编译原理中间代码生成器实验报告

小组成员：郑星云 张子菡 陈凌锐

完成时间：2023/12/10

1. 需求分析

1.1 作业要求

1. 在前面实验的基础上（词法、语法分析），进行语义分析和中间代码生成器的设计，输入源程序，输出等价的中间代码序列（以四元式的形式作为中间代码）
2. 实现静态语义错误的诊断和处理
3. 在此基础上，考虑更为通行的高级语言的语义检查和中间代码生成所需要注意的内容，并给出解决方案。

1.2 使用方式

1.2.1 bat运行

直接运行 `build.bat` (Windows) `bash build.sh` (Linux/MacOS)，包含了编译和运行的过程。

1.2.2 可执行文件运行

在提交的源代码文件的 `output` 文件夹中，已经包含windows环境可执行文件 `QuartetTranslator.exe`.

运行方法：

```
QuartetTranslator.exe <输入文件路径> <文法文件路径>
```

例如：

```
QuartetTranslator.exe ./test/input/input.txt ./test/grammer/grammer.txt
```

1.2.3 重新编译生成可执行文件

如需重新从源代码编译生成可执行文件可使用如下命令：

- Windows：

```
g++ -std=c++17 -O2 .\src\main.cpp .\src\Lexer.cpp .\src\LR1Parser.cpp -o
.\output\LR1Parser.exe
```

- Linux/MacOS：

```
g++ -std=c++17 -O2 ./src/main.cpp ./src/Lexer.cpp ./scr/LR1Parser.cpp -o
./output/LR1Parser
```

(注：g++需要支持c++17)

1.3 输入格式

1.3.1 文法文件

文法文件仿照**扩展巴克斯范式(EBNF)**声明，格式具体要求如下：

1. 第一行需包含两个字符，分别代表起始符和终结符。
2. 第二行需列出所有的终结符。
3. 后续任意行，每行需给出一个产生式。产生式的右边用 `|` 分隔每个子句，`::=` 用于分割产生式的左边和右边，每个子句中的终结符或非终结符之间用空格 隔开。
4. 提供的文法需满足LR(1)文法规范，不含左递归、回溯，且不包含 ε。

以下是一个简单的文法例子：

$$
\begin{aligned}
\mathrm{S'} &\rightarrow \mathrm{S} \\
\mathrm{S} &\rightarrow \mathrm{BB} \\
\mathrm{B} &\rightarrow \mathrm{aB} \mid \mathrm{b}
\end{aligned}
$$

则其在在 `grammer.txt` 文法文件中应为：

```
S' $
a b
S' ::= S
S  ::= B B
B  ::= a B | b
```

1.3.2 代码文件

1.3.2.1 支持的符号

中间代码生成器支持的符号包括：

1. **操作符（Operators）**：
   - 算术运算符：`+`, `-`, `*`, `/`, `%`（求余）
   - 赋值运算符：`=`
   - 比较运算符：`==`, `!=`, `<`, `<=`, `>`, `>=`
   - 逻辑运算符：`&&`（逻辑与），`||`（逻辑或），`!`（逻辑非）
   - 位运算符：`&`, `|`, `^`, `~`, `<<`, `>>`
   - 其他运算符：`++`, `--`（自增和自减）
2. **关键字（Keywords）**：
   - 控制流程语句：`if`, `else`, `while`, `for`, `return`
   - 数据类型：`int`, `float`, `char`, `void`
   - 结构体声明：`struct`

- 这个映射表可以根据需要添加更多关键字，如 `switch`, `case` 等。

3. **界定符（Delimiters）**：
   - 分号、大括号、小括号、方括号等标准程序结构界定符：`;`, `{`, `}`, `(`, `)`, `[`, `]`
   - 其他符号：`,`, `.`, `->`, `:`, `?`
4. **字面量和标识符**:
   - 字符字面量：通过 `getCharLiteral()` 函数处理。
   - 字符串字面量：通过 `getStringLiteral()` 函数处理。
   - 数字（包括整数和浮点数）：通过 `getNumber()` 函数处理。
   - 标识符（如变量和函数名）：通过 `getIdentifierOrKeyword()` 函数处理，这个函数同时也用于识别关键字。
5. **注释**:
   - 代码中似乎有一个 `skipComment()` 函数，这可能用于跳过注释，但具体实现不在提供的代码段中。
6. **空白字符**:
   - 通过 `skipWhitespace()` 函数处理，这通常用于忽略代码中的空格、制表符和换行符等空白字符。

1.3.2.2 支持的语句种类

- 函数定义语句，例如

```
int solve(int a, int b) {}
```

- 变量定义语句，例如

```
int x = 1;//整形
float f = 0.2;//浮点型
char a[10];//数组
char b = 'b';//字符
int ci = 10;//定义时赋初值
```

- 变量赋值语句，例如

```
char b = 'b';//定义时赋初值
str = "Hello World";//定义后常量赋值
x = x % 5;//定义后表达式赋值
int result = solve(x, y);//定义后返回值赋值
```

- if分支语句,例如

```
//单分支
if (x > y) {
    x = y;
}
//双分支
if (x > y) {
    x = y;
}else {
    x = 0;
```

```
}
//多分支
if (x > y) {
    x = y;
} else if (x < y) {
    x = y + 1;
} else {
    x = 0;
}
```

- for循环语句，例如

```
for (i = 0; i < 10; i++) {
    x = x + i;
}

for (i = 0; i < 10; i++) {
    a[i] = i;
}
```

- while循环语句，例如

```
while (x > 0) {
    x--;
}
```

- 变量自增自减语句，例如

```
x++;
++x;
x--;
--x;
```

- 运算赋值语句，例如

```
x = x + i;//加运算
x = x - i;//减运算
x = !x;//取非运算
x = x % 5;//取模运算
x = x & 1;//且运算
x = x | 1;//或运算
x = x ^ 1;//异或运算
x = ~x;//取反运算
```

- 函数调用语句

```
int result = solve(x, y);
```

- 函数返回语句

```
return 0;//返回常量
return a + b;//返回表达式值
```

1.4 输出形式

由于输出内容过长，输出被重定向到日志文件 `./output/test.log`。程序的输出内容主要有以下几个部分：

1. 语法分析结果
2. 变量表
3. 中间代码序列（四元式形式）

以提交文件中已存在的文法和输入文件为例，输出如下：

语法分析结果：

```
Accept
```

变量表：

Variable Name	Type	Initial Value
c	char	'a'
d	int	5
i	int	NULL
sum	int	NULL

中间代码序列（四元式形式）

ID	Quarter
0	(= , 0 , _ , i)
1	(< , i , 50 , t0)
2	(jnz , t0 , _ , 4)
3	(j , _ , _ , 12)
4	(% , i , 2 , t1)
5	(== , t1 , 0 , t2)
6	(jnz , t2 , _ , 8)
7	(j , _ , _ , 10)
8	(+ , sum , i , t3)
9	(= , t3 , _ , sum)
10	(+ , i , 1 , i)
11	(j , _ , _ , 1)

2.概要设计

2.1 任务分解

整体上分为词法分析器Lexer、语法分析器LR1Parser、中间代码生成器，对应代码中不同的模块，具体如下：

- **Lexer：**

  实现词法分析器，将源代码分解成一个个的 Token，即可供词法分析器识别的终结符。

  输入：一个字符串，表示待分析的真实代码

  输出：一个Token数组，表示解析后的终结符语句

- **LR1Parser：**

  实现 LR(1) 文法分析器，进行语法分析。

  - 从文件中以EBNF格式读取文法

    输入：文本文件（保证格式正确）

    输出：一个产生式数组，表示该文法的所有产生式

  - 构造ACTION表和GOTO表

    输入：一个产生式数组

    输出：ACTION表和GOTO表

    - 构造DFA
      - 构造FIRST集

        - 求项目集族闭包
    - 分析语句是否符合文法

      输入：ACTION表和GOTO表、待分析语句

      输出：分析结果

- **SemanticAnalyzer：**

  实现中间代码生成器，进行语义分析和中间代码生成

  输入：语法分析树根节点

  输出：变量表，存储声明的变量

  中间代码，一个数字编号对应一个四元式

2.2 数据类型的定义

2.2.1 词法分析器

- **Token**

`Token` 类，用于表示编程语言中的词法单元。

一个 Token 对象由两部分组成：`type` 表示 Token 的类型（使用 `TokenType` 中的值），`value` 表示 Token 的具体值（如标识符的名字、整数字面量的值等）。

```
struct Token
{
    TokenType type;
    std::string value;
};
```

定义 `TokenType` 来规定Token的类型有哪些，包括字面量、关键字、操作符和界定符等。

```
enum TokenType {
    // 字面量 (Literals)
    T_IDENTIFIER,       // 例如: variableName, 任意其他变量名、函数名
    T_INTEGER_LITERAL,  // 例如: 42
    T_FLOAT_LITERAL,    // 例如: 3.14
    T_STRING_LITERAL,   // 例如: "hello"
    T_CHAR_LITERAL,     // 例如: 'a'

    // 关键字 (Keywords)
    T_IF,       // if
    T_ELSE,     // else
    T_WHILE,    // while
    T_FOR,      // for
    T_RETURN,   // return
    T_INT,      // int
    T_FLOAT,    // float
    T_CHAR,     // char
    T_VOID,     // void
    T_STRUCT,   // struct
    // ... 可以继续添加其他关键字，如: switch, case, etc.
```

```
    // 操作符 (Operators)
    T_PLUS,          // +
    T_MINUS,         // -
    T_MULTIPLY,      // *
    T_DIVIDE,        // /
    T_ASSIGN,        // =
    T_EQUAL,         // ==
    T_NOTEQUAL,      // !=
    T_LESS,          // <
    T_LESSEQUAL,     // <=
    T_GREATER,       // >
    T_GREATEREQUAL,  // >=
    T_AND,           // &&
    T_OR,            // ||
    T_NOT,           // !
    T_MOD,           // %
    T_INCREMENT,     // ++
    T_DECREMENT,     // --
    T_BITAND,        // &
    T_BITOR,         // |
    T_BITXOR,        // ^
    T_BITNOT,        // ~
    T_LEFTSHIFT,     // <<
    T_RIGHTSHIFT,    // >>
    // ... 其他操作符

    // 界符 (Delimiters/Punctuators)
    T_SEMICOLON,     // ;
    T_LEFT_BRACE,    // {
    T_RIGHT_BRACE,   // }
    T_LEFT_PAREN,    // (
    T_RIGHT_PAREN,   // )
    T_LEFT_SQUARE,   // [
    T_RIGHT_SQUARE,  // ]
    T_COMMA,         // ,
    T_DOT,           // .
    T_ARROW,         // ->
    T_COLON,         // :
    T_QUESTION,      // ?
    // ... 其他界符

    T_UNKNOWN,  // 未知Token
    T_EOF,      // 文件结束
};
```

2.2.2 语法分析器

- **符号类（Symbol）**

`SymbolType` 枚举定义了语法分析中符号的类型，包括终结符、非终结符和空串。

`Symbol` 结构体表示文法中的符号，包括符号类型 `type` （属于终结符、非终结符），和符号值 `value`。

例如，非终结符 S 被定义为 `Symbol(NonTerminal, "S")` ，终结符 var 被定义为 `Symbol(Terminal, "var")`

