**数据结构与算法课程**

**综合训练**

**成员\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**提交日期\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**联系电话\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**实验一**

**实验题目:SE 程序语言**

**实验描述:**

无论用什么编程语言编写程序，最终都要翻译成计算机系统能够识别的表示形式。程序

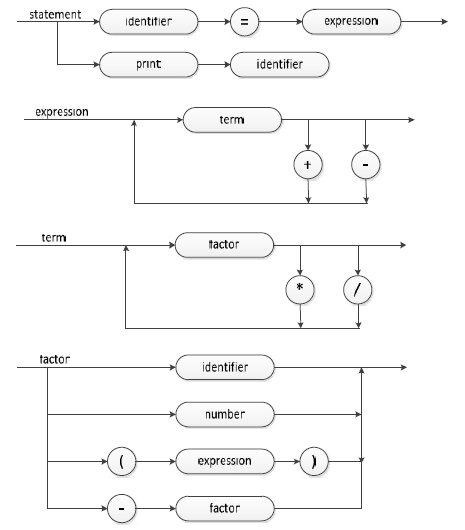
的执行过程有两种：编译执行和解释执行。我们即将要设计的程序语言，简称SE Language，

将采用解释执行的方式运行，也就是读入一句执行一句。当然为了能够让SE 运行起来，必

须要实现一个程序解释器。本实验的目标就是针对SE 实现一个可运行的解释器。

SE Language 的定义：

* 只有两种语句：一种是赋值语句，一种是打印语句；
* 标示符只用来表示变量，标示符的组成是由英文字母字符、‘\_’字符和数字字符构
* 成，其中只有下划线和英文字母字符作为标示符的起始字符，标示符的长度最多10 个字符，
* 标示符不区分大小写；
* 表达式是语句构成的主要成分，其主要由变量、‘+’、‘—’、‘\*’、‘/’、‘^’、‘（’、
* ‘）’构成，其中这些运算符的优先级及结合型与普通程序设计语言一样；
* 每条语句由分号表示结束；
* 变量在使用前必须要有值；
* 具体的语句定义如图1 所示：



**实验要求：**

1、程序解释器要能甄别出语法错误，并进行报告

**问题分析：**

这个问题需要我们设计一个解释器，类似WSH的解释器，因此不必要生成而二进制代码。根据编译原理，我们可以将即使程序分为两部分：词法分析、解释执行。

词法分析部分主要识别出一个个单词，并标识单词的种别，在这里根据题目可以分为标识符、整形常量、界符，对于关键字，由于这里只有一个print，且位置简单，一般位于一个语句的第一个，所以这里没有设置关键字这个种别。这一步主要生成一个单词表，并且顺序一定。

而在解释执行时，就是借助上一部产生的单词表，在这里看作是表达式中缀形式的表示，而在解释执行中主要就是将中缀表达式转化为后缀表达式，并且完成运算。

在这其中，对于每一条语句均执行一次上面的步骤，始终维持一个保持运行时表，这个表主要用于记录变量的值以及更新、删除。

流程如下：

脚本代码源文件

单词记录表

变量运行时表

1》读取并分割出一条语句

2》使用词法分析器分析出单词，并记录

解释执行，更新变量运行时表

解释执行完毕，且源文件还有代码未解释

运行结束

解释执行完毕，且源代码中无待解释代码

其中关于语法问题，例如括号匹配以及单词的顺序问题，由于这种语言只有运算语句以及打印语句，所以没有使用编译原理中的语法分析，而是在生成的单词表中通过逐个验证每个操作符来达到验证的目的。

另外是其中使用了配置文件，主要是配置了单词的最大长度以及数字的最大长度，还有就是错误时显示的字符串。即：

|  |
| --- |
| <?xml version="1.0" encoding="utf-8" ?>  <configuration>  <appSettings>  <!--定义标示符的长度限制 10-->  <add key="NameMaxLength" value="10"/>  <!--定义标示符越界的错误语句-->  <add key ="NameLengthFalse" value="非法标识符,标识符长度不可超越10位!"/>  <!--定义数字的长度限制-->  <add key ="NumberMaxLength" value="9"/>  <!--定义数字长度越界的错误语句-->  <add key="NumberLengthFalse" value="非法数字,数字长度不可超越9位!"/>  </appSettings>  </configuration> |

**涉及到的数据结构：**

在单词表以及变量运行时表中均用到了ArrayList数据结构，主要是存储分析出来的信息。设计的操作有：

单词表中设计的操作：

publicvoidAddItem**(**VarTableItemitem**)**

publicVarTableItemGetItem**(**intindex**)**

publicintLength**()**

publicvoidPrintAll**()**

运行时表中设计的操作：

**public** bool Exists**(**string varName**)**

**public** **object** getItem**(**string varName**)**

**public** void UpdateItem**(object** obj**)**

**public** void insertItem**(object** obj**)**

**public** void printAll**()**

在进行解释运行时，用到了栈ObjectStack数据结构，主要是进行表达式的中缀形式到后缀形式的转换。涉及到的操作有：

**public** void Push**(object** item**)**

**public** **object** Pop**()**

**public** **object** GetTop**()**

**public** bool IsEmpty**()**

**public** int Length**()**

**算法描述：**

三个主要的算法：FileControl中的调度算法、AnalySentence中的分析算法、Calculate中的解释算法。

**FileControl中的调度算法**：主要代码如下：

|  |
| --- |
| public void ExplainRun()  {  StreamReader streamReader = new StreamReader(fileInfo.OpenRead());  string lineString=streamReader.ReadLine();  while(lineString != null)  {  string[] expression = lineString.Split(new char[] { ';' });  for (int i = 0; i < expression.Length; i++)  {  if (!expression[i].Trim().Equals(""))  {    AnalySentence analy = new AnalySentence(expression[i].Trim());  analy.AnalySentenceFunc();    if (analy.GetVarTable().GetItem(0).VarValue.Equals(ReserveTable.Print))  {  string printString = analy.GetVarTable().GetItem(1).VarValue;  System.Console.WriteLine(((RuntimeVarItem)runtimeVarTable.getItem(printString)).VarValue.ToString());  }  else  {  Calculate cal = new Calculate(analy.GetVarTable(), runtimeVarTable);  cal.CalculateFunc();  }      }    }  System.Console.ReadLine();  lineString = streamReader.ReadLine();  }  streamReader.Close();  }  } |

