语法分析实验报告

141250019 崔浩

2016-10-25

南京大学软件学院

目录

1. 实验目的 2
2. 内容概述 2
3. 思路方法 2
4. 语言定义 2
   - 4.1 保留字 2
   - 4.2 操作符 3
   - 4.3 分隔符 3
   - 4.4 标识符 4
   - 4.5 数值 4
5. 相关 FA 描述 4
   - 5.1 合并后的 NFA 4
   - 5.2 转换为 DFA 4
   - 5.3 优化 DFA 5
   - 5.4 DFA 模型 6
6. 重要数据结构 6
7. 核心算法描述 7
8. 困难与解决方案 9
9. 测试用例 10
10. 总结 11

1. 实验目的

本实验的目的是自己编写一个词法分析程序，对自定义的程序语言进行词法分析，并输出分析完成后的 Token 序列。

2. 内容概述

本报告描述了编写词法分析程序的过程，包括语言词法的正则表达式定义，生成的 NFA，转化的 DFA 及 DFA 优化。此外，报告描述了程序的主要数据结构和核心算法，以及最终产品的输入输出示例。

3. 思路方法

（1） 定义各个类别的正则表达式
（2） 写出每个正则表达式对应的 NFA
（3） 合并 NFA
（4） 将合并的 NFA 转化为 DFA
（5） 优化 DFA
（6） 根据 DFA 生成词法分析程序

4. 语言定义

4.1 保留字

类别	保留字	类别	保留字
KEYWORD_0	abstract	**KEYWORD_1**	assert
KEYWORD_2	boolean	**KEYWORD_3**	break
KEYWORD_4	byte	**KEYWORD_5**	case
KEYWORD_6	catch	**KEYWORD_7**	char
KEYWORD_8	class	**KEYWORD_9**	const
KEYWORD_10	continue	**KEYWORD_11**	default
KEYWORD_12	do	**KEYWORD_13**	double
KEYWORD_14	else	**KEYWORD_15**	enum
KEYWORD_16	extends	**KEYWORD_17**	final
KEYWORD_18	finally	**KEYWORD_19**	float
KEYWORD_20	for	**KEYWORD_21**	goto
KEYWORD_22	if	**KEYWORD_23**	implements

KEYWORD_24	import	KEYWORD_25	instanceof
KEYWORD_26	int	KEYWORD_27	interface
KEYWORD_28	long	KEYWORD_29	native
KEYWORD_30	new	KEYWORD_31	package
KEYWORD_32	private	KEYWORD_33	protected
KEYWORD_34	public	KEYWORD_35	return
KEYWORD_36	strictfp	KEYWORD_37	short
KEYWORD_38	static	KEYWORD_39	super
KEYWORD_40	switch	KEYWORD_41	synchronized
KEYWORD_42	this	KEYWORD_43	throw
KEYWORD_44	throws	KEYWORD_45	transient
KEYWORD_46	try	KEYWORD_47	void
KEYWORD_48	volatile	KEYWORD_49	while

4.2 操作符

类别	操作符	类别	操作符
OPERATOR_0	=	OPERATOR_1	+
OPERATOR_2	-	OPERATOR_3	*
OPERATOR_4	/	OPERATOR_5	%
OPERATOR_6	>	OPERATOR_7	<
OPERATOR_8	&	OPERATOR_9	|
OPERATOR_10	!	OPERATOR_11	?
OPERATOR_12	:	OPERATOR_13	+=
OPERATOR_14	-=	OPERATOR_15	*=
OPERATOR_16	/=	OPERATOR_17	!=
OPERATOR_18	>=	OPERATOR_19	<=
OPERATOR_20	<<	OPERATOR_21	>>
OPERATOR_22	==	OPERATOR_23	&&
OPERATOR_24	||		

4.3 分隔符

类别	分隔符	类别	分隔符
SEPARATOR_0	;	SEPARATOR_1	{
SEPARATOR_2	}	SEPARATOR_3	[
SEPARATOR_4	]	SEPARATOR_5	(
SEPARATOR_6	)	SEPARATOR_7	,

4.4 标识符

<digit> -> 0|1|2|3|4|5|6|7|8|9
<letter>->a |b |c |d |e |f |g |h |i |j |k |l |m |n |o | p |q |r |s |t |u |v |w | x |y |z

