同济大学
TONGJI UNIVERSITY

同济大学编译原理课程
词法分析暨LR(1)语法分析实验报告

组　　长　**2154312　郑博远**
组　　员　**2152496　郭桉齐**
组　　员　**2154318　蔡一锴**
专　　业　计算机科学与技术
授课老师　丁志军

1. 实验概述

1.1. 目的与意义

深入研究、掌握并灵活运用编译原理的相关知识，全面完成一个类似于 C 语言的编译器的开发。目前的任务是基于先前学过的词法分析和 LR（1） 语法分析，通过仔细设计、编写和调试一个标志性的 LR（1）语法分析界面，以更深层次地掌握语法分析的方法。

1.2. 主要任务

基本功能：对不含过程调用的类 C 示例程序实现词法和语法分析，并输出分析结果；

扩展功能：不限。

1.3. 需求分析

1.3.1. 程序输入

- 选择一个类 C 的源程序文件输入；
- 进行词法分析，输出词法分析的结果；
- 从文件读入文法基本信息，即文法的产生式集合等（文法的起始符号由产生式集合中第一个产生式的左边的文法符号），产生 LR（1）分析表 action 表和 goto 表；
- 根据之前词法分析的结果，测试该源程序文件是不是一个符合既定文法的源程序。若符合既定文法则显示规约移进分析过程和最终的语法分析树，若不符合既定文法则给出错误提示。

程序输入范例如下：

```
/*
   这是一个简单的乘法函数，接受一个整数和一个浮点数，
   返回它们的乘积。
*/
float multiplyNumbers(int a, float b) {
    float result = 0.0;  // 初始化结果为 0.0

    // 使用 while 循环进行乘法运算
    while (a > 0) {
        result += b;     // 将浮点数累加到结果中
        a -= 1;          // 递减整数 a，控制循环次数
    }

    return result;  // 返回最终乘积
}
```

1.3.2. 文法输入

文法输入如下：

```
Program -> DeclarationString

DeclarationString -> Declaration Declarations | Declaration

Declarations -> Declaration Declarations | Declaration

Declaration ->  Type identifier DeclarationType | void
identifier FunctionDeclaration

DeclarationType -> VarDeclaration | FunctionDeclaration

VarDeclaration -> ; | = Expression ;

FunctionDeclaration -> ( Parameters ) Block | ( ) Block |
( void ) Block

Type -> int | float

Parameters -> ParamVar ParamVars | ParamVar

ParamVars -> , ParamVar ParamVars | , ParamVar

ParamVar -> int identifier | float identifier

Block -> { InnerDeclaration StatementString } |
{ StatementString } | { InnerDeclaration }

InnerDeclaration -> InnerVarDeclaration ; InnerVarDeclarations |
InnerVarDeclaration ;
```

```
InnerVarDeclarations -> InnerVarDeclaration ;
InnerVarDeclarations | InnerVarDeclaration ;

InnerVarDeclaration -> Type identifier | Type identifier =
Expression

StatementString -> Statement Statements | Statement

Statements -> Statement Statements | Statement

Statement -> IfStatement | WhileStatement | ReturnStatement |
AssignStatement | FunctionCallStatement

IfStatement -> if ( Expression ) Block ElseStatement | if
( Expression ) Block

ElseStatement -> else Block

WhileStatement -> while ( Expression ) Block

ReturnStatement -> return Expression ; | return ;

AssignStatement -> identifier = Expression ; | identifier +=
Expression ; | identifier -= Expression ; | identifier *=
Expression ; |

FunctionCallStatement -> FunctionCall ;

identifier /= Expression ; | identifier %= Expression ; |
identifier >>= Expression ; | identifier <<= Expression ;

Expression -> Add Relops | Add

Relops -> Relop Add Relops | Relop Add

Relop -> < | <= | > | >= | == | !=

Add -> Item Items | Item

Items -> + Item Items | + Item | - Item Items | - Item | >> Item
Items | >> Item | << Item Items | << Item

Item -> Factor Factors | Factor

Factors -> * Factor Factors | * Factor | / Factor Factors | /
Factor | % Factor Factors | % Factor

Factor -> identifier | integer_constant |
floating_point_constant | FunctionCall | ( Expression )

FunctionCall -> identifier ( Arguments ) | identifier ( )

Arguments -> Expression Expressions | Expression

Expressions -> , Expression Expressions | , Expression
```

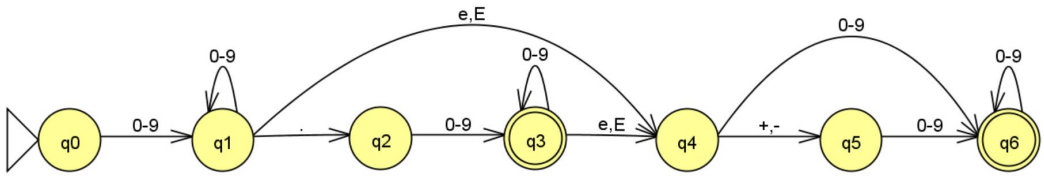
1.4. 拓展功能

1.4.1. 浮点数常量

本次实验中扩展增加了对浮点数常量的识别。支持的浮点数格式包括：

- **普通形式：**如 1.25, 34.2
- **科学计数法形式：** 如 3e4, 5.4e-9, 8E12

如下图所示的状态转移图，构造 DFA 以识别浮点数常量：



1.4.1. float 关键词

在给定识别单词的基础上新增关键字 float，与新增的浮点数变量联动。

1.4.2. 过程调用

通过增加部分文法，支持如下格式的过程调用：

```
int a;
int b;
int program(int a, float b, int c) {
  int i;
  int j;
  i = 0;
  if (a > (b + c)) {
    j = a + (b * c + 1);
  } else {
    j = a;
  }
  while (i <= 100) {
```

```
    i = j * 2;
  }
  return i;
}

int demo(int a) {
  a = a + 2;
  return a * 2;
}

void main(void) {
  int a;
  float b;
  int c;
  a = 3;
  b = 4e5;
  c = 2;
  a = program(a, b, demo(c));
  return;
}
```

2. 使用说明

2.1. 环境配置说明

以下脚本需要在代码目录的命令行中执行。

需要先行安装 pnpm, Python 及 pip。建议使用 virtual env。

2.1.1. 安装依赖

```
pnpm install
pip install -r requirements.txt
```

2.1.2. 运行

```
pnpm dev
```

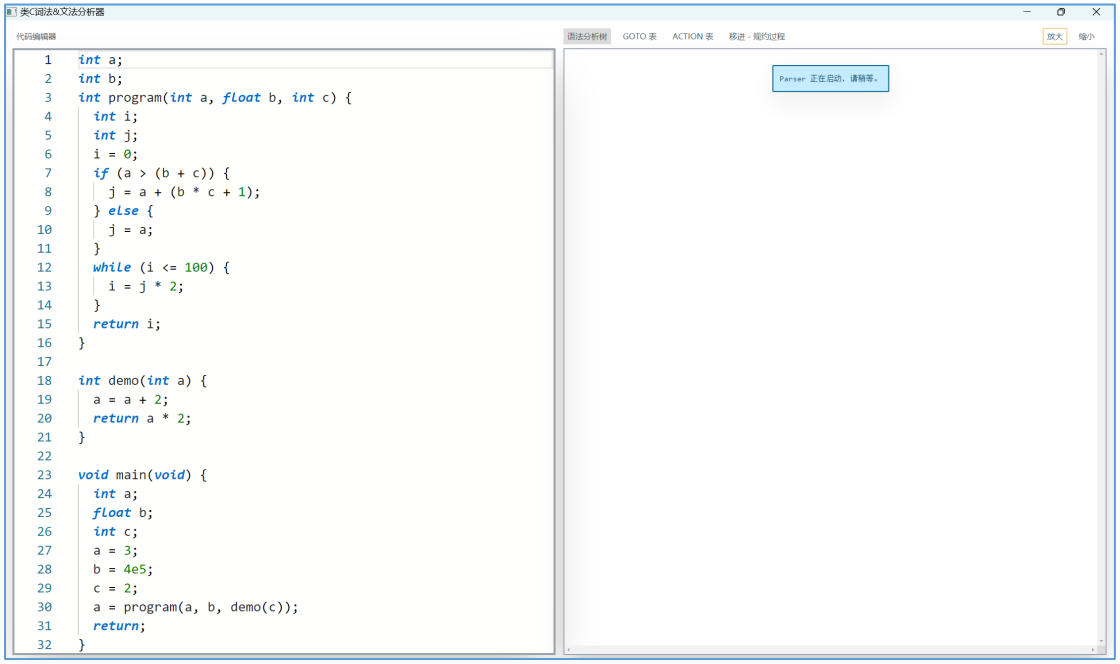
2.1.3. 打包

```
pnpm build
```

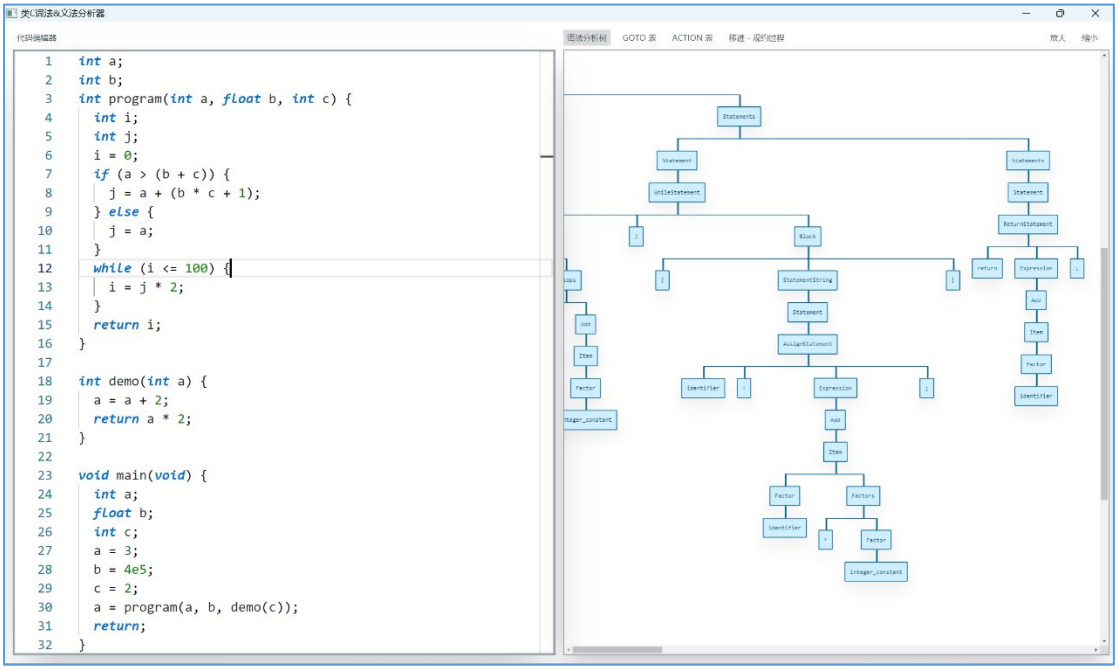
2.2. 整体设计

2.2.1. 初始界面

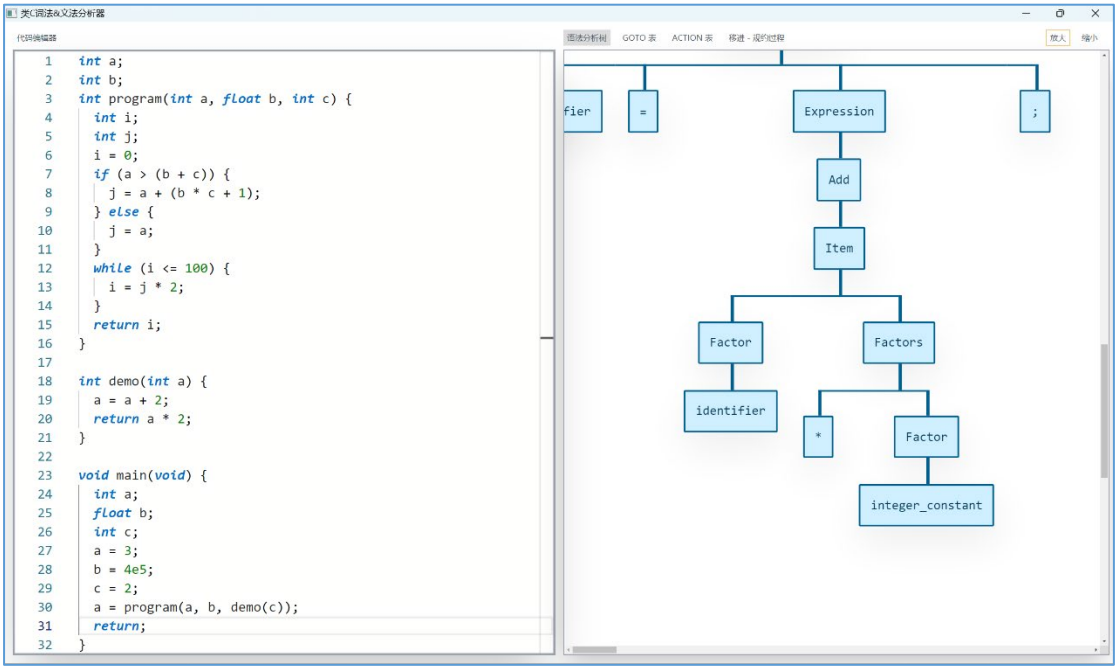
初始界面展示，左侧的代码是从与 main.py 同目录下的 test.c 文件中读取的，我们的分析器支持读取文件和左侧编辑器输入，可以看到左侧的代码是带有语法高亮功能的，右侧的内容需要一定时间来初始化（因此初始情况下为“parser 正在启动，请稍等”，如下图所示），包括从产生式配置文件（production.cfg）中读取产生式，并以此为依据构建闭包，进而构建 goto 表和 action 表。