1. 首先读取文件，如果没有到达文件结尾就读取一行，如果打到文件结尾则结束；
2. 对一行中的文本使用“；”封号进行分割，分割出来的即可认为是一条语句；
3. 针对这一条语句，进行词法分析，从中找出每个单词；
4. 如果第一个单词是关键字print，则直接打印出后面的内容，如果变量不存在则报错并退出；
5. 如果不是关键字，则使用解释程序对表达式进行计算；

**AnalySentence中的分析算法**：关键算法如下：

|  |
| --- |
| while (index < sentenceChar.Length)  {  int startIndex = index;  //发现标识符  if (sentenceChar[index] == '\_' || IsLetter(sentenceChar[index]))  {  //标识符长度限制  int limitLength =int.Parse(ConfigurationManager.AppSettings["NameMaxLength"]);  while (limitLength > 0 && index < sentenceChar.Length-1)  {  index++;  if (sentenceChar[index] == '\_' || IsLetter(sentenceChar[index]) || IsDigital(sentenceChar[index]))  {  limitLength--;  }  else  {  index--;  break;  }  }  if (limitLength == 0 || index == sentenceChar.Length)  {  if (limitLength == 0 && sentenceChar[index] == '\_' || IsDigital(sentenceChar[index]) || IsLetter(sentenceChar[index]))  {  System.Console.WriteLine(ConfigurationManager.AppSettings["NameLengthFalse"]);  System.Environment.Exit(0);  }  index--;  }  VarTableItem vti=new VarTableItem();  vti.VarValue=CharSubString(sentenceChar,startIndex,index);  if (vti.VarValue.Equals(ReserveTable.Print))  {  vti.VarType = VarTypes.reserve;  }  else  {  vti.VarType = VarTypes.variable;  }  varTable.AddItem(vti);  }  else if (IsDigital(sentenceChar[index]))  {  //数字的长度限制  int limitLength = int.Parse(ConfigurationManager.AppSettings["NumberMaxLength"]);  while (limitLength > 0 && index < sentenceChar.Length - 1)  {  index++;  if (IsDigital(sentenceChar[index]))  {  limitLength--;  }  else  {  index--;  break;  }  }  if (limitLength == 0 || index == sentenceChar.Length)  {  if (limitLength == 0 && IsDigital(sentenceChar[index]))  {  System.Console.WriteLine(ConfigurationManager.AppSettings["NumberLengthFalse"]);  System.Environment.Exit(0);  }  index--;  }  VarTableItem vti = new VarTableItem();  vti.VarValue = CharSubString(sentenceChar, startIndex, index);  vti.VarType = VarTypes.intNum;  varTable.AddItem(vti);  }  else if (IsOperation(sentenceChar[index]))  {  VarTableItem vti = new VarTableItem();  vti.VarValue = CharSubString(sentenceChar, startIndex, index);  vti.VarType = VarTypes.operation;  varTable.AddItem(vti);  }  index++;  } |

这个一部分是识别代码：使用状态转换图表述算法：

0

1

3

2

字母、下划线

数字

字母、下划线、数字

非字母、非数字、非下划线

数字

4

非数字

6

5

7

8

9

10

=

+

-

\*

/

(

11

)

其中还有关于检错的算法：部分代码如下：

|  |
| --- |
| int bracketRight = 0;  for (int i = 0; i < varTable.Length(); i++)  {    VarTableItem vti = varTable.GetItem(i);  //查看括号的匹配状态  if (vti.VarType == VarTypes.operation && vti.VarValue.Equals("("))  bracketRight++;  else if (vti.VarType == VarTypes.operation && vti.VarValue.Equals(")"))  bracketRight--;  if (i == 0)  {    }  //考虑操作符的情况  if (vti.VarType == VarTypes.operation)  {  if (i == 0)  {  //第一个单元是减号情况  if (vti.VarValue.Equals("-"))  {  VarTableItem vti2 = varTable.GetItem(i + 1);  //后面为操作符的情况  if (vti2.VarType == VarTypes.operation)  {  if (!vti2.VarValue.Equals("("))  {  System.Console.WriteLine("表达式\"" + analyString + "\"不合法,错误位置:第" + (i + 1) + "个标识符附近");  System.Environment.Exit(0);  }  }  //后面为保留字的情况  else if (vti2.VarType == VarTypes.reserve)  {  System.Console.WriteLine("表达式\"" + analyString + "\"不合法,错误位置:第" + (i + 1) + "个标识符附近");  System.Environment.Exit(0);  }  }  else if (!vti.VarValue.Equals("("))  {  System.Console.WriteLine("表达式\"" + analyString + "\"不合法,错误位置:第" + (i + 1) + "个标识符附近");  System.Environment.Exit(0);  }  } |

1. 对于括号的匹配问题：在单词解释出来后，会一次检索每一个单词，其中顺便地，在碰到“（”时变量bracketRight自增1，在碰到“）”时变量bracketRight自减1，所以最后检测一下变量 bracketRight，如果不是0则说明括号不匹配，此时报错并退出。
2. 关于单次以及操作符的顺序问题，我们在定义语言时定义了这一规则，所以在这里就是根据匹配原则检错，如果不匹配则报错并退出。

