南京林业大学

编译原理实验报告

班级	23508014
姓名	方泽宇
学号	2351610105
任课老师	谢德红
得分	

目录

实验 1: 词法分析 ..... - 3 -

实验 2: 语法分析法 1-递归子程序法 ..... - 10 -

实验 3: 语法分析法 2-预测分析法 ..... - 17 -

实验 4: 语法分析 3-LR(1)分析程序 ..... - 22 -

实验 5: 中间代码生成 ..... - 27 -

实验 1：词法分析

一. 实验目的、内容

1. 实验目的：通过完成词法分析程序，了解词法分析的过程；
2. 实验内容： 用 C 语言实现对 C 的子集程序设计语言的词法识别程序；
3. 实验要求：将该语言的源程序，也就是相应字符流转换成内码，并根据需要是否对于标识符填写相应的符号表供编译程序的以后各阶段使用.

二. 实验过程

核心部分，词法分析有关函数(JavaScript)：

```
function lexicalAnalyze(code) {
    const tokens = [];
    let pos = 0;
    let line = 1;
    let column = 1;
    while (pos < code.length) {
        // 跳过空白字符
        if (/\s/.test(code[pos])) {
            if (code[pos] === '\n') {
                line++;
                column = 1;
            } else {
                column++;
            }
            pos++;
            continue;
        }
        // 跳过注释
        if (code[pos] === '/' && code[pos + 1] === '/') {
            while (pos < code.length && code[pos] !== '\n') {
                pos++;
                column++;
            }
            continue;
        }
        if (code[pos] === '/' && code[pos + 1] === '*') {
            pos += 2;
            column += 2;
            while (pos < code.length - 1 && !(code[pos] === '*' && code[pos + 1]
=== '/')) {
                if (code[pos] === '\n') {
                    line++;
```

```
            column = 1;
        } else {
            column++;
        }
        pos++;
    }
    pos += 2;
    column += 2;
    continue;
}

// 字符串字面量
if (code[pos] === '"') {
    const start = pos;
    const startColumn = column;
    pos++;
    column++;

    while (pos < code.length && code[pos] !== '"') {
        if (code[pos] === '\\') {
            pos++;
            column++;
        }
        if (code[pos] === '\n') {
            line++;
            column = 1;
        } else {
            column++;
        }
        pos++;
    }

    if (pos < code.length) {
        pos++;
        column++;
        tokens.push({
            type: 3,
            value: code.substring(start, pos),
            line: line,
            position: startColumn
        });
    }
```

```
        continue;
    }

    // 字符字面量
    if (code[pos] === "'") {
        const start = pos;
        const startColumn = column;
        pos++;
        column++;

        if (code[pos] === '\\') {
            pos++;
            column++;
        }

        if (pos < code.length) {
            pos++;
            column++;
        }

        if (pos < code.length && code[pos] === "'") {
            pos++;
            column++;
            tokens.push({
                type: 3,
                value: code.substring(start, pos),
                line: line,
                position: startColumn
            });
        }
        continue;
    }

    // 数字
    if (/\d/.test(code[pos])) {
        const start = pos;
        const startColumn = column;

        while (pos < code.length && /\d/.test(code[pos])) {
            pos++;
            column++;
        }
```

```
    if (code[pos] === '.') {
        pos++;
        column++;
        while (pos < code.length && /\d/.test(code[pos])) {
            pos++;
            column++;
        }
    }

    tokens.push({
        type: 3,
        value: code.substring(start, pos),
        line: line,
        position: startColumn
    });
    continue;
}

// 标识符和关键字
if (/[a-zA-Z_]/.test(code[pos])) {
    const start = pos;
    const startColumn = column;

    while (pos < code.length && /[a-zA-Z0-9_]/.test(code[pos])) {
        pos++;
        column++;
    }

    const value = code.substring(start, pos);
    const type = keywords.has(value) ? 1 : 2;

    tokens.push({
        type: type,
        value: value,
        line: line,
        position: startColumn
    });
    continue;
}

// 运算符和分隔符
```

```
        const startColumn = column;

        // 检查双字符运算符
        if (pos + 1 < code.length) {
            const twoChar = code.substring(pos, pos + 2);
            if (operators.has(twoChar)) {
                tokens.push({
                    type: 4,
                    value: twoChar,
                    line: line,
                    position: startColumn
                });
                pos += 2;
                column += 2;
                continue;
            }
        }

        // 单字符
        const char = code[pos];
        let type = 5; // 默认分隔符

        if (operators.has(char)) {
            type = 4;
        } else if (delimiters.has(char)) {
            type = 5;
        }

        tokens.push({
            type: type,
            value: char,
            line: line,
            position: startColumn
        });

        pos++;
        column++;
    }

    return tokens;
}
```

上述代码采用基于字符扫描的有限状态机设计，通过 while 循环逐个字符处理输入字符串，根据当前字符的特征进入不同的解析状态。识别的优先级为：空白字符（跳过）>注释（跳过）>字符串字面量>字符字面量>数字常量>标识符和关键字>运算符和分隔符。
在识别复合 token 时采用贪心算法，尽可能匹配最长的有效序列。

```
// 标识符识别：持续读取直到遇到非标识符字符
while (pos < code.length && /[a-zA-Z0-9_]/.test(code[pos])) {
    pos++;
    column++;
}
```

词法分析程序通过前瞻字符检查来识别多字符运算符，避免不必要的回溯：

```
// 检查双字符运算符（如==、>=等）
if (pos + 1 < code.length) {
    const twoChar = code.substring(pos, pos + 2);
    if (operators.has(twoChar)) {
        // 直接识别双字符运算符
    }
}
```

