西安建筑科技大学

实 验 报 告

课程名称：编译原理

实验名称：词法分析程序设计

院 （系）：信控学院

专业班级：计算机 2203

姓　　名：梁桐

学　　号：2209060322

指导教师：叶 娜

2025 年　4 月　22 日

实验二 语法分析程序设计

一、实验目的

理解词法分析器的任务和工作原理；掌握计算机语言语法分析程序的设计方法，并能够针对给定语言的语法规则，使用某种高级编程语言实现其语法分析器。

二、实验内容

对于只含有+、*运算的算术表达式，编写相应的语法分析程序，要求：

1. 用表驱动的预测分析法进行语法分析。
2. 采用某种高级程序设计语言，设计并实现语法分析程序。
3. 设计恰当的测试用例对语法分析程序进行测试。

三、实验设备

计算机、Windows 操作系统、编程语言集成开发环境。

四．实验原理（或程序框图）及步骤

4.1 语法规则与文法描述

4.1.1 语法规则

本实验设计的语言包含以下算术表达式的语法规则：

表达式（Expression）：由项（Term）和可选的加法操作组成。

项（Term）：由因子（Factor）和可选的乘法操作组成。

因子（Factor）：可以是数字、标识符或括号内的表达式。

4.1.2 上下文无关文法（CFG）

根据上述语法规则，构造的上下文无关文法如下：

E → TE'

E' → +TE' | ε

T → FT'

T' → *FT' | ε

F → (E) | num | id

E：表达式

E'：表达式的后续部分

T：项

T'：项的后续部分

F：因子

num：数字

id：标识符

ε：空串

4.1.3 文法分析

该文法不含直接左递归，适用于 LL(1)分析。

4.2LL(1)分析表构造

4.2.1 FIRST 集合

FIRST(E) = { num, id, (}

FIRST(E') = { +, ε }

FIRST(T) = { num, id, (}

FIRST(T') = { *, ε }

FIRST(F) = { num, id, (}

4.2.2 FOLLOW 集合

FOLLOW(E) = {), $ }

FOLLOW(E') = {), $ }

FOLLOW(T) = { +,), $ }

FOLLOW(T') = { +,), $ }

FOLLOW(F) = { *, +,), $ }

4.2.3 预测分析表

非终结符	+	*	(	)	num	id	$
E			E → TE'		E → TE'	E → TE'	
E'	E' → +TE'			E' → ε			E' → ε
T			T → FT'		T → FT'	T → FT'	
T'	T' → ε	T' → *FT'		T' → ε			T' → ε
F			F → (E)		F → num	F → id	

4.3 数据结构设计

4.3.1. 文法存储结构

map<string, vector<vector<string>>> G;

存储方式：使用嵌套容器存储 CFG

Key：非终结符（如"E"、"T'"）

Value：二维字符串向量，表示该非终结符的所有产生式

4.3.2. 集合存储

map<string, set<string>> FIRST, FOLLOW;

FIRST 集：记录每个非终结符能推导出的首符号集合

FOLLOW 集：记录可能跟随非终结符的符号集合

实现特点：

使用 set 保证元素唯一性

Key 为非终结符（如"T"）

Value 为终结符集合（如"num", "id", "("）

4.3.3. 预测分析表

map<string, map<string, vector<string>>> PREDICT;

三级映射结构：

外层 map：非终结符（如"E"）

中层 map：终结符（如"+"）

内层 vector：产生式右部符号序列

查询方式：PREDICT[A][a]表示在非终结符 A 遇到输入符号 a 时应采用的产生式

4.4 特殊字符处理

4.4.1. ε（空串）表示

存储方式：显式使用字符串"ε"

处理逻辑：

```
if (*it != "ε") st.push(*it);   // 压栈时跳过空串
```

4.4.2. 终结符识别

```
inline bool isNonTerminal(const string& s) {
    return !s.empty() && isupper(s[0]);
}
```

判断规则：

首字母大写：非终结符（如"E"）

其他情况：终结符（如"num"、"*"）

4.4.3. 输入符号转换

```
vector<string> tokens;   // 输入 token 序列
```

转换规则：

plaintext

"a+3*(b)" → ["id", "+", "num", "*", "(", "id", ")", "$"]

处理函数：

isDigitChar()识别数字序列为"num"

isLowerChar()识别小写字母为"id"

4.5 输入输出设计

4.5.1. 输入模式

命令行模式示例

输入表达式: a+num*(b+3)$

文件模式

文件处理特点：

逐行读取表达式

自动追加$结束符

支持多行批量分析

4.5.2. 输出形式

表头类型：

栈顶 | 剩余输入 | 动作

输出特性：

三列对齐表格（使用<iomanip>控制格式）

错误信息红色高亮（ANSI 转义码）

最终接受/拒绝状态显示

4.6 错误恢复机制

4.6.1. 错误类型处理

错误类型 恢复策略

终结符不匹配 跳过当前输入符号

预测表项为空 同步到 FOLLOW 集符号

4.6.2. 错误信息显示

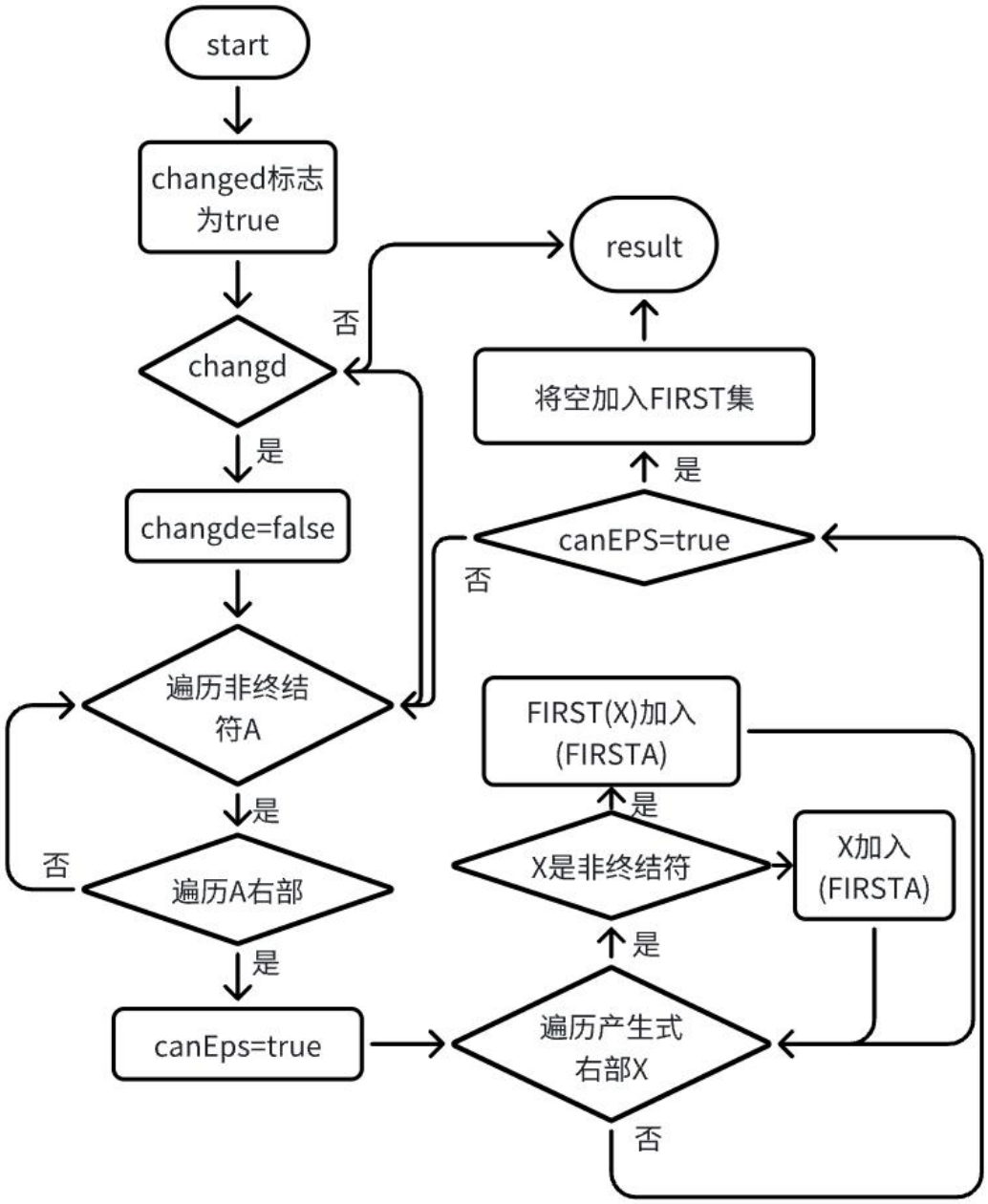
错误位置精确定位

红色文本高亮显示（跨平台兼容）

错误后继续分析流程

4.7 算法流程图

4.7.1 构建 FIRST 集流程



整体流程：

初始化与迭代控制

函数通过 changed 标记控制循环，持续迭代直到所有非终结符的 FIRST 集不再变化。每次迭代开始时重置 changed 为 false，若在后续处理中有新符号加入 FIRST 集，则重新标记 changed 为 true，触发下一轮迭代。

