

Compiling Principles

Lab1 Lexical Analyzer

Report

141250017

陈自强

2016/10/16

Table of Contents

1. Motivation/Aim	3
2. Content description.....	3
3. Ideas/Methods	3
4. Assumptions	3
5. Related FA descriptions	5
6. Description of Important Data Structures	6
7. Description of core Algorithms	8
8. Use cases on running.....	8
Simple case:	8
Complicated Case:	9
9. Problems occurred and related solutions	10
10. Your feelings and comments	10

1. Motivation/Aim

In chapter3 Lexical Analysis, we learned some about how to deal with the lexical sequence and transform them into a series of tokens. First, we need to define some regular expressions to assure that which part of the sequence is a legal identifier. Then based on the RE, we construct a NFA, optimize it and finally yield an optimal DFA which has the minimum states. At this time, we can simply use this DFA to help us determine the legal tokens and output it for the next procedure of the program analysis.

In this lab, we try to imitate the process mentioned above to build our own Lexical Analyzer.

2. Content description

The Lexical Analyzer is expected to perform the following functions:

- 1)Read a sequences of characters from a file
- 2)Omit the blank code inside the sequences
- 3)Identify each type of tokens
- 4)Detect simple error or invalid identifiers
- 5)Output detected tokens and report error(s) if exist(s)

3. Ideas/Methods

Generally, there are two ways available to build a Lexical Analyzer. First, follow the process: Res => NFA => DFA => DFA (Optimal) and program based on the optimal DFA. Another available method is to build a program to automatically construct a NFA.

In this lab, I will use the first way.

4. Assumptions

In this lab, we assume that the input is a regular Java source file *.java

In a Java source file, there are some following key elements:

- Key Word: These words are the retained words that cannot be used as identifiers. They include:

abstract	continue	for	new	switch
assert	default	goto	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const	float	native	super	while

- Identifiers: These are the legal identifiers that start with a letter and may consist of a random number of letters, digits, or underscores (_).

Regular Expression: `letter(letter|digit)*`

Notice : In this lab, we assume that identifiers do not start with _.

- Operator: Legal Java Operators we allow in this lab are the following:

+	+=	-	--	*
*=	/	/=	=	==
&		&&		>=
--	<	<=	>	++
>=				

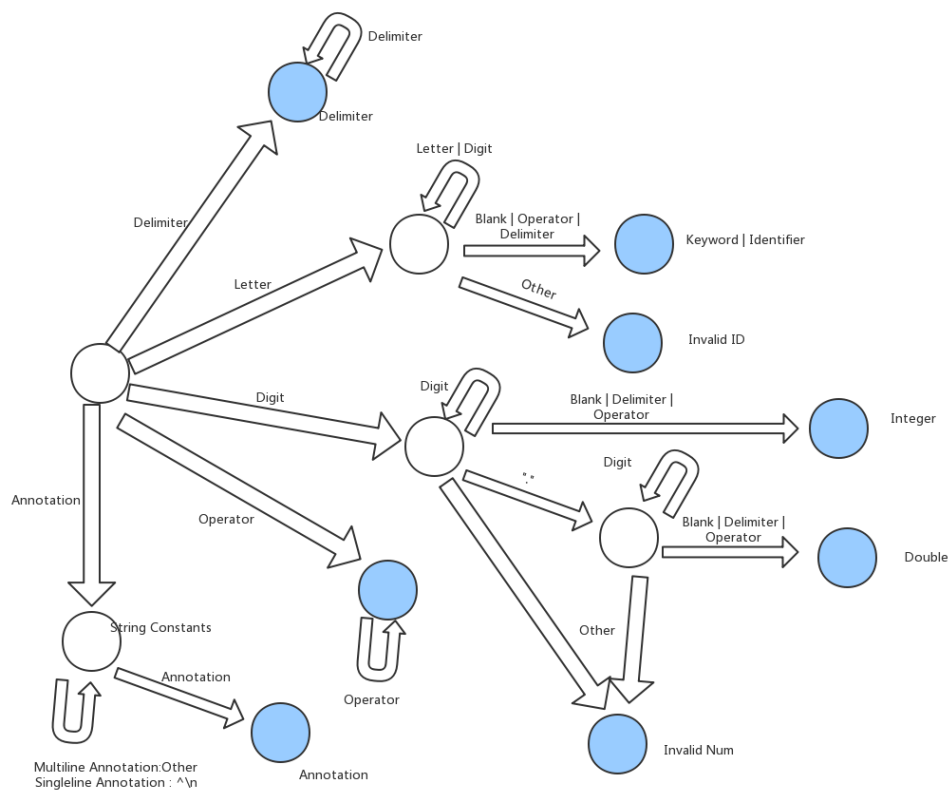
- Delimiter: Legal Java Delimiters we allow in this lab are the following:

%	()	{	}
---	---	---	---	---

!	;	,	.	[
]	?	:	~	^
`				

- String Constants: Anything included in the binate ' or " will be treated as string constants.
- Annotation: Anything between the annotation will be omitted.
Annotation Strings include: `////**/`
- Primitive Value Type: In this lab, we support integer and double as legal primitive value types.
Integer: `digit (digit)*`
Double: `digit (digit)*. digit (digit)*`
- Blank Code: `\n | \t | SPACE`
- Other: other will be treated as invalid identifiers or invalid numbers

5. Related FA descriptions



The plot is the description of the FA.

The related regular expressions of the elements have been described in the above.

6. Description of Important Data Structures

1. Class Constants store those significant constants including token types, keywords, delimiters and so on.

7. Description of core Algorithms

Since many preparation work has been done before the algorithm, the algorithm itself is fairly simple.

It simply use a series of "If-Else" or "Switch-Case" clauses to perform the FA depicted in the above. When the program meets a character, it judge which state it should turn to basing on the FA.

Sometimes you have to read more characters to assure its type, so rollback is also necessary in the algorithm.

8. Use cases on running

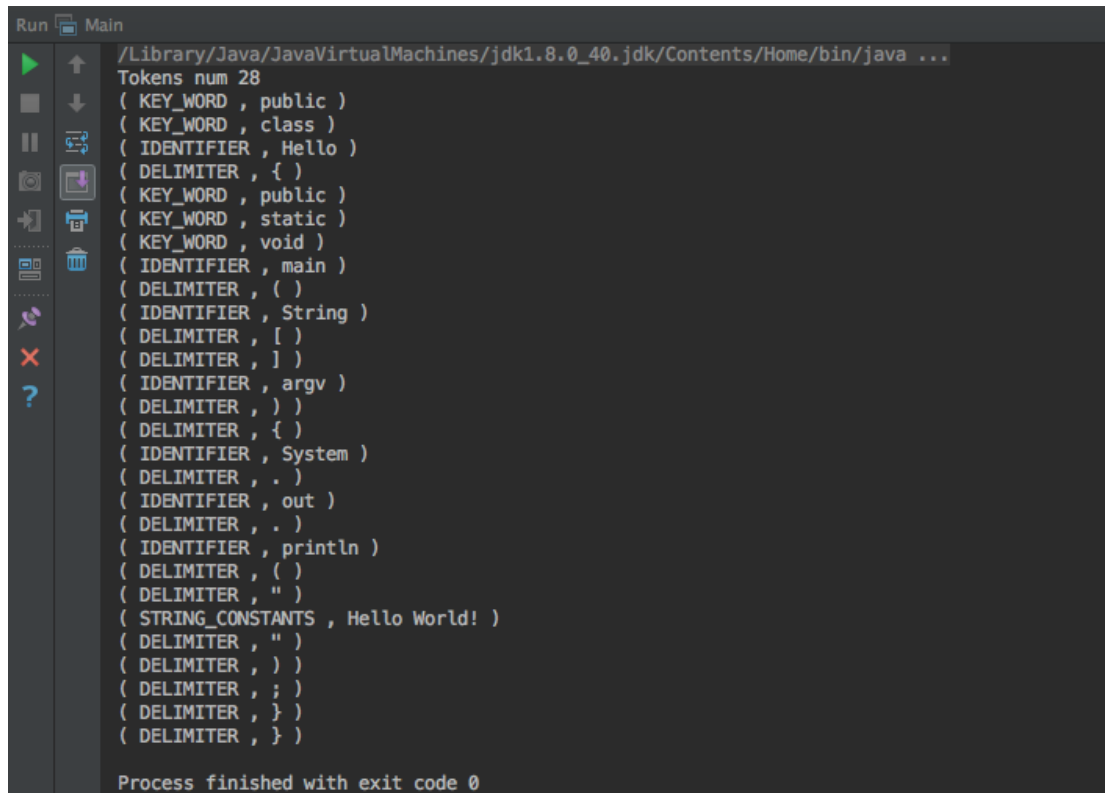
Test cases is rather easy to write. Any Java source files can be test cases.

