

天津大学



编译原理大作业

C--编译器的实现

姓名： 陆子毅 郑力壘 武承霖 王鑫培 王嘉仑

学号： 3022206045 3022244344 3022244338 3022244362 3022244358

专业： 计算机科学与技术

2025 年 5 月 19 日

一、引言

1.1 背景知识

本次大作业旨在设计并实现一个针对 C--语言的编译器前端，并将 C--源代码编译生成 LLVM IR (Intermediate Representation, 中间表示)。

C--语言是 C 语言的一个简化子集，它保留了 C 语言的核心特性，如变量定义、函数、控制流 (if-else)、基本数据类型 (int, float) 和表达式运算等，但去除了诸如预处理指令 (#include, #define)、指针、结构体 (struct) 等复杂特性。C--语言源文件以 .sy 为后缀，采用单文件编译模式。

LLVM (Low Level Virtual Machine) 是一个模块化的、可重用的编译器和工具链集合。它提供了一套现代的、基于 SSA (Static Single Assignment) 的中间表示 LLVM IR，能够支持多种编程语言的静态和动态编译。LLVM IR 作为源语言和目标体系架构之间的桥梁，是本次作业最终需要生成的目标代码。

1.2 实验目标

本次大作业的主要目标是：

[必做] 实现 C--语言的词法分析器：基于自动机理论，能够识别 C--语言的关键字、标识符、运算符、界符、整数和浮点数，输出二元属性的 Token 序列，并管理符号表。

[必做] 实现 C--语言的语法分析器：采用自上而下或自下而上的语法分析方法，根据 C--语法规则，对词法分析器输出的 Token 序列进行分析，判断其语法正确性，并按最左推导或规范规约顺序生成语法树所用的产生式序列。

二、总体设计

2.1 编译器流程

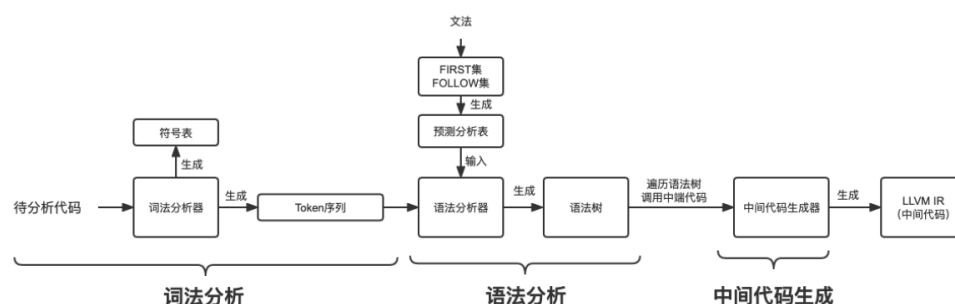
本编译器遵循经典的编译器设计流程，主要包括以下几个阶段（参考实验手册图 1）：

词法分析：读取 C--源代码文件 (.sy 文件)，将其分解为一系列的 Token（词法单元）。同时，在此阶段可以构建或更新符号表。

语法分析：以词法分析器输出的 Token 序列为输入，根据 C--语言的语法规则进行分析，检查语法的正确性。

中间代码生成（未完成）：遍历抽象语法树，将 C--语言的语义结构转换为 LLVM IR。此阶段会调用实验提供的中端代码库接口。

整体流程图如下所示



2.2 模块划分

根据编译器流程，我们将项目划分为以下主要模块：

词法分析模块 (Lexer): 负责将源代码字符流转换为 Token 流。

语法分析模块 (Parser): 负责根据语法规则解析 Token 流。

主程序模块 (Main/Driver): 负责协调各个模块的工作，处理输入输出。

三、词法分析器 (Lexer)

3.1 设计目标

词法分析器的核心任务是将输入的 C++ 源代码字符流（从 .sy 文件读取）转换为有意义的 Token 序列，为后续语法分析提供输入。具体实现位于 src/lexical/lexical.cpp 和 include/lexical/ 目录下，核心类为 LexicalAnalyzer（定义于 include/lexical/lexical.h）。

3.2 算法描述

词法分析器通过 LexicalAnalyzer 类实现，其核心逻辑在 scanToken() 方法中（src/lexical/lexical.cpp），模拟有限状态机（FSM）行为逐字符扫描输入流。

3.2.1 核心扫描算法

lexicalAnalysis 函数是词法分析的入口，负责逐字符读取文件、识别 Token 并输出结果。其核心流程如下：

1、初始化与文件打开

```
void lexicalAnalysis(string fileName) {
    FILE* fp;
    fp = fopen(lexicalTxtPath, "w");
    fwrite("", 0, 1, fp);
    ifstream file;
    file.open(fileName, std::ios::in);
    if (!file.is_open()) {
        cout << "Error: " << fileName << " could not open!" << endl;
    }
    // 重置行号
    lineNum = 0;
}
```

2、跳过空白字符：

```
while ((c = file.get()) != EOF) {
    if (c == ' ' || c == '\t' || c == '\n') {
        analyseToken(token);
        token = "";
        if (c == '\n') {
            lineNum++;
        }
        continue;
    }
}
```

遇到空格（）、制表符（\t）、换行符（\n）时，触发 analyseToken 处理已缓存的 Token，并重置缓冲区。换行符会递增 lineNum

3、Token 识别：在跳过空白和注释后，根据当前字符 (ch = is.get()) 进行分支判断：

```
// 使用自动机分析 token
void analyseToken(string token) {
    if (token.length() <= 0) {
        return;
    }
    // 先判断是不是关键字
    if (IS_All_Letter(token) && keyword.count(toLower(token))) {
        printToken(token, tokenCodeMap[token], lineNum);
        return;
    }
}
```

首先检查输入字符串是否符合标识符的构成规则（例如，以字母或下划线开头，后跟字母、数字或下划线）。如果符合，则进一步在预定义的关键字集合（如

`keyword` set, 定义了 "int", "if" 等) 中查找该字符串。若在关键字集合中找到, 则将其识别为相应的关键字 Token。若未找到, 则识别为普通标识符 Token。

```
else if (boundary.count(c)) { // 界符
    analyseToken(token);
    token = string(1, c);
    printToken(token, tokenCodeMap[token], lineNum);
    token = "";
    continue;
} else if (operation.count(c) ||
           operationBeginChar.count(c)) { // 运算符
    analyseToken(token);
    if (operationBeginChar.count(c)) { // 有可能是两个字符组成的运算符
        char nextChar = file.get();
        string tryOp = "";
        tryOp += c;
        tryOp += nextChar;
        if (operationOf2Char.count(
            tryOp)) { // 真的是两个字符组成的运算符
            token = tryOp;
            printToken(token, tokenCodeMap[token], lineNum);
        } else if (operation.count(
            c)) { // 普通的由一个字符组成的运算符
            token = string(1, c);
            printToken(token, tokenCodeMap[token], lineNum);
            file.putback(nextChar);
        } else {
            // 出错了, 错误 token
            token = string(1, c);
            printToken(token, TokenCode::UNDIFNIE, lineNum);
            file.putback(nextChar);
        }
    } else { // 这就是一个字符组成的运算符
        token = string(1, c);
        printToken(token, tokenCodeMap[token], lineNum);
    }
    token = "";
} else {
    token += c;
}
```

将输入字符串 (通常是单个字符) 与预定义的界符列表 (如 `(`, `)`, `{, `}` 等) 进行匹配。

将输入字符串与预定义的运算符列表进行匹配。对于可能由多个字符组成的运算符 (如 `==`, `<=`, `&&`), 词法分析器在构建词素时, 或者在 `analyseToken` 中, 会考虑向前查看或根据已形成的字符串长度和内容来确定是单字符运算符还是双字符运算符。例如, 遇到 `=` 时, 会检查之前或之后是否有另一个 `=` 来构成 `==`。

3.2.2 自动机驱动的 Token 识别 (analyseToken)

`analyseToken` 函数负责将缓冲区中的字符串交给自动机判断, 并输出 token

```
// 使用自动机分析 token
void analyseToken(string token) {
    if (token.length() <= 0) {
        return;
    }
    // 先判断好是不是关键字
    if (IS_All_Letter(token) && keyword.count(toLower(token))) {
        printToken(token, tokenCodeMap[token], lineNum);
        return;
    }
    int tokenCode = IDENTITY(miniDFA, token);
    printToken(token, tokenCode, lineNum);
    if (tokenCode == TokenCode::IDN) {
        symbolTable.addSymbol(token);
    }
}
```