由于程序要求有词法分析前端，因此选用 HTML5+CSS3+Javascript 作为编程语言。

三. 实验结果

使用 html5+css3+javascript 的 web 开发框架写出 index.html 词法分析工具：

```
main()
{
int i,j,m,n;
printf("please input data to i,j");
i=1;
j=4;
m=i+j;
n=i-j;
printf("m=%d,n=%d"
,m,n);
}
```

将上述示例 c 语言程序输入词法分析器，得到如下结果：

```
（1，"main"）（5，"("）（5，")"）（5，"{"）（1，"int"）（2，"i"）（5，","）（2，"j"）（5，","）（2，"m"）（5，","）（2，"n"）（5，";"）（1，"printf"）（5，"("）（3，""please input data to i,j""）（5，")"）（5，";"）（2，"i"）（4，"="）（3，"1"）（5，";"）（2，"j"）（4，"="）（3，"4"）（5，";"）（2，"m"）（4，"="）（2，"i"）（4，"+"）（2，"j"）（5，";"）（2，"n"）（4，"="）（2，"i"）（4，"-"）（2，"j"）（5，";"）（1，
```

"printf"）（5，"("）（3，""m=%d,n=%d""）（5，","）（2，"m"）（5，","）（2，"n"）（5，")"）（5，";"）（5，"}"）

实验 2：语法分析法 1-递归子程序法

一．实验目的、内容

1） 实验目的：通过完成语法分析程序，了解语法分析的过程和作用。

2） 实验内容：用递归子程序法实现对 C 的子集程序设计语言的分析程序。

3） 实验要求：根据某一文法编制调试递归下降分析程序，以便对任意输入的符号串进行分析。

二．关键代码/实验过程

由题设可知，各非终结符的推导式，对于右部中的每个符号 xi

①如果 xi 为终结符号：

```
if(xi= = 当前的符号)
{
    NextChar();
    return;
}
else
    出错处理
```

② 如果 xi 为非终结符号，直接调用相应。

这里是各产生式向下递推的代码，以 G->+TG|-TG| ε 为例：

```
// G -> + T G | - T G | ε
void G() {
   skipWhitespace();
   if (current_char == '+') {
      replaceLeftmost('G', "+TG");
      printStep("G -> + T G");
      if (!match('+')) {
         reportError("期望 '+'");
         return;
      }
      T();
      if (!error) G();
   } else if (current_char == '-') {
      replaceLeftmost('G', "-TG");
      printStep("G -> - T G");
      if (!match('-')) {
         reportError("期望 '-'");
         return;
      }
      T();
      if (!error) G();
   } else {
```

```
        replaceLeftmost('G', "");
        printStep("G -> ε");
    }
}
```

字符串匹配的函数：

```
bool match(char expected) {
    skipWhitespace();
    if (current_char == expected) {
        nextChar();
        return true;
    }
    return false;
}
```

空字符串处理、读取下一个字符的函数：

```
void nextChar() {
    position++;
    if (position < input.length()) {
        current_char = input[position];
    } else {
        current_char = '\0';
    }
}

void skipWhitespace() {
    while (current_char == ' ' || current_char == '\t' || current_char == '\n')
{
        nextChar();
    }
}
```

