**编译原理**

班级：21计算机科学与技术（4）班

姓名：陈昊天 学号：2021329600006

实验二 自顶向下语法分析器的构建

1. 实验要求

运用LL(1)分析法，针对给定的上下文无关文法，给出实验方案。预估实验中可能出现的问题。

1. 实验方案

（评价依据实验方案设计是否合理，包括输入输出的设计）

使用递归下降分析技术。

基本思想：

识别程序由一组子程序构成，其中每个子程序对应于文法的一个非终结符；这些子程序通常是递归子程序。

文法中每个非终结符的对应的子程序，根据选定的产生式右部出现的符号来设计，具体构造方法如下：

1. 每遇到一个终结符，则判断当前读入token是否与该终结符匹配。若匹配，则继续读下一个token，若不匹配，则进行错误处理。
2. 每遇到一个非终结符，则调用相应的子程序。

输入设计：

输入的序列分解为token序列，一个字符识别为一个token，简化了输入部分的处理。

输出设计：

在每个子程序的处理中，若成功则继续处理，若发生错误则进行错误处理。在错误处理时如果无法回退，则输出 LL1 parse failed 表示解析错误。

1. 预估问题

预估问题（是否有预估的问题，预估的问题是否合理）

1. 需要将给定文法转换为LL1文法
2. 输入的句子不符合给定文法，进行错误处理

理论基础（评价依据 理论知识非常清楚）

一般情况下，假定关于A的产生式是：

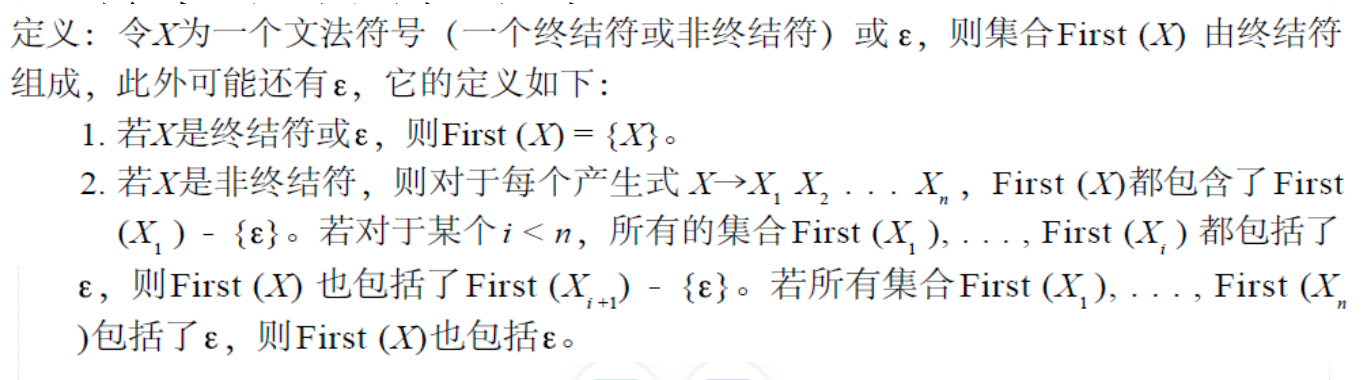
A→Aα1| Aα2 |… |Aαm|β1|β2 |…|βn

