**编译原理文法分析实验报告**

*石若非 141250108*

*注：项目的src文件夹下，SyntaxParser为单纯的文法分析类，而lex\_syntax文件夹里则是与第一次实验词法分析的结合程序，单纯的文法分析的测试用例见文档的测试a-c，和词法分析结合的文法分析测试用例见文档的测试d*

1. Motivation/Aim

本次实验的目的是在词法分析完成的基础上，对上下文无关文法分析过程进行模拟。实验选用的编程环境是java语言，采用了自底向上的分析方法，利用LR(0)或LR(1)的状态转换过程，输出文法分析的移入-归约过程。

1. Content description

本文档描述了在本次文法分析实验中采用的逻辑方法与数学算法，对一些关键数据结构和算法进行具体描述，并整理了实验过程中遇到的问题与难点。

1. Ideas/Methods

本次实验选择了自底向上的分析方法，与对应的自顶向下的分析模式相比，该方法虽然过程较为繁琐，但在分析过程与移入-归约序列的展示上更加直观，同时可以适应各种类型的上下文无关文法，因此我选择其作为完成本次实验的算法。

具体过程：

* 增加1条形如S’ -> S的新的文法，将原文法集拓展为其增广形式
* 建立LR(0)或LR(1)的DFA状态转换图
* 根据状态转换图建立SLR或规范LR分析表
* 输入源程序、上下文无关文法集、LR分析表
* 程序采用表驱动的形式，依次读取源程序的词素，根据当前状态和当前词素找到对应的Action（s，a）或Goto（t，A）的跳转动作，并输出对应的状态栈、符号栈、输入序列和动作内容，直到在源程序结尾成功Accept为止
* 若出现错误词素或文法不接受的序列，则停止循环并报错

1. Assumptions

编程语言：java

运行环境：jdk1.8版本

* 本程序不支持从CFGs到LR分析表的转换过程，需要提前手动进行分析与转换
* 输的CFGs按行输入不同产生式，同时每行用“,”分隔，前半为产生式字符串，后半为该产生式的符号数量
* 输入的LR分析表必须正确且与输入的CFG集相对应，输入的源程序所使用的字符集在输入的CFG集中
* 输入的LR分析表第一行为表头，以制表符\t分隔，同时代表Action的字符以输入终止符$结束，即$以左为Action，$以右为Goto
* 输入的LR分析表表格内容一行中用制表符\t分隔，同时所有空的地方以0填充，以便程序读入时可以正常处理
* 输入的源程序必须在末尾加上结束符$
* 本程序可对满足以上条件的任意CFGs进行分析

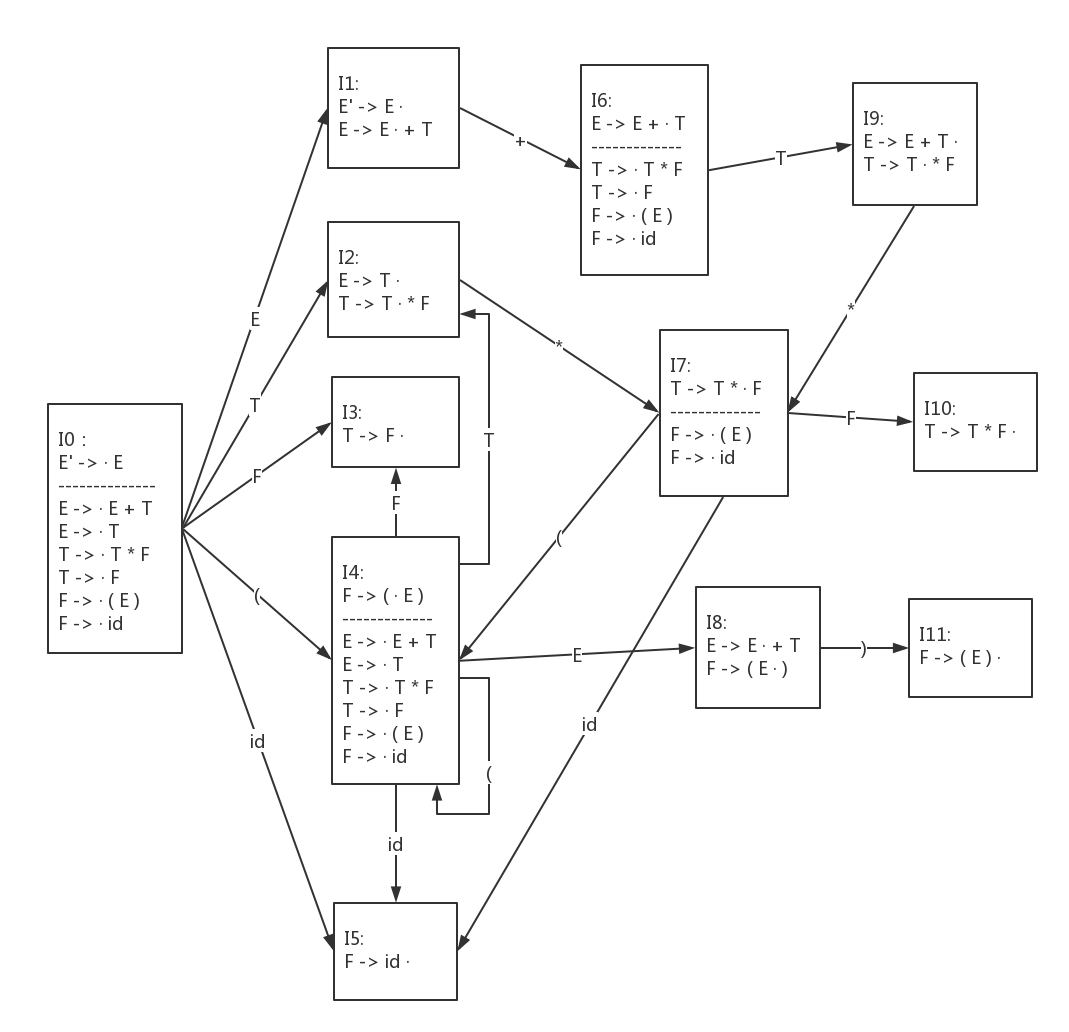
1. Related FA descriptions

因为本程序的输入并不局限于某些特定CFGs，这里以书上的+\*混合运算为例：

CFGs：

1. E -> E **+** T
2. E -> T
3. T -> T **\*** F
4. T -> F
5. F -> ( E )
6. F -> **id**

LR（0）状态转换图：



SLR状态转换分析表：

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| State | Action | | | | | | Goto | | |
| id | ( | ) | + | \* | $ | E | T | F |
| 0 | S5 | S4 |  |  |  |  | 1 | 2 | 3 |
| 1 |  |  |  | S6 |  | acc |  |  |  |
| 2 |  |  | R2 | R2 | S7 | R2 |  |  |  |
| 3 |  |  | R4 | R4 | R4 | R4 |  |  |  |
| 4 | S5 | S4 |  |  |  |  | 8 | 2 | 3 |
| 5 |  |  | R6 | R6 | R6 | R6 |  |  |  |
| 6 | S5 | S4 |  |  |  |  |  | 9 | 3 |
| 7 | S5 | S4 |  |  |  |  |  |  | 10 |
| 8 |  |  | S11 | S6 |  |  |  |  |  |
| 9 |  |  | R1 | R1 | S7 | R1 |  |  |  |
| 10 |  |  | R3 | R3 | R3r | R3 |  |  |  |
| 11 |  |  | R5 | R5 | R5 | R5 |  |  |  |