等待初始化完成后，可以看到右侧的语法分析树，在左侧词法语法都正确的情况下可以成功生成，如下图所示。



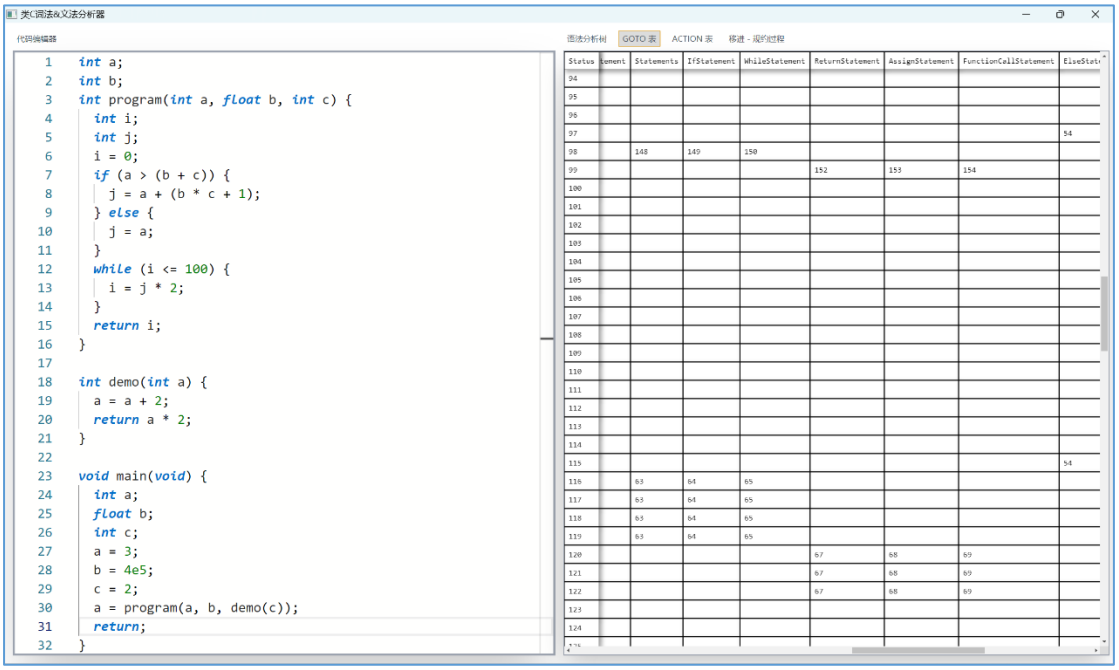
右侧的语法分析树可以与左侧编辑器一起通过 CTRL+鼠标滚轮的方式进行缩放，除此之外，语法分析树还可以通过右上角的放大缩小按钮进行缩放，下图所示的就是其放大后的样式。



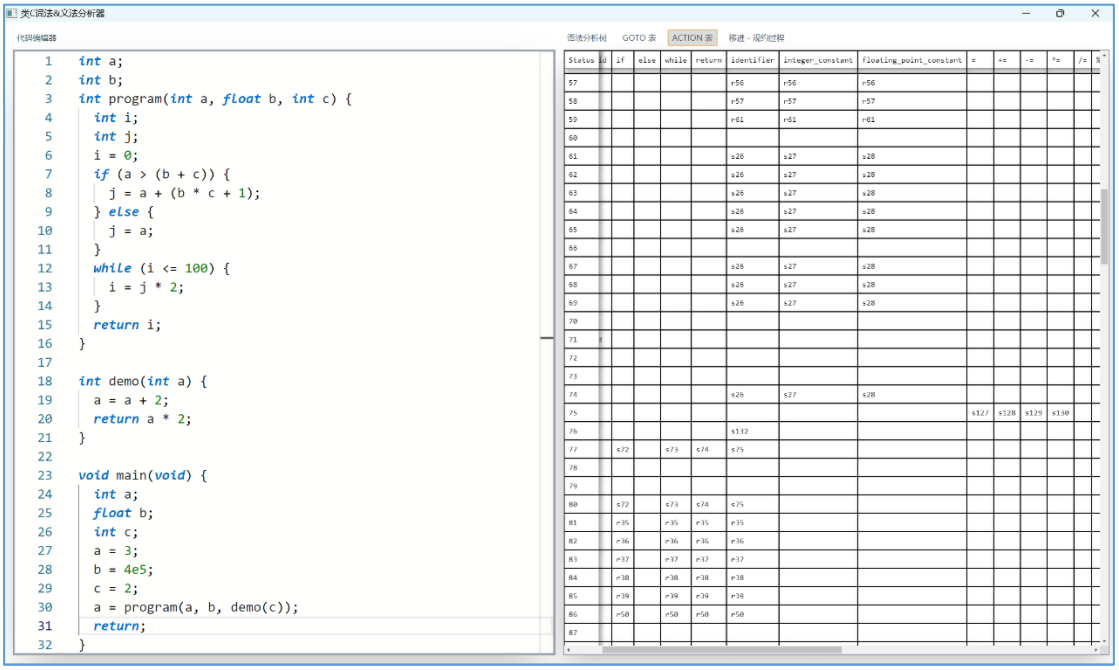
2.2.2. 分析表展示

除了语法分析树外，我们还对 GOTO 表、ACTION 表和移进规约过程表进行了展示。首先是 GOTO 表，其在整个程序运行中不变，仅有初始化时的产生

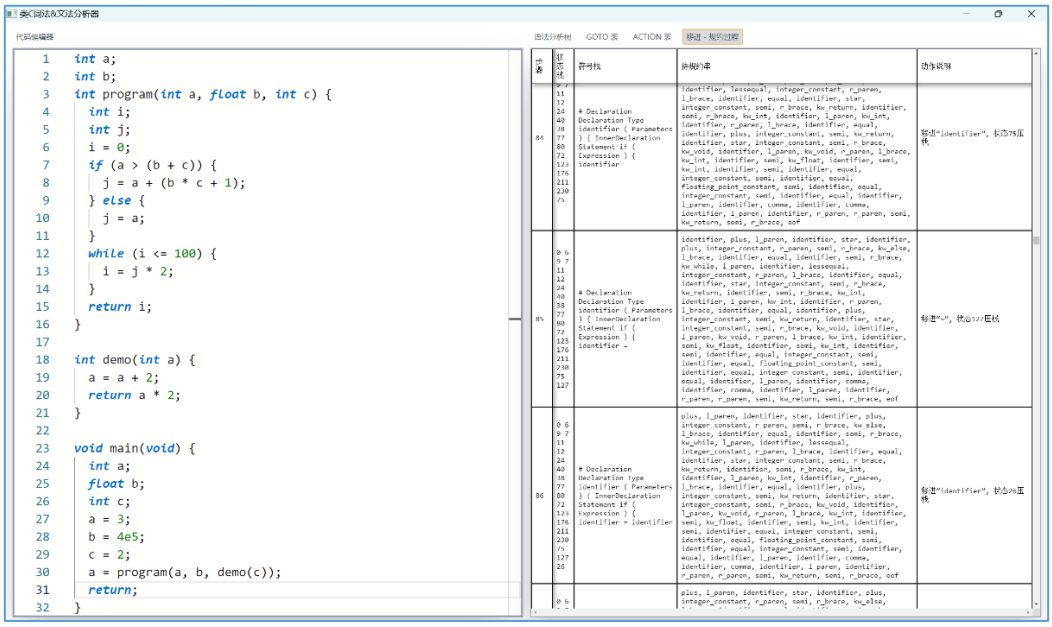
式文件来决定，由于表格过大，因此需要拖拽右侧和下方的滚轮进行查看，在拖拽过程中，最左侧和最上方的行列名称位置固定，可以更好地观察结果。



ACTION 表的整体情况与 GOTO 表类似，实例如下图所示：



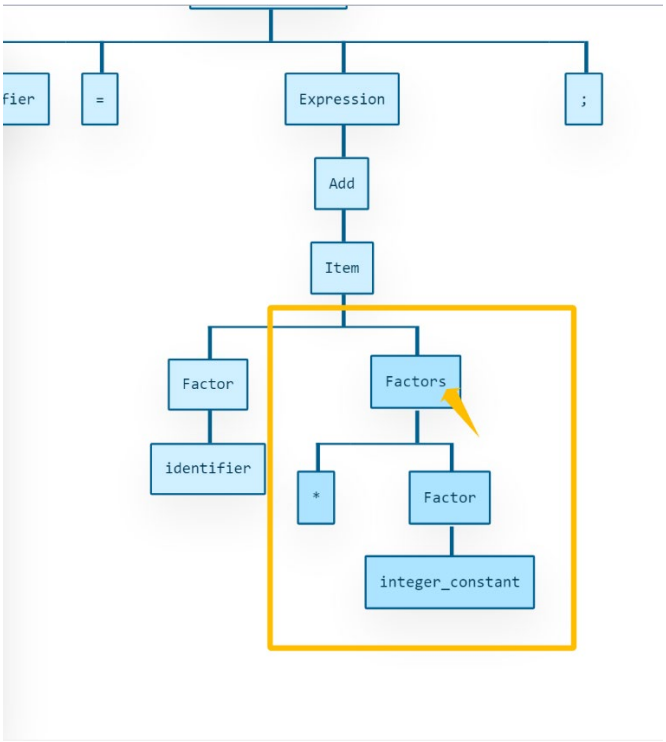
移进规约过程与左侧代码框内内容有关。值得一提的是，我们的语法分析器做到了交互的实时性，即在左侧更新内容的同时，右侧的语法分析树和移进规约过程表也同步更新。



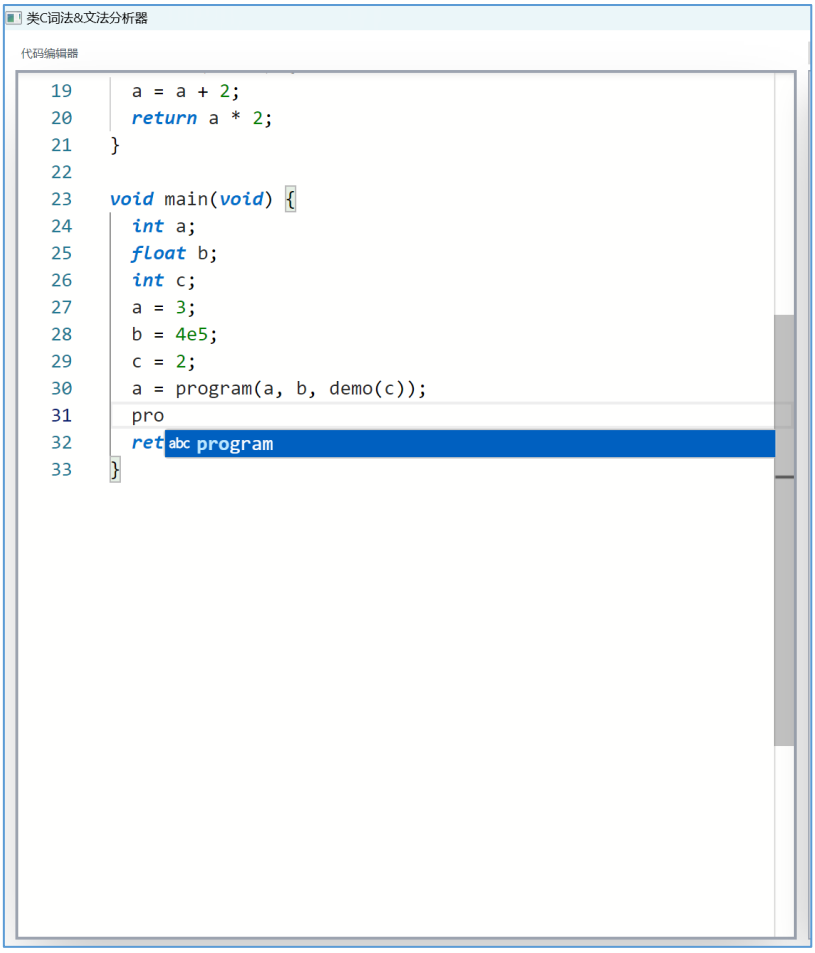
2.2.3. 扩展功能

以上的三张表和语法分析树是本项目的基本功能，为了方便用户使用，我们还增加了一些扩展功能。

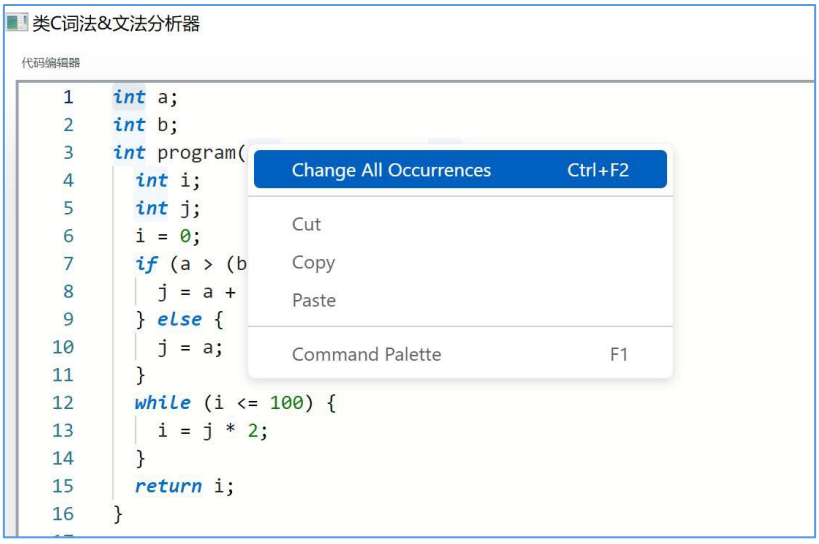
下图是鼠标在语法分析树上悬停功能的展示，当鼠标悬停在某叶结点时，以其为根结点的整个字数都获得相同的 hover 效果。



