**华东交通大学理工学院**

**实 验 报 告 书**

所属课程： 编译原理

题 目： DFA模拟程序

分 院： 电气与信息工程分院

专业班级： 计科2班

学 号： 20180210440111

学生姓名： 尹绪柏

指导教师： 熊婷

2021 年 4 月 13 日

目 录

[第一章：概要设计 1](#_Toc58425608)

[1.1 实验原理 1](#_Toc58425609)

[第二章：详细设计 2](#_Toc58425610)

[2.1：实验步骤 2](#_Toc58425611)

[2.2：核心功能实现 5](#_Toc58425612)

[第三章：结果分析 6](#_Toc58425613)

[3.1 不符合的结果实例 6](#_Toc58425614)

[3.2 符合结果的实例 6](#_Toc58425615)

[实验心得 7](#_Toc58425616)

第一章 概要设计

1.1 实验原理

有限自动机或称为有穷状态的机器，它由一个有限的内部状态集和一组控制规则组成，这些规则是用来控制在当前状态下读入输入符号后应转向什么状态。

一个确定的有限自动机（DFA）M是一个五元组，可由M=（S, Y, f, S0, Z）表示。

1 S是一个非空有限集，其每一个元素称为一个状态。

2 Y是有穷输入字母表，其每一个符号称为输入符号。

3 f是一个从SXY至S的单值函数。

4 S0属于S是惟一的一个初态。

5 Z属于S是一个终态集。

通过有限状态机的数学模型可以编写代码实现简易的词法分析程序。

第二章 详细设计

2.1：实验步骤

核心算法伪代码

# 状态节点匹配算法

For(状态列表长度)

      令s=栈的第一个元素；

      IF s等于初态集

栈push s的当前状态

IF s 等于终态集

栈 push 当前状态的下一个状态

将栈中的元素倒置输出转换成列表，形成路径

END

递归本方法;

END

END

IF 栈.Length > 0

栈pop

END

}

C#代码实现

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace C\_Sharp\_Project01.BianYiJiShu.Week04

{

/// <summary>

/// <para>FSM(Finite State Machine) 有限状态机</para>

/// <para>一般分为DFA(Deterministic Finite Automaton)有限自动机, NFA(Nondeterministic Finite Automaton)非确定有限自动机</para>

/// </summary>

public class FSM

{

public static string EmptySymbol = "ε";

private List<State> stateList = new List<State>(); // 所有状态集合

private Stack<string> statePathStack = new Stack<string>(); // 栈

private List<string> allStatesPathList = new List<string>(); // 所有状态构成的路径 {SUWYZ, SYZ, SXWYZ, SXWZ}

private List<string> allSymbolsPathList = new List<string>(); // 所有状态的符号构成的路径 {aaba, aaa, ba, bbba, bba}

private StringBuilder stringBuilder = new StringBuilder();

/// <summary>

/// <para>词法解析器，接受正规文法构成的数组进行解析，返回有限状态(节点)集合；默认数组第一个为初态，最后一个为终态，</para>

/// <para>使用FSM.EmptySymbol代表ε空符号</para>

/// </summary>

/// <param name="regularGrammer">正规文法构成的字符串数组</param>

public void LexicalParser(string[] regularGrammer, out List<State> res)

{

List<State> list = new List<State>();

for (int i = 0; i < regularGrammer.Length; i++)

{

string grammer = regularGrammer[i];// 获取一行正规文法

// 判断是否包含->符号

if (grammer.IndexOf("->") > -1)

{

// 分割S->aU，得到S aU

string[] items = grammer.Split("->");

// 判断aU长度

if (items[1].Length == 2)

{

list.Add(new State(items[0], items[1].Substring(0, 1), items[1].Substring(1, 1)));

}

else if (items[1].Length == 1) // 当判断当前为终态集时，即识别为形如S->a的形式

{

list.Add(new State(items[0], EmptySymbol, EmptySymbol));

}

}

}

stateList = list;

res = list;

// 对allStatesPathList属性进行赋值

FindAllStatesPath(list, list[0].state, list[list.Count - 1].state);

// 对allSymbolsPathList属性进行赋值

FindAllSymbolsPath();

