Report
Compiler Design
comp-6421

Assignment 1
Lexical Analyser

Professor: Dr Joey Paquet

Neona Sheetal Pinto
40172626

Lexical Specifications:

As per the requirements of the assignment, the lexical specifications are as follows.

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**
nonzero	::=	**1..9**

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

Comments

- Block comments start with /* and end with */ and may span over multiple lines.
- Inline comments start with // and end with the end of the line they appear in.

For the notations in the Transition table, I have substituted the sequences with characters.

Notations: : {'L', 'e', '0', 'N', '_', '"', '/', '*', '=', '<', '>', ':', '+', '-', '&', '.', ' ', '\n', '\r', '~', '\u0000'}

Notation	Character
[A..Z|a…Z]	L(except e)
e	e
0	0

[1..9]	N(non zero)
|&!(){}[];,	&
Space, \n, \r, \t	‘ ‘
Invalid char	~
: . * = < > _ + - * /	No changes

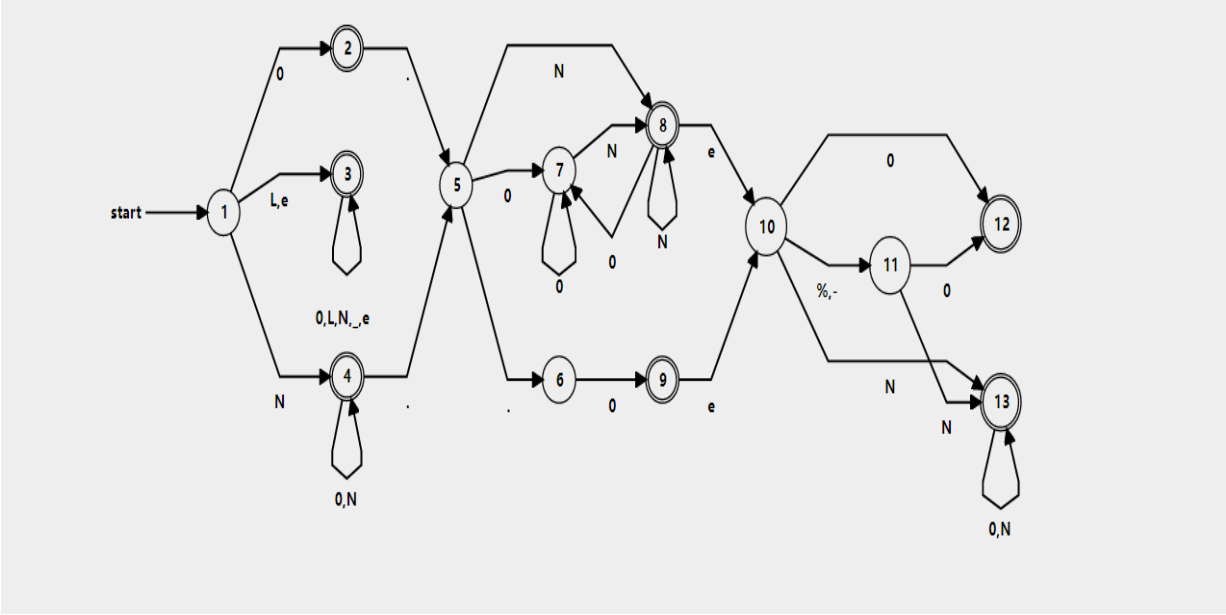
Regular expressions:

- id::= (L|e)(L|e|0|N|_)*
- integer::= N(0|N)*|0
- float::= (N(0|N)*|0).(((0|N)* N)|0)(e(+|-)?(N(0|N)*|0))?

