python实现语法分析大作业（含词法分析）实验报告

小组成员：2151130 童海博 2152214 赵克祥 2151140 王谦

目录

python实现语法分析大作业（含词法分析）实验报告
目录
一、需求分析
程序任务输入及其范围
输入的简单描述
输出形式
程序功能
测试数据
二、概要设计
任务分解
主要文件结构
数据类型定义
主程序流程与模块间的调用关系
三、详细设计
（1）词法分析`LexicalAnalysis`
（2）文法获取`Grammer`
（3）LR(1)文法分析`LR1Parser`
`LR1Item`类
`Action`类
`Closure`类
`move_closure`函数
`class LR1Parser`类
`construct_first_sets()`:
`cal_first(contents)`
`construct_closure(initial_closure)`
`construct_dfa(index)`
`construct_closures()`
`construct_action()`
`recognize(program)`
四、调试分析
两种不同的文法输入方式说明
测试数据一（单个正确文件输入）
输入
输出
测试数据二（单个错误文件）
输入
输出
测试数据三（同一文件夹下的一批文件）
输入
输出
调试问题思考与解决
五、总结与收获

收获
对课程的认识

一、需求分析

程序任务输入及其范围

输入的简单描述

输入为c语言测试文件`test.c`和LR(1)文法文件`grammar.txt`.

其中`test.c`是相对简单的c语言代码文件，相对灵活，可以自行更改测试代码；

举例`test.c`：

```
int main() {
    int i;
    int n;
    for (i = 0; i < n; i = i + 1) {
        int j;
        j = i;
    }
}
```

`grammar.txt`为我们小组经过不断地分析和测试最终构造得到的能够满足测试c语言的LR(1)文法，相对固定。

最终完成的`grammar.txt`展示：

```
program -> declaration_list

declaration_list -> declaration declaration_list | declaration

declaration -> var_declaration | fun_declaration

var_declaration -> type_specifier ID opt_init ; | type_specifier ID [ NUM ]
opt_init ; | type_specifier ID [ ] opt_init ; | type_specifier ID [ NUM ] ; |
type_specifier ID ;

opt_init -> = expression

type_specifier -> const simple_type | simple_type

fun_declaration -> type_specifier ID ( params ) compound_stmt | type_specifier ID
( ) compound_stmt

params -> param_list | void

param_list -> param | param_list , param

param -> type_specifier ID | type_specifier ID [ ] | type_specifier ID [ NUM ]
```

```
compound_stmt -> { block_items } | { }

block_items -> statement_list

statement_list -> statement statement_list | statement

statement -> var_declaration | expression_stmt | compound_stmt | selection_stmt |
iteration_stmt | return_stmt

selection_stmt -> if ( expression ) statement | if ( expression ) statement else
statement | else if ( expression ) statement

iteration_stmt -> while ( expression ) statement | for ( expression_stmt
expression_stmt expression ) statement | for ( expression_stmt expression_stmt
expression ) ;

return_stmt -> return ; | return expression ;

expression_stmt -> expression ;

expression -> simple_expression | var = expression | type_specifier ID opt_init

var -> ID | ID [ expression ]

simple_expression -> additive_expression relop additive_expression |
additive_expression

additive_expression -> term | additive_expression addop term

term -> factor | term mulop factor

factor -> ( expression ) | var | call | NUM | STRING | CH

call -> ID ( args ) | ID ( )

args -> arg_list

arg_list -> expression | arg_list , expression

simple_type -> void | float | double | int | long | char

addop -> + | -

mulop -> * | /

relop -> < | > | == | >= | <=
```

其中产生符号由右箭头`->`表示，不同的产生式由`|`分隔开，这种呈现格式和课程是保持一致的。算符之间都用空格分隔开，以作区分。

输出形式

既可以直接在调试窗口上直接打印出来，又可以将结果以文件的形式输出保存，语法树可以以图片方式生成。

程序功能

1. 能够读取单个或者批量读取测试文件进行处理
2. 能够根据c语言测试文件输出词法分析的结果，并能进一步利用词法分析结果进行语法分析
3. 能够得到ACTION、GOTO表
4. 能够得到各个项目集的闭包CLOSURE
5. 能够得到FIRST集
6. 能够逐步输出移进规约的识别过程
7. 能够最终判断测试文件是否符合文法能被正确识别
8. 能够得到语法树

测试数据

大致测试数据形式已经在上面给出，更多测试数据和测试过程将在第四模块调试分析进一步展示说明。

二、概要设计

任务分解

- 测试文件读取
- 词法分析
- 文法和产生式的获取和储存
- `ACTION` 和 `GOTO` 的产生和储存
- `ClOSURE` 的产生和储存
- `DFA` 的构造和储存
- `FIRST` 集的构造和储存
- 识别的实现与逐步表示
- 语法树的获取和表示

主要文件结构

- 文件夹`LexicalAnalysis`（词法分析部分）
  - `Lexer.py`（词法分析）
  - `SpecialTokens.py`（词法分析中的特殊关键词）
- 文件夹`Parser`（LR(1)文法分析部分）
  - `Grammer.py`（读取语法文件并储存）
  - `LR1Parser.py`（LR(1)文法分析）
  - `grammar.txt`（文法文件）
- 文件夹`prog`（输入的测试数据文件）

    - 文件夹`accepted`（能成功识别的正确测试用例）
    - 文件夹`error`（无法识别的错误测试用例）
- 文件夹`result`（测试输出结果）
    - `accepted.txt`（能成功识别的正确测试用例的输出结果）
    - `error.txt`（无法识别的错误测试用例的输出结果）
- `FileReader.py`（读取测试文件）
- `Compiler.py`（对特定文法和特定测试文件整合调用代码实现整体功能）
- `playground.py`（选择调用的语法文件和测试文件）

