**编译原理**

班级：23计算机科学与计算机（4）班

姓名：李凯涛 学号：2023327100056

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，构建出相应的语法分析树。预估实验中可能出现的问题。

1. 实验方案

**实验方案设计**

**目的**：验证LL(1)语法分析器对括号表达式的解析准确性。

**输入**：设计合法（如**"(a(b(2))(c))"**）和非法（如**"(a(b(2))(c)"**）的括号表达式，包含数字和小写字母，符合LL(1)语法定义。

**输出**：记录分析器对每个输入的解析结果（成功/失败）。

**验证步骤**：

1. 输入合法表达式，验证解析成功。
2. 输入非法表达式，验证解析失败。
3. 对比实验结果与预期，检查分析器准确性。

**结论**：根据实验结果，评估LL(1)语法分析器对括号表达式的解析是否符合预期。

该方案覆盖多种输入情况，确保验证全面。

1. 预估问题

**预估问题：**

**分析：**

1. **合法表达式**：验证LL(1)分析器能否正确解析。
2. **非法表达式**：验证能否准确判定解析失败。
3. **复杂嵌套结构**：测试多层嵌套及混合元素的解析能力。
4. **结果判定**：检查分析器能否正确输出解析成功/失败。
5. **评估有效性**：实验数据能否客观评估分析器的准确性和合理性。

以上预估问题围绕实验目标，全面验证LL(1)分析器的功能，确保实验结果的可靠性和有效性。

**理论基础：**

**定义**：  
LL(1)语法分析器是一种**自顶向下**的语法分析器，特点：

* **从左到右（L）**扫描输入
* **最左推导（L）**
* **1个向前符号**预测产生式

**核心组成（四元组）**：

* **G**：文法规则集合
* **N**：非终结符集合
* **T**：终结符集合
* **P**：产生式集合

**LL(1)文法要求**：  
对任意非终结符 **A** 和输入符号 **a**，若存在多个产生式 **A→β** 和 **A→γ**，需满足：

1. **FIRST(β) ∩ FIRST(γ) = ∅**（无冲突预测）
2. 若 **β 可推导出 ε**，则 **FIRST(γ) ∩ FOLLOW(A) = ∅**

**工作流程**：

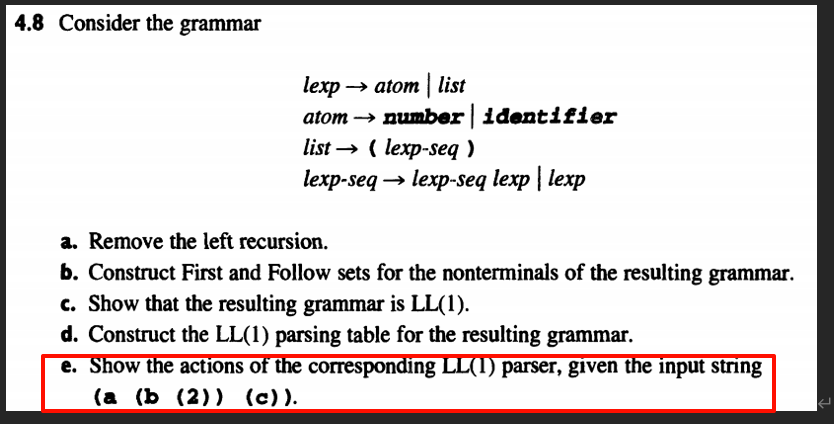
1. **构建分析表**：基于 **SELECT 集**（FIRST + FOLLOW）确定产生式选择。
2. **递归下降分析**：根据当前符号和栈顶非终结符查表预测，匹配失败则报错。
3. **推导验证**：逐步展开产生式，直至输入串被接受或判定为非法。

**特点**：

* 高效（无回溯）
* 严格依赖文法LL(1)性质
* 适用于简单、无二义性的语法结构（如算术表达式、括号匹配等）

1. 内容和步骤

1、针对4.8习题输入和输出的设计及代码



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。

3、实验具体步骤

实现LL(1)语法分析器代码，按需调整。设计合法和非法的括号表达式作为输入，调用分析器解析并输出结果。检查解析结果是否符合LL(1)语法定义，并记录输入、输出及结论。

1. 实验结果:
2. 代码

**第一个任务：**

# 定义特殊符号

EPSILON = "ε"

EOF\_SYMBOL = "$"

# --- 预定义的LL(1)分析表和产生式 ---

# 对于这个特定的文法，我们将分析表和产生式规则硬编码

# 非终结符

NON\_TERMINALS = ["lexp", "atom", "list", "lexp\_seq", "lexp\_seq\_prime"]