Calculate中的解释算法：关键代码如下：

|  |
| --- |
| for (int index = 0; index < varTable.Length(); index++)  {  VarTableItem item = varTable.GetItem(index);  if (item.VarType == VarTypes.intNum || item.VarType==VarTypes.variable)  {  dataStack.Push(item);  }  else if (item.VarType == VarTypes.operation)  {  if (opStack.IsEmpty())  {    opStack.Push(item);  }  else  {  int result = Compare2Op(item.VarValue.ToCharArray()[0], ((VarTableItem)opStack.GetTop()).VarValue.ToCharArray()[0]);  if (result == 1)  {  opStack.Push(item);  }  else  {  int result2=0;  do  {    char opTemp = ((VarTableItem)opStack.Pop()).VarValue.ToCharArray()[0];  if (item.VarValue.ToCharArray()[0] == ')' && opTemp == '(')  break;  switch (opTemp)  {  ………………………………..省略  case '-':  int num11 = 0;  if (dataStack.Length() > opStack.Length())  {  VarTableItem objNum11 = (VarTableItem)dataStack.Pop();  if (objNum11.VarType == VarTypes.intNum)  {  num11 = int.Parse(objNum11.VarValue);  }  else if (objNum11.VarType == VarTypes.variable)  {  RuntimeVarItem tempNum11 = (RuntimeVarItem)runtimeVarTable.getItem(objNum11.VarValue);  if (tempNum11 == null)  {  System.Console.WriteLine("无法引用未赋值的变量\"" + objNum11.VarValue + "\"");  System.Environment.Exit(0);  }  else  {  num11 = (int)tempNum11.VarValue;  }  }  int num22 = 0;  VarTableItem objNum22 = (VarTableItem)dataStack.Pop();  if (objNum22.VarType == VarTypes.intNum)  {  num22 = int.Parse(objNum22.VarValue);  }  else if (objNum22.VarType == VarTypes.variable)  {  RuntimeVarItem tempNum22 = (RuntimeVarItem)runtimeVarTable.getItem(objNum22.VarValue);  if (tempNum22 == null)  {  System.Console.WriteLine("无法引用未赋值的变量\"" + objNum22.VarValue + "\"");  System.Environment.Exit(0);  }  else  {  num22 = (int)tempNum22.VarValue;  }  }  VarTableItem num33\_Item = new VarTableItem();  num33\_Item.VarType = VarTypes.intNum;  num33\_Item.VarValue = (num22 - num11).ToString();  dataStack.Push(num33\_Item);  }  else  {  //确定该符号是　负号  VarTableItem objNum11 = (VarTableItem)dataStack.Pop();  if (objNum11.VarType == VarTypes.intNum)  {  num11 = int.Parse(objNum11.VarValue);  }  else if (objNum11.VarType == VarTypes.variable)  {  RuntimeVarItem tempNum11 = (RuntimeVarItem)runtimeVarTable.getItem(objNum11.VarValue);  if (tempNum11 == null)  {  System.Console.WriteLine("无法引用未赋值的变量\"" + objNum11.VarValue + "\"");  System.Environment.Exit(0);  }  else  {  num11 = (int)tempNum11.VarValue;  }  }  VarTableItem num33\_Item = new VarTableItem();  num33\_Item.VarType = VarTypes.intNum;  num33\_Item.VarValue = (-num11).ToString();  dataStack.Push(num33\_Item);  }  break;    if (opStack.IsEmpty())  {  break;  }  else  {  result2 = Compare2Op(item.VarValue.ToCharArray()[0], ((VarTableItem)opStack.GetTop()).VarValue.ToCharArray()[0]);  }  } while (result2==-1 || result2==0);  if (item.VarValue.ToCharArray()[0]!=')')  opStack.Push(item);  }  }  }  } |

这里主要是一个表达式转换算法：

1. 对于一个单词，如果是标识符或数字则直接压入数据栈；
2. 对于一个操作符，如果此时操作符栈为空，这直接压入栈；如果不为空，则对比栈顶操作符的优先级，如果比栈顶操作符优先级高，则压入；否则，弹出所有不比他优先级高的操作符；
3. 对于弹出的每一个操作符，取数据栈中的数据运算即可，最后将运算后的数据压入数据栈。
4. 需要注意的是，注意负号与减号的识别，这里在拿到“-”符号时，对比数据栈与符号栈中的实体数量，如果数据栈内容数量>=符号栈内容数量，则是别为负号，否则识别为减号。

**算法时间性能分析：**

三个算法均是线性的，即：

FileControl中的调度算法o(n)

AnalySentence中的分析算法o(n)

Calculate中的解释算法o(n)