1. Description of important Data Structure

* HashMap<String, Integer> action\_map

利用哈希map建立每个action动作输入符号与分析表的序号的映射，以便表驱动时进行查找

* HashMap<String, Integer> goto\_map

利用哈希map建立每个goto动作归约符号与分析表的序号的映射，以便表驱动时进行查找

* ArrayList<String> CFGs ArrayList<Integer> CFGs\_symbol\_count

CFGs用于按顺序记录输入的每一条上下文无关文发，而CFG\_symbol\_count则用来按顺序记录每一条上下文无关文法的产生式的符号个数，用于执行出栈操作时的计数工作

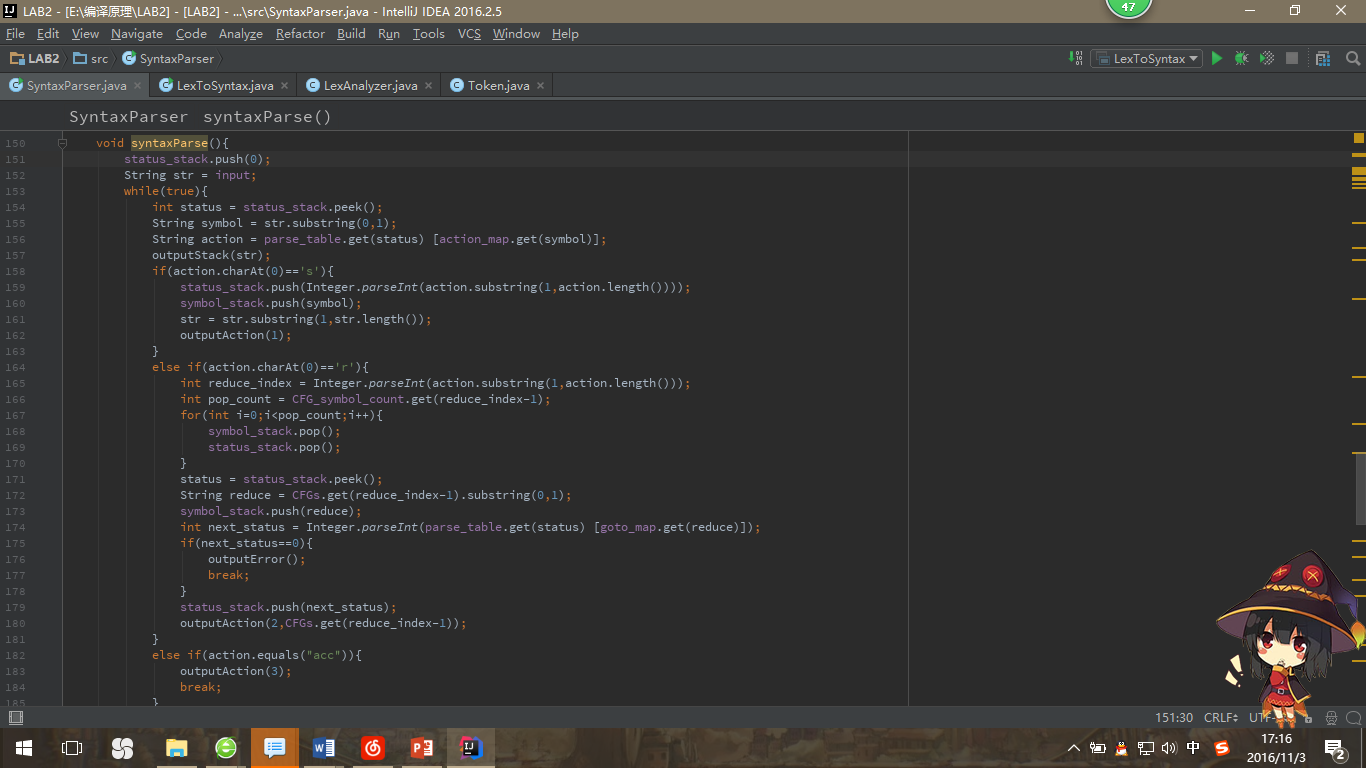
* ArrayList<String[]> parse\_table

parse\_table用于记录输入的LR分析表，每行为一个ArrayList的元素，由一个String数组组成。宏观上为一个二维数组，是本程序进行表驱动的基础

* Stack<Integer> status\_stack Stack<String> symbol\_stack

状态栈与符号栈，用于记录移入-归约的过程，同时简化对每一步过程的输出

1. Description of core Algorithms



方法syntaxParse()

该方法是对书上图4-36的语法分析算法的具体实现（有部分是额外的输出操作）

条件分支采用了表驱动的实现方法，大大简化了代码结构

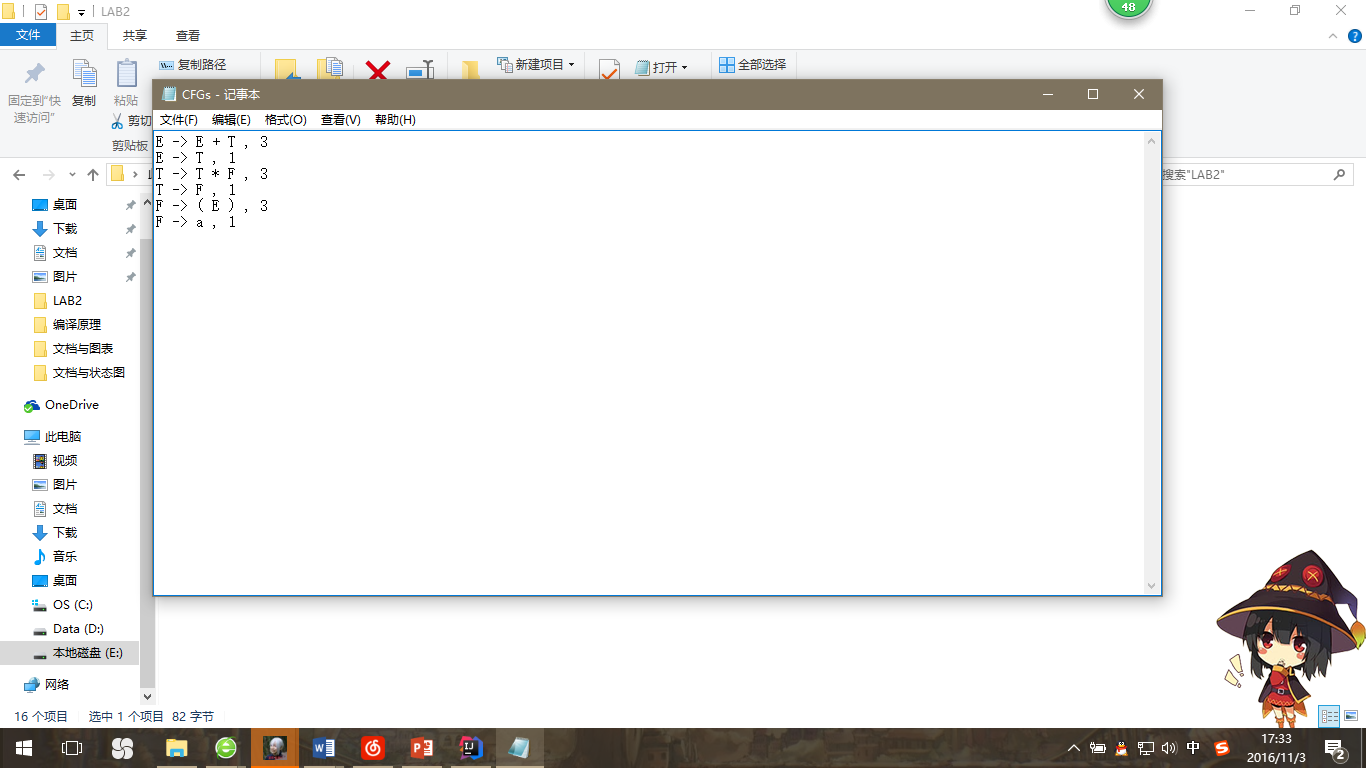
表驱动查找运算为: parse\_table.get(status) [action\_map.get(symbol)]