特别的，我们规定 ε 为 `Symbol(Epsilon, "")`

```
enum class SymbolType {
  Terminal,     // 终结符
  NonTerminal,  // 非终结符
  Epsilon       // 空串
};

struct Symbol
{
    SymbolType type;
    std::string value;
};
```

- **产生式类（Production）**

`Production` 结构体表示文法中的产生式，由两部分组成，分别是产生式左边的非终结符和产生式右边的语句。语句是一个符号数组 `std::vector<Symbole>` ，既可以同时包含非终结符和终结符。

```
struct Production
{
    Symbol lhs;                // 左边，非终结符
    std::vector<Symbol> rhs;   // 右边，非终结符和终结符的组合
};
```

- **LR1项类（LR1Item）**

定义一个名为 `LR1Item` 的类，表示 LR(1) 语法分析中的 LR(1) 项目。

LR1项目有 4 中状态，分别是移进（Shift），归约（Reduce），待约（Goto）和接受（Accept）。

`Production production` 表示 LR(1) 项目的产生式。

`size_t dot_position` 表示 LR(1) 项目中的点（`·`）的位置，指示其在产生式右侧的哪个符号前面。

`Symbol lookahead` 表示 LR(1) 项目中的展望符，即在特定条件下进行移进或归约的符号。

例如，对于项目 $[\mathrm{B} \rightarrow \mathrm{a}.\mathrm{B}, \mathrm{a}]$ 记作 `LR1Item(prod, 1, Symbol(Terminal, a))`. 其中 `prod` 表示产生式 $[\mathrm{B} \rightarrow \mathrm{aB}]$

```
class LR1Item {
public:
    enum class State {
        SHIFT,    // 移进
        REDUCE,   // 归约
        GOTO,     // 待约
        ACCEPT    // 接受
    };

    Production production;
    size_t dot_position;
    Symbol lookahead;
};
```

- **Action类**

定义了一个名为 `Action` 的结构体，用于表示 LR(1) 分析表中的动作。

`Type type`: 表示动作的类型，即移进、归约、接受或错误。

`size_t number` 表示当动作类型为移进时，移进到的新状态的编号；其他情况下为无效值。

`Production production` 表示当动作类型为归约时，使用的产生式；其他情况下为无效值。

```
struct Action
{
    enum class Type {
        SHIFT,
        REDUCE,
        ACCEPT,
        ERROR
    : };
    Type type;
    size_t number;          // shift的新状态编号
    Production production;  // reduce使用的产生式
};
```

2.2.3 中间代码生成器

Quater类

定义了一个名为`Quater`的类，它表示一个四元式。

`Quater` 类中定义了四个成员变量：`op`、`arg1`、`arg2` 和 `result`。

`op` 表示操作符，`arg1` 和 `arg2` 表示前两个操作数，`result` 表示结果。

```
class Quater {
public:
    std::string op;
    std::string arg1;
    std::string arg2;
    std::string result;
};
```

VaribleMeta类

定义了一个名为`VaribleMeta`的结构体，它用于存储变量的元数据。

`VaribleMeta`结构体中定义了两个成员变量：`type`和`value`。

`type`是一个字符串，用于存储变量的类型。类型可以是`int`、`float`、`char`等。

`value`也是一个字符串，用于存储变量的值。为了简化操作，无论变量的实际类型是什么，都将其值存储为字符串。在需要使用变量的值时，再根据变量的类型将其转换为相应的类型

```
struct VaribleMeta
{
    std::string type;   // 变量类型，int，float，char等
    std::string value;  // 变量的值，为了简化操作，全部存为string，在需要使用时进行转换
};
```

SemanticAnalyzer类

定义了一个名为`SemanticAnalyzer`的类，它用于进行语义分析和中间代码生成。

`SemanticAnalyzer`类中定义了一些私有的成员变量：

- `root`是一个指向`SemanticTreeNode`的指针，它指向语法分析树的根节点。
- `varible_table`是一个映射，它存储了已经声明过的变量。映射的键是变量的名字，值是一个`VaribleMeta`结构体，包含了变量的类型和值。
- `next_temp_varible_id`是一个整数，用于生成新的临时变量的名字。

`SemanticAnalyzer`类中还定义了一些私有的成员函数，这些函数用于处理不同的语法结构

- `handle_defalt(SemanticTreeNode*& node)`处理默认的语法结构，通常是在没有特定处理函数的情况下调用。
- `handle_var_declaration(SemanticTreeNode*& node)`处理变量声明语法结构。
- `handle_opt_init(SemanticTreeNode*& node)`处理可选的初始化语法结构，通常用于变量声明时的初始化。
- `handle_expression(SemanticTreeNode*& node)`处理表达式语法结构。
- `handle_simple_expression(SemanticTreeNode*& node)`处理简单表达式语法结构，通常是没有运算符的表达式。
- `handle_additive_expression(SemanticTreeNode*& node)`处理加法表达式语法结构，包括加法和减法。

- `handle_term(SemanticTreeNode*& node)`处理项语法结构，通常是乘法和除法表达式。
- `handle_postfix_expression(SemanticTreeNode*& node)`处理后缀表达式语法结构，例如函数调用。
- `handle_var(SemanticTreeNode*& node)`处理变量语法结构。
- `handle_factor(SemanticTreeNode*& node)`处理因子语法结构，因子是表达式的基本组成部分。
- `handle_prefix_expression(SemanticTreeNode*& node)`处理前缀表达式语法结构，例如负数和逻辑非表达式。
- `handle_selection_stmt(SemanticTreeNode*& node)`处理选择语句语法结构，例如if语句。
- `handle_iteration_stmt(SemanticTreeNode*& node)`处理迭代语句语法结构，例如for和while循环。
- `handle_opt_expression_stmt(SemanticTreeNode*& node)`处理可选的表达式语句语法结构，通常用于处理可以省略的表达式语句。

`SemanticAnalyzer`类还定义了一些公有的成员函数

- `semantic_analyze`用于进行语义分析
- `print_intermediate_code`用于打印中间代码
- `print_variable_table`用于打印变量表。

```
class SemanticAnalyzer {
public:
    void semantic_analyze();
    void print_intermediate_code();
    void print_variable_table();

private:
    void handle_defalt(SemanticTreeNode*& node);
    void handle_var_declaration(SemanticTreeNode*& node);
    void handle_opt_init(SemanticTreeNode*& node);
    void handle_expression(SemanticTreeNode*& node);
    void handle_simple_expression(SemanticTreeNode*& node);
    void handle_additive_expression(SemanticTreeNode*& node);
    void handle_term(SemanticTreeNode*& node);
    void handle_postfix_expression(SemanticTreeNode*& node);
    void handle_var(SemanticTreeNode*& node);
    void handle_factor(SemanticTreeNode*& node);
    void handle_prefix_expression(SemanticTreeNode*& node);
    void handle_selection_stmt(SemanticTreeNode*& node);
    void handle_iteration_stmt(SemanticTreeNode*& node);
    void handle_opt_expression_stmt(SemanticTreeNode*& node);


private:
    SemanticTreeNode* root;
    std::map<std::string, VaribleMeta> varible_table;
    size_t next_temp_varible_id;
};
```

2.3 运行流程

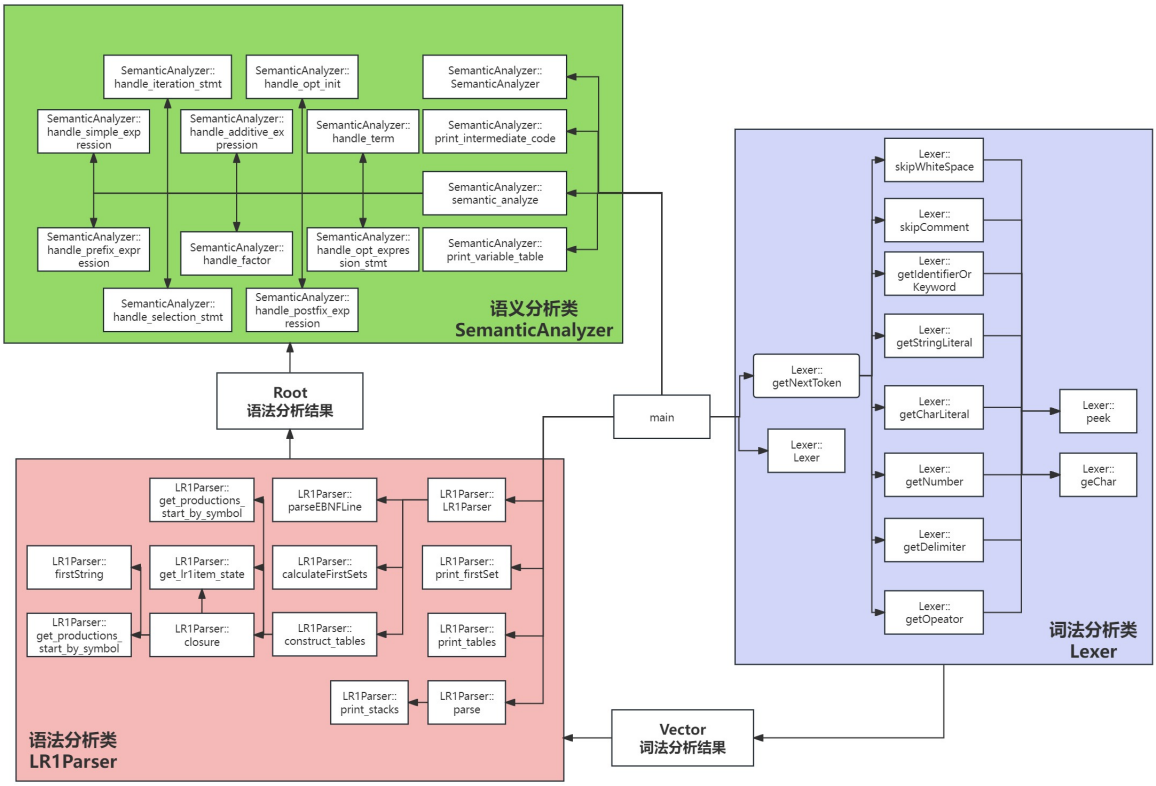
主程序运行流程如下：

1. **命令行参数检查：**
   - 检查命令行参数数量，如果小于 3，输出用法信息，表示需要提供输入文件和文法文件的路径。
   - 如果命令行参数不符合要求，程序返回 1，表示出现错误。
2. **日志文件创建：**
   - 调用 `createLogFileIfNotExists` 函数创建日志文件（如果不存在）。
3. **重定向标准输出到日志文件：**
   - 使用 `freopen` 将标准输出重定向到日志文件。
4. **读取输入文件内容：**
   - 打开指定的输入文件，读取文件内容到字符串流中。
5. **词法分析（Lexical Analysis）：**
   - 使用 `Lexer` 类对文件内容进行词法分析，生成 Token 序列。
   - 逐个获取 Token，并将每个 Token 转换为 Symbol 对象，并将其添加到句子向量中，直到遇到 T_EOF（表示文件结束）为止。
6. **语法分析（Syntactic Analysis）：**
   - 调用 `parse` 函数，进行语法分析，传入之前词法分析得到的 Token 序列。
   - 输出语法分析的详细过程。
7. **中间代码生成（SemanticAnalyzer）：**
   - 代码创建一个 SemanticAnalyzer 对象，并将语法分析树作为参数传递给它。。
   - 调用 `analyzer.semantic_analyze` 方法进行语义分析。这个方法将遍历语法分析树，进行语义分析。
   - 调用 `analyzer.print_variable_table` 和 `analyzer.print_intermediate_code` 方法打印变量表和中间代码。