总的程序思路：

1） 定义部分：定义常量、变量、数据结构。

2） 初始化：从文件将输入符号串输入到字符缓冲区中。

3） 利用递归下降分析法分析，对每个非终结符编写函数，在主函数中调用文法开始符号的函数。

三．实验结果

达成的实验目标：

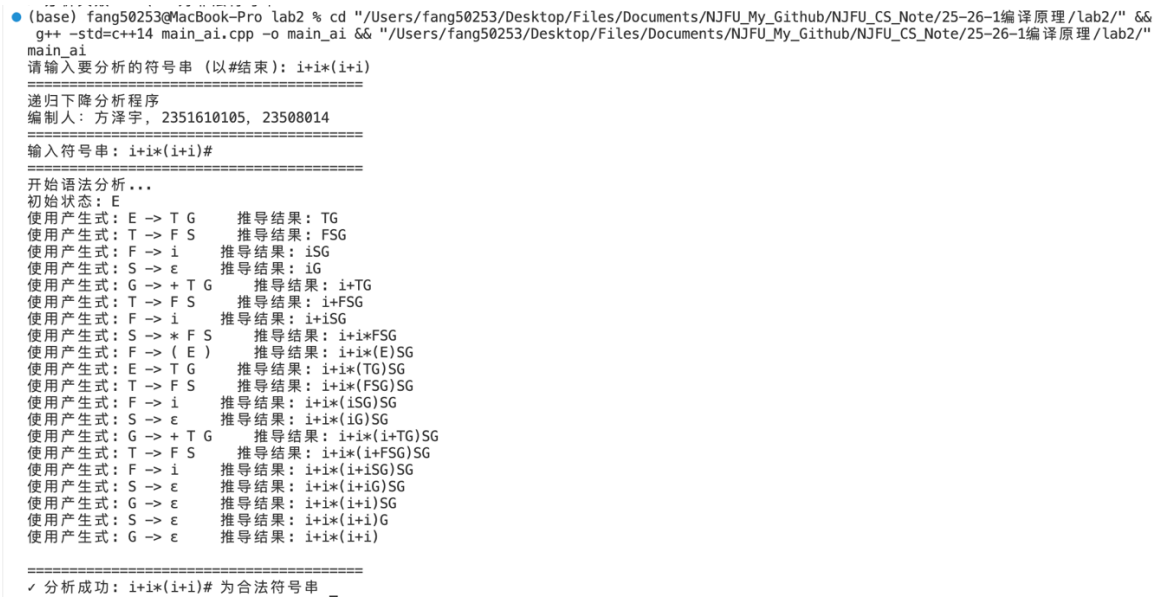
1） 表达式中允许使用运算符(+-*/)、分割符(括号)、字符 I、结束符#；

2） 如果遇到错误的表达式，应输出错误提示信息(我可以告诉你我期待你输入什么，而你

给我输入了什么)；

3） 输出了推导的过程，即详细列出每一步使用的产生式。

对于一个合法的字符串 i+i*(i+i)#，输入语法分析程序后，得到结果如下：



初始状态: E
使用产生式: E -> T G 推导结果: TG
使用产生式: T -> F S 推导结果: FSG
使用产生式: F -> i 推导结果: iSG
使用产生式: S -> ε 推导结果: iG
使用产生式: G -> + T G 推导结果: i+TG
使用产生式: T -> F S 推导结果: i+FSG
使用产生式: F -> i 推导结果: i+iSG
使用产生式: S -> * F S 推导结果: i+i*FSG
使用产生式: F -> (E) 推导结果: i+i*(E)SG
使用产生式: E -> T G 推导结果: i+i*(TG)SG
使用产生式: T -> F S 推导结果: i+i*(FSG)SG
使用产生式: F -> i 推导结果: i+i*(iSG)SG
使用产生式: S -> ε 推导结果: i+i*(iG)SG
使用产生式: G -> + T G 推导结果: i+i*(i+TG)SG
使用产生式: T -> F S 推导结果: i+i*(i+FSG)SG
使用产生式: F -> i 推导结果: i+i*(i+iSG)SG
使用产生式: S -> ε 推导结果: i+i*(i+iG)SG
使用产生式: G -> ε 推导结果: i+i*(i+i)SG
使用产生式: S -> ε 推导结果: i+i*(i+i)G
使用产生式: G -> ε 推导结果: i+i*(i+i)

对于一个不合法的字符串 i+i*(#，输入语法分析程序后，得到结果如下：

```
(base) fang50253@MacBook-Pro lab2 % cd "/Users/fang50253/Desktop/Files/Documents/NJFU_My_Github/NJFU_CS_Note/25-26-1编译原理/lab2/
 g++ -std=c++14 main_ai.cpp -o main_ai && "/Users/fang50253/Desktop/Files/Documents/NJFU_My_Github/NJFU_CS_Note/25-26-1编译原理/la
main_ai
请输入要分析的符号串 (以#结束): i+i*(#
========================================
递归下降分析程序
编制人: 方泽宇, 2351610105, 23508014
========================================
输入符号串: i+i*(#
========================================
开始语法分析...
初始状态: E
使用产生式: E -> T G     推导结果: TG
使用产生式: T -> F S     推导结果: FSG
使用产生式: F -> i     推导结果: iSG
使用产生式: S -> ε     推导结果: iG
使用产生式: G -> + T G     推导结果: i+TG
使用产生式: T -> F S     推导结果: i+FSG
使用产生式: F -> i     推导结果: i+iSG
使用产生式: S -> * F S     推导结果: i+i*FSG
使用产生式: F -> ( E )     推导结果: i+i*(E)SG
使用产生式: E -> T G     推导结果: i+i*(TG)SG
使用产生式: T -> F S     推导结果: i+i*(FSG)SG
错误: 期望 '(', 'i' 或标识符

========================================
✗ 分析失败: i+i*(# 为非法符号串
```

初始状态: E
使用产生式: E -> T G　　推导结果: TG
使用产生式: T -> F S　　推导结果: FSG
使用产生式: F -> i　　推导结果: iSG
使用产生式: S -> ε　　推导结果: iG
使用产生式: G -> + T G　　推导结果: i+TG
使用产生式: T -> F S　　推导结果: i+FSG
使用产生式: F -> i　　推导结果: i+iSG
使用产生式: S -> * F S　　推导结果: i+i*FSG
使用产生式: F -> (E)　　推导结果: i+i*(E)SG
使用产生式: E -> T G　　推导结果: i+i*(TG)SG
使用产生式: T -> F S　　推导结果: i+i*(FSG)SG
错误: 期望 '(', 'i' 或标识符

错误分析：F->(E)，但是 E 无法推出 ε，因此“(”的右边一定有其他的标志符、和“)”。

四. 实验样例测试

1） 测试 i+i#

```
==========================================
递归下降分析程序
编制人：方泽宇，2351610105，23508014
==========================================
输入符号串：i+i#
==========================================
开始语法分析...
初始状态：E
使用产生式：E -> T G     推导结果：TG
使用产生式：T -> F S     推导结果：FSG
使用产生式：F -> i     推导结果：iSG
使用产生式：S -> ε     推导结果：iG
使用产生式：G -> + T G     推导结果：i+TG
使用产生式：T -> F S     推导结果：i+FSG
使用产生式：F -> i     推导结果：i+iSG
使用产生式：S -> ε     推导结果：i+iG
使用产生式：G -> ε     推导结果：i+i

==========================================
✓ 分析成功：i+i# 为合法符号串
```

2） 测试 i-i#

```
==========================================
递归下降分析程序
编制人：方泽宇，2351610105，23508014
==========================================
输入符号串：i-i#
==========================================
开始语法分析...
初始状态：E
使用产生式：E -> T G     推导结果：TG
使用产生式：T -> F S     推导结果：FSG
使用产生式：F -> i     推导结果：iSG
使用产生式：S -> ε     推导结果：iG
使用产生式：G -> - T G     推导结果：i-TG
使用产生式：T -> F S     推导结果：i-FSG
使用产生式：F -> i     推导结果：i-iSG
使用产生式：S -> ε     推导结果：i-iG
使用产生式：G -> ε     推导结果：i-i

==========================================
✓ 分析成功：i-i# 为合法符号串
```

