语法分析器设计分析报告

1850845 虞枫毅

1852839 李培然

1. **需求分析**
   1. **程序任务输入范围**

本程序用于针对符合语法规则的类C语言段进行语法分析，并辨别此段输入是否符合之前确定好的语义规则，因此，此程序针对的输入为一段属于类C语言的代码段，或一句符合类C语言语法规则的代码语句。

* 1. **输出形式**

本程序对符合程序要求输入的代码段或代码文件进行语法分析，可选择将分析过程重定向至文件，分析过程和语法分析树都将被输出至文件，使用一句简单的类C语言代码作为示例，基础输出文件情况如下（需要使用强大一点的文字编辑器打开避免自动换行破环此树）：

Token元组中的数据为当前词法分析器所返回的单词，里面包括单词种类，单词值，单词所在位置的索引，Token下面的list列表为当前状态栈内的情况，下面的元组，前一个元素为Action表中此时应该执行的动作，如果下一步为移进，则右侧的数字表示下一个该被压进状态栈的状态，如果下一步为归约，则右侧数字代表归约时使用生成式在生成式列表中的索引。

最下方SUCCESSFUL之后的值，为语法分析成功之后输出的语法分析树，此语法分析树是逆时针旋转90度之后输出的树，我们正在寻找更好的表现方式来表现语法分析树。

* 1. **程序功能**

本程序是一个类C语言的语法分析器，其中包含词法分析程序和语法分析程序，其中词法分析程序被语法分析程序调用。

本程序可将输入程序符合条件的文法转为LR(1)分析表，并可将其保存以便于下一次使用，或者加载符合程序要求的LR(1)文法分析表使用文法分析器进行分析。此程序中包含的语法分析程序为LR(1)分析程序，其调用词法分析器对源文件进行初步处理并提取词元素，之后根据程序自动生成或加载的LR(1)分析表对文法进行分析，分析的同时可输出分析过程，在分析成功后可以PDF或ANSI字符的形式输出源文件的语法分析树。

* 1. **测试数据**
     1. 正确数据：C语言实现的二分查找

int main()

{

int c, first, last, middle, n, search, array[100];

printf("Enter number of elements");

scanf("%d",&n);

printf("Enter %d integers", n);

for (c = 0; c < n; c++)

scanf("%d",&array[c]);

printf("Enter value to find");

scanf("%d", &search);

first = 0;

last = n - 1;

middle = (first+last)/2;

while (first <= last) {

if (array[middle] < search)

first = middle + 1;

else if (array[middle] == search) {

printf("%d found at location %d.", search, middle+1);

break;

}

else

last = middle - 1;

middle = (first + last)/2;

}

if (first > last)

printf("Not found! %d isn't present in the list.", search);

return 0;

}

* + 1. 错误数据：非法定义

int main()

{

int 334a;

int 33qraf;

void www;

1. **概要设计**
   1. **任务分解**

一个LR(1)语法分析器主要由两部分构成，词法分析器和语法分析器，而要想使用语法分析器对源文件进行分析，则需要依赖一张由Action和GOTO表组成的LR分析表，本程序的设计要求我们完成自动生成此分析表的功能，因此整个任务最终被分为了三部分：

1. 词法分析器
2. 语法分析器
3. LR分析表生成器
   1. **数据类型的定义**
      1. 词法分析器类

词法分析器类是文法分析程序的重要组成部分，此类负责从文件中读取数据并封装成格式化的Token，交由语法分析器进行分析。

此类包含以下属性：

|  |  |
| --- | --- |
| 属性名及其类型 | 作用及功能 |
| Data:string | 保存从文件中读取的数据 |
| REs:List[Word(type,RE)] | 保存正则表达式列表以及正则表达式所对应的词类型，例如ID |
| Index:int | 保存当前词的位置索引便于输出报错信息和正则匹配 |
| SizeofFile:int | 保存文件大小，用于判断匹配停止条件 |

