# 编译原理语法分析器实验报告

小组成员：郑星云 张子菡 陈凌锐
完成时间：2023/11/14

# 1. 需求分析

## 1.1 程序功能

该程序是一个用C++编写的LR(1)语法分析器，能够解析指定的LR(1)文法并确定语句是否符合该文法。

其主要功能包括：

1. 解析特定格式的文法，生成对应的ACTION和GOTO表。
2. 对给定的语句进行类C语言词法分析，生成供语法分析使用的Token数组。
3. 判断该语句是否符合指定的文法。

我们自行定义了一个类C语言文法，包含常用的C语言语法，以该文法作为输入，即可解析常用的类C语言语句。

## 1.2 使用方式

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 `LR1Parser.exe`.

如需编译可使用如下命令：

- 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)

运行方法：

```
LR1Parser.exe <输入文件路径> <文法文件路径>
```

例如：

```
LR1Parser.exe ./test/input/input.txt ./test/grammer/grammer.txt
```

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

解析文法大约需要5s，请耐心等待……

程序需要两个文件路径作为参数：

- **输入文件路径：** 包含待分析源代码的文件路径。

- **文法文件路径：** 包含语法分析规则的文件路径。

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 $\varepsilon$。

以下是一个简单的文法例子：

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

在 `grammer.txt` 文法文件中被定义为：

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

## 1.3 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. **词法分析输出：**每个从词法分析器获取的 Token 的字符串表示形式。
2. **FIRST 集输出：**每个非终结符的 FIRST 集合。
3. **ACTION 表和 GOTO 表输出：**输出文法解析器构建的 ACTION 表和 GOTO 表。
4. **语法分析输出：**对输入句子进行语法分析的过程和结果，包括每一步状态栈、符号栈的输入栈的状态。

以一个简单文法为例，输出如下：

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

- **FIRST 集**

  ```
  FIRST(b) = { b }
  FIRST(a) = { a }
  FIRST(B) = { b a }
  FIRST(S) = { a b }
  FIRST(S') = { b a }
  ```

- **ACTION 表和 GOTO 表**

  其中S表示移，紧随新增的状态；R表示归约，紧随归约所用的产生式。

  ```
  ACTION Table:
  ```

```
State 0, Symbol a: S 2
State 0, Symbol b: S 1
State 1, Symbol a: R [B -> b]
State 1, Symbol b: R [B -> b]
State 2, Symbol a: S 2
State 2, Symbol b: S 1
State 3, Symbol a: S 7
State 3, Symbol b: S 6
State 4, Symbol $: Accept
State 5, Symbol a: R [B -> aB]
State 5, Symbol b: R [B -> aB]
State 6, Symbol $: R [B -> b]
State 7, Symbol a: S 7
State 7, Symbol b: S 6
State 8, Symbol $: R [S -> BB]
State 9, Symbol $: R [B -> aB]

GOTO Table:
State 0, Symbol B: 3
State 0, Symbol S: 4
State 2, Symbol B: 5
State 3, Symbol B: 8
State 7, Symbol B: 9
```

- **语法分析** (句子：$\mathrm{abab\$}$)

```
State Stack: 0
Symbol Stack:
Input Stack: a b a b $

State Stack: 0 2
Symbol Stack: a
Input Stack: b a b $

State Stack: 0 2 1
Symbol Stack: a b
Input Stack: a b $

State Stack: 0 2 5
Symbol Stack: a B
Input Stack: a b $

State Stack: 0 3
Symbol Stack: B
Input Stack: a b $

State Stack: 0 3 7
Symbol Stack: B a
Input Stack: b $

State Stack: 0 3 7 6
Symbol Stack: B a b
Input Stack: $

State Stack: 0 3 7 9
Symbol Stack: B a B
```

```
Input Stack: $

State Stack: 0 3 8
Symbol Stack: B B
Input Stack: $

State Stack: 0 4
Symbol Stack: S
Input Stack: $
```

## 1.3 支持的语言

词法分析支持的符号包括：

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。

语法分析支持的语句包括：

- 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
```

# 2.概要设计

## 2.1 任务分解

整体上分为词法分析器Lexer和语法分析器LR1Parser，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

## 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 $\mathrm{var}$ 被定义为 `Symbol(Terminal, "var")`

特别的，我们规定 $\varepsilon$ 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
};
```