数据类型定义

1. `Token` **类：**
    - 表示识别到的令牌。
    - 属性：
        - `value`：令牌的值，即具体的词法单元。
        - `token_type`：令牌的类型，标识该令牌是标识符、关键字、运算符等。
    - 方法：
        - `__init__(self, value, token_type)`：构造方法，初始化 `Token` 对象。
        - `__str__(self)`：返回包含令牌值和类型的字符串表示。
2. `ErrorCode` **类：**
    - 表示词法分析中的错误。
    - 属性：
        - `position`：错误位置。
        - `value`：错误编码。
        - `error_type`：错误类型，例如未知字符、不完整字符串等。
    - 方法：
        - `__init__(self, position, value, error_type)`：构造方法，初始化 `ErrorCode` 对象。
        - `__str__(self)`：返回包含错误位置、错误编码和错误类型的字符串表示。
3. `Lexer` **类：**
    - 词法分析器类。
    - 属性：
        - `source_code`：源代码字符串。
        - `tokens`：存储识别到的令牌的列表。
        - `current_position`：当前处理的位置。
        - `mode`：表示词法分析的模式，用于处理字符串、字符、注释等情况。
    - 方法：
        - `__init__(self, source_code)`：构造方法，初始化 `Lexer` 对象。
        - `tokenize(self)`：对源代码进行词法分析，识别并存储令牌。
        - `_identify_identifier(self)`：识别标识符或关键字。
        - `_identify_number(self)`：识别数字常量。
        - `_identify_operator(self)`：识别运算符。
        - `_identify_delimiter(self)`：识别分隔符。

        - `_identify_unknown(self)`: 识别未知字符。
        - `_identify_string(self)`: 识别字符串。
        - `_identify_char(self)`: 识别字符。
        - `_identify_comment(self)`: 识别注释。
4. `LR1Item` **类:**
    - 用于表示LR(1)项。
    - 属性:
        - `lhs`: 左侧（前缀）。
        - `rhs`: 右侧（产生式的右侧）。
        - `lookahead`: 前瞻符号。
        - `dot_index`: 点的位置。
    - 方法:
        - `__str__`: 返回LR(1)项的字符串表示。
        - `__eq__`: 判断两个LR(1)项是否相等。
        - `__hash__`: 用于在集合中正确处理LR(1)项的哈希。
5. `Action` **类:**
    - 用于表示LR(1)分析表中的动作。
    - 属性:
        - `action`: 动作类型，包括 "move"、"reduce"、"accept"。
        - `state`: 下一个状态的位置。
        - `production`: 归约时使用的产生式。
    - 方法:
        - `__str__`: 返回动作的字符串表示。
6. `Closure` **类:**
    - 用于表示LR(1)项集合的闭包。
    - 属性:
        - `items`: 包含的LR(1)项列表。
    - 方法:
        - `__str__`: 返回闭包的字符串表示。
        - `__eq__`: 判断两个闭包是否相等。
        - `__hash__`: 用于在集合中正确处理闭包的哈希。
7. `LR1Parser` **类:**
    - 用于实现LR(1)分析器。
    - 属性:
        - `rules`: 文法规则。
        - `start_symbol`: 文法的起始符号。
        - `non_terminals`: 非终结符集合。
        - `terminals`: 终结符集合。
        - `action`: LR(1)分析表中的动作。
        - `goto`: LR(1)分析表中的goto表。
        - `closures`: 存储所有闭包的列表。
        - `first_sets`: 存储First集合的字典。
    - 方法:

        - `construct_first_sets`: 构造First集合。
        - `construct_closure`: 构造闭包。
        - `cal_first`: 计算First集合。
        - `construct_dfa`: 构造DFA。
        - `construct_closures`: 构造所有闭包。
        - `construct_action`: 构造LR(1)分析表中的动作。
        - `construct_all`: 一次性构造所有内容。
        - `save_to_file` 和 `load_from_file`: 保存和加载分析器状态。
        - `print_action` 和 `print_closure`: 打印LR(1)分析表和闭包集合。
        - `recognize`: 使用LR(1)分析器识别给定的程序。
8. `FileReader` **类:**
    - 用于从文件中读取源代码。
    - 属性:
        - `filename`: 文件名。
        - `data`: 文件内容。
    - 方法:
        - `__init__`: 初始化对象。
        - `__str__`: 返回文件内容的字符串表示。
        - `content`: 返回文件内容。
9. `Grammer` **类:**
    - 用于表示文法规则和生成语法树。
    - 属性:
        - `rules`: 一个字典，表示文法规则。键是产生式的左侧（非终结符），值是一个包含右侧产生式的列表。
        - `non_terminal`: 一个列表，包含所有的非终结符。
        - `terminal`: 一个列表，包含所有的终结符。
    - 方法:
        - `__init__(self, filename)`: 构造方法，从文件中加载文法规则，初始化 `rules`、`non_terminal` 和 `terminal`。
        - `add_edges(self, graph, parent, children_lists)`: 递归地向图中添加边，用于构建语法树。
        - `generate_syntax_tree(self, save_filename)`: 生成语法树，并将其保存为图像文件。
10. `FileReader` **类:**