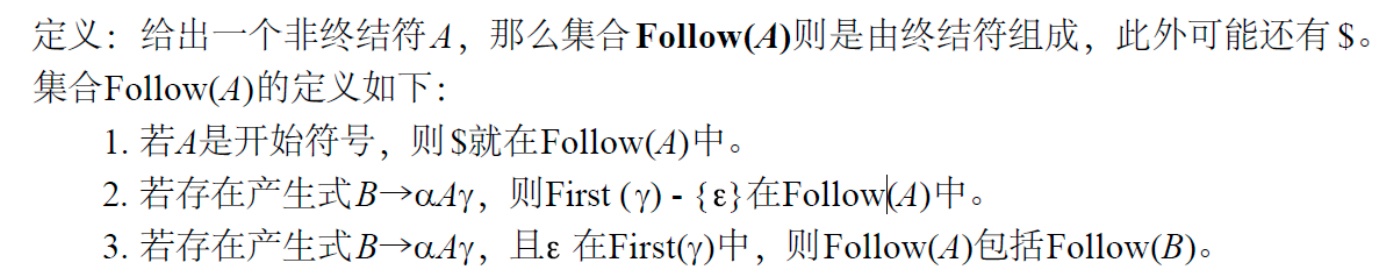
其中，αi(1≤i≤m)均不为空，βj(1≤j≤n)均不以A打头。

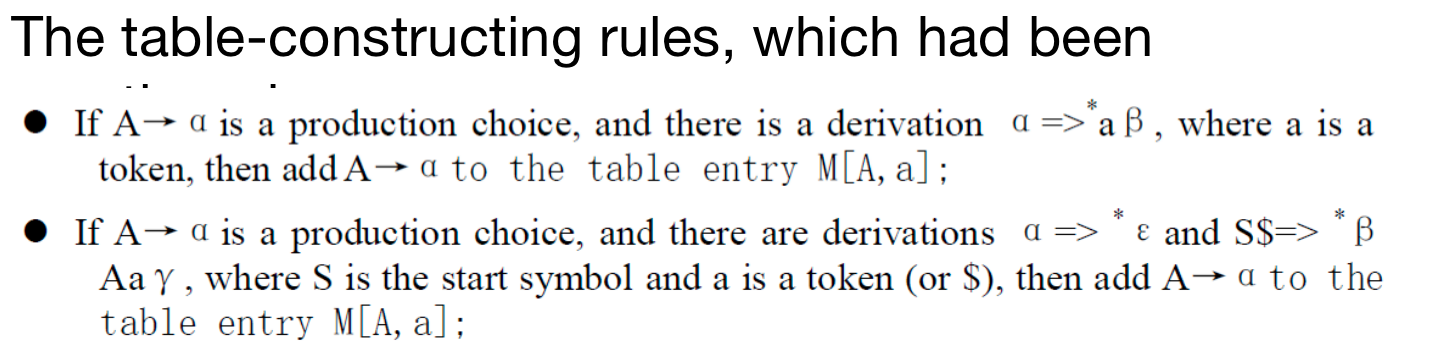
则消除直接左递归后改写为：

A→ β1A'| β2 A' |…| βnA'

A'→ α1A' | α2A' |…| αmA' |ε







输入：文法G

输出：分析表M

步骤：

1、对G中任意一个产生式 A --> α 执行第2步和第3步

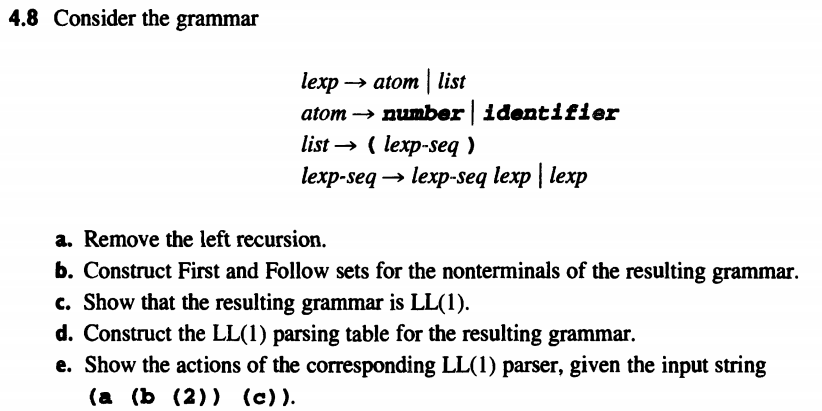
2、for 任意a ∈ First(α)，将 A --> α 填入M[A,a]

3、if ε ∈ First(α) then 任意a ∈ Follow(A)，将 A --> α 填入M[A,a]

if ε ∈ First(α) & # ∈Follow(A)， then 将 A --> α 填入M[A,#]

1. 内容和步骤

1、针对4.8习题输入和输出的设计及代码



**原文法**

lexp -> atom | list

atom -> **number** | **identifier**

list -> **(** lexp-seq **)**

lexp-seq -> lexp-seq lexp | lexp

1. 消除左递归

lexp -> atom | list

atom -> **number** | **identifier**

list -> **(** lexp-seq **)**

lexp-seq -> lexp lexp-seq’

