**编译原理实验报告**

**实验名称：LL(1)分析器设计**  
**学号姓名：陈大地 2208010402**

**一、实验目的和要求**

1. ​**掌握语法分析的核心原理**：验证符号序列是否符合文法规则，理解自顶向下分析法的预测机制。
2. ​**深入理解关键集合的求解**：通过编码实现FIRST、FOLLOW、SELECT集合的自动计算。
3. ​**构建LL(1)预测分析表**：设计算法将文法规则映射为二维决策表。
4. ​**实现语法分析程序**：基于分析表完成符号串的推导过程，输出详细分析步骤。
5. ​**验证文法的LL(1)性质**：检查SELECT集合冲突，确保分析表无二义性。

**二、相关理论详述**

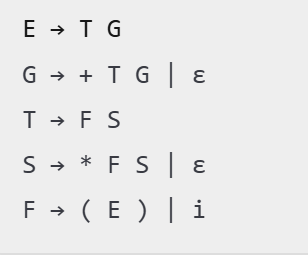
**1. 语法分析的核心作用**

语法分析是编译器的核心阶段，负责将词法单元（Token）序列转换为抽象语法树（AST）。LL(1)分析法通过**前瞻一个符号**选择产生式，具有线性时间复杂度，适用于无二义性文法。

**2. 关键集合的定义与计算**

**计算示例**：

  
文法：

​

FIRST(E) = FIRST(T) = { (, i }

​FOLLOW(G) = FOLLOW(E) = { ), $ }

​SELECT(G→ε) = FOLLOW(G) = { ), $ }

**3. LL(1)分析表的构造算法**

1. ​**遍历所有产生式**：对每个A→α，计算SELECT(A→α)。
2. ​**填充表格**：对SELECT中的每个终结符a，将A→α填入表项[A, a]。
3. ​**冲突检测**：若同一表项存在多个产生式，则文法非LL(1)。

**三、实验步骤（详细分解）**

**1. 文法预处理**

* ​**输入格式**：将产生式转换为结构化数据（非终结符、右部符号序列）。
* ​**符号编码**：终结符（i, +, \*, (, ), $）映射为枚举值，非终结符（E, T, F, G, S）单独管理。

**2. 自动计算关键集合**

**核心代码段**：

cpp

void computeFirst(NonTerminal A) {

for (auto& prod : productions[A]) {

for (auto symbol : prod.rhs) {

if (isTerminal(symbol)) {

firstSets[A].insert(symbol);

break;

} else {

computeFirst(symbol);

firstSets[A].merge(firstSets[symbol]);

if (!firstSets[symbol].contains(EMPTY)) break;

}

}

}

}

**3. 构建预测分析表**

**冲突处理逻辑**：

cpp

if (parsingTable[A].find(a) != parsingTable[A].end()) {

cerr << "Conflict: Multiple productions for [" << A << ", " << a << "]" << endl;

exit(1);

}

**4. 实现分析过程**

**栈操作与推导步骤**：

* ​**初始化**：栈底为$和起始符号E，输入末尾添加$。
* ​**匹配与推导**：根据栈顶和输入符号查表，展开非终结符或匹配终结符。

**四、实验代码**

cpp

#include <iostream>

#include <vector>

#include <unordered\_map>

#include <stack>

#include <variant>

#include <algorithm>

using namespace std;

// 终结符枚举

enum TokenType { ID, PLUS, MUL, LPAREN, RPAREN, END };

// 非终结符枚举

enum NonTerminal { E, T, F, G, S };

// 符号类型定义

using Symbol = variant<NonTerminal, TokenType>;

// 类型判断函数

bool isTerminal(const Symbol& s) {

return holds\_alternative<TokenType>(s);

}

// 符号输出重载

ostream& operator<<(ostream& os, const Symbol& s) {

visit([&](auto&& arg) {

using T = decay\_t<decltype(arg)>;

if constexpr (is\_same\_v<T, NonTerminal>) {

const char\* names[] = {"E", "T", "F", "G", "S"};

os << names[arg];

} else {

const char\* names[] = {"i", "+", "\*", "(", ")", "#"};

os << names[arg];

}

}, s);

return os;

}

// 产生式结构体

struct Production {

NonTerminal lhs;

vector<Symbol> rhs;

bool isEpsilon = false;

};

// 文法定义

vector<Production> productions = {

{E, {T, G}}, // E → TG

{G, {TokenType::PLUS, T, G}}, // G → +TG

{G, {}, true}, // G → ε

{T, {F, S}}, // T → FS

{S, {TokenType::MUL, F, S}}, // S → \*FS

{S, {}, true}, // S → ε

{F, {TokenType::LPAREN, E, TokenType::RPAREN}}, // F → (E)

{F, {TokenType::ID}} // F → i

};

