

**本 科 毕 业 论 文（设计）**

**院 系** 软件学院

**题 目** 基于JavaScript的词法分析器自动生成工具的设计与实现

**学生姓名** 葛羽航 **学 号** 081251041

**年 级** 2012级 **专 业**  软件工程

**指导教师** 葛季栋 **职 称** 讲师

**论文提交日期**

# 摘 要

\*\*项目背景(为什么要做这个项目)。

\*\*技术简介(项目开发中使用了哪些技术，有何特点)。

\*\*项目组整体完成了项目中的哪些功能。

\*\*本人在项目中承担了哪些工作。

**关键词**：\*\*项目，\*\*技术(关键名词)1，\*\*技术(关键名词)2，…..

# Abstract

.

.

.

**Keywords**：, ,

**目 录**

[摘 要 I](#_Toc323148668)

[Abstract II](#_Toc323148669)

[图目录 IV](#_Toc323148670)

[表目录 V](#_Toc323148671)

[第一章 引言 1](#_Toc323148672)

[1.1 项目背景 1](#_Toc323148673)

[1.2 国内(外)工作流建模技术研究现状 2](#_Toc323148674)

[1.X可补充二级标题 2](#_Toc323148675)

[1.X.1 可补充三级标题 2](#_Toc323148676)

[1.X.2 可补充三级标题 2](#_Toc323148677)

[1.Z 论文的主要工作和组织结构 2](#_Toc323148678)

[第二章 \*\*技术概述 4](#_Toc323148680)

[2.1 \*\*技术 4](#_Toc323148681)

[2.1.1 \*\* 4](#_Toc323148682)

[2.1.2 \*\* 4](#_Toc323148683)

[2.2 \*\*技术 5](#_Toc323148684)

[2.X \*\*本章小结 5](#_Toc323148685)

[第三章 \*\*系统需求分析与概要设计 6](#_Toc323148686)

[3.1 \*\*项目整体概述 6](#_Toc323148687)

[3.1.1 \*\*可补充三级标题 6](#_Toc323148688)

[3.1.2 \*\*可补充三级标题 6](#_Toc323148689)

[3.2 \*\*系统的需求分析 6](#_Toc323148690)

[3.2.1 \*\*可补充三级标题 6](#_Toc323148691)

[3.2.2 \*\*可补充三级标题 6](#_Toc323148692)

[3.3 \*\*系统的概要设计 6](#_Toc323148693)

[3.3.1 \*\*可补充三级标题 6](#_Toc323148694)

[3.3.2 \*\*可补充三级标题 6](#_Toc323148695)

[3.X \*\*本章小结 7](#_Toc323148696)

[第四章 \*\*项目\*\*模块的详细设计与实现 8](#_Toc323148697)

[4.1 \*\*模块概述 8](#_Toc323148698)

[4.2\*\*模块的详细设计 8](#_Toc323148699)

[4.2.1 \*\*可补充三级标题 8](#_Toc323148700)

[4.2.2 \*\*可补充三级标题 8](#_Toc323148701)

[4.3 \*\*模块的实现 8](#_Toc323148702)

[4.3.1 \*\*可补充三级标题 8](#_Toc323148703)

[4.3.2 \*\*可补充三级标题 9](#_Toc323148704)

[4.X \*\*本章小结 9](#_Toc323148705)

[第五章 总结与展望 10](#_Toc323148706)

[5.1 总结 10](#_Toc323148707)

[5.2 展望 10](#_Toc323148708)

[参考文献 11](#_Toc323148709)

[致谢 12](#_Toc323148710)

# 图目录

[图2.1 \*\*图 4](#_Toc323067900)

[图2.2 \*\*图 4](#_Toc323067901)

[图4.1 MyWorkController类代码 9](#_Toc323067902)

# 

# 表目录

[表2.1 \*\*表 5](#_Toc323067881)

# 第一章 引言

## 1.1 项目背景

词法分析作为编译原理中的理论之一，在软件应用中有着广泛的应用：编译器中的scanner，网络游戏中的脚本引擎，文本编辑器中的自动纠错等等，都会用到词法分析作为语法分析或者词素检测的前趋工作。

词法分析器自动生成工具（传统称之为lex），则是为了满足不用应用场景下的词法分析需求而设计的用于生成特定语言平台的词法分析器源代码的工具。Lex工具通过解析词法规则说明书的要求来生成词法分析程序，从而简化了编写词法分析器的难度。

当前lex程序已经在得到广泛的使用，在各个语言平台中都有对应的lex程序。常见的lex程序包括：Flex[[1]](#footnote-1)、JFlex[[2]](#footnote-2)、PLY[[3]](#footnote-3)、CSFlex[[4]](#footnote-4)、RAA[[5]](#footnote-5)等，涵盖了主流的程序语言平台。但是目前在JavaScript语言上，却没有对应的lex程序。