步骤如下：

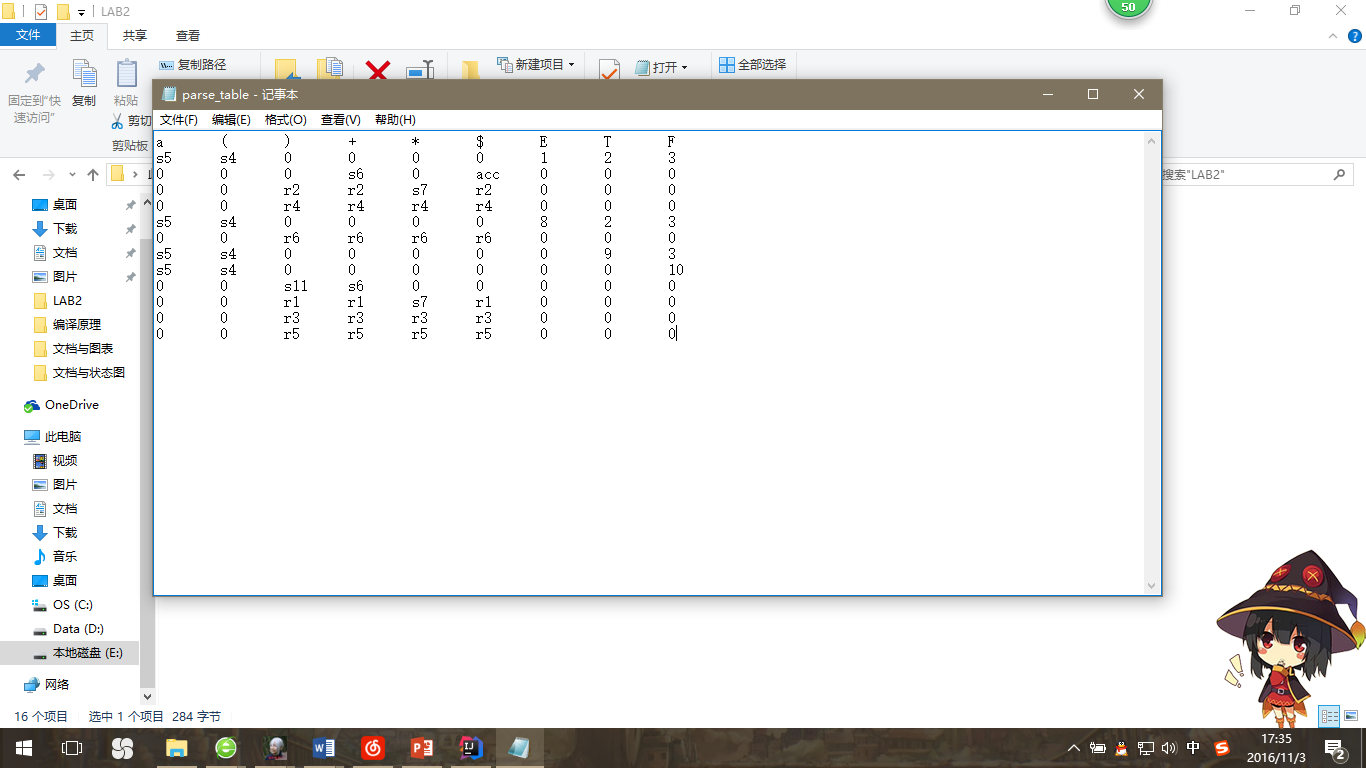
* 每次循环读入下一个符号，从status\_stack栈顶读取当前状态
* 若该符号在action\_map中，则根据status和该符合在map中映射的值查找跳转应执行的操作，若不在map中则报错
* 若操作对应移入操作，则将符号压入symbol\_stack，跳转的状态值压入status\_stack
* 若操作对应归约操作，则将symbol\_stack和status\_stack弹出对应的归约文法的符号个数的元素，再将归约为的符号压入symbol\_stack
* 若操作对应结束，则跳出循环