2.4 模块间的调用关系

1. 主程序调用Lexer类（词法分析器）和LR1parser类（LR1语法分析器）。
2. Lexer类词法分析器分析输入文件源码后，输出一系列Token组成语句。
3. LR1Parser类对Lexer类生成的语句进行分析，输出结果。

函数调用图如下：



3.详细设计

3.1 词法分析器设计

3.1.1 总体设计

词法分析器仿照 Lex 词法分析的结构。整体上看，词法分析的过程是一个有限自动机，根据当前符号串的输入判断是否继续读入符号、是否获取到一个可解析的单词。

词法分析Lexer类的结构如下：

```
class Lexer {
private:
    std::string input;
    size_t index = 0;

    char peek();
    char getChar();
    void skipWhitespace();
    Token getIdentifierOrKeyword();
    Token getCharLiteral();
    Token getStringLiteral();
    Token getNumber();
    Token getOperator();
    Token getDelimiter();
    void skipComment();
```

```
    std::map<std::string, TokenType> operatorMap = {
        {"+", T_PLUS},
        // ... 其他操作符
    };

    std::map<std::string, TokenType> keywordMap = {
        {"if", T_IF},
        // ... 其他关键字
    };

    std::map<std::string, TokenType> delimiterMap = {
        {";", T_SEMICOLON},
        // ... 其他界定符
    };

public:
    Lexer(const std::string& input);
    Token getNextToken();
};
```

3.1.2 私有数据成员

1. **输入字符串和索引：**
   - `std::string input`：存储整个待分析的输入字符串。
   - `size_t index = 0`：当前处理到的字符串索引。
2. **词法元素映射表：**
   - `std::map<std::string, TokenTypeoperatorMap`：存储操作符与其对应的Token类型。
   - `std::map<std::string, TokenType> keywordMap`：存储关键字与其对应的Token类型。
   - `std::map<std::string, TokenType> delimiterMap`：存储界定符与其对应的Token类型。

3.1.3 私有成员函数

1. **基础读取函数：**

- `char peek()`：返回当前索引位置的字符，但不增加索引。
- `char getChar()`：返回当前索引位置的字符，并将索引后移。

2. **辅助函数：**

- `void skipWhitespace()`：跳过空白字符。
- `void skipComment()`：跳过注释。

3. **Token提取函数：**

- `Token getIdentifierOrKeyword()`：获取标识符或关键字Token。
- `Token getCharLiteral()`：获取字符字面量Token。
- `Token getStringLiteral()`：获取字符串字面量Token。
- `Token getNumber()`：获取数字（整数或浮点数）Token。
- `Token getOperator()`：获取操作符Token。
- `Token getDelimiter()`：获取界定符Token。

3.1.4 公共成员函数

Token提取函数：

- `Token getNextToken()`：从输入中获取下一个Token。

3.1.5 部分函数实现细节

1. `char Lexer::peek()`：返回当前索引指向的字符，但不移动索引。如果索引超出输入字符串的范围，则返回空字符 `'\0'`。

```
char Lexer::peek()
{
    if (index < input.size()) {
        return input[index];
    }
    return '\0';
}
```

2. `char Lexer::getChar()`：返回当前索引指向的字符，并将索引向前移动一位。

```
char Lexer::getChar()
{
    return input[index++];
}
```

3. `void Lexer::skipWhitespace()`：跳过输入中的空白字符（包括空格、制表符、换行符等）。

```
void Lexer::skipWhitespace()
{
    while (std::isspace(peek())) {
        getChar();
    }
}
```

4. `Token Lexer::getIdentifierOrKeyword()`：识别标识符或关键字。通过循环读取字母、数字和下划线，构建标识符或关键字的字符串。如果字符串是关键字，则返回相应的 Token 类型；否则，返回标识符的 Token 类型。

```
Token Lexer::getIdentifierOrKeyword()
{
    std::string value;
    while (std::isalnum(peek()) || peek() == '_') {
        value += getChar();
    }

    if (keywordMap.count(value)) {
        return {keywordMap[value], value};
    }
    return {T_IDENTIFIER, value};
}
```

5. `Token Lexer::getOperator()`：识别操作符。通过读取一个字符，检查是否是操作符的一部分（例如双字符操作符），构建操作符的字符串。如果字符串是有效的操作符，则返回相应的 Token 类型；否则，返回未知的 Token 类型。
    1. **初始化变量：** 创建字符串 `value` 并通过 `getChar()` 添加当前字符。
    2. **检查双字符操作符：** 若 `operatorMap` 包含 `value + peek()`，则添加下一字符到 `value`。
    3. **返回Token：** 若 `operatorMap` 包含 `value`，返回对应 `Token`；否则，返回类型为 `T_UNKNOWN` 的 `Token`。

```
Token Lexer::getOperator()
{
    std::string value;
    value += getChar();
    if (operatorMap.count(value + peek())) {
        value += getChar();
    }
    if (operatorMap.count(value)) {
        return {operatorMap[value], value};
    }
    return {T_UNKNOWN, value};
}
```

6. `Token Lexer::getStringLiteral()`：识别字符串字面量。通过循环读取字符，直到遇到闭合的双引号，构建字符串的字符串。返回字符串字面量的 Token。

```
Token Lexer::getStringLiteral()
   {
    std::string value;
    getChar();  // 消耗开始的"
    while (peek() != '"' && peek() != '\0') {
        value += getChar();
    }
    getChar();  //消耗结束的"
    return {T_STRING_LITERAL, value};
   }
```

7. `Token Lexer::getCharLiteral()`：识别字符字面量。通过读取字符，构建字符的字符串。返回字符字面量的 Token。

```
Token Lexer::getCharLiteral()
{
    std::string value;
    getChar();
    value += getChar();
    getChar();
    return {T_CHAR_LITERAL, value};
}
```

8. `Token Lexer::getNumber()`：识别数字字面量。通过循环读取数字和可能的小数点，构建数字的字符串。根据字符串中是否包含小数点确定返回整数或浮点数的 Token。

```
Token Lexer::getNumber()
{
    std::string value;    // 用于存储数字的字符串表示
    bool isFloat = false;  // 用于标识是否是浮点数

    // 循环读取数字和小数点
    while (std::isdigit(peek()) || (!isFloat && peek() == '.')) {
        if (peek() == '.') {
            isFloat = true;  // 标记为浮点数
        }
        value += getChar();  // 将字符添加到数字字符串中
    }

    // 检查数字后是否紧跟字母或下划线，如果是，则消耗直到非字母、非数字、非下划线的字符为止
    if (std::isalpha(peek()) || peek() == '_') {
        while (std::isalnum(peek()) || peek() == '_') {
            value += getChar();  // 将字符添加到数字字符串中，以处理数字后的标识符部
分
        }
        return {T_UNKNOWN, value};  // 返回未知标记类型和值
    }

    // 根据是否为浮点数返回相应的标记类型
    if (isFloat) {
        return {T_FLOAT_LITERAL, value};  // 返回浮点数标记类型和值
    } else {
        return {T_INTEGER_LITERAL, value};  // 返回整数标记类型和值
    }
}
```

9. `Token Lexer::getDelimiter()`：识别界符。通过读取一个字符，检查是否是双字符界符（例如"->"），构建界符的字符串。如果字符串是有效的界符，则返回相应的 Token 类型；否则，返回未知的 Token 类型。

```
Token Lexer::getDelimiter()
{
    std::string value;
    value += getChar();
    // 检查是否有双字符界定符，如 "->"
    if (delimiterMap.count(value + peek())) {
        value += getChar();
    }
    if (delimiterMap.count(value)) {
        return {delimiterMap[value], value};
    }
    return {T_UNKNOWN, value};
}
```

10. `void Lexer::skipComment()`：跳过注释。如果当前字符是 '/'，则根据后续字符是 '/' 还是 '*'，分别处理单行注释和多行注释。

```
void Lexer::skipComment() {
    if (peek() == '/') {
        if (input[index + 1] == '/') {// 单行注释
            while (peek() != '\n' && peek() != '\0') {
                getChar();
            }
            if (peek() == '\n') {
                getChar(); // 消耗换行符
            }
        }
        else if (input[index + 1] == '*') {
            // 多行注释
            getChar(); // 消耗/
            getChar(); // 消耗*
            while (!(peek() == '*' && input[index + 1] == '/')) {
                if (peek() == '\0') {
                    // 提示错误：多行注释没有正确关闭
                    std::cout << "Error: Unclosed multi-line comment";
                    return;
                }
                getChar();
            }
            getChar(); // 消耗*
            getChar(); // 消耗/
        }
    }
}
```

11. `Token Lexer::getNextToken()`：获取下一个 Token。该函数通过一系列的条件判断来确定当前字符属于哪一类 Token，并调用相应的函数进行处理。如果当前字符无法被识别为任何已知的 Token，则返回一个未知 Token。流程如下：

    1. **跳过空白字符和注释：** 调用 `skipWhitespace()` 和 `skipComment()` 函数，以便在输入中跳过空格、制表符、换行符等空白字符，并且跳过注释。

2. **检查是否到达输入末尾：** 如果当前索引 `index` 超过或等于输入字符串的大小 `input.size()`，则返回一个代表文件结束（EOF）的Token，其类型为 `T_EOF`，值为空字符串。
3. **获取当前字符：** 获取当前字符 `c`，它是当前索引位置的字符。
4. **根据字符类型获取Token：** 根据字符 `c` 的类型进入不同的分支，获取相应的Token。
    - 如果 `c` 是字母或下划线，调用 `getIdentifierOrKeyword()` 函数，返回标识符或关键字的Token。
    - 如果 `c` 是双引号，调用 `getStringLiteral()` 函数，返回字符串字面量的Token。
    - 如果 `c` 是单引号，调用 `getCharLiteral()` 函数，返回字符字面量的Token。
    - 如果 `c` 是数字或是小数点，调用 `getNumber()` 函数，返回整数或浮点数字面量的Token。
    - 如果 `c` 是操作符，通过查找操作符映射表 `operatorMap`，调用 `getOperator()` 函数，返回操作符的Token。
    - 如果 `c` 是分隔符，通过查找分隔符映射表 `delimiterMap`，调用 `getDelimiter()` 函数，返回分隔符的Token。
5. **未识别的字符处理：** 如果字符不符合上述任何一种类型，说明它是一个未识别的字符。返回一个Token，类型为 `T_UNKNOWN`，值为包含当前字符的字符串。