3） 测试 i*i#

```
请输入要分析的符号串（以#结束）: i*i#
========================================
递归下降分析程序
编制人：方泽宇，2351610105，23508014
========================================
输入符号串: i*i#
========================================
开始语法分析...
初始状态: E
使用产生式: E -> T G     推导结果: TG
使用产生式: T -> F S     推导结果: FSG
使用产生式: F -> i     推导结果: iSG
使用产生式: S -> * F S     推导结果: i*FSG
使用产生式: F -> i     推导结果: i*iSG
使用产生式: S -> ε     推导结果: i*iG
使用产生式: G -> ε     推导结果: i*i

========================================
✓ 分析成功: i*i# 为合法符号串
```

4） 测试 i/i#

```
请输入要分析的符号串（以#结束）: i/i#
========================================
递归下降分析程序
编制人：方泽宇，2351610105，23508014
========================================
输入符号串: i/i#
========================================
开始语法分析...
初始状态: E
使用产生式: E -> T G     推导结果: TG
使用产生式: T -> F S     推导结果: FSG
使用产生式: F -> i     推导结果: iSG
使用产生式: S -> / F S     推导结果: i/FSG
使用产生式: F -> i     推导结果: i/iSG
使用产生式: S -> ε     推导结果: i/iG
使用产生式: G -> ε     推导结果: i/i

========================================
✓ 分析成功: i/i# 为合法符号串
```

5） 测试 i+i*i#

```
========================================
输入符号串: i+i*i#
========================================
开始语法分析...
初始状态: E
使用产生式: E -> T G     推导结果: TG
使用产生式: T -> F S     推导结果: FSG
使用产生式: F -> i     推导结果: iSG
使用产生式: S -> ε     推导结果: iG
使用产生式: G -> + T G     推导结果: i+TG
使用产生式: T -> F S     推导结果: i+FSG
使用产生式: F -> i     推导结果: i+iSG
使用产生式: S -> * F S     推导结果: i+i*FSG
使用产生式: F -> i     推导结果: i+i*iSG
使用产生式: S -> ε     推导结果: i+i*iG
使用产生式: G -> ε     推导结果: i+i*i

========================================
✓ 分析成功: i+i*i# 为合法符号串
```

实验 3：语法分析法 2-预测分析法

一、实验内容、目的

1） 实验目的：通过完成预测分析的语法分析程序，了解预测分析法和递归子程序法的区别和联系。

2） 实验内容：构造预测分析程序，并利用分析表和一个栈来实现对上程序设计语言的分析程序。

3） 实验要求：根据某一文法编制调试LL(1)分析程序，以便对任意输入的符号串进行分析。本次实验的目的主要是加深对预测分析 LL(1)分析法的理解。

二、实验过程、代码

程序设计思路：

1） 符号枚举：定义终结符和非终结符。

2） 分析表：使用 map 数据结构存储预测分析表。

3） 分析栈：使用 stack 存储分析过程中的符号。

4） 分析算法：实现 LL(1)预测分析的核心逻辑。

关键过程：LL(1)分析表的初始化

```
void initializeParsingTable() {
    // E -> T G
    table[E][ID] = {T, G};
    table[E][LPAREN] = {T, G};

    // G -> + T G
    table[G][PLUS] = {PLUS, T, G};
    // G -> - T G
    table[G][MINUS] = {MINUS, T, G};
    // G -> ε
    table[G][RPAREN] = {EMPTY};
    table[G][END] = {EMPTY};

    // T -> F S
    table[T][ID] = {F, S};
    table[T][LPAREN] = {F, S};

    // S -> * F S
    table[S][MULTIPLY] = {MULTIPLY, F, S};
    // S -> / F S
    table[S][DIVIDE] = {DIVIDE, F, S};
    // S -> ε
    table[S][PLUS] = {EMPTY};
    table[S][MINUS] = {EMPTY};
```

```
    table[S][RPAREN] = {EMPTY};
    table[S][END] = {EMPTY};

    // F -> ( E )
    table[F][LPAREN] = {LPAREN, E, RPAREN};
    // F -> i
    table[F][ID] = {ID};
}
```

关键过程：获取产生式字符串

```
std::string getProductionString(Symbol non_terminal, const
std::vector<Symbol>& production) {
    std::string result = symbolToString(non_terminal) + " -> ";
    if (production.empty() || production[0] == EMPTY) {
        result += "ε";
    } else {
        for (size_t i = 0; i < production.size(); i++) {
            result += symbolToString(production[i]);
            if (i < production.size() - 1) {
                result += " ";
            }
        }
    }
    return result;
}
```

1） 预测分析器：打开主见面，根据预测语法分析器 LL 分析合法字符串，在请输入字符串位置输入文法，并且根据预测语法分析器 LL 分析输入字符串 i+i*i#的分析过程,在分析过程中，可以看到分析栈中入栈字符，剩余输入串还没有识别的字符，推到产生式或匹配中反映的是每一个入栈字符的状态，并且在分析结果中反映出识别的结果。

2） 在这个页面中，在同样的文法中分析不合法字符，如果输入字符串 i+(i*i#不符合文法规则字符串时，利用预测语法分析器 LL 具体分析此文法的分析过程，并报出分析结果为此字符串为非法字符串。