包含以下方法：

|  |  |
| --- | --- |
| 方法名 | 作用及功能 |
| \_\_init\_\_(path) | 类构造函数，接受一个string类型的路径输入，对文件进行读取并初步处理，初始化对象。 |
| Scan() | 使用正则表达式对文件内容进行匹配，返回匹配到的最长串及其类型 |
| Next() | 此方法为一个生成器，调用此对象可生成一个迭代器，每次返回一个Token，其中调用Scan()取得其返回值做进一步处理 |

除这些属性和方法外，此类中还使用了两个简单定义的命名元组Token和Word，Token用于封装一个固定规格的返回值，其中包含三个值，type，用于指定此词的类型，例如ID，value用于指定整个词的值，例如“main”，index用于指定当前词的位置索引。Word用于帮助正则表达式确定匹配的类型，它只包含两个类型的值。Type为当前正则表达式对应的词类型，RE为当前匹配的正则表达式。

* + 1. 语法分析器类

语法分析器类是语法分析器的核心，此类负责根据从词法分析器送来的数据和LR(1)分析表，对源文件输入的数据进行文法分析，并生成语法分析树。

此类包含以下属性：

|  |  |
| --- | --- |
| 属性名及其类型 | 作用及功能 |
| S:Scanner | 词法分析器实例 |
| Token:List(Token) | 保存从词法分析器实例传过来的Token队列 |
| Status:List(int) | 当前状态栈 |
| Symboi:List(Node(type,Children)) | 当前符号栈 |

此类中也使用了一个简单定义的命名元组Node，此元组是为了生成语法树而存在的，当LR分析表指示动作为归约时，Token将被归约，形成一个新的Node，原来的Token成为这个Node的一个Children，被归约后的类型成为了这个Node的type，如果之后这个Node也被归约了，那么这个Node将成为其他Node的Children，按照这种规律，如果最终分析成功，那么符号栈将仅剩下一个节点，并且这个节点就是语法生成树的根节点，根据这样的设计，我们可以轻松地在分析完文件内容之后生成一棵语法生成树。

此类包含以下属性：

|  |  |
| --- | --- |
| 方法 | 作用及功能 |
| \_\_init\_\_() | 类构造函数，接受一个string类型的路径输入，完成对对象的初始化 |
| parser(extendTable,actionTable,gotoTable) | 核心函数，使用分析表，利用状态栈符号栈等对语法进行分析 |
| showParserTree() | 将语法分析树展现出来（打印） |

* + 1. LR分析表构造器

LR分析表因为函数繁多，且联系不密切，因此并没有封装成为一个类，但放在了一个文件中，因此我们来查看这些函数的作用：

|  |  |
| --- | --- |
| 函数名 | 作用及功能 |
| Nullable\_Set(Grammar\_Table) | 求解Nullable集 |
| Terminal\_Set(Grammar\_Table): | 求解Terminal集 |
| First\_Closure(terms,FirstS,TerminalS, NullS) | 求解单独一个式子的First集 |
| First\_Set(Grammar\_Table,Terminal,NullS) | 求解整个产生式的First集 |
| Follow\_Set(Grammar\_Table,FirstS,Terminal, NullS) | 求解整个产生式的Follow集 |
| Nullable\_Closure(terms, NullS) | 求解单独一个式子的Nullable集 |
| Calc\_Closure(items,Extended\_Table,FirstS, TerminalS, NullS) | 求出一个项集的闭包 |
| LR1\_Table(Extended\_Table,FirstS,TerminalS, NullS, Start\_Symbol) | 求出LR分析表 |

