北京郵電大學

设计报告

课程名称: 编译原理与技术
实验名称: 语法分析
学　　院: 计算机学院
班　　级: 2022211312
学　　号: 2022211404
姓　　名: 唐梓楠

2024 年 *12* 月 *7* 日

目录

语法分析设计报告

唐梓楠

2024 年 12 月 7 日

1 编写 LL(1) 语法分析程序

1.1 题目要求

1.1.1 实验背景

来自《编译原理与技术（第 2 版）》第 4 章的**程序设计 2**（方法 2）。

1.1.2 任务描述

编写一个 LL(1) 语法分析程序，能对算数表达式进行语法分析。

要求 在给定的消除左递归的文法 G' 的基础上：

编号	产生式
1	$E \to TA$
2	$A \to +TA$
3	$A \to -TA$
4	$A \to$
5	$T \to FB$
6	$B \to *FB$
7	$B \to /FB$
8	$B \to \varepsilon$
9	$F \to (E)$
10	$F \to \text{num}$

1. 编程实现算法 4.2，为给定文法自动构造预测分析表。
2. 编程实现算法 4.1，构造 LL(1) 预测分析程序。

1.1.3 编程要求

根据注释提示，补充代码。

要求 使用 C/C++ 实现

平台环境说明

- 编译器版本：gcc `7.3.0`
- OS 版本：Debian GNU/Linux `9`

1.1.4 测试说明

输入格式 从标准输入（使用 `cin`/`scanf` 等输入）读入数据。
输入仅包含 1 行:

- 第 1 行输入为一个算术表达式。构成该算术表达式的字符有: {'`n`', '`+`', '`-`', '`*`', '`/`', '`(`', '`)`'}。

输出格式 输出到标准输出（使用 `cout`/`printf` 等输出）中。
输出包括若干行 LL(1) 分析过程:
假设输出有 n 行，则第 i 行 ($1 \leq i \leq n$) 表示分析进行到第 i 步，它的输出包含制表符 ('`\t`') 分隔的三个部分:

- 分析栈：以 `$` 符号表示栈底的字符串（左侧为栈底；由终结符和非终结符构成）
- 输入串栈：以 `$` 符号表示栈底的字符串（右侧为栈底）
- 分析动作：产生式编号，或 `match` 或 `error` 或 `accept`（表示当前步骤应执行的动作）

1.2 程序设计说明

- `prods`：映射产生式编号到（左部, 右部）。
- `grammar`：记录每个非终结符号的多个产生式右部，用于求 FIRST 和 FOLLOW 集。
- `FIRSTSets`、`FOLLOWSets`：储存每个非终结符号的 FIRST 和 FOLLOW 集。
- `M`：预测分析表，$M[A][a]$ 给出在 X 为非终结符 A 且当前输入符号为 a 时应选用的产生式编号。
- `input_line`：存储输入串。

- 使用 `@` 表示 ε。

1.2.1 读入输入串

在主函数中从标准输入读取一行字符串作为待分析的算术表达式，然后在末尾添加结束符号 `$`。

```
    std::string input_line;
    std::getline(std::cin, input_line);
    input_line.push_back('$');
```

1.2.2 构建文法

利用 `buildGrammar()` 函数，将产生式存入 `grammar` 字典中，并将每条产生式存入 `prods` 映射中以便后续查表；同时设置非终结符与终结符集合。

```
void buildGrammar()
{
    grammar['E'].emplace_back("TA");
    grammar['A'].emplace_back("+TA");
    grammar['A'].emplace_back("-TA");
    grammar['A'].emplace_back(""); //
    grammar['T'].emplace_back("FB");
    grammar['B'].emplace_back("*FB");
    grammar['B'].emplace_back("/FB");
    grammar['B'].emplace_back(""); //
    grammar['F'].emplace_back("(E)");
    grammar['F'].emplace_back("n");

    nonterminals = { 'E', 'A', 'T', 'B', 'F' };
    terminals = { 'n', '+', '-', '*', '/', '(', ')', '$' };

    prods[1] = std::make_pair('E', "TA");
    prods[2] = std::make_pair('A', "+TA");
```

```
    prods[3] = std::make_pair('A', "-TA");
    prods[4] = std::make_pair('A', "");
    prods[5] = std::make_pair('T', "FB");
    prods[6] = std::make_pair('B', "*FB");
    prods[7] = std::make_pair('B', "/FB");
    prods[8] = std::make_pair('B', "");
    prods[9] = std::make_pair('F', "(E)");
    prods[10] = std::make_pair('F', "n");
}
```

1.2.3 求 FIRST 集

调用 buildFIRST() 函数，利用 FIRST_str() 函数为每个非终结符号计算 FIRST 集。其中 FIRST_str() 用于计算任意符号串的 FIRST 集。

- 若首个符号是终结符，FIRST 集即为该终结符本身。
- 若首个符号是非终结符，则将该非终结符的 FIRST 集（除 ε 之外的符号）加入结果中；如果包含 ε，则继续处理下一个符号。
- 若最终可推导出 ε，则在 FIRST 中加入 @。