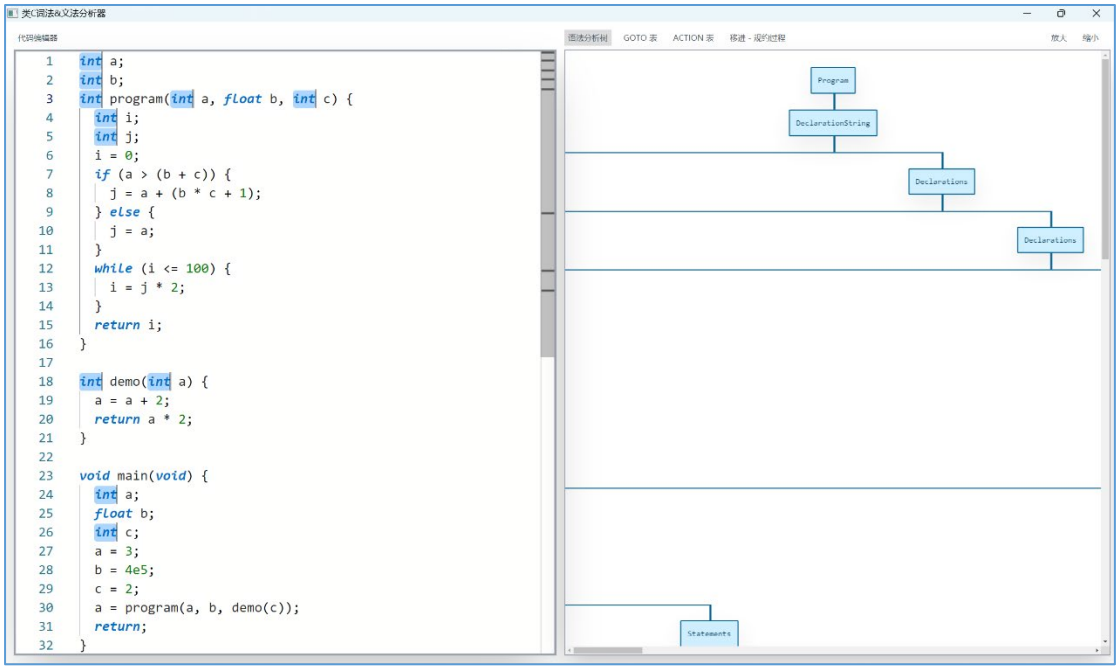
下图是输入联想功能的展示，上文已经定义了函数 program，因此在打出 pro 时输入联想为 program。



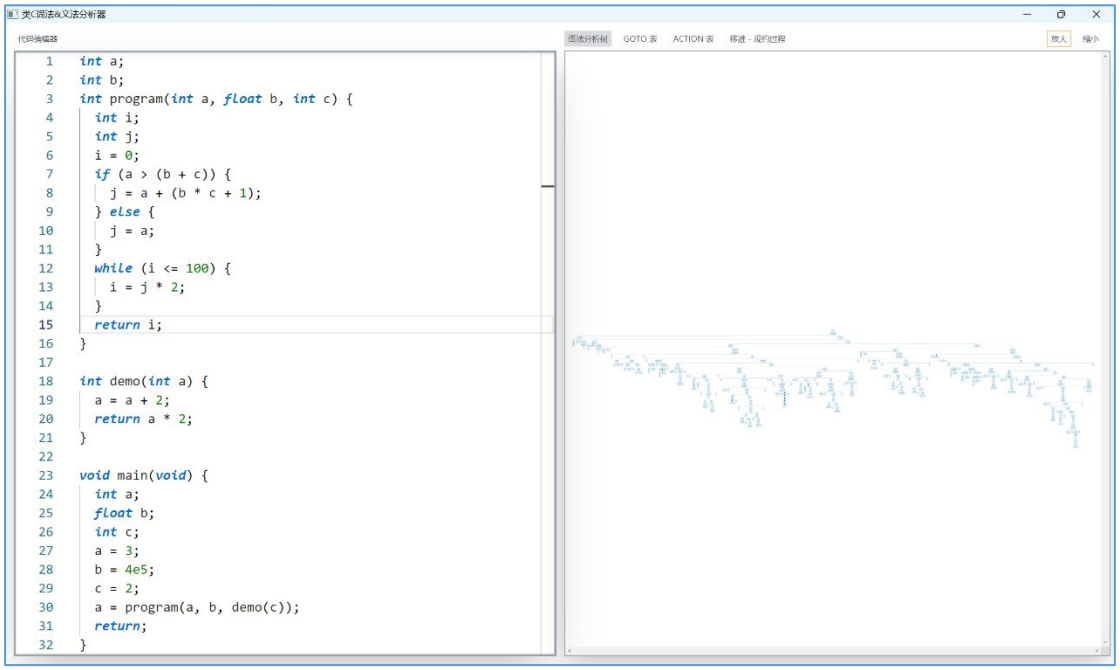
下图是批量修改功能的展示，通过右键选中的部分，点击 change all occurrence 可以进行批量修改。



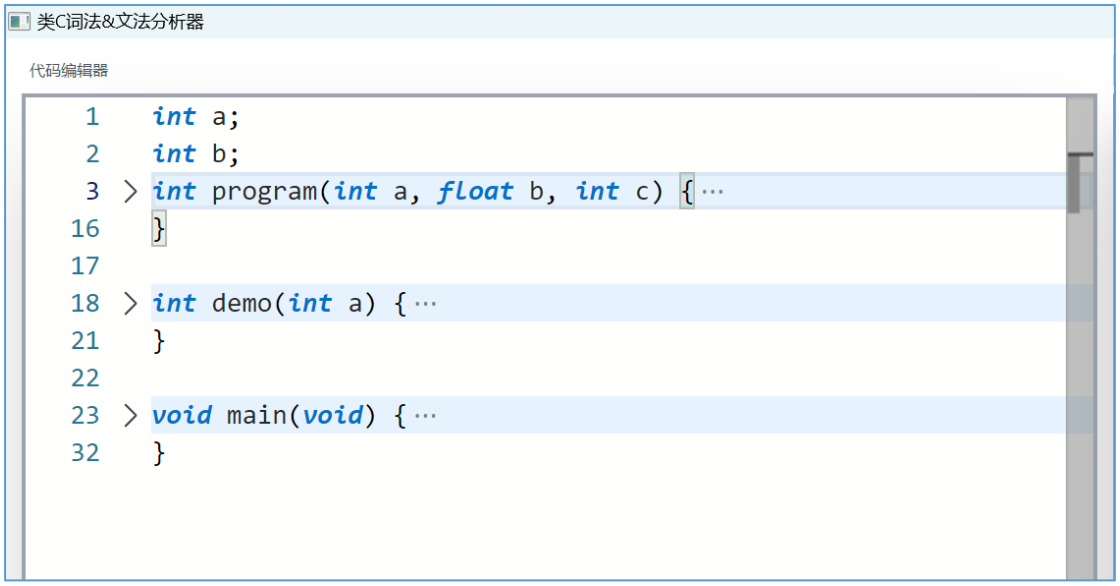
修改效果如下图所示，所有的 int 都被选中，可以按需修改。



下图是语法分析树单独缩放功能的展示，上文我们展示的是将其放大，下图展示的是其缩小功能。



下图为代码块收缩功能的展示：

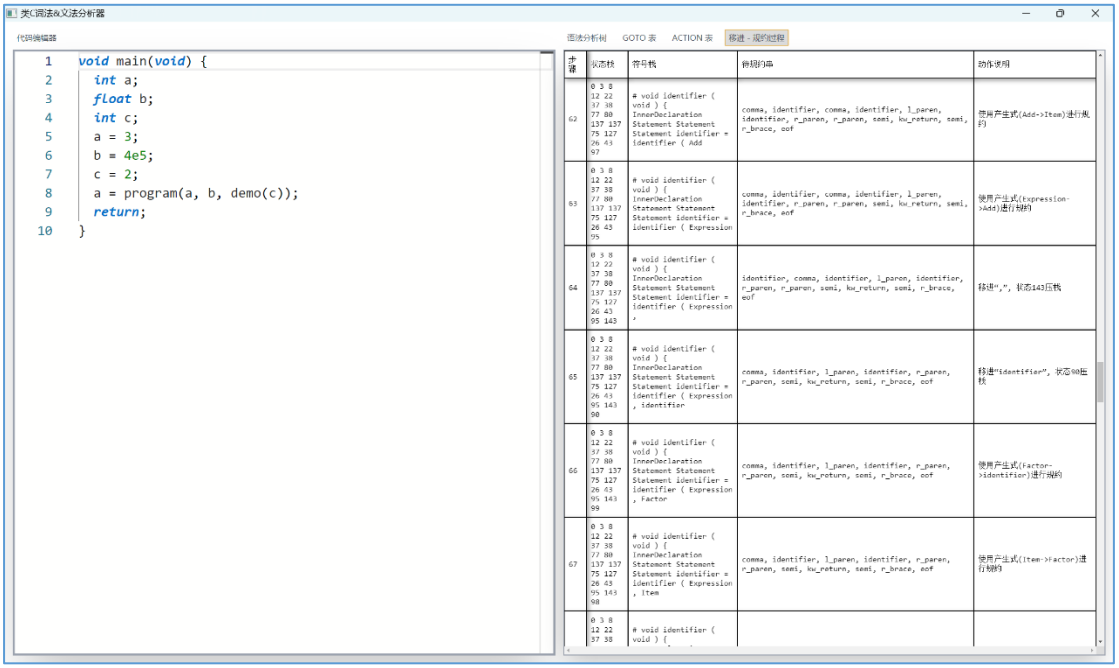


2.3. 运行实例

2.3.1. 测试样例 1

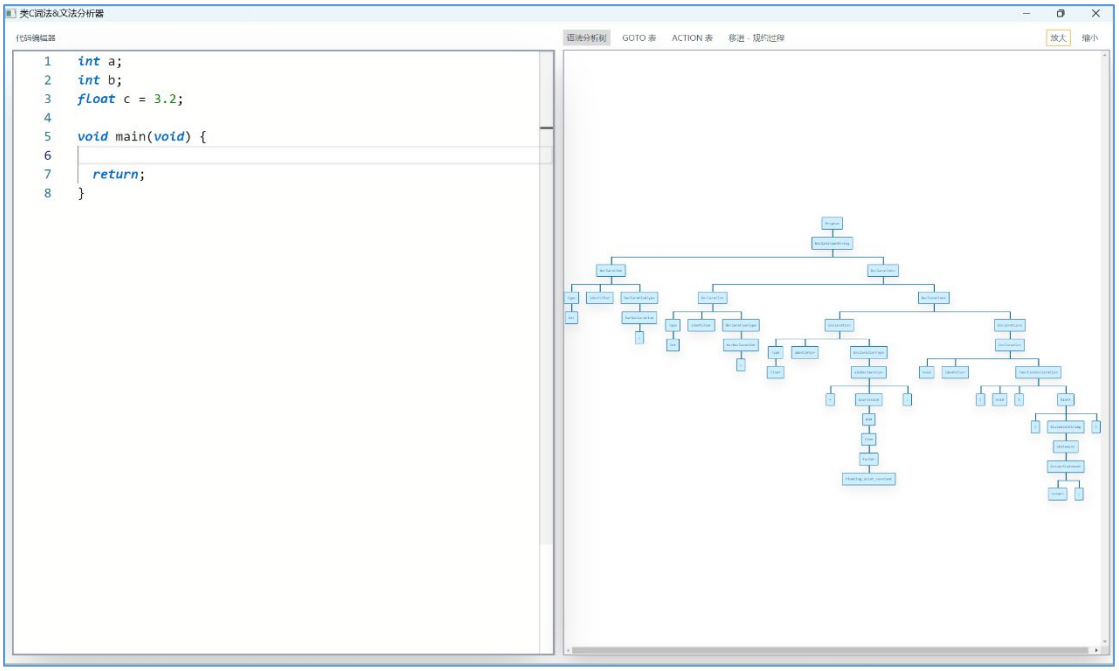
在本测试样例中，我们测试了函数声明（参数为 void）、整数和浮点数定义语句、赋值语句、科学计数法、过程调用和 return 语句，成功画出语法树，说明语法分析成功。