遍历非终结符与产生式

外层循环遍历文法 G 中的每个非终结符 A，并对 A 的每一条产生式 prod 进行分析。例如，对产生式 A→BC，需计算其右部符号串 BC 的 FIRST 集。

逐符号分析右部符号

对产生式右部的每个符号 X 进行处理：

若 X 是非终结符：

将 FIRST[X] 中除 ε 外的所有符号加入 FIRST[A]。若 X 的 FIRST 集不含 ε，则标记 canEps 为 false，终止当前产生式后续符号的分析（因为后续符号无法影响推导结果）。

示例：若 X 是 T 且 FIRST(T)={ (,i }，则将这些符号加入 FIRST(A)。

若 X 是终结符或 ε：

将 X 直接加入 FIRST[A]（ε 需特殊处理），并终止后续符号分析。例如，若 X 是 +，则 FIRST(A) 中加入 +。

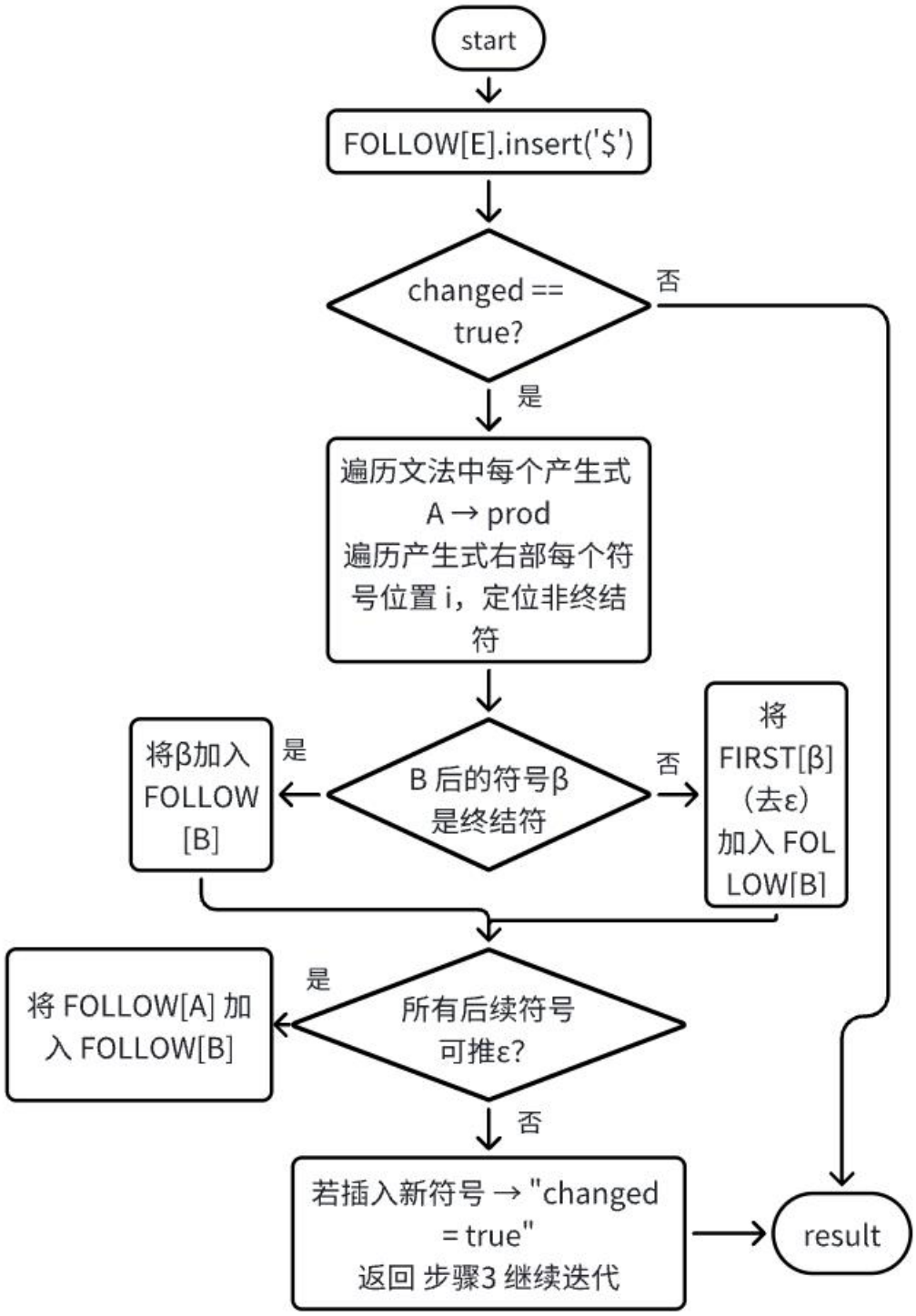
处理 ε 推导能力

若整个产生式右部符号均能推导出 ε（即 canEps 保持为 true），则将 ε 加入 FIRST[A]。例如，若产生式 A→BC 且 B 和 C 的 FIRST 集均包含 ε，则 A 的 FIRST 集需加入 ε。

动态更新标记

每次向 FIRST 集插入新符号时（无论终结符或 ε），均将 changed 设为 true，确保循环持续直到所有集合稳定。

4.7.2 构建 FOLLOW 集流程



主要流程：

函数从初始化起始符号 E 的 FOLLOW 集开始，将结束符 $ 加入其中（规则①：开始符号的 FOLLOW 集必须包含 $）。随后进入迭代循环，反复遍历文法中所有产生式的右部符号，直到没有新的符号能添加到任何非终结符的 FOLLOW 集中为止。