随着web2.0的不断发展和进步，网络应用在日常生活和工业生产中的作用逐渐凸显，基于前端JavaScript语言的应用越来越多也越来越复杂。此外，随着Html5的普及和推进，包括微软windows 8在内的系统平台逐步以JavaScript脚本为核心的App开发模式做为主流，JavaScript语言的使用场景将会越来越丰富。可以预见，在以下web应用场景中都会使用到词法分析的相关技术：

1. Html5游戏（特别是WebGL推广后的大型网页3D游戏）中的游戏脚本引擎
2. 在线文档编辑工具（如Google Doc, 微软的在线Office）中的拼写检查
3. Window 8 系统中大量会使用到词法分析器的App
4. 云计算平台的在线IDE中的代码高亮
5. 其它各种会使用到编译原理的应用场景

所以，开发一个基于JavaScript的词法分析器自动生成工具越来越有着潜在的必要性的作用。基于这些背景和考虑，本项目完成了基于JavaScript的词法分析器自动生成工具。

## 1.2 国内(外)lex技术研究现状

基于Petri网的工作流建模技术在国内外研究机构及工业界中，都得到了广泛的应用。在基于Petri网的工作流建模、分析和管理工具中，比较常见的有：YAWL、PIPE[[6]](#footnote-6)、ProM等，它们基于不同的模型，侧重于工作流建模的不同方面。虽然它们各自解决了工作流建模不同方面的问题，但它们在开源性、可扩展性以及分析算法支持方面存在一定的问题。

当前国内外基于Petri网的工作流建模相关的技术包括底层文件存储类型、仿真技术、数据分析技术等方面，它们的研究现状如下：

…………………

## 1.X可补充二级标题

### 1.X.1 可补充三级标题

### 1.X.2 可补充三级标题

## 1.3 论文的主要工作和组织结构

本文介绍了词法分析和词法分析自动生成工具的相关理论知识，以及一个基于JavaScript的lex工具——AliceLex的设计和实现。主要是对词法分析相关的编译原理进行描述，同时描述一个基于JavaScript的Lex的设计和实现过程以及这个过程中出现的一些问题，论文主要工作包括：

1. 介绍了Web2.0的发展现状，阐述了编译原理，尤其是词法分析在JavaScript上面的应用前景。

2. 介绍了编译原理的相关理论知识，词法、文法的相关概念。包括正则表达式，NFA，DFA，以及相关的算法理论。

3. 介绍了词法分析器自动生成工具和其实现的难点要点，展现当前Lex工具的现状。

5. 详细介绍了基于JavaScript的词法分析自动生成工具AliceLex的设计与实现，展示了该工具的使用典示例。

本篇论文的组织结构如下：

第一章：概述和前言部分，主要介绍了项目背景，当前词法分析和词法分析自动生成工具的研究及现状 ，并描述了该论文的主要工作。

第二章：主要介绍基于JavaScript的词法分析器自动生成工具的开发和实现过程中用到的相关理论知识和理论研究，以及基于JavaScript的Lex工具与传统Lex工具相比较的特点和实现难点。

第三章：从需求分析和总体设计两个方面，描述基于JavaScript的Lex工具的提出背景和开发者信息，同时分析和总结出功能性和非功能性需求。

第四章：对AliceLex工具的详细设计和实现进行描述，重点包括Lex工具相关算法的设计和具休实现。

第五章：AliceLex工具的使用说明和相关示例，包括展示经典Lex示例在AliceLex上的实现，以及介绍了基于AliceLex核心的在线代码编辑器。

第六章：总结该项目已实现的功能，探讨项目的缺点和不足，并指出该项目未来的扩展和发展方向

# 第二章 AliceLex工具相关技术概述

## 2.1词法分析相关理论

词法分析是编译的第一阶段，其主要任务是读入源程序的输入字符、将它们组成词素，生成并输出一个词法单元序列，每个词法单元对应于一个词素。这个词法单元序列可以被输出到词法处理单元进行下一步的分析处理（典型的就是输出到语法分析器中进行语法分析）。

### 2.1.1 词法单元、模式和词素

词法分析中的三个关键术语：

* **词法单元**由一个单元名和一个可选的属性值组成。词法单元名是一个表示某种词法单位的抽象符号，比如一个特定的关键字，或者代表一个标识符的输入字符序列。
* **模式**描述了一个词法单元的词素可能具有的形式。当词法单元是一个关键字时，它的模式就是组成这个关键字的字符序列。对于标识符和其他词法单元，模式是一个更加复杂的结构，它可以很多符号串匹配。
* **词素**是源程序中的一个字符序列，它和某个词法单元的模式匹配，并被词法分析器识别为该词法单元的一个实例。

### 2.1.2 正则表达式

正则表达式是一种用来描述词素模式的重要表示方法，它可以高效地描述在处理词法单元时要用到的模式类型。Lex程序中对于词法规则的定义部分，正是使用正则表达式及其扩展来描述。下面给出相关的定义。

****定义2.1 正则表达式****

设是字母表。上的正则表达式（regular expressions）和正则表达式表示的语言递归定义如下：

1. 是一个正则表达式，表示空集；
2. 是一个正则表达式，表示{};
3. 是一个正则表达式，表示{}，这里；
4. 如果 和 分别表示语言*R*和*S*的正则表达式，那么：