三、实验结果

合法的串：

```
========================================
输入符号串：i+i*i#
========================================
步骤 | 分析栈              | 剩余输入串      | 所用产生式
----|-----------------|---------------|-------------------
   1 | #E                | i+i*i#        | E -> T G
```

```
 2 | #GT          | i+i*i#      | T -> F S
 3 | #GSF         | i+i*i#      | F -> i
 4 | #GSi         | i+i*i#      | 匹配 i
 5 | #GS          | +i*i#       | S -> ε
 6 | #G           | +i*i#       | G -> + T G
 7 | #GT+         | +i*i#       | 匹配 +
 8 | #GT          | i*i#        | T -> F S
 9 | #GSF         | i*i#        | F -> i
10 | #GSi         | i*i#        | 匹配 i
11 | #GS          | *i#         | S -> * F S
12 | #GSF*        | *i#         | 匹配 *
13 | #GSF         | i#          | F -> i
14 | #GSi         | i#          | 匹配 i
15 | #GS          | #           | S -> ε
16 | #G           | #           | G -> ε
17 | #            | #           | 分析成功
```

==========

分析步骤

步骤	分析栈	剩余输入串	所用产生式
1	#E	i+i*i#	E → T G
2	#GT	i+i*i#	T → F S
3	#GSF	i+i*i#	F → i
4	#GSi	i+i*i#	匹配 i
5	#GS	+i*i#	S → ε
6	#G	+i*i#	G → + T G
7	#GT+	+i*i#	匹配 +
8	#GT	i*i#	T → F S
9	#GSF	i*i#	F → i
10	#GSi	i*i#	匹配 i
11	#GS	*i#	S → * F S
12	#GSF*	*i#	匹配 *
13	#GSF	i#	F → i
14	#GSi	i#	匹配 i
15	#GS	#	S → ε
16	#G	#	G → ε
17	#	#	分析成功

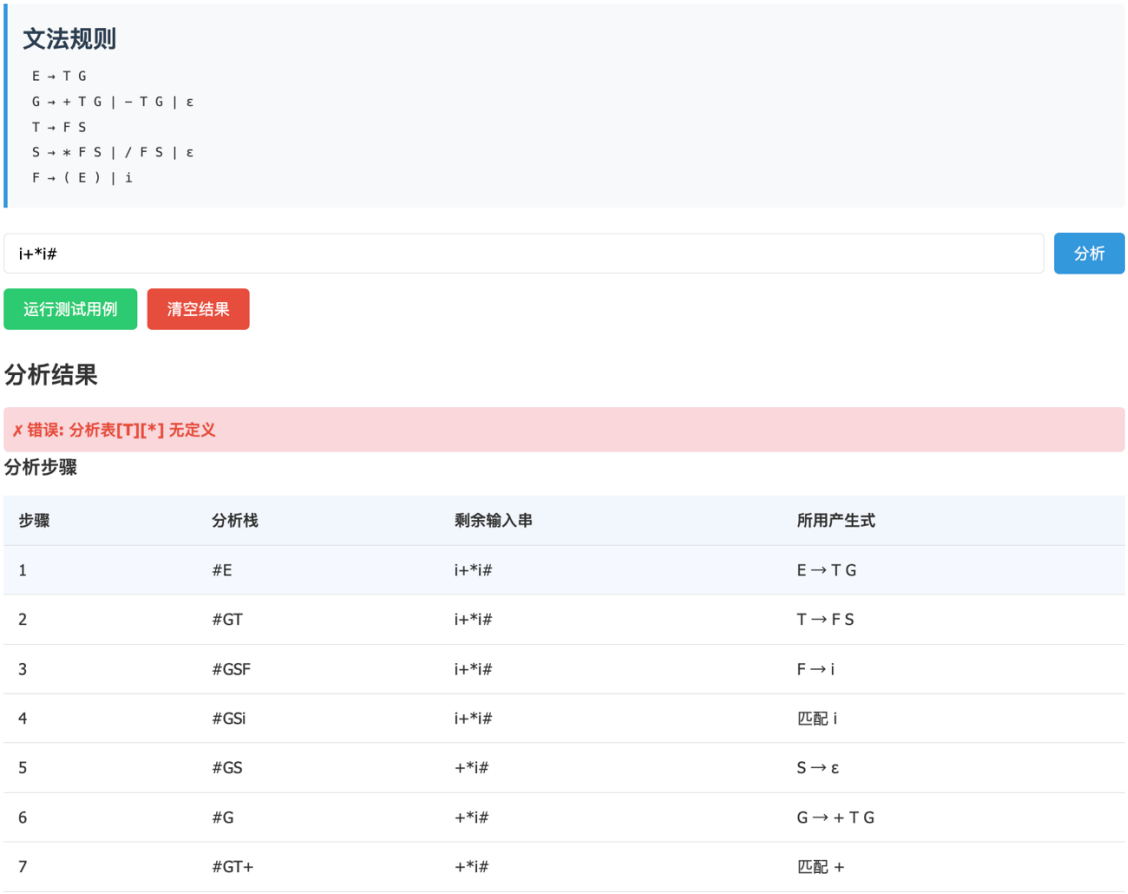
非法的串：

==========

输入符号串：i*i+#

```
========================================
步骤 | 分析栈              | 剩余输入串      | 所用产生式
-----|--------------------|---------------|--------------------
   1 | #E                 | i*i+#         | E -> T G
   2 | #GT                | i*i+#         | T -> F S
   3 | #GSF               | i*i+#         | F -> i
   4 | #GSi               | i*i+#         | 匹配 i
   5 | #GS                | *i+#          | S -> * F S
   6 | #GSF*              | *i+#          | 匹配 *
   7 | #GSF               | i+#           | F -> i
   8 | #GSi               | i+#           | 匹配 i
   9 | #GS                | +#            | S -> ε
  10 | #G                 | +#            | G -> + T G
  11 | #GT+               | +#            | 匹配 +
```

