Lexical Analyzer Design

Tatum Alenko

January 27, 2019

Table of Contents

Contents

1 Lexical Specifications

The lexical definitions of the atomic lexical elements along with their regular expressions are given in the table that follows.

Lexical Definition	Regular Expression
`id ::= Letter alphanum*`	`([a-zA-Z](([a-zA-Z]|[0-9]|_))*)`
`alphanum ::= Letter | digit | _`	`([a-zA-Z]|[0-9]|_)`
`integer ::= nonzero digit* | 0`	`(([1-9][0-9]*)|0)`
`float ::= integer fraction [e[+|-] integer]`	`(((([1-9][0-9]*)|0))(((\.[0-9]*[1-9])|\.0))(e(\+|-)? ((([1-9][0-9]*)|0)))?)`
`fraction ::= .digit* nonzero | .0`	`((\.[0-9]*[1-9])|\.0)`
`Letter ::= a..z | A..Z`	`[a-zA-Z]`
`digit ::= 0..9`	`[0-9]`
`nonzero ::= 1..9`	`[1-9]`
`partialfloat ::= integer.digit* [e[+|-]]`	`((((([1-9][0-9]*)|0))(\.)([0-9]*)(e(\+|-)?)?)`

Note that only the `id`, `float`, and `integer` lexical elements are considered to be valid tokens, whereas the remaining ones are considered to be partial tokens. A better distinction between their meanings will be given in a later section. Moreover, an additional lexical element, the `partialfloat`, was defined for use in the lexical analyzer. This will be given proper justification in section 3.

For the sake of completeness, a snippet of F# code is given on the next page which outlines the `Token` (discriminated union) type. This type describes every possible partial and final tokens made use of in the lexical analyzer. Furthermore, in comments next to each token shows the regular expression literal (or the name of the element from the previous table) associated to it.

One can quickly notice that the lexical definitions of the inline and block comment to not be present. This was an inherent design decision and will also be explained in section 3.

```
type Token =
    // Atomic lexical elements
    | Letter // letter
    | Nonzero // nonzero
    | Digit // digit
    | IntegerLiteral // integer
    | Fraction // fraction
    | PartialFloat // partialfloat
    | FloatLiteral // float
    | Alphanum // alphanum
    | Id // id
    // Operators (single character)
    | Equal // =
    | Plus // +
    | Minus // -
    | Asterisk // *
    | Lt // <
    | Gt // >
    | Colon // :
    | Slash // /
    | SemiColon // ;
    | Comma // ,
    | Period // .
    | OpenBracket // (
    | ClosedBracket // )
    | OpenSquareBracket // [
    | ClosedSquareBracket // ]
    | OpenBrace // {
    | ClosedBrace // }
    // Operators (double character)
    | EqualEqual // ==
    | LtEqual // <=
    | GtEqual // >=
    | LtGt // <>
    | ColonColon // ::
    | SlashSlash // //
    | SlashAsterisk // /*
    | AsteriskSlash // */
    // Reserved words
    | If // if
    | Then // then
    | Else // else
    | While // while
    | Class // class
    | Integer // integer
    | Float // float
    | Do // do
    | End // end
    | Public // public
    | Private // private
    | Or // or
    | And // and
    | Not // not
    | Read // read
    | Write // write
    | Return // return
    | Main // main
    | Inherits // inherits
    | Local // local
    // Invalid (anything else)
    | Invalid // .*
```

2 Finite State Automaton

A manual lexical analyzer was implemented. For the purposes of the this assignment's requirements, the finite state automata for the atomic lexical elements (and overall) were diagramed; however, the form of the finite state automata are nondeterministic. The reasoning for this decision will be given, once again, in *section 3*.

The overall NFA obtained is given in the following figure.

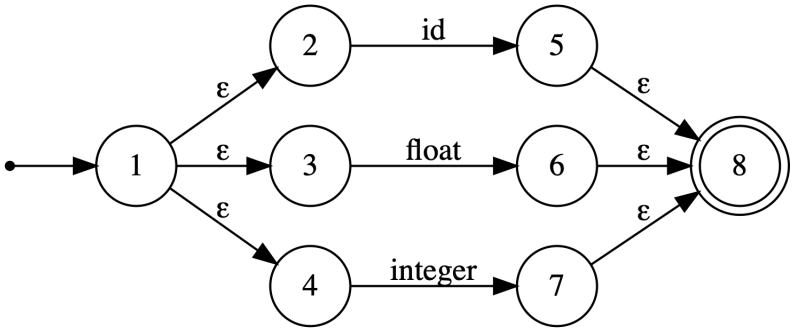

Figure 1: Overall NFA

The individual components of the NFA (`id`, `float`, and `integer`) were chosen to be depicted in separate diagrams for the sake of clarity. They are given in the following figures.