(*r|s*) 是表示的正则表达式；

(*rs*) 是表示*RS*的正则表达式；

(*r\**) 是表示*R\** 的正则表达式。

****定义2.2 正则表达式扩展****

在lex的词法规则定义块中使用的正则表达式扩展表示法在词法分析的规约中有着普遍的使用，其典型的扩展表示定义如下：

1. 一个或多个实例。单目后缀运算符表示一个正则表达式及其语言的正闭包。即如果 *r* 是一个正则表达式，那么(*r*)+ 就表示语言(*L*(*r*))+，*r*+ = *r*\**r* = *rr*\*，*r*\* = *r*+|
2. 零个或一个实例。单目后缀运算符 ？ 表示零个或一个出现，r? = r|e，即L(r) = L(r) U {e}
3. 字符类。正则表达式a1|a2|…|an（其中ai是字母表中的各个符号）可以缩写成[a1a2…an]。当a1,a2,…,an形成一个逻辑上的连续的序列时，比如连续的大写字母，连续的小写字母时，可以表示成[a1-an]。如果在[]内第一个字符是^，表示排除[]内之后字符范围内的所有字符。
4. 重复。后续花括弧{}运算符表示重复之前的正则表达式。r{m,n}表示最少m个，最多n个r的重复出现。如果省略m，表示最少0个，省略n，表示最多不限制。
5. 预定义表示。转义符 \ 用来表示预定义的正则表达式。比如\d表示数字，相当于[0-9]。更多的预定符包括 \D, \s, \S, \w, \W[[7]](#footnote-7)

### 2.1.2 有穷自动机

词法分析器的工作核心是有穷自动机的状态转移变化，而lex程序的工作原理则是将词法规则分析转换成自动机并对自动机进行处理和优化，包括正则表达式转到不确定有穷自动机（NFA）、不确定有穷自动机（NFA）转到确定有穷自动机（DFA）、确定有穷自动机的压缩（DFA compression）等等。最后将自动机转换成数组表的表示形式。下面给出相关的定义：

****定义 2.3 不确定有穷自动机（NFA）****

一个不确定的有穷自动机（NFA）由以下几个部分组成：

1. 一个有穷的状态集合S。
2. 一个输入符号集合E，即输入字母表（input alphabet）。
3. 一个转换函数（transition function），它为每个状态和EU{e}中的每个符号给出了相应的后继状态（next state）的集合。
4. S中的一个状态s0被指定为开始状态或叫初始状态。
5. S中的一个子集F被指定为接受状态（或者叫终止状态）集合。

****定义 2.4 确定有穷自动机（DFA）****

设是一个不含-规则的有穷自动机。如果对每个和，有，则是一个确定有穷自动机（deterministic finite automaton）。

即确定有穷自动机（DFA）是不确定有穷自动机（NFA）的一个特例，没有输入之上的转换动作，对于每个状态s和每个输入符号a，有且只有一条标号为a的边离开。

## 2.2 lex技术

## 2.3 本章小结

本章介绍了词法分析和词法分析器

# 第三章 AliceLex工具需求分析与概要设计

## 3.1 项目整体概述

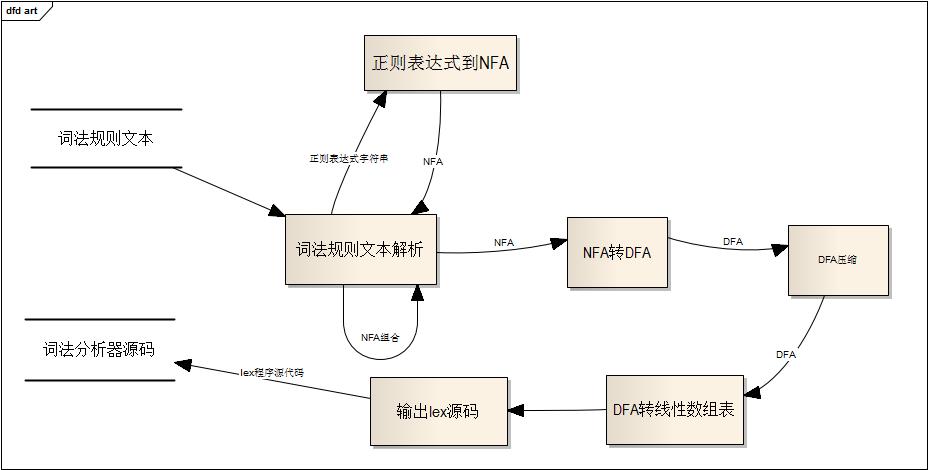
## 3.2 项目需求分析

## 3.3 项目概要设计

### 3.3.1 体系结构设计

AliceLex是JavaScript语言上的lex工具，跟传统lex工具一样，其核心流程包括：读取词法规则(lex)文本—>解析词法规则文本，构造规则文本对应的NFA—>将NFA转换成DFA—>DFA压缩（最小化）—>DFA转换成线性表—>将线性表生成词法分析器程序源码。