lexp-seq’ -> lexp lexp-seq’ | ε

1. 构造First和Follow集
2. First集

First( lexp ) = { number, identifier, ( }

First( atom ) = { number, identifier }

First( list ) = { ( }

First( lexp-seq ) = { number, identifier, ( }

First( lexp-seq’ ) = { ε, number, identifier, ( }

1. Follow集

Follow( lexp ) = { $, number, identifier, (, ) }

Follow( atom ) = { $, number, identifier, (, ) }

Follow( list ) = { $, number, identifier, (, ) }

Follow( lexp-seq ) = { ) }

Follow( lexp-seq’ ) = { ) }

1. 判定是LL1文法

文法G是LL(1)的，当且仅当G的任意两个具有相同左部的

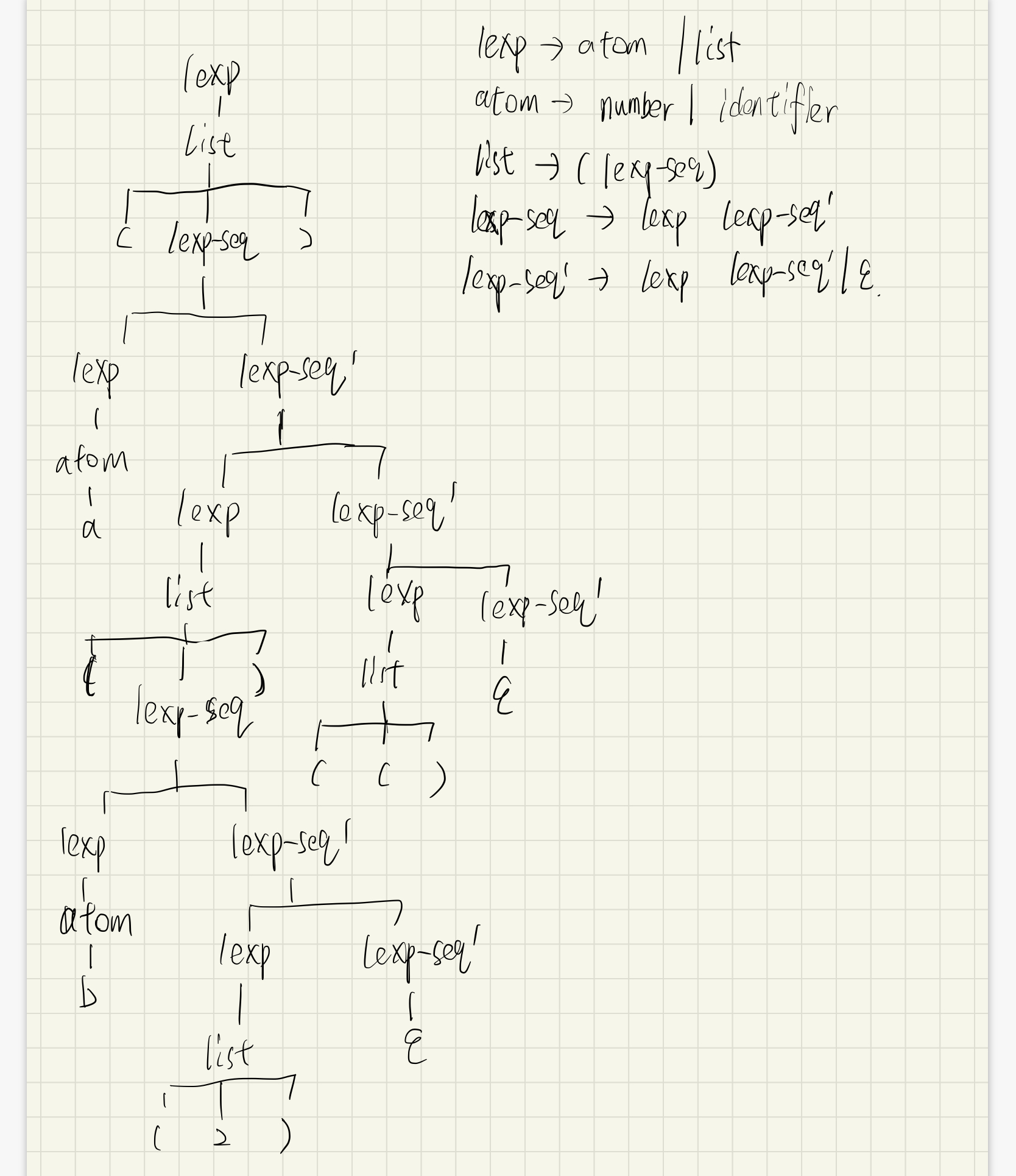
产生式A -> α|β满足下面的条件：

1. 不存在终结符a使得α和β都能够推导出以a开头的串
2. α和β至多有一个能推导出ε
3. 如果β可以广义推导出ε，则First (α)∩Follow( A ) = 空集
4. 如果α可以广义推导出ε，则First (β)∩Follow( A ) = 空集
5. 构造分析表

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | number | identifier | ( | ) | $ |
| lexp | lexp -> atom | lexp -> atom | lexp -> list |  |  |
| atom | Atom -> number | Atom -> identifier |  |  |  |
| list |  |  | list -> **(** lexp-seq **)** |  |  |
| lexp-seq | lexp-seq -> lexp lexp-seq’ | lexp-seq -> lexp lexp-seq’ | lexp-seq -> lexp lexp-seq’ |  |  |
| lexp-seq’ | lexp-seq’ -> lexp lexp-seq’ | lexp-seq’ -> lexp lexp-seq’ | lexp-seq’ -> lexp lexp-seq’ | lexp-seq’ -> ε |  |

D-2: 分析树

其中list子树有一定简化



1. 用语法分析器处理(a (b (2)) (c))