1. **详细设计**
   1. **顶层总体设计及函数调用关系**



* 1. **模块详细设计**

本次的语法分析器主要分为了三个模块进行实现，有词法分析器，语法分析器和LR分析表生成器三个部分。以下将对此三个模块的设计分别进行描述。

* + 1. 词法分析器

词法分析器的功能主要是读取源文件，对源文件进行初步处理，并按照词法对其进行拆分，便于语法分析器进行语法分析。

词法分析器最主要的作用就是识别当前输入的字符串，识别其类型后发送给语法分析器供其进行语法分析，我使用了正则表达式匹配来完成此项功能，一方面，词法分析的实质即使用多个DFA进行匹配，找到最终能到达接受状态的一个DFA，并将该DFA对应的词类型返回，正则表达式可看作是DFA的一种表达方式，并且Python对正则表达式提供了很好的支持，我们只需提前构造好正则表达式，并将所需进行匹配的正则表达式放进一个数组中，依次进行匹配，找到匹配长度最长的一个，此时被匹配的词即可被分类为当前正则表达式所对应的类型。正则表达式的使用不仅可以帮助识别词的种类，也可以帮助我们进行错误识别、空白跳过和注释处理。在词法分析器类的Scan()方法中实现了上述算法，但只有此方法对于词法分析器还是不够的，上述方法仅能做出初步匹配，其中还包含了注释、空白等无用信息，因此我们需要用另一个函数调用它，对它的匹配值进行进一步的处理，并作为可供调用的接口供语法分析器使用，使用的正则表达式可见KeyWord.py文件。

本来认为，为便于语法分析器的调用，将词法分析器接口作为一个迭代器使用很合适，因为这个场景非常适合迭代器来发挥作用，每当语法分析器需要下一个词时，它调用next()方法即可轻松获得一个值，而词法分析器也根据需求按次执行，这样的实现方法不仅使得程序执行效率提高，也在一定程度上节省了很多空间，因此词法分析器类中的next()方法变成了一个生成器，此方法可以帮助进一步处理Scan()方法识别的词类型，返回符合规范的Token，并且使用一个循环，使其成为一个源源不断识别词类型的生成器，我们只要调用此方法，即可获得一个迭代器，此迭代器的作用就是不断返回识别好词的Token，具体实现如下：

def next(self):

while self.index < self.sizeofFile:

res = self.scan()

if res.type != SPACE:

if res.type == 'KEY\_WORDS':

yield Token("'%s'" % res.value,res.value,res.index)

elif res.type == ERR\_ID:

yield Token("ERROR","ID\_INVALID",res.index)

else:

yield res

yield Token("$","$",self.index)

* + 1. LR(1)分析表生成

大多数用上下文无关文法描述其语法的程序设计语言都有一个LR（1）文法，构造LR（1）文法构造LR（0）类似。

LR(1)文法的意思是从左向右扫描，最右推导，往前多看一个字符。

LR(1)文法也需要构造要给预测分析表，但是LR(1)的的预测分析表有两部分，分别是Action表和Goto表。

LR(1)状态是由LR(1)的项组成的集合，并且存在着合并该超前符号的LR(1)的Closure操作和Goto操作。

Closure操作：

（1） 假定I是一个项目集， I 的任何项目都属于CLOSURE(I)。

（2） 若有项目 A→α·Bβ,a 属于CLOSURE(I)，B→γ是文法中的产生式，β∈V\*，b∈FIRST(βa)， 则 B→·γ,b 也属于CLOSURE(I)中。

（3） 重复b)直到CLOSURE(I)不再增大为止。

for 任意产生式

for 任意在中，

没有改变

Goto操作：

中的任意项

Action和Goto表的构建：

假设已构造出LR(1)项目集规范族为：C={I0,I1, … , In}，其中Ik为项目集的名字，k为状态名，令包含S′→·S项目的集合Ik的下标k为分析器的初始状态。那么分析表的ACTION表和GOTO表构造步骤为：

1. 若项目A→α·aβ属于Ik且转换函数GO(Ik,a)= Ij，当a为终结符时则置ACTION[k,a]为Sj。
2. 若项目A→α· 属于Ik，则对任何终结符a 和’#'号置ACTION[k,a]和ACTION[k,#]为”rj”，j为在文法G′中某产生式A→α的序号。
3. 若GO(Ik,A)＝Ij，则置GOTO[k,A]为”j”，其中A为非终结符。
4. 若项目S′→S·属于Ik，则置ACTION[k,#]为”acc”，表示接受。
5. 凡不能用上述方法填入的分析表的元素，均应填上”报错标志”。为了表的清晰我们仅用空白表示错误标志。
   * 1. 语法分析器