1. Use cases on running
2. 测试用例a

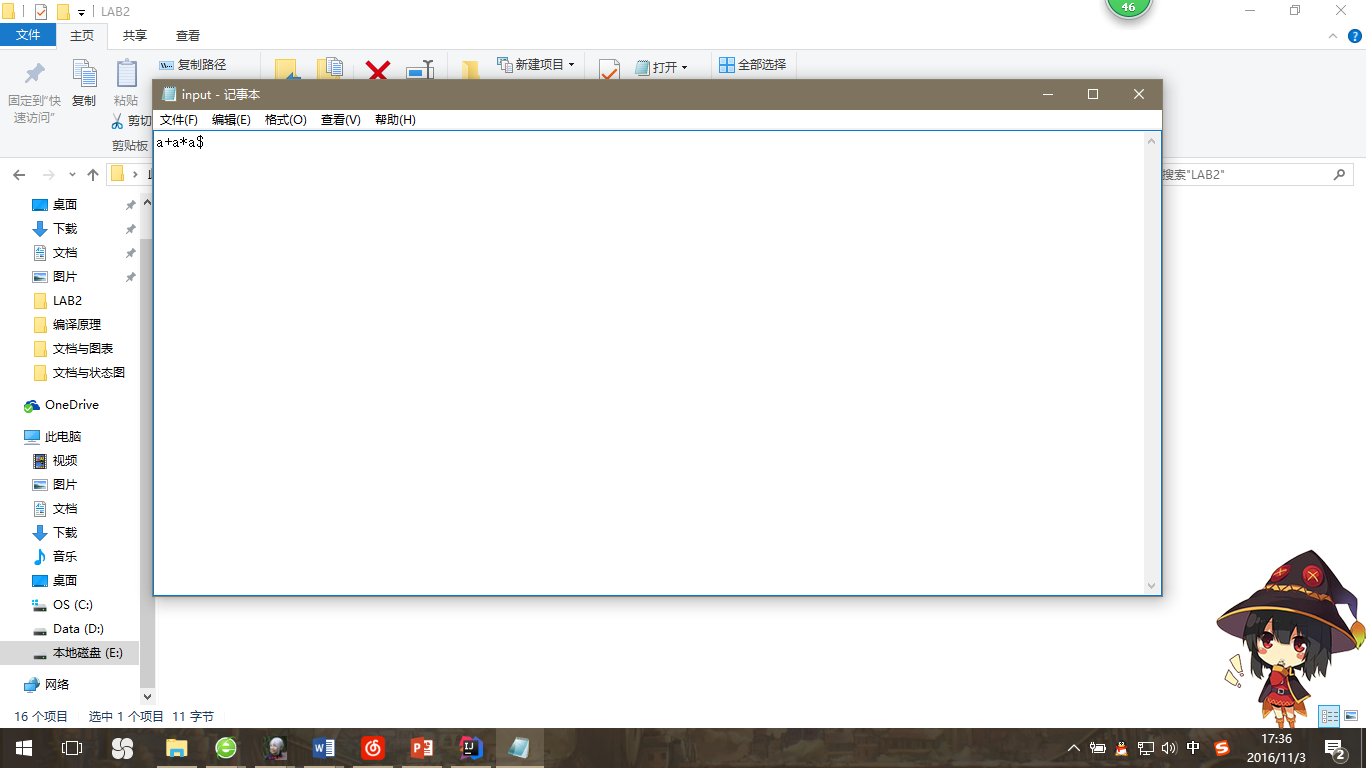
CFGs输入：CFGs.txt



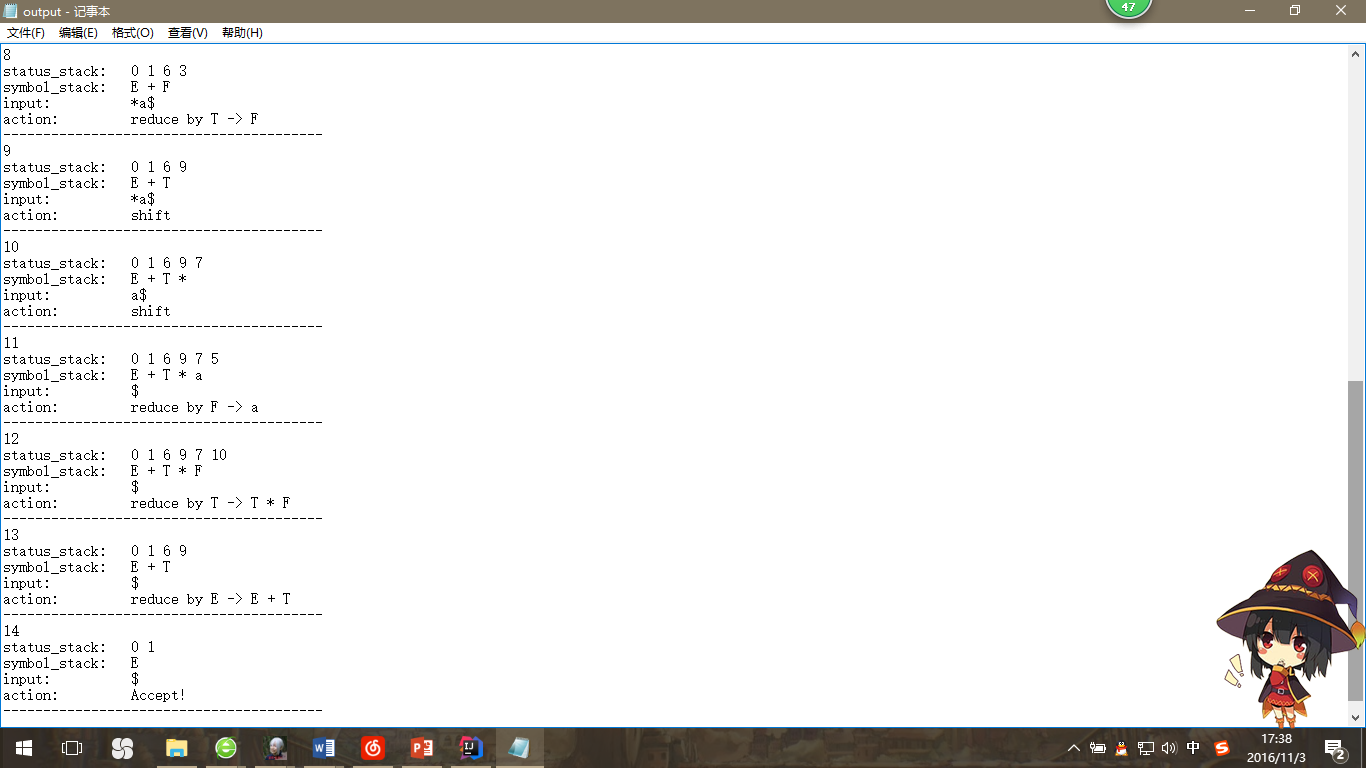
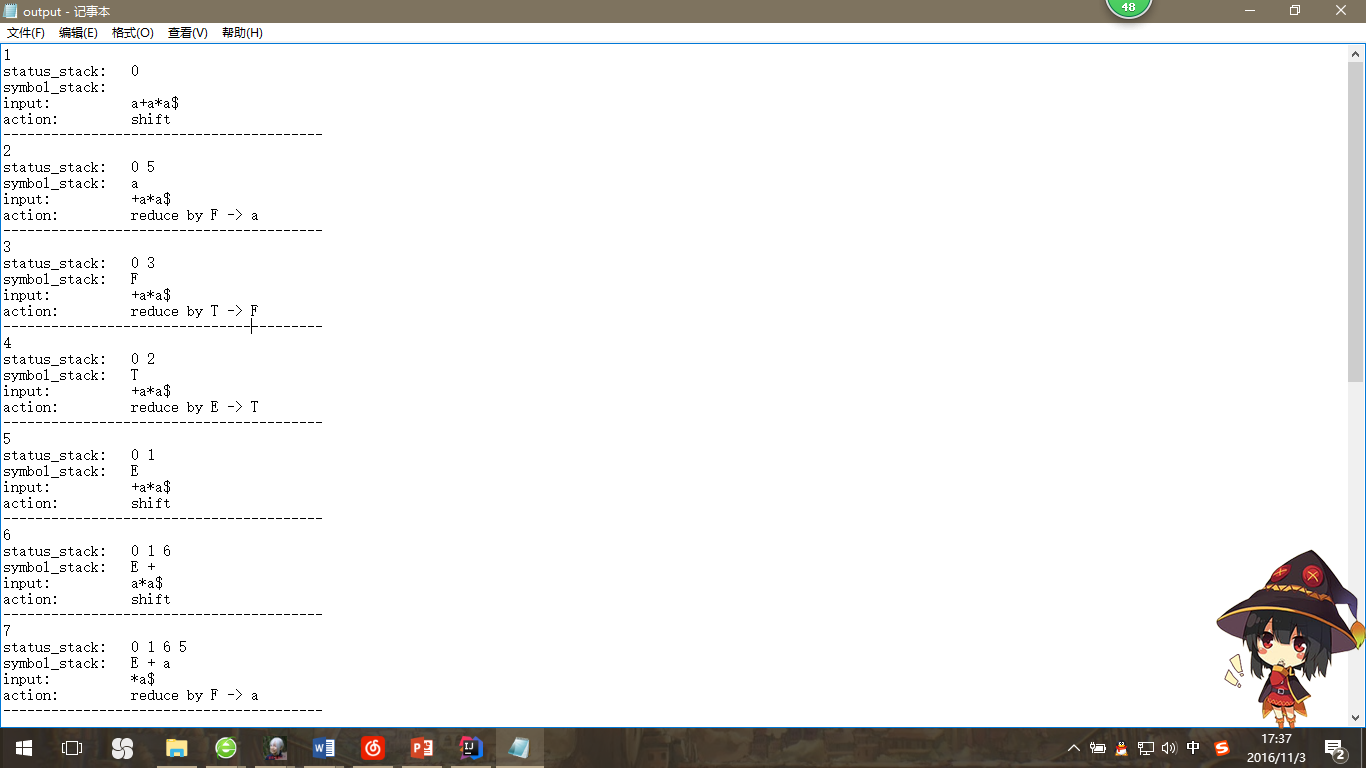
分析表输入：parse\_table.txt