}

/// <summary>

/// 词法匹配检查，用于判断字符串str是否符合构建的正规语法

/// </summary>

/// <param name="stateList">有限状态集合</param>

/// <param name="str">目标字符串</param>

/// <returns>符合正规语法返回true，否则返回false</returns>

/// List<State> stateList, string target

public bool LexicalMatching(string target)

{

//State currentState = stateList[0]; // 当前状态

//string currentSymbol = ""; // 当前接收符号

//for (int i = 0; i < target.Length; i++) // 开始逐个匹配每一个字符

//{

// // 获取当前接收符号

// currentSymbol = target.Substring(i, 1);

// // 根据当前的接收符号，查询出有多少个后继状态

// int subStateNum = SubState(stateList, currentState, currentSymbol);

// while (subStateNum > 0)

// {

// // 查询后继状态

// }

//}

//// 当遍历完目标串后，判断当前的状态是否是终态

////if (tmpState != null && tmpState.nextState.Equals(FSM\_Original.EmptySymbol))

//// return true;

///

////return false;

if (allSymbolsPathList.Contains(target))

return true;

return false;

}

/// <summary>

/// 根据接收符号，查询出当前状态有多少个后继状态

/// </summary>

/// <param name="stateList"></param>

/// <param name="currentState">当前状态</param>

/// <param name="symbol">接收符号</param>

/// <returns></returns>

private int SubState(List<State> stateList, State currentState, string symbol)

{

int num = 0;

foreach (State s in stateList)

{

if (s.state.Equals(symbol))

num++;

}

return num;

}

public void FindAllStatesPath(List<State> stateList, string startState, string endState)

{

if (statePathStack.Contains(startState))

{

Console.Write("环:[");

foreach (string symbol in statePathStack)

Console.Write(symbol);

Console.Write("]\n");

statePathStack.Pop();

return;

}

for (int i = 0; i < stateList.Count; i++)

{

State state = stateList[i];

if (state.state.Equals(startState))

{

statePathStack.Push(state.state);

if (state.nextState.Equals(endState))

{

statePathStack.Push(state.nextState);

// 获取路径

foreach (string s in statePathStack)

stringBuilder.Append(s);

char[] chArr = stringBuilder.ToString().ToCharArray();

stringBuilder.Clear();

for (int n = chArr.Length - 1; n >= 0; n--)

{

stringBuilder.Append(chArr[n]);

}

allStatesPathList.Add(stringBuilder.ToString());

stringBuilder.Clear();

statePathStack.Pop(); // 添加了终点路径,所以要返回两次

statePathStack.Pop();

continue; // 已经找到路径,返回上层找其他路径

}

FindAllStatesPath(stateList, state.nextState, endState);

}

}

// 如果找不到下个节点,返回上层

if (statePathStack.Count > 0)

{

statePathStack.Pop();

}

}

private void FindAllSymbolsPath()

{

stringBuilder.Clear();

for (int i = 0; i < allStatesPathList.Count; i++)

{

for (int j = 0; j < allStatesPathList[i].Length - 1; j++)

{

foreach (State state in stateList)

{

// Console.WriteLine(state.state + " " + state.symbol + " " + state.nextState);

// Console.WriteLine(state.state.Equals(allStatesPathList[i].Substring(j, 1)) + " " + state.nextState.Equals(allStatesPathList[i].Substring(j + 1, 1)));

if (state.state.Equals(allStatesPathList[i].Substring(j, 1)) && state.nextState.Equals(allStatesPathList[i].Substring(j + 1, 1)))

{

// Console.Write(state.symbol);

stringBuilder.Append(state.symbol);

}

}

}

allSymbolsPathList.Add(stringBuilder.ToString());

stringBuilder.Clear();

//Console.WriteLine();

}

}

public void Print(List<State> stateList)

{

foreach (State s in stateList)

Console.WriteLine(s);

}

public void PrintAllStatesPath()

{

foreach (string state in allStatesPathList)

Console.WriteLine(state);

}

public void PrintAllSymbolsPath()

{

foreach (string str in allSymbolsPathList)

Console.WriteLine(str);

}

}

}

namespace C\_Sharp\_Project01.BianYiJiShu.Week04

{

// 表示状态转换图中的一个状态

public class State

{

/// <summary>

/// 表示当前的状态

/// </summary>

public string state;

/// <summary>

/// 当前状态的识别字符(符号)

/// </summary>

public string symbol;