语法分析器的实现是整个系统相对比较简单的部分，词法分析器和LR分析表已经替它负重前行了，所以它要做的事仅仅是，向词法分析器索要下一个词，按照action和GOTO表的指令更新自己的符号栈和状态栈，总体而言非常简单，不过在我们的语法分析器中包含一个较为特殊的设定，之前有提到，我们在我们的符号栈中保存了语法树。这里有个蛮有趣的设计，我们的符号栈会直接保存词法分析器发来的Token，此Token包含三个属性，type,value和index，之前也有提到，在进行归约时，我们的符号不会出栈，而是会成为一个Node的Children，此Node将成为新加入符号栈的元素，这样做会影响符号栈符号的读取吗，不会的，因为Node中有一个和Token的同名属性type用来保存符号值，因此虽然我们使用符号栈保存了整棵语法树，但甚至不需要进行特殊处理就能进行正常的文法分析。而保存好的语法树，也正是我们需要的。

由于时间的关系，我们仅采用了一种简单的树可视化方法，即将树逆时针旋转了90度使用GBK编码进行输出，而由于整棵语法树较大，所以需要使用一些特殊的编辑器进行查看，例如Notepad++或VS code，这部分算法不重要，暂时略过不谈。

1. **调试分析**
   1. **简单正确代码测试**

源代码：

int main()

{

int a = 10;

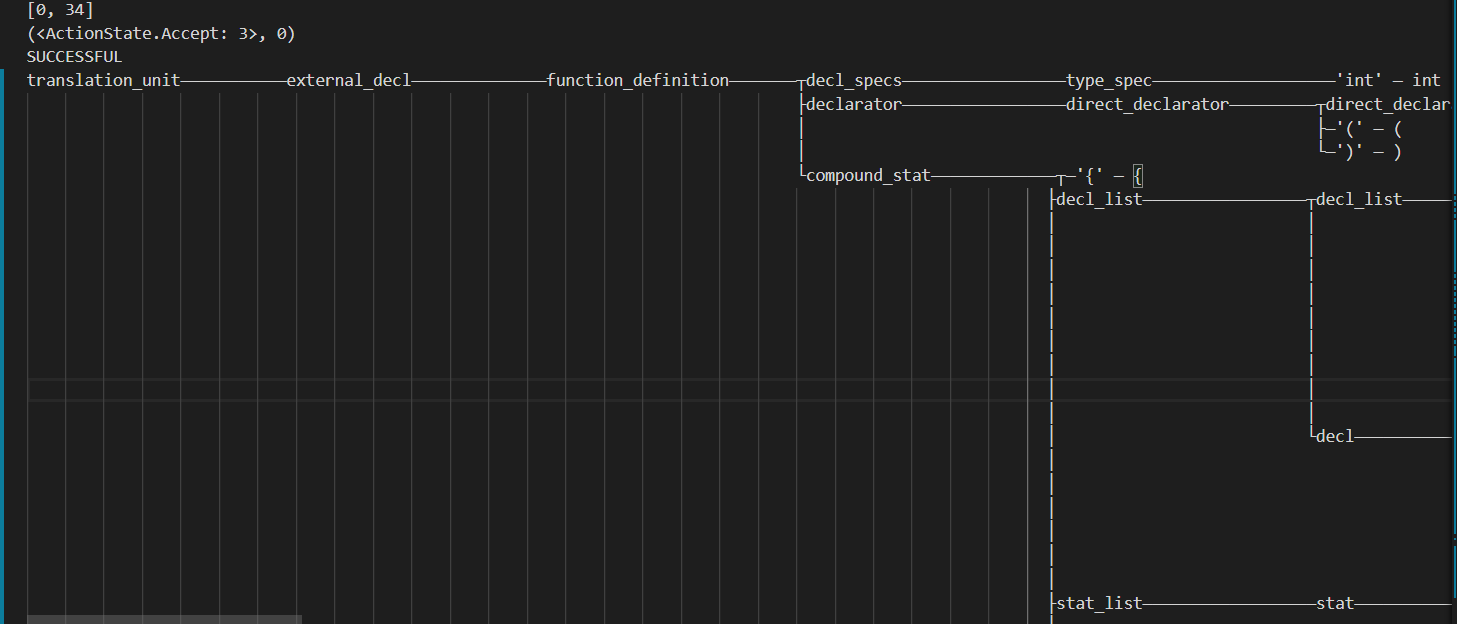
int b = 20;