整体而言，这段代码实现了一个简单的词法分析器，根据输入字符的不同类型，返回相应的Token。这些Token包括标识符、关键字、字符串字面量、字符字面量、数字字面量、操作符和分隔符等。

```
Token Lexer::getNextToken()
{
    // 跳过空白字符和注释
    skipWhitespace();
    skipComment();
     // 到达输入末尾，返回EOF标记
    if (index >= input.size()) {
        return {T_EOF, ""};
    }

    char c = peek();
    // 根据字符类型,进入不同的分支获取Token
    if (std::isalpha(c) || c == '_') {
        return getIdentifierOrKeyword();
    }
    if (c == '"') {
        return getStringLiteral();
    }
    if (c == '\'') {
        return getCharLiteral();
    }
    if (std::isdigit(c) || (c == '.' && index + 1 < input.size() &&
std::isdigit(input[index + 1]))) {
        return getNumber();
    }
    if (operatorMap.count(std::string(1, c))) {
        return getOperator();
    }
    if (delimiterMap.count(std::string(1, c))) {
        return getDelimiter();
```

```
    }

    // 未识别的字符，返回UNKNOWN类型Token
    return {T_UNKNOWN, std::string(1, getChar())};
}
```

3.2 语法分析器设计

3.2.1 总体设计

语法分析Lexer类的结构如下：

```
class LR1Parser {
public:
    // 构造函数，接受文法产生式、起始符和终止符
    LR1Parser(const std::vector<Production>& productions, Symbol start, Symbol
end);

    // 从文件路径构造LR1Parser对象，用于读取文法产生式
    LR1Parser(const std::string file_path);

    // 打印FIRST集
    void print_firstSet() const;

    // 打印LR1分析表
    void print_tables() const;

    // 对给定句子进行LR1语法分析
    bool parse(const std::vector<Symbol>& sentence) const;

private:
    // 解析EBNF文法行，用于从文法文件中读取产生式
    void parseEBNFLine(const std::string& line);

    // 打印LR1分析器的状态栈、符号栈和输入栈
    void print_stacks(const std::stack<int>& stateStack,
                      const std::stack<Symbol>& symbolStack,
                      const std::vector<Symbol>& inputStack) const;

    // 构造LR1分析表
    void construct_tables();

    // 计算LR1项目集的闭包
    std::set<LR1Item> closure(std::set<LR1Item> lr1ItemSet) const;

private:
    // 获取LR1项目的状态
    LR1Item::State get_lr1item_state(const LR1Item& item) const;

    // 获取以指定符号为起始符的产生式集合
    std::vector<Production> get_productions_start_by_symbol(const Symbol&
symbol) const;
```

```
private:
    // 求FIRST集和FOLLOW集
    void calculateFirstSets();
    std::set<Symbol> firstString(std::vector<Symbol> content) const;

private:
    // 存储文法产生式和相关信息的数据成员
    std::vector<Production> productions;
    std::map<Symbol, std::vector<Production>> productionMap;
    std::map<Symbol, std::set<Symbol>> firstSet, followSet;

    // 起始符和终止符
    Symbol start_symbol;
    Symbol end_symbol;

    // 存储LR1分析表的数据成员
    std::map<std::pair<int, Symbol>, Action> actionTable;  // ACTION表
    std::map<std::pair<int, Symbol>, int> gotoTable;       // GOTO表
    std::vector<std::set<LR1Item>> lr1ItemSets;            // 项目集族

    // 存储终结符集
    std::set<std::string> terminals;
};
```

3.2.2 私有数据成员

- `std::vector<Production> productions;` 存储了语法分析器中使用的所有产生式。
- `std::unordered_map<Symbol, std::vector<Production>, SymbolHash, SymbolEqual> productionMap;` 从 `Symbol` 到 `Production` 向量的映射。它根据一个特定的 `Symbol` 查找以该符号为左侧的所有产生式
- `std::unordered_map<Symbol, std::unordered_set<Symbol, SymbolHash, SymbolEqual>, SymbolHash, SymbolEqual> firstSet` 每个符号的 FIRST 集
- `std::unordered_map<Symbol, std::unordered_set<Symbol, SymbolHash, SymbolEqual>, SymbolHash, SymbolEqual> firstSet` 每个符号的FOLLOW 集
- `Symbol start_symbol;` 语法分析器的起始符号。
- `Symbol end_symbol;` 语法分析器的结束符号。
- `std::map<std::pair<int, Symbol>, Action> actionTable;` 存储 ACTION 表。
- `std::map<std::pair<int, Symbol>, int> gotoTable;` 存储 GOTO 表。
- `std::vector<std::unordered_set<LR1Item, LR1ItemHash, LR1ItemEqual>> lr1ItemSets;` 包含 LR1 项目集的向量。每个项目集都是一个包含 `LR1Item` 的无序集合，用于 LR 解析算法。
- `std::unordered_set<std::string> terminals;` 所有终结符的字符串集合。

3.2.3 私有成员函数

1. **重要功能函数：**

- `void parseEBNFLine(const std::string& line)`：解析EBNF文法行，用于从文件中读取产生式。
- `void calculateFirstSets()`：计算FIRST集。
- `void construct_tables()`：构造LR1分析表（ACTION表和GOTO表）。

2. **辅助功能函数**

- `std::set<LR1Item> closure(std::set<LR1Item> lr1ItemSet) const`：计算LR1项目集的闭包。
- `LR1Item::State get_lr1item_state(const LR1Item& item) const`：获取LR1项目的状态。
- `std::vector<Production> get_productions_start_by_symbol(const Symbol& symbol) const`：获取以指定符号为起始符的产生式集合。
- `std::set<Symbol> firstString(std::vector<Symbol> content) const`：计算一个符号串的FIRST集。

私有数据成员

1. **文法相关：**

- `std::vector<Production> productions`：存储文法产生式。
- `std::map<Symbol, std::vector<Production>> productionMap`：符号到产生式的映射。用于优化查询以某字符开头的产生式。
- `std::map<Symbol, std::set<Symbol>> firstSet`：FIRST集。

2. **符号和项目集：**

- `Symbol start_symbol, end_symbol`：起始符和终止符。
- `std::set<std::string> terminals`：终结符集，用于EBNF解析。

3. **分析表：**

- `std::map<std::pair<int, Symbol>, Action> actionTable`：ACTION表。
- `std::map<std::pair<int, Symbol>, int> gotoTable`：GOTO表。
- `std::vector<std::set<LR1Item>> lr1ItemSets`：项目集族。

3.2.4 公共成员函数

1. **构造函数：**

- `LR1Parser(const std::vector<Production>& productions, Symbol start, Symbol end)`：基于提供的产生式集合、起始符和终止符来构造分析器。（用于测试）
- `LR1Parser(const std::string file_path)`：从文件路径读取文法产生式，用于构造LR1Parser对象。

2. **分析功能：**

- `bool parse(const std::vector<Symbol>& sentence) const`：对给定的句子进行LR1语法分析。

3. **打印功能：**

- `void print_firstSet() const`：打印FIRST集。

- `void print_tables() const`：打印LR1分析表。
- `void print_stacks(const std::stack<int>& stateStack, const std::stack<Symbol>& symbolStack, const std::vector<Symbol>& inputStack) const`：打印LR1分析器的状态栈、符号栈和输入栈。

3.2.5 部分函数实现细节：

1. `LR1Parser::LR1Parser(const std::string file_path)`：构造函数，从文件中读取文法规则，初始化LR1分析器。
    1. **打开文件：**
        - 使用提供的 `file_path` 打开文件，检查文件是否成功打开。
    2. **读取并处理第一行：**
        - 读取文件的第一行，检查是否成功。
        - 使用字符串流解析起始符号和终止符号，并分别存储。
    3. **读取并处理第二行（终结符）：**
        - 读取文件的第二行，检查是否成功。
        - 使用字符串流循环读取终结符，并添加到 `terminals` 集合中。
    4. **解析EBNF规则：**
        - 使用 `while` 循环读取剩余行，并对每行调用 `parseEBNFLine` 方法进行解析。
    5. **计算FIRST集合和构建表：**
        - 调用 `calculateFirstSets` 方法计算FIRST集合。
        - 调用 `construct_tables` 方法构建解析所需的表。
2. `LR1Parser::parseEBNFLine(const std::string& line)`：解析EBNF语法规则中的一行，将其转换为产生式。
    1. **初始化字符串流：** 使用输入行 `line` 初始化字符串流 `iss`。
    2. **读取左侧非终结符：** 从 `iss` 读取左侧非终结符 `lhs`，忽略前导空格。
    3. **忽略产生式定义符号“::=”：** 使用 `std::getline` 和 '=' 分隔符读取并忽略 `::=`。
    4. **读取产生式右侧：** 使用 `std::getline` 读取产生式右侧到 `rhsPart`，并初始化另一个字符串流 `rhsStream`。
    5. **分割产生式右侧：** 使用 `'|'` 分割 `rhsStream`，循环处理每个分割得到的 `token`。
    6. **解析并构建符号列表：**
    - 对于每个 `token`，初始化字符串流 `tokenStream`。
    - 循环读取 `tokenStream` 中的符号 `sym`，判断其类型（终结符、非终结符或Epsilon），并添加到符号列表 `rhsSymbols`。
    7. **创建并添加产生式：**
    - 对于每个分割的 `token`，使用 `lhs` 和 `rhsSymbols` 创建产生式。
    - 将产生式添加到 `productionMap` 和 `productions` 容器中。

```
void LR1Parser::parseEBNFLine(const std::string& line)
{
    std::istringstream iss(line);
    std::string lhs, rhsPart, token;

    // 读取非终结符，自动忽略前导空格
    iss >> lhs;
    // 忽略 "::="
```

```
    std::string ignoreStr;
    std::getline(iss, ignoreStr, '=');

    // 读取并解析产生式右侧
    std::getline(iss, rhsPart);
    std::istringstream rhsStream(rhsPart);

    // 用 '|' 分割产生式右边的语句
    while (std::getline(rhsStream, token, '|')) {
        std::istringstream tokenStream(token);
        std::vector<Symbol> rhsSymbols;

        // 获取产生式右边的符号
        std::string sym;
        while (tokenStream >> sym) {
            SymbolType type;
            if (sym == "Epsilon") {
                type = SymbolType::Epsilon;
            } else {
                type = terminals.find(sym) == terminals.end() ?
SymbolType::NonTerminal : SymbolType::Terminal;
            }
            rhsSymbols.push_back(Symbol(type, sym));
        }

        // 创建产生式并添加到某个容器中
        productionMap[Symbol(SymbolType::NonTerminal,
lhs)].push_back(Production{Symbol(SymbolType::NonTerminal, lhs),
rhsSymbols});
        productions.push_back(Production(Symbol(SymbolType::NonTerminal,
lhs), rhsSymbols));
    }
}
```

