

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](#_Toc465195312)

[2. 内容概述 2](#_Toc465195313)

[3. 思路方法 2](#_Toc465195314)

[4. 语言定义 2](#_Toc465195315)

[4.1 保留字 2](#_Toc465195316)

[4.2 操作符 3](#_Toc465195317)

[4.3 分隔符 3](#_Toc465195318)

[4.4 标识符 4](#_Toc465195319)

[4.5 数值 4](#_Toc465195320)

[5. 相关FA描述 4](#_Toc465195321)

[5.1 合并后的NFA 4](#_Toc465195322)

[5.2 转换为DFA 4](#_Toc465195323)

[5.3 优化DFA 5](#_Toc465195324)

[5.4 DFA模型 6](#_Toc465195325)

[6. 重要数据结构 6](#_Toc465195326)

[7. 核心算法描述 7](#_Toc465195327)

[8. 困难与解决方案 9](#_Toc465195328)

[9. 测试用例 10](#_Toc465195329)

[10. 总结 11](#_Toc465195330)

# 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。

# 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。

# 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

# 语言定义

## 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 |

## 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 | || |  |  |

## 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** | , |

## 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

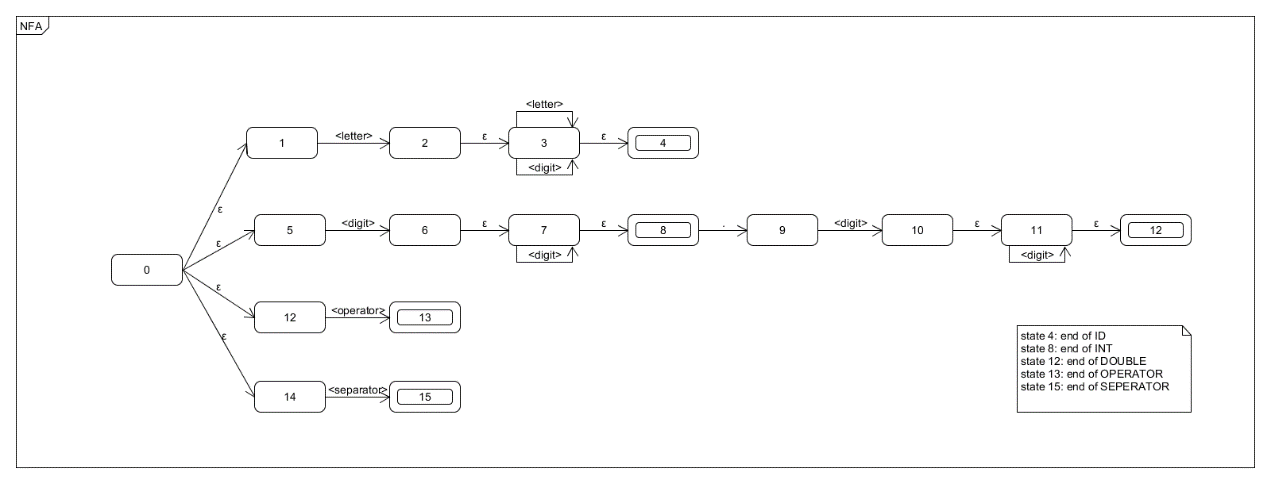
|  |  |
| --- | --- |
| 类别 | 正则表达式 |
| ID | <letter> (<letter> | <digit> ) \* |