**涉及到的技术：**

C#语言基本语法

XML配置文件

C#文件读写

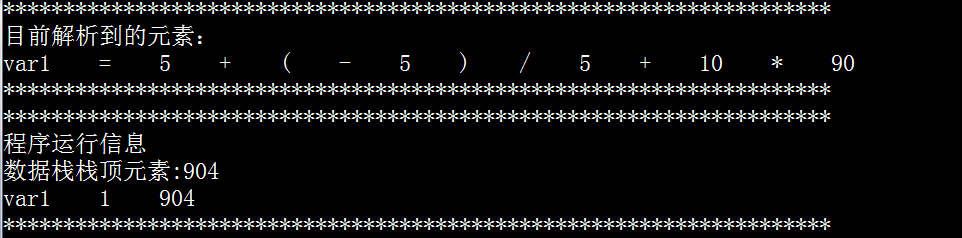
字符串解析

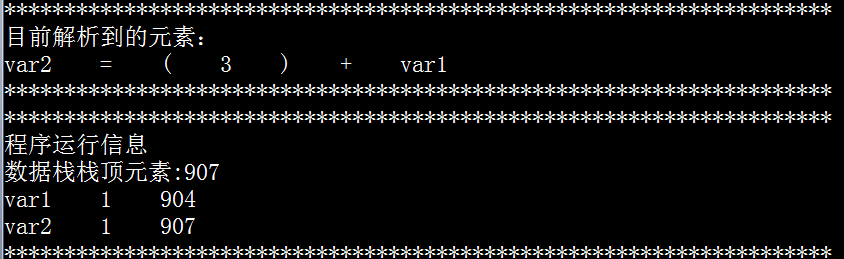
**运行结果**

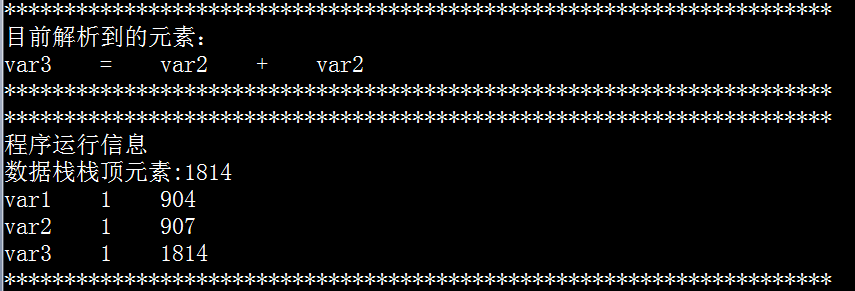
源文件：w.se

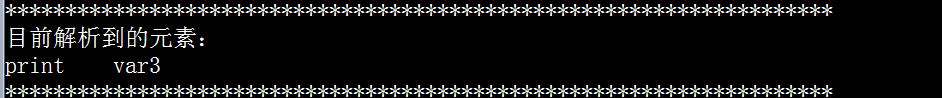
|  |
| --- |
| var1=5+(-5)/5+10\*90;  var2=(3)+var1;  var3=var2+var2;  print var3; |

控制台运行结果（其中包含一些程序运行的显示信息，便于演示）：

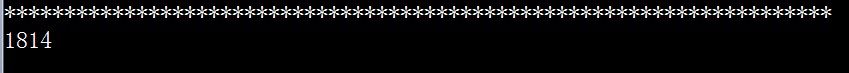








最终结果：



**实验二**

**实验题目：记事本**

**实验描述：**

“记事本”是一个用来创建简单的文档的基本的文本编辑器。“记事本”最常用来查看或编辑文本(.txt) 文件，在我们开发的这个记事本中，需要支持的功能如下：

* 常规编辑，不需要支持格式；
* 查找特定的字符或词；
* 替换特定的字符或词；
* 具有英文单词的拼写检查能力。

**实验要求：**

1、界面必须是图形界面

**问题分析：**

这个题目主要是设计一个文本编辑器，重点是在愈发检测这样模块，为了便于编辑，我在记事本中加入了类似VS编辑器的即使窗口的功能。

对于常规的编辑，需要实现打开文件、新建文件、保存文件、另存为文件、退出，换行。

对于查找替换，使用基本的查找算法即可实现。

对于英文检错，根据题目提供的词典，建立一棵树来实现。

对于即使窗口功能，使用继承组件Component即可实现。

**涉及到的数据结构：**

在英文单词检错方面使用了单词查找树；

在提取数据时使用了.Net中提供的DataTable数据结构，以及DataView数据结构；

在从文件中分析单词以及解释时，使用了字典Dictionary数据结构；

**算法描述：**

首先对于一些基本的保存，新建操作，由于在空间中提供了相应的接口，因此这里没有做得太底层：部分代码（在相应的事件中）：

|  |
| --- |
| private void OpenFileStripItem\_Click(object sender, EventArgs e)  {  if ((!string.IsNullOrEmpty(this.ContentTxt.Text) && this.isNew) || (this.isNew && !this.savedFileName.Equals("")))  {  DialogResult result = MessageBox.Show("当前文件未保存，是否保存文件", "提示", MessageBoxButtons.YesNoCancel);  if (result == DialogResult.Yes)  {  if (this.isNew && !this.savedFileName.Equals(""))  {  this.ContentTxt.SaveFile(this.savedFileName, RichTextBoxStreamType.PlainText);  this.Text = this.savedTitle;  this.isNew = false;  }  else if (this.isNew)  {  string fileFullName = this.SaveFile("保存文件");  if (fileFullName != null)  {  this.Text = this.savedTitle;  this.statusPanelFile.Text = "文件:" + fileFullName;  this.isNew = false;  this.savedFileName = fileFullName;  }  }  string loadFileName = this.OpenFile();  if (loadFileName != null)  {  this.Text = this.savedTitle;  this.statusPanelFile.Text = loadFileName;  this.isNew = false;  this.savedFileName = loadFileName;  }  this.ContentTxt.SelectionStart = 0;  }  else if (result == DialogResult.No)  {  string loadFileName = this.OpenFile();  if (loadFileName != null)  {  this.Text = this.savedTitle;  this.statusPanelFile.Text = loadFileName;  this.isNew = false;  this.savedFileName = loadFileName;  }  this.ContentTxt.SelectionStart = 0;  }  else if (result == DialogResult.Cancel)  {  }  }  else  {  string loadFileName = this.OpenFile();  if (loadFileName != null)  {  this.Text = this.savedTitle;  this.statusPanelFile.Text = loadFileName;  this.isNew = false;  this.savedFileName = loadFileName;  }  this.ContentTxt.SelectionStart = 0;  }  } |