3. `LR1Parser::calculateFirstSets()`：计算文法符号的FIRST集合。算法流程及代码如下：
    1. **初始化FIRST集：** 遍历所有产生式，对终结符和空串进行初始化。对于终结符和空串，将其加入对应的FIRST集，并对非终结符的FIRST集进行初始化为空。
    2. **迭代计算FIRST集：** 使用do-while循环进行迭代，直到没有更改。在每次迭代中，遍历所有产生式，更新每个非终结符的FIRST集。
    3. **遍历每个非终结符的产生式：** 对于每个非终结符，遍历其所有产生式。
    4. **遍历产生式右侧的符号：** 对于产生式右侧的每个符号，考虑其FIRST集。
    5. **处理非终结符：** 如果符号是非终结符，将其FIRST集加入当前非终结符的FIRST集。
    6. **处理终结符：** 如果符号是终结符，将其作为独立的符号加入当前非终结符的FIRST集。
    7. **处理空串：** 如果符号是空串，将空串加入当前非终结符的FIRST集。
    8. **检查FIRST集的变化：** 在每次迭代中，检查FIRST集是否有变化，如果发生变化，将 `changed` 标记为 `true`。
    9. **处理非终结符的epsilon：** 对于非终结符，如果它的所有产生式都可以导出空串，则将空串加入该非终结符的FIRST集。
    10. **重复直到没有变化：** 重复上述步骤，直到没有FIRST集的变化发生。

```
void LR1Parser::calculateFirstSets()
{
```

```
    for (const auto& [lhs, rhs] : productions) {
        firstSet[lhs] = {};  // 初始化非终结符的FIRST集合为空
        for (const auto& symbol : rhs) {
            if (symbol.type == SymbolType::NonTerminal) {
                firstSet[symbol] = {};  // 初始化非终结符的FIRST集合为空
            } else if (symbol.type == SymbolType::Terminal) {
                firstSet[symbol] = {symbol};  // 终结符的FIRST集合包含它自己
            } else if (symbol.type == SymbolType::Epsilon) {
                firstSet[symbol] = {Symbol(SymbolType::Epsilon, "")};  // 空
串的FIRST集合包含空串
            }
        }
    }

    bool changed;
    do {
        changed = false;
        for (const auto& [nonTerminal, productions] : productionMap) {
            for (const auto& prod : productions) {
                size_t i;
                for (i = 0; i < prod.rhs.size(); ++i) {
                    const Symbol& symbol = prod.rhs[i];
                    std::set<Symbol> symbolFirstSet = firstSet[symbol];
                    size_t oldSize = firstSet[nonTerminal].size();
                    bool has_epsilon = true;
                    // 如果符号不包含空串，跳出循环
                    if (symbolFirstSet.find(Symbol(SymbolType::Epsilon, ""))
== symbolFirstSet.end()) {
                        has_epsilon = false;
                    }
                    // 如果不是最后一个符号，移除空串
                    if (i < prod.rhs.size() - 1) {
                        symbolFirstSet.erase(Symbol(SymbolType::Epsilon,
""));
                    }
                    firstSet[nonTerminal].insert(symbolFirstSet.begin(),
symbolFirstSet.end());
                    // 检查FIRST集是否发生变化
                    if (firstSet[nonTerminal].size() != oldSize) {
                        changed = true;
                    }

                    if (!has_epsilon) break;
                }
                // 如果产生式中的所有符号都能导出空串，将空串加入到
FIRST(nonTerminal)中
                if (i == prod.rhs.size()) {
                    firstSet[nonTerminal].insert(Symbol(SymbolType::Epsilon,
""));
                }
            }
        }
    } while (changed);  // 重复直到没有变化为止
}
```

4. `LR1Parser::firstString(std::vector<Symbol> content) const`：计算一个符号串的FIRST集合。算法流程及代码如下：
    1. **初始化空集合：** 创建一个空的Symbol集合 `ret`，用于存储生成的FIRST集。
    2. **遍历输入符号序列：** 对于输入的每个符号 `symbol`，进行以下操作：
        - 如果 `symbol` 是空串，继续下一次循环（不将空串加入FIRST集）。
        - 如果 `symbol` 是终结符，将其加入到 `ret` 中，然后跳出循环。
        - 如果 `symbol` 是非终结符，获取其FIRST集，并将其中的空串移除。将该FIRST集合并入 `ret` 中。
        - 如果 `ret` 中的元素个数大于0，说明当前符号的FIRST集非空，跳出循环。
    3. **返回结果：** 返回构建的Symbol集合 `ret`，表示输入符号序列的FIRST集。

```
std::set<Symbol> LR1Parser::firstString(std::vector<Symbol> content) const
{
    // 用于存储计算结果的集合
    std::set<Symbol> ret;
    // 遍历输入的符号序列
    for (const auto& symbol : content) {
        // 如果当前符号是ϵ（空串），则忽略
        if (symbol.type == SymbolType::Epsilon) continue;
        // 如果当前符号是终结符，将其加入结果集并结束循环
        if (symbol.type == SymbolType::Terminal) {
            ret.insert(symbol);
            break;
        }
        // 如果当前符号是非终结符，获取其FIRST集
        std::set<Symbol> symbolFirstSet = firstSet.at(symbol);
        // 从FIRST集中移除ϵ（空串）
        symbolFirstSet.erase(Symbol(SymbolType::Epsilon, ""));
        // 将当前符号的FIRST集合并入结果集
        ret.insert(symbolFirstSet.begin(), symbolFirstSet.end());
        // 如果结果集非空，结束循环
        if (ret.size() > 0) break;
    }
    // 返回计算得到的FIRST集
    return ret;
}
```

5. `LR1Parser::get_lr1item_state(const LR1Item& item) const`：获取LR1项的状态，包括移进、归约、待约和接受。

```
LR1Item::State LR1Parser::get_lr1item_state(const LR1Item& item) const
{
    const auto& production = item.production;
    const auto& dot_position = item.dot_position;
    const auto& lookahead = item.lookahead;

    // 检查是否是接受状态
    if (production.lhs == start_symbol && dot_position ==
production.rhs.size() && lookahead == end_symbol) {
        return LR1Item::State::ACCEPT;
    }
```

```
    // 检查是否是归约状态
    if (dot_position == production.rhs.size()) {
        return LR1Item::State::REDUCE;
    }

    // 检查是否是移进状态
    if (dot_position < production.rhs.size() &&
production.rhs[dot_position].type == SymbolType::Terminal) {
        return LR1Item::State::SHIFT;
    }

    // 检查是否是转换状态
    if (dot_position < production.rhs.size() &&
production.rhs[dot_position].type == SymbolType::NonTerminal) {
        return LR1Item::State::GOTO;
    }

    // 如果没有匹配的状态，可能是一个错误或未定义的状态
    throw std::runtime_error("Undefined state for LR1 item.");
}
```

6. `LR1Parser::closure(std::set<LR1Item> old_set) const`：计算LR1项集的闭包。算法流程及代码如下：
    1. **初始化循环：** 进入无限循环，直到闭包不再发生变化。
    2. **创建新的闭包集合：** 创建一个新的闭包集合 `new_set`，并将其初始化为当前闭包集合 `old_set`。
    3. **遍历当前闭包集合：** 对于当前闭包集合中的每个LR(1)项 `item`，执行以下操作：
        - 只处理GOTO状态（.后面是非终结符）的项，其他状态不进行处理。
    4. **获取当前项的下一个和下下一个符号：** 从当前项中获取下一个符号 `next_symbol` 和下下一个符号 `nnext_symbol`，用于后续操作。
    5. **查找以 `next_symbol` 为左部的产生式：** 获取文法中所有以 `next_symbol` 为左部的产生式 `next_prods`。
    6. **对每个产生式进行处理：** 对于 `next_prods` 中的每个产生式 `prod`，执行以下操作：
        - 计算当前项的 `nnext_symbol` 和 `lookahead` 的FIRST集合 `lookheads`，调用 `firstString` 函数。
        - 对于 `lookheads` 中的每个符号，创建新的LR(1)项并加入到 `new_set` 中。
    7. **检查闭包集合是否发生变化：** 检查当前闭包集合 `old_set` 和新闭包集合 `new_set` 是否相等，如果相等说明闭包已经不再发生变化，跳出循环。
    8. **更新闭包集合：** 将新的闭包集合 `new_set` 赋值给当前闭包集合 `old_set`。
    9. **返回结果：** 返回最终构建的闭包集合 `old_set`。

```
std::set<LR1Item> LR1Parser::closure(std::set<LR1Item> old_set) const
{
    // 不断迭代，直到闭包不再增大
    while (true) {
        // 复制当前闭包
        std::set<LR1Item> new_set(old_set);
```

```
        // 遍历当前闭包中的每个LR1项目
        for (auto it = old_set.begin(); it != old_set.end(); it++) {
            auto& item = *it;

            // 只有GOTO状态(.后面是非终结符)才需要处理
            if (get_lr1item_state(item) != LR1Item::State::GOTO) continue;

            // 获取当前项目的后继符号和后继的后继符号
            Symbol next_symbol = item.next_symbol();
            Symbol nnext_symbol = item.nnext_symbol();

            // 找到文法中所有以next_symbol为lhs的产生式
            std::vector<Production> next_prods =
get_productions_start_by_symbol(next_symbol);
            for (auto& prod : next_prods) {
                // 计算产生式右侧的符号串的FIRST集
                std::set<Symbol> lookaheads = firstString({nnext_symbol,
item.lookahead});
                // 将新的LR1项目加入闭包
                for (auto p_lh = lookaheads.begin(); p_lh !=
lookaheads.end(); p_lh++) {
                    new_set.insert(LR1Item(prod, 0, *p_lh));
                }
            }
        }
        // 如果新旧闭包相等，结束循环
        if (old_set == new_set) break;
        // 更新闭包为新的闭包
        old_set = new_set;
    }
    // 返回计算得到的闭包
    return old_set;
}
```

7. `LR1Parser::construct_tables()`：构造LR1分析表，包括ACTION表和GOTO表。算法流程及代码如下：
    1. **获取文法的开始产生式：** 通过调用 `get_productions_start_by_symbol` 函数，获取文法的开始产生式 `begin_production`。
    2. **初始化LR1分析表的第一个状态集：** 创建LR1分析表的初始状态集，其中包含初始项目 `[S' -> .S, $]`，并计算该状态集的闭包。
    3. **遍历LR1分析表的所有状态集：** 使用循环遍历LR1分析表中的每个状态集，执行以下步骤。
    4. **分类LR1项目：** 对于当前状态集中的每个LR1项目，根据项目的状态类型（ACCEPT、GOTO、REDUCE、SHIFT）将项目分类，并将相关信息存储在不同的容器中。
    5. **处理移进动作：** 遍历当前状态集中的移进项目，对每个终结符 `vt` 执行以下操作：
        - 构造新的项目集 `new_set`，包含当前状态集中具有SHIFT状态的项目，并计算该项目集的闭包。
        - 如果新的项目集在LR1分析表中不存在，将其加入LR1分析表。
        - 在LR1分析表的 `actionTable` 中添加移进动作。
    6. **处理GOTO动作：** 遍历当前状态集中的GOTO项目，对每个非终结符 `vn` 执行以下操作：

- 构造新的项目集 `new_set`，包含当前状态集中具有GOTO状态的项目，并计算该项目集的闭包。
- 如果新的项目集在LR1分析表中不存在，将其加入LR1分析表。
- 在LR1分析表的 `gotoTable` 中添加GOTO动作。