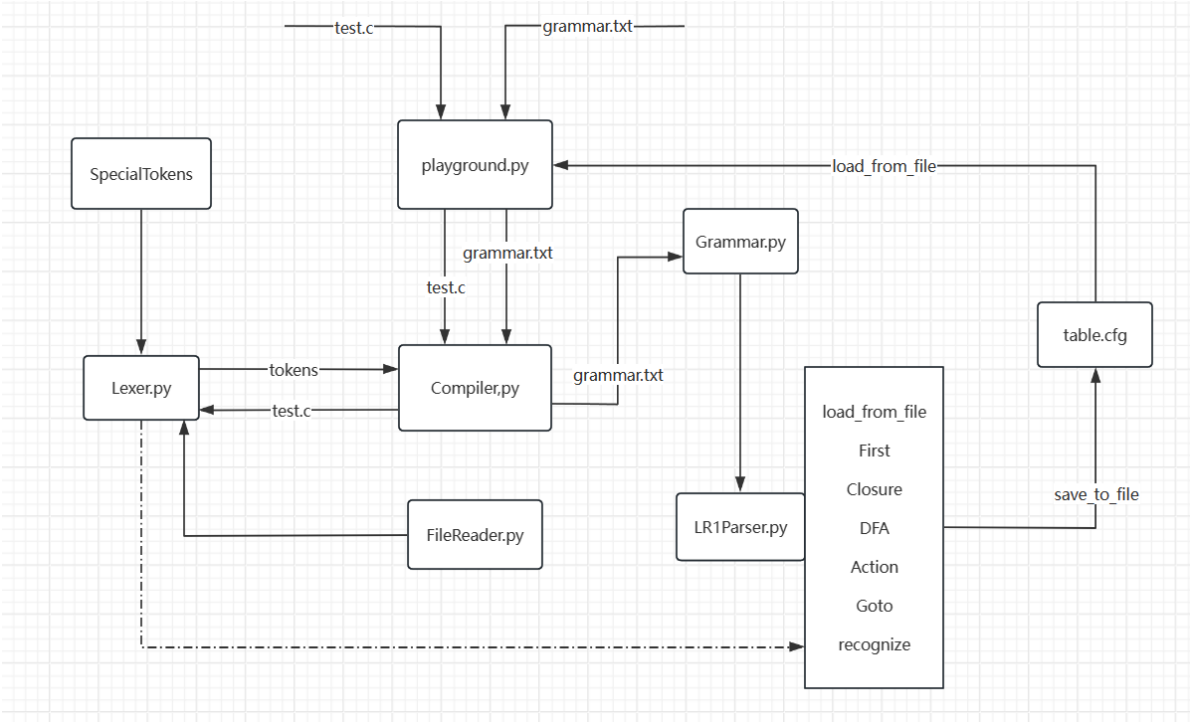
- 用于读取文件内容的类。
- 属性:
    - `filename`: 文件名，表示要读取的文件的路径。
    - `data`: 字符串，存储文件的内容。
- 方法:
    - `__init__(self, filename)`: 构造方法，初始化 `FileReader` 对象。
    - `__str__(self)`: 返回包含文件内容的字符串表示。
    - `content(self)`: 返回文件内容的字符串。

11. `Compiler` **类:**
    - 用于编译源代码的类。

- 属性：
  - `grammer`：一个 `Grammer` 对象，表示文法规则。
  - `lr1parser`：一个 `LR1Parser` 对象，用于进行 LR(1) 文法分析。
- 方法：
  - `__init__(self, grammer_filename, start_symbol="program", load_from_file=None)`：构造方法，初始化 `Compiler` 对象。
  - `recognize(self, filename)`：识别单个源代码文件。
  - `recognize_files(self, folder_path, output_file)`：批量识别指定文件夹中的源代码文件。

主程序流程与模块间的调用关系

简单的流程和调用关系如图所示，详细请参考源代码和第三模块的详细设计



三、详细设计

(1) 词法分析`LexicalAnalysis`

具体分析前先对c语言的关键词和符号做特殊定义，存放在`SpecialTokens.py`中

```
KEYWORDS = ["int", "char", "define", "struct", "double", "if", "for", "return",
"while",
            "else", "float", "case", "switch", "return", "void", "long", "auto",
"const", "default"]
DELIMITERS = [",", ";", "(", ")", "{", "}", "[", "]", "'", '"']
OPERATORS = ["+", "-", "*", "/", ">", "<", "="]
```

负责词法分析主要功能的是`Lexer.py`

对词法分析的表示形式如下所示，`value`为内容，`token_type`为类型标注

```
class Token:
    def __init__(self, value, token_type):
        self.value = value
        self.token_type = token_type

    def __str__(self):
        return f"Token({self.value}, {self.token_type})"
```

词法分析大部分情况是从头扫到尾，期间进行分析判断并记录。

但是此处对注释、字符串等相对特殊的内容使用了一个`mode`作为标志，用于辅助读取舍弃和判断

```
class Lexer:
    def __init__(self, source_code):
        self.source_code = source_code
        self.tokens = []
        self.current_position = 0
        self.mode = 0

    def tokenize(self):
        while self.current_position < len(self.source_code):
            # 分几种状态:
            # 常态    mode = 0
            # 读到 "  mode = 1
            # 读到 '  mode = 2
            # 读到 // mode = 3
            # 读到 /* mode = 4
            ……
```

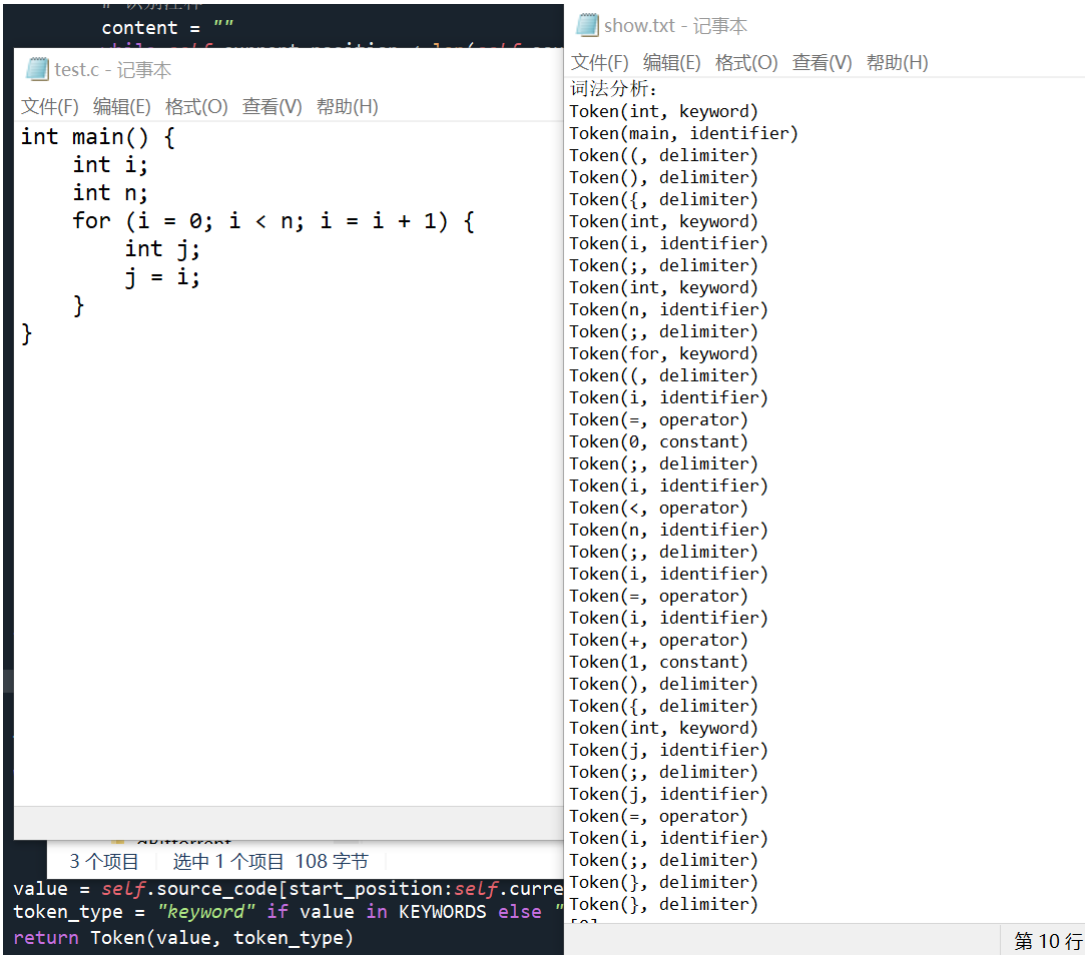
例如读到`"`就进入`mode = 1`，知道再次遇到`"`才认为字符串读完并恢复。