代码和结果见《三、实验结果》部分

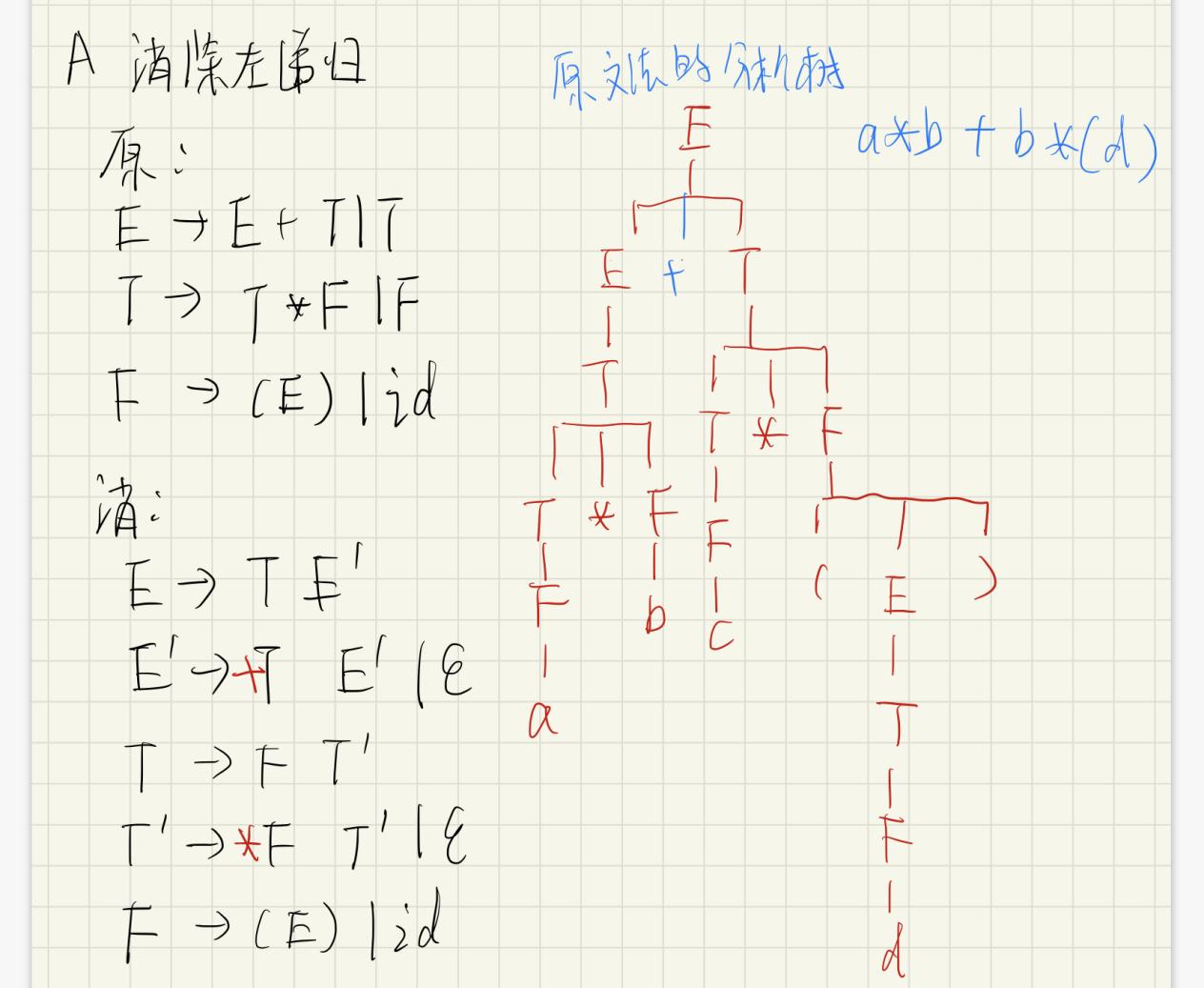
2、考虑简单算术表达式文法G:

E→E + T | T

T→T \* F | F

F→(E) | id

试设计LL(1)分析程序，以对任意输入的符号串进行语法分析。



代码见实验结果

3、实验具体步骤

已在1、2步中写出

1. **实验结果:**

**4-8**

1. 代码

#include <iostream>

#include <vector>

using namespace std;

class PARSER {

public:

vector<string> token;

string input;

char symbol;

int index = 0, oldIndex = 0, slen = 0;

void getToken() {

// cin >> input;

input = "(a(b(2))(c))";

slen = input.size();

/\*for (auto i : input) {

token.push\_back("" + i);

}\*/

}

void getSymbol() {

if (index >= slen) {

cout << "getSymbol: out of token range" << endl;

symbol = '`';

index++;

return;

}

symbol = input[index];

index++;

// cout << symbol << endl;

}

void start(string func, int depth) {

for (int i = 0; i < depth; i++) {

cout << " |";

}

cout << "--" + func << endl;

}

void success(string func) {}

void failed(string func, int depth) {

for (int i = 0; i < depth; i++) {

cout << " |";

}

cout << "--" + func + ": rollback" << endl;

// cout << "Err: " + func + " rollback" << endl;

// cout << func + " failed" << endl;

}

bool lexp(int depth) {

start("lexp", depth);

if (atom(depth + 1) or list(depth + 1)) {

success("lexp");

return true;

}

failed("lexp", depth);

return false;

}

bool atom(int depth) {

start("atom", depth);

oldIndex = index;

getSymbol();

if (symbol >= '0' and symbol <= '9') {

string tmp(1, symbol);

start(tmp, depth + 1);

success("atom");

return true;