其对应的移进规约过程表如下图所示：

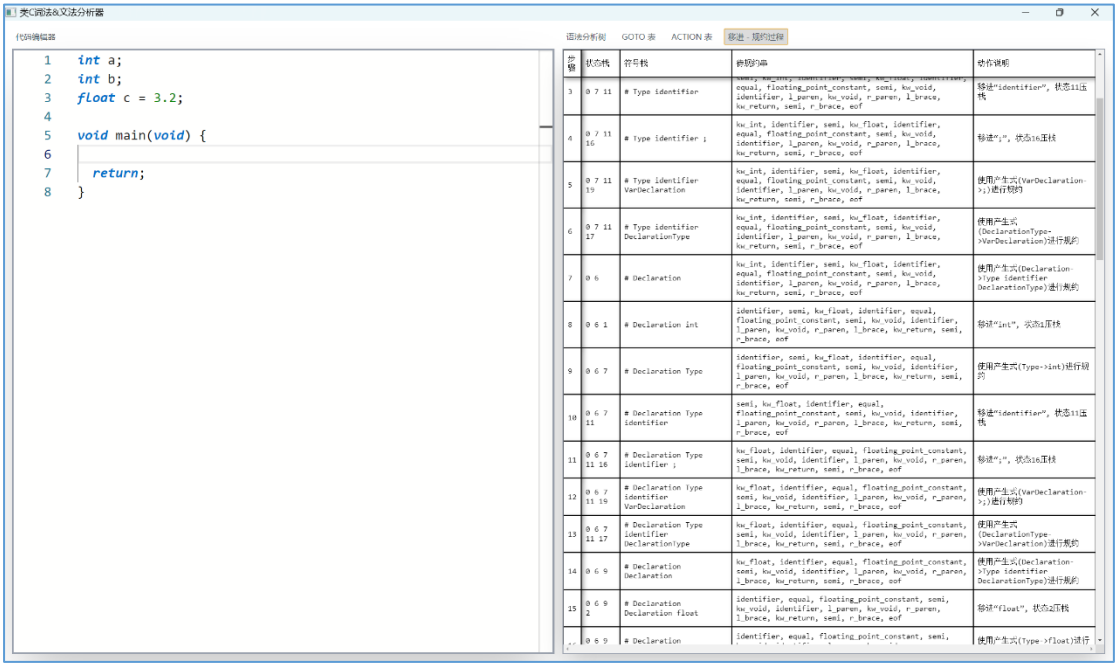


2.3.2. 测试样例 2

在本测试样例中，我们测试了全局变量声明和函数声明混合的情况，成功画出语法树，说明语法分析成功。



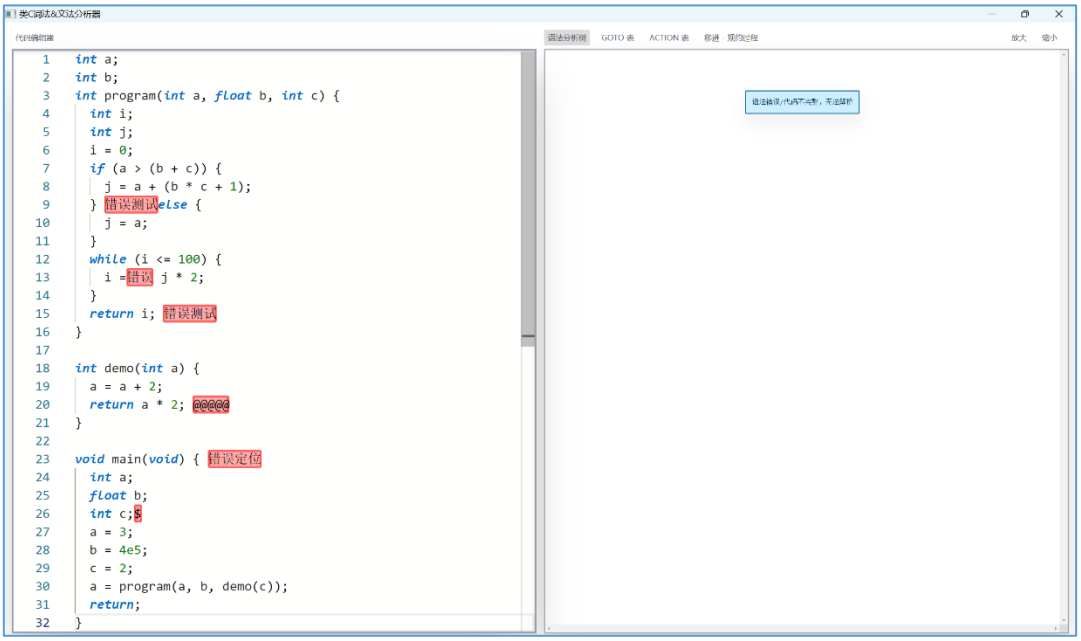
其对应的移进规约过程表如下图所示：



2.4. 错误分析

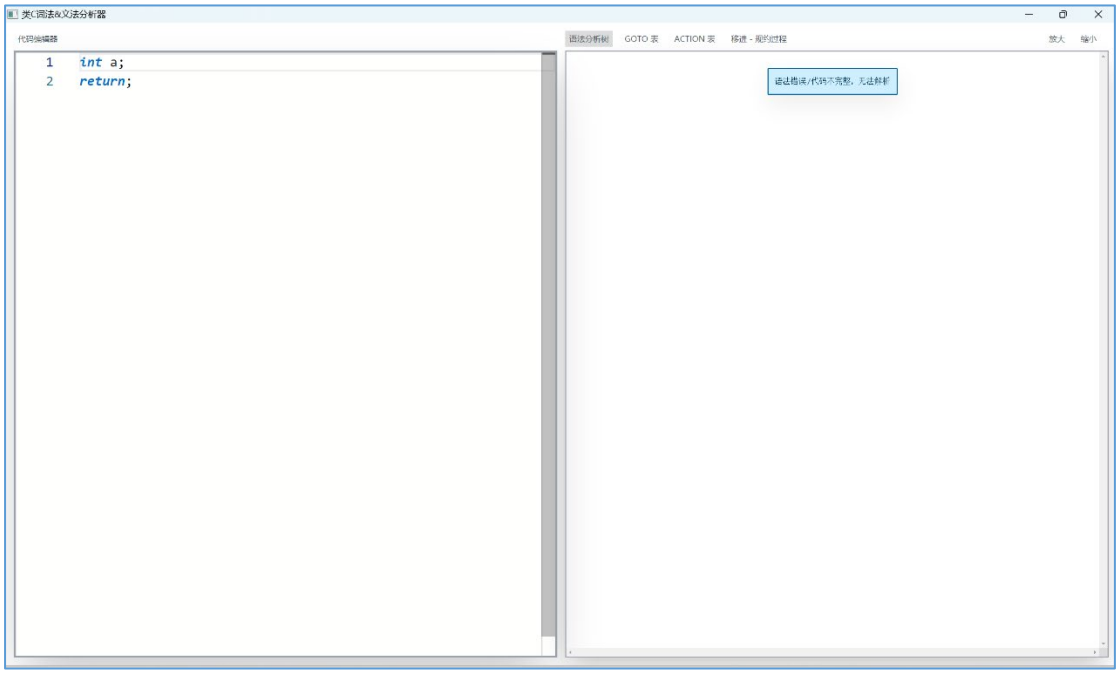
2.4.1. 词法分析错误

测试词法错误的情况，在每行的中段和末尾位置添加一些未定义的符号（如中文字符），可以发现该非法符号位置被标红，同时右侧的语法分析树显示“语法错误/代码不完整，无法解析”。



2.4.2. 语法分析错误

测试语法错误的情况如下图，return 语句需要在函数块中存在，不可以单独存在，因此右侧语法分析树显示“语法错误/代码不完整，无法解析”。



3. 详细设计

3.1. 小组分工

2152496 郭桉齐：语法规则设计、语法分析器构造部分

2154312 郑博远：词法分析器、语法分析器分析部分

2154318 蔡一锴：前端可视化、界面设计

3.2. 词法分析

3.2.1. 数据结构

```
# DFA 的状态
```

```python
class DFA_state(object):
    # 初始化 DFA 状态对象
    def __init__(self):
        self.transfer = {}
        self.tokenType = tokenType.UNKNOWN

# 确定有限状态自动机（DFA）
class DFA(object):
    # 初始化 DFA 对象
    def __init__(self) -> None:
        self.root = DFA_state()
        self.cur = self.root
        self.len = 0
        self.initAlpha()
        self.initDigit()
        self.initSymbol()
    # 初始化字母转移状态
    def initAlpha(self): pass
    # 初始化数字转移状态
    def initDigit(self): pass
    # 初始化符号转移状态
    def initSymbol(self): pass
    # 初始化关键词转移状态
    def initToken(self, word: str, type): pass
    # DFA 前进一步
    def forward(self, ch: str): pass
    # 重置 DFA 状态
    def reset(self): pass

# 词法分析器
class Lexer(object):
    # 初始化 Lexer 对象
    def __init__(self, lines) -> None:
        self.dfa = DFA()
        self.lines = lines
    # 获取词法分析结果
    def getLex(self): pass
```

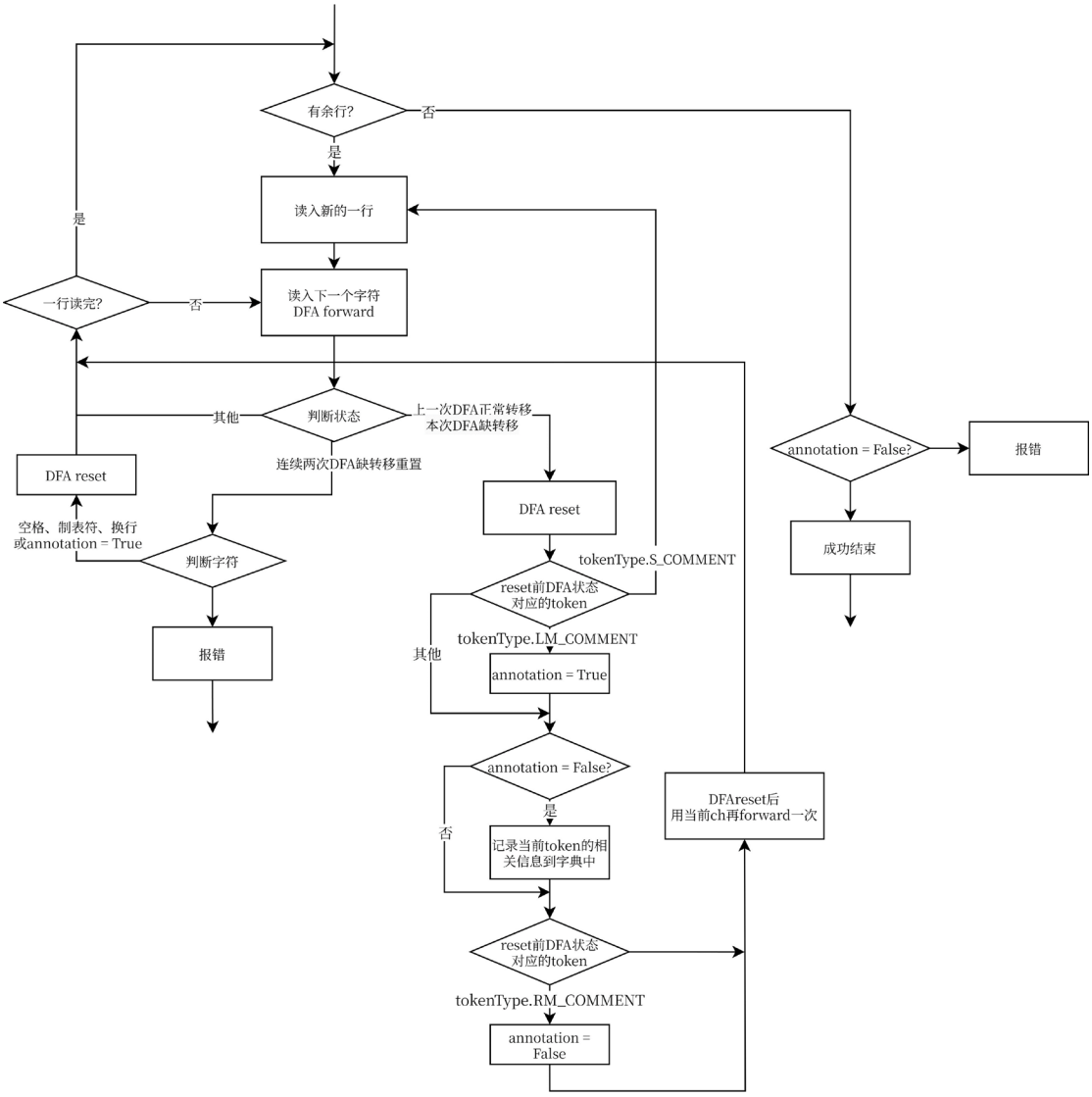
```python
class tokenType(Enum):
    UNKNOWN = "unknown"
```

```
S_COMMENT = "s_comment"
LM_COMMENT = "lm_comment"
RM_COMMENT = "rm_comment"
IDENTIFIER = "identifier"
INTEGER_CONSTANT = "integer_constant"
FLOATING_POINT_CONSTANT = "floating_point_constant"
EOF = "eof"
L_PAREN = "l_paren"
R_PAREN = "r_paren"
L_BRACE = "l_brace"
R_BRACE = "r_brace"
STAR = "star"
STAR_EQUAL = "starequal"
PLUS = "plus"
PLUS_EQUAL = "plusequal"
MINUS = "minus"
MINUS_EQUAL = "minusequal"
PERCENT = "percent"
PERCENT_EQUAL = "percentequal"
EXCLAMATION_EQUAL = "exclaimequal"
SLASH = "lash"
SLASH_EQUAL = "slashequal"
LESS = "less"
LESS_EQUAL = "lessequal"
LESS_LESS = "lessless"
LESS_LESS_EQUAL = "lesslessequal"
GREATER = "greater"
GREATER_GREATER = "greatergreater"
GREATER_EQUAL = "greaterequal"
SEMI = "semi"
EQUAL = "equal"
EQUAL_EQUAL = "equalequal"
COMMA = "comma"
KW_ELSE = "kw_else"
KW_IF = "kw_if"
KW_INT = "kw_int"
KW_FLOAT = "kw_float"
KW_RETURN = "kw_return"
KW_VOID = "kw_void"
KW_WHILE = "kw_while"
```

```
tokenSymbols = {
    '=': tokenType.EQUAL,
    '+': tokenType.PLUS,
    '+=': tokenType.PLUS_EQUAL,
    '-': tokenType.MINUS,
    '-=': tokenType.MINUS_EQUAL,
    '*': tokenType.STAR,
    '*=': tokenType.STAR_EQUAL,
    '/': tokenType.SLASH,
    '/=': tokenType.SLASH_EQUAL,
    '%': tokenType.PERCENT,
    '%=': tokenType.PERCENT_EQUAL,
    '==': tokenType.EQUAL_EQUAL,
    '>': tokenType.GREATER,
    '>>': tokenType.GREATER_GREATER,
    '>>=': tokenType.GREATER_EQUAL,
    '>=': tokenType.GREATER_EQUAL,
    '<': tokenType.LESS,
    '<<': tokenType.LESS_LESS,
    '<<=': tokenType.LESS_LESS_EQUAL,
    '<=': tokenType.LESS_EQUAL,
    '!=': tokenType.EXCLAMATION_EQUAL,
    ';': tokenType.SEMI,
    ',': tokenType.COMMA,
    '#': tokenType.EOF,
    '(': tokenType.L_PAREN,
    ')': tokenType.R_PAREN,
    '{': tokenType.L_BRACE,
    '}': tokenType.R_BRACE,
    '//': tokenType.S_COMMENT,
    '/*': tokenType.LM_COMMENT,
    '*/': tokenType.RM_COMMENT
}

tokenKeywords = {
    'int': tokenType.KW_INT,
    'float': tokenType.KW_FLOAT,
    'void': tokenType.KW_VOID,
    'if': tokenType.KW_IF,
    'else': tokenType.KW_ELSE,
    'return': tokenType.KW_RETURN,
```

```
    'while': tokenType.KW_WHILE
}
```

