Assignment 1 Lexical Analyzer

Section 1 Lexical specifications expressed as regular expressions

- id: [a..zA..Z]([a..zA..Z]|[0..9]|_)*
- integer: [1..9][0..9]*|0
- float: ([1..9][0..9]*|0).([0..9]*[1..9]|0)(e(+|-)?[1..9][0..9]*|0)?
- For operators, punctuation and reserved keywords: Can match directly based on tables.

Operators, punctuation and reserved words

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

- Single line comments: //.*\r\n
- Multiline comments: /*(.|\n)**/

Section 2 Finite statue automation

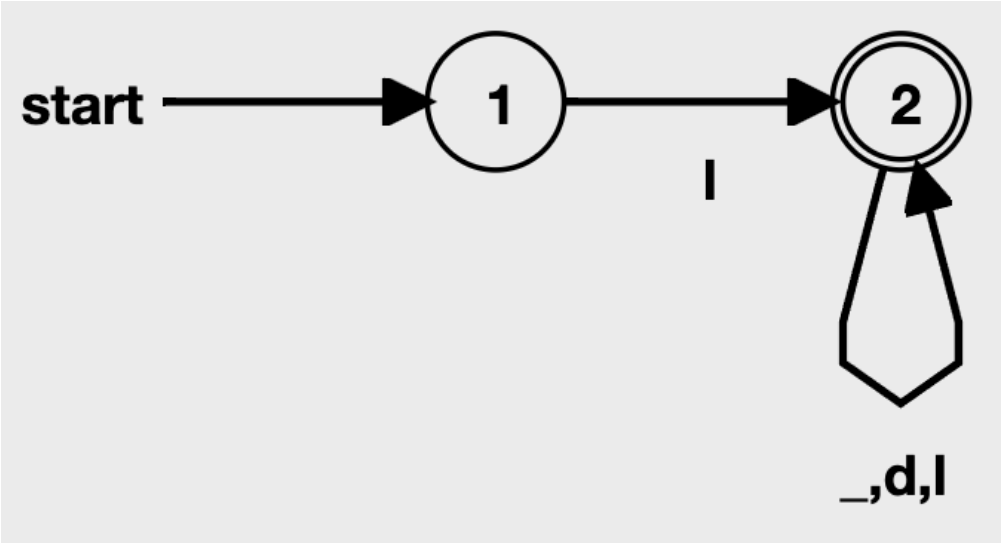
To reduce the state and transition in DFA, using following mappings to represent char groups

- Letter: l -> [a..zA..Z]
- Zero: 0 -> 0
- Nonzero: n -> [1..9]
- Other symbol: & -> |&!(){}[];,
- Space: sp

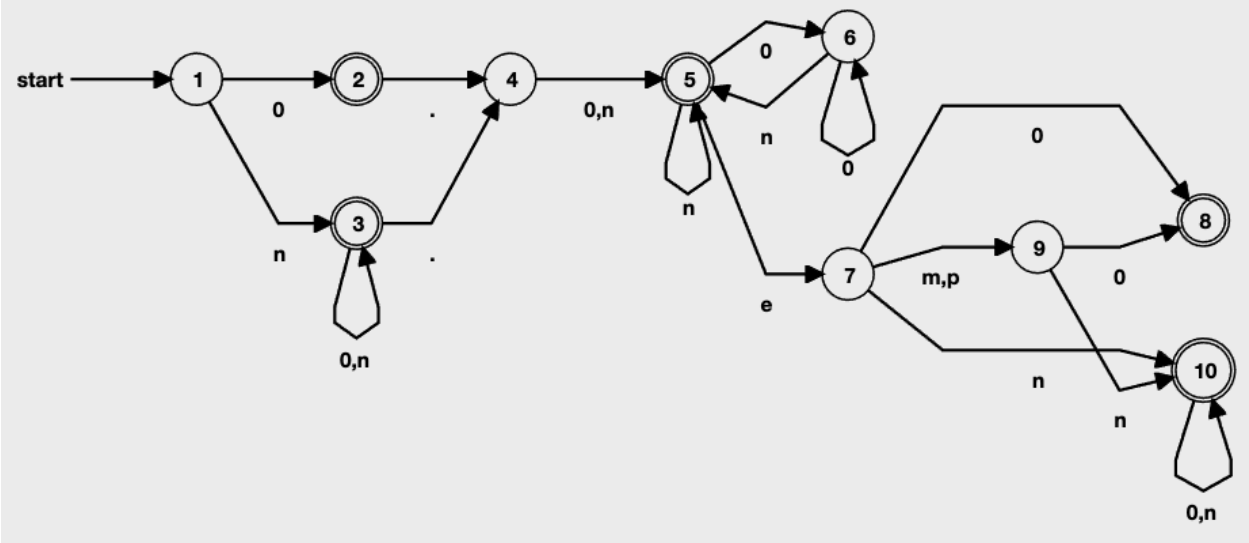
Regular expression can be rewritten as following:

