Comp 442: Compiler Design
Concordia University, Montreal
Winter 2024
William Zijie Zhang
Concordia ID: 40176870

Assignment 1

William Zijie Zhang

1 Lexical Specifications

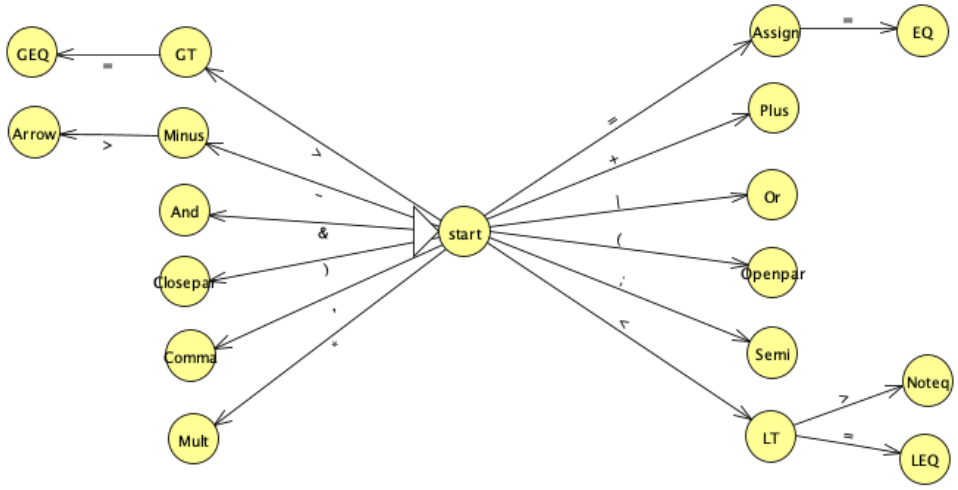
The following are the atomic lexical elements of the language:
id ::= letter alphanum*
alphanum ::= letter | digit | _
integer ::= nonzero digit* | 0
float ::= integer fraction [e[+|−] integer]
fraction ::= .digit* nonzero | .0
letter ::= [a..z] | [A..Z]
digit ::= [0..9]
non-zero ::= [1..9]
The only change I have made to the lexical elements is adding square brackets around digit, non-zero and letters to represent matching one character in the range of characters in brackets.
The following image represents the list of operators, punctuation and reserved keywords:

==	+	|	(	;	if	public	read
<>	-	&	)	,	then	private	write
<	*	!	{	.	else	func	return
>	/		}	:	integer	var	self
<=	=		[	->	float	struct	inherits
>=			]		void	while	let
							impl

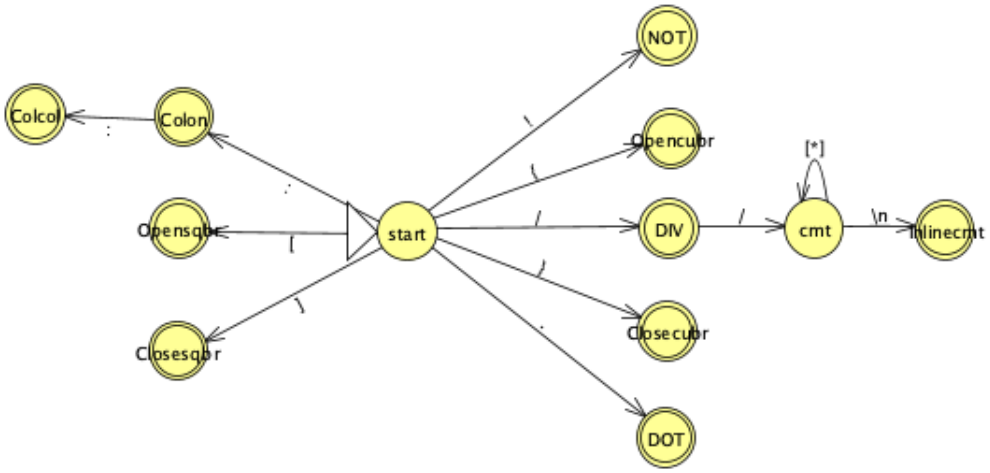
My way of creating tokens for the reserved keywords will be to check if an identifier is identical to one of the following keywords, more details will be given later.