综合以上分析，AliceLex项目的体系结构使用管道过滤器风格，其体系结构图如下：



### 3.3.2 模块划分及功能

AliceLex项目的模块及其功能的描述如下：

## 3.4 本章小结

# 第四章 AliceLex工具的详细设计与实现

## 4.1 模块概述

## 4.2模块的详细设计

### 4.2.1 公共核心模块—Core

类图

### 4.2.2 NFA模块—Nfa

### 4.2.3 DFA模块—Dfa

### 4.2.4 线性表和字符集模块—Table

### 4.2.5 实用集模块—Utility

### 4.2.5 词法分析器源码模板模块—Template

## 4.3 算法的详细设计和实现

### 4.3.1 正则表达式构造NFA

伪码描述

算法主逻辑：使用自顶向下递归对表达式字符串reg进行语法分析，分解出组成它的子表达式，在这个过程中使用两个规则将语法分析的子表达式构造成NFA。两个规则包括基本规则和归纳规则，基本规则处理不包含运算符的子表达式，而归纳规则根据一个给定表达式的直接子表达式的NFA构造出这个子表达式的NFA。

基本规则：对于表达式ε，构造下面的NFA。

这里，i是一个新的状态，也是这个NFA的开始状态；f是另一个新的状态，也是这个NFA的接受状态。

对于字母表E中的子表达式a，构造下面的NFA。

同样，i和f是新状态，分别是这个NFA的开始和接受状态。在基本规则中，对于ε或某个a的作为reg的子表达式的每次出现，都使用新状态分别构造出一个独立的NFA。

归纳规则：

1. 选择关系的运算，包括|运算符和[]运算符，比如r=s1|s2|…|sn或r=[s1-sn]，使用如下图示进行构造：

其中i和f是新状态，分别是这个NFA的开始和接受状态。从i到sk(k=1…n)的开始状态各有一个ε转换，从sk的接受状态到f各有一个ε转换。

1. 连接关系的运算，比如r=s1s2…sn，使用如下图示进行构造：

其中i是第一个NFA即s1的开始状态，f是最后一个NFA即sn的接受状态。第k个NFA即sk的开始状态和第k+1个NFA即sk+1的接受状态合并为一个状态，合并后的状态拥有原来进入和离开合并前两个状态的全部转换。

1. 闭包关系运算，即r=s\*，使用如下图示进行构造：

其中i和f是两个新状态，分别是新NFA的开始和接受状态。

1. 对于其它扩展关系运算，包括+，？，{}等都转换成上面的3种运算构造。

**关键实现代码**

对正则表达式字符串的语法分析由Nfa模块中的Str2Nfa类完成，以下是其中的自顶向下递归中的部分关键代码：

|  |
| --- |
| //处理或运算，即 r=s|f  \_r : function() {  var nfa1, nfa2;  nfa1 = this.\_e();  while(true) {  if(this.cur\_t.tag === N.Tag['|']) {  //$.dprint('|')  this.read\_token();  nfa2 = this.\_e();  nfa1 = N.NFA.createOrNFA(nfa1, nfa2);  //$.dprint(nfa1);  } else  break;  }  //$.dprint(nfa1);  return nfa1;  },  //处理连接运算，即r = sf  \_e : function() {  var nfa1 = this.\_t();  var nf2;  while(true) {  if(this.cur\_t.tag !== N.Tag['|'] && this.cur\_t.tag !== N.Tag[')'] && this.cur\_t !== N.Token.EOF) {  nfa2 = this.\_t();  nfa1 = N.NFA.createJoinNFA(nfa1, nfa2);  //$.dprint(nfa1);  } else  break;  }  //$.dprint(nfa1);  return nfa1;  }, |

在以上代码中，只对正则字符串进行了语法分析和构造，更具体的生成NFA实例的职责交由Nfa模块中NFA类的静态函数实现，以下是其中两个函数的示例：

|  |
| --- |
| /\*\*  \* 将两个nfa进行并运算，返回一个新的nfa。  \* r=s|t  \*/  createOrNFA : function(nfa1, nfa2) {  var s = new N.NFAState();  var f = new N.NFAState(true);  s.addMove(C.Input.e, nfa1.start);  s.addMove(C.Input.e, nfa2.start);  nfa1.finish.isAccept = false;  nfa1.finish.addMove(C.Input.e, f);  nfa2.finish.isAccept = false;  nfa2.finish.addMove(C.Input.e, f);  var rtn = new N.NFA(s, f);  rtn.addState(nfa1.states);  rtn.addState(nfa2.states);  rtn.addState(s, f);  return rtn;  },  /\*\*  \* 将两个nfa进行连接运算，返回一个新的nfa。  \* r=st  \*/  createJoinNFA : function(nfa1, nfa2) {  // $.dprint('\*\*\*\*\*\*\*\*\*');  // $.dprint(nfa1);  // $.dprint(nfa2);  var rtn = new N.NFA(nfa1.start, nfa2.finish);  nfa1.finish.isAccept = false;  //合并nfa1的接受状态和nfa2的开始状态为同一个状态  if(nfa2.start.move)  nfa1.finish.addMove(nfa2.start.move[0], nfa2.start.move[1]);  for(var i = 0; i < nfa2.start.e\_moves.length; i++)  nfa1.finish.addMove(C.Input.e, nfa2.start.e\_moves[i]);  //将nfa1的状态和nfa2状态增加到新的nfa中，因为nfa1的开始态和nfa2开始态已经合并，  //不需要将nfa2的开始态添加。  rtn.addState(nfa1.states);  for(var i = 0; i < nfa2.states.length; i++) {  if(nfa2.states[i] !== nfa2.start)  rtn.addState(nfa2.states[i]);  }  // $.dprint(rtn);  // $.dprint('\*\*\*\*\*\*\*\*\*\*');  return rtn;  }, |