对于每个产生式 A → X_1 X_2 ...X_n ，逐个检查右部符号 X_i：

处理非终结符 B ：若 X_i 是非终结符 B，则分析其后续符号 β = X_{i+1}...X_n ：

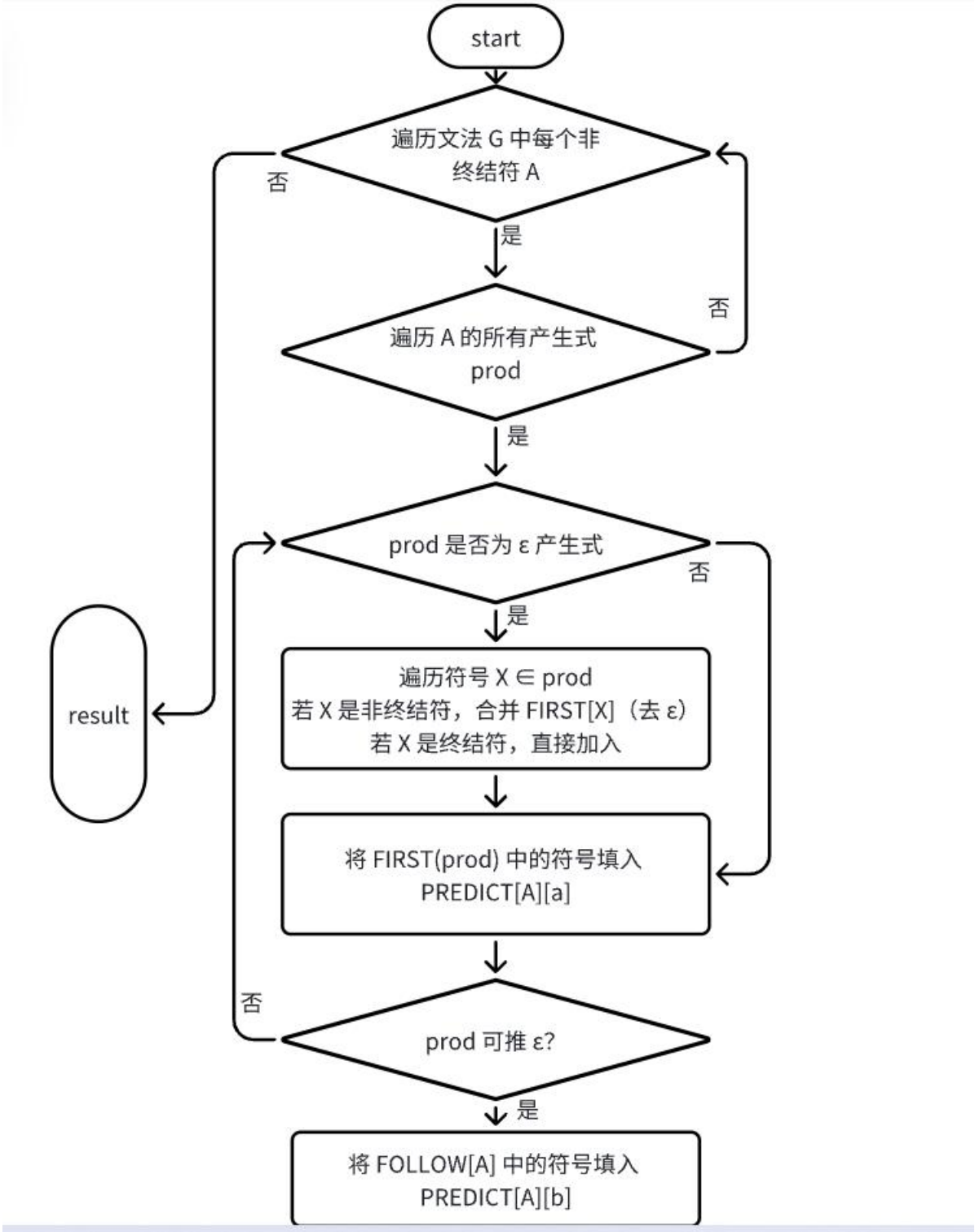
终结符处理 ：若 β 中存在终结符 a，直接将 a 加入 FOLLOW[B]（规则②：终结符直接作为 FOLLOW 元素）。

非终结符处理 ：若 β 中存在非终结符 C，将 FIRST[C] 中除 ε 外的符号加入 FOLLOW[B]（规则②：非终结符的 FIRST 集贡献 FOLLOW 元素）。若 C 的 FIRST 集不含 ε，则停止后续符号分析；否则继续向后传播。

ε 推导链处理 ：若 B 后的所有符号均可推导出 ε（即 canEps = true），则将 FOLLOW[A] 的全部符号加入 FOLLOW[B]（规则③：左部 FOLLOW 集的传播）。

每次向 FOLLOW 集插入新符号时，标记 changed = true 以触发下一轮迭代。当所有 FOLLOW 集稳定后，循环终止。

4.7.3 构建分析表流程



整体流程：

初始化与遍历文法

函数首先遍历文法 G 中的每个非终结符 A，并清空其对应的预测表项。例如，对非终结符 E，初始化 PREDICT["E"] 为空。

处理每条产生式

对 A 的每条产生式 prod（如 E→TE' 或 E'→ε），判断其是否为 ε 产生式（例如 prod 为

["ε"]）。若非 ε 产生式，则计算其右部符号串的 FIRST(prod) 集。

计算 FIRST(prod) 集

遍历产生式右部的每个符号 X：

若 X 是非终结符：将其 FIRST[X] 中的非 ε 符号加入 firstAlpha，并检查 X 是否能推导 ε。若不能，终止后续符号分析（例如 X 是 T 且 FIRST(T) 不含 ε，则停止遍历）。

若 X 是终结符：直接将其加入 firstAlpha，并终止遍历（例如 X 是 +，则 firstAlpha 中加入 +）。

这一过程对应规则“若 $A \rightarrow X_1X_2 \ldots X_n$，则 FIRST(prod) 是 X_1 的 FIRST 集去 ε，若 X_1 可推 ε 则继续合并 X_2 的 FIRST 集，依此类推”。

填充预测表项（非 ε 情况）

将 FIRST(prod) 中的所有符号 a 填入 PREDICT[A][a] 中。例如，产生式 E→TE' 的 FIRST(TE') 为 {(,i}，则 M[E][(] 和 M[E][i] 均填入 E→TE'。

处理 ε 产生式或可推导 ε 的情况

若产生式是 ε（如 E'→ε）或其右部符号均可推导 ε（如 A→BC 且 B 和 C 的 FIRST 集均含 ε），则将 FOLLOW[A] 中的每个符号 b 填入 PREDICT[A][b]。例如，若 FOLLOW(E') 包含 $ 和)，则 M[E'][$] 和 M[E'][)] 填入 E'→ε。

动态更新与迭代

每次插入新表项时，通过 changed 标记触发后续迭代，确保所有可能的符号均被处理，直至预测表稳定。

4.8 程序创新点：

4.8.1. 多模式输入支持：控制台与文件读取灵活切换

程序支持命令行交互输入和文件批量读取两种模式。，用户可直接输入表达式或指定文件路径加载待分析内容。

4.8.2. 错误提示醒目化：红色高亮与精准定位

终端红色错误输出，在语法不匹配、符号未定义等场景下，程序会高亮显示错误位置及预期符号（如网页 1 的错误处理逻辑）。同时，结合 LL(1)分析表的空项检测（参考网页 4 的 M(X,a)查询），动态生成错误类型描述（如“期望运算符，找到标识符”），提升调试效率。

4.8.3. 分析过程透明化：栈操作动态跟踪

通过实时打印分析栈与剩余输入串，程序完整记录每一步推导动作。例如，当栈顶为非终

结符时，输出所选产生式（如 E→TE'）；匹配终结符时标记“符号匹配”。

4.8.4. 智能错误恢复机制：应急处理与短语级修复

程序集成多级错误恢复策略：

应急恢复：跳过非法符号直至遇到 FOLLOW 集元素；

动态同步：通过 FOLLOW 集引导栈弹出，避免分析中断。

4.8.5. 跨平台可移植性：代码封装与一键打包

将程序与依赖库打包为安装包，确保不同设备无需配置环境即可运行。

五. 程序源代码

5.1computeFirst()函数：

```
// 计算 FIRST 集的函数
void computeFirst() {
    bool changed = true;
    // 迭代直到不再有新的符号加入任何 FIRST 集
    while (changed) {
        changed = false;
        // 遍历文法中每个非终结符 A
        for (auto& entry : G) {
            const string& A = entry.first;
            // 遍历 A 的每条产生式 prod
            for (auto& prod : entry.second) {
                bool canEps = true;  // 标记当前产生式是否能推 ε
                // 遍历产生式右部的每个符号 X
                for (auto& X : prod) {
                    if (isNonTerminal(X)) {
                        // 若 X 是非终结符，将 FIRST[X] 中除 ε 外的所有符号加入
FIRST[A]
                        for (auto& a : FIRST[X]) {
                            if (a != "ε" && FIRST[A].insert(a).second) {
                                changed = true;
```

```
                        }
                    }
                    // 若 X 的 FIRST 中不含 ε，则该产生式不能再继续推 ε
                    if (!FIRST[X].count("ε")) {
                        canEps = false;
                        break;
                    }
                }
                else {
                    // X 是终结符或 ε，将其加入 FIRST[A]（ε 也需加入时在后面处理）
                    if (X != "ε" && FIRST[A].insert(X).second) {
                        changed = true;
                    }
                    canEps = false;  // 遇到终结符或 ε，不再继续查看后续符号
                    break;
                }
            }
            // 若整条产生式都能推 ε，则将 ε 加入 FIRST[A]
            if (canEps) {
                if (FIRST[A].insert("ε").second) {
                    changed = true;
                }
            }
        }
    }
}
```

代码分析：

5.1.1. 算法核心逻辑

迭代更新机制

函数通过 while(changed)循环反复遍历文法规则，直到所有 FIRST 集合不再变化（changed 标志位为 false）。这种迭代方式保证了所有可能的符号推导都被覆盖，符合 FIRST 集合需要重复应用规则直至收敛的定义。

非终结符遍历

外层循环 for (auto& entry : G)遍历文法中所有非终结符 A，每个非终结符对应一个产生式集合。这对应规则中“对每个非终结符计算 FIRST 集”的要求。