} else if (symbol >= 'a' and symbol <= 'z') {

string tmp(1, symbol);

start(tmp, depth + 1);

success("atom");

return true;

}

failed("atom", depth);

index = oldIndex;

return false;

}

bool list(int depth) {

start("list", depth);

oldIndex = index;

getSymbol();

if (symbol == '(') {

start("(", depth + 1);

if (lexpSeq0(depth + 1)) {

getSymbol();

if (symbol == ')') {

start(")", depth + 1);

success("list");

return true;

}

}

}

failed("list", depth);

index = oldIndex;

return false;

}

bool lexpSeq0(int depth) {

start("lexp-seq", depth);

if (lexp(depth + 1)) {

if (lexpSeq1(depth + 1)) {

success("lexp-seq");

return true;

}

}

failed("lexp-seq", depth);

return false;

}

bool lexpSeq1(int depth) {

start("lexp-seq'", depth);

if (lexp(depth + 1)) {

if (lexpSeq1(depth + 1)) {

success("lexp-seq'");

return true;

}

}

success("lexp-seq'");

return true;

}

};

int main() {

// cout << "unit test" << endl;

PARSER parser;

parser.getToken();

if (parser.lexp(1)) {

cout << "LL(1) parse success" << endl;

} else {

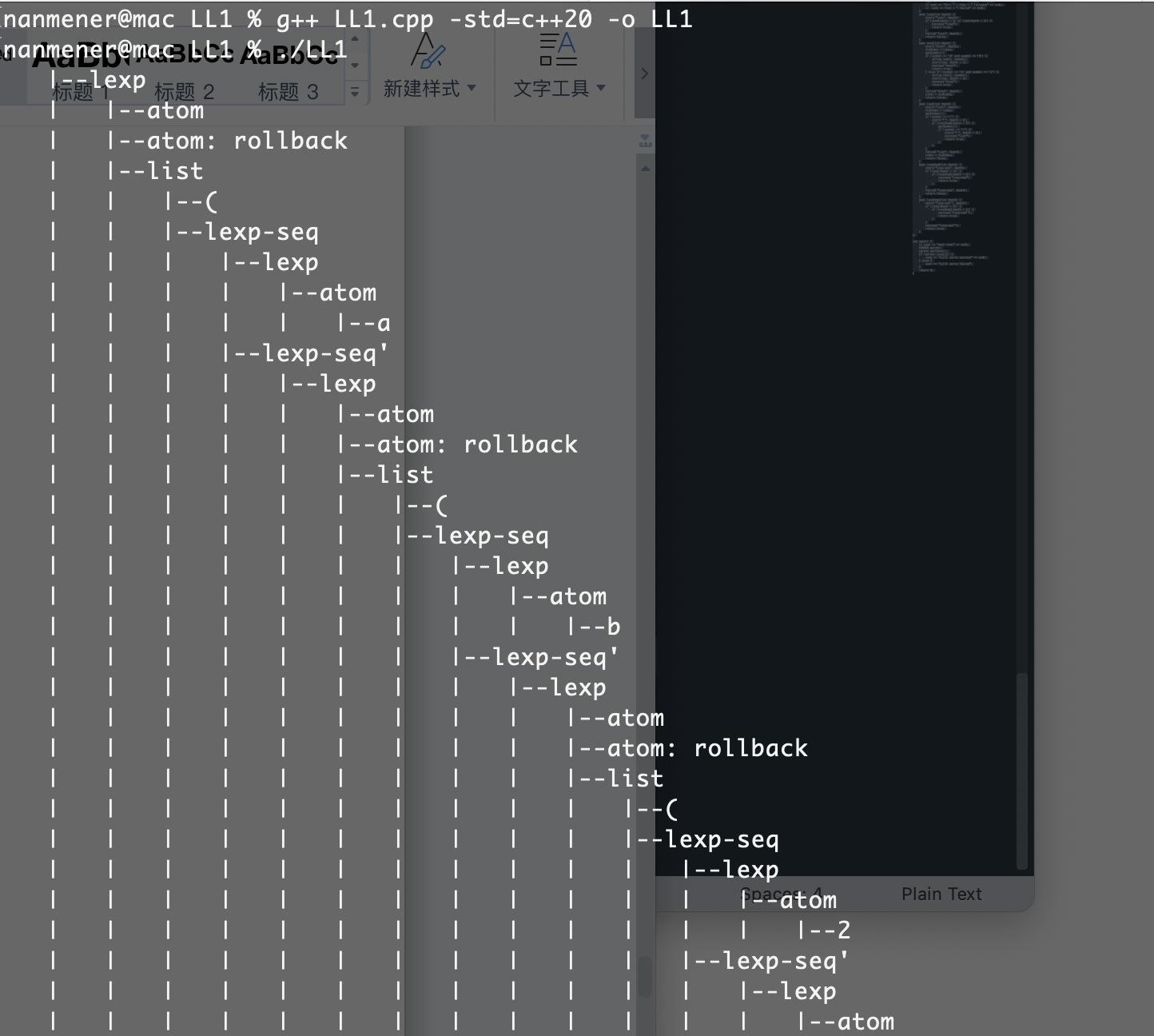
cout << "LL(1) parse failed";