### 4.3.2 NFA转换为DFA

**伪码描述**

算法主逻辑

|  |
| --- |
| 定义DFAStates， DFATran  一开始，e-closure(s0)是DFAStates中唯一状态，且它未加标记；  while(在DFAStates中有一个未标记的状态T){  给T加上标记；  for(每个输入符号a){  U=e-closure(move(T, a));  if(U不在DFAStates中){  将U加入到DFAStates中，且不加标记；  }  DFATran[T, a]=U;  }  }  返回状态集DFAStates和其上的转换函数DFATran，则是构造而成的DFA |

图。。。

主逻辑中使用到以下函数：

|  |  |
| --- | --- |
| 函数 | 描述 |
| e-closure(s) | 能够从NFA的状态s开始只通过ε转换到达的NFA状态集合 |
| e-closure(T) | 能够从状态集T中某个状态s开始只通过ε转换到达的NFA状态集合，即 |
| Move(T, a) | 能够从T中某个状态s出发通过标号为a的转换到达的NFA状态的集合 |

其中e-closure(s)和move(T, a)只需要使用简单的线性遍历可以得到，e-closure(T)使用如下描述算法得到：

|  |
| --- |
| 将T的所有状态压入栈stack中；  将e-closure(T)初始化为T；  while( stack )非空 {  将栈顶元素t弹出栈；  for (每个满足如下条件的u：从t出发有一个标号为ε的转换到达状态u)  {  if(u不在e-closure(T)中)  {  将u加入到e-closure(T)中;  将u压入到栈中；  }  }  } |

**关键实现代码**

|  |
| --- |
| while( S = this.get\_untag\_state()) {  S.tag = true;    for(var i = 0; i < eqc.length; i++) {    var mv = this.get\_move(S.nfaset, eqc[i]);  if(!mv)  continue;  //$.dprint(mv.move);  /\*  \* 计算move集合的e\_closure集合  \*/  var ec = this.e\_closure(mv);    /\*\*  \* 从已经存在的e\_closure集中查找该e\_closure对应的dfa状态，  \* 如果没有打到，说明该dfa状态还示存在，需要新建  \*/  var dfa\_state = this.closure\_hash\_table[ec.hash\_key];  if(!dfa\_state) {  dfa\_state = new D.DFAState(ec.is\_accept, U.\_n.get());  dfa\_state.nfaset = ec.closure;    this.add\_dfa\_state(dfa\_state, ec.hash\_key);  }    var eqc\_index = Alice.CharTable.getEqc(eqc[i]);  S.addMove(eqc\_index, dfa\_state);  }  } |

e-closure函数

|  |
| --- |
| if(this.move\_hash\_table[mv.hash\_key])  return this.move\_hash\_table[mv.hash\_key];  var T = mv.move, is\_accept = false, stack = [], e\_c = [], e\_id = [], new\_closure = null, hash\_key = "";  for(var i = 0; i < T.length; i++) {  stack.push(T[i]);  e\_c.push(T[i]);  e\_id.push(T[i].id);  /\*if(T[i].isAccept === true){  is\_accept = true;  }\*/  is\_accept |= T[i].isAccept;  }  while(stack.length > 0) {  var t = stack.pop();  //$.dprint(t);  var u = t.e\_moves;  //$.dprint(u);  for(var i = 0; i < u.length; i++) {  if(e\_id.indexOf(u[i].id) === -1) {  e\_c.push(u[i]);  stack.push(u[i]);  e\_id.push(u[i].id);  is\_accept |= u[i].isAccept;  }  }  }  /\*\*  \* 计算哈希值，同样是对集合中每个状态的id进行排序后连接成字符串  \*/  hash\_key = e\_id.sort(function(a, b) {  return a > b;  }).join(",");  is\_accept = is\_accept ? true : false;  new\_closure = {  hash\_key : hash\_key,  closure : e\_c,  is\_accept : is\_accept  };  return new\_closure;  }, |

### DFA状态最小化

伪码描述

DFA状态最小化算法的目的在于将任意DFA转化为等价的状态最少的DFA，其作用是使得最终生成的线性表尽可能小，从而提升生成的词法分析器程序的空间效率。对于输入的DFA（接下来用D表示），使用如下方法实现对其最小化：