自动机构建流程

```
FSM NFA = CREATE_NFA(); // 非确定自动机
FSM DFA = NFA_TO_DFA(NFA); // 有限自动机的确定化
FSM miniDFA = MIN_DFA(DFA); // 有限自动机的最小化
```

(1) 正则表达式到 NFA 的转换 (Thompson 构造法 - 理论)

算法描述 (Thompson 构造法) C--语言中的各种词法单元 (关键字、标识符、数字、运算符等) 都可以用正则表达式来描述。Thompson 构造法是一种经典的将正则表达式

转换为等价 NFA 的算法。

基本操作：

对于单个字符 'a'，构造一个 NFA，包含一个起始状态、一个接受状态和一条标记为 'a' 的边连接它们。

对于空串 ϵ ，构造一个 NFA，起始状态和接受状态相同，或通过 ϵ 边连接。

组合操作（递归定义）：

连接 (R1R2)：将 R1 的 NFA 的接受状态通过 ϵ 边连接到 R2 的 NFA 的起始状态。

新 NFA 的起始状态是 R1 的起始状态，接受状态是 R2 的接受状态。

选择 (R1|R2)：创建一个新的起始状态，用 ϵ 边分别连接到 R1 和 R2 的 NFA 的起始状态。创建一新的接受状态，R1 和 R2 的 NFA 的接受状态分别用 ϵ 边连接到这个新的接受状态。

闭包 (R*)：创建新的起始状态和接受状态。用 ϵ 边从新起始状态连接到 R 的 NFA 的起始状态和新接受状态。用 ϵ 边从 R 的 NFA 的接受状态连接到 R 的 NFA 的起始状态和新接受状态。

通过递归应用这些规则，可以为每个词法单元的正则表达式构建一个 NFA，然后将这些 NFA 组合起来（通常用一个新的起始状态通过 ϵ 边连接到所有词法单元 NFA 的起始状态）形成一个总的 NFA。

```
FSM CREATE_NFA() {
    set<set<char>> charList;
    set<char> letterList = {' '};
    for (char c = 'a'; c <= 'z'; c++) {
        letterList.insert(c);
    }
    for (char c = 'A'; c <= 'Z'; c++) {
        letterList.insert(c);
    }
    charList.insert(letterList);
    set<char> numList;
    for (char c = '1'; c <= '9'; c++) {
        numList.insert(c);
    }
    charList.insert(numList);
    set<char> zeroList = {'0'};
    charList.insert(zeroList);
    set<char> pointList = {'.'}; // 小数点
    charList.insert(pointList);

    FSM NFA = FSM(charList);
    //.....
}
```

(2) NFA 到 DFA 的转换 (子集构造法)

算法描述（子集构造法）：

1. 初始化 DFA：

计算 NFA 起始状态的 ϵ -闭包，这个集合成为 DFA 的第一个状态，并标记为未处理。

2. 迭代构建 DFA 状态与转移：

当存在未处理的 DFA 状态 'D_state'（它是一个 NFA 状态集合）时：

i. 标记 'D_state' 为已处理。

ii. 对于输入字母表中的每一个符号 'a'：

1. 计算 'move(D_state, a)'。

2. 计算结果集合的 ϵ -闭包，得到新的 NFA 状态集合 'U'。

3. 若 'U' 非空且之前未作为 DFA 状态出现过，则将 'U' 添加为新的 DFA 状态，并标记为未处理。

4. 在 DFA 中添加一条从 'D_state' 到 'U'（或代表 'U' 的 DFA 状态）的、标记为 'a' 的转移边。

3. 确定 DFA 的接受状态：任何包含至少一个 NFA 接受状态的 DFA 状态，都被标记

为 DFA 的接受状态。

此过程持续进行，直到没有新的 DFA 状态可以被生成。

```
FSM NFA_TO_DFA(FSM NFA) {
    FSM DFA = FSM(NFA.getCharList());

    stack<set<int>> nodes; // 遍历产生新节点的栈
    map<set<int>, int>
        statesMap; // 存储当前的 eplision-closure(move(I,a))对应的 DFA 中的标号，避免重复
    set<int> nowStates; // 当前 NFA 状态集的标号集合
    nowStates.insert(NFA.getFisrt()->id);
    statesMap.insert({nowStates, DFA.getFisrt()->id});
    nodes.push(nowStates);
    int newNum = 0;
    while (!nodes.empty()) {
        nowStates = nodes.top();
        nodes.pop();
        for (int id : nowStates) {
            FSM_Node* node = NFA.getNode(id);
            for (set<char> chList : NFA.getCharList()) {
                if (node->trans.count(chList)) { // 通过当前符号集有边能够转移
                    set<int> temp; // 当前状态集转移到的 I 的标号集
                    int tokenCode = TokenCode::UNDIFNIE;
                    bool isEnd = false;
                    for (unsigned int i = 0; i < node->trans[chList].size();
                        i++) {
                        FSM_Node* toNode =
                            node->trans[chList][i]; // 当前转换到的
                        if (toNode->state != TokenCode::UNDIFNIE) {
                            tokenCode = toNode->state;
                        }
                        if (toNode->isEndState) {
                            isEnd = true;
                        }
                        int toId = toNode->id;
                        temp.insert(toId);
                    }
                    if (statesMap.count(temp) ==
                        0) { // 到达了一个新的状态节点!
                        FSM_Node* newNode =
                            new FSM_Node(++newNum, tokenCode, isEnd);
                        statesMap.insert({temp, newNode->id});
                        DFA.addNode(newNode);
                        DFA.addTrans(statesMap[nowStates], newNode->id, chList);
                        nodes.push(temp); // 新发现的状态集需要加入栈中进行遍历
                    } else { // 之前就存在的状态集，添加边即可
                        int toId = statesMap[temp];
                        DFA.addTrans(statesMap[nowStates], toId, chList);
                    }
                }
            }
        }
    }
    return DFA;
}
```

(3) DFA 的最小化

1. 初始划分：将 DFA 的所有状态划分为接受状态组 'F' 和非接受状态组 'S-F'，构成初始分区 'P'。

2. 迭代细化分区：

- a. 创建一个工作列表 'W'，初始包含 'F' 和 'S-F'。
- b. 当 'W' 非空时：
 - i. 从 'W' 中取出一个组 'A'。
 - ii. 对于输入字母表中的每一个符号 'a'：

尝试用 '(A, a)' 来划分分区 'P' 中的所有组 'G'。若 'G' 被成功划分为多个子组，则在 'P' 中用这些子组替换 'G'，并将新产生的子组，加入 'W'（如果它们或其父组不在 'W' 中，或按特定规则更新 'W'）。

3. 构建最小化 DFA：当 'W' 为空时，'P' 中的每个组代表最小化 DFA 的一个状态。根据原 DFA 的起始状态、接受状态和转移关系，构建新的最小化 DFA。

```
FSM MIN_DFA(FSM DFA) {
    set<set<int>> partition; // 状态的划分
    set<int> endState, notEndState; // 初始两个状态
    for (int i = 0; i < DFA.getNum(); i++) {
        if (DFA.getNode(i)->isEndState) {
```

```

        endState.insert(i);
    } else {
        notEndState.insert(i);
    }
}
partition.insert(endState);
partition.insert(notEndState);
bool isChanged = true;
while (isChanged) { //.....具体见代码仓库

```

3.2.3 关键数据结构

关键字集合 `set<string> keyword = {"int", "void", "return", "const", "main", "float", "if", "else"};`

运算符集合

单字符运算符: `operation = {'+', '-', '*', '/', '%', '=', '>', '<', '!'}`

双字符运算符: `operationOf2Char = {"==", "<=", ">=", "!=", "&&", "||"}`

运算符起始字符: `operationBeginChar = {'=', '<', '>', '!', '&', '|'} (用于前瞻判断双字符运算符)`

界符集合 `set<char> boundary = {'(', ')', '{', '}', ';', ','};`

Token 类型映射表

```

map<string, int> tokenCodeMap = {
    {"int", TokenCode::KW_int}, {"void", TokenCode::KW_void},
    {"+", TokenCode::OP_plus}, {"==", TokenCode::OP_equal},
    // 其他 Token 类型映射...
};

```

3.3 单词符号的类型与格式

Token 定义 (Token.h):