7. **处理接受动作：** 对于当前状态集中的接受项目，将接受动作添加到LR1分析表的 `actionTable` 中。
8. **处理归约动作：** 对于当前状态集中的归约项目，将归约动作添加到LR1分析表的 `actionTable` 中。
9. **返回结果：** 完成LR1分析表的构建。

```
void LR1Parser::construct_tables()
{
  // 获取文法的开始产生式
  Production begin_production =
get_productions_start_by_symbol(start_symbol).at(0);

  // 初始化LR1分析表的第一个状态集，包含了初始项目 [S' -> .S, $]
  lr1ItemSets.push_back(std::set<LR1Item>({LR1Item(begin_production, 0,
end_symbol)}));
  lr1ItemSets[0] = closure(lr1ItemSets[0]); // 计算闭包

  // 遍历LR1分析表的所有状态集
  for (size_t index = 0; index < lr1ItemSets.size(); ++index)
  {
    std::vector<LR1Item> shift_items, reduce_items, accept_items,
goto_items;
    std::vector<Symbol> VT_shift, VN_goto;

    // 遍历当前状态集中的LR1项目
    for (auto &item : lr1ItemSets[index])
    {
      switch (get_lr1item_state(item))
      {
      case LR1Item::State::ACCEPT:
        accept_items.push_back(item);
        break;
      case LR1Item::State::GOTO:
        goto_items.push_back(item);
        VN_goto.push_back(item.next_symbol());
        break;
      case LR1Item::State::REDUCE:
        reduce_items.push_back(item);
        break;
      case LR1Item::State::SHIFT:
        shift_items.push_back(item);
        VT_shift.push_back(item.next_symbol());
        break;
      default:
        std::cerr << "construct_tables() 中LR1Item类型错误" << std::endl;
      }
    }

    // 处理移进动作
```

```
    for (auto &vt : VT_shift)
    {
      std::set<LR1Item> new_set;
      for (auto &item : lr1ItemSets[index])
      {
        if (get_lr1item_state(item) == LR1Item::State::SHIFT &&
item.next_symbol() == vt)
        {
          // 计算新的项目集
          new_set.insert(LR1Item(item.production, item.dot_position + 1,
item.lookahead));
        }
      }
      new_set = closure(new_set); // 计算闭包
      size_t id = find(lr1ItemSets.begin(), lr1ItemSets.end(), new_set) -
lr1ItemSets.begin();
      if (id == lr1ItemSets.size())
      {
        lr1ItemSets.push_back(new_set); // 将新的项目集加入LR1分析表
      }
      actionTable[{index, vt}] = Action(Action::Type::SHIFT, id,
Production());
    }

    // 处理GOTO动作
    for (auto &vn : VN_goto)
    {
      std::set<LR1Item> new_set;
      for (auto &item : lr1ItemSets[index])
      {
        if (get_lr1item_state(item) == LR1Item::State::GOTO &&
item.next_symbol() == vn)
        {
          // 计算新的项目集
          new_set.insert(LR1Item(item.production, item.dot_position + 1,
item.lookahead));
        }
      }
      new_set = closure(new_set); // 计算闭包
      size_t id = find(lr1ItemSets.begin(), lr1ItemSets.end(), new_set) -
lr1ItemSets.begin();
      if (id == lr1ItemSets.size())
      {
        lr1ItemSets.push_back(new_set); // 将新的项目集加入LR1分析表
      }
      gotoTable[{index, vn}] = id;
    }

    // 处理接受动作
    for (auto &item : accept_items)
    {
      actionTable[{index, item.lookahead}] = Action(Action::Type::ACCEPT,
-1, Production());
    }
```

```
    // 处理归约动作
    for (auto &item : reduce_items)
    {
      actionTable[{index, item.lookahead}] = Action(Action::Type::REDUCE,
-1, item.production);
    }
  }
}
```

8. `LR1Parser::parse(const std::vector<Symbol>& sentence) const`：对给定输入符号串进行LR1语法分析。
    1. **初始化栈和输入栈：**
        - 创建一个整数类型的状态栈 `stateStack`，用于存放LR1分析器的状态。
        - 创建一个符号类型的栈 `symbolStack`，用于存放LR1分析器的符号。
        - 创建一个符号类型的输入栈 `inputStack`，用于存放待分析的输入符号串（反向排列）。
    2. **推入初始状态：**
        - 将初始状态0推入状态栈。
    3. **分析循环：**
        - 进入循环，直到输入栈为空。
    4. **获取当前状态和输入符号：**
        - 获取状态栈的顶部状态作为当前状态。
        - 获取输入栈的末尾符号作为当前输入符号。
    5. **打印当前栈状态：**
        - 调用 `print_stacks` 函数打印当前状态栈、符号栈和输入栈的状态。
    6. **查找并执行ACTION表中的动作：**
        - 在ACTION表中查找对应的动作，由 `{currentState, currentSymbol}` 键值对确定。
        - 如果找到了对应动作，执行相应的操作：
            - 移进操作（`Action::Type::SHIFT`）：
                - 将移进的状态推入状态栈。
                - 将当前输入符号推入符号栈。
                - 从输入栈中弹出当前输入符号。
            - 归约操作（`Action::Type::REDUCE`）：
                - 根据产生式右侧的长度，从符号栈和状态栈中弹出相应数量的符号和状态。
                - 将产生式左侧的非终结符推入符号栈。
                - 根据GOTO表，获取下一个状态，推入状态栈。
            - 接受操作（`Action::Type::ACCEPT`）：
                - 打印 "Accept"，表示分析成功。
                - 返回 `true`。
        - 如果在ACTION表中未找到对应的动作，打印错误信息并返回 `false`。
    7. **循环结束：**
        - 如果循环结束时输入栈未完全消耗，打印错误信息并返回 `false`。

```
bool LR1Parser::parse(const std::vector<Symbol> &sentence) const
{
  std::stack<int> stateStack;               // 状态栈，用于存放LR1分析器的状态
```

```
  std::stack<Symbol> symbolStack;          // 符号栈，用于存放LR1分析器的符号
  std::vector<Symbol> inputStack(sentence.rbegin(), sentence.rend());

  // 初始状态
  stateStack.push(0);

  while (!inputStack.empty())
  {
    int currentState = stateStack.top();      // 当前状态
    Symbol currentSymbol = inputStack.back(); // 当前输入符号

    // 打印当前栈的状态
    print_stacks(stateStack, symbolStack, inputStack);

    auto actionIt = actionTable.find({currentState, currentSymbol});
    if (actionIt != actionTable.end())
    {
      const Action &action = actionIt->second;

      switch (action.type)
      {
      case Action::Type::SHIFT:
      {
        // 移进操作
        stateStack.push(action.number);
        symbolStack.push(currentSymbol);
        inputStack.pop_back();
        break;
      }
      case Action::Type::REDUCE:
      {
        // 根据产生式右侧的长度，从栈中弹出相应数量的符号和状态
        for (size_t i = 0; i < action.production.rhs.size(); ++i)
        {
          symbolStack.pop();
          stateStack.pop();
        }
        // 将产生式左侧的非终结符压入符号栈
        symbolStack.push(action.production.lhs);

        // 更新状态栈
        int nextState = gotoTable.at({stateStack.top(),
action.production.lhs});
        stateStack.push(nextState);
        break;
      }
      case Action::Type::ACCEPT:
        // 接受状态，分析成功
        std::cout << "Accept\n";
        return true;
      default:
        // 未定义的操作类型，发生错误
        std::cerr << "Parse error\n";
        return false;
      }
```

```
        }
        else
        {
            // 在ACTION表中未找到对应的动作，发生错误
            std::cerr << "Parse error: no action\n";
            return false;
        }
    }

    // 输入符号未完全消耗，发生错误
    std::cerr << "Parse error: input not consumed\n";
    return false;
}
```

3.3 中间代码生成器设计

3.3.1 总体设计

语法分析Lexer类的结构如下：

```
class SemanticAnalyzer {
public:
    SemanticAnalyzer(SemanticTreeNode*& root) : root(root),
next_temp_varible_id(0) {}

    void semantic_analyze();
    void print_intermediate_code();
    void print_variable_table();

private:
    void handle_defalt(SemanticTreeNode*& node);
    void handle_var_declaration(SemanticTreeNode*& node);
    void handle_opt_init(SemanticTreeNode*& node);
    void handle_expression(SemanticTreeNode*& node);
    void handle_simple_expression(SemanticTreeNode*& node);
    void handle_additive_expression(SemanticTreeNode*& node);
    void handle_term(SemanticTreeNode*& node);
    void handle_postfix_expression(SemanticTreeNode*& node);
    void handle_var(SemanticTreeNode*& node);
    void handle_factor(SemanticTreeNode*& node);
    void handle_prefix_expression(SemanticTreeNode*& node);
    void handle_selection_stmt(SemanticTreeNode*& node);
    void handle_iteration_stmt(SemanticTreeNode*& node);
    void handle_opt_expression_stmt(SemanticTreeNode*& node);

private:
    std::string
    new_temp_varible()
    {
        return "t" + std::to_string(next_temp_varible_id++);
    }

private:
    bool exists_var_declaration(std::string var)
    {
```

```
        return varible_table.find(var) != varible_table.end();
    }

private:
    SemanticTreeNode* root;

    std::map<std::string, VaribleMeta> varible_table;  // 变量表，存储已经声明过的变
量

    std::vector<std::pair<size_t, Quater>> intermediate_code() { return root-
>quater_list; }  // 中间代码，一个数字编号，一个四元式

    size_t next_temp_varible_id;
};
```

3.3.2 私有数据成员

- `SemanticTreeNode* root;` 这个指针指向语义树的根节点，语义分析过程基于这个语义树进行。
- `std::map<std::string, VaribleMeta> varible_table;` 用于存储已经声明的变量及其信息（`VaribleMeta`）。这个表在语义分析过程中用于查找变量的声明和属性。
- `std::vector<std::pair<size_t, Quater>> intermediate_code;` 中间代码。每个元素是一个由编号（`size_t`）和四元式（`Quater`）组成的对。这代表了在分析过程中生成的中间表示形式。
- `size_t next_temp_varible_id;` 这是一个计数器，用于生成新的临时变量名。每次生成新的临时变量时，这个值会自增。

3.3.3 私有成员函数

- `std::string new_temp_varible()` 生成新的临时变量名。基于 `next_temp_varible_id` 计数器生成名字，并更新计数器。
- `bool exists_var_declaration(std::string var)` 检查给定的变量名是否已在变量表中声明。防止变量名重复声明。
- `handle_defalt(SemanticTreeNode*& node)` 处理默认的语法结构，通常是在没有特定处理函数的情况下调用。
- `handle_var_declaration(SemanticTreeNode*& node)` 处理变量声明语法结构。
- `handle_opt_init(SemanticTreeNode*& node)` 处理可选的初始化语法结构，通常用于变量声明时的初始化。
- `handle_expression(SemanticTreeNode*& node)` 处理表达式语法结构。
- `handle_simple_expression(SemanticTreeNode*& node)` 处理简单表达式语法结构，通常是没有运算符的表达式。
- `handle_additive_expression(SemanticTreeNode*& node)` 处理加法表达式语法结构，包括加法和减法。
- `handle_term(SemanticTreeNode*& node)` 处理项语法结构，通常是乘法和除法表达式。
- `handle_postfix_expression(SemanticTreeNode*& node)` 处理后缀表达式语法结构，例如函数调用。
- `handle_var(SemanticTreeNode*& node)` 处理变量语法结构。
- `handle_factor(SemanticTreeNode*& node)` 处理因子语法结构，因子是表达式的基本组成部分。
- `handle_prefix_expression(SemanticTreeNode*& node)` 处理前缀表达式语法结构，例如负数和逻辑非表达式。
- `handle_selection_stmt(SemanticTreeNode*& node)` 处理选择语句语法结构，例如if语句。