重复计算直到 FIRST 集不再变化。

```
std::set<char> FIRST_str(const std::string& str)
{
    std::set<char> result;
    if (str.empty()) {
        result.insert('@');
        return result;
    }
    for (int i = 0; i < static_cast<int>(str.size()); i++) {
        char X = str[i];
        if (isTerminal(X)) {
            result.insert(X);
            return result;
```

```
        }
        bool hasEpsilon = false;
        for (auto c : FIRSTSets[X]) {
            if (c == '@') {
                hasEpsilon = true;
            } else {
                result.insert(c);
            }
        }
        if (!hasEpsilon) {
            return result;
        }
        if (i == static_cast<int>(str.size()) - 1) {
            result.insert('@');
        }
    }
    return result;
}

void buildFIRST()
{
    bool changed = true;
    for (auto nt : nonterminals) {
        FIRSTSets[nt]; // init empty
    }

    while (changed) {
        changed = false;
        for (auto nt : nonterminals) {
            for (auto& rhs : grammar[nt]) {
```

```
                std::set<char> f = FIRST_str(rhs);
                int oldSize = static_cast<int>(FIRSTSets[nt].size());
                for (auto c : f) {
                    FIRSTSets[nt].insert(c);
                }
                if (static_cast<int>(FIRSTSets[nt].size()) > oldSize) {
                    changed = true;
                }
            }
        }
    }
}
```

1.2.4 求 FOLLOW 集

调用 buildFOLLOW() 函数为每个非终结符号计算 FOLLOW 集。

- 将 $ 放入开始符号 E 的 FOLLOW 集。
- 对文法中每条产生式 $A \to \alpha B \beta$:
- 将 FIRST(β) 除去 ϵ 加入 FOLLOW(B);
- 若 β 可推出 ε，则 FOLLOW(A) 中的所有符号也加入 FOLLOW(B)。

```
void buildFOLLOW()
{
    for (auto nt : nonterminals) {
        FOLLOWSets[nt];
    }
    FOLLOWSets['E'].insert('$');

    bool changed = true;
    while (changed) {
        changed = false;
        for (auto nt : nonterminals) {
```

```
            for (auto& rhs : grammar[nt]) {
                for (int i = 0; i < static_cast<int>(rhs.size()); i++) {
                    char B = rhs[i];
                    if (nonterminals.count(B) != 0U) {
                        std::string beta = rhs.substr(i + 1);
                        std::set<char> firstBeta = FIRST_str(beta);
                        int oldSize = static_cast<int>(FOLLOWSets[B].size());
                        if (firstBeta.count('@') != 0U) {
                            firstBeta.erase('@');
                            for (auto c : firstBeta) {
                                FOLLOWSets[B].insert(c);
                            }
                            for (auto c : FOLLOWSets[nt]) {
                                FOLLOWSets[B].insert(c);
                            }
                        } else {
                            for (auto c : firstBeta) {
                                FOLLOWSets[B].insert(c);
                            }
                        }
                        if (static_cast<int>(FOLLOWSets[B].size()) > oldSize) {
                            changed = true;
                        }
                    }
                }
            }
        }
    }
}
```

1.2.5 构造预测分析表 M

求出 FIRST 和 FOLLOW 集后，我们可以方便地构造 LL(1) 分析表，算法如 algorithm 1 所示。

Algorithm 1: 构造预测分析表 M 的算法

Input: 文法 G
Output: 预测分析表 M

```
foreach 产生式 A → α 属于 G do
    foreach a ∈ FIRST(α) do
        将 A → α 加入到 M[A, a] 中;
    if ϵ ∈ FIRST(α) then
        foreach b ∈ FOLLOW(A) do
            将 A → α 加入到 M[A, b] 中;
foreach 未定义的表项 M[A, a] do
    标记为错误;
```

根据算法实现 `buildParsingTable()` 函数，根据 LL(1) 文法构造预测分析表 M。

对于每个产生式 $A \to \alpha$：

- 对 FIRST(α) 中的每个终结符号 a（若非 ε）将 $A \to \alpha$ 填入表项 $M[A, a]$。
- 若 ε 在 FIRST(α) 中，则对 FOLLOW(A) 中的每个符号 b，将 $A \to \alpha$ 填入 $M[A, b]$。

未定义表项表示 `error`。

```
void buildParsingTable()
{
    for (auto& p : prods) {
        char A = p.second.first;
        std::string alpha = p.second.second;
        std::set<char> firstSet = FIRST_str(alpha);
        bool hasEpsilon = (firstSet.count('@') != 0U);
        for (auto a : firstSet) {
            if (a != '@') {
                M[A][a] = p.first;
            }
        }
        if (hasEpsilon) {
```

```
            for (auto b : FOLLOWSets[A]) {
                M[A][b] = p.first;
            }
        }
    }
}
```

1.2.6 预测分析

最后调用 LLparse() 函数进行 LL(1) 分析。

- 初始化分析栈，将 $ 和开始符号 E 压栈。
- 输入串置于输入缓冲区（末尾有 $），指针 ip 指向首符号。
- 重复以下步骤直到栈顶为 $ 并匹配输入 $:
  1. 若栈顶 X 为终结符或 $:
     - 若 $X = a$（当前输入符号），弹栈并前进输入指针 ip;
     - 否则报错。
  2. 若栈顶 X 为非终结符:
     - 查表 $M[X, a]$，若有产生式 $X \to Y_1 Y_2 \dots Y_k$，弹出 X，逆序将 $Y_k \dots Y_1$ 入栈;
     - 若无匹配产生式则报错。
     - 过程中输出分析步骤,包括栈内容、剩余输入串和动作(产生式编号或 match、error、accept)。

转化成伪代码如 algorithm 2 所示。

根据伪代码，实现 LLparse() 函数如下:

```
void LLparse(const std::string& input_line)
{
    std::stack<char> st;
    st.push('$');
    st.push(startSymbol);

    int ip = 0;
    bool done = false;
```