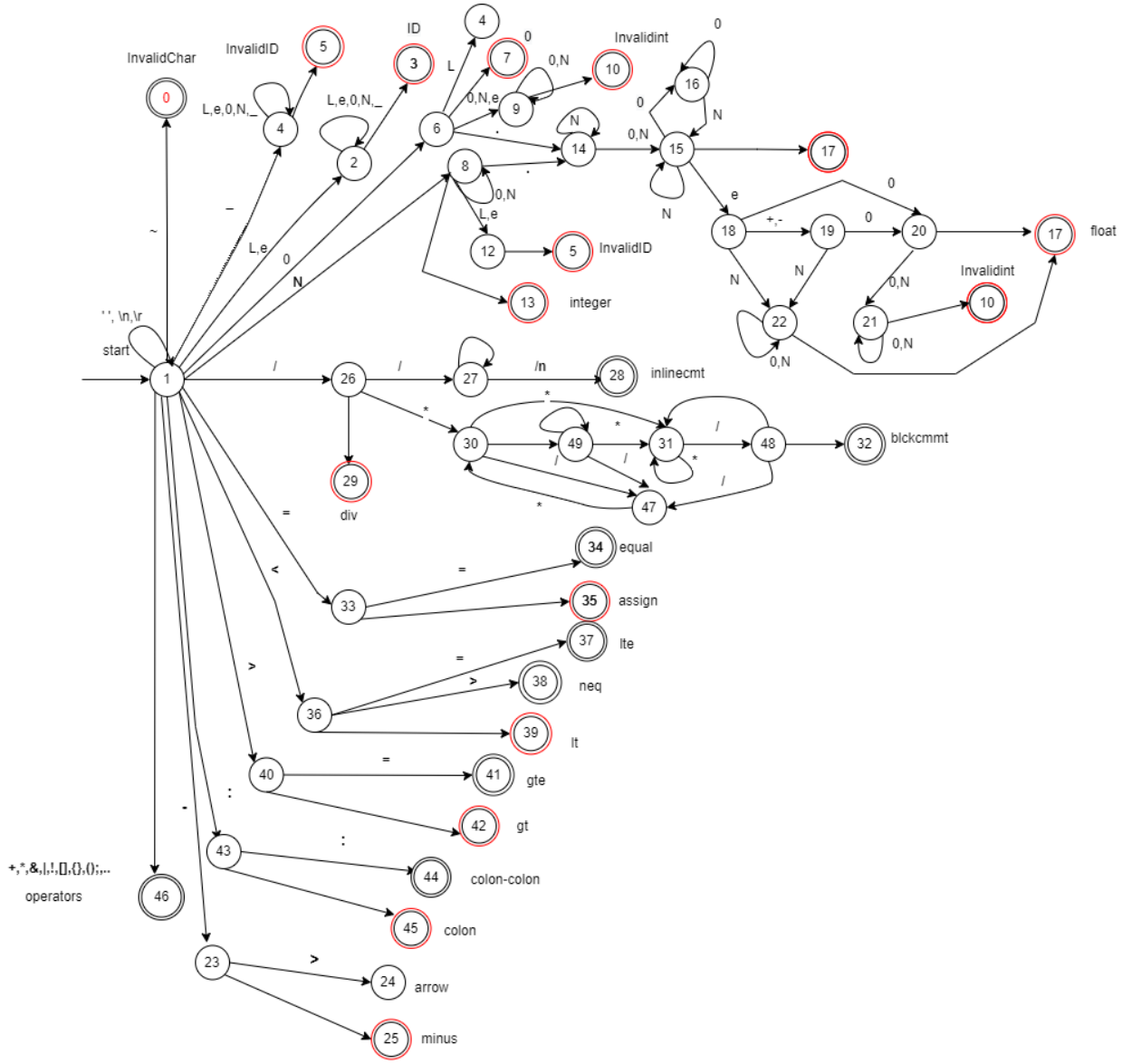
Note: ‘e’ has been excluded from the letter and ‘0’ from the digits to keep it deterministic.

Finite State Automata:

- DFA for ID, Integer and Float using the “Cyberzhg” tool. This minified DFA contains only the regular expression for the id, integer and float.



- The overall DFA for the Lexical Analyser.It includes the ID, integer, float, all the operators, comments, and other characters included in the regular expression. I have also added the invalid cases in the DFA.



Design:

The design is based on the Table-driven scanner, which is reading character by character from the file(.src), and based on which the next state is decided as per the transition table. We check if the state is a final state in which case it will create a valid token or if an invalid character or condition is met we create an error token. Keep repeating the process until the end of the file.

Classes:

- ***State.java*** : The class represents the state in DFA, including attributes such as the ID of the state(an integer), the name of the state, etc. The details of the transition table are recorded by the Map(Character, Integer). For each state(an integer), there is a map with the key as input character and the value as the next state(Integer).
- Token.java: The class contains information such as the token type, the lexeme, the location.
- ***LexicalAnalyser.java:*** The class of LexicalAnalyzer is responsible for scanning every file, tokenizing and writing to output files. It contains an array of States, where it stores the entire transition table. The state is represented as an integer, in order to use an array which can utilize the index as an integer.
- ***LexicalDriver.java*** : reads every file and creates a LexicalAnalyzer object to handle the task of tokenizing.

Note: Detailed explanation for every class and function is present in the source code.

Use of Tools

1. ***Cyberzhg*** : It's an online tool which is user-friendly to create DFA using the regular expression. On entering the regular expression generated above you can easily get the DFA without any hassle. It's a free online tool where you can use it to create DFA. It also provides a transition table for the regular expression.
2. ***IntelliJ IDEA***: IDE platform for writing the source code. The IDE has smooth, efficient and clever code suggestions which makes it easy to type. Debugging is much faster compared to other IDE's.
3. ***LucidChart***: It's an online tool used to draw diagrams. I have used it for creating the overall DFA and the class diagram structure. The tool has many options for drawing and is user friendly. Every change is saved to the drive which avoids any mishaps.
4. ***Java.util***: Used util files in the source code for creating tables, hashmaps,. etc.