Simple case:

Input:

```
public class Hello{
    public static void main(String[] argv){
        System.out.println("Hello World!");
    }
}
```

Output:



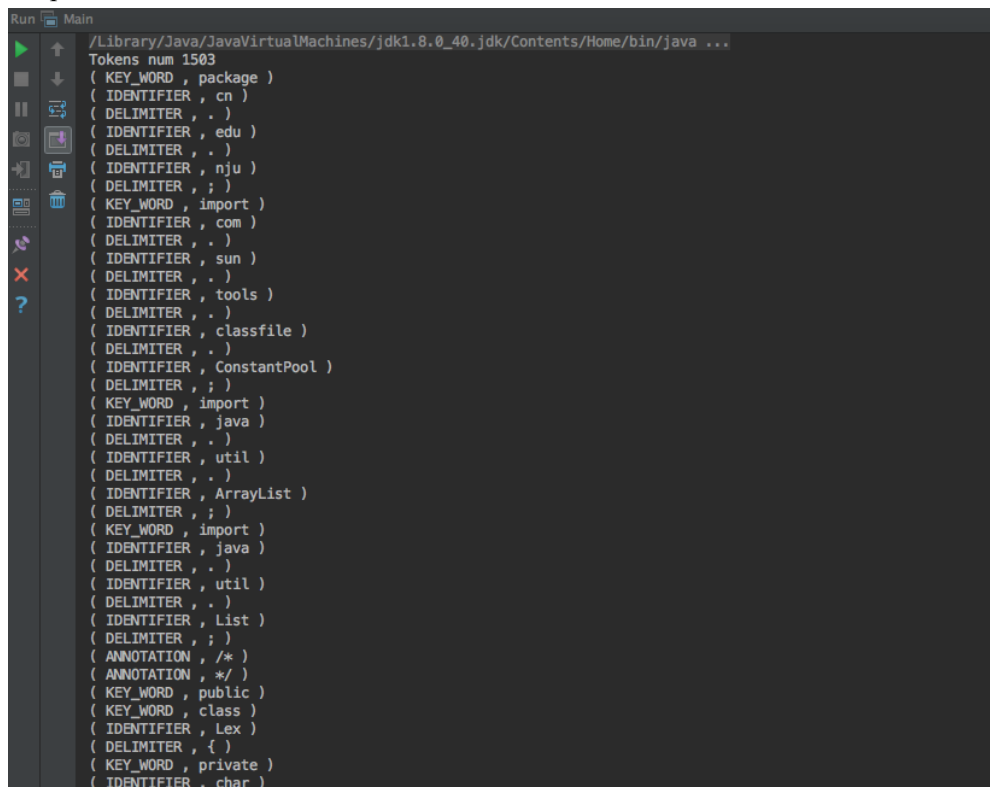
```
Run Main
/Library/Java/JavaVirtualMachines/jdk1.8.0_40.jdk/Contents/Home/bin/java ...
Tokens num 28
( KEY_WORD , public )
( KEY_WORD , class )
( IDENTIFIER , Hello )
( DELIMITER , { )
( KEY_WORD , public )
( KEY_WORD , static )
( KEY_WORD , void )
( IDENTIFIER , main )
( DELIMITER , ( )
( IDENTIFIER , String )
( DELIMITER , [ )
( DELIMITER , ] )
( IDENTIFIER , argv )
( DELIMITER , ) )
( DELIMITER , { )
( IDENTIFIER , System )
( DELIMITER , . )
( IDENTIFIER , out )
( DELIMITER , . )
( IDENTIFIER , println )
( DELIMITER , ( )
( DELIMITER , " )
( STRING_CONSTANTS , Hello World! )
( DELIMITER , " )
( DELIMITER , ) )
( DELIMITER , ; )
( DELIMITER , } )
( DELIMITER , } )

Process finished with exit code 0
```