另外还有包括`SpecialTokens`在内的一些特殊判断

```
def _identify_identifier(self):
        start_position = self.current_position
        while (self.current_position < len(self.source_code) and
               (self.source_code[self.current_position].isalnum() or
                self.source_code[self.current_position] == '_')):
            self.current_position += 1

        value = self.source_code[start_position:self.current_position]
        token_type = "keyword" if value in KEYWORDS else "identifier"  # 使用
KEYWORDS
        return Token(value, token_type)

    def _identify_number(self):
        start_position = self.current_position
        while self.current_position < len(self.source_code) and
self.source_code[self.current_position].isdigit():
```

```
            self.current_position += 1

        value = self.source_code[start_position:self.current_position]
        return Token(value, "constant")

    def _identify_operator(self):
        op = self.source_code[self.current_position]
        self.current_position += 1
        return Token(op, "operator")

    def _identify_delimiter(self):
        delim = self.source_code[self.current_position]
        self.current_position += 1
        return Token(delim, "delimiter")

    def _identify_unknown(self):
        content = self.source_code[self.current_position]
        self.current_position += 1
        return Token(content, "unknown")
```

可以通过简单的输出语句查看结果



(2) 文法获取`Grammer`

此前展示并介绍过我们文法的格式，根据该格式进行读取，依次依照 回车、`->`、`|`和空格 进行拆分，除了分为左右两边，还用如下属性来进行分析判断：

- `rules`：存储文法规则的字典，以非终结符为键，对应的产生式列表为值。
- `non_terminal`：存储非终结符的列表。
- `terminal`：存储终结符的列表。

只在右侧出现而不在左侧出现的是终结符，出现在左侧的是非终结符，依次遍历分出，期间做好重复处理。

```
lines = grammar_content.split("\n")
        for line in lines:
            if "->" in line:
                # 分离产生式的左侧和右侧
                left, right = line.split("->")
                left = left.strip()
                right = [r.strip() for r in right.split("|")]
                if left in self.rules:
                    self.rules[left].extend(right)
                else:
                    self.rules[left] = right
        # 进一步根据空格拆分
        for key in self.rules.keys():
            content = []
            for state in self.rules[key]:
                states = state.split(" ")
                content.append(states)
            self.rules[key] = content
        for key in self.rules.keys():
            if key not in self.non_terminal:
                self.non_terminal.append(key)
        for key in self.rules.keys():
            for states in self.rules[key]:
                for state in states:
                    if state not in self.non_terminal and state not in
self.terminal:
                        self.terminal.append(state)
```

同时，在`Grammer.py`中也添加了生成语法树的功能

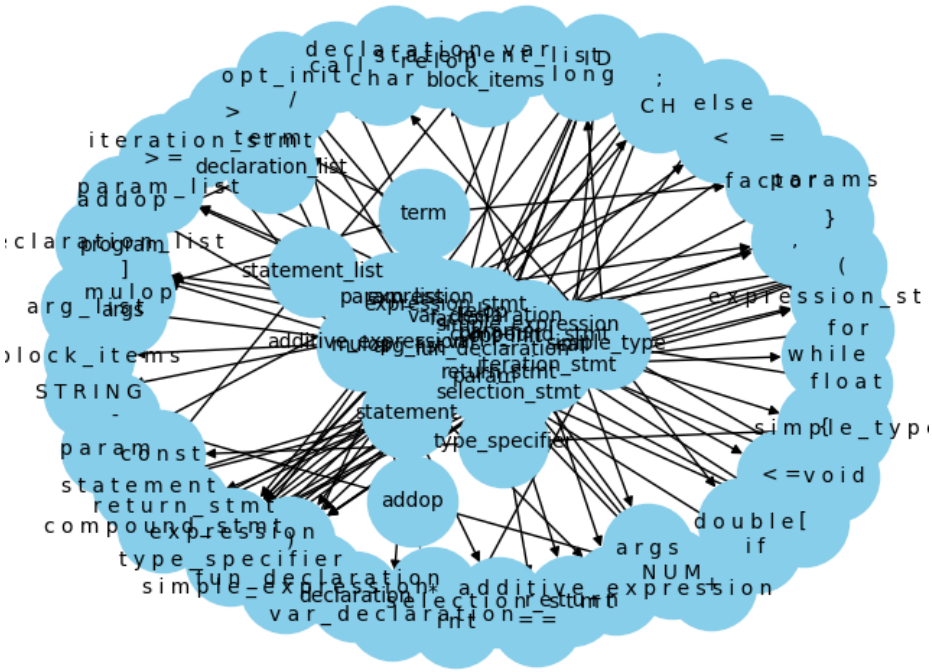
1. `add_edges`：
    - 用于向图中添加边，构建语法树的边。
        - `graph`：图对象，用于构建语法树的有向图。
        - `parent`：当前节点的标签，表示产生式左侧的非终结符。
        - `children_lists`：子节点列表，表示产生式右侧的可能展开。
    - 思路：
        - 遍历 `children_lists` 中的每个子节点，将其与父节点连接起来。
        - 如果子节点是非终结符（存在于语法规则中），则递归调用 `add_edges` 方法，进一步添加其子节点。