这一段是打开文件的代码，完成新建文件的功能，但是考虑到文本框中可能有为保存的内容，所以需要检测标识变量，如果发现是新文件且为保存，则弹出提示保存文件；在这些前提工作做完后在展示打开文件窗口。其中需要注意更改一些窗体中的控件的状态，例如在状态栏那种显示新打开文件的路径、在文本窗口中标题前加“\*”以表示文件以修改但未保存。

类似的，新建文件，这里不赘述。

对于查找替换，同样在控件中提供相应的接口，但是为了支持连续查找的功能，添加了一个标识变量searchString以及searchPosition，代码如下：

|  |
| --- |
| private void FindStripItem\_Click(object sender, EventArgs e)  {  FindReplaceForm findReplaceForm = new FindReplaceForm(true);  if (DialogResult.OK == findReplaceForm.ShowDialog())  {  if (!this.isFind)  {  this.searchPosition = 0;  }  string findString = findReplaceForm.FindTxt.Text;  searchString = findString;  if (!this.searchString.Equals(findString))  searchPosition=0;  int nowIndex = this.ContentTxt.Find(findString, searchPosition, RichTextBoxFinds.None);  if (nowIndex != -1)  {  searchPosition = nowIndex + this.ContentTxt.SelectionLength;  }  else  {  searchPosition = 0;  }  this.isFind = true;  }    } |

然后是对于单词查找树，主要分为两部分：建树以及遍历。

类的结构如下：

NODE

LEAFNODE

BRANCHNODE

建树的代码如下：

|  |
| --- |
| private void BuildBranch(Node node,List<string> wordList,int searchDepth)  {  char searchChar='a';  while (searchChar <= 'z')  {  List<string> nowList = GetGroup(wordList,searchDepth,searchChar);  if (nowList.Count == 1)  {  Node nowLeaf = new LeafNode(nowList[0],node);  if (node.GetFirstChildNode() != null)  {  Node tempNode = node.GetFirstChildNode();  while (tempNode.GetBrotherNode() != null)  {  tempNode = tempNode.GetBrotherNode();  }  tempNode.SetBrother(nowLeaf);  }  else  node.SetChild(nowLeaf);  }  else if (nowList.Count > 1)  {  List<string> nextList = new List<string>();  foreach (string word in nowList)  {  char[] wordArray = word.ToCharArray();  if (searchDepth == (wordArray.Length-1))  {  Node nowLeaf = new LeafNode(word, node);  if (node.GetFirstChildNode() != null)  {  Node tempNode = node.GetFirstChildNode();  while (tempNode.GetBrotherNode() != null)  {  tempNode = tempNode.GetBrotherNode();  }  tempNode.SetBrother(nowLeaf);  }  else  node.SetChild(nowLeaf);  }  else  {  nextList.Add(word);  }  }  if (nextList.Count > 1)  {  Node nowBranch = new BranchNode(NodeType.Branch, node, searchChar.ToString());  if (node.GetFirstChildNode() != null)  {  Node tempNode = node.GetFirstChildNode();  while (tempNode.GetBrotherNode() != null)  {  tempNode = tempNode.GetBrotherNode();  }  tempNode.SetBrother(nowBranch);  }  else  node.SetChild(nowBranch);  BuildBranch(nowBranch, nextList, searchDepth + 1);  }  else if (nextList.Count == 1)  {  Node nowLeaf = new LeafNode(nextList[0], node);  if (node.GetFirstChildNode() != null)  {  Node tempNode = node.GetFirstChildNode();  while (tempNode.GetBrotherNode() != null)  {  tempNode = tempNode.GetBrotherNode();  }  tempNode.SetBrother(nowLeaf);  }  else  node.SetChild(nowLeaf);  }  }  searchChar++;  }  } |

建树的过程是一个递归的过程，这个函数主要是检索单词列表的第searchDepth个单词，首先根据这个位置的单词进行分组，最多有26个组（a-z），然后分析组中的数量。

1. 如果组中只有一个元素，这说明此时已经可以唯一标识该单词，则创建叶节点并添加到当前的父节点的子节点中；
2. 如果组中的元素多个，但是此时的深度已经到达某些单词的末尾，则这些单词也应该i拿出来直接建为叶节点，然后添加到父节点的子节点中。
3. 根据上面分析后的列表再分析个数，如果只有一个则直接建立叶节点并添加到父节点的子节点中；如果多于一个，则生成一个分支节点，并且基于该分支节点递归调用。

如此往复，即可建成单词查找树。从中可以看出这个单词查找树使用的是第一个子节点与父节点连接，但是其他子节点作为第一个子节点的右兄弟节点，并且叶节点并不是孤立的，他也可能存在有兄弟节点。