源程序输入：input.txt

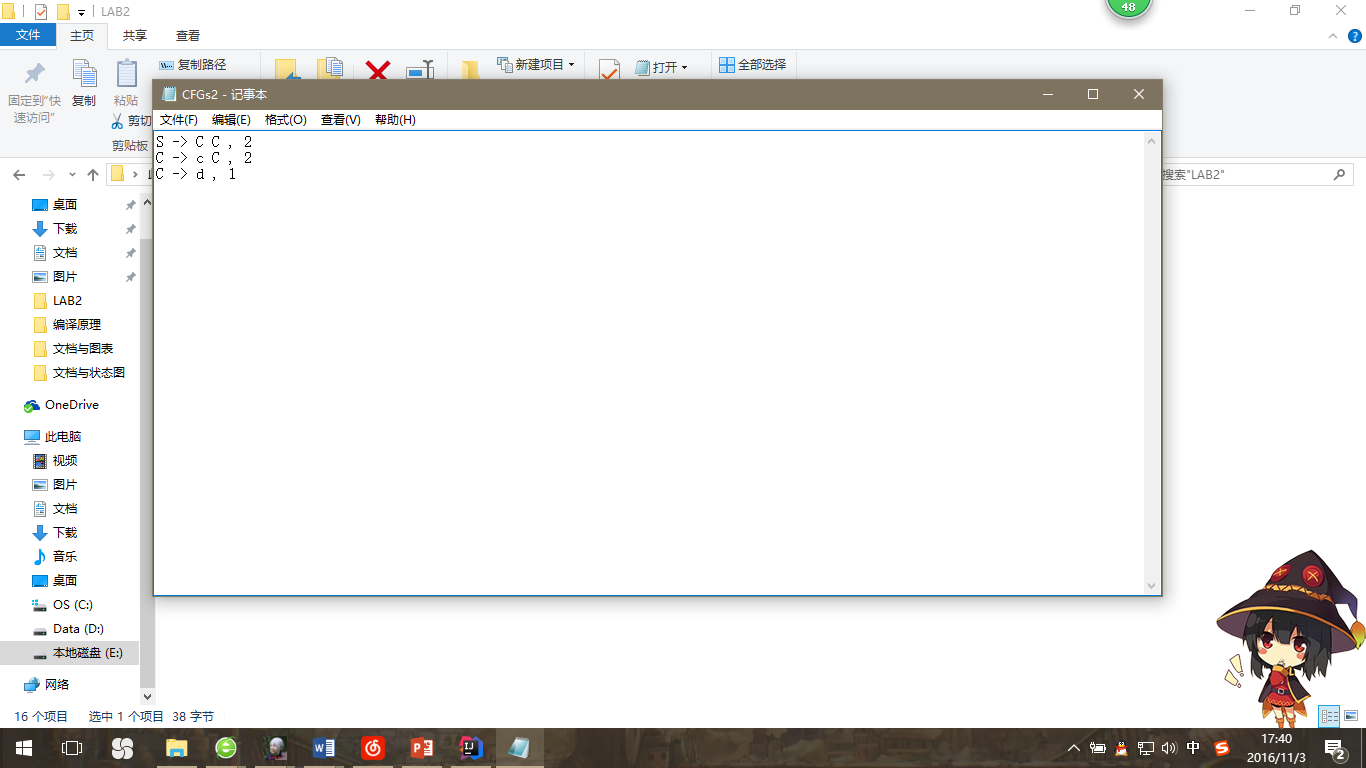


输出：output.txt

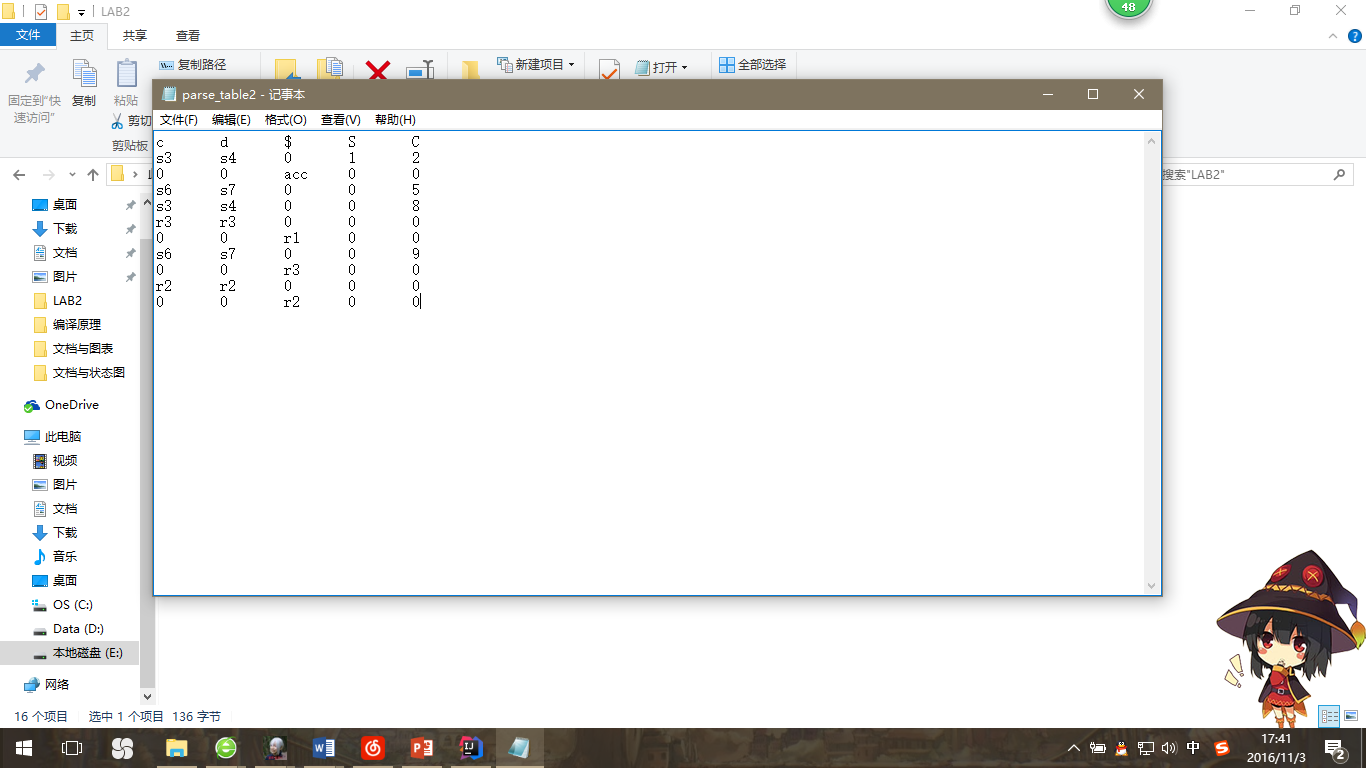


1. 测试用例b

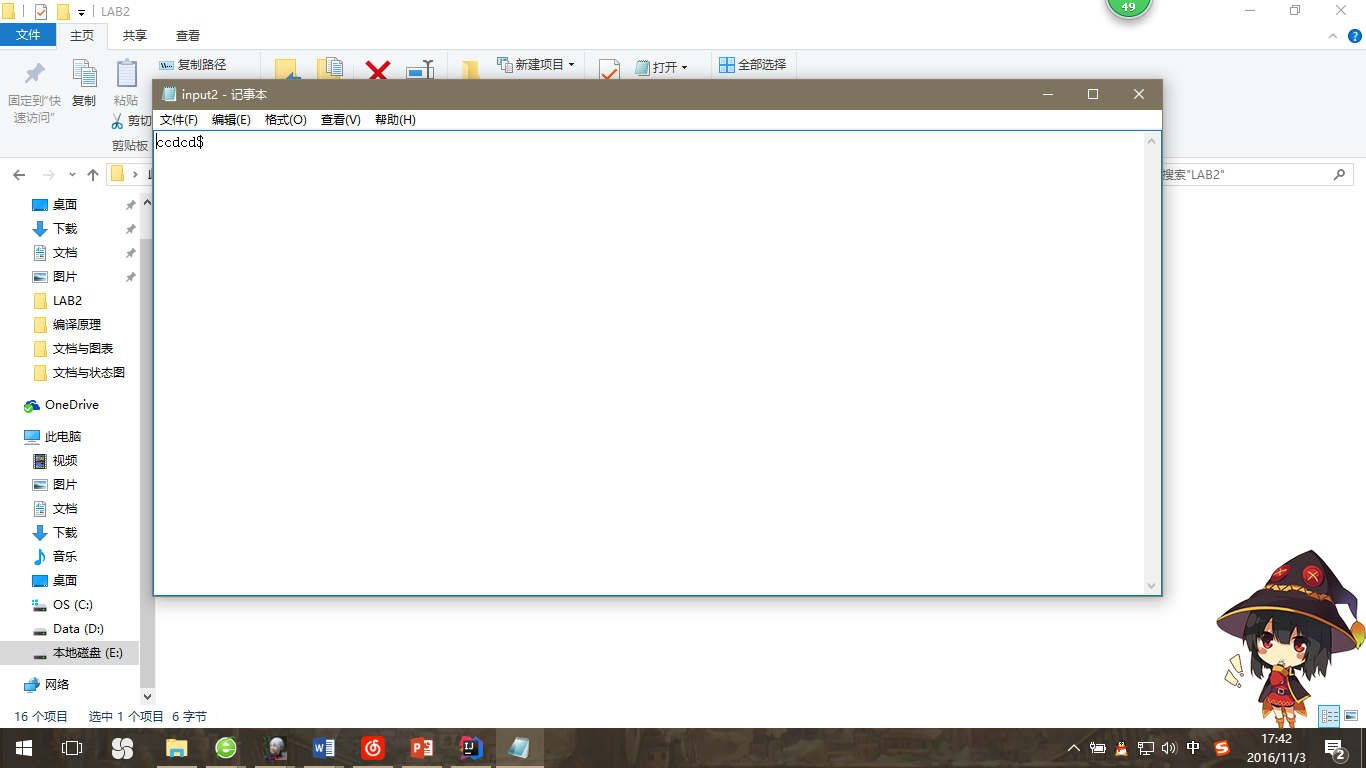
