得分：_____

南京林業大學
NANJING FORESTRY UNIVERSITY

编译原理

学号：2351610105

姓名：方泽宇

班级：23508014

目录

一、 词法分析

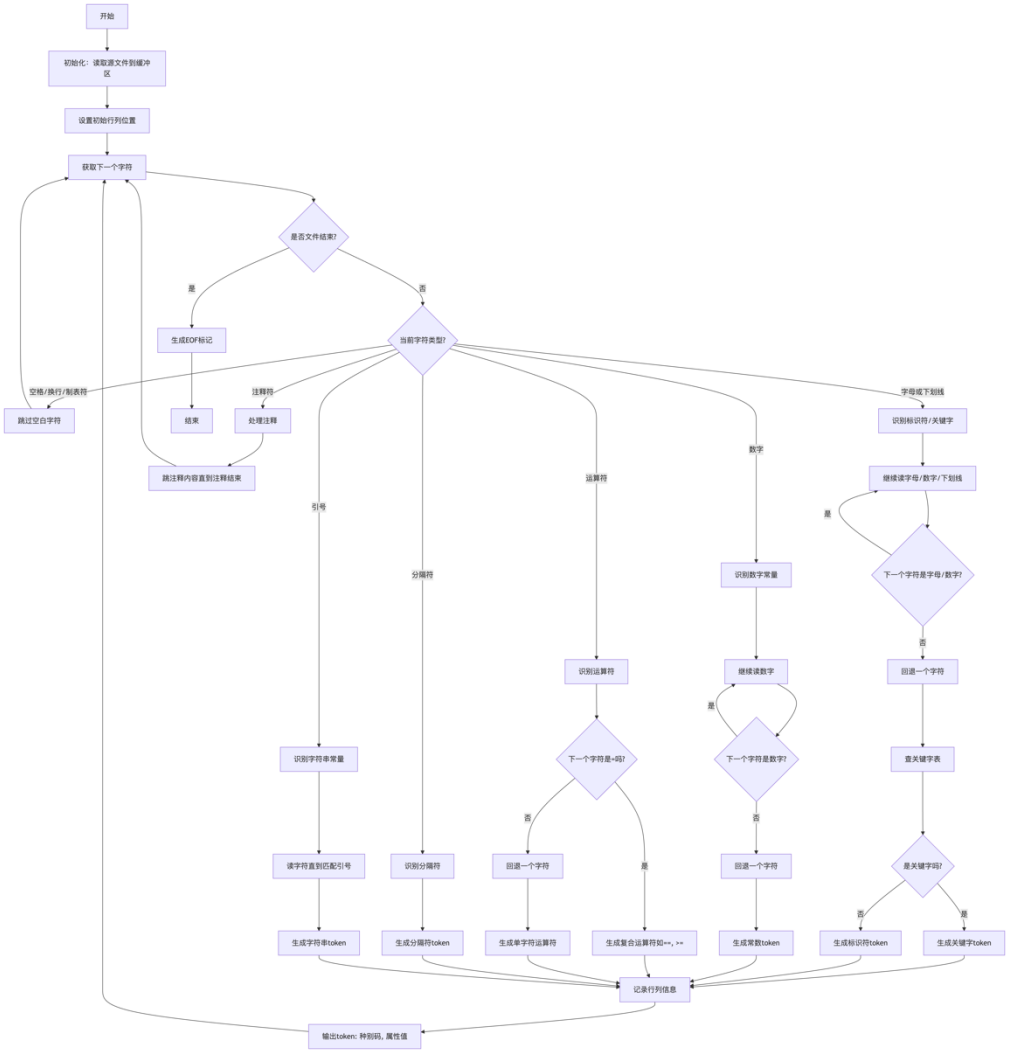
(一) 研究目的

1. 了解词法分析的过程 - 通过实际编写词法分析程序，深入理解编译器前端工作的基本原理。
2. 掌握字符流到内码的转换 - 学习如何将源程序的字符序列转换为有意义的单词序列

(二) 研究意义

1. 形式语言和自动机的理论验证。
2. 作为编译器前端设计的入门。

(三)　算法流程图



伪代码：

```
while (!endOfFile) {
        char ch = getNextChar();
        switch (getCharType(ch)) {
            case LETTER:
                processIdentifierOrKeyword();
```

```
                break;
            case DIGIT:
                processNumber();
                break;
            case OPERATOR:
                processOperator();
                break;
            // ... 其他情况
        }
    }
```

(四)　代码

因为原代码过长，仅展示核心代码(JavaScript)：

```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;
}
```

(五)　运行结果

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，"}"）

(六)　小结

1. 成功实现了 C 语言子集的词法分析器
2. 完成了字符流到单词内码的转换
3. 建立了完整的（种别码，属性值）二元式输出体系
4. 实现了关键字的准确识别和标识符的合理分类

二、 语法分析法 1-递归子程序法

(一)　研究目的

通过完成语法分析程序，了解语法分析的过程和作用。

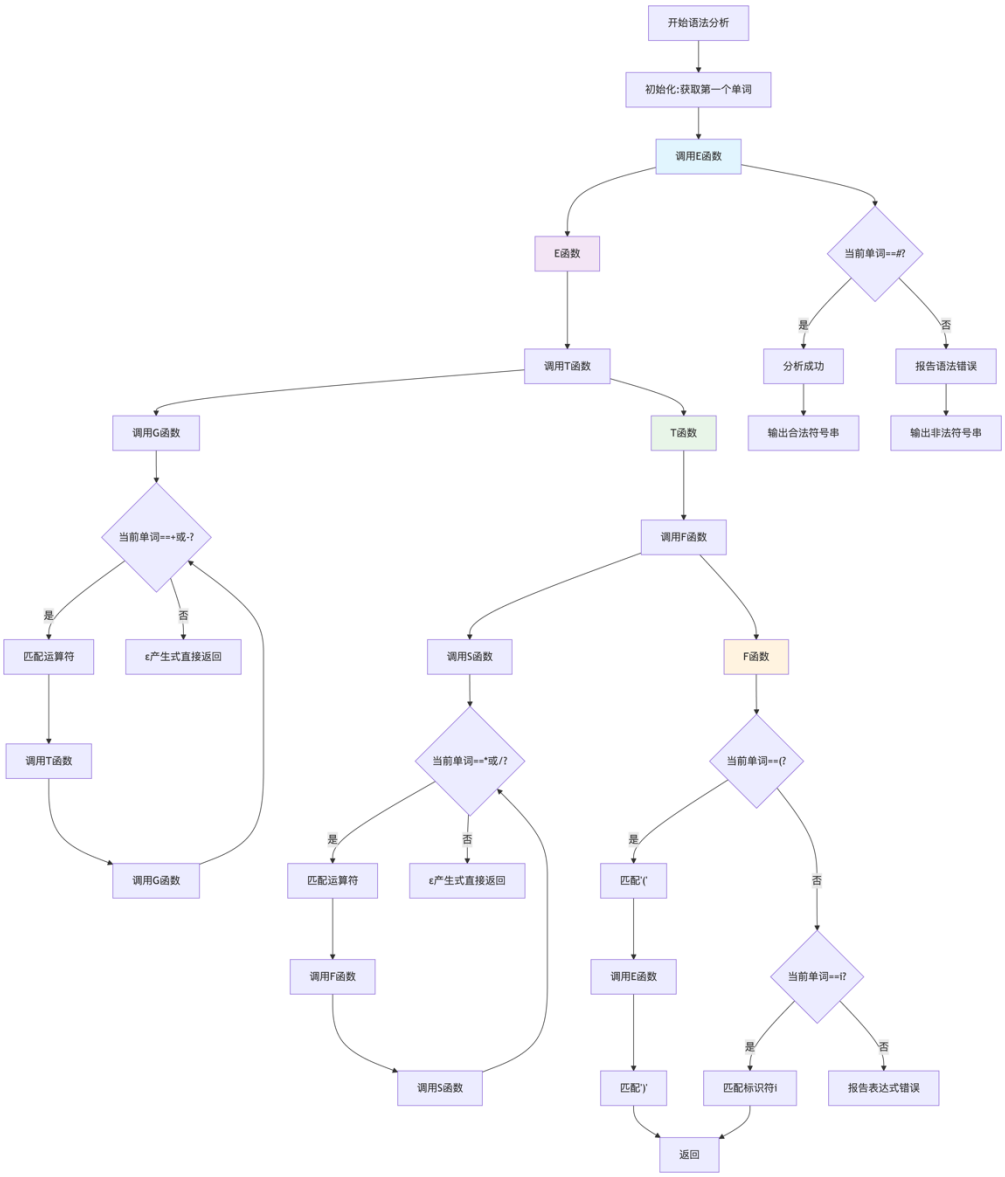
(二)　研究意义

1. 编译器构建核心技术的实践掌握

2. 形式文法的工程化应用

3. 自动机理论的深化理解

(三)　算法流程图



(四)　代码

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。

(五) 运行结果

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/lab2/"
main_ai
请输入要分析的符号串 (以#结束): i+i*(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*(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
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)