# 产生式右部 (RHS)

# productions\_map 的键是 (非终结符, 当前输入符号)

# 值是该产生式的RHS符号列表

productions\_rhs\_map = {

("lexp", "id"): ["atom"],

("lexp", "num"): ["atom"],

("lexp", "("): ["list"],

("atom", "id"): ["id"], # 特殊：产生式直接产生终结符

("atom", "num"): ["num"], # 特殊：产生式直接产生终结符

("list", "("): ["(", "lexp\_seq", ")"],

("lexp\_seq", "id"): ["lexp", "lexp\_seq\_prime"],

("lexp\_seq", "num"): ["lexp", "lexp\_seq\_prime"],

("lexp\_seq", "("): ["lexp", "lexp\_seq\_prime"],

("lexp\_seq\_prime", "id"): ["lexp", "lexp\_seq\_prime"],

("lexp\_seq\_prime", "num"): ["lexp", "lexp\_seq\_prime"],

("lexp\_seq\_prime", "("): ["lexp", "lexp\_seq\_prime"],

("lexp\_seq\_prime", ")"): [EPSILON],

("lexp\_seq\_prime", EOF\_SYMBOL): [EPSILON],

}

# 用于打印的产生式字符串表示

production\_rules\_str\_map = {

("lexp", "id"): "lexp -> atom",

("lexp", "num"): "lexp -> atom",

("lexp", "("): "lexp -> list",

("atom", "id"): "atom -> id",

("atom", "num"): "atom -> num",

("list", "("): "list -> ( lexp\_seq )",

("lexp\_seq", "id"): "lexp\_seq -> lexp lexp\_seq\_prime",

("lexp\_seq", "num"): "lexp\_seq -> lexp lexp\_seq\_prime",

("lexp\_seq", "("): "lexp\_seq -> lexp lexp\_seq\_prime",

("lexp\_seq\_prime", "id"): "lexp\_seq\_prime -> lexp lexp\_seq\_prime",

("lexp\_seq\_prime", "num"): "lexp\_seq\_prime -> lexp lexp\_seq\_prime",

("lexp\_seq\_prime", "("): "lexp\_seq\_prime -> lexp lexp\_seq\_prime",

("lexp\_seq\_prime", ")"): "lexp\_seq\_prime -> ε",

("lexp\_seq\_prime", EOF\_SYMBOL): "lexp\_seq\_prime -> ε",

}

def is\_terminal(symbol):

"""判断一个符号是否是终结符"""

return symbol not in NON\_TERMINALS or symbol == EPSILON

# --- 词法分析器 (Tokenizer) ---

def tokenize(input\_string):

"""将输入字符串转换为token列表"""

tokens = []

i = 0

while i < len(input\_string):

char = input\_string[i]

if char.isspace():

i += 1

continue

if char.isalpha():

# 在这个特定文法中，我们将单个字母视为 'id' 类型的token

# 如果文法更复杂，这里需要更完善的标识符识别

tokens.append("id") # 也可以是 char 本身，如果分析表这样定义

elif char.isdigit():

# 单个数字视为 'num' 类型的token

tokens.append("num") # 也可以是 char 本身

elif char in ['(', ')']:

tokens.append(char)

else:

raise ValueError(f"词法错误: 输入字符串中存在未知字符 '{char}'")

i += 1

tokens.append(EOF\_SYMBOL) # 添加输入结束标记

return tokens

# --- LL(1) 分析器 ---

def ll1\_parser(input\_tokens):

"""

使用LL(1)分析表和栈来分析token流

"""

# 初始化分析栈，放入结束标记和开始符号

# 栈顶在列表的末尾 (使用list的append和pop方法模拟栈)

stack = [EOF\_SYMBOL, "lexp"] # 开始符号是 "lexp"

token\_idx = 0 # 指向当前输入token的指针

max\_steps = 100 # 防止因文法或表错误导致的无限循环

current\_step = 0

print(f"{'步骤':<5} | {'分析栈 (栈顶在右)':<40} | {'剩余输入':<30} | {'动作'}")

print("-" \* 100)

while len(stack) > 0 and current\_step < max\_steps:

current\_step += 1

# 当前分析栈的可视化 (栈顶在右边)

# 为了打印方便，我们反转栈列表进行显示，但实际操作仍在列表末尾

stack\_visual = " ".join(stack)

# 当前剩余输入的可视化

input\_visual = " ".join(input\_tokens[token\_idx:])

top\_of\_stack = stack[-1] # 获取栈顶符号

# 获取当前输入符号

if token\_idx < len(input\_tokens):

current\_input\_token = input\_tokens[token\_idx]

else:

# 这通常不应该发生，因为EOF\_SYMBOL应该匹配

current\_input\_token = EOF\_SYMBOL # 以防万一

action\_taken = ""

if top\_of\_stack == EOF\_SYMBOL and current\_input\_token == EOF\_SYMBOL:

action\_taken = "接受 (Accept)"

print(f"{current\_step:<5} | {stack\_visual:<40} | {input\_visual:<30} | {action\_taken}")

print("-" \* 100)

print("\n语法分析成功!")

return True

if is\_terminal(top\_of\_stack):

if top\_of\_stack == current\_input\_token:

action\_taken = f"匹配终结符: '{top\_of\_stack}'"

stack.pop()

token\_idx += 1

elif top\_of\_stack == EPSILON: # 如果栈顶是ε，直接弹出

action\_taken = "处理ε产生式 (弹出 ε)"

stack.pop()

# 注意：处理ε时不消耗输入符号

else:

action\_taken = f"错误: 终结符不匹配! 栈顶: '{top\_of\_stack}', 输入: '{current\_input\_token}'"

print(f"{current\_step:<5} | {stack\_visual:<40} | {input\_visual:<30} | {action\_taken}")

print("-" \* 100)

print("\n语法分析失败!")

return False

else: # 栈顶是非终结符

table\_lookup\_key = (top\_of\_stack, current\_input\_token)

if table\_lookup\_key in productions\_rhs\_map:

production\_to\_use\_rhs = productions\_rhs\_map[table\_lookup\_key]

rule\_string = production\_rules\_str\_map[table\_lookup\_key]

action\_taken = f"应用产生式: {rule\_string}"

stack.pop() # 弹出当前非终结符

# 将产生式右部反向压入栈 (epsilon不入栈)

if production\_to\_use\_rhs != [EPSILON]:

for symbol in reversed(production\_to\_use\_rhs):

stack.append(symbol)

else:

action\_taken = f"错误: 分析表中无对应规则 M['{top\_of\_stack}', '{current\_input\_token}']"

print(f"{current\_step:<5} | {stack\_visual:<40} | {input\_visual:<30} | {action\_taken}")

print("-" \* 100)

print("\n语法分析失败!")

return False

print(f"{current\_step:<5} | {stack\_visual:<40} | {input\_visual:<30} | {action\_taken}")

if current\_step >= max\_steps:

print("-" \* 100)

print("\n错误: 超过最大分析步数，可能存在死循环。")

return False

# 如果循环结束但未显式接受，则通常意味着失败

# (例如，栈空了但输入未消耗完，或者输入完了但栈中还有非$符号)

# 之前的接受条件应该能覆盖成功的情况。

print("-" \* 100)

print("\n语法分析失败 (循环意外结束).")

return False

# --- 主程序 ---

if \_\_name\_\_ == "\_\_main\_\_":

while True:

user\_input\_string = input("\n请输入要分析的表达式 (例如 (a(b(2))(c)) )，或输入 'quit' 退出：\n> ")

if user\_input\_string.lower() == 'quit':

break

if not user\_input\_string:

print("输入不能为空，请重试。")

continue

print(f"\n正在分析输入: \"{user\_input\_string}\"")

try:

tokens = tokenize(user\_input\_string)

print(f"词法分析结果 (Tokens): {tokens}\n")

ll1\_parser(tokens)

except ValueError as ve: # 来自词法分析器的错误

print(f"错误: {ve}")

except Exception as e: # 其他意外错误

print(f"发生了一个意外错误: {e}")

**第二个任务：**  
#define \_CRT\_SECURE\_NO\_WARNINGS 1

#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() {

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

slen = input.size();

}

void getSymbol() {

if (index >= slen) {

symbol = '`';

index++;

return;

}

symbol = input[index];

index++;

}

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;

}

// E0文法的解析

bool E0(int depth) {

start("E", depth);

if (T0(depth + 1)) {

if (E1(depth + 1)) {

success("E");

return true;

}

}

failed("E", depth);

return false;

}

// E1文法的解析

// 在深度为depth的E'部分进行递归下降分析

bool E1(int depth) {

start("E'", depth);

oldIndex = index;

getSymbol();

if (symbol == '+') {

string tmp(1, symbol);

start(tmp, depth + 1);

if (T0(depth)) { // 对T0部分进行递归下降分析

if (E1(depth)) { // 对E'部分进行递归下降分析

success("E'");

return true;

}

}

}

index = oldIndex;

success("E'");

return true;

}

// T0文法的解析

bool T0(int depth) {

start("T", depth);

if (F(depth + 1)) {

if (T1(depth + 1)) {

success("T");

return true;

