

汇编实验语法分析器实验报告

09017423 杨彬

一、实验目的

本实验我结合编译原理课程所学习语法分析的知识，基于 C++，结合上一个实验中的词法分析器，利用 LR(1) 分析方法，构造语法分析器。从而达到加深对编译原理语法分析的理解的目的。

二、内容简介

实验内容：

Input: Stream of characters

CFG(产生式)

Output: Syntax tree

Sequence of reductions if bottom-up syntax analyzing methods are used.

实验过程：

- 1、 本实验我设计了一个 输入是产生式，根据输入的产生计算 LR(1)分析方法的 状态图和分析表的系统：LR1Paser
- 2、 在 1 的基础上，设计自己的语法，语法所用到的关键字和词法分析器的实验 完全相同。
- 3、 将自己设计的语法的产生式送入 LR1Paser，LR1Paser 将会自动计算所需要的 分析表和状态图。得到 LR1Paser 的一个实例 myPaser
- 4、 将词法分析得到的 token 流送入 myPaser，myPaser 将扫描 token 流，根据计 算好的分析表生成语法树和归约过程的堆栈状态，若归约过程出现错误，报 语法错误。

三、 思路方法

(ps:事实上按照原本的实验要求，分析表是我们自己手动算出来的，但是出于兴趣，我设计了一个分析表和状态图生成系统 LR1Paser 能够做到输入{产生式、非终结符序列、终结符序列}得到输出{分析表、状态图}。但是这一部分并不是实验的重点，出于篇幅限制我仅做简单介绍，重点放在根据分析表生成语法树的部分)

LR1Paser 部分：为了设计一个可用的 LR1Paser，我的思路是必须先设计好 表示产生式 Production、状态 State、状态图 States、分析表 PaserTable 的数据结构，然后按照编译原理课程上所学习的

- 1、 设置 LR1Paser 的初始状态，并将产生式 $S1 \rightarrow S \{ "\$"\}$ ，设置为初态的 第一个产生式
- 2、 对于一个 LR1Paser 的 state，需要根据已经有的 production 计算出这个 state 的所有的 production，例如假设 0 状态的初始的 production 为 $S1 \rightarrow S \{ "\$"\}$ ，那么 state0 就能根据这个 production 计算剩下的 production
- 3、 算剩下的 production 的过程中又涉及到，计算相应非终结符的 first 集， 计算相应 production 的 follow 集。
- 4、 有了一个完整的 state 之后，我们需要根据这个 state 中的 production 自动推导出这个 state 能够产生的其它的 state 并判断是否是已有的 state
- 5、 不断地从已有的 state 生成新的 state，最终构造出状态图

- 6、 根据构造好的状态图, 遍历所有的状态, 根据每个状态下产生式的 follow 集, 一个状态到另一个状态转换关系 (类似于词法分析器中的状态图) 从状态图 states 生成分析表 ParserTable

总结: 构造出一个 LR1Paser 我需要做的是 计算 First 集, 计算 follow 集, 状态生长 (从已有的表达式推导出全部的表达式知道不能再推导位置), 从已知状态推导新状态、构造状态图、构造分析表。

从分析表生成语法树和推导过程部分:

这一部分是本次实验的重点。我的思路是,

- 1 先要设计语法分析表的数据结构 (和上一部分的语法分析表的结构相同), 基于 LR1Parser 生成的分析表, 我还需要解决相应的冲突 (因为语法可能存在二义性)。
- 2 其次要设计语法树的数据结构。
- 3 利用编译原理课程上学习的归约方法, 每次读入一个 token, 根据当前的状态和语法分析表中对应的操作, 进行状态的转移或者是归约操纵, 并更改相应的堆栈, 在更改堆栈的同时逐步创建语法树。在这一过程中如果发现读入 token 找不到相应的操作或者 token 序列全部读玩但是符号栈和状态栈还有内容时报语法错误。
- 4 在上述操作的过程中每一次状态转移或者归约都将当前堆栈状态输出到文本文件中。
- 5 将构造好的语法树输出到文件中。

四、假设定义

1、语法定义

本次实验, 所需要的词法和词法分析器实验 (<https://github.com/chonepieceyb/SEU-CS-LEARNING/tree/master/编译原理/实验/LexicalAnalyzer>)

所使用的词法完全相同, 同时我仿照我常用的编程语言设计了相应的语法。产生式如下:

```

1 #####产生式#####
2 (0) C --> if ( E ) then
3 (1) E --> EA E
4 (2) E --> EO E
5 (3) E --> not E
6 (4) E --> ( E )
7 (5) E --> E rop E
8 (6) E --> id
9 (7) E --> true
10 (8) E --> false
11 (9) E --> num
12 (10) E --> E + E
13 (11) E --> E - E
14 (12) E --> E * E
15 (13) E --> E / E
16 (14) EA --> E and
17 (15) EO --> E or
18 (16) S --> id = E ;
19 (17) S --> S S
20 (18) S --> T { S }
21 (19) S --> C { S }
22 (20) S --> WD { S }
23 (21) S1 --> S
24 (22) T --> C { S } else
25 (23) WD --> while ( E ) do
26 (24) rop --> >
27 (25) rop --> >=
28 (26) rop --> <
29 (27) rop --> <=
30 (28) rop --> ==
31 (29) rop --> !=