1. 首先构造初始划分∏，构造原则是所有非接受状态为一组，其它所有接受状态各自为一组。
2. 应用如下过程来构造一个新的划分∏new。

|  |
| --- |
| 最初，令∏new=∏；  for(∏中的每个组G){  将G分划为更小的组，使得两个状态s和t在同一组中当且仅当对于所有的输入符号a，状态s和t在a上的转换都到达∏中的同一组；  在∏new中将G替换为对G进行分划得到的那些小组；  } |

1. 如果∏new=∏，令∏final=∏并接着执行步骤4；否则用∏new替换∏并重复步骤2。
2. 在分划∏final的每个组中选取一个状态作为该组的代表。这些代表构成了状态最小DFA的状态。该DFA（接下来用D’表示）的其它部分按如下步骤构建：
   1. D’的开始状态是包含了D的开始状态的组的代表；
   2. D’的接受状态是那些包含了D的接受状态的组的代表。
   3. 令s是∏final中某个组G的代表，并令D中在输入a上离开s的转换到达状态t。令r为t所在组H的代表，那么在D’中存在一个从s到r在输入a上的转换。

### DFA转换为线性数组表

## 4.X \*\*本章小结

# 第五章 AliceLex的使用说明和示例

本章详细介绍了基于JavaScript的词法分析器自动生成工具AliceLex的使用说明，包括AliceLex项目的构建，引入和使用方法。同时会给出lex程序的经典示例在JavaScript上的实现，最后介绍基于AliceLex为核心的在线代码编辑器。

## 5.1 使用说明

### 5.1.1 项目配置

AliceLex项目作为开源项目，托管在GitHub[[8]](#footnote-8)上，使用如下方法配置AliceLex的最新代码：

1. 安装和配置nodejs[[9]](#footnote-9)
2. 安装和配置github
3. 新建目录alicelex，在此目录上执行命令：

git clone <https://github.com/YuhangGe/alicelex>

node make.js

1. 在alicelex目录的根下会生成两个文件，alicelex.js和alicelex-node.js，其中alicelex.js是浏览器使用的前端代码，alicelex-node.js是后端用于nodejs的代码。

### 5.1.2 使用方法

对于alicelex.js，在浏览器端使用如下方法使用：

1. 通过<script>标签引入alicelex.js
2. 这时候全局会有唯一的公开对象Alice，通过调用其唯一的公开接口doLex函数将词法规则文本转换成词法分析器的源代码。具体如下：

|  |
| --- |
| var js\_src = Alice.doLex(lex\_text) |

其中lex\_text是需要转换的源始词法规则文本，更详细的规则说明参见5.1.3。返回值是string类型的字符串，是该规则对应生成的词法分析器的JavaScript源代码。

1. 使用eval函数执行js\_src，这样就会产生一个词法分析器可以直接使用。具体来说，如果lex\_text文本中通过参数$name定义的词法分析器的名字是 JSLexer，则可以使用如下代码执行分析过程：

|  |
| --- |
| var text = “这里是你要进行词法分析的文本”;  var lexer = new window.JSLexer();  lexer.lex(text); |

对于alicelex-node.js，相当于是本地的一个命令行软件，使用方法为：

1. 安装和配置nodejs
2. 使用命令node alicelex-node.js lex\_file output\_file

其中lex\_file是词法规则文件名，output\_file是需要输出的词法分析器源代码文件。然后你可以在html文件中引入该output\_file，使用生成的词法分析器。

### 5.1.3 词法规则

**基本框架**

AliceLex使用传统lex程序相似的基本框架来定义词法规则：

|  |
| --- |
| {参数声明}  {规则声明}  $$  {规则动作定义}  $$  {全局函数} |

在当前版本中有两点与传统lex软件不一样，一是使用美元符代替了百分号，二是规则文本中暂时还没有支持注释。

**参数声明**

AliceLex通过使用 ${参数名} {参数值} … 这样的格式来定义参数。目前已经实现的参数如下：

* $caseignore true/false

生成的词法分析器是否忽略大小写，默认为false

* $lexname name

生成的词法分析器的名称，默认是 Daisy。该名称是一个对像实例的名称，通过window.name可以得到该实例。

* $template normal/editor/…

生成的词法分析器的源代码模板，默认为normal，生成可以在浏览器出使用的源代码。指定editor参数则会为在线代码编辑器Daisy Editor[[10]](#footnote-10)的代码高亮部分生成核心代码。

* $unicode true/false

是否是unicode模式。Unicode模式可以处理包括中文在内的所有字符，但会占用更大的内存空间。默认是false，会把所有非ascii码字符交给错误处理函数处理。

**规则声明**

规则声明部分规则如下：

rule\_name expression

其中rule\_name是定义的规则名称，expression是该规则对应的正则表达式。正则表达式的规则跟JavaScript语言中的正则表达式[[11]](#footnote-11)含义一致，支持+\*[]?^等操作符。但有一些不同说明如下：