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

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“)”。

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
```

(六)　小结

1. 建立了编译原理理论到实践的完整桥梁。

2. 培养了复杂系统设计的工程思维。

3. 掌握了递归算法解决实际问题的能力。

4. 为后续更复杂的编译技术学习奠定了坚实基础。

三、 语法分析法 2-预测分析法

(一)　研究目的

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。

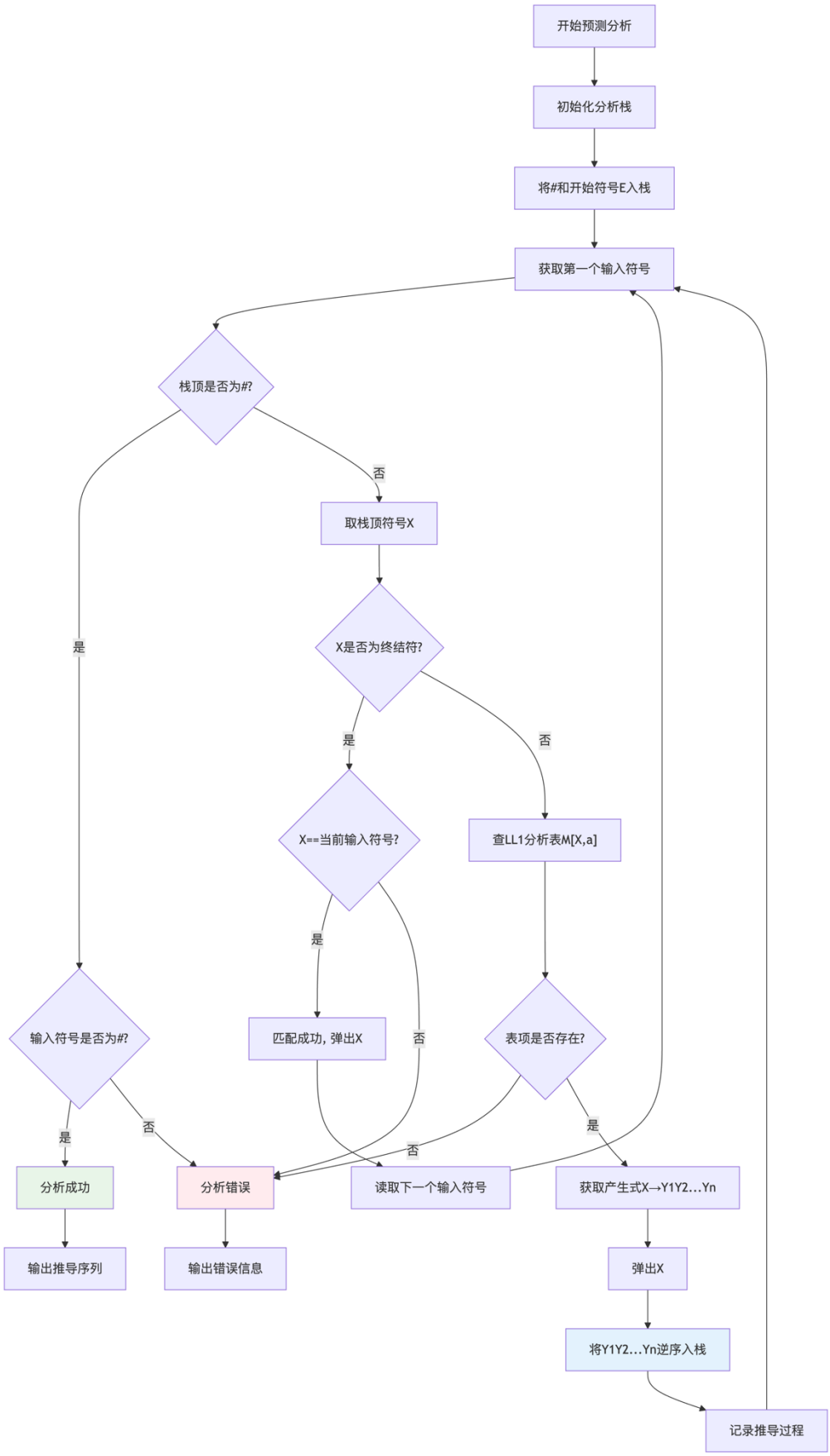
(二)　研究意义

1. 形式化分析方法的深度掌握

2. 自动机理论的系统化应用

3. 文法分析与改造技术

(三)　算法流程图


开始预测分析
初始化分析栈
将#和开始符号E入栈
获取第一个输入符号
栈顶是否为#?
是
否
取栈顶符号X
X是否为终结符?
是
否
X==当前输入符号?
查LL1分析表M[X,a]
是
否
输入符号是否为#?
匹配成功, 弹出X
表项是否存在?
是
否
否
是
分析成功
分析错误
读取下一个输入符号
获取产生式X→Y1Y2...Yn
输出推导序列
输出错误信息
弹出X
将Y1Y2...Yn逆序入栈
记录推导过程

(四)　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。

(五)　运行结果

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

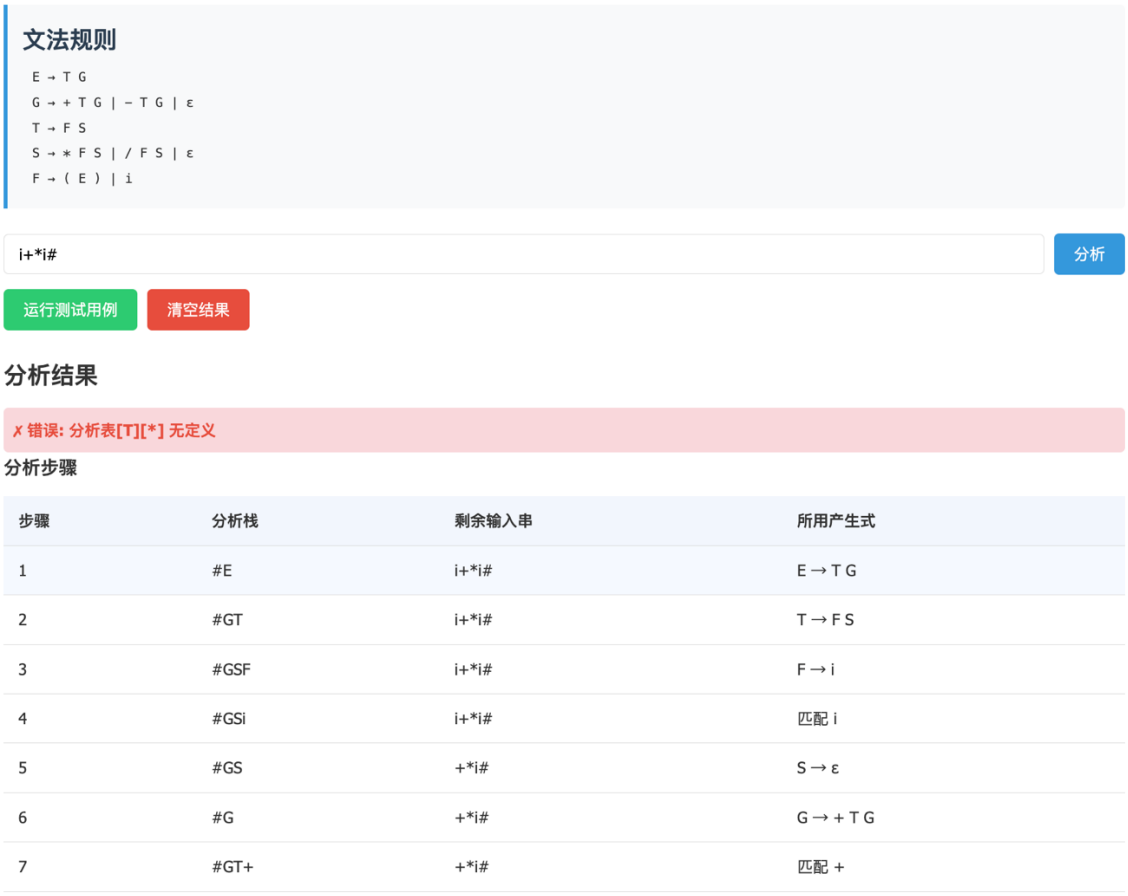
8 | #GSi | i+# | 匹配 i

9 | #GS | +# | S -> ε

10 | #G | +# | G -> + T G

11 | #GT+ | +# | 匹配 +

错误: 分析表[T][#] 无定义



(六) 小结

1. 技术掌握：深入理解了 LL(1)分析法的原理与实现
2. 工程实践：掌握了表驱动编程和栈管理的核心技术
3. 对比分析：清楚了预测分析法与递归子程序法的优劣
4. 基础奠定：为学习更复杂的 LR 分析方法提供了重要基础

四、 语法分析法 3-LR(1)分析程序

(一) 研究目的

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。

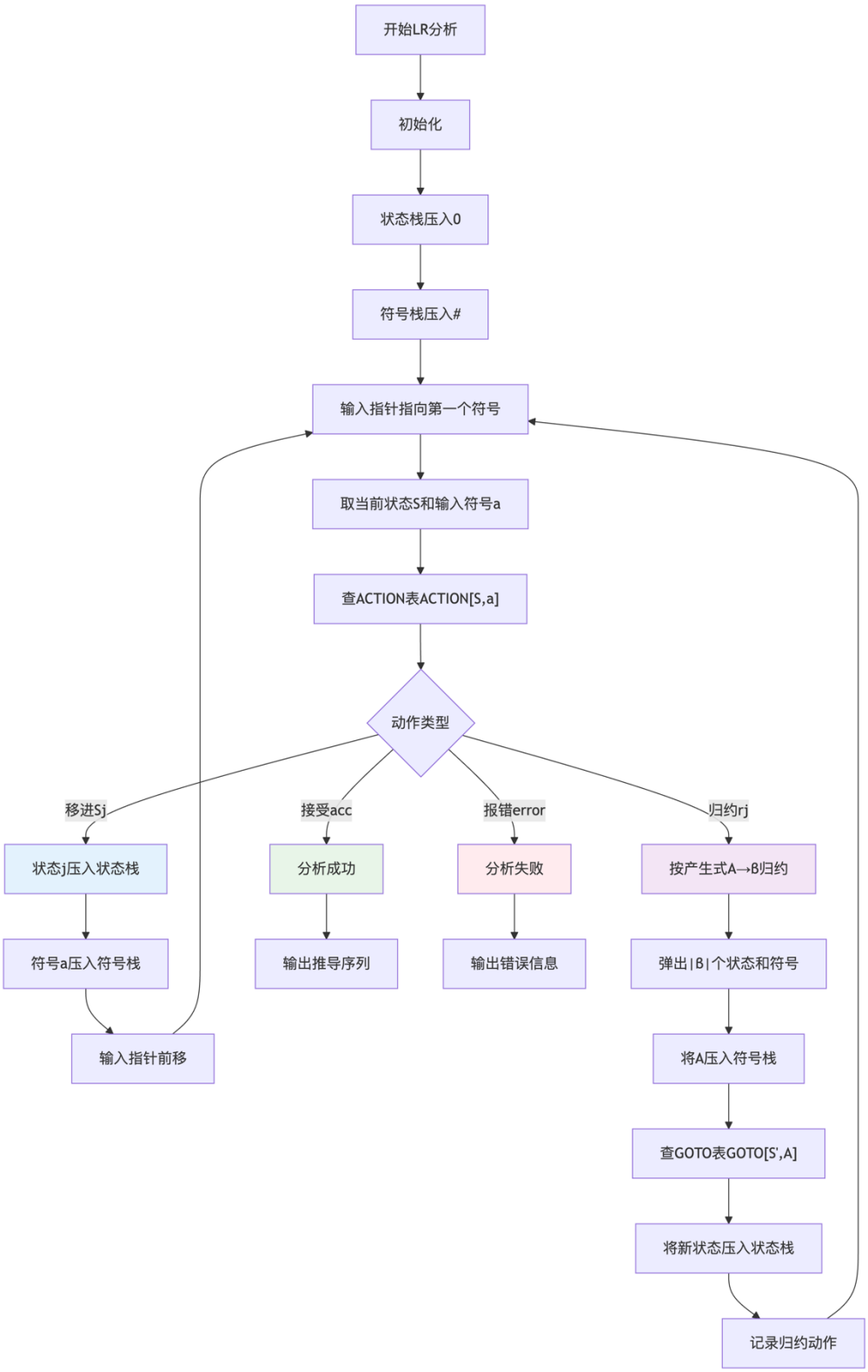
(二)　研究意义

1. 掌握 LR 分析原理

2. 文法分析能力扩展

3. 自动机理论深化

(三)　算法流程图



(四)　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;
}
```