```

该语法涉及 赋值语句，条件语句，循环语句，以及一些常用的表达式，运算符的优先级同 C++（除了 ==和 !=优先级，在我的语法中 ==、!=，>，<，>=，<=优先级相同，但 C++不然）。同时该语法存在二义性，二义性主要由于，运算符的优先级和结合性 以及 ;的结合性没有体现在这套产生式中，因此我需要在生成分析表之后，手动根据相应的优先级和结合性消除二义性。

2 分析表构造说明

如前所述，本次实验的分析表我通过程序进行构造，该程序（LR1Paser）所需要的输入为 非终结符字符集，终结符字符集，产生式字符集，通过调用 LR1Paser 的 buildPaserGraph 函数，将自动生成状态图和分析表作为 LR1Paser 的成员变量。该 LR1Paser 具有如下特性：

- 1 支持存在自递归和左递归的文法（我在代码中有相应的检测函数）
- 2 支持存在空产生式的语法。
- 3 该 LR1Paser 还需要指定 文法开始符 begin（默认为” S1”），结束符 end（默认

为”\$”)。和空串符（必须为”none”）

4 LR1Paser 并不会消除二义性语法，需要我在分析表构造完成之后手动消除分析表中的冲突

5 由于在生成分析表之后，我并没有将分析表以文本的方式保存下来（这点不难），所以每次进行语法分析都需要重新计算分析表，较为大型的语法（比如上述产生式）运算速度可能较慢。

3 实验输入输出说明

在本实验的源代码中还有词法分析器的源代码。本实验的输入和词法分析器实验的输入一样，都是一个内容为代码片段的文本文件，为了体现完整性，在 main 函数中先由词法分析器读入文本文件生成 token 序列（vector<token> tokenstream），在将 tokenstream 作为构造好的 LR1Paser 对象 myPaser 的 scanTokenStream 函数的参数，该函数扫描 token 序列，将每一步的推导的堆栈状态以及最终的语法树输出到文本文件中。

4 实验附带的文本文件说明

本次实验的文本文件如下：

input.txt :输入的代码片段

token.txt :词法分析产生的 token 序列

model.txt :语法分析所使用的分析模型，包括所有的产生式，状态图，和分析表。也就是 LR1Paser 的计算结果（ps 状态图排列顺序并不是按 stateID 排列，而是按产生式的多少进行排序）

model.txt 示例：

```
1 #####产生式#####
2 (0) C --> if ( E ) then
3 (1) E --> EA E
4 (2) E --> EO E
5 (3) E --> not E
6 (4) E --> ( E )
7 (5) E --> E rop E
8 (6) E --> id
9 (7) E --> true
10 (8) E --> false

33 #####状态图#####
34 state: 48
35 C --> .if ( E ) then { {}
36 S --> .id = E ; { $,id,if,while}
37 S --> .S S { $,id,if,while}
38 S --> S .S { $,id,if,while}
39 S --> S S .{ $,id,if,while}
40 S --> .T { S } { $,id,if,while}
41 S --> .C { S } { $,id,if,while}
42 S --> .WD { S } { $,id,if,while}
43 T --> .C { S } else { {}
44 WD --> .while ( E ) do { {}
45 input:if --> state3
46 input:WD --> state2
47 input:while --> state1
48 input:id --> state4
49 input:S --> state48
50 input:C --> state5
51 input:T --> state7
```

1721	*****分析表*****	!	\$	()	*	+	-	/	%	<	<=	=	==	>	>=	and
1722	0																
1723	1			S47													
1724	2																
1725	3			S8													
1726	4																
1727	5												S43				
1728	6																
1729	7		R21														
1730	8			S17													
1731	9	R8			R8	R8	R8	R8	R8		R8	R8		R8	R8	R8	R8
1732	10	R6			R6	R6	R6	R6	R6		R6	R6		R6	R6	R6	R6
1733	11	S31			S18	S23	S20	S22	S25		S28	S29		S30	S26	S27	S24
1734	12	R9			R9	R9	R9	R9	R9		R9	R9		R9	R9	R9	R9
1735	13			S17													
1736	14	R7			R7	R7	R7	R7	R7		R7	R7		R7	R7	R7	R7
1737	15			S17													
1738	16			S17													
1739																	

report.txt: 对 input.txt 进行语法分析得到的推导步骤和最终的语法树。

(ps:所有的文本文件最好使用 notepad++、VSCode 或者其它功能较强的文本编辑器打开查看, 如果直接用 windows 的 txt 打开将不会有排版)

五、状态图

如前所述, 本实验利用程序生成状态图和分析表, 由于本实验的语法规模较大, LR(1) 状态图一共有 101 个状态, 分析表也十分巨大, 该语法的状态图和分析表都保存在附件中的 model.txt 中了。请详细查看四中对相关内容的说明, 在 model.txt 下查看。或者查看