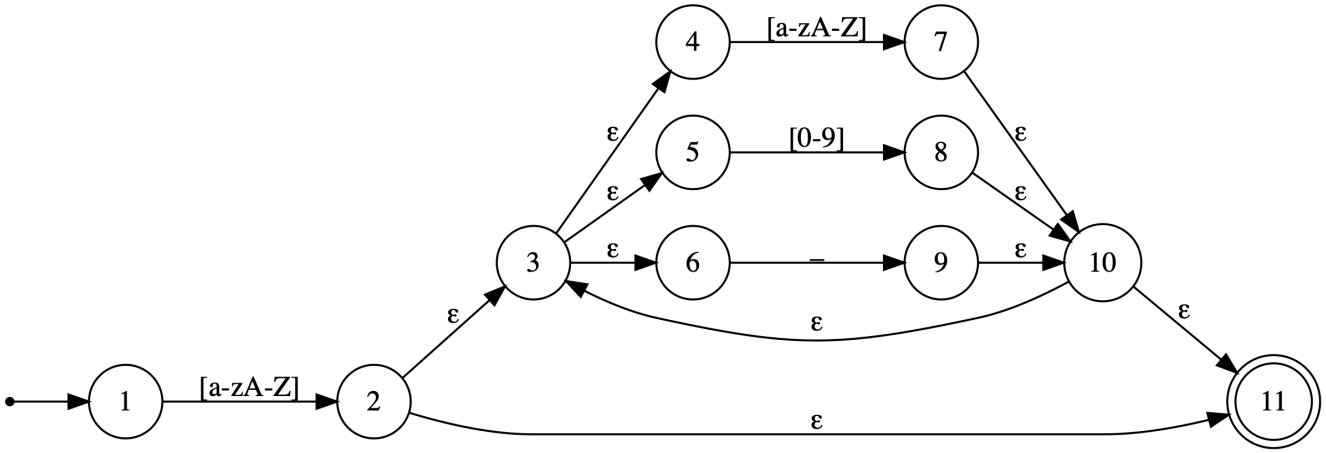

Figure 2: `id` NFA

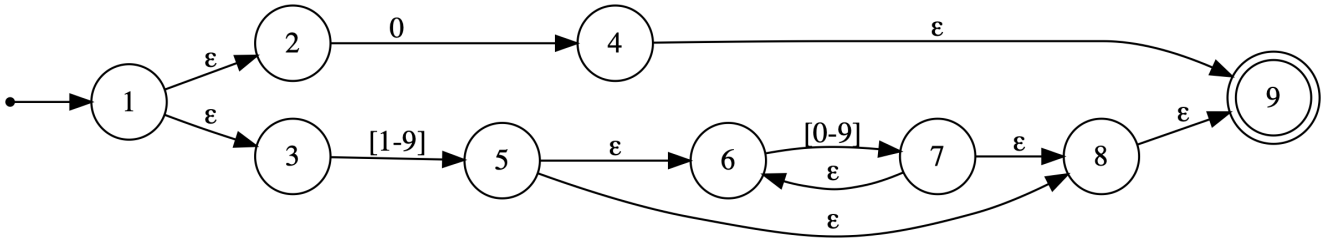

Figure 3: `integer` NFA

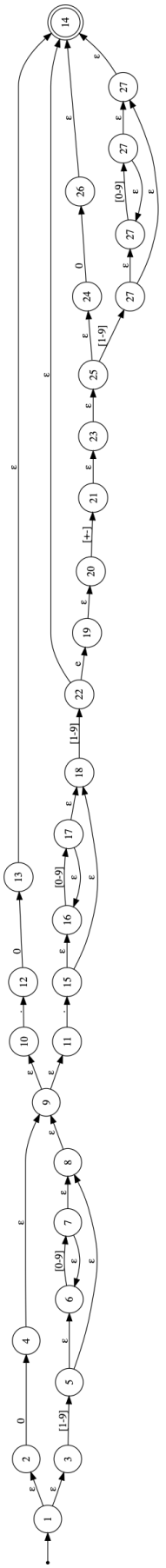

Figure 4: `float` NFA

3 Design

3.1 Overall description

As previously mentioned, a manual lexical analyzer was implemented. This was chosen over using a table-driven lexical analyzer simply because the generation of such a table involved error-prone and tedious steps. Of all the tools available online and in various software distributions, none of these were capable of taking character sets (e.g. `[a-zA-Z]`). The only full-proof method of generating a DFA diagram/table would be to convert by hand the NFA to a DFA using the Rabin-Scott Powerset Construction. Not only does that imply keeping track for the set of all possible characters in the lexical definition, but making any minor adjustements to the regular expressions would involve a rework of the entire DFA generation process, especially when the individual atomic lexical elements are interdependent (e.g. when an `id`'s character set can be cross-coupled with a reserved word).

The manual implementation of the lexical analyzer involves a trivial use of the .NET framework's `Regex` class to iteratively capture possible valid tokens while applying the principle of maximum munch. The principle of maximum munch is quite simple in nature. The basic idea behind the principle is to continuously try to test if a valid token is matched within a character stream. At every character increment of a lexeme, if that lexeme is found to match a valid token, then the algorithm attempts to match the lexeme including the next character in the stream (if present). If another match is found, then the result is tracked and the next lexeme is tested with an additional character from the stream. Either the entire stream is matched as a valid token, or at some point no match is found and the previous match is returned as the result (if present, else it simply returns an invalid token result).

An example might help illustrate the principle idea behind the algorithm.

Example: `1234!`. The algorithm starts by taking only the first element of this stream, i.e. `1`. It attempts to match it with one of the `Token` types (each with their associated regular expressions). If a match is found, in this case the `IntegerLiteral` token, then it is stored and the next lexeme is chosen as the preceeding one and the next character up in the stream, i.e. `12`. This is then again matched with a `IntegerLiteral`, its result is stored (including its position within the stream), and the process is repeated. When `1234!` finally happens to be matched, the result will match against no token. At this point, the algorithm returns the previously stored valid lexeme recognized as a token. The algorithm then starts anew, but now with the stream consisting only of `!`, which is also not matched against any valid token, and hence is returned as an `InvalidCharacter` lexical error.

