Lab1实验报告

141250106

申彬

## 目标

制作一个lex.读取myLex.lex文件,解析其中定义的模式和动作,并以此对给定的代码文件进行词法分析.由于时间关系,未能完成代码生成部分,而使用最小化DFA对代码文件进行词法分析.

## 使用说明

1. 假设与依赖

正则表达式支持的运算符为: \* + [] () |

正则表达式支持的特殊字符: .

需要java8运行环境

如需编译,需要maven环境.

1. 使用

myLex.lex为lex文件,input.java为待解析文件

1. 在jar文件所在目录下打开命令行
2. java –jar LexicalAnalyzer.jar
3. 解析默认lex文件大约需要5秒,之后可以在命令行上看到对input.java的解析结果.
4. 解析结果同时会保存在analyzeResult文件中

输出结果实例:

**<keyword,public>  
<keyword,class>  
<id,Main>  
<Punctuation,{>  
<keyword,public>  
<keyword,static>  
<keyword,void>**

## 内容描述

1. **myLex.lex文件描述:**

下面是一个实例,实验中使用的lex文件内容略多一些.

%{  
 start  
}%  
  
**delim [ \t\n]  
ws {delim}+  
letter [abcdefghijklnmopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ]  
digit [0123456789]  
id {letter}({letter}|{digit})\*  
number {digit}+(\.{digit}\*)|{digit}+**  
  
%%

# this is a comment  
**{ws} : { nothing }  
if : { keyword }  
; : {Punctuation}  
\. : {Punctuation}  
  
{id} : { id }  
{number}: { number }  
= : { operator }**  
%%

End part.

myLex.lex文件格式模仿了lex文件,分为4部分:

1. 开始部分,位置是文件开头的 **%{ }%** 中
2. 定义部分,位置是开始部分之后,格式为 name regex
3. 模式-动作部分,位置是 %% %% 之间,格式是 regex : { some thing to print }
4. 结尾部分,位置是定义部分之后.

此外,以#开头的行视为注释,此功能是为了方便调试.

1. **可运行jar文件**

LexicalAnalyzer.jar

1. **待解析文件**

input.java

1. **解析结果**

analyzeResult

1. **源码**

./SourceCode

## 思路与方法

程序的流程为:

读取lex文件 --> 解析正则表达式 -> 构造NFA -> 构造DFA -> 最小化DFA ->根据DFA解析代码文件.

因此程序的架构比较直观:

1. Lex解析模块:

读取lex文件,去掉注释 -->

提取 开始,定义,模式,结尾 四部分 -->

处理模式部分的regex,根据定义部分对其进行宏替换 -->

产生一张 id,regex,action的表

1. Regex解析模块:

接受id,regex表 -->

去掉regex中的[]运算符 -->

为regex添加连接运算符 -->

将regex从字符序列变为对象序列,方便之后的运算 -->

生成 id,regex对象序列 表

1. NFA模块:

接受 id,regex对象序列 表 -->

对每个regex对象序列,利用汤普森算法构造NFA,并在接受状态中标记其对应的id -->

将各个NFA拼接为一个NFA

1. DFA模块 核心模块 :

接受NFA,计算开始节点的 ε-闭包 -->

利用一个类似于广度优先遍历的算法,不断从一个ε-闭包通过边生成新的ε-闭包,直到没有新的ε-闭包出现 -->

上面的算法中记录了一张 ε-闭包通过各个边到达其他ε-闭包的表,将这张表转化为DFA

1. 最小化DFA模块:

接受DFA,对其进行分组 -->

不断对组进行划分,直到没有新的组出现 -->

各个组选出代表,重新构造DFA

1. 解析源文件模块:

接受DFA -->

读取源文件 -->

不断以字符作为输入,改变状态机状态,并记录下经过的接受状态的模式id -->

直到状态转化出错,则输出 模式id对应的动作,匹配到的字符串 -->

重置状态机状态,从失败的字符开始重新匹配 -->

直到文件结束,将词法分析结果写入analyzeResult.

## 关键数据结构

1. **状态机FA:**

属性: 开始状态 State startState;

方法: 获得开始节点 State getStartState();

广度优先遍历 void forEach( function\*(State) );

1. **状态 State:**

属性: 是否为接受状态 boolean isAccept;

模式id的有序列表,按模式在lex中出现的顺序排序

TreeSet<Integer> patternIds;

出边的map Map<Integer,List<State>>;

方法: 状态转换 move(Integer);

增加边 addEdge(Integer,State);

1. **Regex运算符 Operator:**

属性 运算符 char,如 ’+’ , ’\*’

优先级 int; //用于regex转后缀运算

## 核心算法

1. 正则表达式添加连接运算符,6种情况

1. 连续两个字符间  
 2. 连续) 和 (间  
 3. 单目运算符后边的 (  
 4. 单目运算符之后的字符  
 5. ) 后边是字符  
 6. 字符后边是 (

2. 正则表达式 --> NFA

程序中 正则表达式通过两个栈直接转为NFA,没有经过后缀表达式的中间步骤.