(<https://github.com/chonepieceyb/SEU-CS-LEARNING/tree/master/%E7%BC%96%E8%AF%91%E5%8E%9F%E7%90%86/%E5%AE%9E%E9%AA%8C/SyntaxAnalysis>)下的 model.txt。

当然或许会有同学怀疑 LR1Paser 生成的分析表是否正确, 在这里我用一个比较小型的文法为例子, 构造 LR1 分析表 and 状态图。

LR1Paser demo:

```

1 #####产生式#####
2 (0) A --> d
3 (1) B --> d
4 (2) S --> A a
5 (3) S --> b A c
6 (4) S --> B c
7 (5) S --> b B a
8 (6) S --> S S
9 (7) S --> none
10 (8) S1 --> S

```

```

47 state: 0
48 A --> .d { a}
49 B --> .d { c}
50 S --> .A a { $,a,b,d}
51 S --> .b A c { $,a,b,d}
52 S --> .B c { $,a,b,d}
53 S --> .b B a { $,a,b,d}
54 S --> .S S { $,a,b,d}
55 S --> none .{ $,a,b,d}
56 S1 --> .S { $}
57 input:d --> state1
58 input:A --> state2
59 input:b --> state4
60 input:B --> state3
61 input:S --> state5

```

```

63 state: 1
64 A --> d .{ a}
65 B --> d .{ c}

```

```

80 state: 2
81 S --> A .a { $,a,b,d}
82 input:a --> state9

```

```

94 state: 3
95 S --> B .c { $,a,b,d}
96 input:c --> state11

67 state: 4
68 A --> .d { c}
69 B --> .d { a}
70 S --> b .A c { $,a,b,d}
71 S --> b .B a { $,a,b,d}
72 input:d --> state6
73 input:A --> state7
74 input:B --> state8

30 state: 5
31 A --> .d { a}
32 B --> .d { c}
33 S --> .A a { $,a,b,d}
34 S --> .b A c { $,a,b,d}
35 S --> .B c { $,a,b,d}
36 S --> .b B a { $,a,b,d}
37 S --> .S S { $,a,b,d}
38 S --> S .S { $,a,b,d}
39 S --> none .{ $,a,b,d}
40 S1 --> S .{ $}
41 input:d --> state1
42 input:A --> state2
43 input:b --> state4
44 input:B --> state3
45 input:S --> state13

76 state: 6
77 A --> d .{ c}
78 B --> d .{ a}

87 state: 7
88 S --> b A .c { $,a,b,d}
89 input:c --> state10

state: 8
S --> b B .a { $,a,b,d}
input:a --> state12

84 state: 9
85 S --> A a .{ $,a,b,d}

state: 10
S --> b A c .{ $,a,b,d}

state: 11
S --> B c .{ $,a,b,d}

state: 12
S --> b B a .{ $,a,b,d}

state: 13
A --> .d { a}
B --> .d { c}
S --> .A a { $,a,b,d}
S --> .b A c { $,a,b,d}
S --> .B c { $,a,b,d}
S --> .b B a { $,a,b,d}
S --> .S S { $,a,b,d}
S --> S .S { $,a,b,d}
S --> S S .{ $,a,b,d}
S --> none .{ $,a,b,d}
input:d --> state1
input:A --> state2
input:b --> state4
input:B --> state3
input:S --> state13

```

#####分析表#####

	\$	a	b	c	d	A	B	S
0	R7	R7	R7		R7	S2	S3	S5
1		R0	S4	R1	S1			
2		S9						
3				S11				
4					S6	S7	S8	
5	R7	R7	R7		R7	S2	S3	S13
6	R8		S4		S1			
7		R1		R0				
8		S12		S10				
9	R2	R2	R2		R2			
10	R3	R3	R3		R3			
11	R4	R4	R4		R4			
12	R5	R5	R5		R5			
13	R6	R6	R6		R6	S2	S3	S13
	R7	R7	R7		R7			
			S4		S1			