Algorithm 2: 自顶向下预测分析算法

Input: 输入符号串 ω，文法 G 的预测分析表 M
Output: 若 $\omega \in L(G)$，则输出 ω 的最左推导，否则报告错误

```
1  将 $ 压入栈底，将文法开始符号 S 压入栈顶;
2  将 ω$ 放入输入缓冲区中，并令 ip 指向其第一个符号;
3  repeat
4      设 X 为栈顶文法符号，a 为 ip 所指向的输入符号;
5      if X 是终结符号或 $ then
6          if X = a then
7              从栈顶弹出 X;
8              将 ip 前移一个位置;
9          else
10             error();
11     else
12         if M[X, a] = X → Y1Y2 ··· Yk then
13             从栈顶弹出 X;
14             将 Yk, Yk−1, ..., Y1 依次压入栈，Y1 位于栈顶;
15             输出产生式 X → Y1Y2 ··· Yk;
16         else
17             error();
18 until X = $;
```

```
    bool error_flag = false;

    while (!done && !error_flag) {
        char X = st.top();
        char a = input_line[ip];

        if ((terminals.count(X) != 0U || X == '$')) {
            if (X == a) {
                if (X == '$') {
                    printStep(st, input_line, ip, "accept");
                    done = true;
                } else {
                    printStep(st, input_line, ip, "match");
                    st.pop();
                    ip++;
                }
            } else {
                printStep(st, input_line, ip, "error");
                error_flag = true;
            }
        } else {
            if (M[X].find(a) != M[X].end()) {
                int p = M[X][a];
                std::string rhs = prods[p].second;
                printStep(st, input_line, ip, std::to_string(p));
                st.pop();
                for (int i = static_cast<int>(rhs.size()) - 1; i >= 0; i--) {
                    if (rhs[i] != '@' && rhs[i] != 0) {
                        st.push(rhs[i]);
                    }
```

```
39                  }
40              } else {
41                  printStep(st, input_line, ip, "error");
42                  error_flag = true;
43              }
44          }
45      }
46  }
```

1.3 测试报告

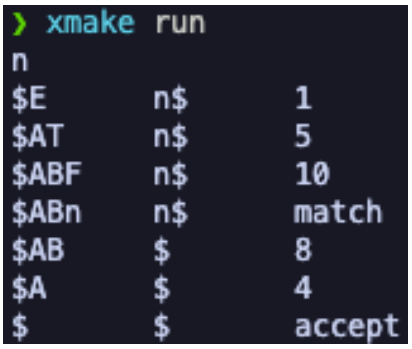

图 1: LL(1) 最简单的单个因子测试

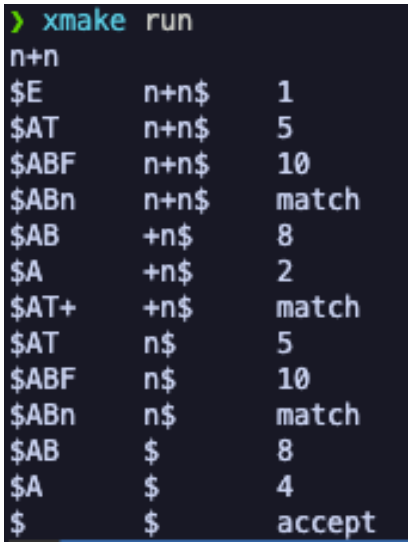

图 2: LL(1) 简单的加法测试

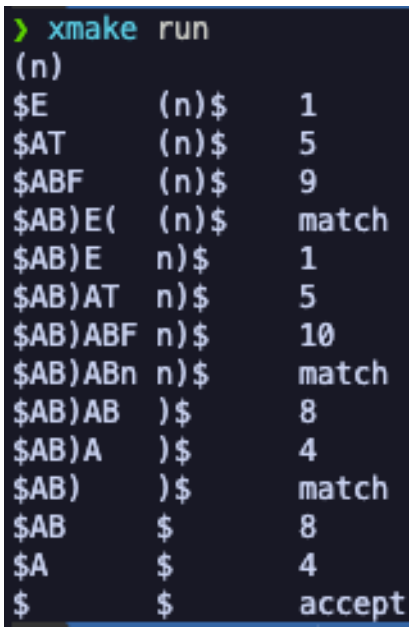

图 3: LL(1) 使用括号包裹单个表达式测试

1.3.1 最简单的单个因子

输入 n

仅有一个操作数，无操作符，测试程序对简单有效输入的处理。测试结果如 Figure 1 所示。

1.3.2 简单的加法

输入 $n+n$

测试程序对基本二元操作的处理。测试结果如 Figure 2 所示。

1.3.3 使用括号包裹单个表达式

输入 (n)