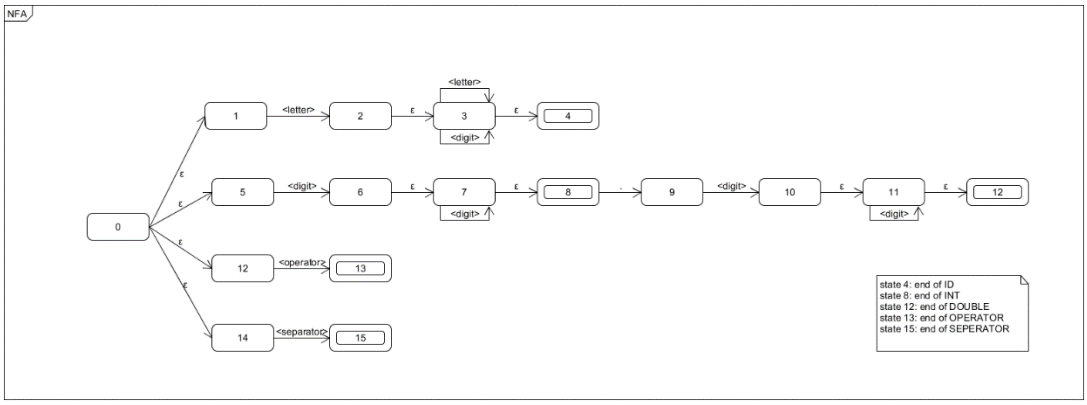
类别	正则表达式
ID	<letter> (<letter> | <digit>) *