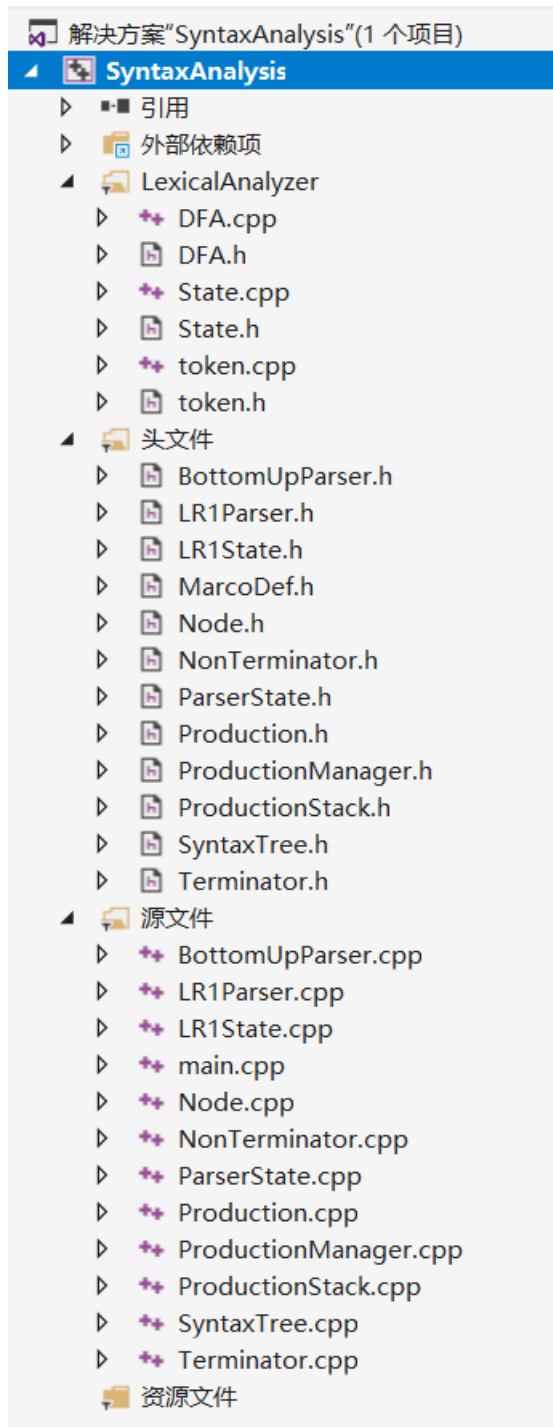
这个 demo 很好地体现了 LR1 的特性，支持产生式中有左递归，支持空产生式（在 none 产生式下直接对 follow 集进行归约），无法解决文法冲突（所以需要手动排除冲突）。可以比较容易地验证上述推导是正确的。本实验所定义的文法的相关文件还请在

model.txt 中查看。

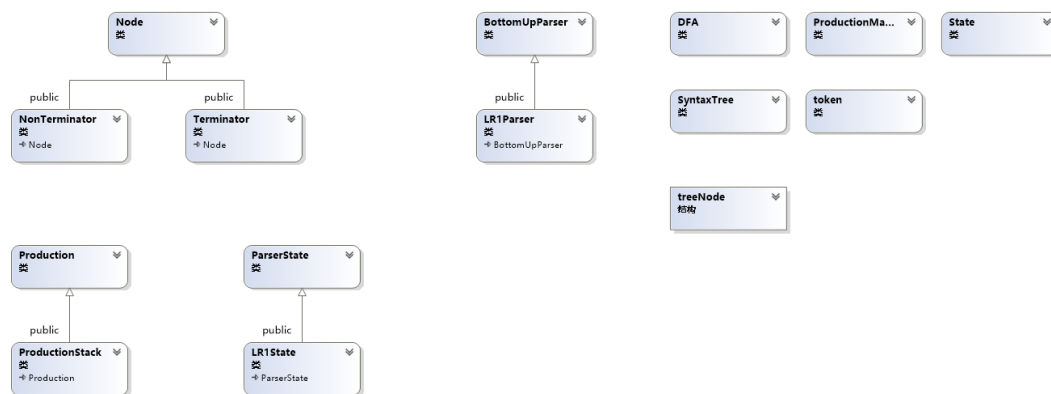
六、数据结构

本实验分为两个部分：LR1Paser 和根据状态分析表来进行语法分析。第一个部分内容很多，在这里我仅给出构建 LR1Paser 所用的类图。本节的重点放在第二部分。如果对第一部分感兴趣还请查看源代码（文末），代码里有较为详细的注释。

代码结构如下：



LR1Paser 部分：



LR1Parser 的 Node 类是用来存放节点，终结符和非终结符都是继承 Node 类，其中非终结符有一个数据结构 `vector<vector<Node*> productions>`，这个数据结构的第一层表明其一个非终结符可能有多个产生式，第二层 `vector<Node*>` 就是非终结符下的产生式的数据结构，其按照产生式的顺序，保存了指向产生式中的节点的指针。Node 有一个函数 `calFirstSet` 用来计算 first 集合。

ProductionStack 类是对 NonTerminator 类所存储的产生式关系的另外一种表示，其类成员为：`NonTerminator* head`，指向产生式头的指针，还有一个 `production_id`，表示在该 head 下的第几个产生式。还有一个成员变量 `stackState`，用来表示 LR1 分析方法中的堆栈状态（即小点点）

ProductionManager 类的定位类似于工厂类，在这个类里保存有一系列 Node 类的实体。通过这个类管理产生式，这个类还负责产生 production 对象。

LR1State 类，这个类里保存有一系列 productionStack 对象，这个类里有一个 `growState` 类函数，该函数能够根据已有的 productions 推导出所有的 productions 直到不能再推导了为止。