输入CFGs：CFGs2.txt



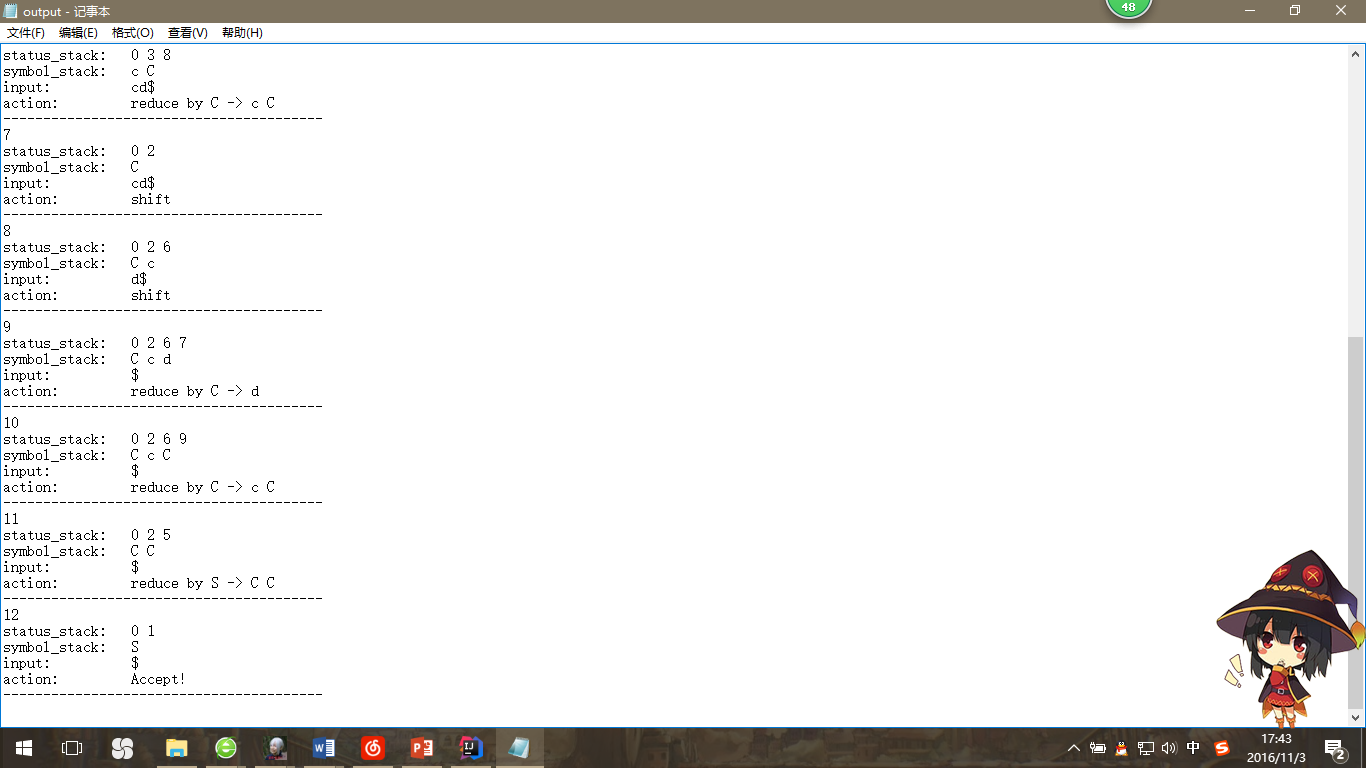
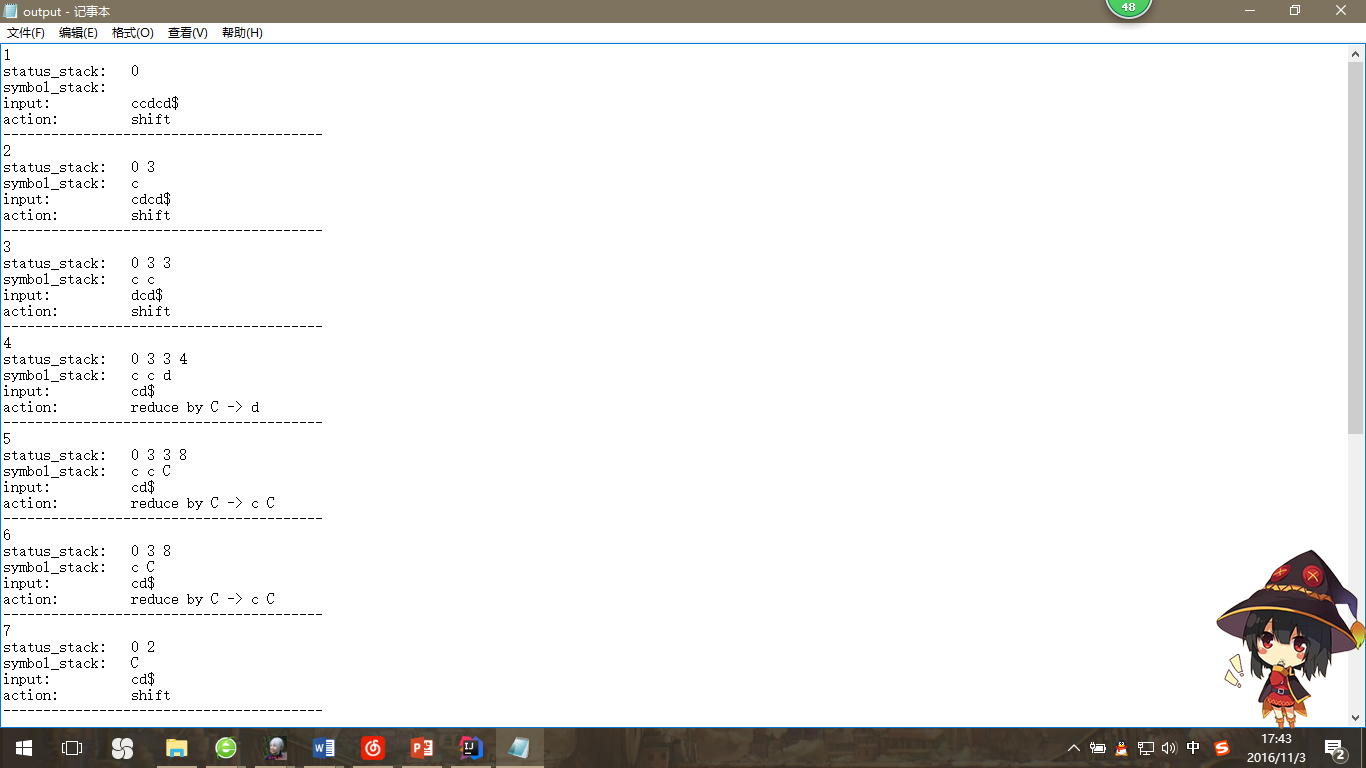
输入分析表：parse\_table2.txt