测试程序对带括号的简单表达式解析。测试结果如 Figure 3 所示。

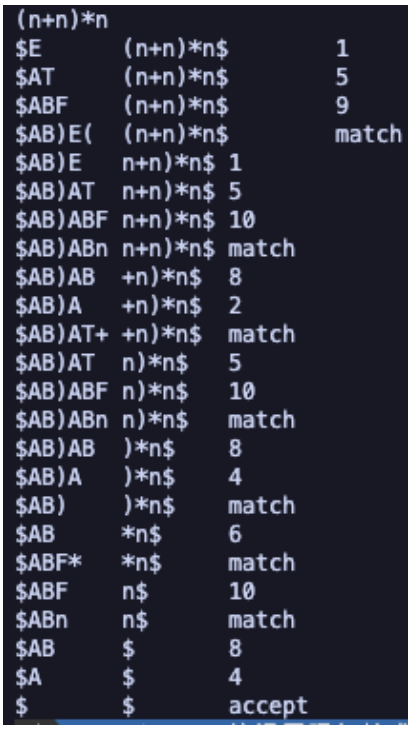

图 4: LL(1) 复杂度稍高，带括号与加法混合

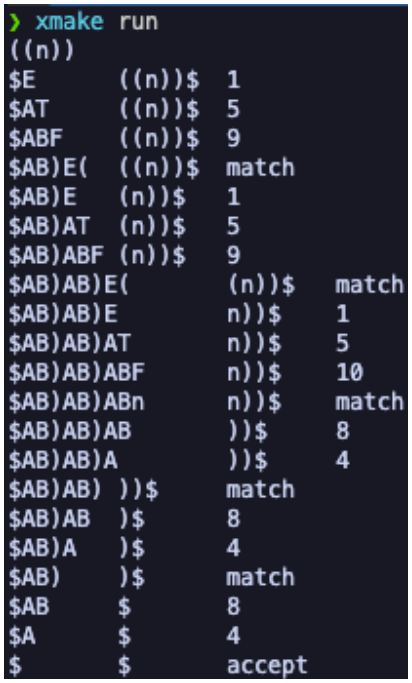

图 5: LL(1) 嵌套括号

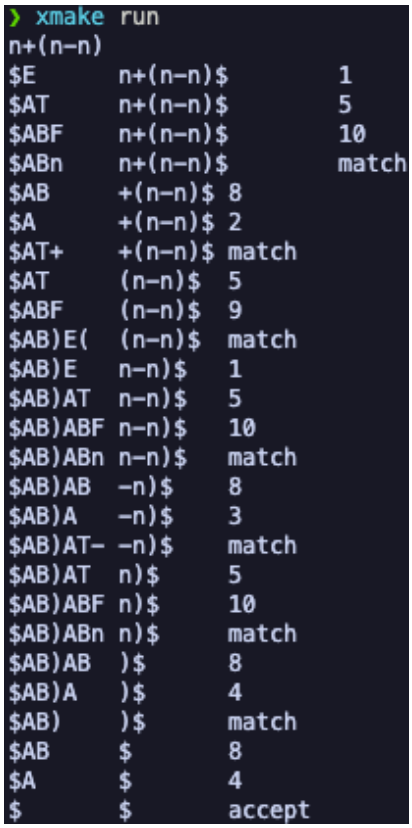

图 6: LL(1) 嵌套加减混合

1.3.4 复杂度稍高，带括号与加法混合

输入 $(n+n)*n$

测试括号内运算与外部运算的组合。测试结果如 Figure 4 所示。

1.3.5 嵌套括号

输入 $((n))$

测试多重括号嵌套的正确处理。测试结果如 Figure 5 所示。

1.3.6 嵌套加减混合

输入 $n+(n-n)$

测试嵌套的子表达式，以及在同一行处理加减混合运算。测试结果如 Figure 6 所示。

```
❯ xmake run
+n
$E      +n$     error
```

图 7: LL(1) 错误用例：表达式以操作符开头

```
❯ xmake run
n+
$E      n+$     1
$AT     n+$     5
$ABF    n+$     10
$ABn    n+$     match
$AB     +$      8
$A      +$      2
$AT+    +$      match
$AT     $       error
```

图 8: LL(1) 错误用例：表达式以操作符结尾

```
❯ xmake run
(n+n
$E      (n+n$   1
$AT     (n+n$   5
$ABF    (n+n$   9
$AB)E(  (n+n$   match
$AB)E   n+n$    1
$AB)AT  n+n$    5
$AB)ABF n+n$    10
$AB)ABn n+n$    match
$AB)AB  +n$     8
$AB)A   +n$     2
$AB)AT+ +n$     match
$AB)AT  n$      5
$AB)ABF n$      10
$AB)ABn n$      match
$AB)AB  $       8
$AB)A   $       4
$AB)    $       error
```

图 9: LL(1) 错误用例：缺少右括号

1.3.7 错误用例：表达式以操作符开头

输入 $+n$

表达式不符合文法规则，期望输出 `error`。测试结果如 Figure 7 所示。

1.3.8 错误用例：表达式以操作符结尾

输入 $n+$

不完整的表达式，缺少操作数，期望输出 `error`。测试结果如 Figure 8 所示。

1.3.9 错误用例：缺少右括号

输入 $(n+n$

缺失右括号，期望输出 `error`。测试结果如 Figure 9 所示。

通过以上测试用例，可以覆盖：

- 基本的合法表达式（单一操作数、简单二元操作、多重操作）
- 带括号的复杂表达式
- 嵌套括号与嵌套表达式
- 非法输入（以操作符开头/结尾、括号不匹配）

这些用例足以全面测试 LL(1) 预测分析程序的正确性和健壮性。

2 编写 LR(1) 语法分析程序

2.1 题目要求

2.1.1 实验背景

来自《编译原理与技术（第 2 版）》第 4 章的**程序设计 2**（方法 3）。

2.1.2 任务描述

编写一个语法分析程序，能对算数表达式进行 LR(1) 语法分析。

要求 对给定文法的产生式进行如下编号后：

编号	产生式
0	$E' \to E$
1	$E \to E + T$
2	$E \to E - T$
3	$E \to T$
4	$T \to T * F$
5	$T \to T/F$
6	$T \to F$
7	$F \to (E)$
8	$F \to \text{num}$
10	$F \to \text{num}$