(五)　运行结果

```
==========================================
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/Documents/NJFU_My_Github/NJFU_CS_Note/25-26-1编译原理/lab4/" && g++ -std=c++14 lab4.cpp -o lab4 && "/Users/fang50253/Desktop/Files/Documents/NJFU_My_Github/NJFU_CS_Note/25-26-1编译原理/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
```

(六) 小结

1. 技术巅峰：掌握了工业级编译器最常用的语法分析技术

2. 理论贯通：理解了自底向上分析与自顶向下分析的本质区别

3. 能力跃升：实现了从简单算法到复杂系统开发的跨越

4. 基础奠定：为后续的语义分析、代码生成等阶段提供了坚实基础

五、 中间代码生成

(一) 研究目的

通过完成逆波兰式分析程序， 了解中间代码的生成。

(二) 研究意义

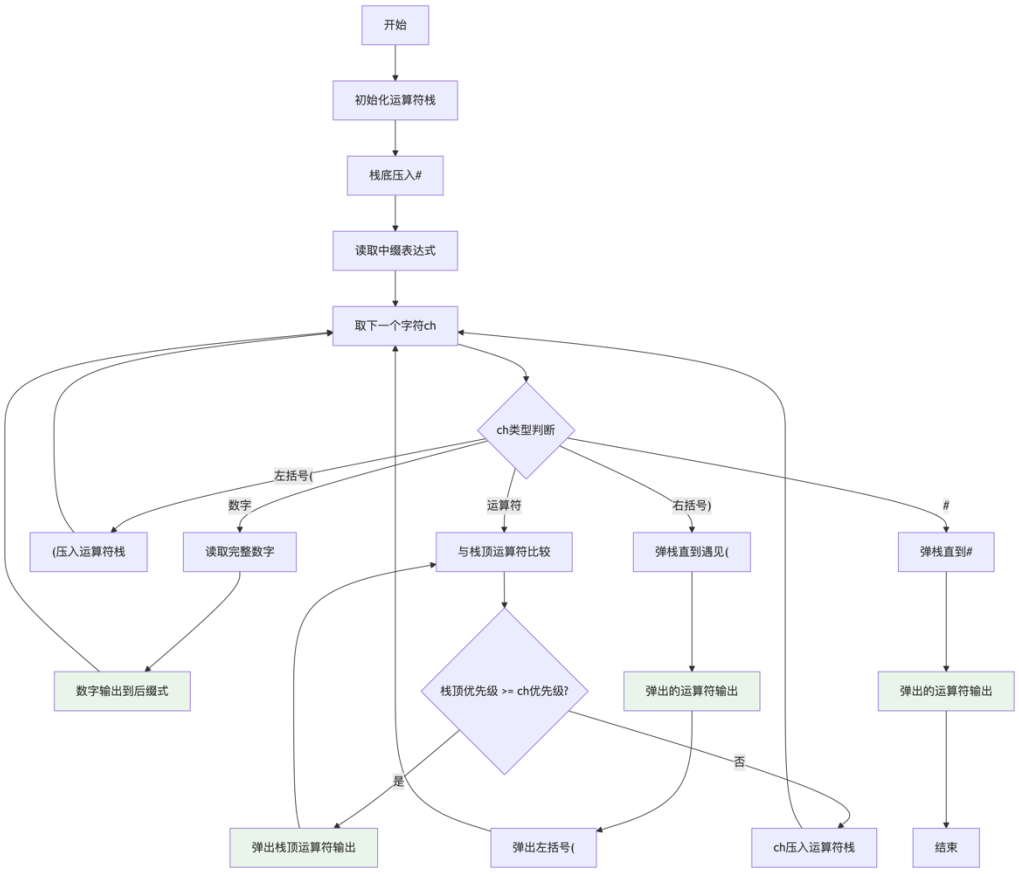
1. 中间代码生成原理的实践掌握

2. 编译过程中间表示的桥梁作用

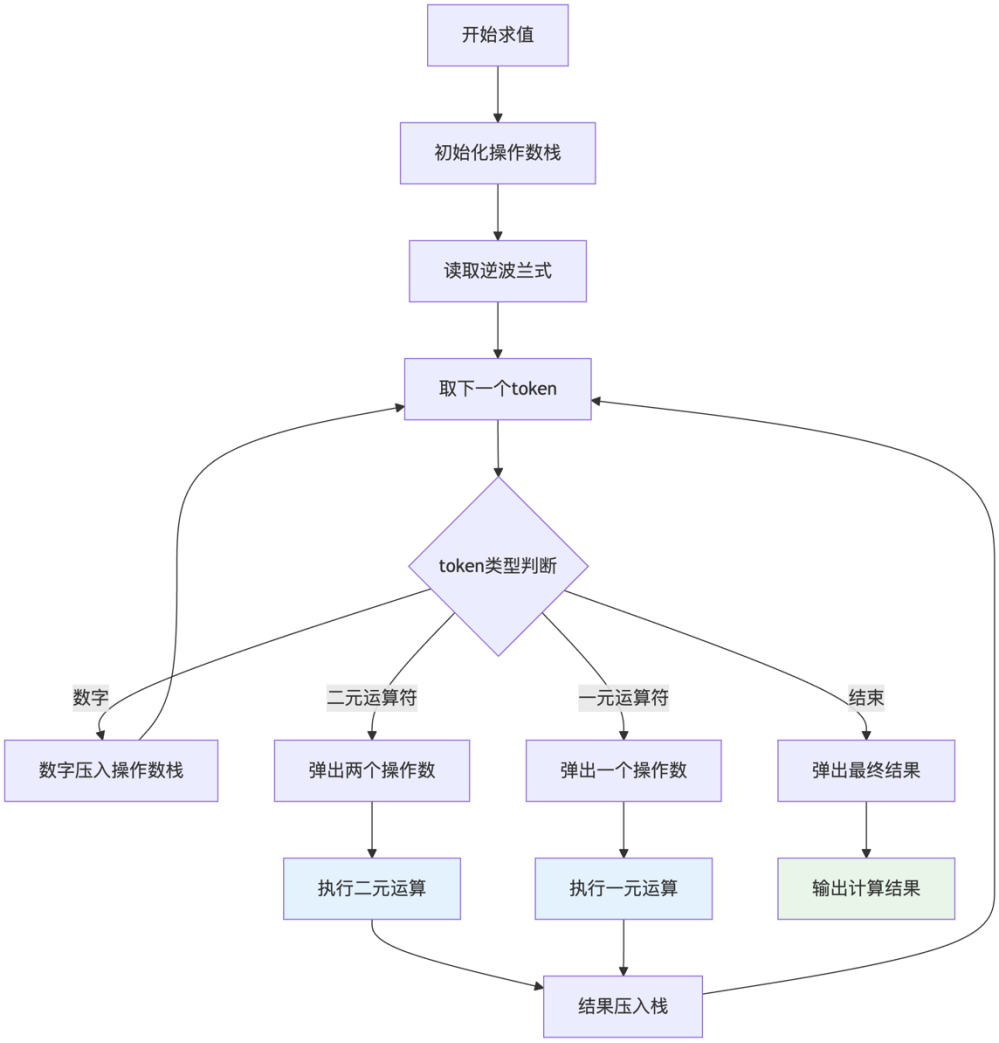
3. 算符优先分析法的具体应用

(三) 算法流程图

逆波兰式生成流程图：


开始
初始化运算符栈
栈底压入#
读取中缀表达式
取下一个字符ch
ch类型判断
左括号(
数字
运算符
右括号)
#
(压入运算符栈
读取完整数字
与栈顶运算符比较
弹栈直到遇见(
弹栈直到#
数字输出到后缀式
栈顶优先级 >= ch优先级?
弹出的运算符输出
弹出的运算符输出
是
否
弹出栈顶运算符输出
弹出左括号(
ch压入运算符栈
结束

逆波兰式求值流程图:



(四)　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];
}
```

(五)　运行结果

样例 1:er421(错误样例)

样例 2：(28+68)*2#

(六)　小结

1. 技术里程碑：完成了从源代码到中间代码的完整转换过程

2. 算法掌握：深入理解了栈在表达式处理中的核心作用

3. 工程实践：掌握了语法指导翻译的基本实现方法

4. 基础奠定：为后续目标代码生成提供了理论和技术准备

六、 Linux 了解和使用

(一)　研究目的

初步学习 linux 的使用

(二)　研究意义

初步掌握 linux 终端常见命令

(三)　算法流程图

（纯操作，无流程图）

(四)　代码

无代码

(五)　运行结果

练习:

1. Linxu 系统由那几部分组成，他们的依赖关系?

1）Linux 内核：内核是系统的核心，直接运行在硬件之上。它充当硬件和应用程序之间的桥梁，负责管理进程调度、内存管理、文件系统、设备驱动和网络通信等核心功能。上层组件完全依赖于内核来访问和操作硬件资源。

2）Shell 与系统工具：Shell（如 Bash）是一个命令解释器，为用户（特别是系统管理员）提供了与内核交互的界面。同时，系统还包含了一系列核心工具（如 ls, cp, grep 等，通常是 GNU 工具集），用于执行文件操作、文本处理和系统管理等任务。这些工具和 Shell 都依赖于内核提供的系统调用来实现其功能。

3）应用程序：这是用户直接接触的顶层，包括图形桌面环境（如 GNOME）、办公软件、浏览器等。这些应用程序依赖于 Shell 和

系统工具提供的运行环境，并最终通过调用内核的服务来完成具体任务。

2. 简答 Linux 系统的启动过程?

1） BIOS/UEFI 初始化：通电后，计算机首先运行主板上的固件程序（传统 BIOS 或现代 UEFI）。它进行硬件自检，然后根据预设的启动顺序，找到引导设备（如硬盘）并将控制权交给该设备上的引导加载程序。

2） 引导加载程序阶段：最常用的引导加载程序是 GRUB。它被加载到内存后，会向用户显示一个启动菜单（如果有多个系统）。用户选择或默认等待后，GRUB 会从硬盘中加载指定的 Linux 内核镜像和初始内存盘到内存中，然后将控制权交给内核。

3） 内核初始化：内核首先解压自己，然后初始化系统的核心功能，如内存管理、进程调度等。接着，它会利用 initramfs 来挂载真正的根文件系统。initramfs 是一个临时的根文件系统，包含了在挂载真实根文件系统前所必需的核心驱动和工具。

4） Systemd 初始化阶段：内核挂载根文件系统后，会启动第一个用户空间进程（传统上是/sbin/init，现代发行版通常是 Systemd）。Systemd 是所有其他进程的父进程。它负责启动系统的各项服务和守护进程，如网络、图形登录管理器等，并最终将系统带入可用的多用户运行级别或图形化目标。

5） 登录界面：最后，Systemd 启动显示管理器，用户看到图形或命令行登录界面。用户成功登录后，系统启动过程完成。

3. Linux 内核的作用是什么？