int c = a + b;

return 0;

}

测试结果：正确



* 1. **复杂正确代码测试**

源代码：

int main()

{

int c, first, last, middle, n, search, array[100];

printf("Enter number of elements");

scanf("%d",&n);

printf("Enter %d integers", n);

for (c = 0; c < n; c++)

scanf("%d",&array[c]);

printf("Enter value to find");

scanf("%d", &search);

first = 0;

last = n - 1;

middle = (first+last)/2;

while (first <= last) {

if (array[middle] < search)

first = middle + 1;

else if (array[middle] == search) {

printf("%d found at location %d.", search, middle+1);

break;

}

else

last = middle - 1;

middle = (first + last)/2;

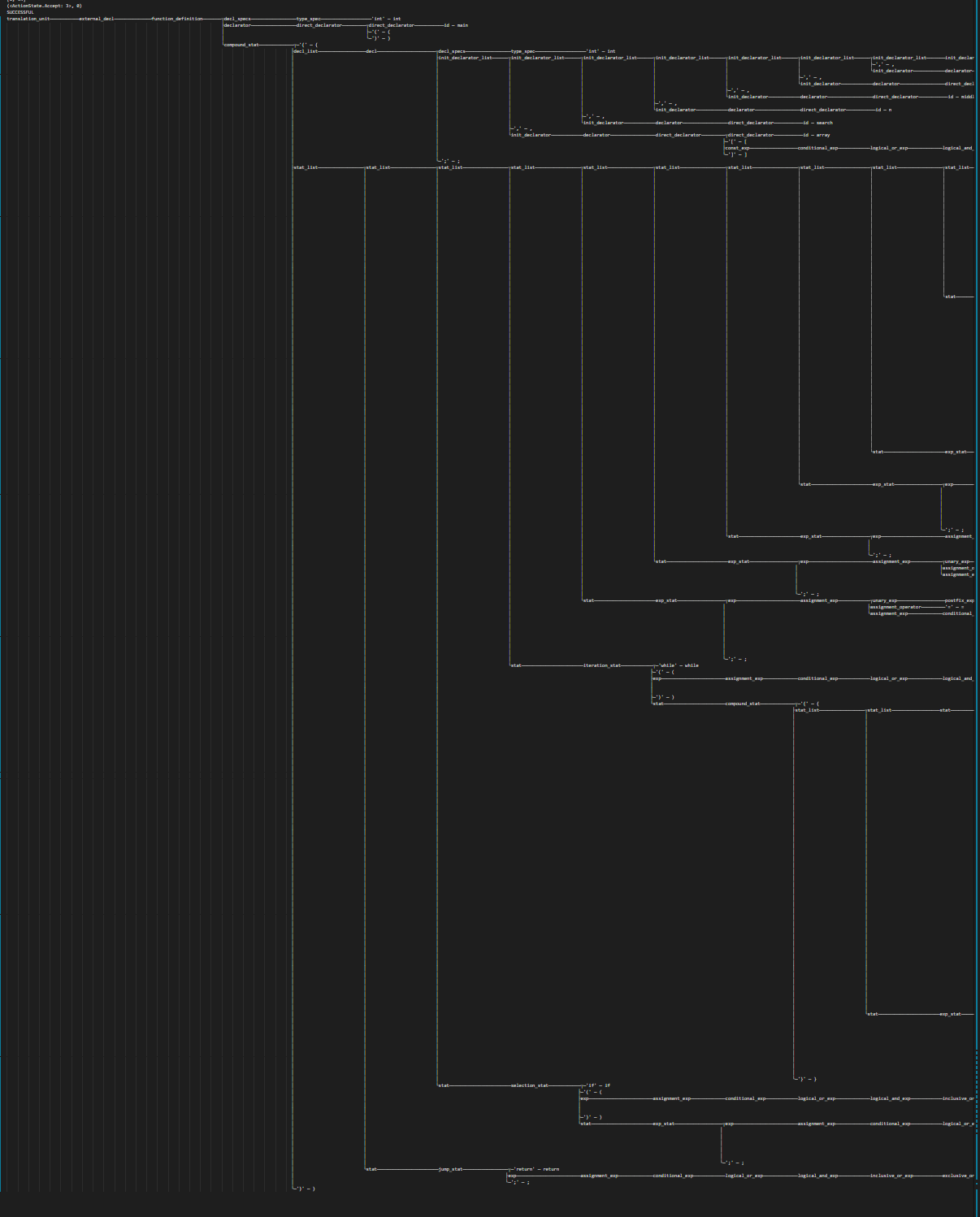
}

if (first > last)

printf("Not found! %d isn't present in the list.", search);

return 0;

}

测试结果：正确

* 1. **错误代码测试**

源代码：定义不符合规范

int main()

{

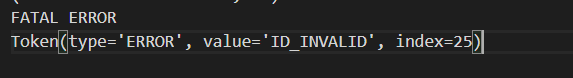
int 334a;

int 33qraf;

void www;