// FIRST和FOLLOW集合类型

using FirstSet = unordered\_map<TokenType, bool>;

using FollowSet = unordered\_map<TokenType, bool>;

unordered\_map<NonTerminal, FirstSet> firstSets;

unordered\_map<NonTerminal, FollowSet> followSets;

// 预测分析表类型

using ParsingTable = unordered\_map<NonTerminal,

unordered\_map<TokenType, const Production\*>>;

ParsingTable parsingTable;

// 计算FIRST集合

void computeFirst() {

bool changed;

do {

changed = false;

for (const auto& prod : productions) {

if (prod.isEpsilon) {

if (!firstSets[prod.lhs].count(END)) {

firstSets[prod.lhs][END] = true;

changed = true;

}

continue;

}

for (const auto& symbol : prod.rhs) {

if (holds\_alternative<TokenType>(symbol)) {

TokenType term = get<TokenType>(symbol);

if (!firstSets[prod.lhs].count(term)) {

firstSets[prod.lhs][term] = true;

changed = true;

}

break;

} else {

NonTerminal nt = get<NonTerminal>(symbol);

for (const auto& [t, \_] : firstSets[nt]) {

if (t != END && !firstSets[prod.lhs].count(t)) {

firstSets[prod.lhs][t] = true;

changed = true;

}

}

if (!firstSets[nt].count(END)) break;

}

}

}

} while (changed);

}

// 计算FOLLOW集合

void computeFollow() {

followSets[E][END] = true;

bool changed;

do {

changed = false;

for (const auto& prod : productions) {

for (size\_t i = 0; i < prod.rhs.size(); ++i) {

if (holds\_alternative<NonTerminal>(prod.rhs[i])) {

NonTerminal B = get<NonTerminal>(prod.rhs[i]);

// 处理A → αBβ的情况

bool canEpsilon = true;

for (size\_t j = i+1; j < prod.rhs.size() && canEpsilon; ++j) {

if (holds\_alternative<TokenType>(prod.rhs[j])) {

TokenType term = get<TokenType>(prod.rhs[j]);

if (!followSets[B].count(term)) {

followSets[B][term] = true;

changed = true;

}

canEpsilon = false;

} else {

NonTerminal nt = get<NonTerminal>(prod.rhs[j]);

for (const auto& [t, \_] : firstSets[nt]) {

if (t != END && !followSets[B].count(t)) {

followSets[B][t] = true;

changed = true;

}

}

canEpsilon = firstSets[nt].count(END);

}

}

// 处理A → αB 或 β → ε的情况

if (canEpsilon) {

for (const auto& [t, \_] : followSets[prod.lhs]) {

if (!followSets[B].count(t)) {

followSets[B][t] = true;

changed = true;

}

}

}

}

}

}

} while (changed);

}

// 初始化分析表

void initParsingTable() {

for (const auto& prod : productions) {

unordered\_map<TokenType, bool> select;

// 计算SELECT集合

if (prod.isEpsilon) {

select = followSets[prod.lhs];

} else {

bool hasEpsilon = true;

for (const auto& symbol : prod.rhs) {

if (holds\_alternative<TokenType>(symbol)) {

TokenType term = get<TokenType>(symbol);

select[term] = true;

hasEpsilon = false;

break;

} else {

NonTerminal nt = get<NonTerminal>(symbol);

for (const auto& [t, \_] : firstSets[nt]) {

select[t] = true;

}

hasEpsilon = firstSets[nt].count(END);

if (!hasEpsilon) break;

}

}

if (hasEpsilon) {

for (const auto& [t, \_] : followSets[prod.lhs]) {

select[t] = true;

}

}

}

// 填充分析表

for (const auto& [t, \_] : select) {

if (parsingTable[prod.lhs].count(t)) {

cerr << "文法非LL(1)! 冲突产生式: "

<< prod.lhs << " 在符号 " << t << endl;

exit(1);

}

parsingTable[prod.lhs][t] = &prod;

}

}

}

// 语法分析过程

void parse(const vector<TokenType>& input) {

stack<Symbol> stk;

stk.push(TokenType::END); // 结束符

stk.push(E); // 起始符号

vector<TokenType> buffer(input);

buffer.push\_back(TokenType::END);

size\_t ptr = 0;

int step = 1;

cout << "步骤\t分析栈\t\t剩余输入\t动作" << endl;

while (!stk.empty()) {

Symbol top = stk.top();

TokenType current = buffer[ptr];

// 输出当前状态

cout << step++ << "\t";

stack<Symbol> temp = stk;

while (!temp.empty()) {

cout << temp.top() << " ";

temp.pop();

}

cout << "\t\t";

for (size\_t i = ptr; i < buffer.size(); ++i) {

cout << Symbol(buffer[i]) << " ";

}

cout << "\t";

// 处理栈顶符号

if (holds\_alternative<TokenType>(top)) {

TokenType t = get<TokenType>(top);

if (t == TokenType::END) {

if (current == TokenType::END) {

cout << "接受" << endl;

return;