}

}

failed("T", depth);

return false;

}

// T1文法的解析

// T1函数用于解析语法中的T'产生式

bool T1(int depth) {

start("T'", depth);

oldIndex = index;

getSymbol();

if (symbol == '\*') {

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

if (F(depth + 1)) { // 调用F函数解析F产生式

if (T1(depth + 1)) { // 如果F解析成功，则调用T1函数解析T'产生式

success("T'");

return true;

}

}

}

index = oldIndex;

success("T'");

return true;

}

// F文法的解析

bool F(int depth) {

start("F", depth);

getSymbol();

if (symbol == '(') {

start("(", depth + 1);

if (E0(depth + 1)) { // 调用E0函数并在深度为depth+1的情况下判断结果

getSymbol();

if (symbol == ')') {

start(")", depth + 1);

success("F");

return true;

}

}

failed("F", depth + 1);

return false;

}

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

string tmp(1, symbol);

start(tmp, depth + 1);

success("F");

return true;

}

failed("F", depth + 1);

return false;

}

};

int main() {

PARSER parser;

parser.getToken();

// 调用 E0 方法，并传入参数 1，判断是否符合 LL(1) 文法

if (parser.E0(1)) {

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

}

else {

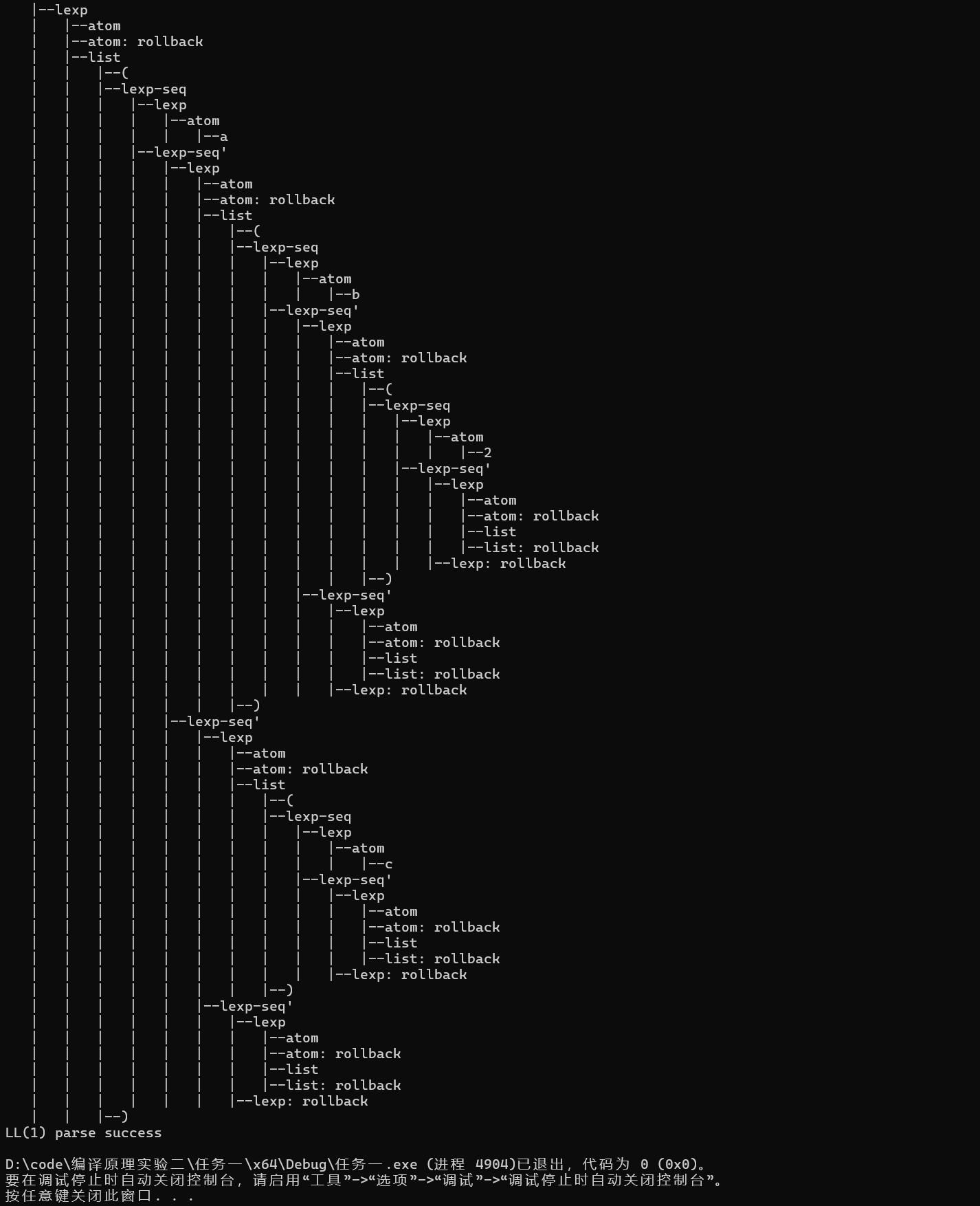
cout << "LL(1) parse failed";