1） 进程管理：内核负责创建、调度和终止进程（运行中的程序）。它使用高效的调度算法，在多个进程之间公平、合理地分配 CPU 时间片，实现多任务并行运行的假象。同时管理进程间的通信。

2） 内存管理：内核管理系统的物理内存和虚拟内存。它为每个进程分配独立的虚拟地址空间，通过分页和交换技术，将有限的物理内存高效地映射给多个进程使用，并保护进程的内存空间互不干扰。

3） 文件系统管理：内核提供了一个虚拟文件系统的抽象层，支持在多种物理介质（如硬盘、U 盘）上读写多种格式的文件系统（如 EXT4, XFS, NTFS）。它统一管理文件和目录的创建、删除、读写和权限控制。

4） 设备驱动与硬件管理：内核通过设备驱动程序来管理和控制所有的硬件设备（如磁盘、网卡、USB 设备）。它对外提供统一的接口，使上层应用程序无需关心硬件的具体细节，就能通过“打开-读写-关闭”的通用模式来访问设备。

5） 网络通信：内核实现了完整的网络协议栈（如 TCP/IP），处理数据包的发送和接收。它管理网络接口，负责路由选择、防火墙过滤等，为系统提供网络连接能力。

4. 实际练习一下本节的命令。

(六) 小结

学习的 linux 系统的基本操作。

七、 Vi 编辑器

(一) 研究目的

初步了解 vi 编辑器

(二) 研究意义

学习使用 vi 编辑器

(三) 算法流程图

（纯操作，无流程图）

(四) 代码

无代码

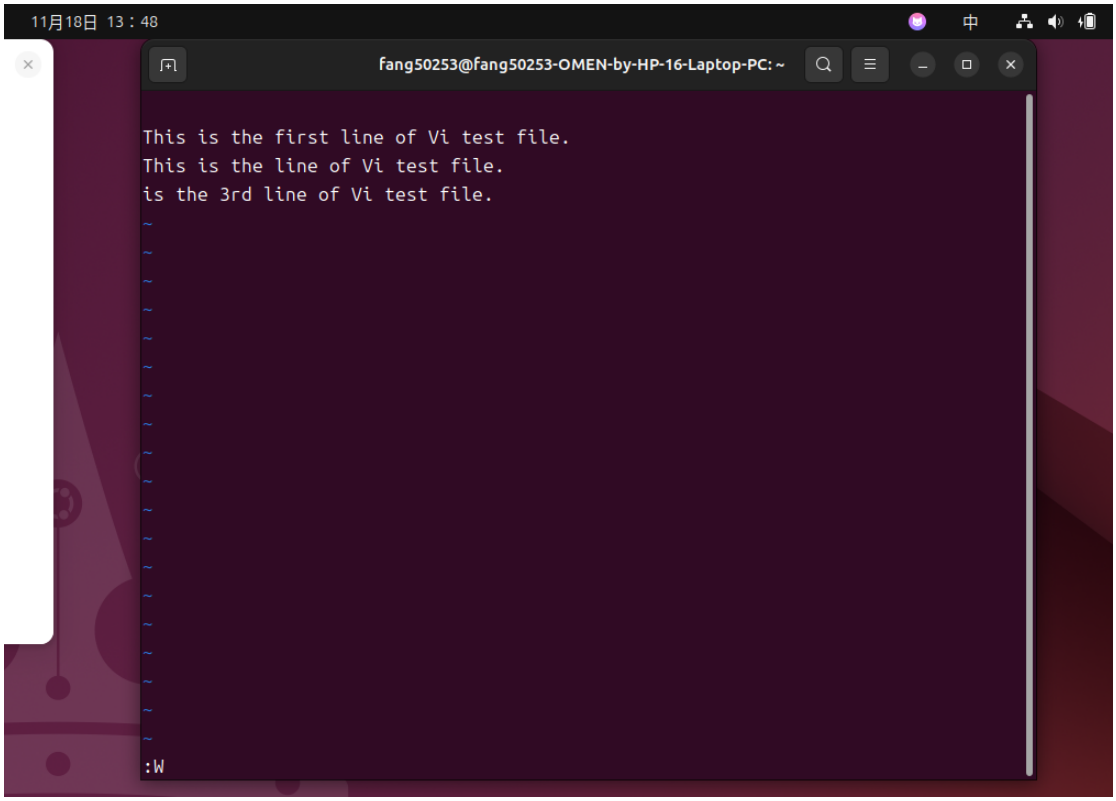
(五) 运行结果

1. 利用 VI 编辑文件，并且保存，修改文件内容?

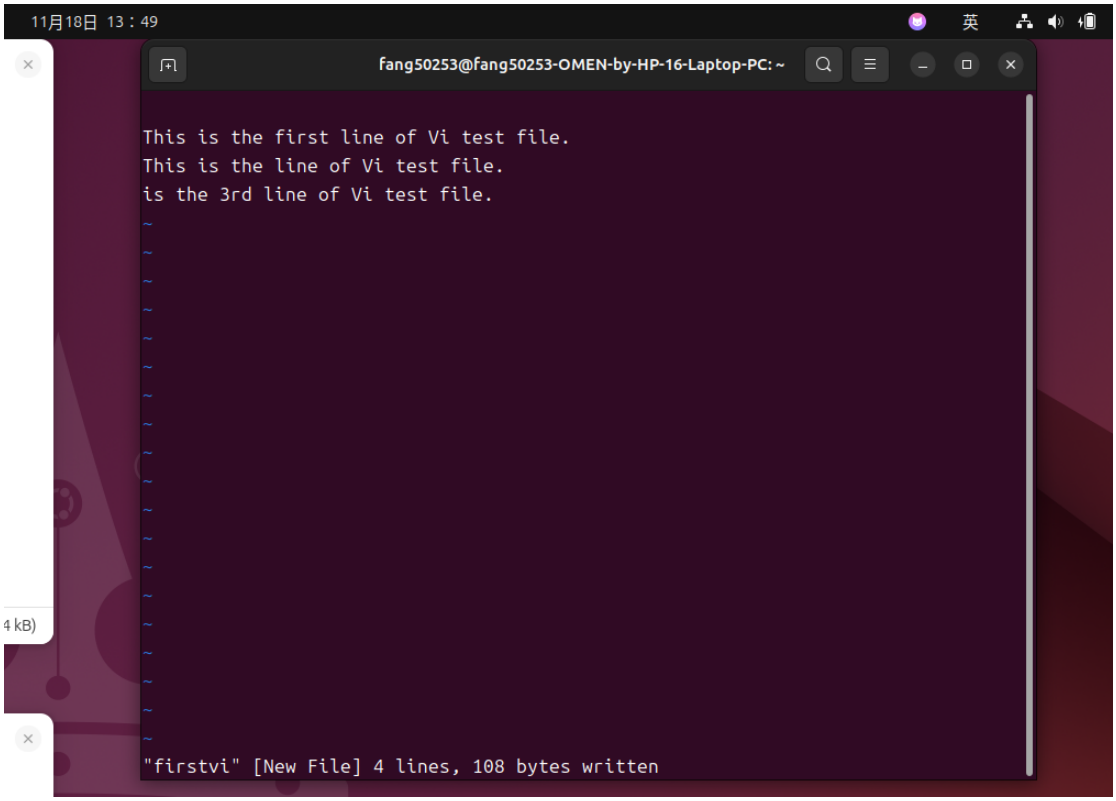
   (答案同 2)

2. 在 VI 命令模式下，试验本次内容的各种命令?

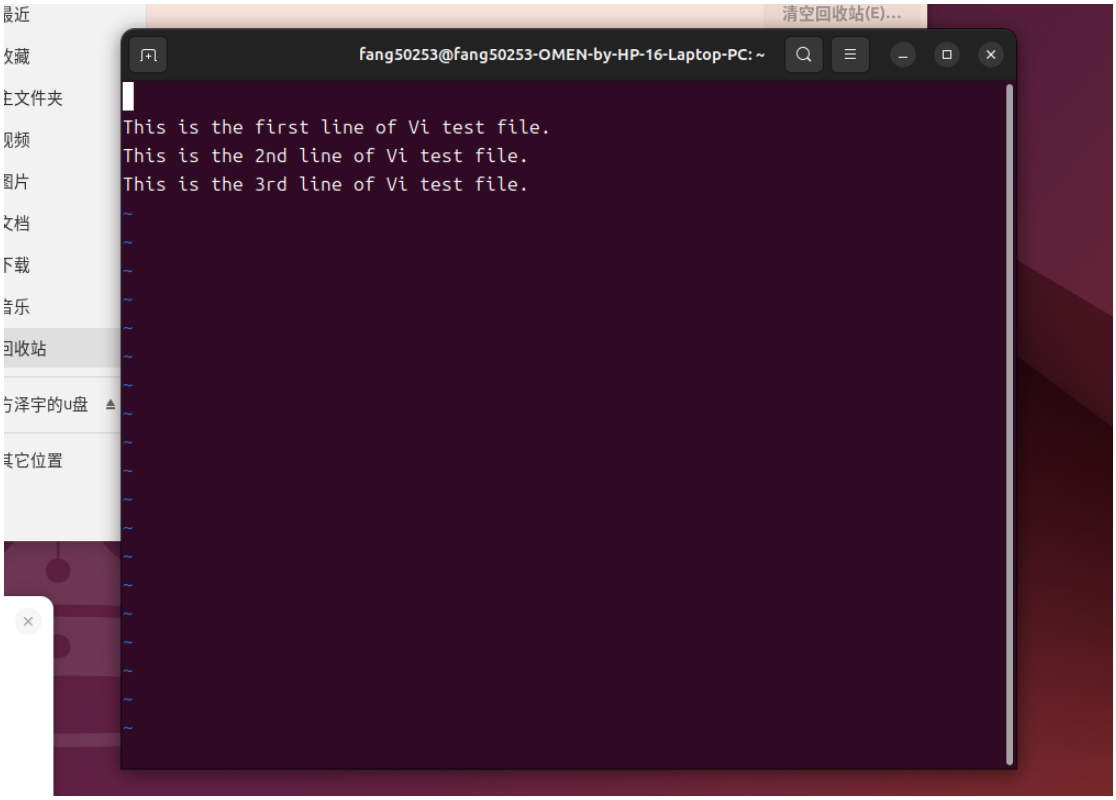
   步骤 6 之前的截图:



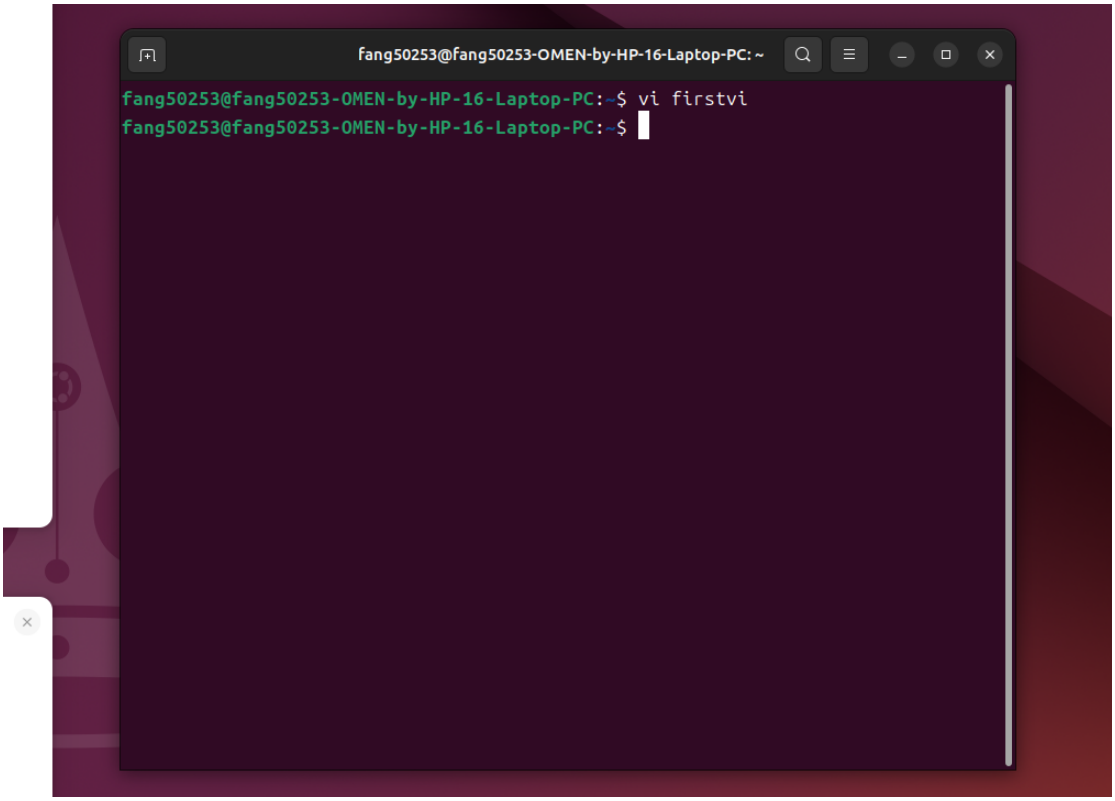
   步骤 6 后的截图:


11月18日 13：49
英
fang50253@fang50253-OMEN-by-HP-16-Laptop-PC: ~
This is the first line of Vi test file.
This is the line of Vi test file.
is the 3rd line of Vi test file.
"firstvi" [New File] 4 lines, 108 bytes written

步骤 17:


清空回收站(E)...
fang50253@fang50253-OMEN-by-HP-16-Laptop-PC: ~
This is the first line of Vi test file.
This is the 2nd line of Vi test file.
This is the 3rd line of Vi test file.

步骤 18:



(六)　小结

初步操作了 vi 编辑器，手册中有一个 i 指令写成了 l 指令，经过我的研究之后修正了。

八、 Lex 工具使用

(一)　研究目的