Token 结构体存储了 TokenType type、std::string value 和 int line_num。

TokenType 枚举定义了所有 C--词法单元类型，与实验手册要求一致，包括：

关键字: INT_KW, VOID_KW, RETURN_KW, CONST_KW, MAIN_KW, FLOAT_KW, IF_KW, ELSE_KW

运算符: PLUS, MINUS, MUL, DIV, MOD, ASSIGN, EQ, NE, LT, GT, LE, GE, AND, OR, NOT

界符: LPAREN, RPAREN, LBRACE, RBRACE, SEMI, COMMA

标识符: IDN

字面量: INT_LIT, FLOAT_LIT

特殊: EOFF, ERROR

```

enum TokenCode {
    UNDEFNIE = 0, // 未定义
    /*KW 关键字*/
    KW_int,      // int
    KW_void,     // void
    KW_return,   // return
    KW_const,    // const
    KW_main,     // main
    KW_float,    // float
    KW_if,       // if
    KW_else,     // else
    /*OP 运算符*/
    OP_plus,     // +
    OP_minus,    // -
    OP_multiply, // *
    OP_divide,   // /
    OP_percent,  // %
    OP_assign,   // =
    OP_greater,  // >
    OP_less,     // <
    OP_equal,    // ==

```

```

OP_leq,      // <=
OP_geq,      // >=
OP_neq,      // !=
OP_and,      // &&
OP_or,       // ||
OP_not,      // !  添加逻辑非运算符
/*SE 界符*/
SE_lparent,  // (
SE_rparent,  // )
SE_lbraces,  // {
SE_rbraces,  // }
SE_semicolon, // ;
SE_comma,    // ,
IDN,        // 标识符: (Letter|_)(Letter|digit|_)*
INT,        // 整数: digit digit*
FLOAT       // 浮点数
};

```

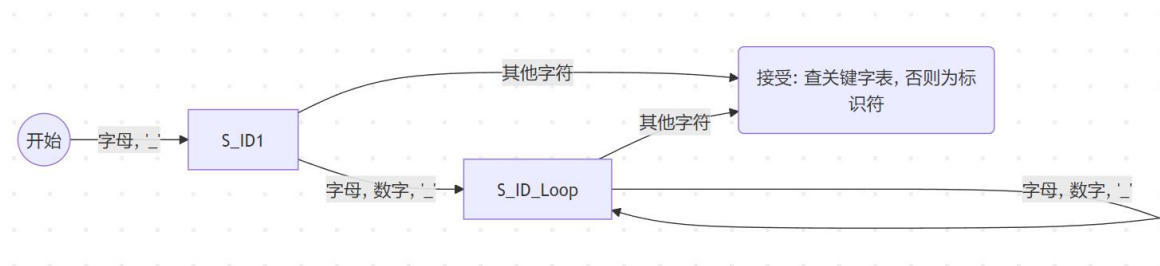
输出格式:

词法分析器输出遵循实验手册的格式:

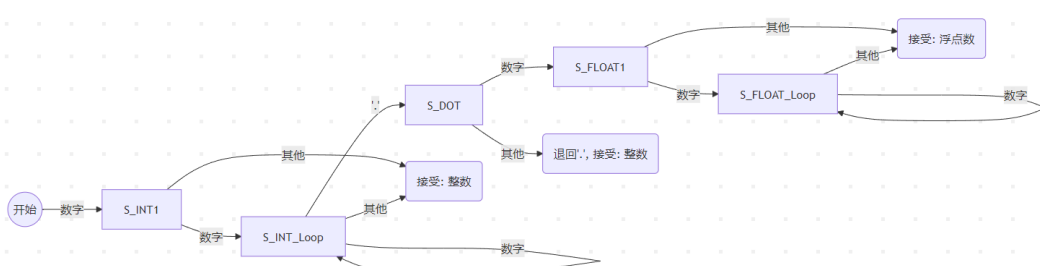
[待测代码中的单词符号] [TAB] <[单词符号种别],[单词符号内容]>

3.4 自动机状态转移逻辑

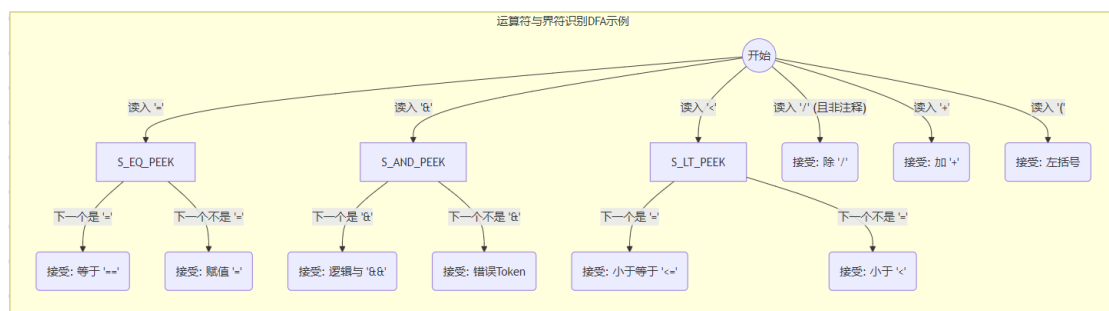
标识符/关键字:



数字 (整数/浮点数) :



运算符 (以 = 和 && 为例) :



3.5 源程序编译步骤 (词法分析阶段)

输入：在 `src/main.cpp` 中，程序首先通过命令行参数获取待编译的 C--源代码文件名。
初始化：创建一个 `std::ifstream` 对象以打开指定的源文件。若文件打开失败，则报告错误并退出。

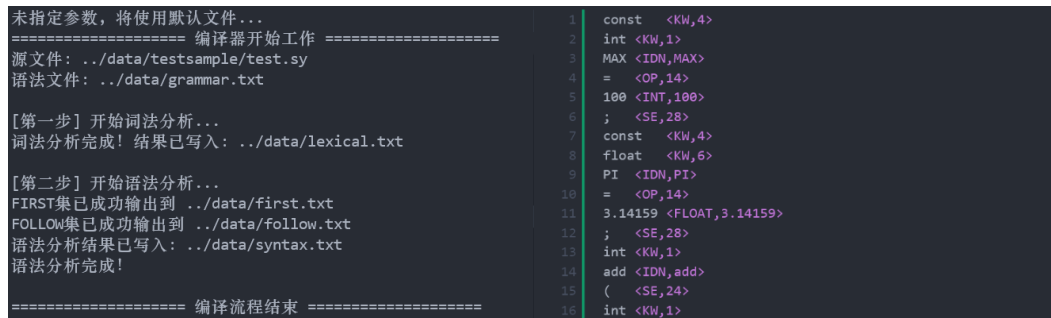
Token 序列生成与输出：

主程序 (`main.cpp`) 驱动词法分析器来逐个或批量获取 Token。

每当获取到一个 Token 对象后，程序会按照实验手册规定的格式（词素 [TAB] <类型, 内容>）将其信息输出到标准控制台或写入到指定的结果文件（如 `data/lexical.txt`）。这个过程持续到处理完所有输入字符并遇到文件结束标记（EOFF Token）。

```
// 第一步：词法分析
cout << "\n[第一步] 开始词法分析..." << endl;
lexicalAnalysis(sourceFileName);
cout << "词法分析完成！结果已写入：" << lexicalTxtPath << endl;
```

最终结果如图：



```
未指定参数，将使用默认文件...
===== 编译器开始工作 =====
源文件: ../data/testsample/test.sy
语法文件: ../data/grammar.txt

[第一步] 开始词法分析...
词法分析完成！结果已写入: ../data/lexical.txt

[第二步] 开始语法分析...
FIRST集已成功输出到 ../data/first.txt
FOLLOW集已成功输出到 ../data/follow.txt
语法分析结果已写入: ../data/syntax.txt
语法分析完成！

===== 编译流程结束 =====

1  const <KW,4>
2  int <KW,1>
3  MAX <IDN,MAX>
4  = <OP,14>
5  100 <INT,100>
6  ; <SE,28>
7  const <KW,4>
8  float <KW,6>
9  PI <IDN,PI>
10 = <OP,14>
11 3.14159 <FLOAT,3.14159>
12 ; <SE,28>
13 int <KW,1>
14 add <IDN,add>
15 ( <SE,24>
16 int <KW,1>
```

四、语法分析器 (Parser) - LR 分析法实现

4.1 设计目标

语法分析器的核心任务是依据 C--语言的上下文无关文法（定义于 `data/grammar.txt`），对词法分析器（`src/lexical/lexical.cpp`）输出的 Token 序列进行结构分析。它需要验证输入序列是否符合语法规则，报告语法错误，并在分析成功时构建程序的中间表示——抽象语法树（AST）。最终，按照实验要求，输出用于生成 AST 的产生式序列（对于 LR 分析，即规范归约过程中使用的产生式）。