5.1.2. 单个产生式处理

产生式右部符号分析

内层循环 for (auto& X : prod)逐个分析产生式右部符号。此过程对应规则中的“若 X 是终结符或非终结符，按不同方式处理”的步骤。

终结符处理

当符号 X 是终结符时，直接将其加入当前非终结符 A 的 FIRST 集合（FIRST[A].insert(X)）。若 X 为 ε 则标记该产生式可能推导出空串。此逻辑对应规则①：“终结符的 FIRST 集是其自身”。

非终结符处理

若 X 是非终结符，将 FIRST[X]中除 ε 外的所有符号加入 FIRST[A]。此操作对应规则③：“将非终结符的 FIRST 集传递到左部”。若 FIRST[X]不包含 ε （!FIRST[X].count(″ε″)），则停止分析后续符号（canEps=false），因为后续符号无法影响推导结果。

5.1.3. ε 推导的特殊处理

空串推导标记

变量 canEps 标记当前产生式是否能推导出ε。当所有右部符号的 FIRST 集均包含ε时（例如产生式 T'→ε），canEps 保持为 true，此时将ε加入 FIRST[A]。

整体推导ε的条件

产生式右部所有符号都允许推导出ε时，才会将ε加入当前非终结符的 FIRST 集。这对应规则④：“若所有符号都能推导出空，则左部符号的 FIRST 集包含ε”。

5.1.4. 算法特性

动态更新机制

每次符号插入时检查 insert().second，若成功插入则触发 changed=true，确保算法能感知集合变化。这种机制避免了无限循环，同时保证结果的完备性。

时间复杂度优化

通过立即终止无法推导ε的产生式分析（如遇到终结符或某个非终结符无法推导ε），减少不必要的计算。例如，在 T→FT'中，若 F 无法推导ε，则不再处理 T'的符号。

5.1.5. 与文法规则的对应关系

规则映射

规则②　　：非终结符推导终结符开头的情况，通过直接加入终结符实现；

规则③　　：非终结符传递 FIRST 集，通过 FIRST[X]的合并实现；

规则④/⑤　　：处理连续非终结符推导ε的逻辑，通过 canEps 标记实现。

示例匹配

以文法 E→TE'为例：

T 的 FIRST 集（如{(, id}）被加入 E 的 FIRST 集；

若 T 的 FIRST 集包含ε，则继续处理 E'的 FIRST 集，否则停止。

5.2computeFollow()函数：

```
// 计算 FOLLOW 集的函数
void computeFollow() {
    // 起始符号 E 的 FOLLOW 集包含 '$'
    FOLLOW["E"].insert("$");
    bool changed = true;
    // 迭代直到不再有新的符号加入任何 FOLLOW 集
    while (changed) {
        changed = false;
        // 遍历每个产生式 A -> prod
        for (auto& entry : G) {
            const string& A = entry.first;
            for (auto& prod : entry.second) {
                // 遍历产生式右部中的每个位置 i
                for (size_t i = 0; i < prod.size(); ++i) {
                    const string& B = prod[i];
                    if (!isNonTerminal(B)) continue;  // 只处理非终结符 B
```

```
            bool canEps = true;  // 标记后续符号是否都能推 ε
            // 检查 B 之后的符号 β
            for (size_t j = i + 1; j < prod.size(); ++j) {
                const string& beta = prod[j];
                if (isNonTerminal(beta)) {
                    // 将 FIRST[beta]（去 ε）加入 FOLLOW[B]
                    for (auto& a : FIRST[beta]) {
                        if (a != "ε" && FOLLOW[B].insert(a).second) {
                            changed = true;
                        }
                    }
                    // 若 beta 的 FIRST 中不含 ε，则停止传播
                    if (!FIRST[beta].count("ε")) {
                        canEps = false;
                        break;
                    }
                }
                else {
                    // beta 为终结符，直接加入 FOLLOW[B]
                    if (FOLLOW[B].insert(beta).second) {
                        changed = true;
                    }
                    canEps = false;
                    break;
                }
            }
            // 若 B 后续符号都可推 ε，则将 FOLLOW[A] 全部加入 FOLLOW[B]
            if (canEps) {
                for (auto& f : FOLLOW[A]) {
```

```
                        if (FOLLOW[B].insert(f).second) {
                            changed = true;
                        }
                    }
                }
            }
        }
    }
}
```

代码分析：

5.2.1. 初始化与迭代机制

起始符号初始化

函数首先将文法开始符号 E 的 FOLLOW 集加入特殊符号$，这对应规则①："对于文法开始符号 S，#（或$）属于 FOLLOW(S)"。

动态更新检测

通过 while(changed)循环和 changed 标志位实现迭代，确保所有可能的符号推导都被覆盖。每次迭代若任何集合发生变动则继续循环，直至收敛。

5.2.2. 产生式遍历逻辑

三层嵌套遍历

外层循环遍历所有非终结符 A 的产生式集合（for (auto& entry : G)），中层遍历每条产生式右部（for (auto& prod : entry.second)），内层遍历右部符号位置（for (size_t i = 0; ...)）。这种结构确保所有符号的上下文关系被处理。

非终结符定位

仅处理右部中的非终结符 B（if (!isNonTerminal(B)) continue），符合 FOLLOW 集定义中"非终结符后接符号"的规则。

5.2.3. 后续符号 β 处理

终结符处理

当 beta 是终结符时，直接将其加入 FOLLOW[B]（如 A→ α B β 且 β 为终结符，则 β ∈FOLLOW(B)）。

非终结符处理

若 beta 是非终结符,将 FIRST[beta]中除 ε 外的符号加入 FOLLOW[B]（规则②:"FIRST(β)-{ε}⊆FOLLOW(B)"）。同时通过 canEps 标记判断是否继续检查后续符号（若 beta 的 FIRST 集不含 ε 则终止传播）。

5.2.4. ε 推导的链式传播

全 ε 推导条件

当产生式右部 B 之后的所有符号都能推导 ε 时（canEps 保持为 true），将 FOLLOW[A]全部加入 FOLLOW[B]。这对应规则③:"若 A→αBβ（β⇒*ε）,则 FOLLOW(A)⊆FOLLOW(B)"。

5.2.5. 算法特性与优化

增量更新

使用 insert().second 检测符号是否首次加入集合，避免重复计算。仅在新符号插入时触发 changed=true，提升效率。

剪枝策略

遇到不可推导 ε 的符号（终结符或非终结符 FIRST 集不含 ε）时，立即终止后续符号处理（break），减少无效计算。

5.3buildPredictTable()函数:

```
// 构建 LL(1) 预测分析表 PREDICT
void buildPredictTable() {
    for (auto& entry : G) {
        const string& A = entry.first;
        PREDICT[A].clear();  // 清空旧表项
        for (auto& prod : entry.second) {
            bool prodEps = (prod.size() == 1 && prod[0] == "ε");
            set<string> firstAlpha;
            bool canEps = true;
            // 计算 FIRST(prod)
            if (!prodEps) {
                for (auto& X : prod) {
                    if (isNonTerminal(X)) {
                        for (auto& a : FIRST[X]) {
```

```
                        if (a != "ε") firstAlpha.insert(a);
                    }
                    if (!FIRST[X].count("ε")) { canEps = false; break; }
                }
                else {
                    firstAlpha.insert(X);
                    canEps = false;
                    break;
                }
            }
        }
        // 将 FIRST(prod) 中的符号作为预测符号，填表
        for (auto& a : firstAlpha) {
            PREDICT[A][a] = prod;
        }
        // 若产生式可推 ε，则对 FOLLOW[A] 中的每个 b 也填 ε 产生式
        if (prodEps || canEps) {
            for (auto& b : FOLLOW[A]) {
                PREDICT[A][b] = prod;
            }
        }
    }
  }
}
```

5.3.1. 函数整体逻辑

目标：根据已计算的 FIRST 和 FOLLOW 集，为每个非终结符构建预测分析表 PREDICT，指导语法分析过程。

输入依赖：需提前完成 computeFirst 和 computeFollow 函数，确保 FIRST 和 FOLLOW 集已正确计算。

5.3.2. 遍历非终结符与产生式

非终结符遍历：

通过 for (auto& entry : G) 遍历文法 G 中所有非终结符 A，对应预测分析表的行（如 E、T 等行）。

产生式遍历：

对每个非终结符 A 的所有产生式 prod（如 E→TE’ 或 T→FT’）进行处理，逐个填充表项。

5.3.3. 计算产生式的 FIRST(α)