- `handle_iteration_stmt(SemanticTreeNode*& node)`处理迭代语句语法结构，例如for和while循环。
- `handle_opt_expression_stmt(SemanticTreeNode*& node)`处理可选的表达式语句语法结构，用于处理可以省略的表达式语句。

3.3.4 公共成员函数

- `SemanticAnalyzer(SemanticTreeNode*& root)`构造函数。接收一个指向语义树根节点的指针，用于初始化 `root` 成员。
- `void semantic_analyze()`执行语义分析。基于语义树 `root`，进行语义规则的检查和中间代码的生成。
- `void print_intermediate_code()`打印中间代码。输出生成的中间代码的表示，通常用于调试和理解代码生成过程。
- `void print_variable_table()`打印变量表。输出所有已声明变量的信息，通常用于调试和验证变量声明。

3.3.5 部分函数实现细节

1. `void semantic_analyze()`函数是语义分析器的实现，它用于对语法树进行语义分析。

- 首先，函数检查语法树的根节点是否为空，如果为空，则直接返回，不进行任何处理。
- 接下来，函数使用两个栈 `stack1` 和 `stack2` 来进行深度优先遍历。将根节点压入 `stack1` 中。将语法树按照深度优先的顺序存储在 `stack2` 中。
- 对于语法树每个节点，我们可以根据其 `literal` 属性来确定它的类型，并根据类型调用相应的处理函数。这些处理函数可以根据需要进行自定义，用于处理特定类型的节点。例如，可以打印节点的信息或执行其他操作。
- 在处理节点之后，如果节点是叶子节点，则跳过后续的处理步骤。如果节点的 `literal` 属性与特定的字符串匹配，那么调用相应的处理函数。这些字符串可能是语法规则中的非终结符或终结符。
- 最后，如果节点只有一个子节点，并且该子节点的 `literal` 属性是 "T_IDENTIFIER"，则将节点的 `literal` 属性更新为该子节点的 `literal` 属性。这可能是为了处理标识符节点的特殊情况。

```
void SemanticAnalyzer::semantic_analyze()
{
    if (root == nullptr) return;

    std::stack<SemanticTreeNode*> stack1, stack2;
    stack1.push(root);

    while (!stack1.empty()) {
        SemanticTreeNode* node = stack1.top();
        stack1.pop();
        stack2.push(node);

        for (SemanticTreeNode* child : node->children) {
            stack1.push(child);
        }
    }

    while (!stack2.empty()) {
        SemanticTreeNode* node = stack2.top();
```

```
        stack2.pop();

        // Process node - this can be customized as needed
        // For example, print node information
        if (node->leaf()) {
            continue;
        }

        handle_defalt(node);

        if (node->literal == "var_declaration") {
            handle_var_declaration(node);
        } else if (node->literal == "opt_init") {
            handle_opt_init(node);
        } else if (node->literal == "expression") {
            handle_expression(node);
        } else if (node->literal == "simple_expression") {
            handle_simple_expression(node);
        } else if (node->literal == "additive_expression") {
            handle_additive_expression(node);
        } else if (node->literal == "term") {
            handle_term(node);
        } else if (node->literal == "postfix_expression") {
            handle_postfix_expression(node);
        } else if (node->literal == "factor") {
            handle_factor(node);
        } else if (node->literal == "prefix_expression") {
            handle_prefix_expression(node);
        } else if (node->literal == "selection_stmt") {
            handle_selection_stmt(node);
        } else if (node->literal == "iteration_stmt") {
            handle_iteration_stmt(node);
        } else if (node->literal == "opt_expression_stmt") {
            handle_opt_expression_stmt(node);
        }

        if (node->children.size() == 1 && node->children[0]->literal ==
"T_IDENTIFIER") {
            node->literal = node->children[0]->literal;
        }
    }
}
```

2. `SemanticAnalyzer::handle_iteration_stmt`是`SemanticAnalyzer`类的一个成员函数，用于处理循环语句的语义分析。

- 函数首先获取节点的子节点列表。然后，根据子节点列表的第一个元素的`literal`属性来判断是哪种类型的循环语句。
- 如果`literal`属性是"T_WHILE"，那么这是一个while循环。
    - 函数获取条件表达式的实际值和循环体的语句，然后计算出循环开始、循环体和循环结束的位置。
    - 接着，函数将条件表达式的四元式列表添加到节点的四元式列表中，然后添加一个条件跳转四元式和一个无条件跳转四元式。

    - 最后，函数将循环体的四元式列表添加到节点的四元式列表中，并添加一个无条件跳转四元式，跳转到循环开始的位置。
- 如果`literal`属性是"T_FOR"，那么这是一个for循环。
    - 函数获取初始化表达式、条件表达式、更新表达式和循环体的语句，然后计算出循环开始、循环体和循环结束的位置。
    - 接着，函数将初始化表达式、条件表达式的四元式列表添加到节点的四元式列表中，然后添加一个条件跳转四元式和一个无条件跳转四元式。
    - 最后，函数将循环体的四元式列表、更新表达式的四元式列表添加到节点的四元式列表中，并添加一个无条件跳转四元式，跳转到循环开始的位置。

```
//循环语句语义分析
void SemanticAnalyzer::handle_iteration_stmt(SemanticTreeNode*& node)
{
    const auto& list = node->children;


    if (list[0]->literal == "T_WHILE") {
        /*
        T_WHILE T_LEFT_PAREN expression T_RIGHT_PAREN statement
        */
        const auto& cond = list[2]->real_value;
        const auto& stmt = list[4];
        size_t LOOP = 0;
        size_t BODY = LOOP + list[2]->quater_list.size() + 2;
        size_t END_LOOP = BODY + stmt->quater_list.size() + 1;

        node->append_quaters(list[2]->quater_list);
        node->add_quater("jnz", cond, "", BODY);
        node->add_quater("j", "", "", END_LOOP);
        node->append_quaters(stmt->quater_list);
        node->add_quater("j", "", "", LOOP);
    } else if (list[0]->literal == "T_FOR") {
        /*
        T_FOR T_LEFT_PAREN opt_expression_stmt opt_expression_stmt expression
T_RIGHT_PAREN statement
        */
        const auto& exp1 = list[2];
        const auto& exp2 = list[3];
        const auto& exp3 = list[4];
        const auto& stmt = list[6];

        size_t START = exp1->quater_list.size();
        size_t BODY = START + exp2->quater_list.size() + 2;
        size_t END_LOOP = BODY + stmt->quater_list.size() + exp3-
>quater_list.size() + 1;

        node->append_quaters(exp1->quater_list);
        node->append_quaters(exp2->quater_list);
        node->add_quater("jnz", exp2->real_value, "", BODY);
        node->add_quater("j", "", "", END_LOOP);
        node->append_quaters(stmt->quater_list);
        node->append_quaters(exp3->quater_list);
        node->add_quater("j", "", "", START);
    }
```

```
}
```

3. `handle_selection_stmt`函数，用于处理选择语句（如if-else语句）的语义分析。

- 函数首先获取节点的子节点列表。
- 然后，获取条件表达式的实际值和if语句的主体部分，将条件表达式的四元式列表添加到节点的四元式列表中。
- 接着，函数根据子节点列表的大小来判断是单独的if语句还是if-else语句。
- 如果子节点列表的大小为5，那么这是一个单独的if语句。
    - 函数计算出if语句主体和结束if语句的位置，
    - 然后添加一个条件跳转四元式和一个无条件跳转四元式。
    - 最后，将if语句主体的四元式列表添加到节点的四元式列表中。
- 如果子节点列表的大小为7，那么这是一个if-else语句。
    - 函数获取else语句的主体部分，然后计算出else语句、if语句主体和结束if语句的位置。
    - 接着，添加一个条件跳转四元式，将else语句主体的四元式列表添加到节点的四元式列表中，
    - 然后添加一个无条件跳转四元式，
    - 最后将if语句主体的四元式列表添加到节点的四元式列表中。

```
void SemanticAnalyzer::semantic_analyze()
  {
    if (root == nullptr) return;

    std::stack<SemanticTreeNode*> stack1, stack2;
    stack1.push(root);

    while (!stack1.empty()) {
        SemanticTreeNode* node = stack1.top();
        stack1.pop();
        stack2.push(node);

        for (SemanticTreeNode* child : node->children) {
            stack1.push(child);
        }
    }

    while (!stack2.empty()) {
        SemanticTreeNode* node = stack2.top();
        stack2.pop();

        // Process node - this can be customized as needed
        // For example, print node information
        if (node->leaf()) {
            continue;
        }

        handle_defalt(node);

        if (node->literal == "var_declaration") {
```

```
            handle_var_declaration(node);
        } else if (node->literal == "opt_init") {
            handle_opt_init(node);
        } else if (node->literal == "expression") {
            handle_expression(node);
        } else if (node->literal == "simple_expression") {
            handle_simple_expression(node);
        } else if (node->literal == "additive_expression") {
            handle_additive_expression(node);
        } else if (node->literal == "term") {
            handle_term(node);
        } else if (node->literal == "postfix_expression") {
            handle_postfix_expression(node);
        } else if (node->literal == "factor") {
            handle_factor(node);
        } else if (node->literal == "prefix_expression") {
            handle_prefix_expression(node);
        } else if (node->literal == "selection_stmt") {
            handle_selection_stmt(node);
        } else if (node->literal == "iteration_stmt") {
            handle_iteration_stmt(node);
        } else if (node->literal == "opt_expression_stmt") {
            handle_opt_expression_stmt(node);
        }

        if (node->children.size() == 1 && node->children[0]->literal ==
"T_IDENTIFIER") {
            node->literal = node->children[0]->literal;
        }
    }
  }
```

4. `handle_var_declaration`的函数，它是`SemanticAnalyzer`类的一个成员函数，用于处理变量声明的语义分析。

- 函数首先获取节点的子节点列表。然后，从子节点列表中获取变量的类型、变量名和初始值。如果没有初始值，那么初始值就设为"NULL"。
- 接着，函数检查变量表中是否已经存在相同的变量名。如果存在，那么就输出错误信息并退出程序。这是为了防止变量的重定义。
- 如果变量表中不存在相同的变量名，那么就将变量的类型、变量名和初始值添加到变量表中。这是为了在后续的语义分析中能够找到这个变量。
- 最后，函数更新节点的实际值。它将变量的类型、变量名和初始值拼接成一个字符串，并将这个字符串赋值给节点的实际值。这是为了在后续的语义分析中能够获取到这个变量的完整信息。

```
void SemanticAnalyzer::handle_var_declaration(SemanticTreeNode*& node)
{
    // type_specifier T_IDENTIFIER opt_init T_SEMICOLON
    // type_specifier T_IDENTIFIER T_SEMICOLON
    const auto& list = node->children;

    const std::string& type = list[0]->real_value;
    const std::string& varible_name = list[1]->real_value;
```

```
    const std::string& init_val = list[2]->literal == "opt_init" ? list[2]->real_value : "NULL";

    if (varible_table.find(varible_name) != varible_table.end()) {
        // 如果变量表中已经有了这个变量，报错
        std::cout << "Error: 重定义变量: " << varible_name << std::endl;
        exit(-1);
    }

    varible_table[varible_name] = {type, init_val};
    node->real_value = type + " " + varible_name + " " + init_val + ";";
}
```