具体目标包括：

- 1、接收词法分析器生成的 Token 序列作为输入。
- 2、使用从 C--文法（`data/grammar.txt`）生成的 LR 分析表（ACTION 表和 GOTO 表）。这些表可能是在程序运行时动态构建或从外部文件加载。
- 3、能够准确识别并报告语法错误。
- 4、输出分析过程中使用的产生式序列，记录每一次归约动作。
- 5、采用自下而上的表驱动 LR 分析方法。核心分析逻辑位

于 `src/syntax/analysis.cpp`，并依赖于 `src/syntax/LRTable.cpp`（用于处理分析表）和 `src/syntax/grammar.cpp`（用于处理文法产生式）。

4.2 实现方法与核心组件

本项目采用表驱动的 LR 分析方法。其主要组件和实现逻辑如下：

4.2.1 文法处理与产生式加载 产生式的定义:

```
// 文法产生式
struct Production {
    string left; // 产生式左部
    vector<string>
        right; // 产生式右部--- (在 grammar.txt 中多个选项则用空格分隔)
    bool operator==(const Production& r) const {
        bool ri = true;
        for (int i = 0; i < right.size(); i++)
            if (right[i] != r.right[i]) ri = false;
        return ((left == r.left) && ri);
    }
};
```

产生式的处理:

```
vector<Production> parseGrammar(const string &filename) {
    vector<Production> productions;
    ifstream file(filename);
    if (!file.is_open()) {
        cerr << "Error: Unable to open file: " << filename << endl;
        return productions;
    }
    string line;
    while (getline(file, line)) {
        line.erase(line.find_last_not_of(" \r\n\t") + 1); // 去除尾部空白字符
        if (line.empty()) continue;
        string arrow = "->";
        productions.push_back(splitProduction(line, arrow));
    }
    return productions;
}
```

4.2.2 FIRST 集与 FOLLOW 集的计算

虽然运行时 LR 分析器直接使用预构建的 ACTION/GOTO 表,但这些表的生成依赖于文法的 FIRST 集和 FOLLOW 集。

First 集的计算:

`src/syntax/SetCalculate.cpp` 中实现了计算每个非终结符 FIRST 集的算法。

计算 FIRST(X)的算法通常是迭代的:

1. 若 X 是终结符,则 $\text{FIRST}(X) = \{X\}$ 。
2. 若 X 是非终结符且有产生式 $X \rightarrow \epsilon$,则 $\epsilon \in \text{FIRST}(X)$ 。
3. 若 X 是非终结符且有产生式 $X \rightarrow Y_1 Y_2 \dots Y_k$:
 - a. 将 $\text{FIRST}(Y_1)$ 中所有非 ϵ 符号加入 $\text{FIRST}(X)$ 。
 - b. 若 $\epsilon \in \text{FIRST}(Y_1)$,则将 $\text{FIRST}(Y_2)$ 中所有非 ϵ 符号加入 $\text{FIRST}(X)$ 。
 - c. 以此类推,若 $\epsilon \in \text{FIRST}(Y_1) \dots \text{FIRST}(Y_{i-1})$,则将 $\text{FIRST}(Y_i)$ 中所有非 ϵ 符号加入 $\text{FIRST}(X)$ 。
 - d. 若 $\epsilon \in \text{FIRST}(Y_1) \dots \text{FIRST}(Y_k)$ (即所有 Y_i 都能推导出 ϵ),则 $\epsilon \in \text{FIRST}(X)$ 。

此过程反复迭代,直到所有非终结符的 FIRST 集不再增大。

```
void calculateFIRST(vector<Production> productions,
    unordered_set<string> &terminal_symbols,
    unordered_map<string, unordered_set<string>> &FIRST) {
    terminal_symbols.insert("$");
    for (const auto &production : productions) {
        FIRST[production.left] = unordered_set<string>(); // 初始化 FIRST 集为空
    }
    bool flag = true;
    while (flag) {
        flag = false;
        for (const auto &production : productions) {
            string left = production.left;
            vector<string> rights = production.right;
            int max_non = 0;
            if (rights.empty()) {
                if (FIRST[left].find("$") == FIRST[left].end()) {
                    FIRST[left].insert("$");
                    flag = true;
                }
                continue;
            }
            for (int index = 0; index < rights.size(); ++index) {
                string current_symbol = rights[index];
                if (terminal_symbols.find(current_symbol) !=
                    terminal_symbols.end()) {
                    if (FIRST[left].find(current_symbol) == FIRST[left].end()) {
```



```
}
```

2、ACTION 表和 GOTO 表的构建

构建 SLR 分析表：

1. 构造增广文法并计算 LR(0)项集族 $C = \{I_0, I_1, \dots, I_n\}$ 。
2. 对每个状态 I_i 和每个终结符 a ：
 - * 若项 $[A \rightarrow \alpha \bullet a \beta]$ 在 I_i 中且 $\text{Goto}(I_i, a) = I_j$ ，则 $\text{ACTION}[i, a] = \text{Shift } j$ 。
 - * 若项 $[A \rightarrow \alpha \bullet]$ 在 I_i 中且 A 不是增广文法的开始符号，则对所有 $b \in \text{FOLLOW}(A)$ ， $\text{ACTION}[i, b] = \text{Reduce } A \rightarrow \alpha$ 。
 - * 若项 $[S' \rightarrow S \bullet]$ 在 I_i 中 (S' 是增广文法开始符号)，则 $\text{ACTION}[i, \$] = \text{Accept}$ 。
3. 对每个状态 I_i 和每个非终结符 A ：若 $\text{Goto}(I_i, A) = I_j$ ，则 $\text{GOTO}[i, A] = j$ 。
4. 其余表项为 Error。

Goto 表的构建：

```
gotoTable fillGotoTable(gotoTable &table, const vector<State> &states,
                        nonTerminal nonterminals, Terminal terminals) {
    unordered_set<string> allsymbol;
    allsymbol.insert(nonterminals.begin(), nonterminals.end());
    allsymbol.insert(terminals.begin(), terminals.end());
    // 先把所有位置填上"-1"
    for (const State &state : states) {
        int state_number = state.stateNumber;
        for (const string &symbol : allsymbol) {
            table[state_number][symbol] = -1;
        }
    }
    for (const State &state : states) {
        int state_number = state.stateNumber;
        for (const auto goto_item : state.transitions) {
            // if(A is Nonterminal): k+A->j, 则加入{k, ("A", j)}
            string symbol = goto_item.first;
            if (nonterminals.find(symbol) != nonterminals.end())
                table[state_number][symbol] = goto_item.second;
        }
    }
    return table;
}
```

Action 表的构建：

```
actionTable fillActionTable(vector<Production> productions, actionTable &table,
                            vector<State> states,
                            unordered_map<string, unordered_set<string>> FOLLOW,
                            nonTerminal nonterminals, Terminal terminals) {
    unordered_set<string> allsymbol;
    allsymbol.insert(nonterminals.begin(), nonterminals.end());
    allsymbol.insert(terminals.begin(), terminals.end());
    // 先把所有位置填上"error"
    for (const State &state : states) {
        int state_number = state.stateNumber;
        for (const string &symbol : allsymbol) {
            table[state_number][symbol] = "error";
        }
    }
    for (const State &state : states) {
        int state_number = state.stateNumber;
        for (const Item &item : state.items) {
            int index = item.productionIndex;
            string left = productions[index].left;
            vector<string> right = productions[index].right;
            unordered_set<string> followset;
            followset = FOLLOW[left];
            // 移进项目
            if (item.dotPos < right.size()) {
                string symbol;
                symbol = right[item.dotPos];
                if (states[state_number].transitions[symbol] != -1) // 终结符
                {
                    if (terminals.find(symbol) != terminals.end()) {
                        int j = state.transitions.at(symbol);
                        table[state_number][symbol] = "s" + to_string(j);
                    }
                }
            }
            // 规约项目
            else if (item.dotPos == right.size() && item.productionIndex != 0) {
                for (const string symbol : terminals) {

```

```

        if (followset.find(symbol) != followset.end()) {
            table[state_number][symbol] = "r" + to_string(index);
        }
    }
} else if (item.productionIndex == 0 &&
            item.dotPos == right.size()) {
    table[state_number]["#"] = "acc";
}
}
}
return table;
}

```

4.2.4 主分析算法

核心组件声明：`include/syntax/analysis.h` 中定义了主分析类（例如`SyntaxAnalyzer`），包含分析栈、指向`LRTable`对象的指针/引用、指向`Grammar`对象的指针/引用等。