}

测试结果：正确



* 1. **错误代码测试**

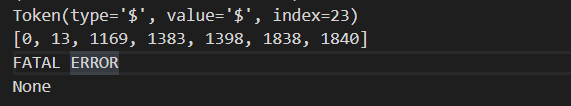
源代码：无最后的大括号

int main()

{

int a;

测试结果：正确



* 1. **模块调用设计问题与思考**

如果还有印象的话应该记得，在词法分析器中有个设计，我们将调用词法分析器的接口设计成了一个迭代器，并且期望这个迭代器在语法分析器中发挥其作用，但在实际的应用过程中，我们对这个迭代器的使用并未达到我们设计时的期望，主要是写代码的过程中未考虑清楚归约的情况，而迭代器每个值只能迭代一次，在哪里迭代，迭代器输出的内容该怎样保存，保存在何处，这些都是需要经过设计的，但在写语法分析器时，这些问题并未得到充分考虑，因此，这样一个迭代器最终被我们先变成了List，然后再使用，但我认为词法分析器作为子模块，迭代器的属性与其非常契合，因此需要重新编写的是语法分析器部分的代码，只要我们重复考虑迭代器的特殊性，并作出设计，就可以解决这一问题，只是因为时间的关系，重构计划将在完成课程设计的同时完成。

1. **总结与收获**

此次的任务是做出一个LR(1)语法分析器，在此过程中感觉自己收获颇丰，不仅锻炼了我们分析问题，写代码的能力，同时也从实践层面学习到了关于LR(1)文法的很多东西，尤其是LR(1)分析表的生成，此算法尤其花费时间，并且很难进行调试，此部分的代码是我们最后完成的部分，一方面是因为其实现较为复杂，另一方面是一个分析表的生成过程非常耗时，很难调试，在整个过程结束之前，我们都不能知晓此过程中是否发生了错误，甚至即使我们验证了几个源程序的正确性，依然不能确定整个分析表是完全正确的。

不过最终经过不断调试，我们终于完成了这个表现尚可的语法分析器，最大的收获就是彻底弄懂了语法分析器的工作原理，自己亲手实践过后，不管是理解还是记忆程度都大大加深了。

附件是源代码以及一些数据文件，源代码使用Python 3.7测试通过，具体使用方法请见文件中的Usage.md文件。