## 2.3 主程序流程

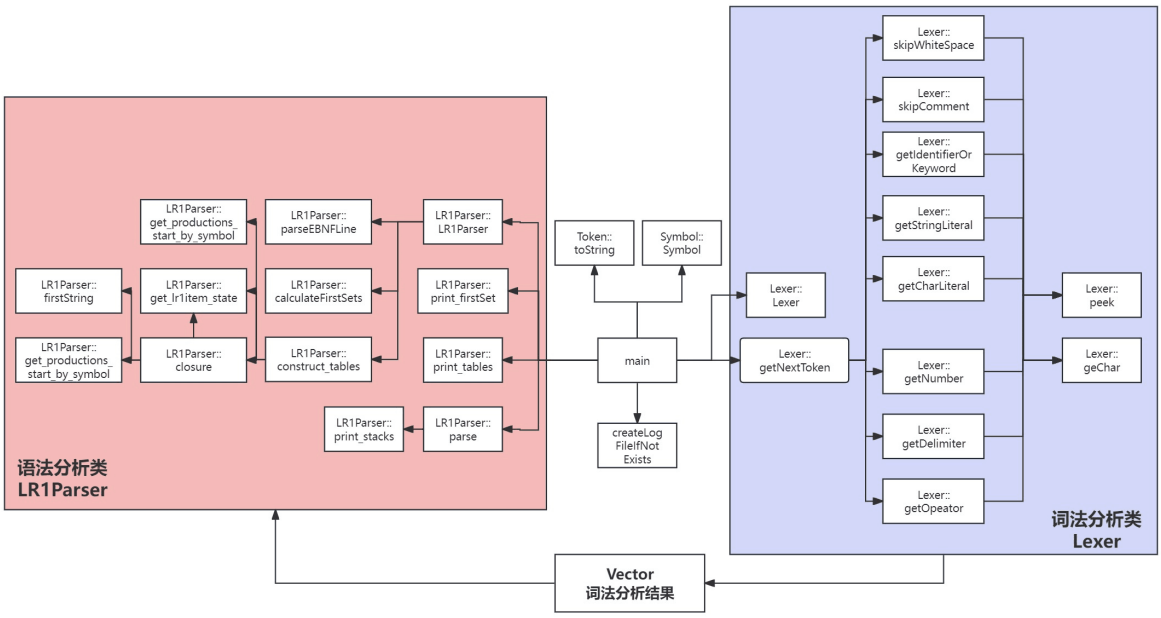
主程序流程如下：

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。
   - 输出每个 Token 的字符串表示形式。
6. **构建文法解析器（Parser）：**
   - 使用 `LR1Parser` 类，传入文法文件路径，构建 LR(1) 文法解析器。
7. **输出 FIRST 集合：**
   - 调用 `print_firstSet` 函数，输出文法的 FIRST 集合。
8. **输出 ACTION 表和 GOTO 表：**
   - 调用 `print_tables` 函数，输出 LR(1) 分析表的 ACTION 表和 GOTO 表。
9. **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。

## 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

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();
};
```

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。

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。

公共成员函数：

**Token提取函数：**

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

各函数详细说明：

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`。

    ```C++
    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

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;
};
```

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。

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。

## 各函数详细说明：

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;
}
```

# 4.调试分析

## 4.1正确数据测试输出

## input_corr_1

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

void emptyFunction() {
}

int main() {
    int x = 1;
    float f = 0.2;
    char a[10];
    char b = 'b';
    int ci = 10;
    char str[256];
    str = "Hello World";

    x++;
    ++x;


    if (x > y) {
        x = y;
    } else if (x < y) {
        x = y + 1;
    } else {
        x = 0;
    }

    while (x > 0) {
```

```
        x--;
    }

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

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


    x = !x;
    x = x % 5;
    x = x & 1;
    x = x | 1;
    x = x ^ 1;
    x = ~x;

    int result = solve(x, y);

    {
        int nestedVar = 5;
        nestedVar = nestedVar + 2;
    }

    return 0;
}
```

词法分析结果

```
--------------------词法分析--------------------
T_INT
T_IDENTIFIER
T_LEFT_PAREN
T_INT
T_IDENTIFIER
T_COMMA
T_INT
省略......
--------------------词法分析--------------------
```

FIRST集结果

每个语法符号显示所有可能的起始终结符，格式为："FIRST(符号) = { 终结符列表 }"。

```
--------------------FIRST集--------------------
FIRST(T_ASSIGN) = { T_ASSIGN }
FIRST(T_BITAND) = { T_BITAND }
FIRST(T_BITNOT) = { T_BITNOT }
FIRST(T_BITOR) = { T_BITOR }
FIRST(T_BITXOR) = { T_BITXOR }
FIRST(T_CHAR) = { T_CHAR }
FIRST(T_CHAR_LITERAL) = { T_CHAR_LITERAL }
省略......
--------------------FIRST集--------------------
```