The algorithm also deals with possible *ambiguities*. For example, when matching the regular expression for the reserved word `do` and for an `id` token with the same lexeme, both in theory will match. At this point, if no further lexeme can be "munched" further into a valid token, the algorithm selects the token with the highest precedence. This precedence is laid out trivially by using an array of all the valid tokens defined in order of highest priority to lowest. This ensures that, when faced with a list of possibly matched tokens, the first token within the list is chosen as the result since it was placed in the matched list prior to a lower priority token (when tested against its regular expression). By placing the reserved words in the list before the `id` token, if said reserved word is matched alongside an `id`, then the reserved word will be taken as the resulting one.

As previously mentioned, the lexical definitions for inline and block comments were omitted. This was intentionally chosen toward the design of the compiler. Instead of letting the lexer be responsible for scanning and disregarding comments, it will instead be delegated to the parser. There are few ramifications in postponing the responsibility to the parser. The decision was mainly motivated by the fact that to properly tokenize block comments, the tokenization would need to lookahead multiple lines of the character stream. In a sense, comments hold a certain aspect of context in order to be properly recognized. This leads to the interpretation that comments (especially block) are somewhat of an expression (starts with a open block comment token, anything else is diregarded, and finally the close block comment token is captured, otherwise it is an invalid block comment "expression"). Because of this interpretation of a comment expression, and the fact that nested comments could potentially complexify the lexer, it was chosen to simply tokenize the open and closed brackets for the comments, but not the comment expressions themselves. As a result, whatever characters that occur after the start of a open comment token will be interpreted as one of the other tokens (if any are found to be valid).

It was noted that an additional partial token type was needed in the lexical definition, namely that of `partialfloat`. The reason for its need is because when performing maximum munch in the algorithm, incomplete float lexemes, such as `1.0e` for example, need to be within the lexical tokens in order to attempt to match the complete lexeme that would form a valid final `float` token. Although on its own it is not a valid *final* token, its partial match allows for the attempt to match the remaining characters in the stream. If the attempt to match a complete float is

unsuccessful and only a `partialfloat` remains after say a whitespace following the character `e` of a partial float, then the resulting token is invalid because although a `partialfloat` is found, it is not deemed to be a final token.