错误：分析表[T][#] 无定义



四、实验感悟

通过本次实验，不仅实现了 LL(1)预测分析器，更重要的是加深了对自上而下语法分析方法的理解，为后续学习更复杂的编译技术奠定了基础。实验过程中遇到的种种问题也锻炼了调试能力和系统思维能力。

实验 4：语法分析 3-LR(1)分析程序

一、实验内容、目的

1） 实验目的：通过完成预测分析的语法分析程序，了解预测分析法和递归子程序法的区别和联系。

2） 实验内容；构造预测分析程序并利用分析表和一个栈来实现对上程序设计语言的分析程序。

3） 实验要求：根据某一文法编制调试 LR(1)分析程序，以便对任意输入的符号串进行。

二、实验过程、代码

LR 分析器由三个部分组成：

(1) 总控程序，也可以称为驱动程序。对所有的 LR 分析器总控程序都是相同的。

(2) 分析表或分析函数，不同的文法分析表将不同，同一个文法采用的 LR 分析器不同时，分析表将不同，分析表又可以分为动作表（ACTION）和状态转换（GOTO）表两个部分，它们都可用二维数组表示。

(3) 分析栈，包括文法符号栈和相应的状态栈，它们均是先进后出栈。分析器的动作就是由栈顶状态和当前输入符号所决定。

程序思路（仅供参考） ：

模块结构：

（1）定义部分：定义常量、变量、数据结构。

（2）初始化：设立 LR(1)分析表、初始化变量空间（包括堆栈、结构体、数组、临时变量等）；

（3）控制部分：从键盘输入一个表达式符号串；

（4）利用 LR(1)分析算法进行表达式处理：根据 LR(1)分析表对表达式符号串进行堆栈 （或其他）操作，输出分析结果，如果遇到错误则显示错误信息。

首先，我们写出了 LR(1)分析表，并提前计算好 SHIFT、ACCEPT 状态等，写在文件中，代码如下：

```
void initializeParsingTable() {
    // 扩展状态表大小
    action_table.resize(20);
    goto_table.resize(20);
    // 状态 0
    action_table[0][ID] = Action(SHIFT, 5);
    action_table[0][LPAREN] = Action(SHIFT, 4);
    goto_table[0][E] = 1;
    goto_table[0][T] = 2;
    goto_table[0][F] = 3;
    // 状态 1
    action_table[1][PLUS] = Action(SHIFT, 6);
```

```
action_table[1][MINUS] = Action(SHIFT, 7);
action_table[1][END] = Action(ACCEPT, 0);
// 状态 2
action_table[2][PLUS] = Action(REDUCE, 2);
action_table[2][MINUS] = Action(REDUCE, 2);
action_table[2][MULTIPLY] = Action(SHIFT, 8);
action_table[2][DIVIDE] = Action(SHIFT, 9);
action_table[2][RPAREN] = Action(REDUCE, 2);
action_table[2][END] = Action(REDUCE, 2);
// 状态 3
action_table[3][PLUS] = Action(REDUCE, 5);
action_table[3][MINUS] = Action(REDUCE, 5);
action_table[3][MULTIPLY] = Action(REDUCE, 5);
action_table[3][DIVIDE] = Action(REDUCE, 5);
action_table[3][RPAREN] = Action(REDUCE, 5);
action_table[3][END] = Action(REDUCE, 5);
// 状态 4
action_table[4][ID] = Action(SHIFT, 5);
action_table[4][LPAREN] = Action(SHIFT, 4);
goto_table[4][E] = 10;
goto_table[4][T] = 2;
goto_table[4][F] = 3;
// 状态 5
action_table[5][PLUS] = Action(REDUCE, 7);
action_table[5][MINUS] = Action(REDUCE, 7);
action_table[5][MULTIPLY] = Action(REDUCE, 7);
action_table[5][DIVIDE] = Action(REDUCE, 7);
action_table[5][RPAREN] = Action(REDUCE, 7);
action_table[5][END] = Action(REDUCE, 7);
// 状态 6
action_table[6][ID] = Action(SHIFT, 5);
action_table[6][LPAREN] = Action(SHIFT, 4);
goto_table[6][T] = 11;
goto_table[6][F] = 3;
// 状态 7
action_table[7][ID] = Action(SHIFT, 5);
action_table[7][LPAREN] = Action(SHIFT, 4);
goto_table[7][T] = 12;
goto_table[7][F] = 3;
// 状态 8
action_table[8][ID] = Action(SHIFT, 5);
action_table[8][LPAREN] = Action(SHIFT, 4);
```

```
goto_table[8][F] = 13;
// 状态 9
action_table[9][ID] = Action(SHIFT, 5);
action_table[9][LPAREN] = Action(SHIFT, 4);
goto_table[9][F] = 14;
// 状态 10
action_table[10][PLUS] = Action(SHIFT, 6);
action_table[10][MINUS] = Action(SHIFT, 7);
action_table[10][RPAREN] = Action(SHIFT, 15);
// 状态 11
action_table[11][PLUS] = Action(REDUCE, 0);
action_table[11][MINUS] = Action(REDUCE, 0);
action_table[11][MULTIPLY] = Action(SHIFT, 8);
action_table[11][DIVIDE] = Action(SHIFT, 9);
action_table[11][RPAREN] = Action(REDUCE, 0);
action_table[11][END] = Action(REDUCE, 0);
// 状态 12
action_table[12][PLUS] = Action(REDUCE, 1);
action_table[12][MINUS] = Action(REDUCE, 1);
action_table[12][MULTIPLY] = Action(SHIFT, 8);
action_table[12][DIVIDE] = Action(SHIFT, 9);
action_table[12][RPAREN] = Action(REDUCE, 1);
action_table[12][END] = Action(REDUCE, 1);
// 状态 13
action_table[13][PLUS] = Action(REDUCE, 3);
action_table[13][MINUS] = Action(REDUCE, 3);
action_table[13][MULTIPLY] = Action(REDUCE, 3);
action_table[13][DIVIDE] = Action(REDUCE, 3);
action_table[13][RPAREN] = Action(REDUCE, 3);
action_table[13][END] = Action(REDUCE, 3);
// 状态 14
action_table[14][PLUS] = Action(REDUCE, 4);
action_table[14][MINUS] = Action(REDUCE, 4);
action_table[14][MULTIPLY] = Action(REDUCE, 4);
action_table[14][DIVIDE] = Action(REDUCE, 4);
action_table[14][RPAREN] = Action(REDUCE, 4);
action_table[14][END] = Action(REDUCE, 4);
// 状态 15
action_table[15][PLUS] = Action(REDUCE, 6);
action_table[15][MINUS] = Action(REDUCE, 6);
action_table[15][MULTIPLY] = Action(REDUCE, 6);
action_table[15][DIVIDE] = Action(REDUCE, 6);
```

```
    action_table[15][RPAREN] = Action(REDUCE, 6);
    action_table[15][END] = Action(REDUCE, 6);
}
```