初始化标记：

prodEps 判断当前产生式是否为 ε（如 B→ε），canEps 标记产生式是否可能推导出 ε。

符号分析：

遍历产生式右部符号 X：

非终结符处理：将 FIRST[X] 中除 ε 外的符号加入 firstAlpha，若 X 的 FIRST 不含 ε，则终止后续符号处理（canEps=false）。

终结符处理：将终结符直接加入 firstAlpha，并终止后续符号分析（canEps=false）。

5.3.4. 填充预测表项

非ε产生式填充：

对于 firstAlpha 中的每个终结符 a，将 A→prod 填入 PREDICT[A][a]（如 FIRST(TE’) 包含 (和 id，则填充 E→TE’ 到 E 行的 (和 id 列）。

ε产生式填充：

若产生式可推导 ε（prodEps 或 canEps 为真），则将 A→prod（即 ε）填入 FOLLOW[A] 中所有终结符对应的列（如 FOLLOW(E’) 包含) 和 #，则填充 E’→ε 到 E’ 行的) 和 # 列）。

5.3.5. 关键算法规则实现

规则映射：

规则①（非ε产生式填充）：a ∈ FIRST(α) ⇒ M[A, a] = A→α，对应代码中的 for (auto& a : firstAlpha) 循环。

规则②（ε产生式填充）：ε ∈ FIRST(α) ⇒ M[A, b] = A→α（b ∈ FOLLOW(A)），对应 if (prodEps || canEps) 后的循环。

冲突检测：

代码隐含要求文法为LL(1)，若同一表项被多次填充（如 M[A, a] 存在多个产生式），则文法非LL(1)，需重构。

5.4parse()函数：

```
// 对给定输入串执行 LL(1) 最左推导，并打印每次栈和输入的状态
// 对给定输入串执行 LL(1) 最左推导，并打印每次栈和输入的状态
void parse(const string& input) {
    stack<string> st;
    st.push("$");  // 底部符号
    st.push("E");  // 起始符号

    // 将输入串转换为 token 序列：num, id 或单字符符号
    vector<string> tokens;
    for (size_t i = 0; i < input.size();) {
        if (isWhitespace(input[i])) { ++i; continue; }
        if (isDigitChar(input[i])) {
            tokens.push_back("num");
            while (i < input.size() && isDigitChar(input[i])) ++i;
        }
        else if (isLowerChar(input[i])) {
            tokens.push_back("id");
            ++i;
        }
        else {
            tokens.push_back(string(1, input[i]));
            ++i;
        }
    }
    if (tokens.empty() || tokens.back() != "$") tokens.push_back("$");

    // 打印表格头
    cout << left << setw(10) << "栈顶" << " | "
```

```
        << right << setw(20) << "剩余输入" << " | "
        << right << setw(10) << "动作" << '\n';
    cout << string(10, '-') << "-+-" << string(20, '-') << "-+-" << string(10, '-')
<< '\n';

    size_t idx = 0;
    bool hasError = false; // 标记是否发生错误
    // 当栈不为空时，重复预测分析过程
    while (!st.empty()) {
        string X = st.top(); st.pop();
        // 构造剩余输入字符串用于打印
        string rem;
        for (size_t j = idx; j < tokens.size(); ++j) rem += tokens[j];

        cout << left << setw(10) << X << " | " << right << setw(20) << rem << " |
";

        // 若栈顶为终结符或 '$'
        if (!isNonTerminal(X)) {
            if (idx < tokens.size() && tokens[idx] == X) {
                cout << right << setw(10) << "匹配" << '\n';
                ++idx;
            }
            else {
                cout << RED_TEXT << right << setw(10) << "错误" << RESET_TEXT << '\n';
                string current_token = (idx < tokens.size()) ? tokens[idx] : "$";
                cerr << RED_TEXT << "语法错误: 期望 '" << X << "', 但找到 '" <<
current_token << "'" << RESET_TEXT << '\n';
                hasError = true;
                // 错误恢复：跳过当前输入符号
```

```
            if (idx < tokens.size()) ++idx;
            // 不压回 X，继续处理栈顶的下一个符号
        }
        continue;
    }

    // 栈顶为非终结符，取当前输入符号 a
    string a = (idx < tokens.size() ? tokens[idx] : "$");
    // 查预测表表项
    if (PREDICT[X].count(a)) {
        auto prod = PREDICT[X][a];
        // 打印产生式动作 X->prod
        string act = X + "->";
        for (auto& s : prod) act += s;
        cout << right << setw(10) << act << '\n';
        // 将产生式右部符号逆序入栈（忽略 ε）
        for (auto it = prod.rbegin(); it != prod.rend(); ++it) {
            if (*it != "ε") st.push(*it);
        }
    }
    else {
        cout << RED_TEXT << right << setw(10) << "错误" << RESET_TEXT << '\n';
        cerr << RED_TEXT << "语法错误: 在 " << X << " 处遇到意外的符号 '" << a
<< "'" << RESET_TEXT << '\n';
        hasError = true;

        // 错误恢复：跳过输入直到遇到 FOLLOW[X] 中的符号
        const auto& follow = FOLLOW[X];
        while (idx < tokens.size()) {
            if (follow.count(tokens[idx])) break;
```

```
                ++idx;
            }
            // X 已被弹出，继续处理栈中的下一个符号
        }
    }

    // 最终检查输入是否完全处理
    if (idx < tokens.size() && tokens[idx] != "$") {
        cout << RED_TEXT << left << setw(10) << "" << " | " << right << setw(20) <<
""
            << " | " << RED_TEXT << right << setw(10) << "错误" << RESET_TEXT << '\n';
        cerr << RED_TEXT << "语法错误: 输入未完全处理，剩余符号 '";
        for (size_t j = idx; j < tokens.size(); ++j) cerr << tokens[j];
        cerr << "'" << RESET_TEXT << '\n';
    }
    else if (!hasError) {
        cout << left << setw(10) << "" << " | " << right << setw(20) << ""
            << " | " << right << setw(10) << "接受" << '\n';
    }
}
```

5.4.1. 初始化与输入预处理

栈初始化

栈初始化为 $（结束符）和起始符号 E，符合 LL(1) 分析器的标准初始化规则。

```
stack<string> st;
st.push("$");  // 结束符
st.push("E");   // 起始符号
```

输入串转换

将输入字符串转换为标准化 token 序列（如 num、id 或单字符符号），过滤空白符并添加结束符 $。例如，输入 id+num*id 会转换为 ["id", "+", "num", "*", "id", "$"]。

```
for (size_t i = 0; i < input.size();) {
```

```
    if (isWhitespace(input[i])) { ++i; continue; }
    if (isDigitChar(input[i])) tokens.push_back("num");
    // ...其他 token 处理逻辑
}
if (tokens.back() != "$") tokens.push_back("$");
```

5.4.2. 表格输出与主循环

表格头打印

输出格式化的表头（栈顶、剩余输入、动作），便于观察每一步分析过程：

```
cout << left << setw(10) << "栈顶" << " | " << right << setw(20) << "剩余输入" <<
" | " << right << setw(10) << "动作" << '\n';
```

主循环逻辑

不断弹出栈顶符号 X，根据其类型（终结符/非终结符）执行不同操作：

```
while (!st.empty()) {
    string X = st.top(); st.pop();
    string rem; // 剩余输入构造
```

5.4.3. 终结符处理与匹配

终结符匹配

若栈顶是终结符，检查是否与当前输入符号匹配：

匹配成功 ：输入指针后移，输出“匹配”。

匹配失败 ：标记错误并触发错误恢复（如跳过当前输入符号）。

```
if (!isNonTerminal(X)) {
    if (idx < tokens.size() && tokens[idx] == X) {
        cout << "匹配"; ++idx;
    } else {
        cerr << "语法错误: 期望 '" << X << "', 但找到 '" << current_token << "'";
        hasError = true;
        if (idx < tokens.size()) ++idx; // 错误恢复：跳过输入符号
    }
}
```

5.4.4. 非终结符处理与预测表查询

预测表查询

若栈顶是非终结符 X，根据当前输入符号 a 查询预测表 PREDICT[X][a]，获取对应的产生式：

存在表项 ：逆序压入产生式右部符号（忽略 ε），输出推导动作（如 E→TE'）。

无表项 ：标记错误，触发错误恢复（如跳过输入直到遇到 FOLLOW[X] 中的符号）。

```
if (PREDICT[X].count(a)) {
    auto prod = PREDICT[X][a];
    cout << X + "->" + 产生式字符串;
    for (auto it = prod.rbegin(); it != prod.rend(); ++it) {
        if (*it != "ε") st.push(*it);
    }
} else {
    cerr << "语法错误: 在 " << X << " 处遇到意外的符号 '" << a << "'";
    // 错误恢复：跳过输入直到 FOLLOW[X] 中的符号
}
```