学会使用 Lex 工具。

(二)　研究意义

学习使用 Lex 工具。

(三)　算法流程图

（纯操作，无流程图）

(四)　代码

（无代码）

(五)　运行结果

1. 指出下列 LEX 正规式所匹配的字符串：

1) “{”[^{}]*”}”

匹配一个最简单的、不嵌套的花括号对及其内部内容。例如：{abc}, {123}, {}

2) ^[^a-z][A-Z][0-9]$

匹配恰好为 3 个字符且独占一行的字符串。第一个字符不是小写字母，第二个字符是大写字母，第三个字符是数字。例如：''A0'', ''@B9'', ''7X5''。

3) [^0-9][\r\n]

匹配一个非数字字符后紧跟一个行终止符的序列。例如：''a\n'', ''@\r'', ''z\n''

4) \'([^\n]|\'\')+\'

匹配一个单引号括起的字符常量。它允许字符串内部包含换行符之外的任何字符，并且使用两个连续的单引号 '' 来表示一个字面上的单引号字符（这是 SQL 等语言中的转义方式，不是 C 语言）。例如：'a', 'abc', 'don''t'（内部 '' 表示一个 ' 字符）。

5) \"([^"\n]|\\["\n])*\"

匹配一个双引号括起的字符串常量。它允许使用反斜线来转义双引号 \" 和换行符 \n，从而可以在字符串中包含这些特殊字符，并且字符串本身可以跨行。例如："hello", "say \"hi\"", "line1\nline2"。

2. 给出识别 C 语言全部实型的自动机?

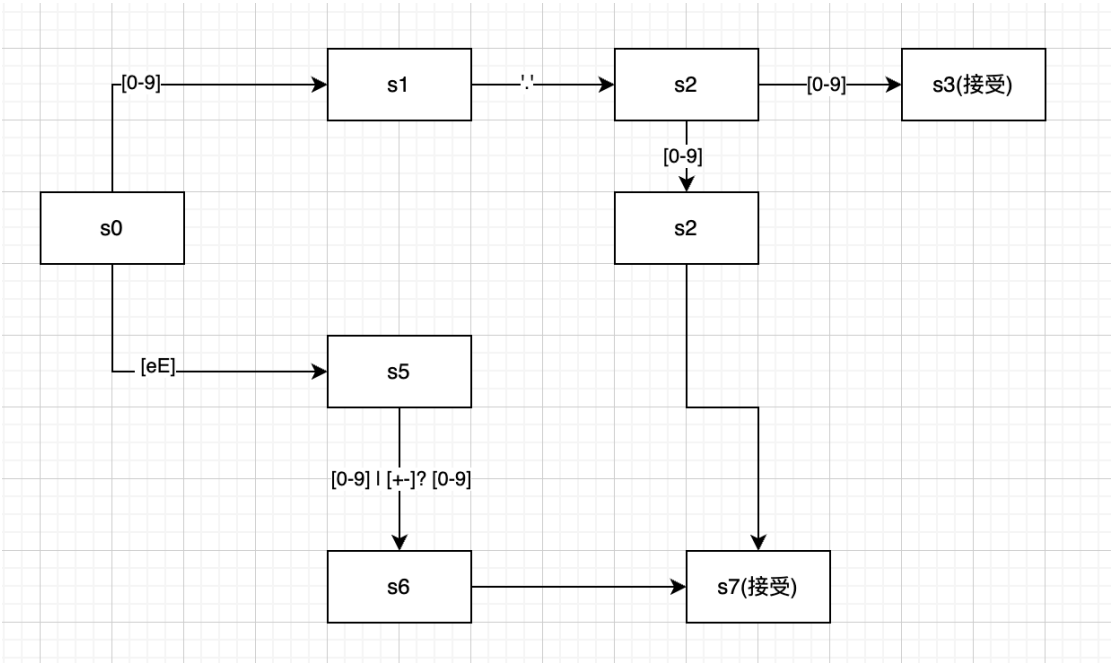
C 语言的实型（浮点数）主要有以下形式（以 E 或 e 表示指数）：

小数形式：123.45, .45, 123.

指数形式：123e+4, 123E-4, .45e10, 123.e5

十六进制浮点常量（C99）：0x1.2p3 (表示 (1 + 2/16) * 2^3)

这里给出识别十进制实型的简化自动机（忽略十六进制形式）：



状态说明:

S0: 初始状态。

S1: 接收到整数部分数字（如 123)。

S2: 接收到小数点，且前面有数字（如 123.)。

S3: 接收到小数点后的数字（如 123.45)。接受。

S4: 直接接收到小数点，然后接收到小数部分数字（如 .45)。接受。

S5: 接收到指数标志 e 或 E。

S6: 接收到可选的指数符号 + 或 -。

S7: 接收到指数部分的数字。接受。

从 S1, S3, S4 出发，遇到[eE]都能进入 S5，从而匹配指数形式。

3. 写出一个 LEX 正规式，它能匹配 C 语言的所用无符号整数。

(0|[1-9][0-9]*|0[0-7]+|0[xX][0-9a-fA-F]+)[uU]?[lL]?[lL]?

4. 写出一个 LEX 正规式，它能匹配 C 语言的标识符。

[a-zA-Z_][a-zA-Z0-9_]*

5. 写出一个 LEX 正规式，它将一个正文文件中的全部小写字符均换成大写字符，并将其中的制表字符，空白字符序列均列单个空格字符进行替换。 （提示:再语义动作中使用全称变量 YYTEXT）