4.5 数值

类别	正则表达式
INT	<digit><digit>*
DOUBLE	<digit><digit>*. <digit><digit>*

5. 相关 FA 描述

5.1 合并后的 NFA

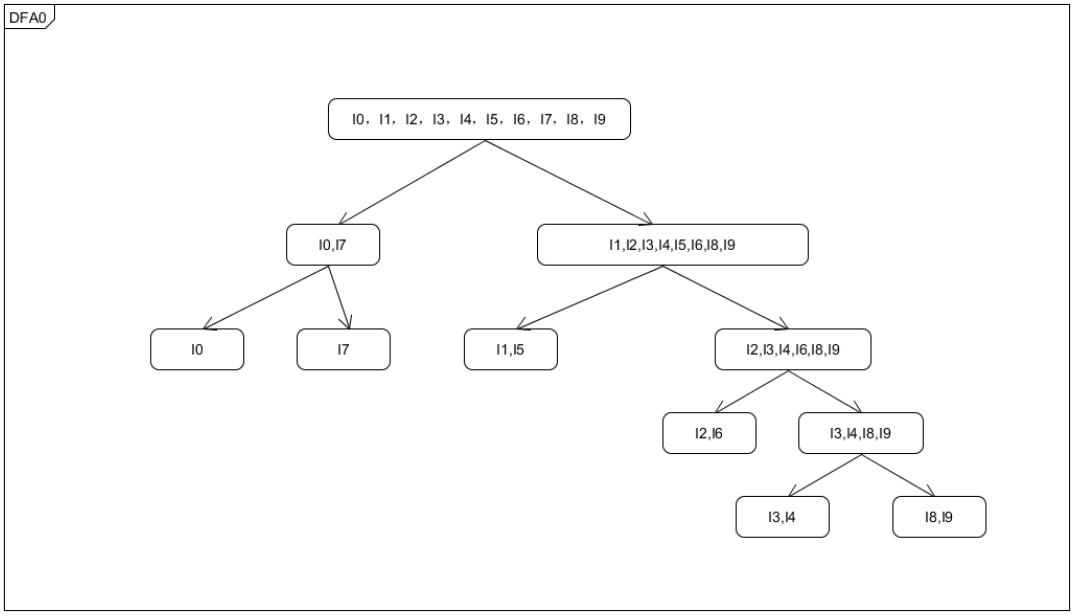


5.2 转换为 DFA

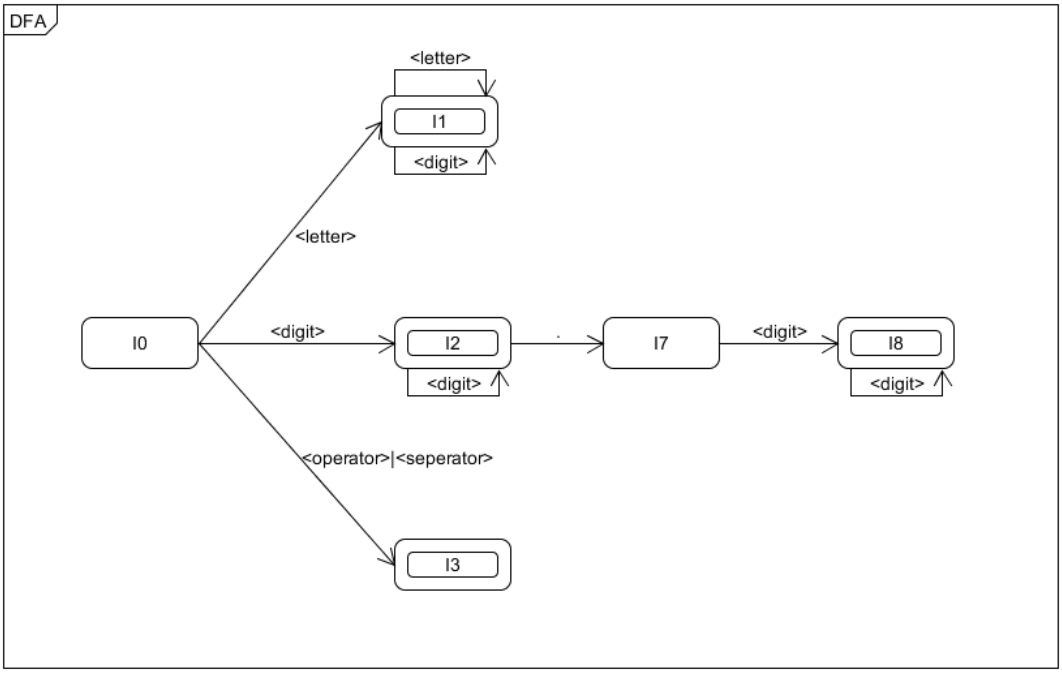
Ii	<letter>	<digit>	<operator>	<separator>	.
I0={0,1,5,12,14}	ε-c({2})=I1	ε-c({6})=I2	ε-c({13})=I3	ε-c({15})=I4	Ø
I1={2,3,4}	ε-c({3})=I5	ε-c({3})=I5	Ø	Ø	Ø
I2={6,7,8}	Ø	ε-c({7})=I6	Ø	Ø	ε-c({9})=I7
I3={13}	Ø	Ø	Ø	Ø	Ø
I4={15}	Ø	Ø	Ø	Ø	Ø
I5={3,4}	ε-c({3})=I5	ε-c({3})=I5	Ø	Ø	Ø

I6={7,8}	∅	ε-c({7})=I6	∅	∅	ε-c({9})=I7
I7={9}	∅	ε-c({10})=I8	∅	∅	∅
I8={10,11,12}	∅	ε-c({11})=I9	∅	∅	∅
I9={11,12}	∅	ε-c({11})=I9	∅	∅	∅

5.3 优化 DFA



5.4 DFA 模型



6. 重要数据结构

用 Catalog 枚举表示大类。

```
public enum Catalog {
    KEYWORD,ID,INT,DOUBLE,OPERATOR,SEPARATOR
}
```

用 Token 类描述生成的 Token，并标注所属大类的具体类别。

```
public class Token {
    private Catalog catalog;
    private String lexeme;
    private int index;

    public Token(Catalog catalog, String lexeme, int index) {
        this.catalog = catalog;
        this.lexeme = lexeme;
        this.index = index;
    }
    public Catalog getCatalog() {
        return catalog;
    }
```

```
    public void setCatalog(Catalog catalog) {
        this.catalog = catalog;
    }
    public String getLexeme() {
        return lexeme;
    }
    public void setLexeme(String lexeme) {
        this.lexeme = lexeme;
    }
}
```

State 枚举类对应 DFA 中的状态。

```
public enum State {
    STATE_0,STATE_1,STATE_2,STATE_3,STATE_4,STATE_5
}
```

7. 核心算法描述

Analyzer 类是本程序的核心算法，其中的 lexicalAnalyze()方法是核心方法。程序每读入一个字符时会判断状态机当前所处的状态，根据不同的状态做状态转换、异常识别或输出成立的 Token。

```
public void lexicalAnalyze() {
    String line = "";
    while ((line = ioHelper.nextLine()) != null) {
        int index = 0;
        String tempWord = "";
        State state = State.STATE_0;
        while (index < line.length()) {
            char current = line.charAt(index);
            switch (state){
                case STATE_0:
                    if (Constant.isDigit(current)) {
                        state = State.STATE_2;
                        tempWord += current;
                    } else if (Constant.isLetter(current)) {
                        state = State.STATE_1;
                        tempWord += current;
                    } else if (Constant.isOperator(current+"")>=0) {
                        state = State.STATE_3;
                        tempWord += current;
                    } else if (Constant.isSeparator(current)>=0) {
                        state = State.STATE_3;
                        tempWord += current;
```

```
                    }
                    break;
                case STATE_1:
                    if (Constant.isDigit(current)) {
                        tempWord += current;
                    } else if (Constant.isLetter(current)) {
                        tempWord += current;
                    } else {
                        index--;
                        state = State.STATE_0;
                        if (Constant.isKeyword(tempWord)>=0) {
                            addToken(tempWord, Catalog.KEYWORD,
Constant.isKeyword(tempWord));
                        } else {
                            addToken(tempWord, Catalog.ID, -1);
                        }
                        tempWord = "";
                    }
                    break;
                case STATE_2:
                    if (Constant.isDigit(current)) {
                        tempWord += current;
                    } else if (current=='.') {
                        tempWord += current;
                        state = State.STATE_4;
                    } else {
                        index--;
                        state = State.STATE_0;
                        addToken(tempWord, Catalog.INT, -1);
                        tempWord = "";
                    }
                    break;
                case STATE_3:
                    char su = tempWord.charAt(0);
                    if (((su=='+'||su=='-
'||su=='*'||su=='/'||su=='<'||su=='>'||su=='!'||su=='=') && current=='=')
                            || ((su=='|' && current=='|')||(su=='&' &&
current=='&')||(su=='<' && current=='<')||(su=='>' && current=='>'))) {
                        addToken(tempWord+current, Catalog.OPERATOR,
Constant.isOperator(tempWord+current));
                    }else {
                        index--;
                        if (Constant.isOperator(tempWord)>=0) {
                            addToken(tempWord, Catalog.OPERATOR,
```

```
Constant.isOperator(tempWord));
                    } else {
                        addToken(tempWord, Catalog.SEPARATOR,
Constant.isSeparator(tempWord.charAt(0)));
                    }
                }
                state = State.STATE_0;
                tempWord = "";
                break;
            case STATE_4:
                if (Constant.isDigit(current)) {
                    state = State.STATE_5;
                    tempWord += current;
                } else {
                    System.out.println("error: state 4");
                }
                break;
            case STATE_5:
                if (Constant.isDigit(current)) {
                    tempWord += current;
                } else {
                    index --;
                    state = State.STATE_0;
                    addToken(tempWord, Catalog.DOUBLE,-1);
                    tempWord="";
                }
                break;
        }
        index++;
    }
    //这里的内容在后面说明
}
```