## 数值

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

# 相关FA描述

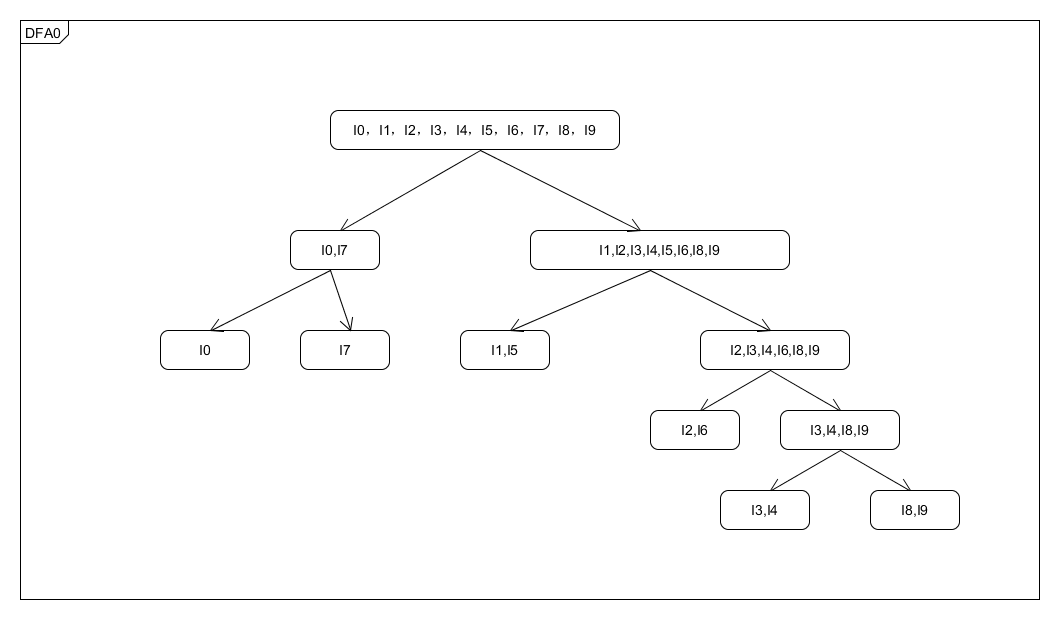
## 合并后的NFA



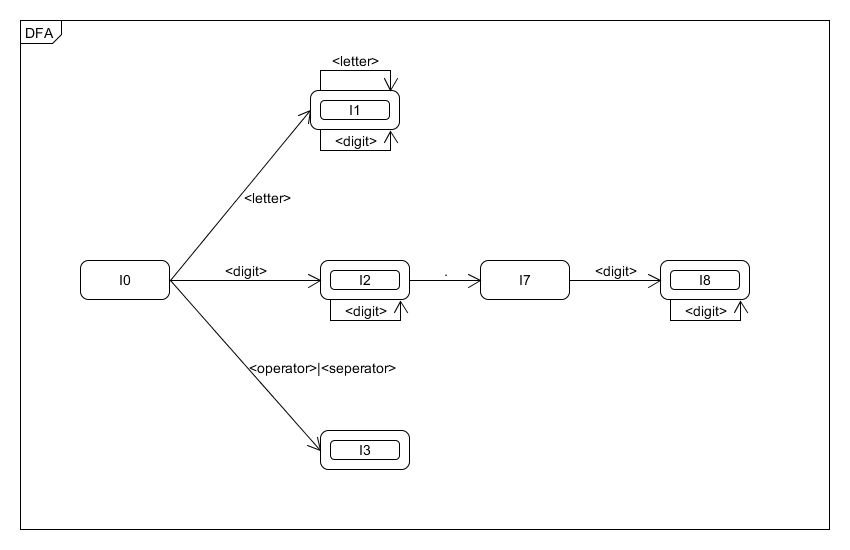
## 转换为DFA

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

## 优化DFA



## DFA模型



# 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*** } |

# 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 } |

# 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 = **""**;

1. **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**;  
}

# 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;

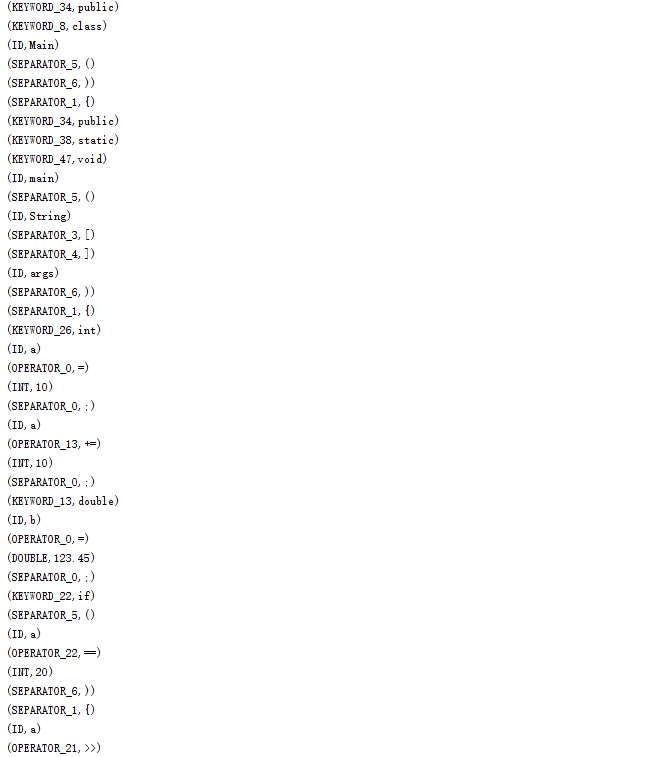
}

System.out.println(10 \* 123.45 + 12 - a \* b);

}

}

输出截图：



# 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。