3.2.2. 词法分析流程图



3.2.3. 重点函数

1. 为给定单词添加状态与转移（initToken）：

该函数用于在确定性有限自动机（DFA）中为给定的单词（保留关键字，如 int、void 等；或是字符 token，如/*、+=等）初始化标记的方法。在函数内部，首先从 DFA 的根节点开始，然后遍历输入单词的每个字符。对于每个字符，函数检查当前状态的转移集合中是否包含该字符，如果没有，则创建一个新的状态。在遍历过程中，将当前状态添加到一个集合中，以便最后返回。最终，函数将输入单词的最后一个字符所在的状态标记为给定的标记类型。函数的返回值是一个包含与输入单词中的字符对应的 DFA 状态的集合。通过这个函数，可以有效地在 DFA 中初始化与特定单词相关联的标记。

```
def initToken(self, word: str, type):
    """
    为给定单词在确定有限自动机（DFA）中初始化标记。

    参数:
    - word (str): 要在 DFA 中初始化标记的输入单词。
    - type: 与给定单词相关联的标记类型。

    返回:
    - stateSet: 包含与输入单词中的字符对应的 DFA 中状态的集合。
    """

    # 从 DFA 的根节点开始
    cur = self.root
    # 创建一个空集合来存储状态
    stateSet = set()

    # 遍历输入单词中的字符
    for char in word:
        # 如果字符不在当前状态的转移集合中，则创建一个新状态
        if char not in cur.transfer:
            cur.transfer[char] = DFA_state()
        # 移动到下一个状态
        cur = cur.transfer[char]
        # 将当前状态添加到集合中
        stateSet.add(cur)

    # 为 DFA 中的最终状态设置标记类型
```

```
    cur.tokenType = type

    # 返回包含与输入单词中的字符对应的 DFA 状态的集合
    return stateSet
```

2. 初始化识别关键字和标识符（initAlpha）：

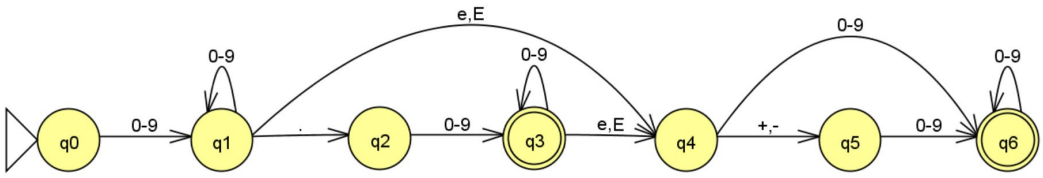
initAlpha 函数用于辨别关键字和标识符的 DFA 初始化。对于关键字，通过调用 initToken 函数，为每个关键字创建相应的 DFA 状态，并记录这些状态形成的集合。此外，其还会创建一个新状态 identifierState，表示标识符的接受态，并将所有该状态下的字母和数字的转移都指向本身，使其能够识别标识符。接下来，遍历之前记录的状态集合中的每个状态。对于那些不是关键字接受态的状态，将其标记为标识符。对于每个状态，检查其字母转移集合。如果某个字母在当前状态的转移集合中不存在，将其转移到 identifierState。类似地，对于数字也进行相似处理。通过这些步骤，initAlpha 函数能够确保 DFA 能够正确识别关键字，并将其余的字母和数字视为标识符。

```
def initAlpha(self):
    """
    初始化 DFA，识别关键字和标识符。

    该函数通过初始化关键字的 DFA 状态并记录状态集合，然后处理剩余的字母和数字的转移，将它们识别为标识符。

    返回:
    - 无，但会更新 DFA 的状态和转移。
    """

    # 初始化所有关键字对应的状态，并记录状态集合
    stateSet = set()
    for keyword in tokenKeywords:
        stateSet |= self.initToken(keyword, tokenKeywords[keyword])

    # 处理剩余的字母、数字转移，应该识别为标识符
```

```
    identifierState = DFA_state()
    identifierState.tokenType = tokenType.IDENTIFIER

    # 将所有字母和数字的转移都设置为标识符的状态
    for letter in ascii_letters + digits:
        identifierState.transfer[letter] = identifierState

    # 将所有字母的转移都设置为标识符的状态，如果字母在根节点的转移集合中不存在
    for alpha in ascii_letters:
        if alpha not in self.root.transfer:
            self.root.transfer[alpha] = identifierState

    # 处理之前记录的状态集合中的每个状态
    for state in stateSet:
        # 对于现在不是关键词接受态的，都改为标识符
        state.tokenType = state.tokenType if state.tokenType != tokenType.UNKNOWN else tokenType.IDENTIFIER
        # 将每个字母的转移都设置为标识符的状态，如果字母在当前状态的转移集合中不存在
        for alpha in ascii_letters:
            if alpha not in state.transfer:
                state.transfer[alpha] = identifierState
        # 将每个数字的转移都设置为标识符的状态，如果数字在当前状态的转移集合中不存在
        for d in digits:
            state.transfer[d] = identifierState
```

3. 初始化识别整型、浮点型常量（initDigit）：

根据下图构造 DFA 的对应状态和转移。其中，下图的接受态为浮点型常量的接受态，q1 状态对应整型常量的接受态。



```
def initDigit(self):
    """
    为整型和浮点型常量初始化对应的状态和转移。

    该函数创建一个包含 7 个状态的 DFA，用于识别整数和浮点数。具体的状态转移图详见状态转移图。

    返回:
    - 无，但会更新 DFA 的状态和转移。
    """

    # 建立 6 个新状态
    states = {}
    for i in range(7):
        if i == 0:
            states[i] = self.root
        else:
            states[i] = DFA_state()

    # 设置接受态
    states[1].tokenType = tokenType.INTEGER_CONSTANT
    states[3].tokenType = tokenType.FLOATING_POINT_CONSTANT
    states[6].tokenType = tokenType.FLOATING_POINT_CONSTANT

    # 添加所有 0-9 的转移
    for d in digits:
        states[0].transfer[d] = states[1]
        states[1].transfer[d] = states[1]
        states[2].transfer[d] = states[3]
        states[3].transfer[d] = states[3]
        states[4].transfer[d] = states[6]
        states[5].transfer[d] = states[6]
        states[6].transfer[d] = states[6]

    # 添加其余琐碎转移
    states[1].transfer['.'] = states[2]
    states[1].transfer['e'] = states[4]
    states[1].transfer['E'] = states[4]
    states[3].transfer['e'] = states[4]
    states[3].transfer['E'] = states[4]
    states[4].transfer['+'] = states[5]
```

```
    states[4].transfer['-'] = states[5]
```

4. 词法分析主体（getLex）：

该函数为词法分析的主体部分。具体的算法思想详见 2.2.2 词法分析流程图部分，此处不再赘述。函数代码如下：

```
def getLex(self):
    """
    从源代码中提取词法单元。

    该函数使用已初始化的 DFA 对源代码进行词法分析，将识别到的词法单元存储在列表中。

    返回:
    - ret: 包含词法单元信息的列表，每个词法单元以字典形式存储，包括 id、content、prop（类型）、loc（位置信息）等字段。
    """

    ret = []            # 存储词法单元的列表
    row = 1             # 记录当前行数
    id = 1              # 词法单元的唯一标识
    annotation = False  # 标记是否处于注释块中

    # 遍历源代码的每一行
    for line in self.lines:
        self.dfa.reset()
        token = None
        col = 0

        # 在每一行的末尾加上一个字符以确认每一行最后一个输入 DFA 的串
        for ch in line + '#':
            newtoken, newlen = self.dfa.forward(ch)

            if token and not newtoken:
                # 处理注释类型的词法单元
                if token == tokenType.S_COMMENT:
                    break
                elif token == tokenType.LM_COMMENT:
```

```
                    annotation = True

                # 存储识别到的词法单元信息
                if not annotation:
                    item = {}
                    item['id'] = id
                    id += 1
                    item['content'] = line[col - len:col]
                    item['prop'] = token
                    item['loc'] = f'{row}, {col - len + 1}'
                    ret.append(item)

                if token == tokenType.RM_COMMENT:
                    annotation = False

                self.dfa.reset()
                token, len = self.dfa.forward(ch)
            elif not token and not newtoken:
                # 处理非注释的无效字符
                if not annotation and ch not in [' ', '\t',
'\n']:
                    raise Exception('Invalid code format!')
            else:
                token, len = newtoken, newlen
            col += 1
        row += 1

    # 添加文件末尾的 EOF 标记
    ret.append({'id': id, 'content': '#', 'prop':
tokenType.EOF, 'loc': f'{row}, 1'})

    return ret
```

3.3. 语法分析

3.3.1. 数据结构

```
class Production:
    def __init__(self):
        self.id = 0
        self.from_id = 0
        self.to_ids = []

    def __lt__(self, other):
        return self.id < other.id
```

Production 类表示形式文法中的产生式规则，它具有三个属性：

id：产生式规则的唯一标识符。

from_id：表示产生式的源或非终结符的标识符。

to_ids：包含目标符号标识符的列表（本项目中是 int），这些符号可以是终结符或非终结符。

__lt__方法被定义，以便根据它们的 id 属性对 Production 对象进行排序。通过实现两个 Production 对象之间的小于比较来实现此功能。

```
class Item:
    def __init__(self, production_id, dot_pos, terminal_id):
        self.production_id = production_id  # 产生式编号
        self.dot_pos = dot_pos  # 点的位置
        self.terminal_id = terminal_id  # 终结符 ID

    def __lt__(self, other):
        return (self.production_id, self.dot_pos,
self.terminal_id) < (other.production_id, other.dot_pos,
other.terminal_id)

    def __eq__(self, other):
        return (self.production_id, self.dot_pos,
self.terminal_id) == (other.production_id, other.dot_pos,
other.terminal_id)
```

Item 类表示 LR(1)文法中的项目，它具有三个属性：

production_id：产生式的编号。

dot_pos：点的位置，表示在产生式右侧哪个符号之前放置了点。

terminal_id：终结符的标识符。

__lt__方法被定义，以便根据产生式编号、点的位置和终结符 ID 对 Item 对象进行排序。通过实现小于比较来实现此功能。__eq__方法被定义，以便在相等性比较时使用相同的标准。

```
class Closure:
    def __init__(self):
        self.id = 0  # 编号
        self.items = []  # 项目集

    def __lt__(self, other):
        return self.id < other.id

    def __eq__(self, other):
        t1 = sorted(self.items)
        t2 = sorted(other.items)
        if len(t1) != len(t2):
            return False
        for i in range(len(t1)):
            if t1[i] != t2[i]:
                return False
        return True
```

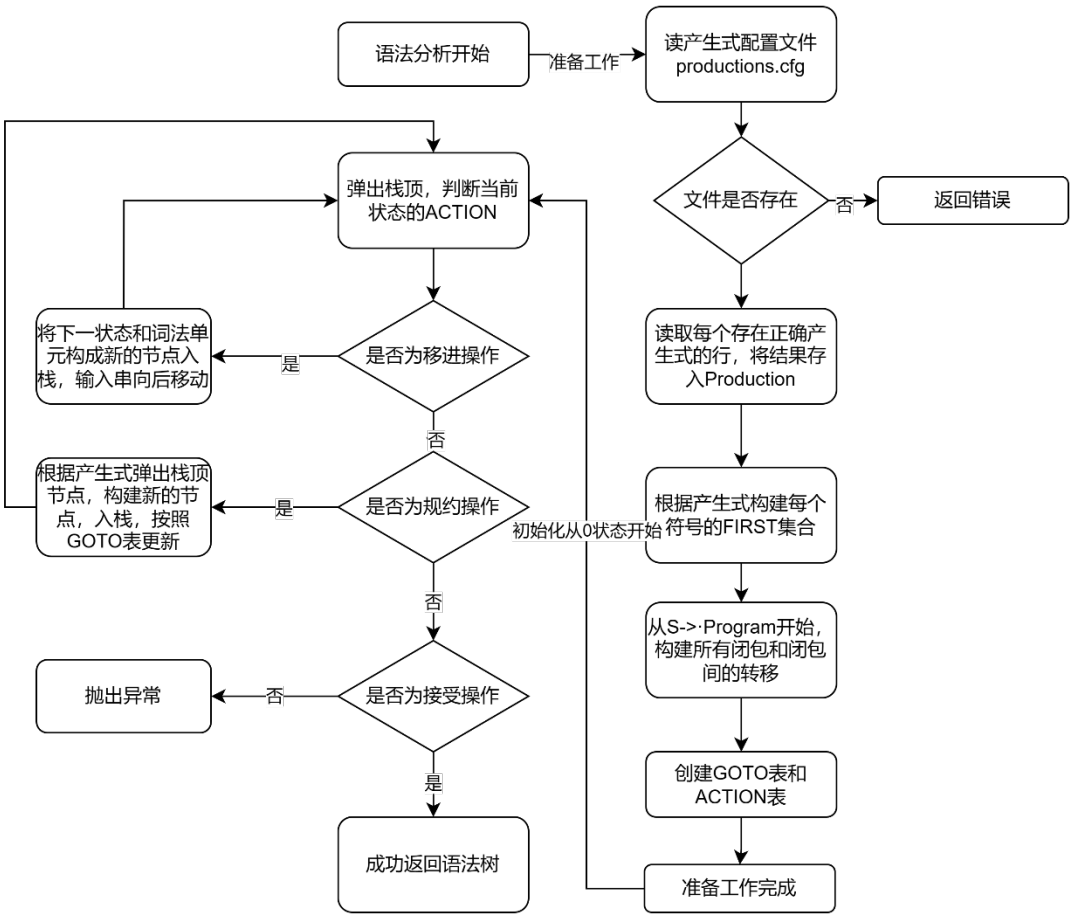
Closure 类表示 LR(1)文法中的闭包（Closure），通常在 LR(1)分析器中用于构建项目集规范族，它具有两个属性：