5.4.5. 错误处理与最终检查

错误恢复机制

终结符不匹配时跳过当前输入符号。

非终结符无表项时跳过输入直到 FOLLOW[X] 中的符号出现。

输入完整性检查

循环结束后检查输入是否完全处理：

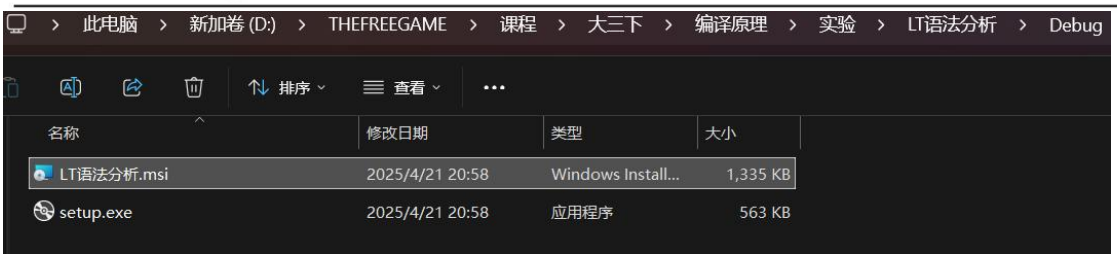
若剩余符号未处理完，输出未完成错误。

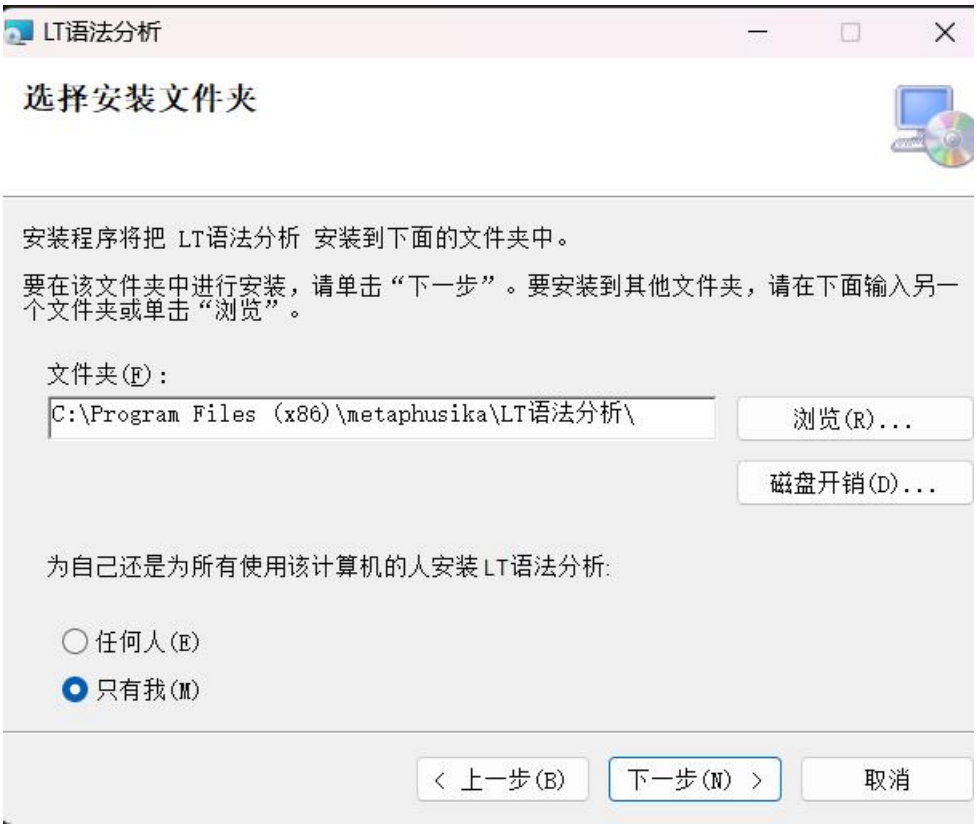
若全部符号处理且无错误，输出“接受”。

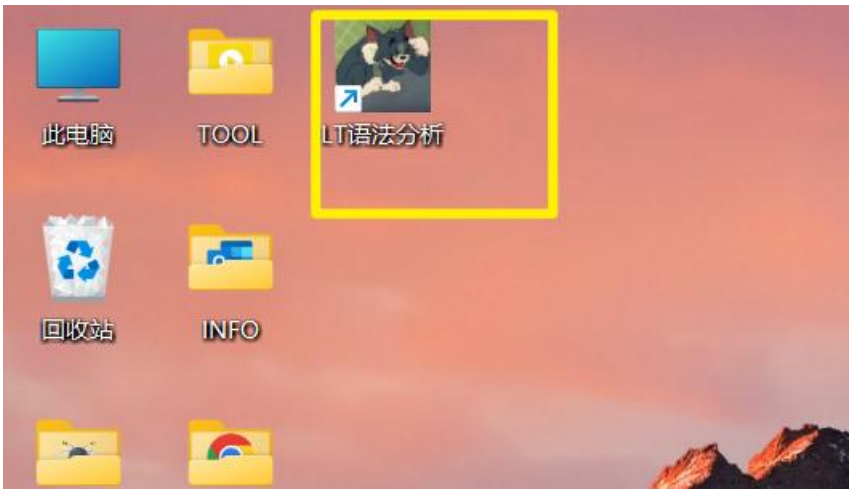
```
if (idx < tokens.size() && tokens[idx] != "$") {
    cerr << "语法错误: 输入未完全处理，剩余符号 '...";
} else if (!hasError) {
    cout << "接受";
}
```