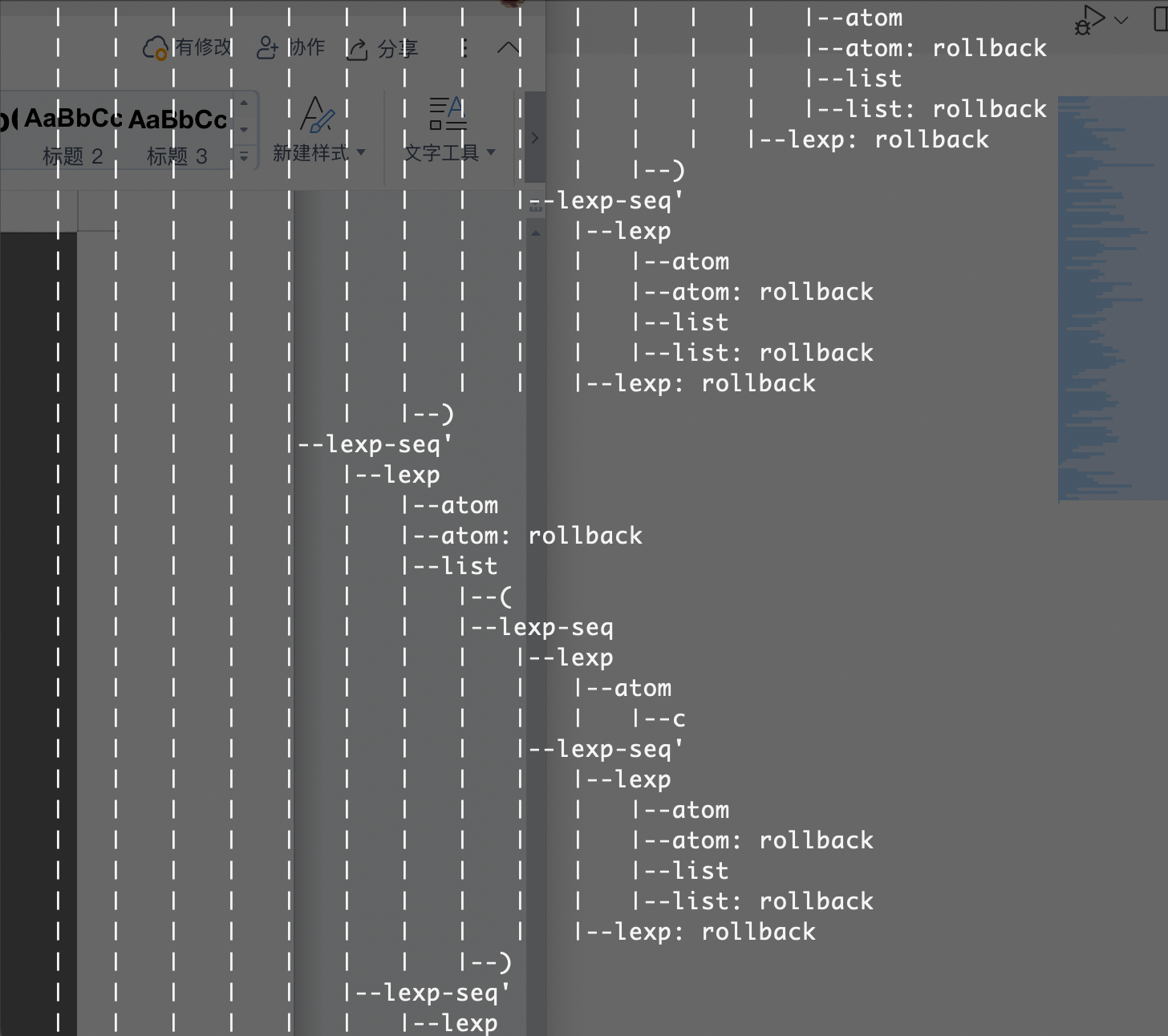
}

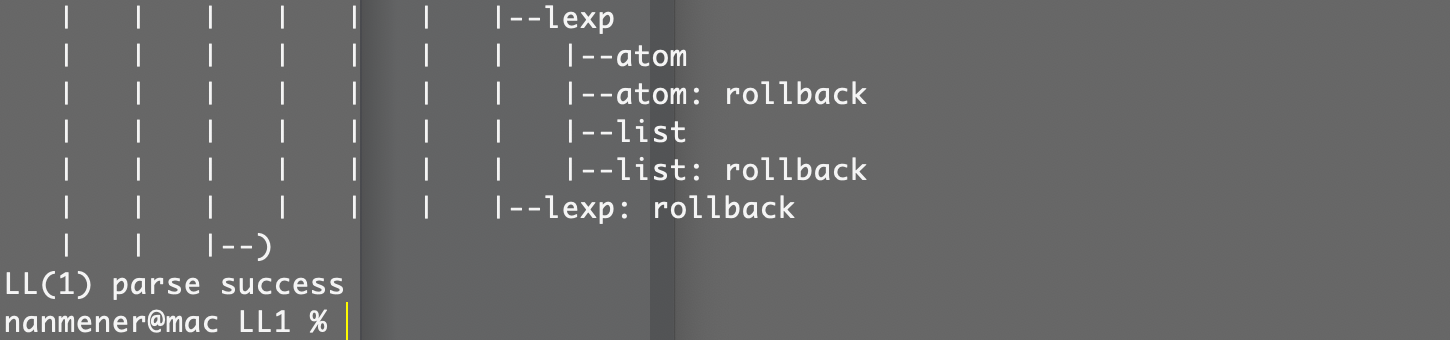
return 0;

}

1. 截图







2、第二题

1. 代码

#include <iostream>

#include <vector>

using namespace std;

class PARSER {

public:

vector<string> token;

string input;

char symbol;

int index = 0, oldIndex = 0, slen = 0;

void getToken() {

// cin >> input;

input = "a\*b+b\*(d)";

slen = input.size();

/\*for (auto i : input) {

token.push\_back("" + i);

}\*/

}

void getSymbol() {

if (index >= slen) {

// cout << "getSymbol: out of token range" << endl;

symbol = '`';

index++;

return;

}

symbol = input[index];

index++;

// cout << symbol << endl;

}

void start(string func, int depth) {

for (int i = 0; i < depth; i++) {

cout << " |";

}

cout << "--" + func << endl;

}

void success(string func) {}

void failed(string func, int depth) {

for (int i = 0; i < depth; i++) {

cout << " |";

}

cout << "--" + func + ": rollback" << endl;

// cout << "Err: " + func + " rollback" << endl;

// cout << func + " failed" << endl;

}

bool E0(int depth) {

start("E", depth);

if (T0(depth + 1)) {

if (E1(depth + 1)) {

success("E");

return true;

}

}

failed("E", depth);

return false;

}

bool E1(int depth) {

start("E'", depth);