1. 编程实现构造该文法的 LR(1) 分析表。
2. 编程实现算法 4.3，构造 LR(1) 分析程序。

2.1.3 编程要求

根据注释提示，补充代码。

要求 使用 C/C++ 实现

平台环境说明

- 编译器版本：gcc `7.3.0`
- OS 版本：Debian GNU/Linux `9`

2.1.4 测试说明

输入格式 从标准输入（使用 `cin`/`scanf` 等输入）读入数据。
输入仅包含 1 行：

- 第 1 行输入为一个算术表达式。构成该算术表达式的字符有：{‘`n`’, ‘`+`’, ‘`-`’, ‘`*`’, ‘`/`’, ‘`(`’, ‘`)`’}。

输出格式 输出到标准输出（使用 `cout`/`printf` 等输出）中。
输出包括若干行 LR(1) 分析过程：
假设输出有 n 行，则第 i 行 $(1 \leq i \leq n)$ 表示分析进行到第 i 步，它的输出仅包含一个部分：

- 分析动作：归约使用的产生式编号或 `shift` 或 `error` 或 `accept`（表示当前步骤应执行的动作）

注：规定第 1 步的分析过程为：根据分析栈中只有状态 0、输入串栈顶为第一个输入字符，来产生分析动作。

2.2 程序设计说明

求取 FIRST 集即之前的方法与代码与 LL(1) 分析程序类似，不再赘述。

Algorithm 3: 构造 closure(I) 的过程

Input: 项目集合 I
Output: $J = \text{closure}(I)$

$J \leftarrow I$;
repeat
  $J_{\text{new}} \leftarrow J$;
  foreach $[A \to \alpha \cdot B\beta, a] \in J_{new}$ **do**
    foreach $B \to \eta$ 为文法 G 中的产生式 **do**
      foreach $b \in \text{FIRST}(\beta a)$ **do**
        if $[B \to \cdot\eta, b] \notin J$ **then**
          将 $[B \to \cdot\eta, b]$ 加入 J;
until $J_{new} = J$;

2.2.1 构造 closure(I)

伪代码如 algorithm 3 所示。

由此实现 `closure(I)` 函数对项目集 I 不断添加满足 $[A \to \alpha \cdot B\beta, a]$ 的项目 $[B \to \cdot\gamma, b]$（其中 b 来自 FIRST(βa)）直至不再增加新项目。

`goFunc(I, X)` 将 I 中所有 $[A \to \alpha \cdot X\beta, a]$ 项目的 *dot* 向右移动一位，变成 $[A \to \alpha X \cdot \beta, a]$，所得集合再取 closure。

```
std::set<Item> closure(const std::set<Item>& I)
{
    std::set<Item> J = I;
    bool changed = true;
    while (changed) {
        changed = false;
        std::set<Item> Jnew = J;
        for (const auto& it : J) {
            if (it.dot < static_cast<int>(it.right.size())) {
                char B = it.right[it.dot];
                if (nonterminals.count(B) != 0U) {
                    std::string beta = it.right.substr(it.dot + 1);
                    beta.push_back(it.lookahead);
                    std::set<char> firstSet = FIRST_str(beta);
```

```
                    if (firstSet.empty()) {
                        firstSet.insert(it.lookahead);
}
                    for (auto& p : prods) {
                        if (p.second.first == std::string(1, B)) {
                            for (auto b : firstSet) {
                                Item newItem = { p.second.first,
                                ↪  p.second.second, 0, b };
                                if (J.count(newItem) == 0U) {
                                    Jnew.insert(newItem);
                                }
                            }
                        }
                    }
                }
            }
        }
        if (Jnew.size() != J.size()) {
            J = Jnew;
            changed = true;
        }
    }
    return J;
}

std::set<Item> goFunc(const std::set<Item>& I, char X)
{
    std::set<Item> J;
    for (const auto& it : I) {
```

```
        if (it.dot < static_cast<int>(it.right.size()) && it.right[it.dot] ==
        ↪ X) {
            J.insert({ it.left, it.right, it.dot + 1, it.lookahead });
        }
    }
    return closure(J);
}
```

2.2.2 构造 LR(1) 项目集规范族

- 定义 `Item` 类表示 LR(1) 项目，其包含 `left`, `right`, `dot`, `lookahead`。
- 初始项目集 I_0 包含 $[E' \to \cdot E, \$]$。通过 `closure()` 函数将该集合闭包扩展，得到 I_0。
- 利用 `goFunc()` 函数对状态集进行扩展：

对规范族中的每个项目集 I 和每个可能的文法符号 X，计算 $go(I, X)$。

若得到的新项目集不在规范族中，则加入规范族，直到不再增加新集合。算法如 algorithm 4 所示。

Algorithm 4: 构造文法 G 的 LR(1) 项目集规范族

Input: 文法 G
Output: G 的 LR(1) 项目集规范族 C

$C \leftarrow \{\text{closure}(\{[S' \to \cdot S,\ \$]\})\}$;
repeat
  foreach $I \in C$ **do**
    foreach 文法符号 X **do**
      if $\text{go}(I, X) \neq \emptyset$ 且 $\text{go}(I, X) \notin C$ **then**
        将 $\text{go}(I, X)$ 加入 C;
until 无新项目集加入 C;

`buildAutomaton()` 从 I_0 开始，反复对所有状态集和可能符号应用 `goFunc()` 构造整个 LR(1) 自动机（即项目集规范族与状态转换）。