* 预定义的表达式：

\d 数字（0-9）

\D 非数字

\w 字符（a-z，A-Z和字符\_）

\W 非字符

\s 空白符（\n\t\v）

\S 非空白符

\a 字母（a-z，A-Z）

\A 非字母

\u 大写字母（A-Z）

\U 非大写字母

\l 小写字母（a-z）

\L 非小写字母

* 转义符：

除了上面指定的预定义表达式，其它“\”符号都理解成跟javascript语言中一致的转义 符，比如\n，\t等等。同时，" \ [ ] ^ - ? . \* + | ( ) / { } % < > 这些字符在正则表达式中 专门含义，如果需要作为字符串使用必须使用转义符。

* 字符串：

使用引号包含的部分，全部当作普通字符串处理。比如 \d“\d”这个正则规则代表一个数字后面紧跟着\d这个字符串。注意在引号包含的字符串不转义。

* 使用已定义的规则：

如果需要使用已经定义的规则，则在正则表达式中使用大括号｛｝将其包含。如果不使用｛｝包含，则会理解成字符串而不是已经定义的规则。比如如下两个规则声明：

DIGIT \d

NUMBER {DIGIT}+

其中的NUMBER会被理解成 \d+，即一个以上的数字。但如果写成如下：

DIGIT \d

NUMBER DIGIT+

则其中的NUMBER会被理解成“DIGI”T+，即一个字符串DIGI后面跟一个以上的字符T

* 不包含的规则

在AliceLex的当前版本中，正则表达式不支持^和$符作为起始和结束的规则，不支持/字符回退规则。

**规则动作定义**

规则声明如下：

{rule\_name} {rule\_action}

其中rule\_name必须是在词法声明块定义过的规则名，rule\_action是对应的动作，即相应匹配的处理代码。Rule\_action的代码，如果是单行代码可以不用｛｝包含，如果是多行代码，必须使用｛｝包含。

在规则动作定义部分，动作处理代码中可以直接使用的变量包括：yylen和yytxt。这两个变量的名称借用了经典lex程序的变量名，其含义也跟经典lex程序一致，分别代表当前匹配的字符串长度和匹配的字符串。

**全局函数**

全局函数部分定义词法分析器的全局函数，包括词法分析开始，词法分析错误处理，词法分析结束。格式如下：

${func\_name} {function}

其中func\_name是函数名，目前支持以下函数：

* $construct

词法分析器的构造函数执行完成后调用此处代码。用来执行额外的构造函数任务，比如添加额外的成员变量。

* $start

词法分析器会在用户调用lex函数执行词法分析之前调用此处代码。

* $finish

词法分析器会在lex函数执行完，即词法分析结束之后调用。

* $error

当词法分析器分析到错误时会调用

{function}是对应的代码，和规则动作定义部分的action一致，如果是单行代码可以不用｛｝包含，如果是多行代码需要使用｛｝包含。

**歧义处理**

AliceLex对于歧义的处理同传统lex程序一致，即：

1. 首选最长的匹配。
2. 在匹配同样多字符的情况下，选择第一个给出的规则。

## 5.2 使用示例

### 5.2.1 lex程序经典示例[[12]](#footnote-12)

行数统计

|  |
| --- |
| $name Example1  NEW\_LINE \n  OTHER .  $$  NEW\_LINE { line\_number++; }  OTHER {}  $$  $start { line\_number = 0; }  $finish { alert(“line count is:” + line\_number);} |

带状态转移的词法分析

|  |
| --- |
| $name Example2  STR\_QUOTE \”  STR\_CHAR [^\”]\*  OTHER .  $$  STR\_QUOTE { yygoto(STR);}  <STR> STR\_CHAR { alert(“string is: “ + yytxt);}  <STR> STR\_QUOTE {yygoto(DEFAULT);}  OTHER {/\* do nothing \*/}  $$ |

### 5.2.2 Daisy Editor[[13]](#footnote-13)

Daisy Editor是一款基于HTML5的在线代码编辑工具，其使用Canvas绘制而不是像传统代码编辑器使用DOM元素构造，从而实现了海量代码文本时的流畅处理。同时，使用AliceLex作为其代码高亮部分词法分析器的生成工具，可以更灵活地支持更多语言的高亮，并且比传统在线代码编辑器具有更高的高亮渲染效率。以下是程序的示例截图：



(图… Daisy Editor编辑jQuery源代码)

Daisy Editor目前还处于原型阶段，还在进一步开发中。当前在网上还没有同类产品的出现，其使用场景主要在云计算平台中用作在线代码查看和编辑。这个项目的目的之一也是为了说明JavaScript语言平台上的词法分析器自动生成工具的使用场景和潜力。