2 Finite Automaton

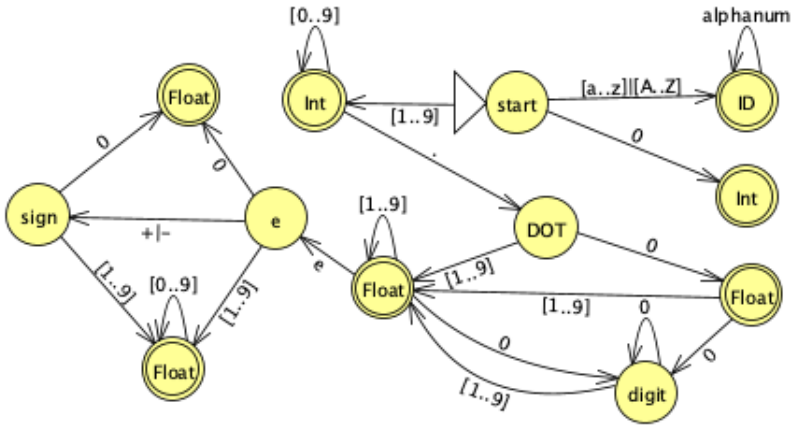
The following three images represent minimal DFAs representing the lexer. In the first image, all states are final states. Most of the transitions are for the operators and they all create unique tokens. The third image goes more into detail about the atomic lexical elements of the language.
This part made extensive use of regular expressions to match identifiers, integers and floats. I made use of an online tool that could convert regular expressions to NFAs then to DFAs. Finally, if you were to combine the three DFAs, you would more or less get the exact representation of all possible tokens in the lexeme.



In the second image below, the division operator can lead to comments. I used the format $[*]$ to represent matching any character. The only way to reach a final state from the *cmt* state is to reach the end of the line, hence creating an inline comment.



The following describes the DFA for the atomic lexical elements of our language.



3 Design and Implementation

In this section I will be detailing the structure of my lexer and how I went about implementing the DFA presented above.
Two options were presented by Professor Paquet on how to go about implementing the lexer: table-driven and hand-written. I chose to go with hand-written, because it allowed me to do fast prototyping and get a feel for the helper functions I would need to write. The assignment hinted that we would need a *Token* data structure to store information about them, such as their type and location. So I started with creating a class for the *Token* object and modified its *toString* function to match that of the expected output. Then I realized that there was no shortcut to keep track of every token type, so I made use of an Enum for *TokenType* that would allow me to access a name and a corresponding value, which will came in handy later. I decided to group the reserved keywords together with values less than 100, operators had values less than 200 and the atomic elements of the lexeme would have values over 200.
Now onto the lexer. After being able to read the entire input, I made note of a few must-haves: keeping track of an index and its corresponding character, being able to peek the next character without modifying the current index and also being able to backtrack the index. Those helper functions were not difficult to implement, but they allowed me to really simplify the code for the most important method: *getToken*. In version 3.10 of Python, *match* statements were introduced which reduces a lot of code from the if-elif alternative. I started off with matching all the single operators and instantiating a *Token* with the corresponding *TokenType*. Then I moved on to more complex tokens such as the $eq, leq, arrow$ and *inlinecmt* which required making use of the peek function to verify that the following character was also valid.
Now onto the difficult parts of the lexer. Firstly, I took care of nested block comments by introducing a counter. I would increment the counter each time I encountered a $'/'$ followed by a $'*'$ and I would decrement the counter each time I encountered a $'*'$ followed by a $'/'$. The index would continue increasing as long as the counter didn't reach zero. Then, I managed to match the identifiers easily by making use of python's built-in string helper functions *isalnum*() and *isalpha*(). After matching an identifier, I would create a reserved keyword if the identifier name was a *TokenType* and its value was less than 100. Finally, to match floating point numbers, I created two new helper functions for matching integers and fractions. Essentially, it was separating behavior between zero and non-zero, making use of *backtrack* whenever necessary. I also had to take into account the dot before the fraction and the possible $e[+|-]$ after the fraction.

4 Tools

In this section I will be describing the tools I have used in the analysis and implementation of the lexer. To create the DFAs for the lexemes and the operators, I used JFLAP which is a software for formal languages. I decided to use JFLAP because I had previous experience using it in COMP335, I was also able to verify my implementation by generating some test cases. Then once I had to write the states for the floating point number, I referred to NFA2DFA which allowed me to generate the corresponding DFA from the regular expression for the integer and fraction components of the float. Finally, in addition to the resources provided by Professor Paquet, I referred to another tutorial to help me gain a greater understanding of the lexer and what would be required.