oldIndex = index;

getSymbol();

if (symbol == '+') {

string tmp(1, symbol);

start(tmp, depth + 1);

if (T0(depth)) {

if (E1(depth)) {

success("E'");

return true;

}

}

}

index = oldIndex;

success("E'");

return true;

}

bool T0(int depth) {

start("T", depth);

if (F(depth + 1)) {

if (T1(depth + 1)) {

success("T");

return true;

}

}

failed("T", depth);

return false;

}

bool T1(int depth) {

start("T'", depth);

oldIndex = index;

getSymbol();

if (symbol == '\*') {

start("\*", depth + 1);

if (F(depth + 1)) {

if (T1(depth + 1)) {

success("T'");

return true;

}

}

}

index = oldIndex;

success("T'");

return true;

}

bool F(int depth) {

start("F", depth);

getSymbol();

if (symbol == '(') {

start("(", depth + 1);

if (E0(depth + 1)) {

getSymbol();

if (symbol == ')') {

start(")", depth + 1);

success("F");

return true;

}

}

failed("F", depth + 1);

return false;

}

if (symbol >= 'a' and symbol <= 'z') {

string tmp(1, symbol);

start(tmp, depth + 1);

success("F");

return true;

}

failed("F", depth + 1);

return false;

}

};

int main() {

// cout << "unit test" << endl;

PARSER parser;

parser.getToken();

if (parser.E0(1)) {

cout << "LL(1) parse success" << endl;