ACTION表和GOTO表

```
---------------ACTION表和GOTO表-----------------
ACTION Table:
State 0, Symbol T_CHAR: S 1
State 0, Symbol T_FLOAT: S 2
State 0, Symbol T_INT: S 3
State 0, Symbol T_VOID: S 4
State 1, Symbol T_IDENTIFIER: R [simple_type -> T_CHAR]
State 2, Symbol T_IDENTIFIER: R [simple_type -> T_FLOAT]
State 3, Symbol T_IDENTIFIER: R [simple_type -> T_INT]
State 4, Symbol T_IDENTIFIER: R [simple_type -> T_VOID]
State 5, Symbol T_EOF: Accept
省略......

GOTO Table:
State 0, Symbol T_CONST: 13
State 0, Symbol T_DOUBLE: 11
State 0, Symbol T_LONG: 12
State 0, Symbol declaration: 8
State 0, Symbol declaration_list: 10
State 0, Symbol fun_declaration: 6
State 0, Symbol program: 5
State 0, Symbol simple_type: 14
省略......
---------------ACTION表和GOTO表-----------------
```

语法分析结果

```
--------------------语法分析--------------------
State Stack: 0
Symbol Stack:
Input Stack: T_INT T_IDENTIFIER T_LEFT_PAREN T_INT T_IDENTIFIER T_COMMA
T_INT.......

State Stack: 0 3
Symbol Stack: T_INT
Input Stack: T_IDENTIFIER T_LEFT_PAREN ......

State Stack: 0 14
Symbol Stack: simple_type
Input Stack: T_IDENTIFIER T_LEFT_PAREN ......
```

```
State Stack: 0 9
Symbol Stack: type_specifier
Input Stack: T_IDENTIFIER T_LEFT_PAREN ......

......

State Stack: 0 8 8 15
Symbol Stack: declaration declaration declaration_list
Input Stack: T_EOF

State Stack: 0 8 15
Symbol Stack: declaration declaration_list
Input Stack: T_EOF

State Stack: 0 10
Symbol Stack: declaration_list
Input Stack: T_EOF

State Stack: 0 5
Symbol Stack: program
Input Stack: T_EOF

Accept
--------------------语法分析--------------------
```

## 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;
}
```

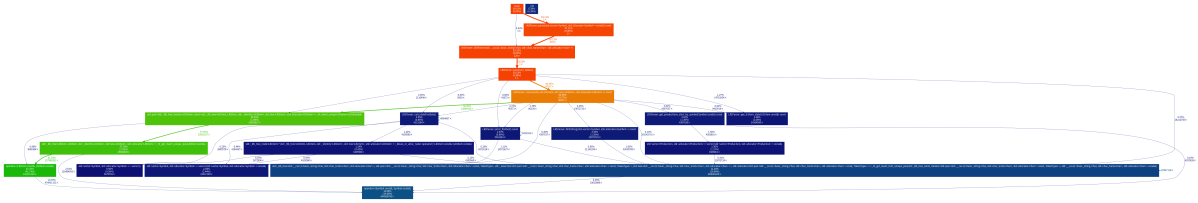
输出：

```
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
```

## 4.3 性能分析与遇到的问题

- **闭包**

使用 Linux 中的 gprof 工具进行性能分析。[分析结果高清图预览]



可以发现，程序最耗时的部分在于求闭包函数。求闭包函数耗时在于，频繁使用了 `std::set` 的插入，`std::set` 的 `insert` 方法包含去重算法，大量的无意义插入成为了性能瓶颈。

优化方案：

- 用一个布尔值 `changed` 来检查 `old_set` 是否有变化，以避免不必要的拷贝。
- 创建了一个 `items_to_add` 集合来存储本次循环中需要添加的元素，从而减少对 `old_set` 的直接修改。
- 在插入之前检查 `old_set` 是否已经包含了该元素，以减少不必要的插入操作。

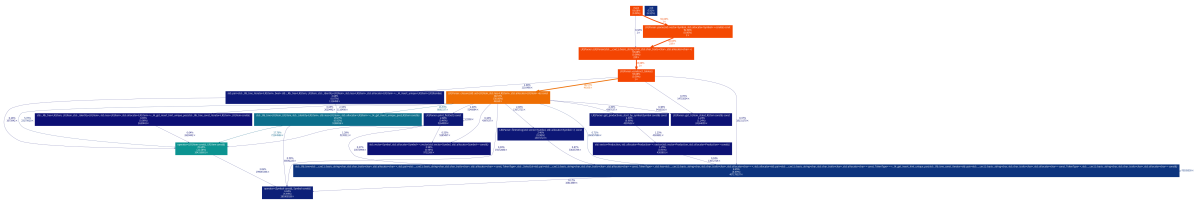
```
void LR1Parser::closure(std::set<LR1Item> &old_set) const {
    bool changed = true;
    while (changed) {
        changed = false;
        std::set<LR1Item> items_to_add;

