The Lexical Analyzer

Document of assignment 1 of COMP 442 in Concordia University, Winter 2022

Lexical Specifications

Terminals:

```
id := letter alphanum*
integer := nonzero digit* | 0
float := integer fraction [e[+|-] integer]
string := "character*"
```

I reserved the string literals in the lexical specifications. This can be useful in the syntax analysis phase.

Non-terminals:

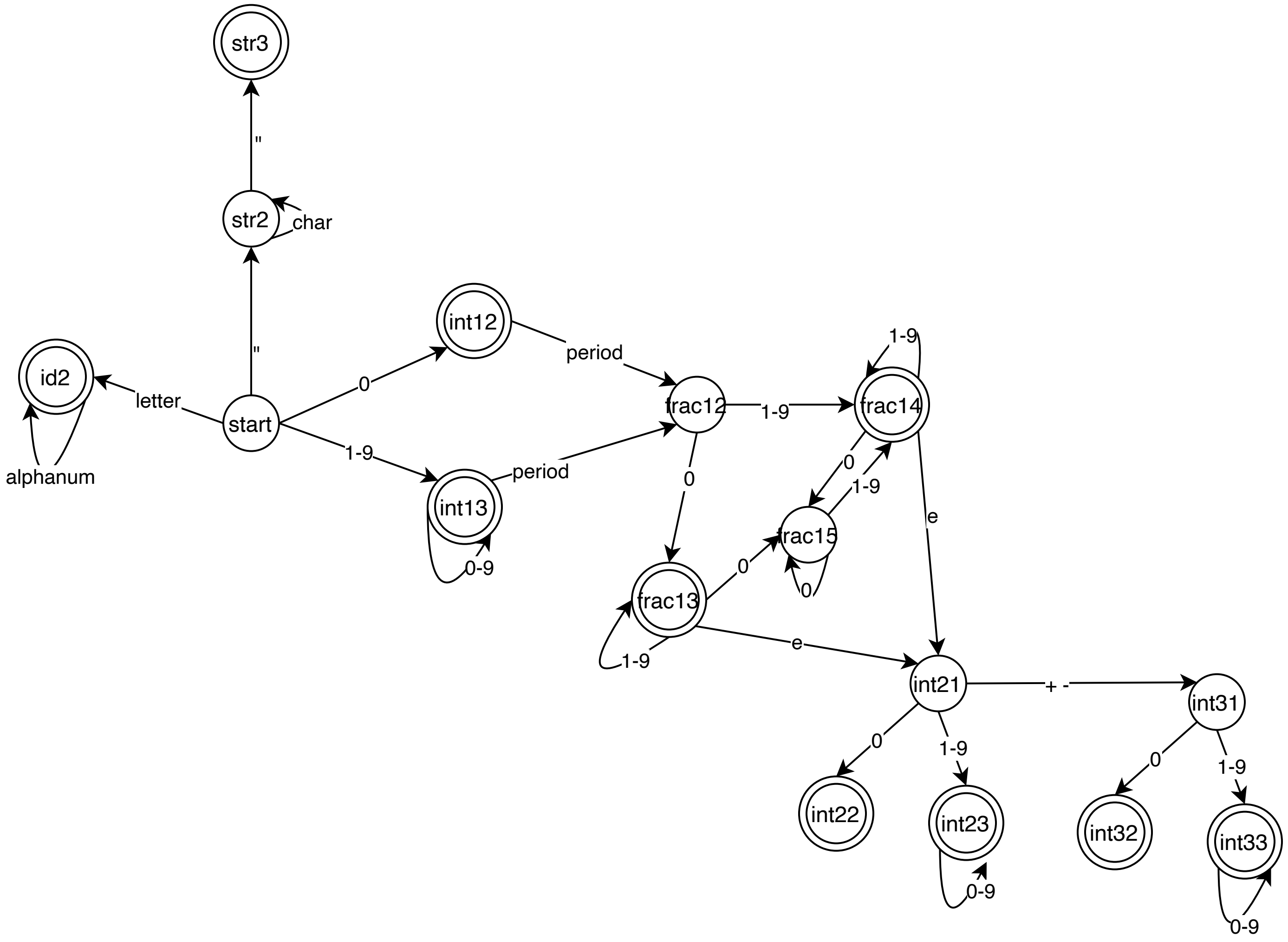
```
letter := a..z|A..Z
digit := 0..9
nonzero := 1..9
alphanum := letter | digit | _
fraction := . digit* nonzero | .0
character := {all characters in ASCII}
```

Operators, punctuations, reserved words, comments are the same with those in the handout.