将每个项目集分配一个状态号，用于构建 LR(1) 分析表时使用。

```
void buildAutomaton()
{
```

```
    std::set<Item> I0;
    I0.insert({ startSymbol, "E", 0, '$' });
    I0 = closure(I0);
    getIndex(I0);
    bool added = true;
    while (added) {
        added = false;
        int n = static_cast<int>(C.size());
        for (int i = 0; i < n; i++) {
            std::set<char> Xs;
            for (const auto& it : C[i]) {
                if (it.dot < static_cast<int>(it.right.size())) {
                    Xs.insert(it.right[it.dot]);
                }
            }
            for (auto X : Xs) {
                std::set<Item> g = goFunc(C[i], X);
                if (!g.empty()) {
                    int j = 0;
                    if (Cidx.find(g) == Cidx.end()) {
                        j = getIndex(g);
                        added = true;
                    } else {
                        j = Cidx[g];
}
                    if (terminals.count(X) != 0U) {
                        action[i][X] = "s" + std::to_string(j);
                    }
                    if (nonterminals.count(X) != 0U) {
                        goTable[i][X] = j;
```

```
                    }
                }
            }
        }
    }
}
```

2.2.3 构建 LR(1) 分析表

算法如 algorithm 5 所示。

实现 `buildActionGoto()` 函数：

- 对每个状态 i：
  - 若存在项目 $[A \to \alpha \cdot a\beta, b]$ 且 $go(i, a) = j$ 且 a 为终结符，则 $action[i, a] = shift j$。
  - 若存在 $[A \to \alpha \cdot, a]$ 且 A 不是拓广文法的开始符号，则 $action[i, a] = reduce by A \to \alpha$。产生式编号从 $prods$ 中获得。
  - 若 $[E' \to E \cdot, \$]$ 在某状态中出现，则 $action[i, \$] = accept$。
- 对非终结符 A 若 $go(i, A) = j$ 则 $goto[i, A] = j$。
- 未定义的表项为 `error`。

```
void buildActionGoto()
{
    for (int i = 0; i < static_cast<int>(C.size()); i++) {
        for (const auto& it : C[i]) {
            if (it.dot == static_cast<int>(it.right.size())) {
                if (it.left == startSymbol && it.lookahead == '$') {
                    action[i]['$'] = "acc";
                } else {
                    for (auto& p : prods) {
                        if (p.second.first == it.left && p.second.second ==
                        ↪  it.right) {
                            action[i][it.lookahead] = "r" +
                            ↪  std::to_string(p.first);
```

Algorithm 5: 构造文法 G 的 LR(1) 分析表的伪代码

Input: 文法 G
Output: LR(1) 分析表 *action* 和 *goto*

根据 G 的开始符号 S，构造拓广文法 G'，加入新的开始符号 S'，添加产生式 $S' \to S$。
$C \leftarrow \{\text{closure}(\{[S' \to \cdot S,\ \$]\})\}$;
repeat
  foreach $I \in C$ **do**
    foreach 文法符号 X **do**
      $J \leftarrow go(I, X)$;
      if $J \neq \emptyset$ 且 $J \notin C$ **then**
        将 J 加入 C;
      end
    end
  end
until 无新项目集加入 C;
foreach $I_i \in C$ **do**
  foreach $[A \to \alpha \cdot a\beta, b] \in I_i$ **do**
    if a 为终结符号且 $go(I_i, a) = I_j$ **then**
      $action[i, a] \leftarrow S_j$;
    end
  end
  foreach $[A \to \alpha\cdot, a] \in I_i$ **do**
    if $A \neq S'$ **then**
      设 $A \to \alpha$ 在原文法中编号为 r，$action[i, a] \leftarrow R_r$;
    end
  end
  if $[S' \to S\cdot,\ \$] \in I_i$ **then**
    $action[i, \$] \leftarrow ACC$;
  end
  foreach 非终结符号 A **do**
    if $go(I_i, A) = I_j$ **then**
      $goto[i, A] \leftarrow j$;
    end
  end
end
foreach i **do**
  foreach 符号 X（包括终结符号和非终结符号）**do**
    if $action[i, X]$ 和 $goto[i, X]$ 均未设定 **then**
      标记 $action[i, X]$ 或 $goto[i, X]$ 为 *error*;
    end
  end
end
初态为包含 $[S' \to \cdot S,\ \$]$ 的项目集对应的状态。

```
                        }
                    }
                }
            }
        }
    }
    for (int i = 0; i < static_cast<int>(C.size()); i++) {
        for (const auto& t : terminals) {
            if (action[i].find(t) == action[i].end()) {
                action[i][t] = "err";
            }
        }
        action[i]['$'] = (action[i].find('$') == action[i].end()) ? "err" :
        ↪ action[i]['$'];
    }
}
```

2.2.4 LR 语法分析

`LRparse()` 函数通过状态栈（`stateStack`）和符号栈（`symbolStack`）进行分析，算法如 algorithm 6 所示。

- 初始化：状态栈压入 0 状态。输入串末尾加 `$`。
- 重复：
  - 取栈顶状态 S 和当前输入符号 a。
  - 查看 $action[S, a]$：
    - 若为 $shifts$：推入符号 a 和状态 s，前移输入指针。
    - 若为 $reduce by A \to \beta$：弹出 $|\beta|$ 个符号及状态，查看当前栈顶状态 S’，然后压入 A 及 $goto[S', A]$。输出产生式编号。
    - 若为 `accept`：输出 `accept`，结束分析。
    - 若为 `error`：输出 `error`，结束分析。

```
void LRparse(const std::string& input)
```