类似的，我们可以写出获取产生式字符串的函数

```
std::string getProductionString(const Production& prod) {
    std::string result = symbolToString(prod.left) + " -> ";
    for (Symbol s : prod.right) {
        result += symbolToString(s);
    }
    return result;
}
```

三、实验结果

```
========================================
输入符号串: i+i*i#
========================================
步骤 | 状态栈          | 符号栈    | 剩余输入串      | 动作
----|----------------|----------|---------------|-------------------
   1 | 0              | $        | i+i*i#        | 移进 s5
   2 | 0 5            | $i       | +i*i#         | 归约 r7 (F -> i)
   3 | 0 3            | $F       | +i*i#         | 归约 r5 (T -> F)
   4 | 0 2            | $T       | +i*i#         | 归约 r2 (E -> T)
   5 | 0 1            | $E       | +i*i#         | 移进 s6
   6 | 0 1 6          | $E+      | i*i#          | 移进 s5
   7 | 0 1 6 5        | $E+i     | *i#           | 归约 r7 (F -> i)
   8 | 0 1 6 3        | $E+F     | *i#           | 归约 r5 (T -> F)
   9 | 0 1 6 11       | $E+T     | *i#           | 移进 s8
  10 | 0 1 6 11 8     | $E+T*    | i#            | 移进 s5
  11 | 0 1 6 11 8 5   | $E+T*i   | #             | 归约 r7 (F -> i)
  12 | 0 1 6 11 8 13  | $E+T*F   | #             | 归约 r3 (T -> T*F)
  13 | 0 1 6 11       | $E+T     | #             | 归约 r0 (E -> E+T)
  14 | 0 1            | $E       | #             | 接受
========================================
```

✓ 分析成功: i+i*i# 为合法符号串

```
(base) fang50253@MacBook-Pro 25-26-1编译原理 % cd "/Users/fang50253/Desktop/Files/Doc
nts/NJFU_My_Github/NJFU_CS_Note/25-26-1编译原理/lab4/" && g++ -std=c++14 lab4.cpp -o
4 && "/Users/fang50253/Desktop/Files/Documents/NJFU_My_Github/NJFU_CS_Note/25-26-1编译
理/lab4/"lab4
选择模式:
1. 交互式分析
2. 运行测试用例
3. 退出
请输入选择 (1-3): 1
请输入要分析的符号串 (以#结束): i+i*i#
========================================
LR(1)分析程序
编制人: 方泽宇, 2351610105, 23508014
========================================
输入符号串: i+i*i#
========================================
步骤 | 状态栈          | 符号栈    | 剩余输入串      | 动作
-----|-----------------|-----------|-----------------|--------------------
   1 | 0               | $         | i+i*i#          | 移进 s5
   2 | 0 5             | $i        | +i*i#           | 归约 r7 (F -> i)
   3 | 0 3             | $F        | +i*i#           | 归约 r5 (T -> F)
   4 | 0 2             | $T        | +i*i#           | 归约 r2 (E -> T)
   5 | 0 1             | $E        | +i*i#           | 移进 s6
   6 | 0 1 6           | $E+       | i*i#            | 移进 s5
   7 | 0 1 6 5         | $E+i      | *i#             | 归约 r7 (F -> i)
   8 | 0 1 6 3         | $E+F      | *i#             | 归约 r5 (T -> F)
   9 | 0 1 6 11        | $E+T      | *i#             | 移进 s8
  10 | 0 1 6 11 8      | $E+T*     | i#              | 移进 s5
  11 | 0 1 6 11 8 5    | $E+T*i    | #               | 归约 r7 (F -> i)
  12 | 0 1 6 11 8 13   | $E+T*F    | #               | 归约 r3 (T -> T*F)
  13 | 0 1 6 11        | $E+T      | #               | 归约 r0 (E -> E+T)
  14 | 0 1             | $E        | #               | 接受
========================================
✓ 分析成功: i+i*i# 为合法符号串
```

四、实验感悟

通过本次实验，不仅实现了 LR(1)预测分析器，更重要的是加深了对自上而下语法分析方法的理解，为后续学习更复杂的编译技术奠定了基础。实验过程中遇到的种种问题也锻炼了调试能力和系统思维能力。

