    g. If still nothing is matched, we return TokenType::UNKNOWN and advance one character.

Section 4 - Use of tools:

I chose regular expressions for the lexical analyzer because they model lexing as a mathematical operation. A regular expression precisely defines the language of valid tokens, and when it is written correctly, the matching process has no side effects or ambiguity. This makes tokenization deterministic and predictable, reducing the risk of unexpected behavior compared to ad-hoc parsing logic.

With that said, I wrote the project using c++ and the standard library offers regex matching (std::regex). This worked out of the box but was incredibly slow. It would "lex" a file of around 100 lines containing around 75 tokens in around 10 milliseconds and when tested on a file with more than a million lines and approx 1.5M tokens it would spin forever (DNF).

This led me to search for 3rd party libraries that would still use Regular Expressions but in a more optimized approach. This is when I found "ctre", it is a c++ library that implements Regular Expressions at compile time instead of runtime. After implementing this library and optimizing my code at diverse places (mainly memory optimizations such as using file size to estimate final token amount and reserving space all at once instead of allocating more memory on the heap every time a token is generated), the same tests as before went down to ~0.05 milliseconds for the simple test and ~500 milliseconds (or half a second) for the 1.5M token test.

The logging library "spdlog" was used for logging asynchronously with levels. The build tool used "xmake" makes it very simple to include those 2 libraries.