3.2 Component descriptions

The program solution (in .NET terms) is split into two distinct projects: `Moon` and `Moon.Tests`. `Moon` contains the source code for the lexer (the `Moon.Lexer` module located in `Moon/Lexer.fs`), a utils module (`Moon/Utils.fs`) containing various helper functions, and the `Main` module (located in `Moon/Main.fs`) which contains the entry point driver for the command-line program. The following subsections summarize the roles of each of the components inside the `Moon.Lexer` module.

3.2.1 Discriminated Unions (DUs)

`Token`: Represents the possible partial and final lexical token types.

`LexicalError`: Represents the possible types of lexical errors that can occur (either `InvalidCharacter` or `InvalidNumber`).

`Outcome`: Represents the possible types of the resulting tokenization process. Either it can be a `Result` record or an `Error` record.

`InputType`: Represents the type of input to supply the `tokenization` function. Either it can be a `FilePath` or some `Text`.

3.2.2 Records

`PartialResult`: Represents a structure to contain information about the result of `tokenizeChars` (i.e. the result from sweeping across a line and finding the first valid token, if any).

`Error`: Represents a structure to contain information about the error of the `tokenize` function if an invalid token is found.

`Result`: Represents a structure to contain information about the successful return of the `tokenize` function if a valid token is found.

3.2.3 Functions

`asString(token:Token):string`: Pattern matches the given `Token` type and returns the associated regex string expression.

`asRegex(token:Token):Regex`: Pattern matches the givne `Token` type and returns the `Regex` object representation of its associated regex expression.

`isMatch(token:Token, lexeme:string):bool`: Tests the given lexeme against the token's associated regex expression.

`tryMatch(lexeme:string, token:Token):Token*string option`: Simply wraps the `isMatch` function return value in an optional.

`matched(lexeme:string):Token list`: Tests the lexeme against all possible tokens and returns a filtered list of the successfully matched tokens.

`makeOutcome(partialResult:PartialResult, line:int, column:int):Outcome`: Constructs an `Outcome` type from a `PartialResult` and other location values such as line and column number.

`tokenizeChars(stream:char list):PartialResult list`: Iteratively tries to match a successive characters from the character stream until a token is not found and returns the previously found stored if any inside a `PartialResult` container.

`tokenizeStrings(stream:string list):Outcome list`: Iteratively calls the `tokenizeChars` function for each line of text received (each element of the string list) and returns a list of every result as an `Outcome` which designates either a `Result` or `Error`.

`tokenizeFile(filePath:string):Outcome list`: Reads the contents of the provided file, stores in it a string list, and calls `tokenizeStrings`.

`tokenize(input:InputType):Outcome list`: Generic wrapper around `tokenizeStrings` and `tokenizeFile`.

`lexicalErrors(outcomes:Outcome list):Error list`: Extracts the `LexicalError`s from a list of `Outcome`.

`display(outcomes:Outcome list):string`: Generates the string representation of a list of `Outcome`.

`writeTokens(outcomes:Outcome list, path:string):unit`: Write sthe string representation of an `Outcome` list to a file given its path.

`writeErrors(outcomes:Outcome list, path:string):unit`: Writes the detailed string representation of an `Error` filtered list from `Outcome` list to a file given its path.

4 Use of Tools

4.1 Programming language, libraries, and tooling

Many different languages were contemplated for the purposes of this project. One of the main requirements was to make use of a functional programming language due to its strong suitability to compiler design, especially one with a strong type system which favors pattern matching and discriminated unions. Two contenders were considered: OCaml and F#. At first OCaml was tried, but due to lack of proper tooling such as a robust IDE and overall abundant documentation with OCaml, F# was chosen. Being a veteran of IntelliJ, finding a familar comfort with Rider by JetBrains made the transition to a functional language much easier, especially with a runtime debugger, unit test runner, and more provided out of the box.

The only libraries needed were the standard library (such as the `Regex` class and other utilities) and auxiliary testing libraries (such as `Xunit` and `FsUnit`).

4.2 GraphViz

GraphViz was used to generate the finite state machines (FSMs). It was used because manually diagramming the FSMs is a very laborious process. Using the `dot` domain-specific language to structure the FSM was found to be slightly less painful and can generate diagrams of high-quality easily.

4.3 Docker

One of the requested components of this assignment was to be able to deliver an executable for the teaching assistant to run locally. In a Java-bytecode world, this is a fairly trivial endavour. However, with so many different platforms and development stacks, the setup required to get a binary to run can be not so trivial. For this reason, Docker was sought out for its widespread containerization capabilities.