大体思想是分为两个栈: 状态机栈 和 运算符栈,运算符有优先级.

//由于汤普森算法需要在原NFA的结束节点处连接新节点,因此,此处的状态机进过特殊处理,记录了一个结束状态,方便运算.

遇到单个字符,就转化为状态机,压栈.

遇到运算符,根据优先级判断是进栈还是出栈.

若出栈,则根据汤普森算法对NFA栈进行运算.

//程序中新加了一个运算符 + ,生成的状态机与 \* 比较类似,只少了最外边一条ε边

1. NFA --> DFA

子方法: 计算ε闭包: 广度优先遍历给定state(s)的ε边

创建Map<ε闭包,state>

计算开始节点的ε闭包,创建新的state,加入map中.

开始广度优先遍历ε闭包:

//平时作业中,边的类型比较少,如只有a,b.对于每一个闭包,都对每一种可能的边进行move运算,程序算法中,边类型比较多,因此只考虑每个节点的非ε边.

新建一个Map<Edge,List<State>>,遍历闭包中每个节点非ε边.然后对map中的list进行闭包运算.

对Map<Edge,List<State>>,中的每一个edge,用list<State> (一个闭包)去Map<ε闭包,state>中查state,然后把当前闭包对应的state,加一条edge边去该闭包对应的state.

遍历结束,DFA就构造完毕了.

1. DFA --> DFA°

大致与书上(龙书中文版P115)相同.在将一个组分为多个组时不太一样.

假如对组中状态都两两比较会很低效,因此处理方法为:

为组中每个状态构造一个key,假如state1通过a到达1组,通过b到达2组,那么这个key类似为 a->1;b->2.

构造一个Map<key,List<State>>,然后将state和它对应的key加进去.这样扫描完一遍state,map中的各个list就是新组.

//从观察来看,每次最小化dfa的时间都很短,因此算法还是比较高效的.

1. 匹配源文件

不断以字符作为输入,改变状态机状态,并记录下经过的接受状态的模式id ,

直到状态转化出错,则输出 模式id对应的动作,匹配到的字符串.

重置状态机状态,从失败的字符开始重新匹配.

不断读取知道文件结束.

//这样的结果是程序会匹配尽可能长的字符串

1. 结束状态的合并

这是书上没讲到的一点.在NFA --> DFA过程中,会遇到结束状态的合并,可能patternID为1和patternID为2的两个状态要合并,那么合并时取的是 在原来lex文件中靠前的id

## 测试用例

词法定义参见myLex.lex //可以修改

待解析文件参见input.java //可以修改

输出参见analyzeResult

## 问题与解决

1. 最大的问题是有诸多琐碎的细节书上和课上没有提到,需要自己处理:
   1. Lex中特殊字符的转义. 如 % { }都是lex文件特殊字符,假如直接用正则表达式去匹配,而不考虑 \{ \} \%会出现问题.
   2. Regex中拓展运算符的处理.如[]运算符,是在预处理regex字符串时,将其转化为|运算符. +运算符是在汤普森算法中新增了一个NFA合并方式.
   3. Regex连接运算符的添加条件,参见核心算法.1,需要自己慢慢尝试.
2. 最小化DFA时结束状态的合并.书上说应该把结束状态划分为一组,实际上这样会导致错误,只有对应的patternId不同的结束状态才能合成一组.
3. NFA-->DFA-->DFA°算法复杂度高.实现中大量采用了Hash技术,如闭包表,分组表都存在hash表中,这样判断存在性和查表速度都会快一点.
4. 时间不够,来不及做代码生成部分了.采用DFA来做词法分析.

## 感受

受益匪浅.软院学生大部分时间在和需求,架构搏斗,平时的代码大部分也是业务代码,基本不会涉及复杂的算法和数据结构.而这次作业相当于复习了一遍数据结构,把之前的知识拾回来一些.

另外,这次作业也是正则表达式的绝佳锻炼机会,解析lex文件时大量用到正则表达式,构造状态机时也实现了一个基本的正则表达式.