2. `generate_syntax_tree`：
    - 用于生成并保存语法树的图像。
    - 参数：
        - save_filename`：保存语法树图像的文件名。
    - 思路：
        - 创建一个有向图对象 `G`。
        - 遍历语法规则中的每个产生式，调用 `add_edges` 方法将产生式的左侧和右侧子节点添加到图中。
        - 使用 `NetworkX` 库中的布局算法计算节点的布局（`spring_layout`）。
        - 使用 `NetworkX` 绘图函数绘制有向图，设置节点标签、箭头等属性。
        - 将绘制的图保存到指定的文件中。

这两个方法共同作用，通过逐步添加边和节点，最终构建并保存了整个语法树的图像。在 `generate_syntax_tree` 方法中，`add_edges` 方法被递归调用以处理语法树的层次结构。

```
def add_edges(self, graph, parent, children_lists):
        for children in children_lists:
            for child in children:
                graph.add_edge(parent, ' '.join(child))
                if ' '.join(child) in self.rules:
                    self.add_edges(graph, ' '.join(child), self.rules['
'.join(child)])

    def generate_syntax_tree(self, save_filename):
        G = nx.DiGraph()
        for parent, children_lists in self.rules.items():
            self.add_edges(G, parent, children_lists)

        pos = nx.spring_layout(G)
        nx.draw(G, pos, with_labels=True, arrows=True, node_size=2000,
node_color="skyblue", font_size=10)
        plt.savefig(save_filename)
```

需要注意的是，对于比较复杂的内容，生成的语法树的图并不太令人满意



(3) LR(1)文法分析`LR1Parser`

`LR1Item`类

LR1Item表示项目，包括 左侧、右侧、点的位置 和 向前看一位，自身结构也可以从代码中看出

```
class LR1Item:
    def __init__(self, lhs, rhs, lookahead, dot_index=0):
        self.lhs = lhs  # 前
        self.rhs = rhs  # 转移状态
        self.lookahead = lookahead  # 前瞻一位
        self.dot_index = dot_index  # 点的位置

    def __str__(self):
        production = ""
        for i in self.rhs:
            production += i + " "
        function = self.lhs + "->" + production
        return f"{function}, dot_index:{self.dot_index}, lookahead:
{self.lookahead}"
```

`__eq__(self, other)`用来比较LR1Item是否和其他重复。

`Action`类

- `action`：表示动作的类型，可以是 "move"、"reduce" 或 "accept"。
- `state`：在状态转移中，表示下一个状态的位置。
- `production`：在规约中，表示使用的产生式。

用于表示 LR 分析表中的动作，根据不同的动作类型，实例中的属性会有不同的值。例如，如果动作是移进，则 `action` 为 "move"，`state` 表示下一个状态的位置；如果动作是规约，则 `action` 为 "reduce"，`production` 表示使用的产生式。

```
class Action:
    def __init__(self, action=None, state=None, production=None):
        """
        :param action: “move”, "reduce", "accept"
        :param state: 位置
        :param production: 产生式
        """
        self.action = action
        self.state = state
        self.production = production

    def __str__(self):
        content = ""
        if self.action is not None:
            content += f"action:{self.action}, "
        if self.state is not None:
            content += f"state:{self.state}"
        if self.production is not None:
            content += f"production:{self.production}"
        return content
```

这里的`action`和课上的功能是十分接近的，与课件的前三个相对应

- **动作ACTION和状态转换GOTO构造如下：**
  1. **若项目[A→α·aβ, b]属于I_k且GO(I_k, a) = I_j， a为终结符，则置ACTION[k, a]为 “ “sj”**
  2. **若项目[A→α·， a]属于I_k，则置ACTION[k, a]为 “rj”；其中假定A→α为文法G′的第j个产生式。**
  3. **若项目[S′→S·, #]属于I_k，则置ACTION[k, #]为 “acc”。**
  4. **若GO(I_k，A) = I_j，则置GOTO[k, A]=j。**
  5. **分析表中凡不能用规则1至4填入信息的空白栏均填上 “出错标志” 。**

`Closure` 类

构造

```
pythonCopy codedef __init__(self, items):
    self.items = items
```

- `items`：项集合，包含多个 `LR1Item` 对象。

`__str__`

```
pythonCopy codedef __str__(self):
    content = ""
    for item in self.items:
        content += str(item) + "\n"
    return content
```

返回闭包的字符串表示，其中包含闭包中每个项的字符串表示，每个项之间用换行符分隔。

`__eq__`

```
pythonCopy codedef __eq__(self, other):
    if not isinstance(other, Closure):
        return False
    # 比较两个闭包是否包含相同的项（忽略顺序）
    return set(self.items) == set(other.items)
```

用于判断两个闭包是否相等。比较两个闭包的项集合是否相同（不考虑顺序）。

`__hash__`

```
pythonCopy codedef __hash__(self):
    # 计算基于项集合的哈希值
    return hash(tuple(sorted(self.items, key=lambda item: hash(item))))
```