LR1Parser 类：这个类就是该 LR1 分析表生成系统的主类，这个类里有一个 ProductionManager 对象用来管理所有的节点和 production，这个类保存所有的 state (LR1State 类对象)，拥有 `buildParserGraph()` 构造状态图函数，和 `buildParserTable` 构造分析表函数。

类继承关系，主要是为了可扩展，比如通过继承将 LR1 扩展为 LALR 或者 SLR 等。

分析过程和生成语法树部分：

为了生成语法树，我需要定义语法树的数据结构。其数据结构如下（下面出现的 token 和语法分析器中定义的 token 完全相同）

1 语法树节点 treeNode 结构体

```

struct treeNode {
    treeNode(token& t) :t(t)
    {}

    treeNode(const treeNode& tn)
    { this->t = tn.t; this->width = tn.width; this->child_ptrs = tn.child_ptrs; }

    treeNode()
    {
        t = token(-1, "none", "_"); child_ptrs = vector<int>();
    };
    token t;
}
  
```



```

    int width;
    vector<int> child_ptrs;
};

```

这里的 width 是打印语法树所需要的属性, child_ptrs 是孩子节点的指针, 语法树是一棵多叉树, 所以所有的节点保存在一个 vector 里, 这里的采用静态数组的方法, 其指针就是该节点在 vector 中的索引。t 就是该节点所保存的 token。

2 语法树类

```

class SyntaxTree
{
public:
    SyntaxTree();
    ~SyntaxTree();
    int addNode(treeNode& t);
    void outputTreeAsFile(ofstream& fout, int widthset = 8, int heightset=4) {
        if (isEmptyTree()) {
            return;
        }
        getTreeWidth(head); //计算宽度并保存下来
        outputTreeAsFile(fout, head, widthset, heightset);
    }
    bool isEmptyTree() {
        if (nodes.size() > 0) {
            return false;
        }
        else {
            return true;
        }
    }
private:
    vector<treeNode> nodes; //存放节点的实际区域
    int head; // head 指向 nodes数组的最后一个索引
    void outputTreeAsFile(ofstream& fout, int subTree, int widthset = 10, int heightset = 4);
    int getTreeWidth(int subTree);
    int getTreeDepth(int subTree); //获取深度
};

```

这个类有一个成员变量 nodes, nodes 保存这颗语法树所有的节点, head 是语法树的根节点的指针(静态数组), head 固定指向 nodes 数组的最后一个元素, getTreeWidth, 和 getTreeDepth 是获取子树 subTree 的宽度和深度的函数采用递归的方法计算, 是打印语法树所需要的。outputTreeAsFile 是将语法树打印到文件中的函数。

而 addNode 类对于构建语法树来说十分重要, 其函数定义如下:

```

int SyntaxTree::addNode(treeNode& t) {
    nodes.push_back(t);
    head = nodes.size() - 1; // 默认指向最后一个位置, 适用于递归建树
}

```

```
return head;           // 返回指向刚刚加入的node的 位置，方便建树
}
```

这一个函数，接受一个 treeNode 的引用作为阐述，将这个 treeNode 加入到 nodes 中，同时返回加入后该 node 在 vector 中的索引，同时将树的跟自动指向新加入的 node。这个函数极大地方便自底向上构建语法树。返回指针使得外部的父节点能够得到孩子节点的指针，方便设置父节点的 childs 数组，自动将 head 设置为新加入的 node 意味着，当最后一个加入的节点是根节点时，head 将自动指向根节点。详细构建过程在算法部分描述。

3 分析表

选择合适的数据结构来保存分析表是十分重要的，在这里我采用的数据结构是：

```
vector<vector<vector<std::pair<string, int>>>> parserTable
```

前两层 vector 构成了一个二维数组，这个二维数据的行代表状态 state，列代表读入的 token 的 token_name 的 id。这个二维数组的单元内容是一系列的操作 vector<std::pair<string, int>>，因为分析表可能存在冲突，所以必须用一个 vector，这个 vector 就代表操作可能有多个，如果没有操作其 size 为 0。操作容器内存放着一系列的操作，表示操作的数据结构是 pair(pair 是 std 内定义的数据结构)，其本质上是一个结构体，该结构体有两个成员变量 first，和 second。在这里 pair.first 是一个 string，存放着“S”，或者“R”，S/R 代表操作的类别，“S”表示使用产生式，“R”表示使用归约式。pair.second 是一个 int，int 表示操作的内容。如果是“S”，int 的值表示转移操作的下一个 state 的 state_id，如果是“R”，int 归约操作所使用的产生式的 ID。

除此之外，为了操作方便我还定义了一个哈希表，unordered_map<string, int> tableItems_map，这个哈希表将每一个 token_name 和上述 vector 的列号进行映射，当读一个 token 时，可以根据这个 token 的 token_name 查找 tableItems_map 获得列号，再根据状态和列号从 parserTable 查找相关的操作。至于为什么不直接将 vector 的第二维定义为哈希表，是因为我希望保存的 parserTable 行列是有序的，而哈希表我并不能知道其确切的排序。