8. 困难与解决方案

(1) 回退处理

在当前状态无法继续，并且输入的字符并没有进入异常处理时，程序会将当前指针回退一个字符，代表当前状态终止，清空缓存的字符，并置状态为初始状态。

```
index--;
state = State.STATE_0;
//根据不同的状态选择输出不同的 Token
tempWord = "";
```

（2） 末尾处理

在读取一行结束时，因为没有触发下一次的回退处理，可能会导致到一行的最后一个字符结束没有处理当前 Token 的问题，因此在每行循环的末尾都要判断一下状态，如果是终止状态则要处理当前缓存的字符。

```
switch (state) {
    case STATE_1:
        if (Constant.isKeyword(tempWord)>=0) {
            addToken(tempWord, Catalog.KEYWORD, Constant.isKeyword(tempWord));
        } else {
            addToken(tempWord, Catalog.ID, -1);
        }
        break;
    case STATE_2:
        addToken(tempWord, Catalog.INT, -1);
        break;
    case STATE_3:
        if (tempWord.length()>0) {
            if (Constant.isOperator(tempWord)>=0) {
                addToken(tempWord, Catalog.OPERATOR, Constant.isOperator(tempWord));
            } else {
                addToken(tempWord, Catalog.SEPARATOR,
Constant.isSeparator(tempWord.charAt(0)));
            }
        }
        break;
    case STATE_5:
        addToken(tempWord, Catalog.DOUBLE,-1);
        break;
}
```

9. 测试用例

输入程序（在/source_code/testFile 文件夹下）：

```
public class Main() {
    public static void main (String[] args) {
        int a = 10;
        a += 10;
        double b = 123.45;
        if (a == 20) {
            a >> 2;
        }
        System.out.println(10 * 123.45 + 12 - a * b);
    }
```

```
}
```

输出截图：

```
(KEYWORD_34,public)
(KEYWORD_8,class)
(ID,Main)
(SEPARATOR_5,()
(SEPARATOR_6,))
(SEPARATOR_1,{)
(KEYWORD_34,public)
(KEYWORD_38,static)
(KEYWORD_47,void)
(ID,main)
(SEPARATOR_5,()
(ID,String)
(SEPARATOR_3,[)
(SEPARATOR_4,])
(ID,args)
(SEPARATOR_6,))
(SEPARATOR_1,{)
(KEYWORD_26,int)
(ID,a)
(OPERATOR_0,=)
(INT,10)
(SEPARATOR_0,;)
(ID,a)
(OPERATOR_13,+=)
(INT,10)
(SEPARATOR_0,;)
(KEYWORD_13,double)
(ID,b)
(OPERATOR_0,=)
(DOUBLE,123.45)
(SEPARATOR_0,;)
(KEYWORD_22,if)
(SEPARATOR_5,()
(ID,a)
(OPERATOR_22,==)
(INT,20)
(SEPARATOR_6,))
(SEPARATOR_1,{)
(ID,a)
(OPERATOR_21,>>)
```

10. 总结

通过这次实验，我对词法分析的过程有了更深入的认识，通过自己看书和查阅资料，我锻炼了自己分析问题和解决问题的能力，为今后语法分析打下基础。