返回基于闭包的哈希值，通过将项集合排序后计算哈希值。这样可以保证相同的项集合具有相同的哈希值，从而用于在集合中进行比较和查找。

`move_closure` 函数

```
pythonCopy codedef move_closure(initial_closure, symbol):
    new_closure = Closure([])
    for item in initial_closure.items:
        index = item.dot_index
        if index + 1 > len(item.rhs):
            continue
        current_char = item.rhs[index]
        if current_char != symbol:
            continue
        new_item = LR1Item(item.lhs, item.rhs, item.lookahead, index + 1)
        if new_item not in new_closure.items:
            new_closure.items.append(new_item)
    return new_closure
```

`initial_closure`: 要进行转移的初始 LR(1) 闭包，是 `Closure` 类的一个实例。

`symbol`: 转移的符号。

返回一个新的 `Closure` 实例，表示通过在项的 rhs 中找到指定符号进行转移后的闭包。

思路

1. 创建一个新的 `Closure` 实例 `new_closure`，用于存储转移后的项集。
2. 遍历 `initial_closure` 中的每个项。

3. 对于每个项，检查 `dot_index` 是否小于 `rhs` 的长度，如果大于等于，说明已到达产生式末尾，无法再进行转移，跳过当前项。
4. 获取当前项 `rhs` 上 `dot_index` 位置的符号 `current_char`。
5. 如果 `current_char` 不等于给定的符号 `symbol`，说明无法进行转移，跳过当前项。
6. 创建一个新的 `LR1Item` 实例 `new_item`，表示转移后的项，`dot_index` 加一。
7. 将新项 `new_item` 添加到 `new_closure` 中，确保不重复。
8. 返回新的 `Closure` 实例 `new_closure`。

`class LR1Parser`类

在前面的基础上，可以构建`LR1Parser`来进一步执行并实现对LR(1)文法的识别。

首先看一下各个属性

```
class LR1Parser:
    def __init__(self, rules, non_terminals, terminals, start_symbol,
load_from_file=None):
        self.rules = rules
        self.start_symbol = start_symbol
        self.non_terminals = non_terminals
        self.terminals = terminals
        self.action = {}
        self.goto = {}
        self.closures = []
        self.first_sets = {}

        if load_from_file:
            self.load_from_file(load_from_file)
        else:
            self.construct_all()
```

- `rules`: 存储文法规则的字典。
- `start_symbol`: 文法的开始符号。
- `non_terminals`: 文法中的非终结符集合。
- `terminals`: 文法中的终结符集合。
- `action`: 存储 LR(1) 分析表的字典，用于保存移进、规约和接受动作。
- `goto`: 存储 LR(1) 分析表的字典，用于保存状态之间的转移关系。
- `closures`: 存储 LR(1) 项集族的列表。
- `first_sets`: 存储文法符号的 First 集合。

`construct_first_sets()`:

构造文法符号的 First 集合

1. 初始化一个字典 `first_sets`，用于存储文法符号的 First 集合，其中每个非终结符对应一个空集合。
2. 将终结符和 `$` 加入其自身的 First 集合中。
3. 进入循环，直到没有新的 First 集合被更新（`modified` 变量为 False）。
4. 对于每个产生式（production），遍历产生式右侧的符号，将其 First 集合加入到产生式左侧符号的 First 集合中。

5. 如果产生式右侧的符号包含空串（'empty'），则继续遍历下一个符号，直到找到不包含空串的符号为止。
6. 如果产生式右侧的所有符号都包含空串，将空串加入到产生式左侧符号的 First 集合中。
7. 检查是否有新的 First 集合被更新，如果有更新，将 `modified` 设置为 True。
8. 如果没有新的 First 集合被更新，跳出循环。

```
def construct_first_sets(self):
        self.first_sets = {non_terminal: set() for non_terminal in
self.non_terminals}
        for terminal in self.terminals:
            self.first_sets[terminal] = {terminal}
            self.first_sets["$"] = {"$"}
        while True:
            modified = False

            for lhs, productions in self.rules.items():
                for production in productions:
                    first_before_update = len(self.first_sets[lhs])
                    for symbol in production:
                        self.first_sets[lhs] |= (self.first_sets[symbol] -
{'empty'})
                        if 'empty' not in self.first_sets[symbol]:
                            break
                    else:
                        self.first_sets[lhs].add('empty')
                    if first_before_update != len(self.first_sets[lhs]):
                        modified = True
            if not modified:
                break
```

`cal_first(contents)`

计算文法符号串的 First 集合

1. 初始化一个空的集合 `first_set`，用于存储计算得到的 First 集合。
2. 对于每个符号 `content`，执行以下步骤：
    - 将 `content` 对应的 First 集合中除了空符号（"empty"）之外的所有元素添加到 `first_set` 中。
    - 如果 "empty" 不在 `content` 的 First 集合中，跳出循环。否则，继续下一个符号的处理。
3. 如果所有符号的 First 集合中都包含空符号 "empty"，则将 "empty" 添加到最终的 `first_set` 中。
4. 返回最终计算得到的 First 集合 `first_set`。

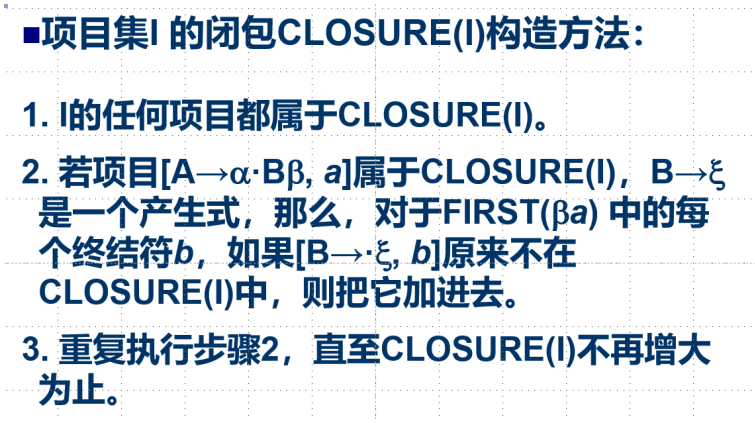
```
def cal_first(self, contents):
        first_set = set()
        for content in contents:
            first_set |= (self.first_sets[content] - {"empty"})
            if "empty" not in content:
                break
        else:
            first_set.add("empty")
        return first_set
```