4 操作栈

正如我在编译原理课程学习的内容，使用 LR(1) 分析方法进行语法分析我需要 3 个堆栈。

```
状态堆 vector<int> state_stack
```

```
符号栈 vector<string> string_stack
```

```
输入栈 vector<token> tokenStream
```

这三个堆栈在语法分析中起到重要的作用。

5 产生式类

这里的产生式类和 LR1Paser 中用到的产生式类相同，其数据结构可以简单概括(只在这一部分用到的，进行了简化，代码中并非如此)为

```
{
    String head;
    Int productionSize;
}
```

Head 代表 产生式的头的 name， productionSize 是产生式大小，eg：对于产生式 $S \rightarrow Abc$ 来说 其 head 为 S，size 为 3。

七、核心算法

1 根据 LR(1) 分析表，读取 token 序列，进行相关转台转移和归约操作，同时在这一个

过程我还需要这是本实验的核心的算法，其算法流程如下：（该算法实现在 LR1Paser 类的成员函数:scanTokenStream 中

input: tokenStream , ouput:将推导过程和语法树报错到文件中

```
(1)  state_stack = {0}
(2)  string_stack={}$}
(3)  vector<int> tree_poses ={}      //用来生成语法树，存放孩子节点的指针
(4)  vector<TreeNode> tree_stack ={}  //语法树节点栈
(5)  SyntaxTree tree = 空树
(6)  将初始堆栈状态输出到文件
(7)  token t
(8)  while 遍历 tokenStream 的内容，当前 token 为 t，还没有遍历完 do
(9)      取当前 state_stack 栈顶状态 s
(10)     根据 t 和 s 查 parserTable，得到 operations    // operations 的数据
    结构是 vector<pair<string,int>>
(11)     if operations.size() ==0 then
(12)         没有找到在该状态下，读入 t 的操作。报语法错误,RETERN
(13)     if operations.size() >1 then
(14)         分析表存在冲突，冲突没有解决，报分析表存在二义性。RETERN
(15)     operation = operations[0]    //只有一个操作
(16)     if 如果是 S 操作 then:
(17)         state_stack.push_back(operation.second) //将操作的 S 操作的目标
    状态压入状态栈（eg S14, 压入 14）
(18)         string_stack.push_back( t.token_name)
(19)         TreeNode(t) tn
(20)         if treePos 不为空 then :    //说明刚才进行了归约操作
(21)             将 treePos 内孩子节点的指针，加入到父节点 tn 的 childs 中
    //建立父节点和孩子节点的关系
(22)             tree_stack.push_back( TreeNode(t) )    //根据 token 建立树节点
    并入栈）
(23)             完成一次操作，将堆栈状态输出到文件中
(24)         Elseif 如果是 R 操作 then:
(25)             将当前栈内容输出到文件中
(26)             production = getProductionById(operation.second)    //根据
    id 获取产生式
(27)             for 做 production.size 次循环 :    //这一步要进行归约，根据产生
    式的大小把堆栈内相应内容弹出
(28)                 state_stack.pop
(29)                 string_stack.pop
(30)                 tree_poses.push(
                    tree.addNode(tree_stack.pop))    //将 treeStack 的内容出
    栈，同时将其加入到语法树种，将加入到语法树后返回的该节点的指针加入到树节点指
    针栈中，这一步是为了自底向上构建语法树。
(31)             t = token(production.name)    //用产生式的 head 作为当前输入
    的 token，这几部是归约操作，实质上就是将栈内的产生式右边的部分用产生式
```

左边的部分代替

- (32) if t 是文法开始符号 (eg S1) :
- (33) 将当前栈状态输出到文件中
- (34) 归约成功
- (35) Else goto (9), 保持 tokenstream 的当前 token 仍然是在计入 R 操作时候的 token, 这点很重要。

2 有了语法树之后, 我需要将语法树打印到文件中, 由于打印的过程比较繁琐, 仅仅是为了显示方便, 和本文的核心内容没有必要的联系, 这里只给出核心思路:

对语法数进行深度遍历, 逐行打印语法树, 利用队列辅助语法树的打印。

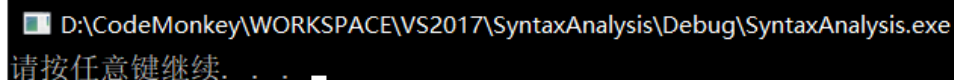
八、测试用例

正确用例:

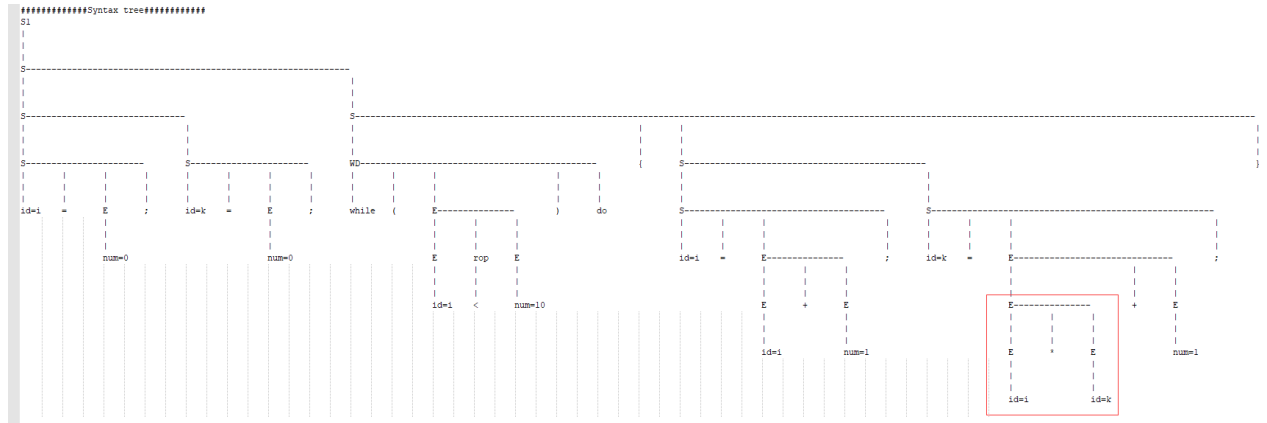
```
1  i = 0 ;
2  k = 0 ;
3  while ( i < 10 ) do {
4      i = i + 1 ;
5      k = i * k + 1 ;
6  }
```

Input:

Output:



```
314 -----step73-----
315 state stack: { 0,6,2,46,55,81 }
316 string stack: { $,S,WD,{,S,} }
317 input token: (-1,$,_)
318 -----step74-----
319 state stack: { 0,6,2,46,55,81 }
320 string stack: { $,S,WD,{,S,} }
321 input token: (-1,$,_)
322 R20: S --> WD { S }
323 -----step75-----
324 state stack: { 0,6,48 }
325 string stack: { $,S,S }
326 input token: (-1,$,_)
327 -----step76-----
328 state stack: { 0,6,48 }
329 string stack: { $,S,S }
330 input token: (-1,$,_)
331 R17: S --> S S
332 -----step77-----
333 state stack: { 0,6 }
334 string stack: { $,S }
335 input token: (-1,$,_)
336 -----step78-----
337 state stack: { 0,6 }
338 string stack: { $,S }
339 input token: (-1,$,_)
340 R21: S1 --> S
341 sucess!
342 #####Syntax tree#####
```



分析成功，为了展示这个用例比较小，但同时也体现了语句、表达是、+号和*号的优先级等特点，从语法树可以很容易看到归约是正确的，推导步骤过程太多这里只给出部分推导过程。

正确用例 2:

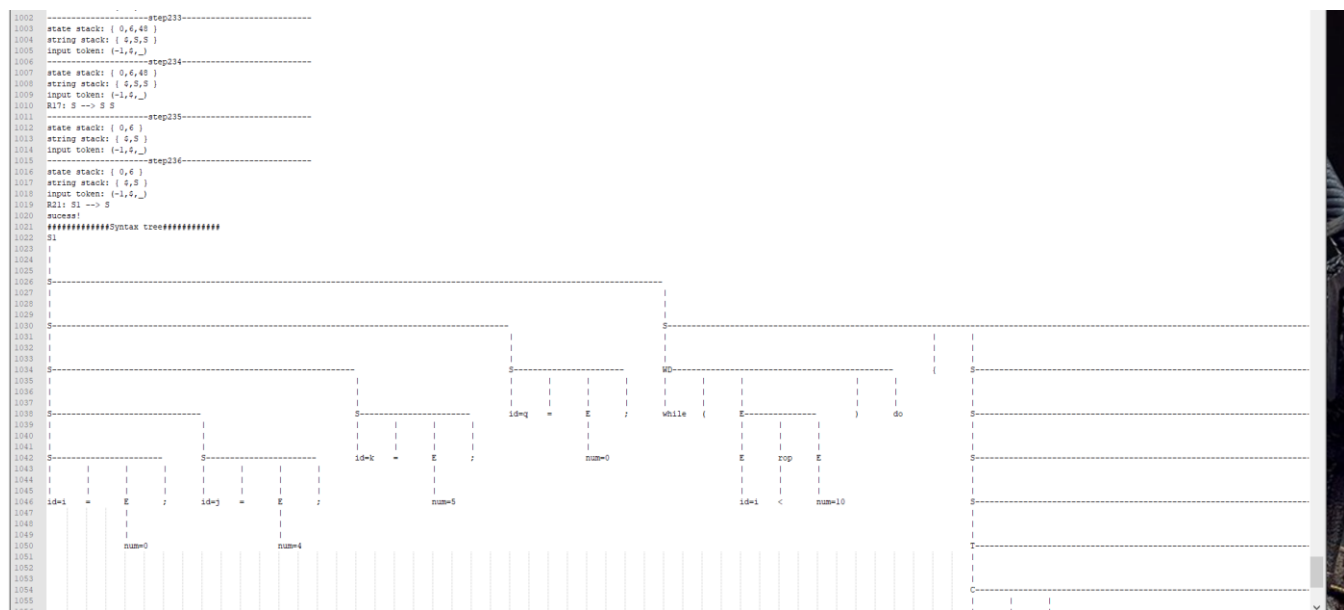
```

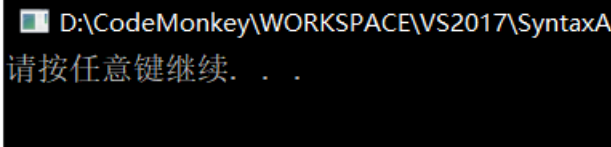
1  i = 0 ;
2  j = 4 ;
3  k = 5 ;
4  q = 0 ;
5  while ( i < 10 ) do {
6      if ( ( i > 5 ) and ( k > j ) ) then {
7          k = k - 1 ;
8      }
9      else {
10         k = k + 1 + k * j ;
11     }
12     if ( k == i ) then {
13         k = j ;
14     }
15     i = i + 1 ;
16     while ( q < 5 ) do {
17         j = j - 1 ;
18     }
19 }