id：闭包的编号。

items：项目集，是由 LR(1)文法中的项目（Item）组成的集合。

__lt__方法被定义，以便根据编号对 Closure 对象进行排序。通过实现小于比较来实现此功能。__eq__方法被定义，以便在相等性比较时使用相同的标准。在这里，相等性是通过比较两个闭包的项目集的排序后是否相等来确定的。

3.3.2. 语法分析流程图



3.3.3. 重点函数

1. 读产生式文件函数（read_productions）：

本函数用于从指定的产生式配置文件中读取产生式规则，并将其处理后存储在相应的数据结构中。

使用 with open 语句打开指定的配置文件，通过正则表达式匹配每一行的产生式规则格式，处理产生式左侧和右侧，将终结符和非终结符添加到相应的列表中，创建 Production 对象，设置其属性，并将其添加到产生式列表中。如果配置文件不存在，捕获 FileNotFoundError 异常，并返回 False，如果成功读取配置文件并处理产生式规则，返回 True。

```
    def read_productions(self, filename="production.cfg"):
        """
        从产生式配置文件中读取产生式规则。

        参数:
        - filename: 产生式配置文件的路径，默认为"production.cfg"。

        返回:
        - success: 如果成功读取配置文件并处理产生式规则，则返回 True；如果文件不存在，返回 False。
        """
        try:
            with open(filename, "r") as fin:  # 打开配置文件进行读取
                for line in fin.readlines():  # 逐行读取文件内容
                    tmp = None
                    match = re.match(r"\s*([^->]+)\s*->\s*(.*)", line)  # 用正则表达式匹配产生式规则的格式
                    if match:
                        left_side = match.group(1).strip()  # 获取产生式左侧（非终结符）并去除空格
                        right_side = match.group(2).strip()  # 获取产生式右侧（可能包含多个替代项）并去除空格
                        alternatives = [alt.strip() for alt in right_side.split("|")]  # 将右侧的替代项分割成列表

                        # 处理非终结符
                        if left_side not in self.non_terminal_symbols:
                            self.non_terminal_symbols.append(left_side)

                        from_id = self.non_terminal_symbols.index(left_side) + len(self.terminal_symbols)  # 计算产生式左侧非终结符的索引

                        # 处理每个替代项
                        for alt in alternatives:
                            tmp = Production()
                            tmp.id = len(self.productions)
                            tmp.from_id = from_id
```

```
                            # 处理替代项中的每个符号
                            alt_split = [x for x in alt.split("
")]
                            for symbol in alt_split:
                                if symbol in
self.terminal_symbols:
                                    to_id =
self.terminal_symbols.index(symbol)
                                elif symbol in
self.non_terminal_symbols:
                                    to_id =
self.non_terminal_symbols.index(symbol) +
len(self.terminal_symbols)
                                else:
                                    # 如果符号不是终结符也不是非终结
符，将其添加到非终结符列表中
                                    self.non_terminal_symbols.appe
nd(symbol)
                                    to_id =
self.non_terminal_symbols.index(symbol) +
len(self.terminal_symbols)

                                tmp.to_ids.append(to_id)  # 将产生
式右侧每个符号的索引添加到产生式对象的 to_ids 列表中

                            self.productions.append(tmp)  # 将产生
式对象添加到产生式列表中
        except FileNotFoundError:
            return False  # 如果文件不存在，返回 False
        return True  # 如果成功读取配置文件并处理产生式规则，返回 True
```

2. 寻找每个终结符和非终结符的 first 集函数（find_firsts）：

函数开始时，通过循环初始化终结符的 First 集合，每个终结符的 First 集合包含自身，同时初始化非终结符的 First 集合为空集，使用 flag 标志来判断是否在一次循环中有新的元素加入了 First 集合。

进入主循环，遍历文法中的每个产生式规则，对于每个产生式规则，遍历产生式右侧的每个符号。

如果符号是终结符或 epsilon，将其加入产生式左侧非终结符的 First 集合，如果 epsilon 在非终结符的 First 集合中，跳出当前循环（不再添加后续符号）。如果符号是非终结符，将其 First 集合中的元素加入产生式左侧非终结符的 First 集合。如果 epsilon 不在符号的 First 集合中，跳出当前循环。如果在一次循环中没有新的元素加入 First 集合，说明已经计算完成，可以退出循环。

通过以上步骤，函数通过多次循环遍历产生式规则，逐步计算和更新每个非终结符的 First 集合，直到没有新元素加入为止。这样就完成了文法的 First 集合的计算。

```
    def find_firsts(self):
        """
        计算文法的 First 集合。

        First 集合是文法中每个非终结符的一个集合，包含其能推导出的所有可能的首终结符（终结符或 epsilon）。

        返回：无，直接更新 self.firsts 列表。

        注意：
        - self.terminal_symbols：终结符列表
        - self.non_terminal_symbols：非终结符列表
        - self.productions：产生式规则列表
        - self.firsts：存储 First 集合的列表，每个元素是一个集合，对应一个非终结符的 First 集合。
        """
        # 初始化终结符
        for i in range(len(self.terminal_symbols)):
            self.firsts.append({i})
        # 初始化非终结符
        for i in range(len(self.terminal_symbols)):
            self.firsts.append(set())
        flag = False # 判断在一次循环中是否有新的元素加入了 first 集合
        while True:
            flag = False
            for production in self.productions:
                from_id = production.from_id
                for i in range(len(production.to_ids)):
```

```
                    if production.to_ids[i] <=
len(self.terminal_symbols):  # 是终结符或 epsilon
                        if production.to_ids[i] not in
self.firsts[from_id]:  # 不在 first 集中
                            flag = True
                            self.firsts[from_id].add(production.t
o_ids[i])
                        break
                    else:
                        for id in
self.firsts[production.to_ids[i]]:
                            if id != len(self.terminal_symbols)
and id not in self.firsts[from_id]:  # 以前没放进去
                                flag = True
                                self.firsts[from_id].add(id)
                        if (len(self.terminal_symbols) not in
self.firsts[production.to_ids[i]] ):  # epsilon 不在非终结符的
first 集中
                            break
                        elif (i == len(production.to_ids)-1 and
len(self.terminal_symbols) not in self.firsts[from_id]):  # 本次
是最后一次，且 epsilon 以前没放进去
                            flag = True
                            self.firsts[from_id].add(len(self.ter
minal_symbols))
            if not flag:
                break
```

3. 寻找给定句子的 first 集函数（find_firsts_alpha）：

本函数用于找句子的 First 集合，即给定一个符号序列（alpha），计算其 First 集合，并将结果存储在传入的 firsts 集合中。

使用 firsts.clear() 清空传入的 firsts 集合，以便重新计算句子的 First 集合，遍历句子中的每个符号，对于每个符号，将其非终结符的 First 集合中的元素添加到句子的 First 集合中，如果 epsilon 不在当前符号的 First 集合中，终止循环，如果当前符号是句子的最后一个符号，并且 epsilon 在其 First 集合中，将 epsilon 添加到句子的 First 集合中。

```
    def find_firsts_alpha(self, alpha, firsts):
        """
        找句子的 First 集合。

        参数:
        - alpha: 包含整数的列表，表示句子中的符号序列。
        - firsts: 用于存储句子 First 集合的集合，该集合将被清空并更新。

        注意:
        - self.firsts: 存储文法非终结符 First 集合的列表。
        - self.terminal_symbols: 终结符列表。
        """
        firsts.clear()  # 清空传入的 firsts 集合

        for i in range(len(alpha)):
            for id in self.firsts[alpha[i]]:
                if id != len(self.terminal_symbols):
                    firsts.add(id)  # 将文法非终结符的 First 集合添加
到句子 First 集合中
            if len(self.terminal_symbols) not in
self.firsts[alpha[i]]:
                break  # 如果 epsilon 不在当前符号的 First 集合中，终止
循环
            if (i == len(alpha) - 1 and
len(self.terminal_symbols) in self.firsts[alpha[i]]):
                firsts.add(len(self.terminal_symbols))  # 如果是句
子的最后一个符号，并且 epsilon 在其 First 集合中，添加 epsilon 到句子
First 集合中
```

4. 寻找闭包函数（find_closures）：

本函数用于找闭包，根据文法中的产生式和给定的闭包，计算新的项目并添加到闭包中，满足闭包中某个项目的点位置后面是非终结符的情况。

对于给定闭包中的每个项目：如果项目的点位置已经到达产生式右侧的末尾，跳过，获取项目点后的符号的 ID，如果该符号是终结符或 epsilon，跳过当前循环，对于每个产生式，检查是否产生式的左侧是当前符号 ID。创建新的项目 alpha，从当前产生式符号的下一个开始遍历，如果不是 epsilon，则添加到 alpha 中，调用 self.find_firsts_alpha 计算后继符号的 First 集，对于每个

First 集合中的符号，创建新的项目，并将其添加到闭包中，如果项目不在闭包中，最后更新循环计数器，进行下一轮迭代。

```
def find_closures(self, closure):
        """
        找闭包的实现，若有项目[A→α·Bβ,a ]属于 CLOSURE(I)，B→γ 是文法中的产生式，β∈V*，b∈FIRST(βa)，则[B→·γ,b]也属于 CLOSURE(I)中。

        参数:
        - closure: 闭包对象，其中包含项目的集合。

        注意:
        - self.productions: 存储文法产生式的列表。
        - self.terminal_symbols: 存储终结符的列表。
        """
        # 对于给定闭包中的每个项目
        i = 0
        while i < len(closure.items):
            # 如果项目的点位置已经到达产生式右侧的末尾，跳过
            if closure.items[i].dot_pos >=
len(self.productions[closure.items[i].production_id].to_ids):
                i += 1
                continue
            # 获取项目点后的符号的 ID
            symbol_id =
self.productions[closure.items[i].production_id].to_ids[closure
.items[i].dot_pos]
            # 如果该符号是终结符或 epsilon，跳过
            if symbol_id <= len(self.terminal_symbols):
                i += 1
                continue
            # 对于每个产生式
            for j in range(len(self.productions)):
                # 如果产生式的左侧是当前符号 ID
                if self.productions[j].from_id == symbol_id:
                    # 创建新的项目
                    alpha = []
                    # 从当前产生式符号的下一个开始遍历，如果不是 epsilon
                    for k in range(closure.items[i].dot_pos + 1,
len(self.productions[closure.items[i].production_id].to_ids)):
```

```
                        if
self.productions[closure.items[i].production_id].to_ids[k] !=
len(self.terminal_symbols):
                            alpha.append(self.productions[closure
.items[i].production_id].to_ids[k])
                    alpha.append(closure.items[i].terminal_id)
                    # 计算后继符号的 first 集
                    firsts = set()
                    self.find_firsts_alpha(alpha, firsts)
                    # 对于每个 first 集合中的符号，创建新的项目并加入闭包
                    for first in firsts:
                        item = Item(j, 0, first)
                        if item not in closure.items:
                            closure.items.append(item)
            i += 1
```

5. 创建 gos 表函数（find_gos）：

本函数用于构建 Go 表，即计算每个闭包的后继闭包及其映射。

首先遍历闭包列表，构建每个闭包的后继闭包及其映射，对于每个终结符和非终结符的编号，进行以下操作：创建新的闭包 tmp，对于当前闭包中的每个项，如果项的点位置已经到达产生式右侧的末尾，跳过当前循环，如果产生式右侧的下一个符号的编号是当前遍历的编号，创建新的项，表示将点向后移动一位，调用 self.find_closures 找新闭包的闭包，如果找到相同的闭包，更新映射，如果未找到相同的闭包，添加新的闭包及其映射，最后更新循环计数器，进行下一轮迭代。

```
    def find_gos(self):
        """
        找 Go 表的实现。

        注意:
        - self.productions: 存储文法产生式的列表。
        - self.terminal_symbols: 存储终结符的列表。
        - self.non_terminal_symbols: 存储非终结符的列表。
        - self.closures: 存储闭包的列表。
```

```
        - self.gos: 存储 Go 表的列表，每个元素是一个字典，表示一个闭包的
后继闭包及其映射。
        """
        # 创建新的产生式 S'->Program，并将其添加到产生式列表中
        new_production = Production()
        new_production.id = len(self.productions)
        new_production.from_id = len(self.terminal_symbols) +
len(self.non_terminal_symbols)  # S
        new_production.to_ids.append(len(self.terminal_symbols)
+ 1)  # Program
        self.productions.append(new_production)  # S -> Program

        # 创建初始项，表示 S->.Program,#
        new_item = Item(len(self.productions)-1, 0,
len(self.terminal_symbols)-1)

        # 创建初始闭包，包含初始项
        start_closure = Closure()
        start_closure.items.append(new_item)

        # 找初始闭包的闭包
        self.find_closures(start_closure)

        # 将初始闭包及其映射加入闭包列表和映射列表
        self.closures.append(start_closure)
        self.gos.append({})

        # 初始化当前闭包标识
        now_closure_id = 0

        # 遍历闭包列表，构建闭包的后继闭包及其映射
        while now_closure_id < len(self.closures):
            # 遍历所有终结符和非终结符的编号
            for i in range(len(self.terminal_symbols) +
len(self.non_terminal_symbols) + 1):
                if i == len(self.terminal_symbols):  # epsilon
                    continue

                # 创建新的闭包
                tmp = Closure()
```

```
                # 遍历当前闭包中的每个项
                for item in self.closures[now_closure_id].items:
                    # 如果项的点位置已经到达产生式右侧的末尾，跳过
                    if
len(self.productions[item.production_id].to_ids) ==
item.dot_pos:
                        continue

                    # 如果产生式右侧的下一个符号的编号是当前遍历的编号
                    if
self.productions[item.production_id].to_ids[item.dot_pos] == i:
                        # 创建新的项，表示将点向后移动一位
                        new_item = Item(item.production_id,
item.dot_pos + 1, item.terminal_id)
                        tmp.items.append(new_item)

                # 如果新的闭包中有项
                if tmp.items:
                    # 找新闭包的闭包
                    self.find_closures(tmp)

                    # 如果找到相同的闭包，更新映射
                    if tmp in self.closures:
                        self.gos[now_closure_id][i] =
self.closures.index(tmp)
                    # 如果未找到相同的闭包，添加新的闭包及其映射
                    else:
                        tmp.id = len(self.closures)
                        self.closures.append(tmp)
                        self.gos.append({})
                        self.gos[now_closure_id][i] = tmp.id

            now_closure_id += 1
```