进行初始化后，进入 LR 分析主循环

1. 获取当前栈顶状态和当前输入 Token。
2. 使用当前状态和输入 Token 查询 ACTION 表得到分析动作。
3. 根据动作执行操作：

移进 (SHIFT): 将输入 Token（或其代表的符号）和 ACTION 表指定的新状态压入分析栈。消耗当前输入 Token，读取下一个。记录移进日志。

```

// 处理移进动作
if (action[0] == 's') {
    int state_num = stoi(action.substr(1));
    current_state_num = goto_state = state_num;
    symbol_stack.push(next_symbol);
    states_stack.push(goto_state);
    current_state = states[goto_state];
    next_symbol = rest_string[++index];
    output_file << "move" << endl;
    seq++;
}

```

归约：首先，根据 ACTION 表提供的产生式编号，从文法产生式列表中获取该产生式的完整定义（左部非终结符和右部符号序列）。

然后，根据产生式右部的长度，从符号栈和状态栈的顶部各弹出相应数量的元素。

弹出元素后，状态栈顶部会暴露一个新的状态。使用此状态和当前归约产生式的左部非终结符，在 GOTO 表中查询应转移到的下一个状态。

将查到的 GOTO 目标状态压入状态栈，并将产生式的左部非终结符名称压入符号栈。

此归约操作及所用产生式被记录到分析日志。

```

// 处理规约动作
else if (action[0] == 'r') {
    int production_num =
        stoi(action.substr(1)); // 规约使用的产生式序号
    Production production =
        productions[production_num]; // 规约使用的产生式
    int right_len = production.right.size(); // 产生式右部长度
    string left_symbol = production.left; // 产生式左部符号
    // 弹出栈中的符号和状态
    while (right_len-- > 0) {
        symbol_stack.pop();
        states_stack.pop();
    }
    int top_state_num = states[states_stack.top()].stateNumber;
    goto_state = gototable[top_state_num][left_symbol]; // GOTO 转移
    if (goto_state == -1) {
        output_file << "error" << endl;
        break;
    }
    states_stack.push(goto_state);
    symbol_stack.push(left_symbol);
}

```

```

        current_state_num = goto_state;
        current_state = states[current_state_num];
        output_file << "reduction" << endl;
        seq++;
    }
}

```

4、处理最终状态

接受 (ACCEPT): 分析成功, 记录接受日志, 获取最终的 AST 根节点, 终止分析。

错误 (ERROR): 调用错误报告机制, 记录错误信息, 终止分析。

```

// 处理最终状态
string final_action = actiontable[current_state_num][next_symbol];
if (final_action == "acc") {
    seq++;
    output_file << seq << " " << symbol_stack.top() << " # "
        << next_symbol << " accept" << endl;
} else if (final_action == "error") {
    seq++;
    output_file << seq << " " << symbol_stack.top() << " # "
        << next_symbol << " error" << endl;
}

```

4.3 中间组件

FIRST 集与 FOLLOW 集

在讨论 LR 分析表的生成与使用之前, 有必要提及 C--文法的两个重要属性集:

FIRST 集和 FOLLOW 集。 尽管在 LR 分析的运行时阶段, 分析器直接使用预先计算好的 ACTION 和 GOTO 表, 但在这些分析表的构建阶段, FIRST 集和 FOLLOW 集扮演着至关重要的角色。

FIRST 集: 对于一个文法符号 (终结符或非终结符) X, FIRST(X) 是指从 X 能够推导出的所有符号串的第一个终结符的集合。如果 X 能推导出空串 ϵ , 则 ϵ 也包含在 FIRST(X) 中。

作用于 LR 表构造: 在构造 LR 分析表时, FIRST 集用于计算项的“向前查看符号 (lookahead symbols)”, 这对于精确地决定何时进行归约以及使用哪个归约规则至关重要, 从而帮助解决冲突。

```

1  constExp: IntConst floatConst - + Ident ! (
2  lAndExpAtom: $ &&
3  relExp: IntConst floatConst - + Ident ! (
4  addExp: + - floatConst IntConst ! ( Ident
5  mulExpAtom: $ / % *
6  mulExp: IntConst floatConst - + Ident ! (
7  argFunctionR: $ ,
8  relExpAtom: $ >= <= < > <
9  unaryOp: ! - +
10 funcRParam: + - floatConst IntConst ! ( Ident
11 funcRParams: IntConst floatConst - + Ident ! ( $
12 eqExpAtom: $ != ==
13 initVal: + - floatConst IntConst ! ( Ident
14 number: floatConst IntConst
15 varDef: Ident
16 varDecl: int float
17 block: (
18 argVarDecl: $ ,
19 constInitVal: + - floatConst IntConst ! ( Ident
20 lOrExp: + - floatConst IntConst ! ( Ident
21 argConst: $ ,
22 lOrExpAtom: $ ||
23 blockItem: ! + $ float int const ; ( if Ident return ( IntConst floatConst -
24 decl: float int const
25 unaryExp: + - floatConst IntConst ! ( Ident
26 argFunctionF: $ ,
27 constDecl: const
28 constDef: Ident
29 compUnit: const float int void $
30 eqExp: + - floatConst IntConst ! ( Ident
31 funcFParam: int float
32 bType: float int
33 callFunc: (
34 funcType: void
35 program: const void int float $
36 lAndExp: IntConst floatConst - + Ident ! (
37 funcFParams: float int $
38 addExpAtom: $ - +
39 stmt: IntConst floatConst - + ! ( ; Ident return if {
40 matched_stmt: IntConst floatConst - + ! ( ; Ident return ( if
41 open_stmt: if
42 primaryExp: IntConst floatConst Ident (
43 funcDef: void int float
44 argExp: + - floatConst IntConst ! ( Ident $
45 exp: IntConst floatConst - + Ident ! (
46 cond: IntConst floatConst - + Ident ! (
47 lVal: Ident
48

```


FOLLOW 集： 对于一个非终结符 A，FOLLOW(A) 是指在文法的任何句型中，可能紧跟在 A 之后出现的终结符的集合。如果 A 可以出现在某个句型的最右端，则句尾符 \$(或本项目中对应的 EOFF) 也包含在 FOLLOW(A)中。

作用于 LR 表构造： 在构造分析表时，当面临一个可能进行归约的状态（即存在形如 $[X \rightarrow \alpha \bullet]$ 的项）时，FOLLOW 集 FOLLOW(X) 被用来确定在哪些输入符号下可以安全地执行该归约动作。这有助于解决 LR 自动机中的移进-归约冲突和归约-归约冲突。

```
1 constExp: ;
2 lAndExpAtom: [ ] ;
3 relExp: ; [ ] || && == !=
4 addExp: [ ] && != < > >= <= ;
5 mulExpAtom: [ ] && != > >= - == < <= ;
6 mulExp: [ ] && != > >= < <= ; <= , == < + -
7 argFunctionR:
8 relExpAtom: ; [ ] && != ==
9 unaryOp: Ident ( ! IntConst floatConst - +
10 funcRParam: ) ,
11 funcRParams:
12 eqExpAtom: ; [ ] || &&
13 initVal: ;
14 number: [ ] && == / % + ; * < , <= - > >= ) !=
15 varDef: ;
16 varDecl: return ( if { # Ident int float void const ; } - floatConst IntConst ! +
17 block: else floatConst IntConst return ( Ident int float void const ; ) - ! + # { if
18 argVarDecl:
19 constInitVal: ;
20 lOrExp: ;
21 argConst: ;
22 lOrExpAtom: ;
23 blockItem:
24 decl: ) - floatConst IntConst ; float int const void Ident ! + # { if ( return
25 unaryExp: [ ] && == / % + ; * < , <= - > >= ) !=
26 argFunctionF:
27 constDecl: return ( if { # Ident int float void const ; } - floatConst IntConst ! +
28 constDef: ;
29 compUnit: #
30 eqExp: ; [ ] &&
31 funcFParam: ) ,
32 bType: Ident
33 callFunc: [ ] && != % + == / - + < , <= ; > >=
34 funcType: Ident
35 program: #
36 lAndExp: ; [ ] ||
37 funcFParams:
38 addExpAtom: [ ] && != > >= < <= ;
39 stmt: ) - ! + float int const ; { if Ident ( return IntConst floatConst
40 matched_stmt: else floatConst IntConst ) - ! + float int const ; { if Ident ( return
41 open_stmt: floatConst IntConst ) - ! + float int const ; { if Ident ( return
42 primaryExp: [ ] && != % + == / - + < , <= ; > >=
43 funcDef: # void int float const
44 argExp: ;
45 exp: ;
46 cond: ;
47 lVal: [ ] && != != / % + ; * < , <= - > >=
```

ACTION 表与 GOTO 表

ACTION 表 (data/action_table.csv):