实验 5：中间代码生成

一、实验内容、目的

1） 实验目的：通过完成逆波兰表达式分析程序，僚机中间代码生成。

2） 实验内容：翻译程序为逆波兰形式。

3） 实验要求：将非后缀式用来表示的算数表达式转换为用逆波兰式来表示的算数表达式的值。

二、实验预习

1） 逆波兰式定义

- 将运算对象写在前面，而把运算符号写在后面。用这种表示法表
- 示的表达式也称做后缀式。逆波兰式的特点在于运算对象顺序不变，运算符号位置反映运算顺序。采用逆波兰式可以很好的表示简单算术表达式，其优点在于易于计算机处理表达式。

2） 产生逆波兰式的前提

中缀算术表达式

3） 逆波兰式生成的实验设计思想及算法

- 首先构造一个运算符栈，此运算符在栈内遵循越往栈顶优先级越高的原则。
- 读入一个用中缀表示的简单算术表达式，为方便起见,设该简单算术表达式的右端多加上了优先级最低的特殊符号“#”。
- 从左至右扫描该算术表达式，从第一个字符开始判断，如果该字符是数字，则分析到该数字串的结束并将该数字串直接输出。
- 如果不是数字，该字符则是运算符，此时需比较优先关系。做法如下：将该字符与运算符栈顶的运算符的优先关系相比较。如果该字符优先关系高于此运算符栈顶的运算符，则将该运算符入栈。 倘若不是的话，则将此运算符栈顶的运算符从栈中弹出，将该字符入栈。
- 重复上述操作(1)-(2)直至扫描完整个简单算术表达式， 确定所有字符都得到正确处理，我们便可以将中缀式表示的简单算术表达式转化为逆波兰表示的简单算术表达式。

4） 逆波兰式计算的实验设计思想及算法

- 构造一个栈，存放运算对象。
- 读入一个用逆波兰式表示的简单算术表达式。
- 自左至右扫描该简单算术表达式并判断该字符，如果该字符是运算对象， 则将该字符入栈。 若是运算符， 如果此运算符是二目运算符，则将对栈顶部的两个运算对象进行该运算， 将运算结果入栈， 并且将执行该运算的两个运算对象从栈顶弹出。 如果该字符是一目运算符， 则对栈顶部的元素实施该运算，将该栈顶部的元素弹出，将运算结果入栈。
- 重复上述操作直至扫描完整个简单算术表达式的逆波兰式，确定所有字符都得到正确处理，我们便可以求出该简单算术表达式的值。

三、实验过程、代码

程序思路：

模块结构：

（1）定义部分：定义常量、变量、数据结构。
（2）初始化：设立算符优先分析表、初始化变量空间（包括堆栈、结构体、数组、临时变量等）；
（3）控制部分：从键盘输入一个表达式符号串；
（4）利用算符优先分析算法进行表达式处理：根据算符优先分析表对表达式符号串进行堆栈（或其他）操作，输出分析结果，如果遇到错误则显示错误信息。
（5）对生成的逆波兰式进行计算。

中缀表达式的验证：

```
function validateInfix(infix) {
    const parenStack = [];
    let lastWasOperator = true; // 开始时期望数字或左括号

    for (let i = 0; i < infix.length; i++) {
        const c = infix[i];

        if (/\d/.test(c)) {
            lastWasOperator = false;
        } else if (c === '(') {
            parenStack.push(c);
            lastWasOperator = true;
        } else if (c === ')') {
            if (parenStack.length === 0 || parenStack[parenStack.length - 1] !==
'(') {
                return false; // 括号不匹配
            }
            parenStack.pop();
            lastWasOperator = false;
        } else if (isOperator(c) && c !== '#' && c !== '(' && c !== ')') {
            if (lastWasOperator) {
                return false; // 连续运算符
            }
            lastWasOperator = true;
        } else if (!/\d/.test(c) && !isOperator(c)) {
            return false; // 非法字符
        }
    }

    return parenStack.length === 0 && !lastWasOperator;
}
```

逆波兰表达式的计算：

```
function calculatePostfix(postfix) {
    const numStack = [];
    let numberBuffer = '';

    for (let i = 0; i < postfix.length; i++) {
        const c = postfix[i];

        if (/\d/.test(c)) {
            // 处理数字
            numberBuffer += c;
        } else if (c === '&') {
            // 数字分隔符，将缓冲区数字入栈
            if (numberBuffer.length > 0) {
                numStack.push(parseFloat(numberBuffer));
                numberBuffer = '';
            }
        } else if (isOperator(c) && c !== '(' && c !== ')' && c !== '#') {
            // 处理运算符
            if (numberBuffer.length > 0) {
                numStack.push(parseFloat(numberBuffer));
                numberBuffer = '';
            }

            if (numStack.length < 2) {
                throw new Error("表达式错误：运算对象不足");
            }

            const b = numStack.pop();
            const a = numStack.pop();
            let result;

            switch (c) {
                case '+': result = a + b; break;
                case '-': result = a - b; break;
                case '*': result = a * b; break;
                case '/':
                    if (b === 0) throw new Error("数学错误：除零错误");
                    result = a / b;
                    break;
                default: throw new Error("未知运算符");
            }
```

```
            numStack.push(result);
        }
    }

    // 处理最后一个数字
    if (numberBuffer.length > 0) {
        numStack.push(parseFloat(numberBuffer));
    }

    if (numStack.length !== 1) {
        throw new Error("表达式错误：结果不唯一");
    }

    return numStack[0];
}
```

四、实验结果

样例 1:er421(错误样例)

样例 2：(28+68)*2#

五、实验感悟

1） 在实现中缀表达式转后缀表达式的过程中，我掌握了栈数据结构的核心应用。通过运算符优先级的比较和栈的压入弹出操作，我深刻体会到。

2） 通过亲手实现逆波兰式的转换和计算，我将抽象的数据结构概念转化为具体的代码实现。栈的"后进先出"特性在表达式转换中得到了完美体现，这比单纯学习理论更加深刻。