Complicated Case:

Input: The program source file Lex.java

Output: Part of it as follows:



```
Run Main
/Library/Java/JavaVirtualMachines/jdk1.8.0_40.jdk/Contents/Home/bin/java ...
Tokens num 1503
( KEY_WORD , package )
( IDENTIFIER , cn )
( DELIMITER , . )
( IDENTIFIER , edu )
( DELIMITER , . )
( IDENTIFIER , nju )
( DELIMITER , ; )
( KEY_WORD , import )
( IDENTIFIER , com )
( DELIMITER , . )
( IDENTIFIER , sun )
( DELIMITER , . )
( IDENTIFIER , tools )
( DELIMITER , . )
( IDENTIFIER , classfile )
( DELIMITER , . )
( IDENTIFIER , ConstantPool )
( DELIMITER , ; )
( KEY_WORD , import )
( IDENTIFIER , java )
( DELIMITER , . )
( IDENTIFIER , util )
( DELIMITER , . )
( IDENTIFIER , ArrayList )
( DELIMITER , ; )
( KEY_WORD , import )
( IDENTIFIER , java )
( DELIMITER , . )
( IDENTIFIER , util )
( DELIMITER , . )
( IDENTIFIER , List )
( DELIMITER , ; )
( ANNOTATION , /* )
( ANNOTATION , */ )
( KEY_WORD , public )
( KEY_WORD , class )
( IDENTIFIER , Lex )
( DELIMITER , { )
( KEY_WORD , private )
( IDENTIFIER , char )
```

9. Problems occurred and related solutions

1. Puzzled about where to start.

During the lab, at first it took me rather a long time to find an entrance, I figured it out by reading chapter 3 in the book again.

2. Unclear logic.

While coding the program, I found it arduous to take all the situations into consideration. For instance, numbers may end with an operator, it is a legal Java expression so you have to deal with it specially.

At last I simplified the situations (☺) and depicted a FA states plot to help me get my mind clear.

10. Your feelings and comments

1. Coding is essential in Software Engineering

Before the lab, I had finished the exercises of chapter 3 and held that I had understood it. However, during the lab, I went back to the book again and again and found a lot of puzzled problems. It was not until then did I realize that I was still ignorant about its essence. Thanks to this lab, I once again reinforce my belief that coding is essential in this vocation. Only by coding can you really grab its point.

Talk is cheap, show me the code.

2. Grateful of the Computer Science Pioneers

Honestly speaking, I hate to deal with such low-level problems due to its complication and monotony. I cannot believe what it was if no so many great CS pioneers build us such a high and reliable base. Thanks to them, we are allowed to use the simpler language like Java and Python rather than the distasteful assembly language or C. From my perspective, no everyone is able to understand C or compiling principles. May be simpler language like Python is written for ordinary person like me to build our own programs.