六．实验数据、结果分析

1. 安装程序

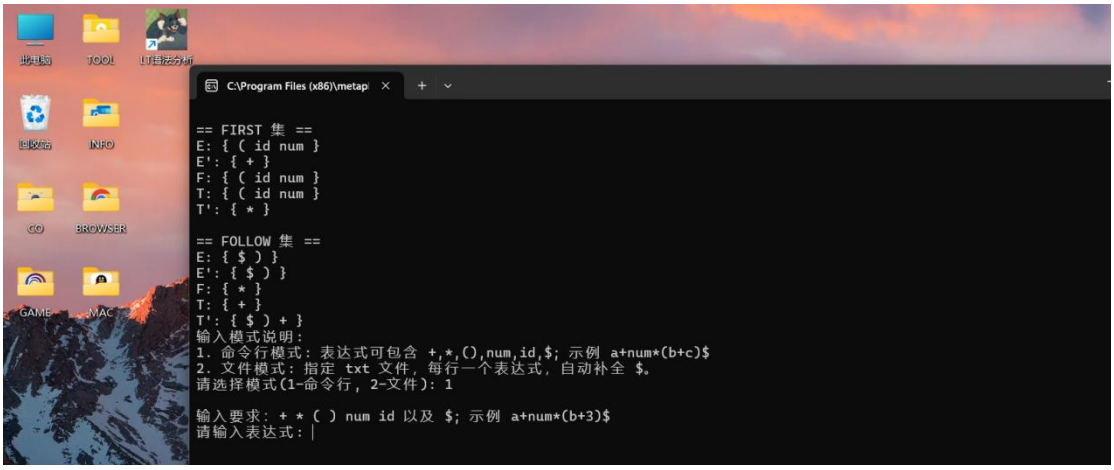




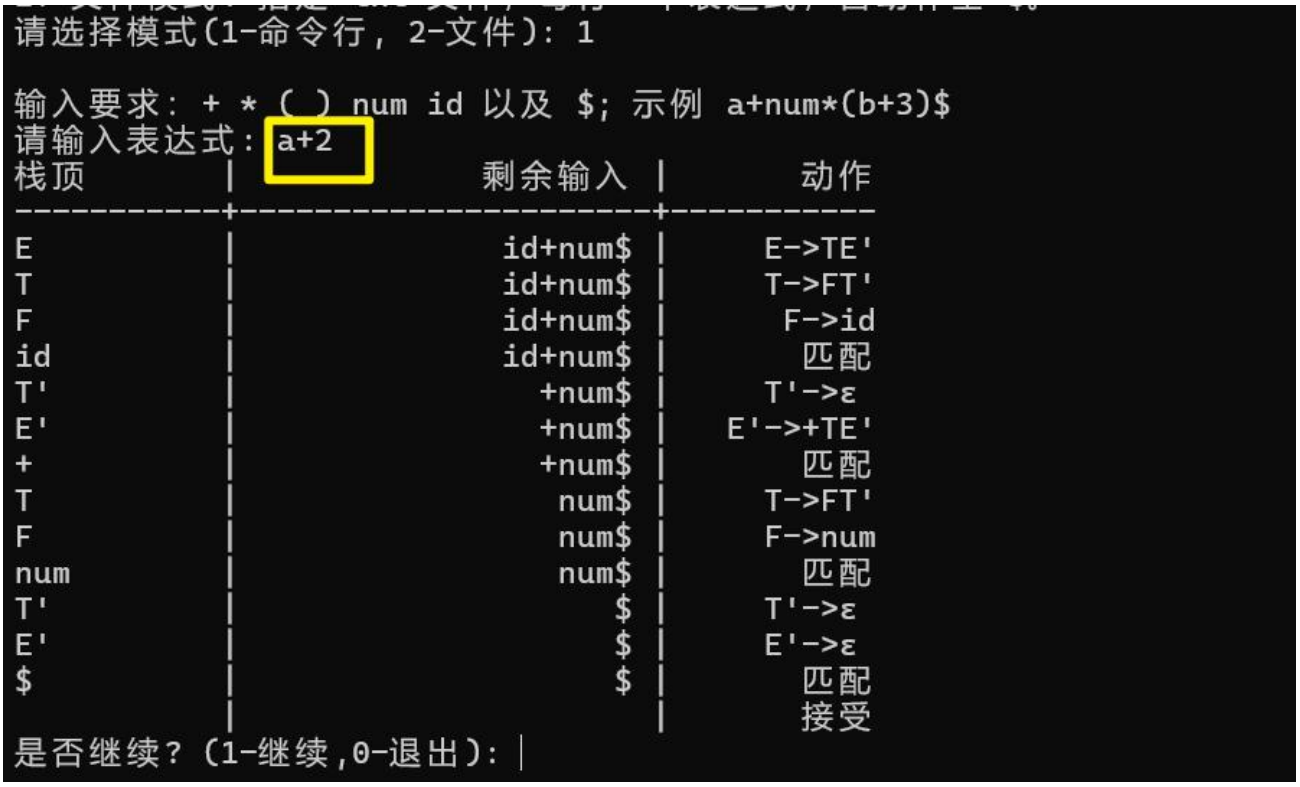


安装成功后桌面会有图标

双击打开程序选择模式 1 测试：



2.1 输入 a+2 测试：



可见运行结果正常。

2.2 输入复杂正确示例

```
输入要求：+ * ( ) num id 以及 $; 示例 a+num*(b+3)$
请输入表达式 b+c+25*(6+m)
栈顶          |                剩余输入 |          动作
--------------+-------------------------+--------------
E             |  id+id+num*(num+id)$ |     E->TE'
T             |  id+id+num*(num+id)$ |     T->FT'
F             |  id+id+num*(num+id)$ |      F->id
id            |  id+id+num*(num+id)$ |        匹配
T'            |    +id+num*(num+id)$ |     T'->ε
E'            |    +id+num*(num+id)$ |   E'->+TE'
+             |    +id+num*(num+id)$ |        匹配
T             |     id+num*(num+id)$ |     T->FT'
F             |     id+num*(num+id)$ |      F->id
id            |     id+num*(num+id)$ |        匹配
T'            |       +num*(num+id)$ |     T'->ε
E'            |       +num*(num+id)$ |   E'->+TE'
+             |       +num*(num+id)$ |        匹配
T             |        num*(num+id)$ |     T->FT'
F             |        num*(num+id)$ |     F->num
num           |        num*(num+id)$ |        匹配
T'            |           *(num+id)$ |   T'->*FT'
*             |           *(num+id)$ |        匹配
F             |            (num+id)$ |     F->(E)
(             |            (num+id)$ |        匹配
E             |             num+id)$ |     E->TE'
T             |             num+id)$ |     T->FT'
F             |             num+id)$ |     F->num
num           |             num+id)$ |        匹配
T'            |                +id)$ |     T'->ε
E'            |                +id)$ |   E'->+TE'
+             |                +id)$ |        匹配
T             |                 id)$ |     T->FT'
F             |                 id)$ |      F->id
id            |                 id)$ |        匹配
T'            |                   )$ |     T'->ε
E'            |                   )$ |     E'->ε
)             |                   )$ |        匹配
T'            |                    $ |     T'->ε
E'            |                    $ |     E'->ε
$             |                    $ |        匹配
```

可见运行结果正常。

2.3 输入错误示例 a++5

```
输入要求: + * ( ) num id 以及 $; 示例 a+num*(b+3)$
请输入表达式: a++5
栈顶        |              剩余输入 |        动作
------------+-----------------------+------------
E           |             id++num$ |      E->TE'
T           |             id++num$ |      T->FT'
F           |             id++num$ |       F->id
id          |             id++num$ |         匹配
T'          |               ++num$ |      T'->ε
E'          |               ++num$ |    E'->+TE'
+           |               ++num$ |         匹配
T           |                +num$ |         错误
语法错误: 在 T 处遇到意外的符号 '+'
E'          |                +num$ |    E'->+TE'
+           |               +num$ |         匹配
T           |                num$ |      T->FT'
F           |                num$ |      F->num
num         |                num$ |         匹配
T'          |                   $ |      T'->ε
E'          |                   $ |      E'->ε
$           |                   $ |         匹配
是否继续? (1-继续,0-退出):
```

红色报错，错误恢复是跳过+。

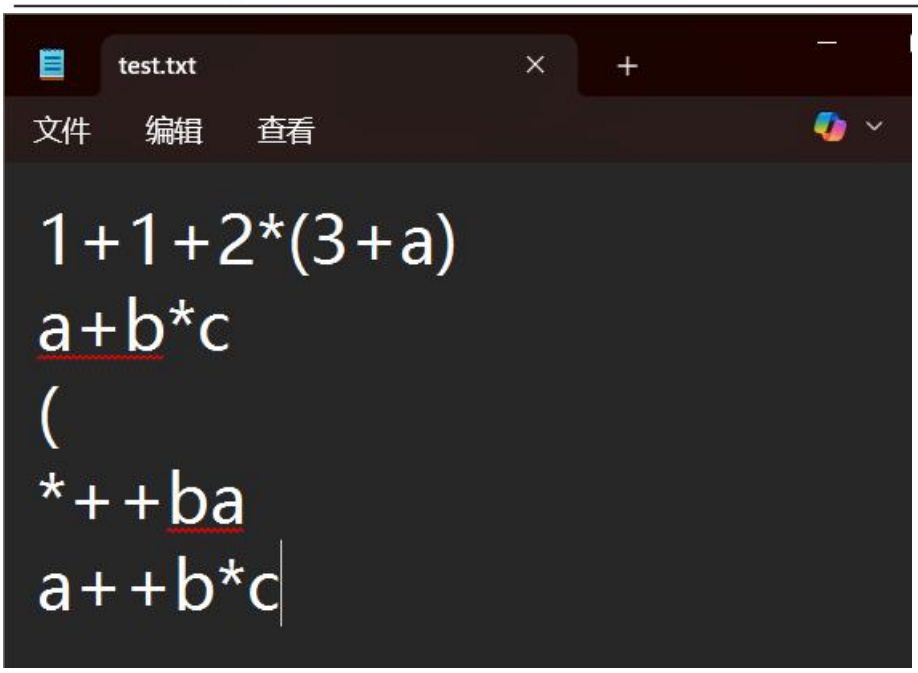
2.4 输入 a))