Algorithm 6: LR 分析程序

Input: 文法 G 的 LR 分析表 (action 和 goto)，输入符号串 ω
Output: 若 $\omega \in L(G)$，则输出 ω 的自底向上分析过程，否则报告错误

```
1  将初始状态 S_0 压入状态栈，将符号栈置空;
2  将 ω$ 放入输入缓冲区，并令 ip 指向其第一个符号;
3  repeat
4  |  设 S 为状态栈顶状态，a 为 ip 所指向的输入符号;
5  |  if action[S, a] = shift S' then
6  |  |  将 a 压入符号栈;
7  |  |  将 S' 压入状态栈;
8  |  |  ip ← ip + 1;
9  |  else if action[S, a] = reduce by A → β then
10 |  |  从符号栈和状态栈中弹出 |β| 个符号和状态;
11 |  |  设当前状态栈顶为 S';
12 |  |  将 A 压入符号栈;
13 |  |  将 goto[S', A] 压入状态栈;
14 |  |  输出产生式 A → β;
15 |  else if action[S, a] = accept then
16 |  |  return;
17 |  else
18 |  |  error();
19 until 终止;
```

```
{
    std::stack<int> stateStack;
    std::stack<char> symbolStack;
    stateStack.push(0);
    int ip = 0;
    bool done = false;
    while (!done) {
        int S = stateStack.top();
        char a = input[ip];
        std::string act = (action[S].find(a) != action[S].end()) ? action[S][a]
        ↪  : "err";
        if (act == "acc") {
            std::cout << "accept\n";
            done = true;
        } else if (act[0] == 's') {
            std::cout << "shift\n";
            int nxt = std::stoi(act.substr(1));
            symbolStack.push(a);
            stateStack.push(nxt);
            ip++;
        } else if (act[0] == 'r') {
            int prodNo = std::stoi(act.substr(1));
            std::string A = prods[prodNo].first;
            std::string B = prods[prodNo].second;
            int len = static_cast<int>(B.size());
            for (int i = 0; i < len; i++) {
                symbolStack.pop();
                stateStack.pop();
            }
            int S2 = stateStack.top();
```

```
            symbolStack.push(A[0]);
            stateStack.push(goTable[S2][A[0]]);
            std::cout << prodNo << "\n";
        } else {
            std::cout << "error\n";
            done = true;
        }
    }
}
```

2.3 测试报告

采用与 LL(1) 同样的测试。

```
❯ xmake run
n
shift
8
6
3
accept
```

图 10: LR(1) 最简单的单个因子测试

```
❯ xmake run
n+n
shift
8
6
3
shift
shift
8
6
1
accept
```

图 11: LR(1) 简单的加法测试

```
❯ xmake run
(n)
shift
shift
8
6
3
shift
7
6
3
accept
```

图 12: LR(1) 使用括号包裹单个表达式测试

2.3.1 最简单的单个因子

输入 n

仅有一个操作数，无操作符，测试程序对简单有效输入的处理。测试结果如 Figure 10 所示。

2.3.2 简单的加法

输入 $n+n$

测试程序对基本二元操作的处理。测试结果如 Figure 11 所示。

2.3.3 使用括号包裹单个表达式

输入 (n)