```
%%
[a-z]      { putchar(toupper(yytext[0])); }     /* 小写转大写 */
[\t ]+     { putchar(' '); }                    /* 制表符和空格序列替换为单个空格 */
.|\n       { ECHO; }                            /* 其他字符原样输出 */
%%
```

6. 写出识别 C 语言中所有单词的 LEX 程序?

```
%{
#include <stdio.h>
%}
DIGIT      [0-9]
ID         [a-zA-Z_][a-zA-Z0-9_]*
%%
```

```
"/*"([^*]|("*"+[^*/]))*"*"+"/" ; /* 忽略多行注释 */
"//".*                                  ; /* 忽略单行注释 */
"auto"|"break"|"case"|"char"|"const"|"continue"|"default"|"do"|"double"
|"else"|"enum"|"extern"|"float"|"for"|"goto"|"if"|"int"|"long"|"register"|"r
eturn"|"short"|"signed"|"sizeof"|"static"|"struct"|"switch"|"typedef"|"uni
on"|"unsigned"|"void"|"volatile"|"while"      { printf("关键字: %s\n",
yytext); }
{ID}                                    { printf("标识符: %s\n",
yytext); }
{DIGIT}+                                { printf("整型常量: %s\n",
yytext); }
({DIGIT}+|{DIGIT}*\.{DIGIT}+)([eE][-+]?{DIGIT}+)?    { printf("实型常
量: %s\n", yytext); }
\"(\\.|[^"\\])*\"                       { printf("字符串常量: %s\n",
yytext); }
\'(\\.|[^'\\])?\'                       { printf("字符常量: %s\n",
yytext); }
"=="|"!="|"<="|">="|"++"|"--"|"&&"|"||"|"->"|"<<"|">>"|"+="|"-
="|"*="|"/="|"%="|"&="|"^="|"|="   { printf("运算符: %s\n", yytext); }
[+\-*/%=&|^<>!]                         { printf("运算符: %s\n",
yytext); }
[(){}[\];,.:?]                          { printf("分隔符: %s\n", yytext); }
```

```
[ \t\n]+                                    ; /* 忽略空白字符 */
.                                           { printf("无法识别的字
符: %s\n", yytext); }
%%
int main(int argc, char **argv) {
    yylex();
    return 0;
}
```

7. 试分析分隔符、空格、跳格及回车对词法分析的影响?

分隔符（Delimiters），如 ; , () {} []。

作用：它们是语言语法的一部分，具有明确的语义。它们标记了单词的边界，是词法分析器切分单词的重要依据。

空格、跳格（制表符）（White Space）作用：它们是词素（Token）之间的分隔符。在大多数情况下，它们本身不产生任何词素，唯一的功用是避免单词粘连，帮助词法分析器正确切分。

回车（换行符）（Newline）作用：除了具备与空格相同的分隔单词的功能外，它在某些语言中还有特殊意义。影响，在 C 语言中：换行符通常被视为普通的空白字符，被词法分析器忽略。但它对预处理指令（如 #define）是重要的，因为它们通常以换行符结束。在其他语言

中（如 Python）：换行符是语法的一部分，用于表示语句的结束，其影响远大于在 C 语言中。

总结：分隔符是产生词素的单词。而空格、跳格和回车在大多数情况下是词法分析器的“忽略”对象，它们的主要价值在于界定其他词素的边界，确保输入流能被正确切分成一个个有意义的单词。如果没有它们，词法分析将无法进行。

8. 编写一个 LEX 程序，它能统计一个 C 语言程序中所含的用户定义标识
符的个数，并能找出最长标识符中的字符个数。

```
%{
#include <stdio.h>
#include <string.h>
int identifier_count = 0;
int max_length = 0;
%}
DIGIT     [0-9]
ID          [a-zA-Z_][a-zA-Z0-9_]*
%%
"/*"([^*]|("*"+[^*/]))*"*"+"/" ; /* 跳过注释 */
"//".*                                    ; /* 跳过注释 */
```

```
/* C 语言关键字列表 */
"auto"|"break"|"case"|"char"|"const"|"continue"|"default"|"do"|"double"
|"else"|"enum"|"extern"|"float"|"for"|"goto"|"if"|"int"|"long"|"register"|"r
eturn"|"short"|"signed"|"sizeof"|"static"|"struct"|"switch"|"typedef"|"uni
on"|"unsigned"|"void"|"volatile"|"while"     { /* 不做任何事，忽略关键
字 */ }
{ID}       {
              identifier_count++;
              int len = strlen(yytext);
              if (len > max_length) {
                    max_length = len;
              }
              printf("找到标识符: %s (长度: %d)\n", yytext, len);
       }
[ \t\n]+                   ; /* 忽略空白 */
.                            ; /* 忽略其他所有字符 */
%%
int main(int argc, char **argv) {
       yylex();
       printf("\n===== 统计结果 =====\n");
       printf("用户定义标识符总数: %d\n", identifier_count);
       printf("最长标识符的字符个数: %d\n", max_length);
```

```
    return 0;
}
```

(六)　小结

学会使用 Lex 工具

九、 Yacc/Bision 工具

(一)　研究目的

学习使用 Yacc/Bision 工具

(二)　研究意义

学习使用 Yacc/Bision 工具

(三)　算法流程图

（纯操作，无流程图）

(四)　代码

（无代码）

(五)　运行结果

1. 给出识别 C 语言全部实型的自动机?

(([0-9]+\.|[0-9]*\.[0-9]+)([eE][+-]?[0-9]+)?) | ([0-9]+[eE][+-]?[0-9]+)

2. 写出识别 C 语言中所有单词的 LEX 程序?