`construct_closure(initial_closure)`

构造 LR(1) 闭包

1. 初始化一个新的闭包 `new_closure`，将传入的初始闭包 `initial_closure` 中的所有项添加到新闭包中。
2. 进入循环，直到闭包中的项数不再增加（`len(new_closure.items) == present_length`）。
3. 遍历新闭包中的每个项，如果项的点位置已经在产生式的最右端（`item.dot_index >= len(item.rhs)`），则跳过该项。
4. 获取当前项的点后的符号 `present_char`。
5. 如果 `present_char` 是非终结符，则计算点后符号的 First 集合，并将产生式的右侧剩余部分和向前看符号添加到新的 LR(1) 项中。
6. 遍历非终结符 `present_char` 的每个产生式，为其 First 集合中的每个符号创建一个新的 LR(1) 项，并将其添加到新闭包中。
7. 如果新闭包中的项数没有增加，表示没有新的项可以添加，跳出循环。

```
def construct_closure(self, initial_closure):
        new_closure = Closure([])
        for item in initial_closure.items:
            new_closure.items.append(item)
        while True:
            present_length = len(new_closure.items)
            for item in new_closure.items:
                if item.dot_index >= len(item.rhs):
                    continue
                present_char = item.rhs[item.dot_index]
                if present_char in self.non_terminals:
                    supervised_content = [] if item.dot_index + 1 >=
len(item.rhs) \
                        else item.rhs[item.dot_index + 1:]
                    supervised_content.append(item.lookahead)
                    first = self.cal_first(supervised_content)
                    for production in self.rules[present_char]:
                        for first_char in first:
                            new_item = LR1Item(present_char, production,
first_char, 0)
                            if new_item not in new_closure.items:
                                new_closure.items.append(new_item)
            if len(new_closure.items) == present_length:
                return new_closure
```



`construct_dfa(index)`

构造 LR(1) 项集族的 DFA

1. 检查给定的 `index` 是否已经在 `self.goto` 中，如果是，则说明对应的项集已经处理过，直接返回。
2. 获取当前项集 `current_closure`。
3. 初始化一个列表 `used_char`，用于追踪已经处理过的字符。
4. 遍历 `current_closure` 中的每个项，获取当前项的 `dot_index` 和 `current_char`。
5. 如果 `dot_index` 大于等于当前项的产生式长度，说明点已经到达产生式末尾，直接跳过。
6. 如果 `current_char` 已经处理过，说明相同的字符已经在当前项集的其他项中处理过，直接跳过。
7. 将 `current_char` 添加到 `used_char` 中，表示已经处理过。
8. 调用 `move_closure` 方法，根据当前项和字符 `current_char` 计算移动后的闭包。
9. 调用 `construct_closure` 方法，构建新的闭包，即计算闭包的闭包。
10. 如果新的闭包不在 `self.closures` 中，说明是新的项集，进行以下处理：
    - 将新的闭包添加到 `self.closures` 中。
    - 获取新的闭包的索引 `new_index`。
    - 如果 `index` 不在 `self.goto` 中，初始化 `self.goto[index]`。
    - 在 `self.goto[index]` 中添加新的转移，字符为 `current_char`，目标项集的索引为 `new_index`。
    - 递归调用 `construct_dfa`，处理新的项集。
11. 如果新的闭包已经在 `self.closures` 中，说明这个项集已经处理过，进行以下处理：
    - 如果 `index` 不在 `self.goto` 中，初始化 `self.goto[index]`。
    - 获取新的闭包的索引 `new_index`。
    - 在 `self.goto[index]` 中添加新的转移，字符为 `current_char`，目标项集的索引为 `new_index`。

```
def construct_dfa(self, index):
        if index in self.goto.keys():
            return
        current_closure = self.closures[index]
        used_char = []
        for item in current_closure.items:
            dot_index = item.dot_index
```

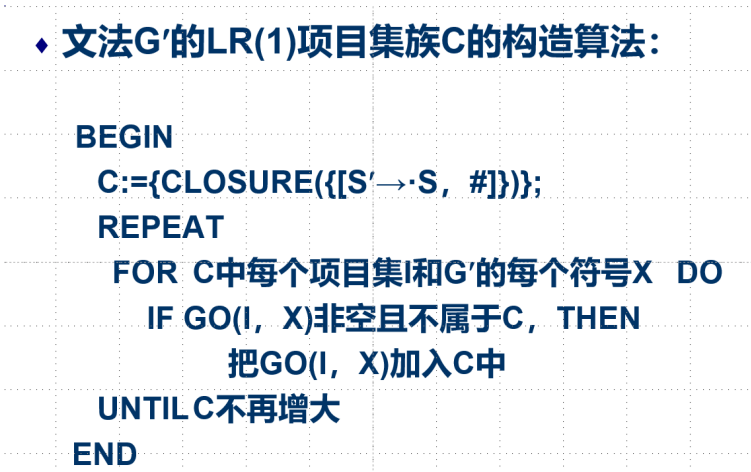
```
            if dot_index >= len(item.rhs):
                continue
            current_char = item.rhs[dot_index]
            if current_char in used_char:
                continue
            used_char.append(current_char)
            new_closure = move_closure(current_closure, current_char)
            new_closure = self.construct_closure(new_closure)
            if new_closure not in self.closures:
                self.closures.append(new_closure)
                new_index = len(self.closures) - 1
                if index not in self.goto.keys():
                    self.goto[index] = {}
                self.goto[index][current_char] = new_index
                self.construct_dfa(new_index)
            else:
                if index not in self.goto.keys():
                    self.goto[index] = {}
                new_index = self.closures.index(new_closure)
                self.goto[index][current_char] = new_index
```