遍历的算法即为搜索单词的过程：代码如下：

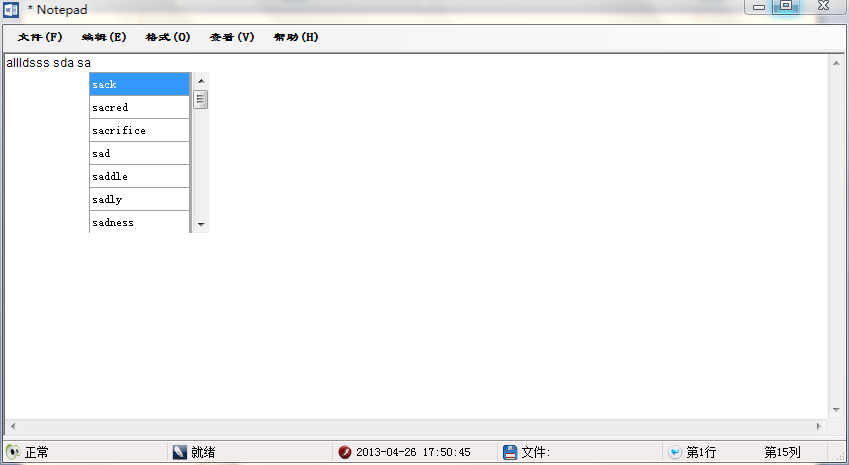
|  |
| --- |
| private bool wordExists(Node node, string letter,int index)  {  Node child = node.GetFirstChildNode();  char[] letterArray = letter.ToCharArray();    while (child != null)  {  if (child.GetNodeType() == NodeType.Leaf)  {  if (child.GetValue().Equals(letter))  return true;  else  child = child.GetBrotherNode();  }  else  {  if (child.GetValue().Equals(letterArray[index].ToString()))  break;  else  child = child.GetBrotherNode();  }  }  if (child == null)  return false;  else  return wordExists(child,letter,index+1);  } |

检索的过程比较简单，也是一个递归调用过程，即根据此时的节点的子节点，如果是叶节点则检查内容是否与搜索的内容一致，如果一致则返回true，否则查找下一个子节点；如果是分支节点则检查当前的搜索字符是否与分支节点保存的字符一致，如果一致则对该自己点进行递归，否则继续查找下一个子节点。

最后是关于及时窗口实现算法，这里涉及到一个控件与组件的一个区别，使用组件是不会占用窗体的控件的，这样我们就可以根据条件的显示DataList控件。另外一个关键是BringToFont函数，这个函数类似CSS中的Z-index标签，可以实现叠加显示。部分显示代码：

|  |
| --- |
| private void sender\_TextChanged(object sender, EventArgs e)  {    RichTextBox richText = sender as RichTextBox;  Control container = \_form;  int nowIndex = richText.SelectionStart;  char nowChar = richText.GetCharFromPosition(richText.GetPositionFromCharIndex(nowIndex));  //MessageBox.Show(nowIndex+":::"+nowChar);  if ((nowChar >= 'a' && nowChar <= 'z') || (nowChar >= 'A' && nowChar <= 'Z') || (nowChar == '\t') || (nowChar == ' ') || (nowChar == '\n'))  {  int wordStartIndex = nowIndex;  if ((nowChar == '\t') || (nowChar == ' ') || (nowChar == '\n'))  wordStartIndex--;  string keyString = "";  char wordStartChar = ' ';  do  {  if (wordStartIndex == 0) break;  wordStartChar = richText.GetCharFromPosition(richText.GetPositionFromCharIndex(wordStartIndex));  //MessageBox.Show(wordStartIndex + "::" + wordStartChar);  wordStartIndex--;  } while (wordStartChar != ' ' && wordStartChar != '\t' && wordStartChar != '\n' && (nowIndex - wordStartIndex) <= 20);  //MessageBox.Show("LastWord:"+wordStartIndex);  if ((nowIndex - wordStartIndex) <= 20)  {  if ((wordStartIndex != 0) || (wordStartIndex == 0 && wordStartChar == ' '))  wordStartIndex += 2;  if (wordStartIndex > nowIndex)  {  \_view.Visible = false;  return;  }  richText.Select(wordStartIndex, nowIndex - wordStartIndex);  keyString = richText.SelectedText;  //MessageBox.Show(keyString);  richText.Select(nowIndex, 0);  }  else  {  \_view.Visible = false;  return;  }  if (keyString.Equals(""))  {  \_view.Visible = false;  return;  }  \_view.DataSource = \_adapters[richText.Name].GetFitData(keyString);  Point editPoint = richText.GetPositionFromCharIndex(richText.SelectionStart);  \_view.Location = new Point(editPoint.X, editPoint.Y + 30 + (int)(richText.Font.Height \* richText.ZoomFactor));  \_view.Visible = true;  container.Controls.Add(\_view);  \_view.BringToFront();  int tableCount = \_adapters[richText.Name].GetFitData(keyString).Count;  if (tableCount == 0)  {  \_view.Visible = false;  }  else if (tableCount <= viewDisplayNumber)  {  \_view.Height = \_view.Rows[0].Height \* tableCount;  }  else  {  \_view.Height = \_view.Rows[0].Height \* viewDisplayNumber;  }  }  else  {  \_view.Visible = false;  richText.Focus();  }  } |

这段代码是根据用户输入的字符串（这里需要检索出字符串），并把它作为DataView的查找字符串，如下：



图中用户输入sa，则这里我们系统就应该分析出sa这个字符串，然后把查找出的匹配字符串用DataGridView显示出来。当然其中会涉及一些Windows消息函数的调节。

|  |
| --- |
| container.Controls.Add(\_view);  \_view.BringToFront(); |

其中的\_form是这样组织的：

|  |
| --- |
| if (\_form == null)  {  \_form = sender;  while (\_form.Parent != null)  {  \_form = \_form.Parent;  }  } |

基本就这三个算法。另外就是委托的使用，例如在AutoCompleteWord中就是使用委托的机制，使文本框在输入文本时显示及时窗口的。

**算法时间性能分析：**

单词查找树中键的深度取决于关键字中字符或数位的个数。假设关键字伪随机的，则在树中查找每一位的平均长度为1/2(1+d)，有假设关键字中字符的个数都相等，则在书中警察找的平均长度为h/2(1+d)。