/// <summary>

/// 接受识别字符后，转换的下一个状态

/// </summary>

public string nextState;

public State()

{

}

public State(string state, string symbol, string nextState)

{

this.state = state;

this.symbol = symbol;

this.nextState = nextState;

}

public override string ToString()

{

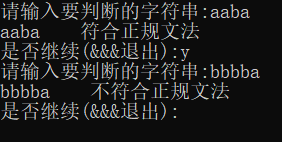
return "State[ " + state + " " + symbol + " " + nextState +" ]";

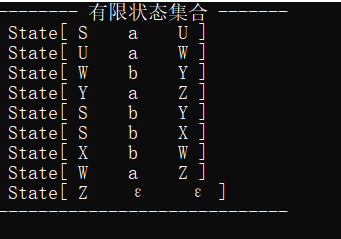
}

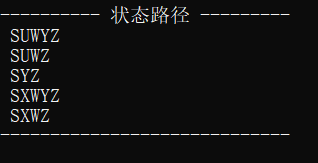
}

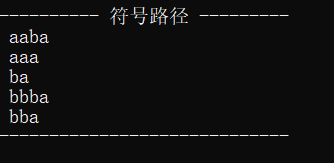
}

结果测试









2.2：核心功能实现

public void FindAllStatesPath(List<State> stateList, string startState, string endState)

{

if (statePathStack.Contains(startState))

{

Console.Write("环:[");

foreach (string symbol in statePathStack)

Console.Write(symbol);

Console.Write("]\n");

statePathStack.Pop();

return;

}

for (int i = 0; i < stateList.Count; i++)

{

State state = stateList[i];

if (state.state.Equals(startState))

{

statePathStack.Push(state.state);

if (state.nextState.Equals(endState))

{

statePathStack.Push(state.nextState);

// 获取路径

foreach (string s in statePathStack)

stringBuilder.Append(s);

char[] chArr = stringBuilder.ToString().ToCharArray();

stringBuilder.Clear();

for (int n = chArr.Length - 1; n >= 0; n--)

{

stringBuilder.Append(chArr[n]);

}

allStatesPathList.Add(stringBuilder.ToString());

stringBuilder.Clear();

statePathStack.Pop(); // 添加了终点路径,所以要返回两次

statePathStack.Pop();

continue; // 已经找到路径,返回上层找其他路径

}

FindAllStatesPath(stateList, state.nextState, endState);

}

}

// 如果找不到下个节点,返回上层

if (statePathStack.Count > 0)

{

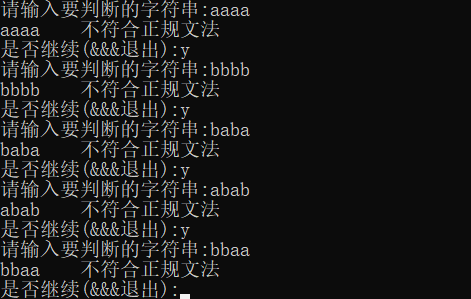
statePathStack.Pop();

}

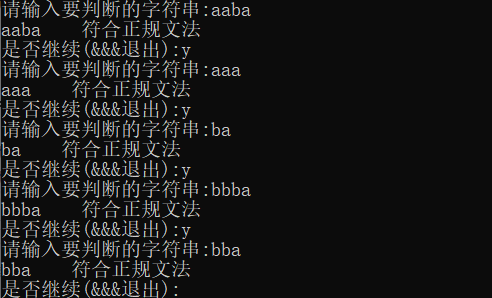
}

第三章 结果分析

3.1 不符合的结果实例



3.2 符合结果的实例



实验心得

通过本次上机设计一个简单的有限自动机算法，来实现简单的词法匹配，使我对有限状态机概念、理论及算法等知识有了一定的了解。同时也学习了有向图的图论知识。接触到了图的遍历算法如：广度优先、深度有限，抑或是对于状态机更加深入的子集构造算法。我也明白了，即使是在稿纸上通过一些数学知识能够容易的绘制节点图，但是在计算机中不一样，我们是通过代码来实现，不是所见即所得，也不是所想即所得。需要实现这些复杂的功能必须了解算法、数据结构等等。

在这次设计中，本以为能够简单的编码实现，但是实际动手时，才发现自己对于算法的薄弱，数学知识的匮乏。但依旧收获十足，感谢老师的指导。