        for (const auto& item : old_set) {
            // 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;

            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 (const auto& prod : next_prods) {
                std::set<Symbol> lookaheads = firstString({nnext_symbol,
item.lookahead});
                for (const auto& p_lh : lookaheads) {
                    LR1Item new_item(prod, 0, p_lh);
                    if (old_set.find(new_item) == old_set.end()) {
                        items_to_add.insert(new_item);
                        changed = true;
                    }
                }
            }
        }

        old_set.insert(items_to_add.begin(), items_to_add.end());
    }
    return old_set;
}
```

优化后的性能分析图如下：分析结果高清图预览



可以看到非常有效的减小了插入占用 `std::set` 的时间，减少比较次数50%。

可能的优化方案：

- **使用 `std::unordered_set` 替换 `std::set`**： `std::unordered_set` 基于哈希表实现，通常在插入和查找操作上比基于红黑树的 `std::set` 更高效。
- **FIRST集**

一开始，采用了递归的方式求FIRST集，这导致在一些左递归文法中产生死循环。

解决方案：将代码修改为不断循环遍历每一个产生式，不断修改FIRST集和，直到一次遍历后FIRST集和不在有变化。这解决了左递归文法的死循环问题。

# 5.总结与收获

通过完成编译原理语法分析器大作业时，我们有以下感想和收获：

1. **深入理解课程知识：** 在完成词法分析器的过程中，我们不仅复习了词法分析的基本方法和流程，而且通过实践加深了对自动机理论的理解。在实现LR1语法分析器的过程中，我们不仅深入学习了自下向上的语法分析方法，更重要的是，我们理解了LR(0)项目集、LR(1)项集的构建以及ACTION表和GOTO表的构造这些概念的实际意义和应用场景。通过实际编写语法分析总控程序，我们更加深入地理解了编译器的工作原理以及语法分析在其中的作用。
2. **积累实践经验：** 我们小组从零开始设计和实现了一个功能完整的语法分析器。这个过程包括了符号的定义和表示、LR项的处理、分析表的构建以及整个项目的模块化设计。我们不仅学习了如何编写清晰、模块化的代码，还练习了高效的算法设计和实现。在项目开发过程中，我们特别注重代码的可读性和维护性，例如规范命名、添加充分的注释，这些都极大地提升了代码质量。同时，我们考虑到未来可能的需求变化，设计了易于扩展的架构，例如本次语法分析器能够无缝地整合词法分析器的输出结果。
3. **提高了问题解决能力：** 在项目开发过程中，我们面对了多种挑战，如调试复杂的语法错误和优化算法性能。通过使用调试工具和逐步的排查方法，我们学会了如何有效地定位和解决问题。在构建分析表时遇到的性能瓶颈也迫使我们学习和应用更高效的算法和数据结构。这些经历极大地锻炼了我们的逻辑思维能力和技术解决问题的能力。
4. **提高了合作能力：** 在这个项目中，我们通过分工合作、沟通协调和任务分配有效地推进了项目进程。每个组员都有自己的优势和专长，通过互相学习和合作，我们不仅共同完成了任务，还实现了技能上的互补和个人能力的提升。在合作过程中，我们遇到了多种挑战和分歧，但通过共同讨论和沟通，我们学会了如何解决问题和消除分歧。这个过程不仅增强了我们的团队合作意识，还提升了我们的沟通技巧和协作能力。
5. **学习性能分析：** 在项目的最后，使用gprof工具对代码进行了性能分析，修改了closure函数的实现，大大提升了代码的性能。

这个项目是一个全面的实践，从中我们发现编译原理不仅仅是理论的堆积，更是对计算机科学知识的综合应用。这种实践有助于培养我们在未来面对新问题时的自学和解决问题的能力。