5. `handle_expression`的函数，它是`SemanticAnalyzer`类的一个成员函数，用于处理表达式的语义分析。

- 函数首先获取节点的子节点列表。如果子节点列表的大小为1，那么函数直接返回，不进行任何操作。这是因为如果子节点列表的大小为1，那么这个表达式就是一个简单表达式或者后缀表达式，不需要进行任何处理。
- 如果子节点列表的大小不为1，那么这个表达式就是一个赋值表达式。函数从子节点列表中获取变量名、操作符和表达式的实际值。
- 接着，函数检查变量名和表达式的实际值是否在变量表中存在。如果不存在，那么就输出错误信息并退出程序。这是为了防止使用未定义的变量。
- 如果变量名和表达式的实际值在变量表中都存在，那么就将这个赋值表达式转换为四元式的形式，并添加到节点的四元式列表中。
- 然后，更新节点的实际值为变量名。这是为了在后续的语义分析中能够获取到这个变量的值。

```
void SemanticAnalyzer::handle_expression(SemanticTreeNode*& node)
{
    /*
    simple_expression
    postfix_expression

    var T_ASSIGN expression
    a = exp
    (=, t, _, a)
    */
    const auto& list = node->children;

    if (list.size() == 1) {
        return;
    }

    const std::string& var = list[0]->real_value;
    const std::string& op = list[1]->real_value;
    const std::string& exp = list[2]->real_value;
    if (list[0]->literal == "T_IDENTIFIER" && !exists_var_declaration(var))
{
        std::cout << "Error: 未定义变量: " << var << std::endl;
        exit(-1);
    }
    if (list[2]->literal == "T_IDENTIFIER" && !exists_var_declaration(exp))
{
```

```
            std::cout << "Error: 未定义变量：" << exp << std::endl;
            exit(-1);
        }
        node->add_quater(op, exp, "", var);
        node->real_value = var;
    }
```

4.调试分析

4.1正确数据测试输出

input_corr_1

输入：

```
int main(int argc, int argv[]) {
    int sum;
    int i;
    for(i = 0; i < 50; ++i) {
        if ((i % 2) == 0) {
            sum = sum + i;
        }
    }

    char c = 'a';
    int d = 5;
    return 0;
}
```

输出：

```
Accept
+---------------+------+---------------+
| Variable Name | Type | Initial Value |
+---------------+------+---------------+
| c             | char | 'a'           |
| d             | int  | 5             |
| i             | int  | NULL          |
| sum           | int  | NULL          |
+---------------+------+---------------+
+----+------------------------+
| ID | Quarter                |
+----+------------------------+
| 0  | (=   , 0   , _  , i  ) |
| 1  | (<   , i   , 50 , t0 ) |
| 2  | (jnz , t0  , _  , 4  ) |
| 3  | (j   , _   , _  , 12 ) |
| 4  | (%   , i   , 2  , t1 ) |
| 5  | (==  , t1  , 0  , t2 ) |
| 6  | (jnz , t2  , _  , 8  ) |
```

```
| 7  | (j   , _   , _  , 10 ) |
| 8  | (+   , sum , i  , t3 ) |
| 9  | (=   , t3  , _  , sum) |
| 10 | (+   , i   , 1  , i  ) |
| 11 | (j   , _   , _  , 1  ) |
+----+------------------------+
```

输出成功包含了代码的语法分析正确性、变量表和四元式形式的中间代码

4.2错误数据测试输出

input_err1(缺少分号)

输入：

```
int main() {
    int x = 1
    return 0;
}
```

输出：

```
D:\Lexer-main>.\output\test.exe .\test\input\input_err1.txt
test\grammer\grammer.txt
日志文件创建成功!
Parse error: no action
```

input_err2(缺少赋值项)

输入：

```
int main() {
    int x;
    x=;
    return 0;
}
```

输出：

```
D:\Lexer-main>.\output\test.exe .\test\input\input_err2.txt
test\grammer\grammer.txt
日志文件创建成功!
Parse error: no action
```

input_err3(分支语句缺少扩号)

输入：

```
int main() {
    int x=1;
    int y=2;
    if x > y {
        x = y;
    } else if x < y {
        x = y + 1;
    } else {
        x = 0;
    }
    return 0;
}
```

输出：

```
D:\Lexer-main>.\output\test.exe .\test\input\input_err3.txt
test\grammer\grammer.txt
日志文件创建成功!
Parse error: no action
```

input_err4(for循环缺语句)

输入：

```
int main() {
    int i=1;
    for (i = 0; i < 10;) {
        a[i] = i;
    }
    return 0;
}
```

输出：

```
D:\Lexer-main>.\output\test.exe .\test\input\input_err4.txt
test\grammer\grammer.txt
日志文件创建成功!
Parse error: no action
```

input_err5(括号不匹配)

输入：

```
int main() {
    int i=1;
    for (i = 0; i < 10;i++) {
        a[i] = i;

    return 0;
}
```

输出：

```
D:\Lexer-main>.\output\test.exe .\test\input\input_err5.txt
test\grammer\grammer.txt
日志文件创建成功!
Parse error: no action
```

input_err6(中括号不匹配)

输入：

```
int main() {
    int i=1;
    for (i = 0; i < 10;i++) {
        a[i = i;
    }
    return 0;
}
```

输出：

```
D:\Lexer-main>.\output\test.exe .\test\input\input_err6.txt
test\grammer\grammer.txt
日志文件创建成功!
Parse error: no action
```

input_err7(while多语句)

输入：

```
int main() {
    int i=1;
    while(i = 0; i < 10;i++) {
        a[i] = i;
    }
    return 0;
}
```

输出：

```
D:\Lexer-main>.\output\test.exe .\test\input\input_err7.txt
test\grammer\grammer.txt
日志文件创建成功!
Parse error: no action
```

input_err8(词法错误)

输入:

```
int main() {
    啊啊啊
    return 0;
}
eee
```

输出:

```
D:\Lexer-main>.\output\test.exe .\test\input\input_err8.txt
test\grammer\grammer.txt
日志文件创建成功!
Parse error: no action
```

input_err9(词法错误)

输入:

```
int main() {
    int x = eee;
    return 0;
}
eee
```

输出:

```
D:\Lexer-main>.\output\test.exe .\test\input\input_err9.txt
test\grammer\grammer.txt
日志文件创建成功!
Parse error: no action
```

input_err10(返回语句错)

输入:

```
int solve(int a, int b) {
    return a  b;
}
```

输出:

```
D:\Lexer-main>.\output\test.exe .\test\input\input_err10.txt
test\grammer\grammer.txt
日志文件创建成功!
Parse error: no action
```

input_err11(赋值错)

输入:

```
int main() {
    int x = float f = a;
    return 0;
}
```

输出:

```
D:\Lexer-main>.\output\test.exe .\test\input\input_err11.txt
test\grammer\grammer.txt
日志文件创建成功!
Parse error: no action
```

input_err12(小括号不匹配)

输入:

```
int main( {
    return 0;
}
```

输出:

```
D:\Lexer-main>.\output\test.exe .\test\input\input_err12.txt
test\grammer\grammer.txt
日志文件创建成功!
Parse error: no action
```

input_err13(变量重复定义)

输入:

```
int main(int argc, int argv[]) {
    int sum;
    int sum;
    int i;
    for(i = 0; i < 50; ++i) {
        if ((i % 2) == 0) {
            sum = sum + i;
        }
    }

    char c = 'a';
    int d = 5;
    return 0;
}
```

输出:

```
Accept
Error：重定义变量：sum
```

input_err14(变量未定义使用)

输入：

```
int main(int argc, int argv[]) {
    //int sum;
    int i;
    for(i = 0; i < 50; ++i) {
        if ((i % 2) == 0) {
            sum = sum + i;
        }
    }
    char c = 'a';
    int d = 5;
    return 0;
}
```

输出：

```
Parse error: no action
Variable table is empty.
```

5.总结与收获

编译原理课程作业报告：中间代码生成器的实现

在本学期的编译原理课程中，通过实现一个基于我之前编写的词法分析器和语法分析器的中间代码生成器，我们不仅锻炼了我们的编程技能，而且对编译原理的理解也有了质的飞跃。下面将总结我们的学习体会和实践收获。

1. 理论与实践的结合

在课堂上，我们学习了关于编译器的理论知识，如词法分析、语法分析、语义分析以及中间代码生成等。但只有当我们开始动手实现中间代码生成器时，我们才真正理解了这些概念。通过实践，我们深刻体会到了理论与实践相结合的重要性。实际编码过程中，我们不断回顾理论知识，确保我们的代码逻辑符合编译原理的基本要求。

2. 代码结构的重要性

在编写中间代码生成器的过程中，我们意识到了良好的代码结构的重要性。我们的代码包括了不同的模块，如打印中间代码、处理变量表、语义分析等。这种模块化的方法不仅使代码更加清晰，也使得调试和维护变得更加容易。此外，这也提高了代码的复用性，为未来可能的扩展提供了便利。

3. 调试和问题解决

在开发过程中，我们遇到了各种挑战，如调试复杂的逻辑错误和解决意想不到的问题。这些经历教会了我们如何有效地利用调试工具和技术，如设置断点、检查变量状态等。同时，我们也学会了如何在遇到问题时不慌张，学会了耐心地分析问题并找出解决方案。

4. 对编译原理的深入理解

通过这次作业，我们对编译原理中的许多概念有了更深入的理解。例如，我们更加清楚地了解了中间代码的作用和重要性。中间代码作为源代码和目标代码之间的桥梁，不仅简化了目标代码的生成，而且使得编译器能够更容易地移植到不同的平台。

5. 总结

总的来说，这次作业经历是极其宝贵的。我们不仅提升了编程能力，而且对编译原理有了更深入的认识。我们相信这些学习和实践经验将在我们的未来学术和职业生涯中发挥重要作用。5.总结与收获

6.分工

中间代码生成器分析部分：郑星云、张子菡

中间代码生成器交互部分：张子菡、陈凌锐

程序功能调试及报告撰写：陈凌锐、郑星云

7.参考文献

[1] 陈火旺等. 程序设计语言编译原理（第 3 版）. 国防工业出版社, 2000

[2] Maoyao233. ToyCC. https://github.com/Maoyao233/ToyCC

[3] GQT. 词法分析器. 2021

[4] LLVM. Clang C Language Family Frontend for LLVM. https://clang.llvm.org/

[5] jsoup. jsoup: Java HTML Parser. https://jsoup.org/

[6] Cplusplus.com. std::stringstream - sstream. https://cplusplus.com/reference/sstream/stringstream/

[7] Jutta Degener. ANSI C Yacc grammar. 2004

[8] Jutta Degener. ANSI C grammar, Lex specification. 2017

[9] ISO. ISO/IEC 9899:1999 Programming languages — C, 1999