- id: l(l|0|n|_)*
- integer: n(0|n)*|0
- Float: (n(0|n)*|0).(((0|n)*n)|0)(e(+|-)?(n(0|n)*|0))?

DFA for ID

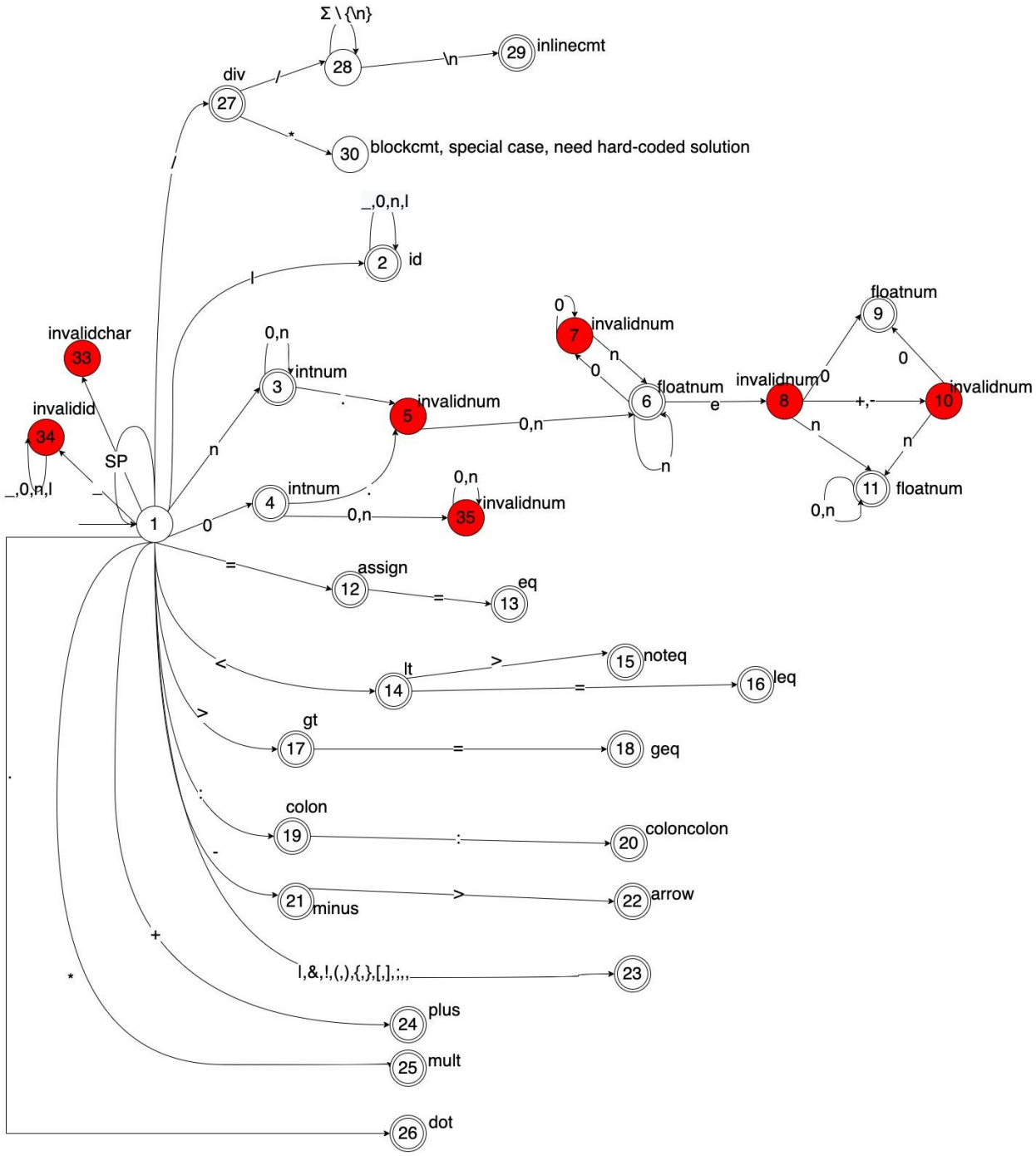


DFA for Integer and float (m for -, p for +)