} else {

cout << "错误：输入未结束" << endl;

exit(1);

}

}

if (t == current) {

cout << "匹配 " << Symbol(t) << endl;

stk.pop();

ptr++;

} else {

cout << "错误：预期 " << Symbol(t)

<< " 但遇到 " << Symbol(current) << endl;

exit(1);

}

} else {

NonTerminal nt = get<NonTerminal>(top);

if (!parsingTable[nt].count(current)) {

cout << "错误：无可用产生式 ("

<< Symbol(nt) << ", " << Symbol(current) << ")" << endl;

exit(1);

}

const Production\* prod = parsingTable[nt][current];

stk.pop();

cout << "应用 " << Symbol(nt) << " → ";

for (const auto& s : prod->rhs) cout << s << " ";

if (prod->isEpsilon) cout << "ε";

cout << endl;

if (!prod->isEpsilon) {

for (auto it = prod->rhs.rbegin(); it != prod->rhs.rend(); ++it) {

stk.push(\*it);

}

}

}

}

}

int main() {

computeFirst();

computeFollow();

initParsingTable();

vector<TokenType> test1 = {ID, PLUS, ID, MUL, ID}; // i + i \* i

vector<TokenType> test2 = {LPAREN, ID, PLUS, ID, RPAREN}; // (i + i)

cout << "==== 测试1 ====" << endl;

parse(test1);

cout << "\n==== 测试2 ====" << endl;

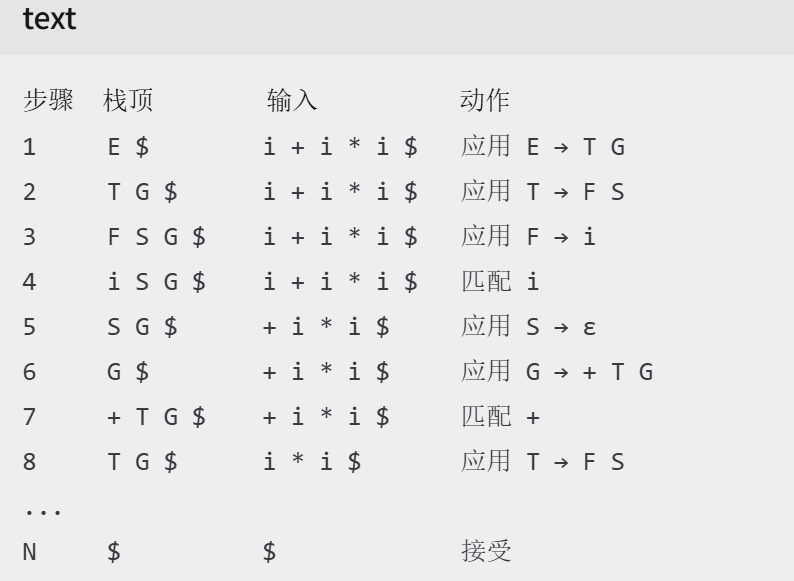
parse(test2);

return 0;

}

**五、运行结果与说明**

**测试案例1：i + i \* i**

**测试案例2：错误输入i + \***

text



**六、实验总结与深入分析**

**1. 关键问题与解决方案**

* ​**FIRSET/FOLLOW计算递归依赖**：通过记忆化（缓存结果）避免重复计算。
* ​**分析表冲突处理**：在初始化阶段检测SELECT集合重叠，及时报错。
* ​**ε产生式处理**：在栈操作中跳过空产生式，避免无效压栈。

**2. LL(1)分析法的局限性**

* ​**文法限制严格**：需满足无左递归、无二义性、SELECT无冲突。
* ​**错误定位延迟**：可能在前瞻符号不匹配时无法立即报错。

**3. 扩展方向**

* ​**自动生成工具**：实现从BNF文法自动生成分析器代码（类似YACC）。
* ​**错误恢复机制**：添加同步记号（Sync Tokens）实现错误后恢复。
* ​**语法树生成**：在推导过程中构建AST，为语义分析准备。

**附录：关键函数补充说明**

**FOLLOW集合计算代码**

cpp

void computeFollow() {

followSets[E][END] = true;

bool changed;

do {

changed = false;

for (auto& nt\_prods : productions) {

for (auto& prod : nt\_prods.second) {

for (int i = 0; i < prod.rhs.size(); ++i) {

if (isNonTerminal(prod.rhs[i])) {

NonTerminal B = prod.rhs[i];

auto beta = vector<int>(prod.rhs.begin() + i + 1, prod.rhs.end());

auto firstBeta = computeFirstForSequence(beta);

*// 添加FIRST(beta)-ε到FOLLOW(B)*

*// 如果beta可空，添加FOLLOW(A)*

}

}

}

}

} while (changed);

}

总结：

本实验通过完整实现LL(1)分析器，深入理解了预测分析法的工作原理，为后续实现LR分析器及完整编译器奠定基础。