6. 创建 goto 表和 action 表函数 （find_gotos）：

本函数实现了构建 Goto 表和 Action 表的逻辑。

Goto 表：对于每个非终结符，将其映射到后继闭包。

Action 表：对于每个闭包中的项目：处理 Reduce 操作：如果 • 在产生式右

侧的末尾，即 [A->α· , a] 在闭包中，且 A≠S，那么将 action[i, a] 置为 reduce j。处理 Accept 操作：如果 [S->Program·, #] 在闭包中，那么将 action[i, #] 置为 acc。处理 Shift 操作：如果 [A->α·aß, b] 在闭包中，且 GO(Ii, a) = Ij ，那么将 action[i, a] 置为 shift j。

```
    def find_gotos(self):
        """
        goto 和 action 表的实现。

        注意:
        - self.terminal_symbols: 存储终结符的列表。
        - self.non_terminal_symbols: 存储非终结符的列表。
        - self.closures: 存储闭包的列表。
        - self.gos: 存储 Go 表的列表，每个元素是一个字典，表示一个闭包的后继闭包及其映射。
        - self.productions: 存储文法产生式的列表。
        - self.goto_table: 存储 Goto 表的列表，每个元素是一个字典，表示一个闭包的后继闭包及其映射。
        - self.action_table: 存储 Action 表的列表，每个元素是一个字典，表示一个闭包的后继闭包及其映射。
        """

        for i in range(len(self.closures)):
            self.goto_table.append({})
            self.action_table.append({})

            # 处理 Goto 表
            for tmp in self.gos[i].items():
                # 如果是非终结符
                if tmp[0] > len(self.terminal_symbols):
                    self.goto_table[i][tmp[0]] = tmp[1]

            # 处理 Action 表
            for item in self.closures[i].items:
                # 如果·在末尾
                if
len(self.productions[item.production_id].to_ids) ==
item.dot_pos:
```

```
                    if
self.productions[item.production_id].from_id !=
len(self.terminal_symbols) + len(self.non_terminal_symbols):
                        # 如果[A->α· , a]在 Ii 中，且 A≠S，那么置
action[i, a]为 reduce j
                        action = (ACTION_R, item.production_id)
                        self.action_table[i][item.terminal_id] =
action
                    else:
                        # 如果[S->Program·, #]在 Ii 中，那么置
action[i, #] = acc
                        if item.terminal_id ==
len(self.terminal_symbols) - 1:
                            action = (ACTION_ACC, 0)
                            self.action_table[i][item.terminal_id
] = action
                else:
                    item_behind_dot =
self.productions[item.production_id].to_ids[item.dot_pos]

                    if item_behind_dot <
len(self.terminal_symbols):
                        # 如果[A->α·aß, b]在 Ii 中，且 GO(Ii, a) =
Ij , 那么置 action[i, a]为 shift j
                        if item_behind_dot in self.gos[i]:
                            action = (ACTION_S,
self.gos[i][item_behind_dot])
                            self.action_table[i][item_behind_dot]
= action
```

7. 移进规约过程语法分析函数（getParse）：

本函数实现了 LR(1)语法分析过程，根据给定的词法分析结果生成语法树

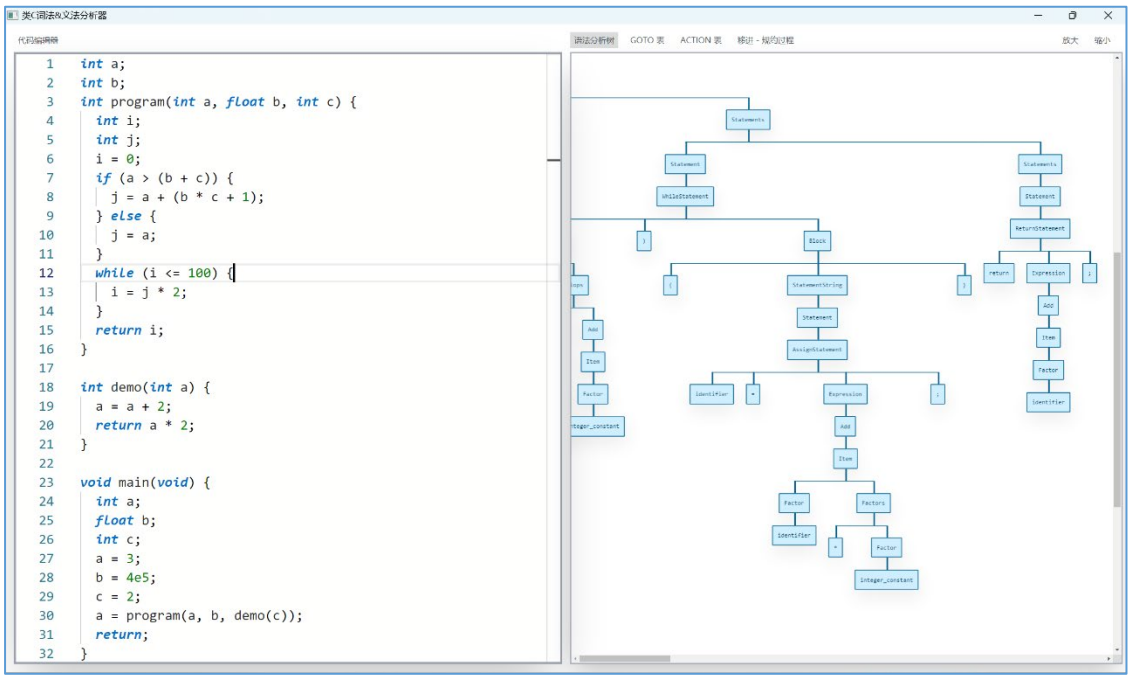
首先初始化栈，设置初始状态为 0，不断从输入串中读取词法单元，根据 Action 表进行相应的移进或规约操作，根据 Goto 表进行状态的更新。如果遇到 ACTION_ACC 表示成功接受，返回生成的语法树，在代码中使用 assert 语句确保 Action 表中包含了所需的转移信息。

```
# 语法分析函数的实现
    def getParse(self, lex):
        """
        执行语法分析。

        参数:
        - lex: 词法分析的输出结果，包含词法单元的信息。

        返回:
        - 树形结构，表示语法分析的结果。
        """
        stack = []
        item = {'state': 0}
        stack.append(item)

        index = 0
        while index < len(lex):
            cur = lex[index]
            token = tokenType_to_terminal(cur['prop'])
            token_id = self.get_id_by_str(token)

            current_state = self.action_table[stack[-
1]['state']]
            assert token_id in current_state, 'Action table
lacks the given transition!'

            if current_state[token_id][0] == ACTION_S:
                next_state_id = current_state[token_id][1]
                item = {'state': next_state_id, 'tree': {'root':
token, 'children': []}}
                stack.append(item)
                index += 1      # 当前输入串移动到下一个字符

            elif current_state[token_id][0] == ACTION_R:
                production =
self.productions[current_state[token_id][1]]

                children = []
                for id in production.to_ids:
                    child = stack.pop()['tree']
                    children.insert(0, child)
```

```
                current_state = self.goto_table[stack[-
1]['state']]
                next_state_id =
current_state[production.from_id]
                item = {'state': next_state_id, 'tree': {'root':
self.get_str_by_id(production.from_id), 'children': children}}

                stack.append(item)

            elif current_state[token_id][0] == ACTION_ACC:
                return stack[-1]['tree']

            else:
                raise Exception('Invalid action type!')

        return None  # 如果无法成功完成语法分析，返回 None
```

3.4. 界面设计

3.4.1. 页面总设计图



3.4.2. 重点函数

1. 语法分析器计算部分和用户界面的异步加载

由于 Parser 需要一段比较长的时间读取 production.cfg 并完成初始化（大约需要十几秒），为了保证用户体验，这里使用 Qt 的信号-信号槽模型，通过一个 worker QThread 来完成 Parser 的初始化，并将准备完成的 Parser 传递给主 Qt 线程。

这样就实现了 Parser 的异步加载，使得用户可以先看到启动完成的 Qt 界面以及“Parser 正在加载中”的提示，等待一段时间后由主 Qt 线程主动刷新界面，展示 Parser 结果。

```
class ParserLauncher(QObject):
    returnParser = pyqtSignal(Parser)
    finished = pyqtSignal()

    def run(self):
        self.returnParser.emit(Parser())  # 加载完成后传递给主线程
        self.finished.emit()  # 通知主线程销毁该子线程

# Compiler 类中与此相关的部分
class Compiler(QObject):
    def __init__(self, parent=None):
        super().__init__(parent)
        self.lexer = Lexer()
        self.parser = None
        self.parser_init_begin = time.time()
        self.getParserNonBlock()  # 非阻塞加载 Parser

    def getParserNonBlock(self):
        self.pl_thread = QThread()
        self.pl_worker = ParserLauncher()
        self.pl_worker.moveToThread(self.pl_thread)
        self.pl_thread.started.connect(self.pl_worker.run)
        self.pl_worker.returnParser.connect(self.setParser)
        self.pl_worker.finished.connect(self.pl_thread.quit)
```

```
        self.pl_thread.finished.connect(self.pl_thread.deleteLa
ter)
        self.pl_thread.start()

    @pyqtSlot(Parser)
    def setParser(self, parser):
        self.parser = parser  # 接收子线程传递过来的 Parser 对象
        delta = time.time() - self.parser_init_begin
        print(f"self.parser is set to {self.parser}")
        print(f"Parser took {delta} seconds to initialize")
        self.goto_table = self.parser.get_goto_table()  # 计算
goto 表（不随输入变化，只需计算一次）
        self.action_table = self.parser.get_action_table()  # 计
算 action 表（不随输入变化，只需计算一次）
        self.parent().page().runJavaScript("window.cpf.flush();
")  # 主动刷新用户界面
```

2. 语法分析器和用户界面的通信，以及对具体计算函数的调用

由于用户界面和语法分析器也是异步运行的，并且用户界面代码由 JavaScript 编写，并且运行在 QWebEngineView 这个类似浏览器的环境中，所以它们二者之间的通信也使用了 Qt 的信号-信号槽模型。UI 代码可以通过 QWebChannel 调用 Compiler.process() 函数，间接调用语法分析函数，并获取其结果。

```
# 省略了 Parser 加载的 Compiler 类的剩余部分
class Compiler(QObject):
    def __init__(self, parent=None):
        super().__init__(parent)
        self.lexer = Lexer()
        self.parser = None
        self.parser_init_begin = time.time()
        self.getParserNonBlock()  # 省略了 Parser 加载的剩余部分

    @pyqtSlot(str, result=str)
    def process(self, code_str):
```

```
        token_list, lexer_success = self.getLex(code_str)
        parse_result = self.getParse(token_list)
        return json.dumps(
            {
                "lexer": self.dumpTokenList(token_list),
                "lexer_success": lexer_success,
                **parse_result,
            }
        )

    def dumpTokenList(self, token_list):
        def dumpToken(r):
            r["prop"] = r["prop"].value
            return r

        return list(map(dumpToken, token_list))

    def getParse(self, token_list):
        if self.parser is not None:
            parsed_result = self.parser.getParse(token_list)
            return {  # 如果 Parser 已经加载完成，返回 Parser 结果
                "ast": parsed_result,
                "goto": self.goto_table,
                "action": self.action_table,
                "process": self.parser.parse_process_display,
            }
        else:
            launching = "Parser 正在启动，请稍等。"
            return {  # 如果 Parser 未加载完成，返回提示信息
                "ast": {
                    "root": launching,
                    "err": "parser_not_ready",
                },
                "goto": [[launching]],
                "action": [[launching]],
                "process": [[launching]],
            }

    def getLex(self, code_str: str):
        return self.lexer.getLex(code_str.splitlines())
```

3. QWebEngineView 与 aiohttp.web.Application 双进程并行

用户界面代码由 HTML + JavaScript + CSS 写成，但 QWebEngineView 本身并没有直接加载它们的能力，需要一个 HTTP Server 来 serve 这些静态文件。我们通过在子进程中分别运行 aiohttp Server 和主 Qt 进程，并通过环回地址来完成 QWebEnginView 对本地静态文件的访问。

```
class MainWindow(QWebEngineView):
    def __init__(self):
        super(MainWindow, self).__init__()
        self.load(QUrl(f"http://localhost:{serverPort}/index.ht
ml"))  # 通过环回地址访问 Server 提供的本地静态文件
        self.webChannel = QWebChannel(self.page())  # 初始化 PyQt
侧的 QWebChannel
        self.webChannel.registerObject("compiler",
Compiler(self))  # 初始化 Compiler 对象，绑定到 QWebChannel 上
        self.page().setWebChannel(self.webChannel)  # 应用
QWebChannel ，触发 JavaScript 侧 QWebChannel 初始化


def ServerProcess(application_path):
    # serve 本地静态文件
    app = web.Application()
    app.add_routes(
        [
            web.static(
                "/",
                os.path.join(application_path, serverDirectory),
                show_index=True,
                follow_symlinks=False,
                append_version=True,
            )
        ]
    )
    web.run_app(app, host="localhost", port=serverPort)  # 监听
环回地址，响应对本地静态文件的 HTTP 请求


def QtProcess():
    app = QApplication(sys.argv)
```

```
    window = MainWindow()
    window.showMaximized()
    app.exec_()
```

4. 语法分析树的绘制

语法分析树本质上与 HTML DOM 树是类似的树形结构。因此，可以对语法分析树进行递归遍历，然后按照遍历顺序输出 HTML 标签，即可得到一颗 HTML DOM 树，从而实现语法分析树可视化。