格式：CSV 文件。第一行为表头，首列为"State", 后续列为所有终结符的字符串名称（如"KW_INT", "IDN", "OP_PLUS", "EOFF"等）。后续每行代表一个状态，第一列为状态编号，其余单元格为动作字符串（"sX", "rX", "acc", 或"error"表示错误）。

```
1 State,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,50,51,52,53,54,55,56,57,58,59,60,61,62,63,64,65,66,67,68,69,70,71,72,73,74,75,76,77,78,79,80,81,82,83,84,85,86,87,88,89,90,91,92,93,94,95,96,97,98,99,100,101,102,103,104,105,106,107,108,109,110,111,112,113,114,115,116,117,118,119,120,121,122,123,124,125,126,127,128,129,130,131,132,133,134,135,136,137,138,139,140,141,142,143,144,145,146,147,148,149,150,151,152,153,154,155,156,157,158,159,160,161,162,163,164,165,166,167,168,169,170,171,172,173,174,175,176,177,178,179,180,181,182,183,184,185,186,187,188,189,190,191,192,193,194,195,196,197,198,199,200,201,202,203,204,205,206,207,208,209,210,211,212,213,214,215,216,217,218,219,220,221,222,223,224,225,226,227,228,229,230,231,232,233,234,235,236,237,238,239,240,241,242,243,244,245,246,247,248,249,250,251,252,253,254,255,256,257,258,259,260,261,262,263,264,265,266,267,268,269,270,271,272,273,274,275,276,277,278,279,280,281,282,283,284,285,286,287,288,289,290,291,292,293,294,295,296,297,298,299,300,301,302,303,304,305,306,307,308,309,310,311,312,313,314,315,316,317,318,319,320,321,322,323,324,325,326,327,328,329,330,331,332,333,334,335,336,337,338,339,340,341,342,343,344,345,346,347,348,349,350,351,352,353,354,355,356,357,358,359,360,361,362,363,364,365,366,367,368,369,370,371,372,373,374,375,376,377,378,379,380,381,382,383,384,385,386,387,388,389,390,391,392,393,394,395,396,397,398,399,400,401,402,403,404,405,406,407,408,409,410,411,412,413,414,415,416,417,418,419,420,421,422,423,424,425,426,427,428,429,430,431,432,433,434,435,436,437,438,439,440,441,442,443,444,445,446,447,448,449,450,451,452,453,454,455,456,457,458,459,460,461,462,463,464,465,466,467,468,469,470,471,472,473,474,475,476,477,478,479,480,481,482,483,484,485,486,487,488,489,490,491,492,493,494,495,496,497,498,499,500,501,502,503,504,505,506,507,508,509,510,511,512,513,514,515,516,517,518,519,520,521,522,523,524,525,526,527,528,529,530,531,532,533,534,535,536,537,538,539,540,541,542,543,544,545,546,547,548,549,550,551,552,553,554,555,556,557,558,559,560,561,562,563,564,565,566,567,568,569,570,571,572,573,574,575,576,577,578,579,580,581,582,583,584,585,586,587,588,589,590,591,592,593,594,595,596,597,598,599,600,601,602,603,604,605,606,607,608,609,610,611,612,613,614,615,616,617,618,619,620,621,622,623,624,625,626,627,628,629,630,631,632,633,634,635,636,637,638,639,640,641,642,643,644,645,646,647,648,649,650,651,652,653,654,655,656,657,658,659,660,661,662,663,664,665,666,667,668,669,670,671,672,673,674,675,676,677,678,679,680,681,682,683,684,685,686,687,688,689,690,691,692,693,694,695,696,697,698,699,700,701,702,703,704,705,706,707,708,709,710,711,712,713,714,715,716,717,718,719,720,721,722,723,724,725,726,727,728,729,730,731,732,733,734,735,736,737,738,739,740,741,742,743,744,745,746,747,748,749,750,751,752,753,754,755,756,757,758,759,760,761,762,763,764,765,766,767,768,769,770,771,772,773,774,775,776,777,778,779,780,781,782,783,784,785,786,787,788,789,790,791,792,793,794,795,796,797,798,799,800,801,802,803,804,805,806,807,808,809,810,811,812,813,814,815,816,817,818,819,820,821,822,823,824,825,826,827,828,829,830,831,832,833,834,835,836,837,838,839,840,841,842,843,844,845,846,847,848,849,850,851,852,853,854,855,856,857,858,859,860,861,862,863,864,865,866,867,868,869,870,871,872,873,874,875,876,877,878,879,880,881,882,883,884,885,886,887,888,889,890,891,892,893,894,895,896,897,898,899,900,901,902,903,904,905,906,907,908,909,910,911,912,913,914,915,916,917,918,919,920,921,922,923,924,925,926,927,928,929,930,931,932,933,934,935,936,937,938,939,940,941,942,943,944,945,946,947,948,949,950,951,952,953,954,955,956,957,958,959,960,961,962,963,964,965,966,967,968,969,970,971,972,973,974,975,976,977,978,979,980,981,982,983,984,985,986,987,988,989,990,991,992,993,994,995,996,997,998,999,1000,1001,1002,1003,1004,1005,1006,1007,1008,1009,1010,1011,1012,1013,1014,1015,1016,1017,1018,1019,1020,1021,1022,1023,1024,1025,1026,1027,1028,1029,1030,1031,1032,1033,1034,1035,1036,1037,1038,1039,1040,1041,1042,1043,1044,1045,1046,1047,1048,1049,1050,1051,1052,1053,1054,1055,1056,1057,1058,1059,1060,1061,1062,1063,1064,1065,1066,1067,1068,1069,1070,1071,1072,1073,1074,1075,1076,1077,1078,1079,1080,1081,1082,1083,1084,1085,1086,1087,1088,1089,1090,1091,1092,1093,1094,1095,1096,1097,1098,1099,1100,1101,1102,1103,1104,1105,1106,1107,1108,1109,1110,1111,1112,1113,1114,1115,1116,1117,1118,1119,1120,1121,1122,1123,1124,1125,1126,1127,1128,1129,1130,1131,1132,1133,1134,1135,1136,1137,1138,1139,1140,1141,1142,1143,1144,1145,1146,1147,1148,1149,1150,1151,1152,1153,1154,1155,1156,1157,1158,1159,1160,1161,1162,1163,1164,1165,1166,1167,1168,1169,1170,1171,1172,1173,1174,1175,1176,1177,1178,1179,1180,1181,1182,1183,1184,1185,1186,1187,1188,1189,1190,1191,1192,1193,1194,1195,1196,1197,1198,1199,1200,1201,1202,1203,1204,1205,1206,1207,1208,1209,1210,1211,1212,1213,1214,1215,1216,1217,1218,1219,1220,1221,1222,1223,1224,1225,1226,1227,1228,1229,1230,1231,1232,1233,1234,1235,1236,1237,1238,1239,1240,1241,1242,1243,1244,1245,1246,1247,1248,1249,1250,1251,1252,1253,1254,1255,1256,1257,1258,1259,1260,1261,1262,1263,1264,1265,1266,1267,1268,1269,1270,1271,1272,1273,1274,1275,1276,1277,1278,1279,1280,1281,1282,1283,1284,1285,1286,1287,1288,1289,1290,1291,1292,1293,1294,1295,1296,1297,1298,1299,1300,1301,1302,1303,1304,1305,1306,1307,1308,1309,1310,1311,1312,1313,1314,1315,1316,1317,1318,1319,1320,1321,1322,1323,1324,1325,1326,1327,1328,1329