Note that spaces (including tabs, line breaks, etc.) are treated as token separators, although tokens are not necessarily sparated by spaces.

Finite state automation

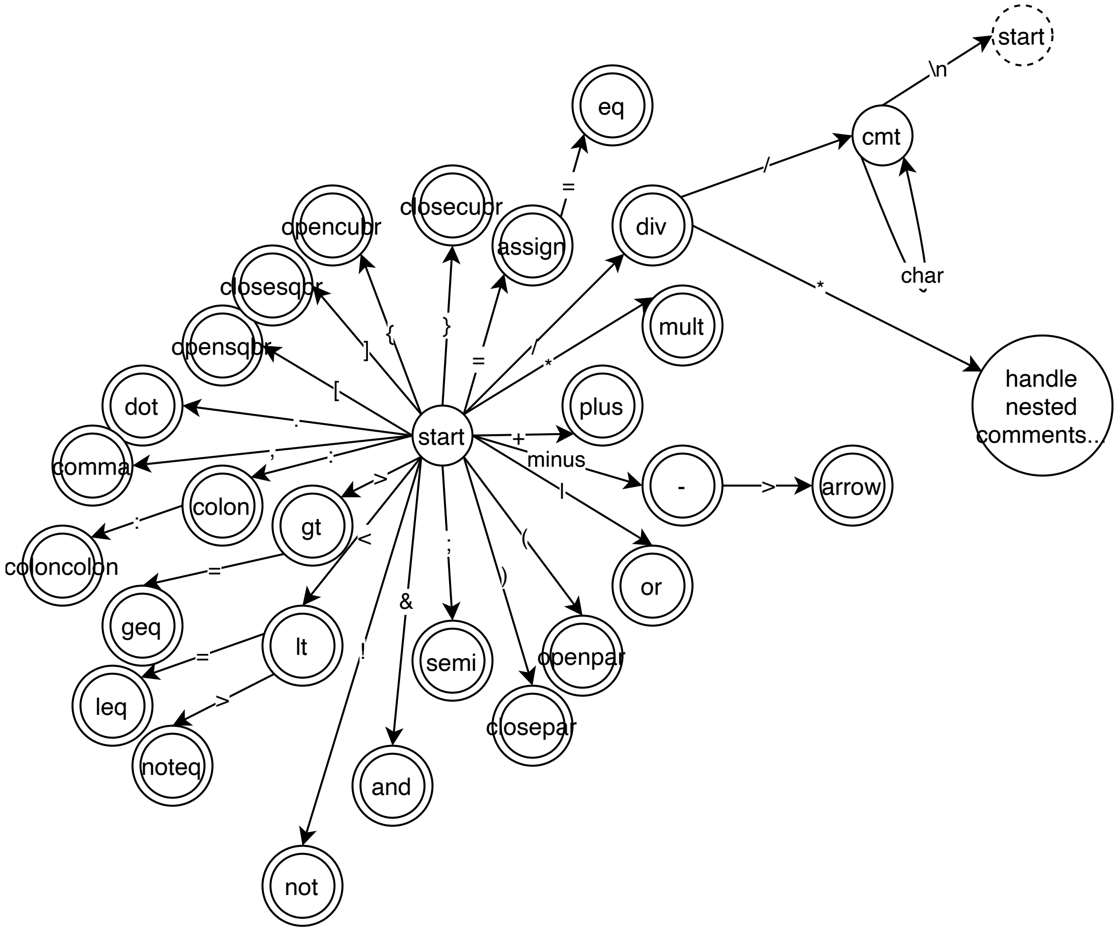
Part 1: The atomic lexical elements



The outputs of this DFA is specified as follow

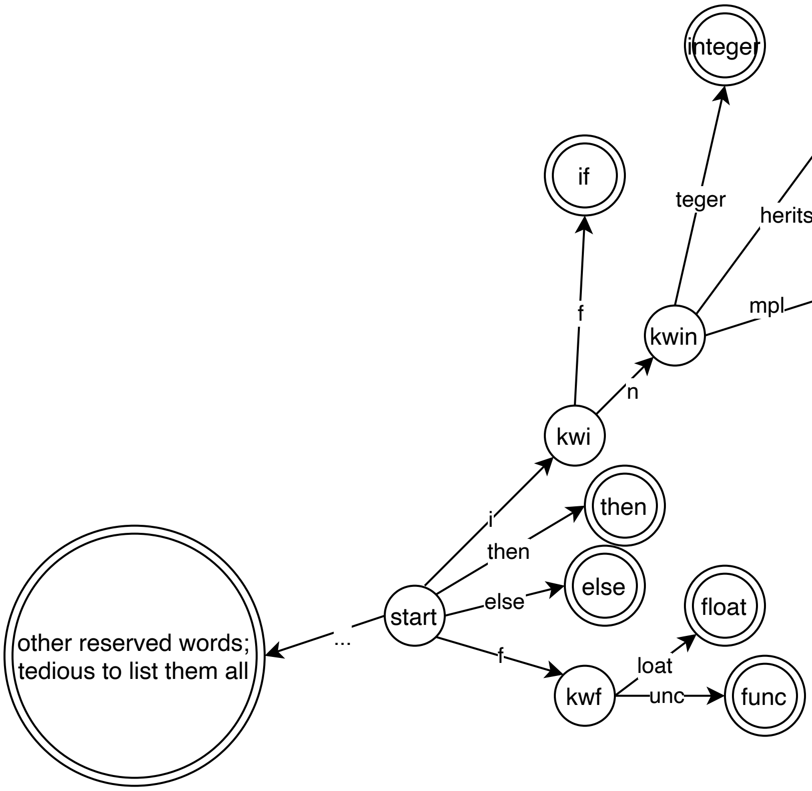
```
id: {id2}
integer: {int12, int13}
float: {frac13, frac14, int22, int23, int32, int33}
string: {str3}
```

Part 2: The operators and comments



Note that the case of nested block comments needs some special handling that beyond the ability of a DFA. It's handled in the lexer program.

Part 3: The keywords



The graph only shows part of the keywords because it's too tedious to list them all. I believe you can get the idea.

Design

The lexer is implemented in Rust. The `Lexer` struct has two public implementations: `read_source` and `next_token`. The `read_source` methods reads a source file and generate a list of tokens from the source. The `next_token` returns an option of token and moves the iterator to the next token. The basic use of `Lexer` looks like this:

```
let source: String = fs::read_to_string("input_file.src")
    .expect("Something went wrong when reading the file");
let mut lexer: Lexer = Lexer::new();
lexer.read_source(&source);
loop {
    let token = lexer.next_token();
    if token.is_none() {
        break;
    }
    println!("{}", token.unwrap());
}
```

The `Lexer` has a state machine implementation in it. I used a Rust crate `rst_fsm` to help with the implementations of the state machine. I don't want to write woo much details here, but the key point is that the state machine is implemented by implementing two Rust trait functions (usually called interfaces in other languages): `transition` and `output`. As you can tell from the name, `transition` defines the transition function of the DFA and `output` defines the output the state machine. The output of the state machine is a `TokenType` enum, which consists of two sub-enums, `ValidTokenType` and `InvalidTokenType`. The `Lexer` gets the token type from the DFA, the lexeme from its reading buffer, to generate a token. In case the token has a `TokenType` of `InvalidTokenType`, a lexical error will be raised.

Use of tools

- Rust programming language and its standard library (https://www.rust-lang.org/)
- rust_fsm: A framework for building finite state machines in Rust (https://docs.rs/rust-fsm/0.6.0/rust_fsm/)

How to run the program (macOS)

The executable binary `lexdriver` is in the root directory. To run the program, put the input in the root directory and rename them as `<file_name>.src`. Now you can run it in terminal by typing:

```
./lexdriver
```