**涉及到的技术：**

C#基本语法

Windows消息机制

C#委托机制

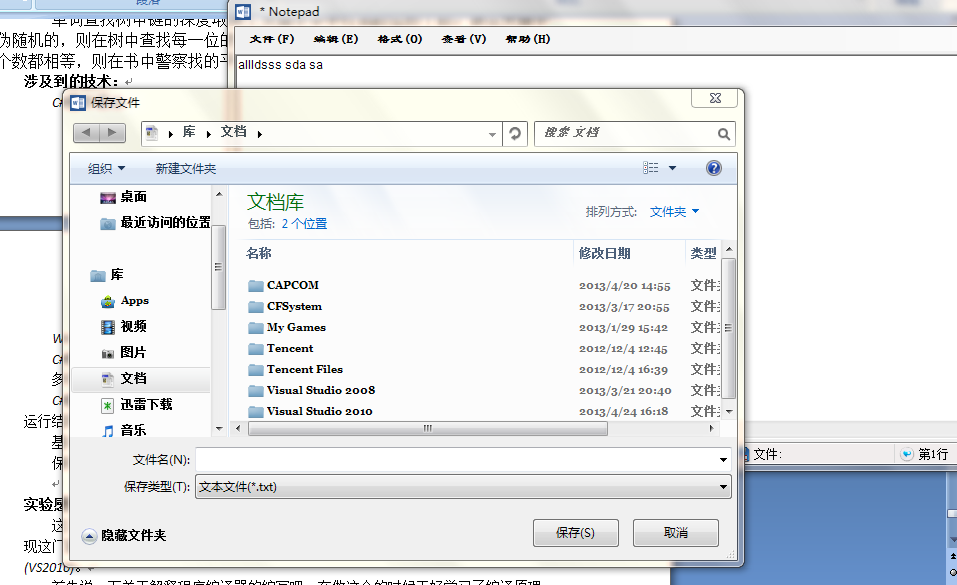
多线程

C#窗体设计

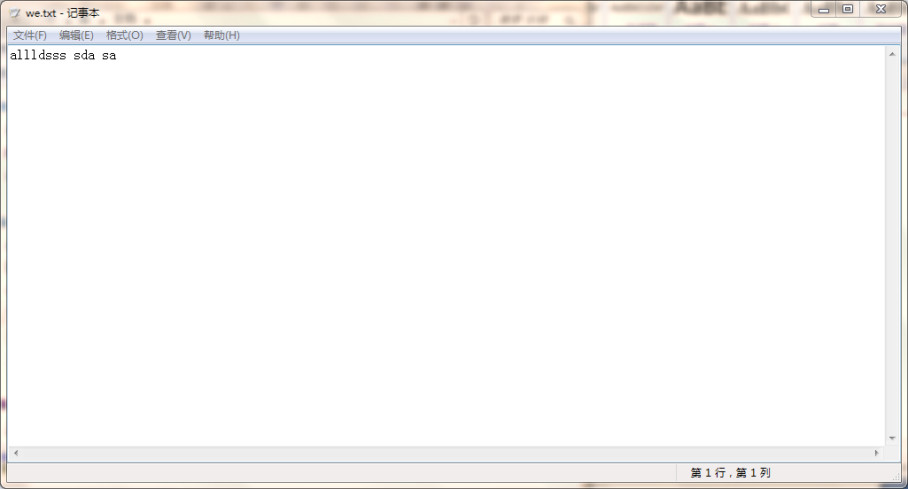
**运行结果：**

基本功能：（以保存和查找为例）

**保存：**

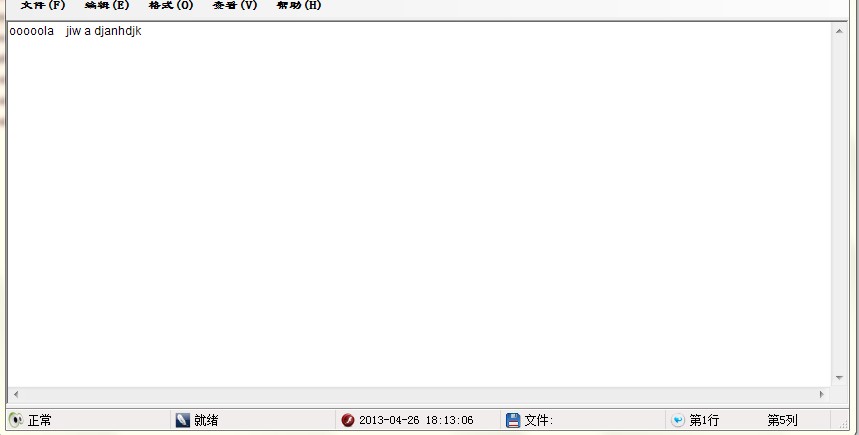


保存后的结果：

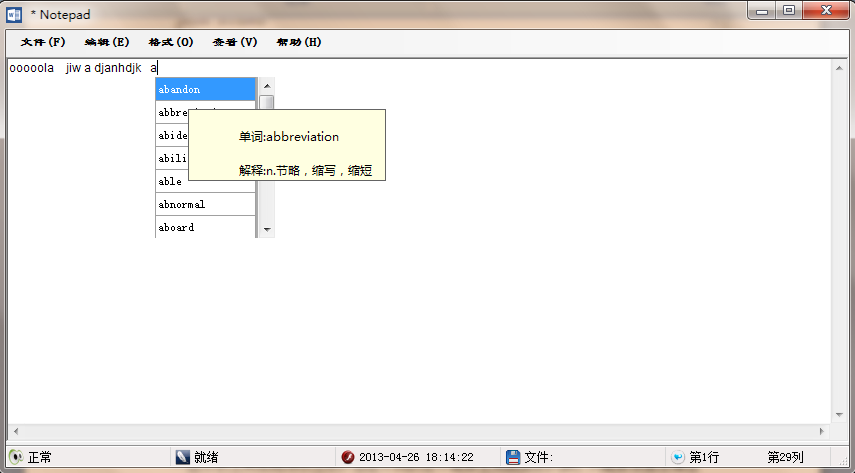


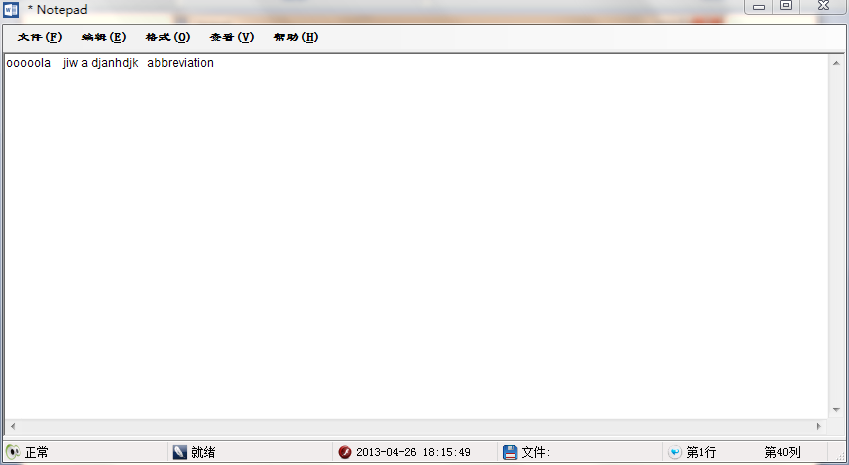
**替换：**



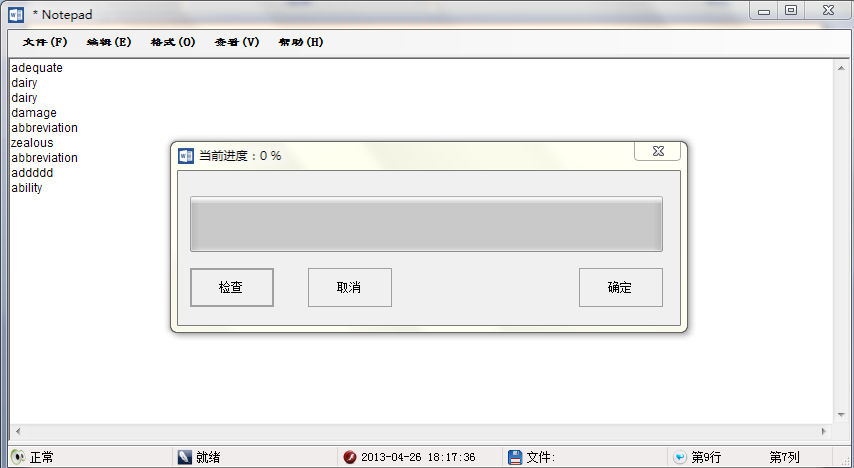


**及时窗口：**

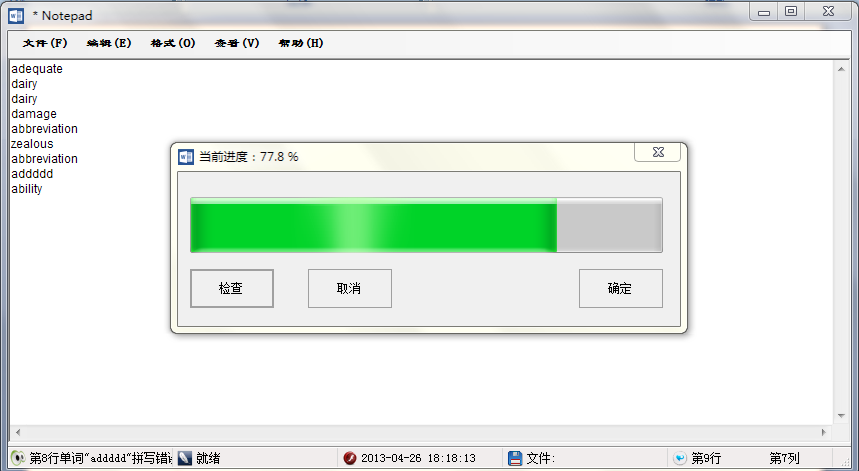




**单词检错：**

****

**发现错误**

****

**实验感悟**

这次实验主要使用C#编写，首先就是对这门语言有了更深的认识，而且越来越发现这门语言的方便之处，不仅仅在于面向对象的封装性，另一方面在于方便的编写环境(VS2010)。

首先说一下关于解释程序编译器的编写吧，在做这个的时候正好学习了编译原理，所以其中的很多方法都可借鉴过来，例如其中的一些记录性的表格以及正则文法、还有一些分析工具如有序状态机。但是不同的是不需要生成底层的代码，只需要根据逻辑显示出结果即可。但是其中还有许多不足，例如语法的检测不够规范，使用了一种比较生涩的方式，代码冗余量较大。

另一方面，在之前学习数据结构时注重了理论，这次使用中缀表达式与后缀表达式的转换算法时才发现当时的许多的错误。例如应该弹出优先级相等以及较低的运算符，在转换为后缀表达式时可以顺便运算，这样提供高了运算速度，也节省了代码。

关于记事本的编写，收益最深的是关于界面的设计，关于及时窗口的编写查阅了很多资料，最后研究一些网上的代码并且结合了自己的项目才得以实现。其中关于DataTable、DataView、DataGridView三种数据结构以及控件的关系，以及控件与组件的区别都在设计过程中有了更深的理解。

英文检错的实现使用了树的数据结构，不仅节省了数据空间，而且提高了检索速度。领悟更深的是对于树这种数据结构的使用。

最后非常感谢元盛老师在数据结构课上的悉心教导。