这部分其实类似一个状态机：由“绘制节点”和“绘制节点的子节点列表”两个状态组成，运行时会根据 Parser 返回的语法分析树结果在其中来回转移。

除此之外，我们还利用 CSS 的 transform: scale() 方法实现了语法分析树的缩放，方便用户观察这棵树的细节与全貌。

```
import $ from "jquery"

class TreeView {
  constructor() {
    this.buffer = []
    this.scaleProp = "ast_tree_scale"
    this.scale(0.75)
    const baseFactor = 11 / 10
    $("#ast-zoom-in").on("click", () => {
      this.scale(baseFactor)  // 响应【放大】按钮
    })
    $("#ast-zoom-out").on("click", () => {
      this.scale(1 / baseFactor)  // 响应【缩小】按钮
    })
  }
  scale(factor) {
    if (!window[this.scaleProp]) {
      window[this.scaleProp] = 1
    }
    window[this.scaleProp] *= factor;
    $("#ast-tree").css("transform",
`scale(${window[this.scaleProp]})`)  // 缩放 TreeView
```

```
  }
  draw(tree) {
    if (typeof tree === "object" && !Array.isArray(tree)) {
      $("#ast-tree").html("");
      this.buffer = ["<ul>"];
      this.drawChildNode(tree);
      this.buffer.push("</ul>");
      $("#ast-tree").html(this.buffer.join(""));
    } else {
      console.error(`Cannot display ${typeof tree} in TreeView:
${tree}`)
    }
  }
  drawNode(node) {
    for (const prop in node) {
      if (prop === "children") {
        if (node["children"].length > 0) {
          this.buffer.push("</div><ul>");
          for (const child of node["children"]) {
            this.drawChildNode(child)  // 绘制子节点列表
          }
          this.buffer.push("</ul>");
        }
      } else if (prop === "root") {
        this.buffer.push(`<span> ${node[prop]} </span>`);  // 绘
制该节点的类型
      }
    }
  }
  drawChildNode(child) {
    this.buffer.push(`<li><div class="ast-tree-node">`);
    // fetch the parent object
    this.drawNode(child);
    // push the closing tag for parent
    this.buffer.push("</li>");
  }
}

// call the function on page load
window.addEventListener("load", () => {
  window.treeView = new TreeView()
```

```
})
```

5. 主用户界面代码

这部分代码主要工作是：初始化 QWebChannel 以及用户界面主体（包括左侧的 Monaco Editor）。值得一提的是，虽然我们使用的 Monaco Editor 自带代码高亮功能，但是我们通过使用纯文本模式将其禁用，然后应用了我们自己根据我们的语法分析器结果生成的代码高亮。

用户界面控制逻辑基本写在 CPF 类中，这个类会在窗口加载完成后实例化，并且在 QWebChannel 初始化完成后从 Python 端接受一个 compiler 对象（Python 代码中的 Compiler 类），来完成 JavaScript 端和 Python 端的连接。

这个类会在编辑器中代码发生变化时，取得其中的文本内容，传递给语法分析器，获取词法分析、语法分析结果，以及其他需要展示的内容，并且调用代码高亮、语法分析树以及各个表格的绘制代码。

```
import 'virtual:uno.css'
import '@unocss/reset/tailwind.css'

import $ from 'jquery'

import testCode from "./test.c?raw"

import * as monaco from 'monaco-editor'
import editorWorker from 'monaco-
editor/esm/vs/editor/editor.worker?worker'
self.MonacoEnvironment = { getWorker: () => new
editorWorker() }

class CPF {
  constructor() {
    this.editor = monaco.editor.create(  // 初始化 Monaco
Editor , 绑定到 HTML DOM 树上
      document.getElementById('cm-container'),
      {
```

```
        value: testCode,  // 默认打开样例代码
        language: 'plain',  // 纯文本模式，禁用自带的代码高亮，由我们的语法分析器生成代码高亮
        minimap: {
          enabled: false
        },
        cursorBlinking: "smooth",  // 光标闪烁
        automaticLayout: true,
        fixedOverflowWidgets: true,
      },
    )
    this.editor.getModel().onDidChangeContent(() => {
      window.cpf.flush()  // 监听内容变更，刷新语法分析结果
    })
    window.addEventListener("resize", () => {
      this.editor.layout()  // 监听页面缩放，刷新编辑器布局
    })
  }

  callCompiler(code_str) {
    if (!this.compiler) return  // 如果 QWebChannel 未初始化完毕，则不进行下面步骤。初始化完毕后 this.compiler 对象将会可用（对应上方 Python 代码中的 Compiler 类）
    this.compiler.process(code_str)
      .then(JSON.parse)  // 通过 JSON 传递信息
      .then((result) => {
        if (window.treeView) {
          window.treeView.draw(result.ast)  // 重绘语法分析树
        }
        this.codeHighlight(result.lexer, code_str)  // 重绘代码高亮
        for (const tid of ["goto", "action", "process"]) {
          $(`#${tid}-table`).html(this.tableHTML(result[tid]))  // 重绘各个表格
        }
      })
  }

  /**
   * @param {Array} token_list
   * @param {string} code
```

```
   */
  codeHighlight(token_list, code) {  // 根据 token 类型，为编辑器中
的字符赋予不同的 class ，并自动应用 CSS 样式
    const lines = code.split("\n")
    const getHightlightClass = (token) => {
      const c = []
      const ci = []
      if (token.prop === "unknown") {
        c.push(`cpf-unique-token-invalid`)
        c.push("cpf-invalid-token")
        ci.push("cpf-invalid-token")
      } else {
        if (!!this.drawAllToken) {
          c.push(`cpf-unique-token-${token.id}`)
          c.push("cpf-all-token")
        }
        if (/.*constant/.test(token.prop)) {
          ci.push(`cpf-token-constant`)
        } else if (/kw_.*/.test(token.prop)) {
          ci.push(`cpf-token-keyword`)
        } else if (/identifier.*/.test(token.prop)) {
          ci.push(`cpf-token-identifier`)
        } else {
          ci.push(`cpf-token-${token.prop}`)
        }
      }
      console.error(`${JSON.stringify(token.loc)}
"${token.content}" ==> ${c.join(" ") + ci.join(" ")}`)
      return {
        "inline": ci.join(" "),
        "normal": c.join(" ")
      }
    }
    this.editor.getModel().deltaDecorations(this.editor.getMode
l().getAllDecorations().map(d => d.id), [])  // 先删除旧的语法高亮
    this.editor.getModel().deltaDecorations(
      [],  //应用新的语法高亮
      token_list
        .map((token) => {
          // ensure token exists in code
          if (token.prop === "unknown") {
```

```
            console.error("unknown: ", JSON.stringify(token))
            token.content = "_"
          }
          const actualRow = lines[token.loc.row - 1]
          if (
            !!actualRow && (actualRow.substring(
              token.loc.col - 1,
              token.loc.col - 1 + token.content.length
            ) === token.content) || token.prop === "unknown"
          ) {
            return token  // 过滤语法分析器自行添加，而代码中不存在的
token
          }
          console.error(`ignored token :
${JSON.stringify(token)}`)
        })
        .filter(t => !!t)
        .map(token => {
          const classes = getHightlightClass(token)
          return {  // 生成 Decoration 对象，并且应用到编辑器上
            range: new monaco.Range(
              token.loc.row,
              token.loc.col,
              token.loc.row,
              token.loc.col + token.content.length
            ),
            options: {
              isWholeLine: false,
              className: classes["normal"],
              inlineClassName: classes["inline"],
            }
          }
        })
    )
  }

  flush() {  // 刷新语法分析结果
    this.callCompiler(this.editor.getModel().getValue())
  }

  setCompiler(compiler) {
```

```
    this.compiler = compiler
  }

  switchRightTab(right_id) {  // 用户界面右侧标签页切换
    const tab_bg = "cpf-active-tab"
    const it_to_display = document.getElementById(`${right_id}-
container`)
    if (!!it_to_display) {
      const right_items = document.getElementsByClassName("cpf-
right")
      for (const it of right_items) {
        it.style.display = "none"
      }
      it_to_display.style.display = "inline"
    }
    const tab_to_display =
document.getElementById(`${right_id}-tab-button`)
    if (!!tab_to_display) {
      const tabs = document.getElementsByClassName("cpf-tab-
button")
      for (const tab of tabs) {
        tab.classList.remove(tab_bg)
      }
      tab_to_display.classList.add(tab_bg)
    }
    {
      const zooms = document.getElementsByClassName("ast-zoom-
button")
      for (const zoomBtn of zooms) {
        zoomBtn.style.display = (right_id === "ast" ?
"inline" : "none")
      }
    }
  }

  tableHTML(arr) {  // 将表格绘制为 HTML table
    const HTML = []
    HTML.push("<table>")
    for (const [i, row] of arr.entries()) {
      HTML.push("<tr>")
      for (const [j, col] of row.entries()) {
```

```
        let style = ""
        if (j === 0 && i !== 0) {
          style = "position: sticky; left: 0; z-index: 2;"
        } else if (i === 0 && j !== 0) {
          style = "position: sticky; top: 0; z-index: 1;"
        } else if (i === 0 && j === 0) {
          style = "position: sticky; top: 0; left: 0; z-index:
3;"
        }
        HTML.push(`<td style="${style}">`)
        HTML.push(col)
        HTML.push(`</td>`)
      }
      HTML.push("</tr>")
    }
    HTML.push("</table>")
    return HTML.join("")
  }
}

window.cpf = new CPF()
window.cpf.switchRightTab('ast')
window.cpf.flush()
const channel = new QWebChannel(window.qt.webChannelTransport,
function (channel) {  // 在 QWebChannel 初始化完成后，执行回调，设置
compiler 对象
  window.cpf.setCompiler(channel.objects.compiler)
  window.cpf.flush()
});
```

4. 总结与展望

4.1. 总结

本项目采用 LR(1)分析方法，能够更好的减少冲突。实验中，通过读取配置文件中的语法产生式分析出相应的语法，通过读入的语法寻找 First 集、闭包、转移 GOs 表，以及 action 表和 goto 表。并且调用上次实验中完成的词

法分析器，对输入的程序进行解析，然后使用 action 表和 goto 表对其进行语法分析。最终能够得到移进-规约过程，并画出语法树。

本项目提供了对变量和函数声明的支持，包括类型、标识符、变量初始化、函数声明以及参数的定义等。定义了表达式的基本结构，包括运算符（加、减、乘、除等）和关系运算符，以及支持函数调用的表达式。提供了条件语句（if-else）、循环语句（while）、返回语句、赋值语句和函数调用语句。提供了定义了代码块的结构，包括内部声明和语句。提供了算符优先级处理，通过定义不同级别的表达式，实现了运算符的优先级处理，确保正确的表达式求值顺序。提供了函数调用和参数，允许函数的定义和调用，支持函数参数的传递。

本项目使用 PyQt5 进行前端可视化设计。我们创建了一个交互式的代码编辑框，编辑框拥有与 vscode 相似的功能，如语法高亮、批量修改、代码块折叠、输入联想等功能，使用户能够轻松输入代码。通过图形化展示语法分析树，并且对语法分析树部分做了单独的缩放按钮，使用户能够更加直观细致地了解代码的解析结构。在展示 Goto 表和 Action 表的同时，提供了一个移进规约过程表，帮助用户追踪 LR(1)自动机的状态变化。除此之外，我们还进行了错误处理，在词法错误时可以通过标红定位位置，在语法错误时可以通过右侧语法分析树报错进行处理。

通过这个项目，我们深刻理解了 LR(1)文法解析的原理和实现方法，并将其应用于一个实际项目。这不仅锻炼了我们的编程能力，也为我们今后深入学习编译原理和语法分析打下了坚实基础。在未来，我们会进一步优化解析器的性能，并考虑添加更多功能，以使其更加实用和全面。

4.2. 展望

在未来的课程学习与实践过程中，我们将进一步考虑对解析器的性能进行优化，包括提高解析速度、减少内存占用或针对大型代码文件的优化。性能优化可以通过改进算法、使用数据结构的高效实现以及进行代码审查来实现。除

此之外，我们还可以通过修改文法规范和相应的解析器实现从而支持更多的语法结构，或扩展现有文法以涵盖更广泛的语言特性。

5. 参考文献

[1] Anderson T, Eve J, Horning J J. Efficient LR (1) parsers[J]. Acta Informatica, 1973, 2: 12-39.

[2] Mickunas M D, Lancaster R L, Schneider V B. Transforming LR (k) grammars to LR (1), SLR (1), and (1, 1) Bounded Right-Context grammars[J]. Journal of the ACM (JACM), 1976, 23(3): 511-533.

[3] Korenjak A J. Efficient LR (1) processor construction[C]//Proceedings of the first annual ACM symposium on Theory of computing. 1969: 191-200.

[4] 张素琴，吕映芝. 编译原理[M]., 清华大学出版社

[5] 蒋立源、康慕宁等，编译原理（第 2 版）[M]，西安：西北工业大学出版社

[6] 陈火旺,刘春林,谭庆平,等.程序设计语言编译原理:第 3 版 [M].国防工业出版社, 2008

[7] 孙冀侠, 迟呈英, 李迎春. LR (1) 语法分析的自动构造[J]. 鞍山科技大学学报, 2003, 26(2): 90-92.

[8] Microsoft (2021). Monaco Editor. Microsoft. https://microsoft.github.io/monaco-editor/

[9] Muhammad Yahya. (2021). Dynamically Building Nested List from JSON Data and Tree View with CSS3. Medium. https://medium.com/oli-systems/dynamically-building-nested-list-from-json-data-and-tree-view-with-css3-2ee75b471744

[10] Mozilla Developer Network. (2018). QWebEngineView.html. Mozilla Developer Network. https://developer.mozilla.org/en-US/search?q=QWebEngineView.html

[11] 海轰 Pro.(2020). 编译原理学习笔记（十）~LR(1)分析. https://blog.csdn.net/weixin_44225182/article/details/105597613

[12] 肖鹏伟.(2019). 《编译原理》LR 分析法与构造 LR(1) 分析表的步骤. https://blog.csdn.net/qq_40147863/article/details/93253171