输入源程序：input2.txt



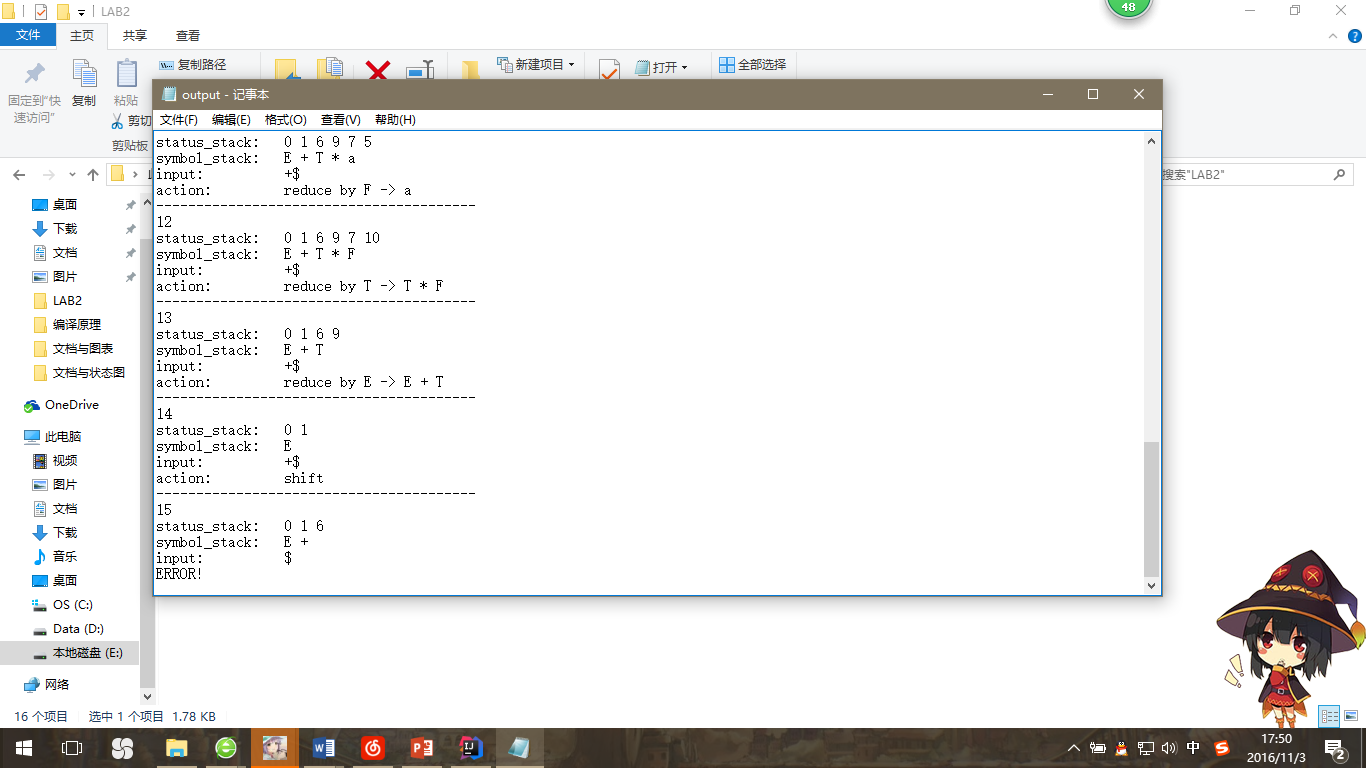
输出：



1. 测试用例c

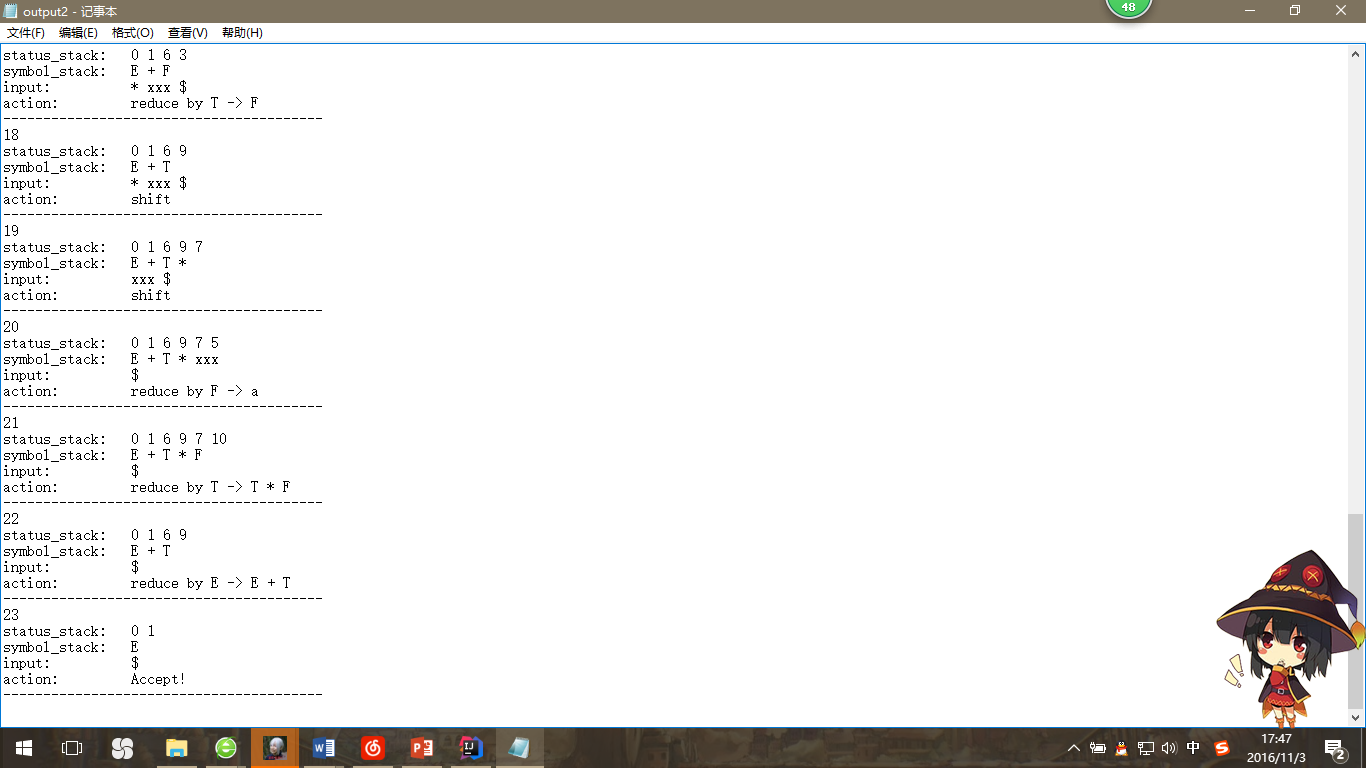
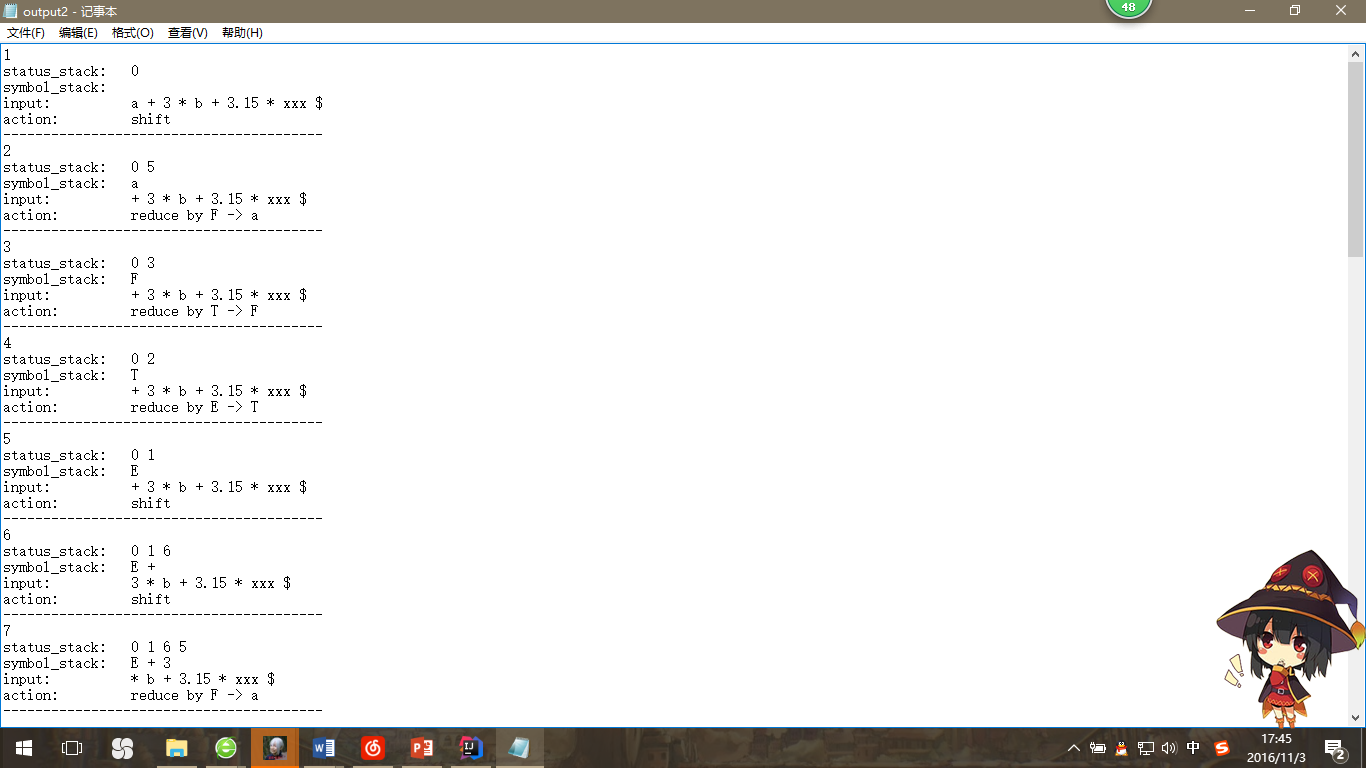
该测试用例与a使用相同的文法和分析表，并在源程序中使用了错误的文法

输出：

在遇到错误文法时会跳出循环并报错

1. 测试用例d

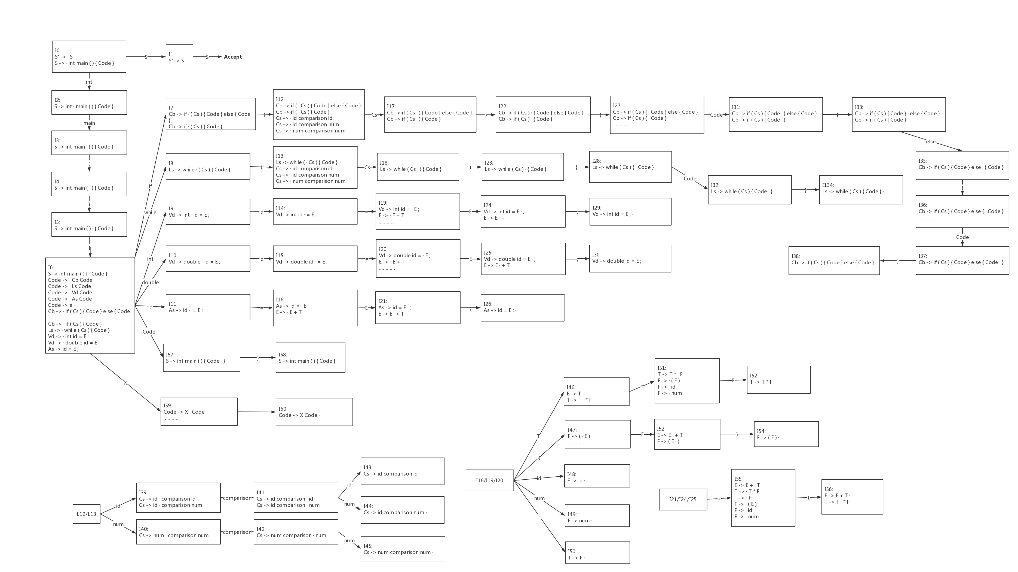
该测试用例加入了LAB1的词法分析过程（使用了自己实现的词法分析器），用测试用例a的上下文无关文法对带变量和数字的加、乘运算进行分析



可正常进行移入与归约

1. Problems occurred and related solutions

* 事实上从CFGs到parse\_table是一个很繁琐而且极易出错的过程，在使用一些文法集时不得不采用LR(1)进行状态转换分析，会产生大量状态。我在实验最初尝试分析一个精简版C语言的文法集，包括22条产生式，在进行状态分析时产生了多达60以上的不同状态：



所以我在进行状态分析表填写时放弃了这一步，而选用一些简单的文法来进行编码实验。事实上，最终的程序是对不同文法有适应性的，即只要文法与分析表正确且对应便能进行分析。所以规律与方法还是应从简单的状态向复杂推导，而不是试图一步登天。

* 我在LAB1时使用了switch-case语句来模拟词法分析的有限状态机，但本次LR分析表显然结构更复杂，而且switch-case语句无法适应不同文法。因此我选用了表驱动的方法，通过维护一个记录LR分析表的二维数组来进行查找与跳转。这个方法大大简化了代码结构，同时有了良好的适应性。

1. Your feelings and comments

虽然我在本次实验中浪费了太多无用功，但在分析、计算、编码的过程中我对自底向上的文法分析，包括LR(0)，SLR，规范LR(1)，LALR的区别有了更加清晰的认识，同时也让我在何为良好的代码构造上有了新的实践体验。