为了说明AliceLex是如何提供高亮分析，展示如下JavaScript语言高亮的规则文本（限于篇幅，只处是部分代码）：

|  |
| --- |
| INT \d+  FLOAT \d\*\.\d+  NUM \-?({INT}|{FLOAT})  VALUE {NUM}|true|false|null|undefined  KEYWORD function|if|else|do|while|break|var|for|in|break|switch|case|return  ID [\w$][\w$\d]\*  THREE\_OPE "==="|">>>"  ONE\_OPE "+"|"-"|"\*"|"/"|"!"|"&"|"|"|"^"|"%"|"="  OPERATOR {THREE\_OPE}|{TWO\_OPE}|{ONE\_OPE}  LINE\_CMT\_B "//"  REG "/"[^\*/]([^\n/]|"\/")\*"/"g?i?m?  STR\_A\_B \"  OTHER [.\n]  $$  VALUE {this.yystyle="value";}  KEYWORD {this.yystyle="keyword";}  PARAM {this.yystyle="param";}  OBJECT {this.yystyle="object";}  ID {this.yystyle="id";}  OPERATOR {this.yystyle="operator";}  LINE\_CMT\_B {this.yystyle="comment";this.yydefault="comment";this.yygoto(LINE\_COMMENT);}  <LINE\_COMMENT> LINE\_CMT\_E{this.yystyle="comment";this.yydefault="default";this.yygoto(DEFAULT);}  <LINE\_COMMENT> LINE\_CMT\_C {this.yystyle="comment";}  REG {this.yystyle="regexp";}  STR\_A\_B {this.yystyle="string";this.yydefault="string";this.yygoto(STRING\_A);}  <STRING\_A> STR\_A\_C {this.yystyle="string";}  <STRING\_A> STR\_A\_D {this.yystyle="string";}  <STRING\_A> STR\_A\_E {this.yystyle="string";this.yydefault="default";this.yygoto(DEFAULT);}  STR\_B\_B {this.yystyle="string";this.yydefault="string";this.yygoto(STRING\_B);}  <STRING\_B> STR\_B\_C {this.yystyle="string";}  <STRING\_B> STR\_B\_D {this.yystyle="string";}  <STRING\_B> STR\_B\_E {this.yystyle="string";this.yydefault="default";this.yygoto(DEFAULT);}  OTHER {this.yystyle="default";}  $$ |

# 第五章 总结与展望

## 5.1 总结

## 5.2 展望

# 参考文献

[1] 作者,译者,书名(版本),出版地:出版社,出版时间,引用部分起止页。

[2] 作者,译者,文章题目,期刊名,年份,卷号(期数):引用部分起止页。

[3] 作者,学位论文名,本科/硕士/博士论文,大学/机构名,年份。

[4] 网页的主题,URL。

# 致谢

时光如白驹过隙，转眼间本科的时光已尽。在过去的四年里，感谢南京大学和南京大学软件学院的培养。四年的本科生涯，让我对软件工程有了系统的认识也具有了足够的实践经验。感谢软件学院给我提供的良好的软硬件条件，感谢软件学院勤劳善良的老师们的谆谆教导。

特别要感谢我的论文导师，南京大学软件学院的葛季栋老师，在论文开题和论文撰写的过程中对我的关心和指导。由于他的耐心提醒，让我很好的控制了项目实践和论文撰写的进度，并在理论研究上给了我很大帮助。感谢他在我修改论文的过程中，一次又一次细心认真的审阅。同时也感谢他在生活和学习中对我的帮助和指导。

在此要感谢郑涛老师在《软件构造》课程中讲述的编译原理对我起到的启蒙作用，使得我对编译原理产生了较浓兴趣，也才能自己去学习和体悟编译原理中的各种算法，并将其在毕业设计中一一实现。

感谢209宿舍的三个哥们，以及后来的211的众多舍友，是你们的关心支持和帮助，让我在大学四年里生活愉快而幸福，才能更好地投入到学习之中，才能专注地去研究和学习自己感兴趣的内容。

另外，衷心感谢一直以来默默支持关心我的家人和朋友们。

1. Flex是C/C++平台的lex工具，参见：http://flex.org [↑](#footnote-ref-1)
2. JFlex是Java平台的lex工具，参见：http://jflex.de [↑](#footnote-ref-2)
3. PLY是Python平台的lex工具，参见：http://www.dabeaz.com/ply/ [↑](#footnote-ref-3)
4. CSFlex是c#(.net)平台的lex工具，参见：http://ostatic.com/csflex [↑](#footnote-ref-4)
5. RAA是Ruby平台的lex工具，参见：http://raa.ruby-lang.org/project/ruby-lex/ [↑](#footnote-ref-5)
6. 关于PIPE的详细信息和介绍，参见：http://pipe2.sourceforge.net/ [↑](#footnote-ref-6)
7. 更多的预定义符号及其含义，参见5.1.3节 [↑](#footnote-ref-7)
8. Github是目前使用广泛的开源托管平台，详情和配置方法参见：http://github.com [↑](#footnote-ref-8)
9. nodejs是本地的（而非浏览器）JavaScript脚本解析引擎，详情和配置方法参见：http://nodejs.org [↑](#footnote-ref-9)
10. 关于Daisy Editor的详情参见5.2.2节 [↑](#footnote-ref-10)
11. 关于JavaScript正则表达式的规则描述，参见：http:// [↑](#footnote-ref-11)
12. 在线演示的demo参见：http://lex.xiaoge.me [↑](#footnote-ref-12)
13. 在线的演示参见：http://editor.xiaoge.me [↑](#footnote-ref-13)