,1330,1331,1332,1333,1334,1335,1336,1337,1338,1339,1340,1341,1342,1343,1344,1345,1346,1347,1348,1349,1350,1351,1352,1353,1354,1355,1356,1357,1358,1359,1360,1361,1362,1363,1364,1365,1366,1367,1368,1369,1370,1371,1372,1373,1374,1375,1376,1377,1378,1379,1380,1381,1382,1383,1384,1385,1386,1387,1388,1389,1390,1391,1392,1393,1394,1395,1396,1397,1398,1399,1400,1401,1402,1403,1404,1405,1406,1407,1408,1409,1410,1411,1412,1413,1414,1415,1416,1417,1418,1419,1420,1421,1422,1423,1424,1425,1426,1427,1428,1429,1430,1431,1432,1433,1434,1435,1436,1437,1438,1439,1440,1441,1442,1443,1444,1445,1446,1447,1448,1449,1450,1451,1452,1453,1454,1455,1456,1457,1458,1459,1460,1461,1462,1463,1464,1465,1466,1467,1468,1469,1470,1471,1472,1473,1474,1475,1476,1477,1478,1479,1480,1481,1482,1483,1484,1485,1486,1487,1488,1489,1490,1491,1492,1493,1494,1495,1496,1497,1498,1499,1500,1501,1502,1503,1504,1505,1506,1507,1508,1509,1510,1511,1512,1513,1514,1515,1516,1517,1518,1519,1520,1521,1522,1523,1524,1525,1526,1527,1528,1529,1530,1531,1532,1533,1534,1535,1536,1537,1538,1539,1540,1541,1542,1543,1544,1545,1546,1547,1548,1549,1550,1551,1552,1553,1554,1555,1556,1557,1558,1559,1560,1561,1562,1563,1564,1565,1566,1567,1568,1569,1570,1571,1572,1573,1574,1575,1576,1577,1578,1579,1580,1581,1582,1583,1584,1585,1586,1587,1588,1589,1590,1591,1592,1593,1594,1595,1596,1597,1598,1599,1600,1601,1602,1603,1604,1605,1606,1607,1608,1609,1610,1611,1612,1613,1614,1615,1616,1617,1618,1619,1620,1621,1622,1623,1624,1625,1626,1627,1628,1629,1630,1631,1632,1633,1634,1635,1636,1637,1638,1639,1640,1641,1642,1643,1644,1645,1646,1647,1648,1649,1650,1651,1652,1653,1654,1655,1656,1657,1658,1659,1660,1661,1662,1663,1664,1665,1666,1667,1668,1669,1670,1671,1672,1673,1674,1675,1676,1677,1678,1679,1680,1681,1682,1683,1684,1685,1686,1687,1688,1689,1690,1691,1692,1693,1694,1695,1696,1697,1698,1699,1700,1701,1702,1703,1704,1705,1706,1707,1708,1709,1710,1711,1712,1713,1714,1715,1716,1717,1718,1719,1720,1721,1722,1723,1724,1725,1726,1727,1728,1729,1730,1731,1732,1733,1734,1735,1736,1737,1738,1739,1740,1741,1742,1743,1744,1745,1746,1747,1748,1749,1750,1751,1752,1753,1754,1755,1756,1757,1758,1759,1760,1761,1762,1763,1764,1765,1766,1767,1768,1769,1770,1771,1772,1773,1774,1775,1776,1777,1778,1779,1780,1781,1782,1783,1784,1785,1786,1787,1788,1789,1790,1791,1792,1793,1794,1795,1796,1797,1798,1799,1800,1801,1802,1803,1804,1805,1806,1807,1808,1809,1810,1811,1812,1813,1814,1815,1816,1817,1818,1819,1820,1821,1822,1823,1824,1825,1826,1827,1828,1829,1830,1831,1832,1833,1834,1835,1836,1837,1838,1839,1840,1841,1842,1843,1844,1845,1846,1847,1848,1849,1850,1851,1852,1853,1854,1855,1856,1857,1858,1859,1860,1861,1862,1863,1864,1865,1866,1867,1868,1869,1870,1871,1872,1873,1874,1875,1876,1877,1878,1879,1880,1881,1882,1883,1884,1885,1886,1887,1888,1889,1890,1891,1892,1893,1894,1895,1896,1897,1898,1899,1900,1901,1902,1903,1904,1905,1906,1907,1908,1909,1910,1911,1912,1913,1914,1915,1916,1917,1918,1919,1920,1921,1922,1923,1924,1925,1926,1927,1928,1929,1930,1931,1932,1933,1934,1935,1936,1937,1938,1939,1940,1941,1942,1943,1944,1945,1946,1947,1948,1949,1950,1951,1952,1953,1954,1955,1956,1957,1958,1959,1960,1961,1962,1963,1964,1965,1966,1967,1968,1969,1970,1971,1972,1973,1974,1975,1976,1977,1978,1979,1980,1981,1982,1983,1984,1985,1986,1987,1988,1989,1990,1991,1992,1993,1994,1995,1996,1997,1998,1999,2000,2001,2002,2003,2004,2005,2006,2007,2008,2009,2010,2011,2012,2013,2014,2015,2016,2017,2018,2019,2020,2021,2022,2023,2024,2025,2026,2027,2028,2029,2030,2031,2032,2033,2034,2035,2036,2037,2038,2039,2040,2041,2042,2043,2044,2045,2046,2047,2048,2049,2050,2051,2052,2053,2054,2055,2056,2057,2058,2059,2060,2061,2062,2063,2064,2065,2066,2067,2068,2069,2070,2071,2072,2073,2074,2075,2076,2077,2078,2079,2080,2081,2082,2083,2084,2085,2086,2087,2088,2089,2090,2091,2092,2093,2094,2095,2096,2097,2098,2099,2100,2101,2102,2103,2104,2105,2106,2107,2108,2109,2110,2111,2112,2113,2114,2115,2116,2117,2118,2119,2120,2121,2122,2123,2124,2125,2126,2127,2128,2129,2130,2131,2132,2133,2134,2135,2136,2137,2138,2139,2140,2141,2142,2143,2144,2145,2146,2147,2148,2149,2150,2151,2152,2153,2154,2155,2156,2157,2158,2159,2160,2161,2162,2163,2164,2165,2166,2167,2168,2169,2170,2171,2172,2173,2174,2175,2176,2177,2178,2179,2180,2181,2182,2183,2184,2185,2186,2187,2188,2189,2190,2191,2192,2193,2194,2195,2196,2197,2198,2199,2200,2201,2202,2203,2204,2205,2206,2207,2208,2209,2210,2211,2212,2213,2214,2215,2216,2217,2218,2219,2220,2221,2222,2223,2224,2225,2226,2227,2228,2229,2230,2231,2232,2233,2234,2235,2236,2237,2238,2239,2240,2241,2242,2243,2244,2245,2246,2247,2248,2249,2250,2251,2252,2253,2254,2255,2256,2257,2258,2259,2260,2261,2262,2263,2264,2265,2266,2267,2268,2269,2270,2271,2272,2273,2274,2275,2276,2277,2278,2279,2280,2281,2282,2283,2284,2285,2286,2287,2288,2289,2290,2291,2292,2293,2294,2295,2296,2297,2298,2299,2300,2301,2302,2303,2304,2305,2306,2307,2308,2309,2310,2311,2312,2313,2314,2315,2316,2317,2318,2319,2320,2321,2322,2323,2324,2325,2326,2327,2328,2329,2330,2331,2332,2333,2334,2335,2336,2337,2338,2339,2340,2341,2342,2343,2344,2345,2346,2347,2348,2349,2350,2351,2352,2353,2354,2355,2356,2357,2358,2359,2360,2361,2362,2363,2364,2365,2366,2367,2368,2369,2370,2371,2372,2373,2374,2375,2376,2377,2378,2379,2380,2381,2382,2383,2384,2385,2386,2387,2388,2389,2390,2391,2392,2393,2394,2395,2396,2397,2398,2399,2400,2401,2402,2403,2404,2405,2406,2407,2408,2409,2410,2411,2412,2413,2414,2415,2416,2417,2418,2419,2420,2421,2422,2423,2424,2425,2426,2427,2428,2429,2430,2431,2432,2433,2434,2435,2436,2437,2438,2439,2440,2441,2442,2443,2444,2445,2446,2447,2448,2449,245
```