Combine all parts to draw DFA for lexical analyzer
note: trap state is ignored to have a clear state diagram

Σ = {l, 0, n, e, _, +, -, ., =, <, >, :, *, sp, /, &, \n}



Section 3 Design

The project is implemented in Java following the Table-driven scanner approach, where the state transition table is implemented to represent the DFA. Here is the state transition table I used in implementation. In the *LexicalAnalyzer* class, I express the state transition function, state name, and symbols as String. By parsing the String, I am able to build a state transition table as a 2D array, and also store relevant state information in the proper data structure for later use. Above steps are done in *LexicalAnalyzer* class constructor.

My table-driven scanner algorithm is slightly differ than the one presented in lecture. Instead of defining the state that needs to “backtrack”. I consider all states need to perform “backtrack”, which means if an input character leads a state transite to a trap state, always backup this character and check previous state for token generation. In other words, each state transition is terminated by reading a character that is part of the next token. The advantage is that less states are needed in the state transition table, since the “backtrack” approach in the lecture requires additional states for “backtrack” and also an additional variable is needed to indicate a “backtrack” state. My “nextToken” algorithm shows below

```
public Token nextToken() throws Exception {
	Token token = null;
	int currentState = START_STATE;
	int nextState = START_STATE;
	String lexeme = "";

	do {
		char lookup = nextChar();
		if (lookup == EOF) {
			return null;
		}
		nextState = tb[currentState][getSymbolNum(lookup)];
		if (nextState == TRAP_STATE) {
			storeChar(lookup);
			token = createToken(currentState, lexeme, line);
		} else {
			currentState = nextState;
			if (currentState != START_STATE) {
				lexeme += lookup;
			}
			if (currentState == BLOCK_CMT_STATE) {
				return buildBlockCmt(lexeme);
			}
		}
	} while (token == null);

	return token;
}
```

However, there is a special case that needs to be handled, which is the block comment. In the block comment case, we need to support imbricated block comments, which cannot be handled by DFA. Because to implement it, the implementation needs to have “memory” that remembers how many open comments match with close comments. We would need a solution such as a PDA. To handle this special case, I implemented a hard-coded solution for block comment. When an open comment is encountered, the program enters a *BLOCK_CMT_STATE*, this

triggers a *buildBockCmt* function to continue reading the characters by counting the number of open comment and close comment.

The *LexicalAnalyzer* class has a *charBuffer*, “backtrack” character is stored inside. Whenever calling the *nextChar()* to retrieve the character, the character in the buffer will be returned first. If the buffer is empty, *nextChar()* returns the next character in the input file.

Driver class opens the input .src file and passes it to *LexicalAnalyzer*. Then, *Driver* class writes the tokens and errors in .outlextokens and outlexerrors. *Driver* class can be used in two ways, either by sending .src file path in command line argument, or by running with the default .src test files in ./input folder.

Token class defines the token schema returned by *LexicalAnalyzer*.

Section 4 Use of tools

- Java programming language: this lexical analyzer and entire compiler project is intended to build in Java.
- Maven: Apache Maven is a software project management and comprehension tool. It’s great for Java project package management and build. The jar executable in the submitted zip file is built by Maven.
- Online DFA generator tool https://cyberzhg.github.io/toolbox/min_dfa. This is a great tool to generate DFA for each regular expression. I use this tool to generate DFA for each part to ensure the correctness.
- Draw.io is a flowchart and online diagram maker software. I use this tool to draw the complete DFA. It also provides better graphical features to label the diagram.
- Eclipse IDE for coding

Assignment Submission

/src - contains the source code
/input - contains the test files
/lexdrive.jar - executable

Jar executable is built by Maven using Java11, which requires Java Runtime Environment 11+
If you wish to rebuild the jar, use the pom.xml. Otherwise, you can compile and run the program from source code.

How to run jar executable

Method1: Run with default .src files in /input folder

```
$ java -jar comp6421-a1-jar-with-dependencies.jar
```

This will use all .src files in ./input folder, once the program is done, there will be a ./output folder contains all .outlextokens and .outlexerrors files to each .src

Method2: Run with specific xxx.src file
e.g.

```
$ java -jar comp6421-a1-jar-with-dependencies.jar input/lexpositivegrading.src
```

Output .outlextokens and .outlexerrors will be in ./output folder