} else {

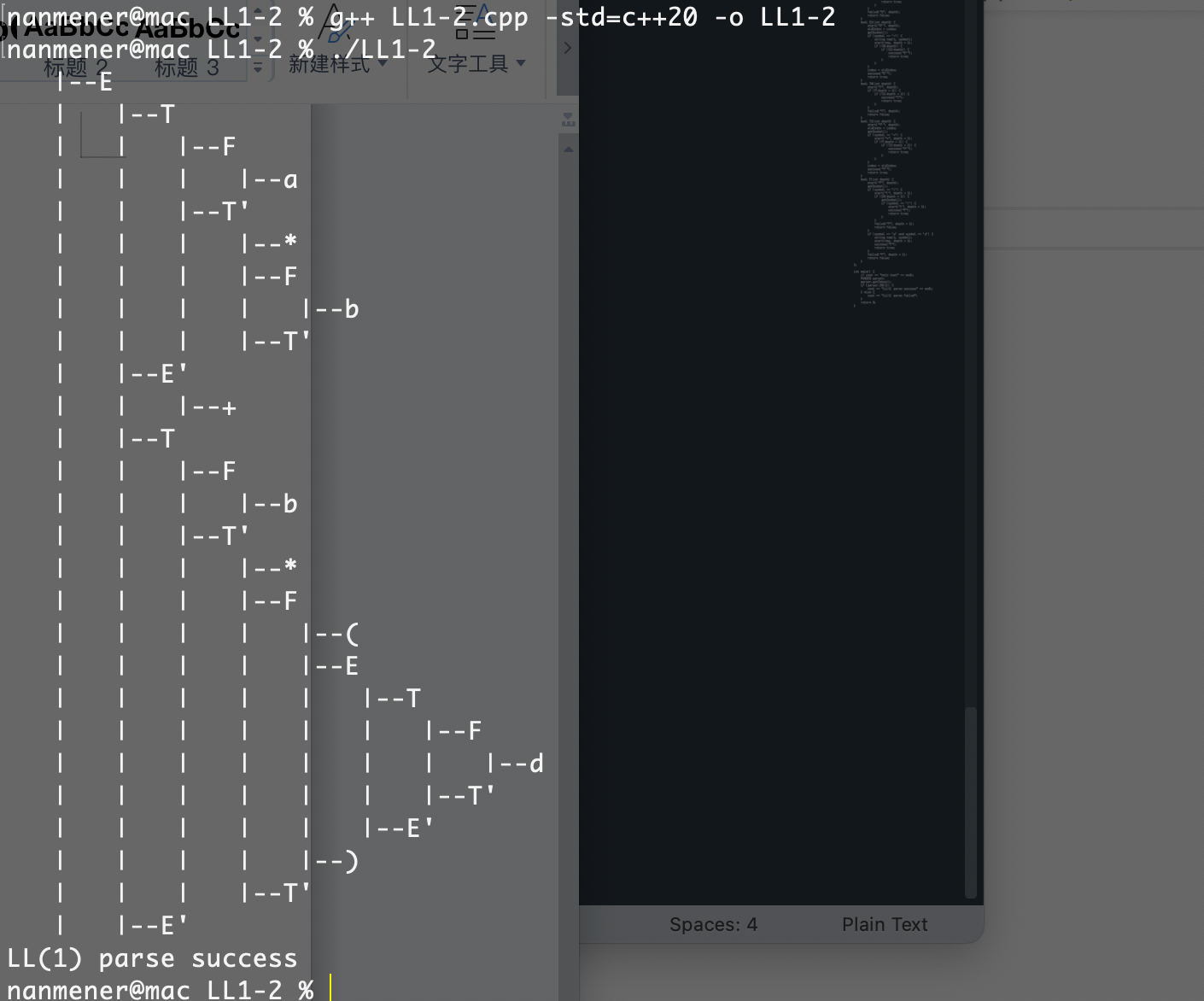
cout << "LL(1) parse failed";

}

return 0;

}

1. 截图



1. 实验结论:

1 、**实验结论**

（是否能够准确描述实验的结论）

递归下降分析：

递归下降分析是一种自顶向下的语法分析技术，它基于文法的产生式规则来分析语法结构。它的主要优点是实现简单、易于理解和调试，因为它的实现可以直接映射到文法的产生式规则上，而且它的错误报告和错误恢复比较容易。但是，递归下降分析的缺点是它不能处理左递归文法，因为它会导致无限递归调用，而且它的效率也不高，因为它可能会重复分析相同的子树。

表驱动分析：

表驱动分析是一种自底向上的语法分析技术，它基于文法的 LR 分析表来分析语法结构。它的主要优点是它可以处理任何类型的文法，包括左递归文法，并且它的效率比递归下降分析更高，因为它只需要分析一次输入。但是，表驱动分析的缺点是它的实现比较复杂，因为它需要构建 LR 分析表，而且它的错误恢复比较困难。

递归下降分析和表驱动分析各有优缺点。递归下降分析简单易于实现和调试，但不能处理左递归文法，效率也不高；表驱动分析可以处理任何类型的文法，效率较高，但实现较为复杂，错误恢复也比较困难。选择哪种技术取决于具体的应用场景和需求。

2、**分析和总结**

1）对输入设计的结论

输入内容为一个句子，需要把它分解为token。在本实验中，输入设计比较简单，一个字符即识别为一个token。

1. 对输出设计的结论

输出内容为一个分析树，横向表示。在解析成功时输出LL(1) parse success，否则输出LL(1) parse failed。在分析树中同时呈现递归进入错误分支、进行错误处理并回滚（恢复现场）的操作。

3）对LL(1)分析法的结论

需要把给定文法转换为LL(1)文法，需要消除左递归。如果不消除左递归，在递归下降分析程序中会出现无限递归的问题。

3、**对预估问题的结论**

1. 需要将给定文法转换为LL(1)文法

如前所述，如果不消除左递归，在递归下降分析程序中会出现无限递归的问题。

2. 输入的句子不符合给定文法或递归时进入错误分支，进行错误处理

如前所述，若不符合给定文法，则进行回滚，直到无法回滚为止（到达第一级）。若无法回滚，则报错并显示LL(1) parse failed。

若递归时进入错误分支，则进行回滚并恢复现场，然后递归另一分支。