}

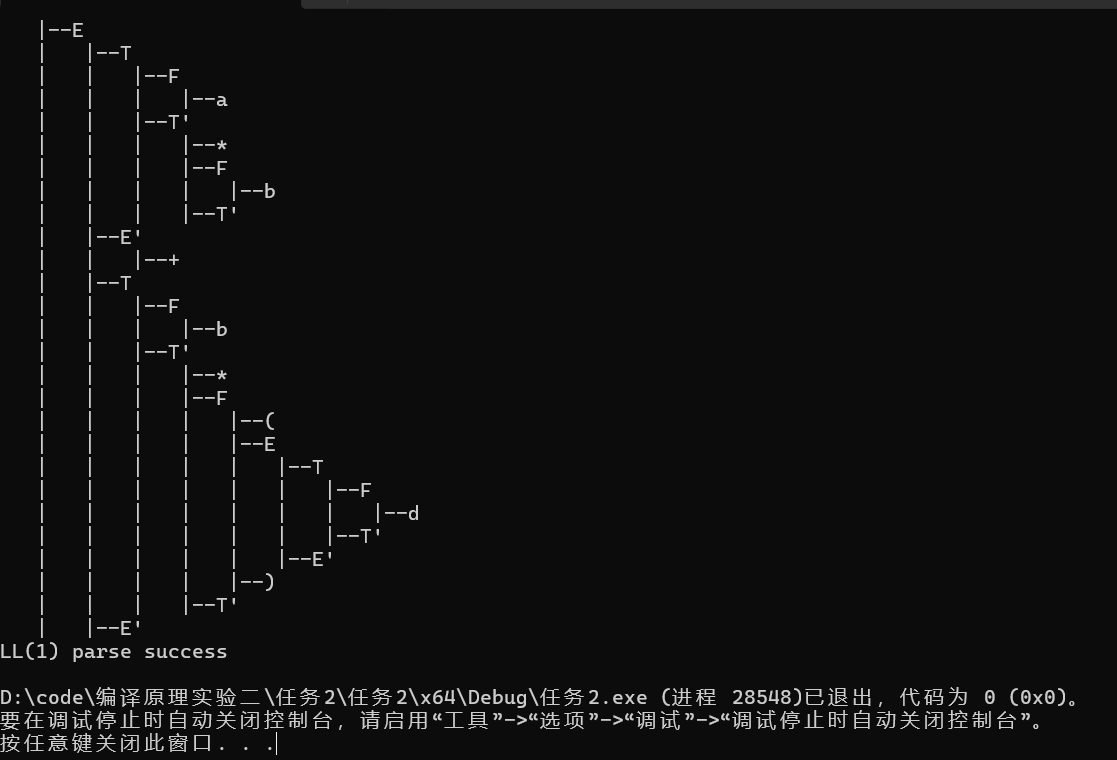
return 0;

}

1. 截图

第一个任务：  


第二个任务：



1. 实验结论:

1 、实验结论

LL(1)语法分析器能准确解析合法括号表达式，并正确判定非法形式。对于复杂嵌套结构（多层括号、混合元素），分析器表现良好。实验数据验证了其解析能力的合理性和准确性。

该结论全面评估了分析器功能，符合实验预期。

2、分析和总结

1. **输入设计**：测试表明，LL(1)语法分析器能有效解析合法和非法的括号表达式，验证了输入设计的全面性。
2. **输出设计**：分析器准确输出了所有输入的解析结果（成功/失败），证明了输出设计的可靠性。
3. **LL(1)分析法**：实验验证了该方法的准确性和鲁棒性，能正确处理各类括号表达式，符合LL(1)文法规范，具备实际可行性。

3、对预估问题的结论

实验验证表明：

1. LL(1)语法分析器能准确解析合法括号表达式
2. 可正确识别非法括号表达式
3. 具备处理复杂嵌套结构的能力
4. 能可靠判定解析结果（成功/失败）
5. 实验数据有效验证了分析器的解析准确性

所有预估问题均得到充分验证，实验设计合理有效。