GOTO 表 (data/goto_table.csv):

格式：CSV 文件。第一行为表头，首列为"State"，后续列为所有非终结符的字符串名称（如"CompUnit", "Decl", "Exp"等）。后续每行代表一个状态，第一列为状态编号，其余单元格为目标状态的整数编号。

[illegible]

4.4 语法错误处理

错误检测： 在核心分析逻辑中，当 ACTION 表查询结果为 "error"，或 GOTO 转移无效时，即检测到语法错误。

错误报告：当前实现主要是在输出到 `data/syntax.txt` 的日志中记录 "error" 动作。

4.5 输出格式说明 (分析过程日志)

LR 语法分析器在执行过程中，会进行一系列的移进和归约操作，最终目标是接受合法的输入串。本项目的 **ParserLR** 将其分析过程的详细步骤记录在 **data/syntax.txt** 文件中（位于 AST 的 DOT 描述之后）。该日志准确地反映了规范归约（最右推导的逆过程）的每一步决策。

所采用的输出格式如下:

[序号] [TAB] [栈表示符] # [面临输入符号] [TAB] [执行动作]

其中：

[序号]: 一个从 1 开始递增的整数, 表示分析步骤的顺序。

[TAB]: 制表符，用于格式化对齐。

[栈表示符]:

在 `move` (移进) 动作之前, 这部分可能表示移进前的状态或概念上的栈顶。在你提供的示例 `1##const move` 中, 第一个 `#` 可能代表栈底或一个空栈的初始状态。

在 reduction (归约) 动作之前, 这部分通常表示将被归约的句柄的最后一个符号, 或者有时是归约后形成的非终结符的名称 (取决于日志记录的具体实现)。例如, 3 int

Ident reduction, 这里的 int 可能是句柄的一部分或与当前状态关联的符号。

#[面临输入符号]: 字符 # 作为分隔符, 后面紧跟的是当前词法分析器提供的下一个输入 Token 的词素值 (例如 const, Ident, ;)。

[执行动作]:

move: 对应 LR 分析中的“移进 (Shift)”操作。表示当前的输入 Token 被接受并和新状态一起压入分析栈。

reduction: 对应 LR 分析中的“归约 (Reduce)”操作。表示分析栈顶部的若干符号 (形成一个产生式的右部, 即句柄) 被替换为该产生式的左部非终结符。

这个日志非常详细地记录了 LR 分析器的每一步决策。

```
2072 2072 stmt # } reduction
2073 2073 blockItem # } reduction
2074 2074 blockItem # } reduction
2075 2075 blockItem # } reduction
2076 2076 blockItem # } reduction
2077 2077 blockItem # } reduction
2078 2078 blockItem # } reduction
2079 2079 blockItem # } reduction
2080 2080 blockItem # } reduction
2081 2081 blockItem # } reduction
2082 2082 blockItem # } reduction
2083 2083 blockItem # } reduction
2084 2084 blockItem # } reduction
2085 2085 blockItem # } reduction
2086 2086 blockItem # } move
2087 2087 } # # reduction
2088 2088 block # # reduction
2089 2089 funcDef # # reduction
2090 2090 compUnit # # reduction
2091 2091 compUnit # # reduction
2092 2092 compUnit # # reduction
2093 2093 compUnit # # reduction
2094 2094 compUnit # # reduction
2095 2095 compUnit # # reduction
2096 2096 compUnit # # reduction
2097 2097 compUnit # # reduction
2098 2099 compUnit # # accept
2099
```

4.6 源程序编译步骤 (语法分析阶段)

输入预处理: 调用 read_from_lexical() 函数从词法分析结果文件转换得到内部符号串。

初始化分析环境: 在主程序 (main.cpp) 中准备空的分析栈、加载文法产生式、加载或构建 ACTION/GOTO 分析表。

执行分析: 调用核心的 analysis() 函数, 传入准备好的输入串、产生式和分析表。

输出/结果: 分析过程的详细步骤被写入 data/syntax.txt。主程序提示用户查看该文件。

五、语义分析与中间代码生成初步设计 (选做任务 - 未完成)

本项目的选做部分旨在将前端生成的抽象语法树 (AST) 转换为 LLVM IR (中间表示), 通过调用实验提供的中端编译器库接口实现。尽管此任务未能最终完成, 我们进行了初步的设计和探索。

5.1 核心思路：AST 遍历与语义驱动的 IR 生成

基本思路是遍历语法分析阶段生成的 AST。AST 的每个节点都蕴含了 C--源代码的特定语法和语义信息。在遍历过程中，根据每个节点的类型和内容，调用中端库提供的 API 来逐步构建 LLVM IR。

5.2 AST 遍历方法：访问者模式

为了有序地处理 AST 中不同类型的节点，我们计划采用访问者模式 (Visitor Pattern)。

定义一个 Visitor 接口，为 AST 中每种节点类型（如函数定义、变量声明、表达式、语句等）声明一个 visit 方法。

每个 AST 节点类实现一个 accept 方法，用于接受一个 Visitor 对象并调用其对应的 visit 方法。

创建一个具体的 IRGenerator 类继承自 Visitor，并实现所有 visit 方法。每个 visit 方法的职责就是将当前 AST 节点的语义翻译成一个或多个 LLVM IR 指令，或调用中端 API 创建 LLVM 结构（如函数、全局变量）。

遍历从 AST 的根节点开始，通过 accept 方法启动，并以深度优先的方式递归访问整个树。

5.3 调用中端 API 的设计思想概要

针对不同类型的 AST 节点，调用中端 API 的初步设计思想如下：

声明（全局/局部变量、函数）：

对于变量声明，根据其作用域（全局或函数局部），使用中端 API 创建 LLVM 全局变量或在函数入口块分配局部变量（alloca）。

对于函数声明，使用中端 API 创建 LLVM Function 对象，并处理其参数和返回类型。

所有声明的实体（变量名、函数名）及其对应的 LLVM 表示（Value*）将被记录到符号表中。

表达式：

visit 方法的目标是为表达式生成计算其值的 LLVM IR，并得到一个代表该值的 LLVM Value*。

常量直接映射为 LLVM 常量。变量引用通过从符号表查找并加载其值。二元/一元运算通过调用 IRBuilder 提供的指令创建方法（如加法、比较等）生成。函数调用则生成 LLVM call 指令。

语句：

赋值语句：计算右侧表达式的值，然后使用 store 指令将其存入左侧变量的内存位置。

条件语句 (if)：生成条件判断 IR，并创建相应的基本块（then, else, merge），使用条件跳转和无条件跳转指令构建正确的控制流。

返回语句：生成 ret 指令，若有返回值则带上表达式计算结果的 Value*。

符号表：一个关键的辅助结构，用于在遍历过程中管理标识符（变量、函数名）及其在 LLVM IR 中的对应表示，并处理作用域。

六、总结

本次 C--编译器大作业是对编译原理核心知识的一次深度实践。通过从零开始构建词法分析器、LR 语法分析器并生成抽象语法树，我们不仅将理论付诸实践，更在解决具体问题的过程中深化了对编译器工作机制的理解。

核心工作回顾：

我们成功实现了 C--语言的词法分析器，它能够准确识别各类 Token 并处理注释。随后，基于预先生成的 ACTION 和 GOTO 表，构建了一个表驱动的 LR 语法分析器。详细的 LR 分析步骤也被记录下来，为调试和理解提供了便利。虽然选做的中间代码生成部分未能最终完成，但相关的设计（如采用访问者模式遍历 AST）也进行了初步探索。

遇到的问题：

在整个开发过程中，我们遇到了诸多挑战，克服它们的过程也是学习和成长的过程。

词法分析阶段，最初看似直接，但在处理如浮点数的精确定义（例如，3.是非法的，小数点后必须有数字）和区分=与=这类多字符与单字符运算符时，确实费了一番功夫。简单地贪婪匹配会导致错误，必须通过更细致的状态机设计和向前查看（lookahead）字符来解决。这让我们真正理解了 DFA 状态转移的精妙之处，以及看似简单的 Token 识别背后所需的严谨逻辑。

LR 语法分析无疑是本次实验的重头戏和主要难点。将 CSV 格式的 ACTION/GOTO 表加载到程序中，并正确解析其内容（比如区分"s5"代表移进到状态 5，"r12"代表按规则 12 归约），就需要编写不少辅助代码。核心的 LR 分析循环，如何维护状态栈，如何在归约时准确地根据产生式右部长度弹出状态，并利用 GOTO 表找到新的状态压栈，这些都是理论上知道但实际编码时容易出错的地方。我们曾一度因为栈操作的错误或者 GOTO 表查询的失误，导致分析器陷入死循环或提前错误退出。通过大量的单步调试，打印栈内容和当前输入，才逐渐摸清了 LR 分析器“移进-归约”舞蹈的每一个舞步。

此外，在整个编译器链条中，各种标识符（Token 类型、文法符号名、分析表中的数字 ID）的统一和转换也是一个容易忽视但至关重要的问题。例如，词法分析器输出的 TokenType 枚举，需要转换成 ACTION 表查询时使用的字符串或整数 ID；产生式中定义的非终结符名称，也需要映射到 GOTO 表查询时使用的内部表示。这些映射关系如果处理不当，整个分析流程就会在不经意间出错。

项目有关文件代码均位于 <https://github.com/luziyi/TinyCompiler/tree/main>
成员分工：

武承霖	3022244338	实现 dfa 最小化，token 的定义与构建
王嘉仑	3022244358	nfa 与 dfa 的实现，主扫描函数的实现
王鑫培	3022244362	实现 LR 项集构造与 LR 自动机，ACTION 表与 GOTO 表构建及加载
陆子毅	3022206045	文法处理与 FIRST/FOLLOW 集，实现 LR 项集构造与 LR 自动机
郑力墉	3022244344	LR 语法分析主驱动与日志输出，ACTION 表与 GOTO 表构建及加载，撰写报告