测试程序对带括号的简单表达式解析。测试结果如 Figure 12 所示。

2.3.4 复杂度稍高，带括号与加法混合

输入 $(n+n)*n$

测试括号内运算与外部运算的组合。测试结果如 Figure 13 所示。

2.3.5 嵌套括号

输入 $((n))$

测试多重括号嵌套的正确处理。测试结果如 Figure 14 所示。

2.3.6 嵌套加减混合

输入 $n+(n-n)$

测试嵌套的子表达式，以及在同一行处理加减混合运算。测试结果如 Figure 15 所示。

2.3.7 错误用例：表达式以操作符开头

输入 $+n$

表达式不符合文法规则，期望输出 `error`。测试结果如 Figure 16 所示。

2.3.8 错误用例：表达式以操作符结尾

输入 $n+$

不完整的表达式，缺少操作数，期望输出 `error`。测试结果如 Figure 17 所示。

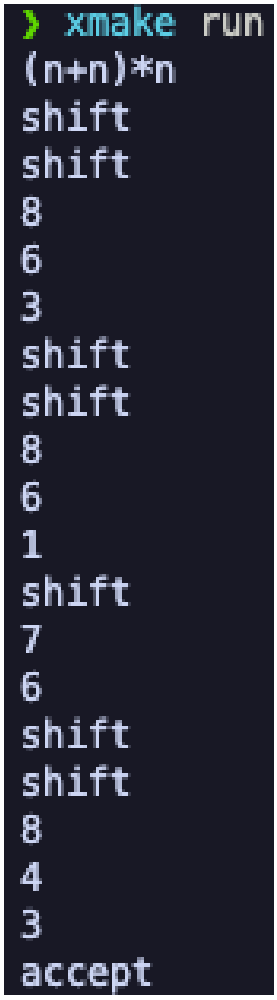

图 13: LR(1) 复杂度稍高，带括号与加法混合

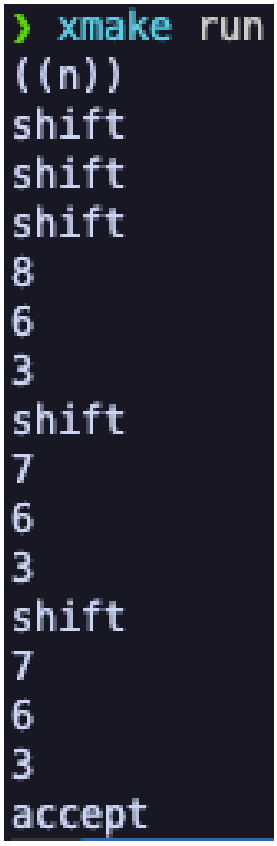

图 14: LR(1) 嵌套括号

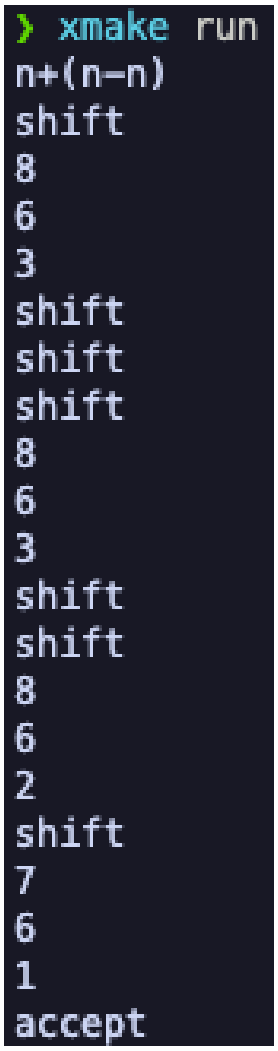

图 15: LR(1) 嵌套加减混合

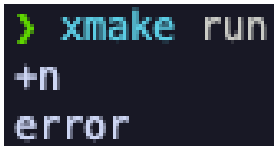

图 16: LR(1) 错误用例：表达式以操作符开头

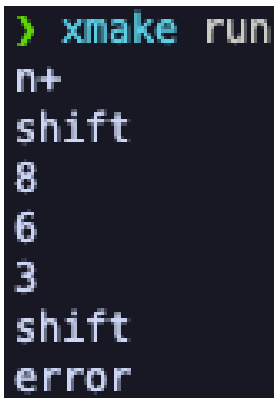

图 17: LR(1) 错误用例：表达式以操作符结尾

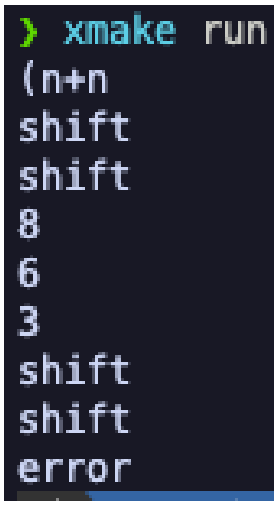

图 18: LR(1) 错误用例：缺少右括号

2.3.9 错误用例：缺少右括号

输入 $(n+n$

缺失右括号，期望输出 `error`。测试结果如 Figure 18 所示。

全部符合预期。

3 总结

在完成 LL(1) 和 LR(1) 语法分析程序的设计与实现后，我深刻体会到编译原理中理论与实践结合的奥妙。这一过程不仅强化了我对语法分析基本原理的理解，也让我对设计和调试复杂算法的耐心与能力有了更高的要求。

首先，通过实现 LL(1) 分析程序，我进一步领会了自顶向下分析的核心思想——如何通过 FIRST 和 FOLLOW 集预测接下来的处理步骤。在构造预测分析表时，我遇到了文法消除左递归和提取公共左因子的问题。这些看似抽象的理论步骤，通过程序的实现和测试，变得清晰且有意义。同时，预测分析表的构建也让我感受到算法的精妙之处，尤其是如何在有限的上下文中高效确定下一步的行为。

接着，在实现 LR(1) 分析程序时，我切实体会到了自底向上分析的强大。相比 LL(1) 的局限性，LR(1) 能够处理更复杂的文法，这使得它的实现也更加繁琐。从项目集规范族的构建，到 closure 和 go 的细致处理，再到生成分析表的过程，每一步都充满了逻辑推导的挑战。尤其是 LR(1) 中

对 lookahead 符号的管理，让我认识到它在消除语法歧义中的重要性。这部分的设计让我理解到，精确控制状态和转移，是解析复杂文法的关键。

在测试阶段，看到程序成功处理各种合法表达式，甚至能准确地报告错误时，那种成就感是难以言喻的。测试用例从简单到复杂，再到边界情况和错误输入，每个结果的正确输出都让我对理论的实用性和代码的可靠性感到欣慰。同时，这一过程也让我意识到测试的重要性——只有严谨的测试才能确保程序的健壮性。

此外，通过比较 LL(1) 和 LR(1) 的性能与适用性，我更深刻地理解了编译器的设计哲学——权衡简单性与能力。LL(1) 的实现相对简单，但对文法要求更高；而 LR(1) 虽然实现复杂，但适用范围广。这种取舍让我联想到工程实践中的实际决策，学会权衡理论上的完美与实际中的可行性。

总的来说，这次实验让我不仅收获了知识，还培养了分析问题和解决问题的能力。从理论到代码再到测试的完整闭环，让我更全面地理解了编译器的内部机制，也让我对后续深入学习和研究充满了信心。每一次调试，每一个测试结果，都是对编译原理魅力的最好诠释。

A 线上测试例通过情况截图

唐梓楠 按时通关

学号：2022211404　　分班：2022211312

截止前完成关卡：2/2　　最新完成关卡：2/2　　完成效率：--　　课堂最高完成效率：--

通关时间	计时规则	实训总耗时	评测次数	查重扣分	补交扣分	最终成绩	总评
2024-12-07 12:51	页面停留时长	1 时 2 分 24 秒	8	--	--	100.0/100.0	优秀

阶段成绩

关卡	任务名称	开启时间	代码修改行数	评测次数	完成时间	实训耗时	是否查看答案	经验值	关卡得分	调分
1	编写 LL (1) 语法分析程序	2024-12-07 11:16	125	6	2024-12-07 12:00	43 分 59 秒	否	300/300	50.00/50	50.00
2	编写 LR (1) 语法分析程序	2024-12-07 12:00	277	2	2024-12-07 12:51	18 分 25 秒	否	400/400	50.00/50	50.00