```

Input:

Output:





这个例子里比较完整地覆盖了我所定义的语法，同样也分析成功了，由于分析步骤比较多，所以详细结果请查看附件里的 report.txt，或 <https://github.com/chonepieceyb/SEU-CS-LEARNING/tree/master/%E7%BC%96%E8%AF%91%E5%8E%9F%E7%90%86/%E5%AE%9E%E9%AA%8C/SyntaxAnalysis下的report.txt>

错误用例:

```
1  i = 0 ;
2  k = 0 ;
3  while ( i = 10 ) do {
4      i = i + 1 ;
5      k = i * k + 1 ;
6  }
```

Input:

Output:

```
Syntax error! can not generate Syntax treestate stack: { 0,6,1,47,10 }
state stack: { $,S,while,(,id }
input token: (15,;,_) , (0,id,k) , (16,=,_) , (0,id,i) , (13,*,_) , (0,id,k) , (11,+,_) , (1,num,1) , (15,;,_) , (31,},_) , (-1,$,_)
请按任意键继续. . .
```

[illegible]

可以看到输入里的 `i = 10` 明显错误的，根据语法这是一个语句，而 `while` 里需要的是一个表达式，这里准确地在这里报错了。

九、出现的问题和解决方案

本实验我做了相当多的工作，很多经历花在构造 LR1Paser 上了，虽然这部分属于额外内容，但是对我更加深入地理解整个语法分析的体系确有很大的帮助。实验的过程中遇到的主要也在于构造 LR1Paser 上。有几个点我认为很有意义

1 我前面提到，我设计的 LR1Paser 是支持左递归文法的。但是，事实上一开始我在这里遇到了很致命的问题。根据我前面提到的数据结构，我计算 first 集，是通过递归进行计算的，如果文法中存在直接或者间接左递归，那么就会导致因为无限递归而报错。我的解决方案是，在递归中加入一个调用栈，这是一个 set，保存整个调用过程的调用者。每次进入这个函数，我首先根据这个调用栈判断是否构成回路，如果构成回路就先跳过。通过这种方案解决了这个问题

2 其次在 LR(1) 文法的状态扩展也很容易出错, 比如我需要从 $S_1 \rightarrow S$, 这个最开始的产生式计算整个状态的所有产生式。判断计算完成的条件必须是在两次计算之后状态里的产生式没有发生任何变化。一旦在某一步的扩展中, 改变了状态我就必须再从头计算以边以保证已经完全计算完成。一开始我只是用简单的 for 循环和给每个产生式加上标记, 导致了最后产生式的 follow 集计算出现问题。

推导部分:

在得到分析表的基础上进行语法分析, 我在一个地方出错了, 就是归约操作之后, 我们需要用产生式的左边来代替右边, 然后把堆栈里的产生式右边的符号弹出, 同时归约操作的产生式的左边作为新的输入, 进行下一步的判断。但是在实际操作中, 我一开始没有注意这个问题, 导致堆栈状态出错。事实上这个问题也是我们在做题的时候比较容易出错的地方。

十、实验感悟

相比于上一个实验, 这个实验里我超额地完成了从产生式用代码构造 LR1 分析表的部分。这个部分花费我很多经历, 虽然如此, 但能成功地、高质量地完成也让我很有成就感。同时这个过程也很好地提升了我的编程能力, 我用了很多学过的设计模式, 变成技巧来完成这个项目。同时这个实验让我发现很多没注意到的细节点, 这些细节点又十分重要。总的来说这个实验让我更加深入地理解了语法分析的过程, 同时我完成了将词法分析和语法分析串联起来的任务, 也深入了对编译原理这门课程的理解, 可谓是收获良多。

Github 源代码 (这是个人的代码仓库) : <https://github.com/chonepieceyb/SEU-CS-LEARNING/tree/master/%E7%BC%96%E8%AF%91%E5%8E%9F%E7%90%86/%E5%AE%9E%E9%AA%8C/SyntaxAnalysis>