```
%{
#include <stdio.h>
/* 在实际编译器中，这里会包含语法分析器的头文件，并定义 yylval 等 */
%}
/* 定义正则表达式的名字，方便后面使用 */
DIGIT       [0-9]
ID          [a-zA-Z_][a-zA-Z_0-9]*
WS          [ \t\n]
%%
/* 关键字 */
"auto"      | "break"     | "case"      | "char"      | "const"     |
"continue" | "default"   | "do"        | "double"    | "else"      |
"enum"      | "extern"    | "float"     | "for"       | "goto"      |
"if"        | "int"       | "long"      | "register" | "return"    |
"short"     | "signed"    | "sizeof"    | "static"    | "struct"    |
"switch"    | "typedef"   | "union"     | "unsigned" | "void"      |
"volatile" | "while"     { printf("<KEYWORD, %s>\n", yytext); }
/* 数据类型和标识符 */
```

```
{ID}            { printf("<IDENTIFIER, %s>\n", yytext); }
/* 整数常量 */
{DIGIT}+      { printf("<INTEGER_CONST, %s>\n", yytext); }
/* 实型常量 - 使用第一问中的逻辑 */
(([0-9]+"."[0-9]*)|([0-9]*"."[0-9]+))([eE][+-]?[0-9]+)? |
([0-9]+[eE][+-]?[0-9]+) { printf("<FLOAT_CONST, %s>\n", yytext); }
/* 字符串常量 (简化，不支持转义字符等) */
\"([^"\n]|\\")*\"      { printf("<STRING_LITERAL, %s>\n", yytext); }
/* 字符常量 */
\'[^'\n]\'              { printf("<CHAR_CONST, %s>\n", yytext); }
/* 运算符和分隔符 */
"+"|"-"|"*"|"/"|"="|"=="|"!="|"<"|">"|"<="|">="|"++"|"--"|
"&"|"|"|"!"|"&&"|"||"|"~"|"^"|"<<"|">>"|"%"|
"("|")"|"{"|"}"|"["|"]"|";"|","|"."|"->"|"?"|":"
{ printf("<OPERATOR/PUNCTUATOR, %s>\n", yytext); }
/* 注释 (忽略) */
"/*"([^*]|"*"[^/])*"*/"        /* 忽略多行注释 */
"//".*                          /* 忽略单行注释 */
/* 空白符 (忽略) */
{WS}            ; /* 不做任何操作，直接忽略 */
/* 其他任何未匹配的字符 */
.               { printf("Error: Unrecognized character %s\n", yytext); }
```

```
%%
int main(int argc, char **argv) {
    yylex();
    return 0;
}
int yywrap() {
    return 1;
}
```

3. 试分析分隔符、空格、跳格及回车对词法分析的影响?

分隔符（Delimiters），如 ; , () {} []。

作用：它们是语言语法的一部分，具有明确的语义。它们标记了单词的边界，是词法分析器切分单词的重要依据。

空格、跳格（制表符）（White Space）作用：它们是词素（Token）之间的分隔符。在大多数情况下，它们本身不产生任何词素，唯一的功用是避免单词粘连，帮助词法分析器正确切分。

回车（换行符）（Newline）作用：除了具备与空格相同的分隔单词的功能外，它在某些语言中还有特殊意义。影响，在 C 语言中：换行符通常被视为普通的空白字符，被词法分析器忽略。但它对预处理指令

（如 #define）是重要的，因为它们通常以换行符结束。在其他语言中（如 Python）：换行符是语法的一部分，用于表示语句的结束，其影响远大于在 C 语言中。

总结：分隔符是产生词素的单词。而空格、跳格和回车在大多数情况下是词法分析器的“忽略”对象，它们的主要价值在于界定其他词素的边界，确保输入流能被正确切分成一个个有意义的单词。如果没有它们，词法分析将无法进行。

4. 考虑问题：

① 编译程序的实现应考虑的问题有那些?

正确性:这是最基本的要求。编译器必须能够正确无误地将合法的源程序翻译成等价的目标程序。必须能清晰地报告源代码中的错误，并且不能因为遇到错误而产生崩溃或生成具有危险行为的目标代码。

编译效率:

编译速度：编译一个大型项目所需的时间。这对于开发效率至关重要。

目标代码效率：生成的目标代码（汇编或机器码）在运行时的速度和内存占用。这是优化器的主要目标。

可维护性和可扩展性:编译器本身也是一个大型软件，需要有清晰的结构（如清晰的阶段划分）、良好的代码设计和文档，以便于后续的 bug 修复、功能增加和目标机移植。

错误处理能力:

错误检测: 能检测出源代码中尽可能多的语法、语义错误。

错误恢复: 在发现一个错误后，能够从错误中恢复过来，继续分析后续的代码，以便在一次编译中发现多个错误。

清晰的错误信息: 提供准确、易懂的错误信息和精确的错误位置。

诊断和调试支持:

生成丰富的调试信息（如 DWARF 格式），使调试器能够将目标代码与源代码关联起来，支持单步执行、查看变量等。

可移植性:

宿主机的可移植性: 编译器本身能在不同平台（如 Windows, Linux, macOS）上运行。

目标机的可移植性: 能够为不同的硬件架构（如 x86, ARM, RISC-V）生成代码。这通常通过一个与目标机无关的中间表示和与目标机相关的后端来实现。

符合语言标准:

实现的语言特性必须符合官方语言标准（如 C11, C++17），同时处理好未定义行为、实现定义行为等。

用户友好性:

提供清晰的命令行界面、有用的警告信息、可配置的优化选项等。

② 编译程序的实现途径有那些?

手工编码:

描述: 使用通用的编程语言（如 C、C++、Java），完全通过手工编写代码来实现词法分析、语法分析、语义分析、中间代码生成、优化和目标代码生成等所有阶段。

优点:

高效: 可以针对特定情况进行高度优化，编译速度快，生成代码质量高。

灵活性和控制力强: 开发者对编译过程的每一个细节都有完全的控制权，便于实现复杂的优化和错误处理。

缺点:

开发效率低： 工作量大，开发周期长。

难以维护： 特别是语法分析器等部分，手写代码可能非常复杂且容易出错。

代表: GCC、Clang（虽然部分使用了自动生成工具，但其核心大量使用手写代码）、早期的编译器。

使用编译器构造工具:

描述: 使用专门的工具来自动生成编译器的某些部分。

词法分析器生成器: 如 LEX/Flex，根据正则表达式规则生成词法分析器。

语法分析器生成器: 如 YACC/Bison，根据上下文无关文法生成语法分析器（通常是 LALR(1)分析器）。

优点:

开发效率高： 大大减少了编写词法、语法分析器的工作量。

可靠性高： 生成的分析器经过充分测试，不易出错。

易于维护和修改： 修改语言规则时，只需更新规则文件并重新生成即可。

缺点:

灵活性受限： 工具本身的能力限制了能处理的文法的复杂度和能实现的错误恢复策略。

可能效率较低： 生成的代码可能不如精心手写的代码高效。

代表: 许多现代编译器（如 PL/I 的编译器）和解释器（如 Ruby 的早期版本）采用这种方式。

混合法:

描述: 这是目前最主流、最实用的方法。结合了上述两种途径的优点。

常见做法:

使用 Flex/Bison 等工具生成词法和语法分析器的框架。

在语法规则中嵌入手写的语义动作代码，来完成语义分析、中间代码生成等工作。

优化器和目标代码生成器等对性能和控制力要求高的部分，则主要采用手工编码实现。

优点:

在保证开发效率和可维护性的同时，又在关键部分保持了高性能和灵活性。

代表: 大多数现代编译器，如 Google 的 Go 语言编译器（gc）、PHP 的 Zend 引擎等，都采用这种混合模式。

(六)　小结

本实验学习了 Yacc/Bision 工具的使用。