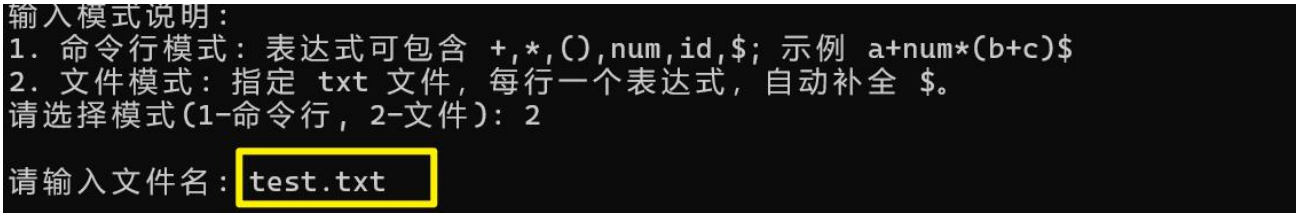
```
输入要求: + * ( ) num id 以及 $; 示例 a+num*(b+3)$
请输入表达式: a))
栈顶        |              剩余输入 |        动作
------------+-----------------------+------------
E           |                id))$ |      E->TE'
T           |                id))$ |      T->FT'
F           |                id))$ |       F->id
id          |                id))$ |         匹配
T'          |                  ))$ |      T'->ε
E'          |                  ))$ |      E'->ε
$           |                  ))$ |         错误
语法错误: 期望 '$', 但找到 ')'
            |                      |         错误
语法错误: 输入未完全处理, 剩余符号 ')$'
是否继续? (1-继续,0-退出):
```

报错清晰正确。

3.1 进行文件读取

文件里提前保存了以下内容：





输入文件名之后将有五个表格输出

3.2

1+1+2*(3+a)正确表达式正确输出

```
-- 第 1 行分析 --
栈顶          |            剩余输入 |       动作
--------------+---------------------+-----------
E             | num+num+num*(num+id)$ |     E->TE'
T             | num+num+num*(num+id)$ |     T->FT'
F             | num+num+num*(num+id)$ |     F->num
num           | num+num+num*(num+id)$ |       匹配
T'            |   +num+num*(num+id)$ |      T'->ε
E'            |   +num+num*(num+id)$ |   E'->+TE'
+             |   +num+num*(num+id)$ |       匹配
T             |    num+num*(num+id)$ |     T->FT'
F             |    num+num*(num+id)$ |     F->num
num           |    num+num*(num+id)$ |       匹配
T'            |       +num*(num+id)$ |     T'->ε
E'            |       +num*(num+id)$ |   E'->+TE'
+             |       +num*(num+id)$ |       匹配
T             |        num*(num+id)$ |     T->FT'
F             |        num*(num+id)$ |     F->num
num           |        num*(num+id)$ |       匹配
T'            |           *(num+id)$ |   T'->*FT'
*             |           *(num+id)$ |       匹配
F             |            (num+id)$ |     F->(E)
(             |            (num+id)$ |       匹配
E             |             num+id)$ |     E->TE'
T             |             num+id)$ |     T->FT'
F             |             num+id)$ |     F->num
num           |             num+id)$ |       匹配
T'            |                +id)$ |     T'->ε
E'            |                +id)$ |   E'->+TE'
+             |                +id)$ |       匹配
T             |                 id)$ |     T->FT'
F             |                 id)$ |      F->id
id            |                 id)$ |       匹配
T'            |                   )$ |     T'->ε
E'            |                   )$ |     E'->ε
)             |                   )$ |       匹配
T'            |                    $ |     T'->ε
E'            |                    $ |     E'->ε
$             |                    $ |       匹配
              |                      |       接受
```

a+b*c 正确表达式正确输出

```
-- 第 2 行分析 --
栈顶          |              剩余输入 |       动作
--------------+----------------------+-----------
E             |           id+id*id$ |     E->TE'
T             |           id+id*id$ |     T->FT'
F             |           id+id*id$ |      F->id
id            |           id+id*id$ |        匹配
T'            |             +id*id$ |     T'->ε
E'            |             +id*id$ |   E'->+TE'
+             |             +id*id$ |        匹配
T             |              id*id$ |     T->FT'
F             |              id*id$ |      F->id
id            |              id*id$ |        匹配
T'            |                *id$ |   T'->*FT'
*             |                *id$ |        匹配
F             |                 id$ |      F->id
id            |                 id$ |        匹配
T'            |                   $ |     T'->ε
E'            |                   $ |     E'->ε
$             |                   $ |        匹配
              |                     |        接受
```

(：错误表达式，相应报错

```
-- 第 3 行分析 --
栈顶          |                      剩余输入 |       动作
--------------+-------------------------------+-----------
E             |                           ($ |     E->TE'
T             |                           ($ |     T->FT'
F             |                           ($ |     F->(E)
(             |                           ($ |        匹配
E             |                            $ |        错误
语法错误：在 E 处遇到意外的符号 '$'
)             |                            $ |        错误
语法错误：期望 ')'，但找到 '$'
T'            |                              |     T'->ε
E'            |                              |     E'->ε
$             |                              |        错误
语法错误：期望 '$'，但找到 '$'
```

*++ba 错误表达式，相应报错

```
-- 第 4 行分析 --
栈顶          |              剩余输入 |        动作
--------------+-----------------------+-----------
E             |              *++idid$ |        错误
语法错误：在 E 处遇到意外的符号 '*'
$             |                    $ |        匹配
```

a++b*c 错误表达式，相应报错

```
-- 第 5 行分析 --
栈顶          |              剩余输入 |        动作
--------------+-----------------------+-----------
E             |           id++id*id$ |     E->TE'
T             |           id++id*id$ |     T->FT'
F             |           id++id*id$ |      F->id
id            |           id++id*id$ |        匹配
T'            |             ++id*id$ |     T'->ε
E'            |             ++id*id$ |   E'->+TE'
+             |             ++id*id$ |        匹配
T             |              +id*id$ |        错误
语法错误：在 T 处遇到意外的符号 '+'
E'            |              +id*id$ |   E'->+TE'
+             |              +id*id$ |        匹配
T             |               id*id$ |     T->FT'
F             |               id*id$ |      F->id
id            |               id*id$ |        匹配
T'            |                 *id$ |   T'->*FT'
*             |                 *id$ |        匹配
F             |                  id$ |      F->id
id            |                  id$ |        匹配
T'            |                    $ |     T'->ε
E'            |                    $ |     E'->ε
$             |                    $ |        匹配
```

4. 退出程序

```
是否继续？(1-继续,0-退出): 0
```

按 0 退出，进程释放。

七. 实验小结

7.1 实验遇到的困难和解决方案

实验报告：LL(1)语法分析器开发中遇到的错误与解决方法

错误 1：括号嵌套不匹配导致分析栈操作异常

现象：在解析形如 (a + (b * c) 的表达式时，分析器因缺少闭合右括号导致栈顶符号与输入符号不匹配，触发语法错误。

原因：文法的括号匹配规则未覆盖所有嵌套场景，且未正确处理嵌套层级。

解决：F → (E) | num | id 修改文法，显式定义括号嵌套规则。在分析表中为 (和) 分别绑定对应的移进和规约操作，确保栈中左括号与输入右括号严格匹配。

错误 2：分析栈初始化错误导致空栈异常

现象：程序启动后立即崩溃，报错 segmentation fault，调试发现分析栈未初始化。

原因：未将起始符号（如 E）和结束符 $ 压入分析栈，导致后续的栈操作（如 pop 或 top）访问非法内存。

解决

```
// 初始化分析栈
analyseStack[0] = '$';   // 栈底结束符
analyseStack[1] = 'E';   // 起始符号
stackTop = 2;                    // 栈顶指针初始化
```

错误 3：错误恢复未正确处理 FOLLOW 集导致死循环

现象：输入 a +) 时，分析器在跳过) 后仍无法匹配 $，陷入死循环。

原因：错误恢复逻辑未将 FOLLOW(E) 的符号（如 $ 和)）作为同步字符，导致无法跳出错误状态。

解决：

在错误处理函数中引入 FOLLOW 集同步机制：

```
void error_recovery() {
    while (current_token != '$' && !is_in_FOLLOW(top_symbol())) {
        advance_input();   // 跳过非法符号直至遇到 FOLLOW 集元素
    }
    pop_stack();   // 弹出栈顶无法处理的非终结符
}
```

7.2 实验心得

通过本次 LL(1)语法分析器的开发实验，我系统掌握了编译原理的核心理论与工程实践方法。在理论层面，深入理解了 LL(1)文法的构造规则，通过消除左递归、提取左公因子构建无冲突的预测分析表，并基于 FIRST/FOLLOW 集实现符号推导的逻辑验证；在编码实践中，利用栈结构模拟最左推导过程，通过动态打印栈与输入串状态实现分析过程的可视化追踪。这次实践让我深刻体会到理论代码化的挑战与乐趣，为后续编译技术探索奠定了扎实基础。