`construct_closures()`

构造 LR(1) 项目集族

1. 创建初始项 `initial_item`，它表示 LR(1) 项目集的第一个项，其中的 `LR1Item` 实例包含了起始符号、起始规则、`$` 符号（表示输入串的结束）以及点的位置（初始为0）。
2. 创建初始闭包 `initial_closure`，它是包含 `initial_item` 的 `Closure` 实例。
3. 将初始闭包 `initial_closure` 添加到 `self.closures` 中，即项目集族的列表中。
4. 调用 `construct_dfa(0)`，从初始项集开始构建 LR(1) 项目集族的 DFA，这个过程会根据每个项目集的闭包和字符的转移关系，递归地构建有限自动机。

```
def construct_closures(self):
        initial_item = LR1Item(self.start_symbol, self.rules[self.start_symbol]
[0], "$", 0)
        initial_closure = Closure([initial_item])
        self.closures.append(self.construct_closure(initial_closure))
        self.construct_dfa(0)
```



`construct_action()`

构造 LR(1) 分析表中的动作部分

1. 对于每个项集（状态），遍历其中的每个项。
2. 如果某个项的点已经到达产生式的末尾（`dot_index >= len(item.rhs)`），表示可以规约，判断是接受状态还是规约状态：
    - 如果当前项的前瞻符号（`item.lookahead`）是 `$` 且左侧符号（`item.lhs`）是起始符号（`self.start_symbol`），则将动作设置为接受（`action="accept"`）。
    - 否则，将动作设置为规约（`action="reduce"`），并记录规约的产生式信息（左侧符号和右侧符号）。
3. 如果点没有到达产生式的末尾，表示可以进行移进操作，对于当前项的下一个字符（`current_char`），获取对应的下一个状态（`new_index`），将动作设置为移进（`action="move"`），并记录下一个状态的信息。
4. 将构建好的动作表添加到 `self.action` 中，其中键是项集的索引，值是一个字典，表示在该项集下的各个前瞻符号对应的动作。

```
def construct_action(self):
        for index, closure in enumerate(self.closures):
            if index not in self.action:
                self.action[index] = {}
            for item in closure.items:
                dot_index = item.dot_index
                if dot_index >= len(item.rhs):
                    if item.lookahead == "$" and item.lhs == self.start_symbol:
                        action = Action(action="accept")
                        self.action[index][item.lookahead] = action
                    else:
                        action = Action(action="reduce", production={
                            "lhs": item.lhs,
                            "rhs": item.rhs
                        })
                        self.action[index][item.lookahead] = action
                    continue
                current_char = item.rhs[dot_index]
                if current_char in self.terminals:
                    new_index = self.goto[index][current_char]
                    action = Action(action="move", state=new_index)
                    self.action[index][current_char] = action
```

`recognize(program)`

LR(1) 分析器的主要识别函数，用于识别输入程序是否符合文法规则。其执行过程如下：

1. 初始化状态栈 `state_stack` 和符号栈 `symbol_stack`，并将起始状态 `0` 和结束符 `$` 入栈。
2. 循环执行以下步骤，直到识别结束：
    - 获取当前输入符号（`current_char`）和符号类型（`current_type`），如果已经到达程序末尾，则用 `$` 表示。
    - 如果当前符号类型为注释（`"note"`），则跳过当前符号。

- 尝试获取当前状态栈的栈顶状态（`top_state`）对应于当前输入符号的动作（`action`）。
- 根据动作执行相应的操作：
  - 如果动作是移进（`"move"`），将下一个状态（`action.state`）入栈，将当前符号入栈，继续处理下一个符号。
  - 如果动作是规约（`"reduce"`），根据规约的产生式将状态栈和符号栈出栈，将规约的非终结符入栈，继续处理当前输入符号。
  - 如果动作是接受（`"accept"`），表示输入符号串符合文法规则，返回 `True`。
  - 如果动作无法匹配，返回 `False` 表示识别失败。
- 更新输入符号位置（`index`）。

3. 如果循环结束仍未接受，返回 `False` 表示识别失败。

```
def recognize(self, program):
        state_stack = [0]
        symbol_stack = ["$"]
        index = 0
        while True:
            print(state_stack)
            print(symbol_stack)
            if index >= len(program):
                current_char = "$"
                current_type = "EOF"
            else:
                current_token = program[index]
                current_char = current_token.value
                current_type = current_token.token_type
            top_state = state_stack[-1]
            if current_type == "note":
                index += 1
                continue
            try:
                if current_type == "identifier":
                    action = self.action[top_state]["ID"]
                    current_char = "ID"
                elif current_type == "string":
                    action = self.action[top_state]["STRING"]
                    current_char = "STRING"
                elif current_type == "ch":
                    action = self.action[top_state]["CH"]
                elif current_type == "constant":
                    action = self.action[top_state]["NUM"]
                    current_char = "NUM"
                else:
                    action = self.action[top_state][current_char]
            except:
                print(top_state)
                print(current_char)
                return False
            if action.action == "move":
                next_index = action.state
                state_stack.append(next_index)
                symbol_stack.append(current_char)
            elif action.action == "reduce":
                production = action.production
```

```
                production_length = len(production["rhs"])
                state_stack = state_stack[:-production_length]
                symbol_stack = symbol_stack[:-production_length]
                new_top_state = state_stack[-1]
                next_index = self.goto[new_top_state][production["lhs"]]
                state_stack.append(next_index)
                symbol_stack.append(production["lhs"])
                index -= 1
            elif action.action == "accept":
                return True
            else:
                return False
            index += 1
        return False
```

四、调试分析

写在之前：

实际上正式输出的文本内容中仅展示逐步的移进规约识别过程；另外也可以选择输出语法分析树的图片，当然面对复杂文法这个的效果不太令人满意。

而在本篇中展示的文件中的其他内容诸如词法分析结果、closure闭包、action、goto、first集等等，均为撰写报告时额外添加了部分代码来作展示用，仅展示功能的实现，其实效果并不理想，实际上真正运行提交代码时结果会更干净。

两种不同的文法输入方式说明

1. **根据LR(1)分析方法，编写一个类C语言的语法分析程序，可以选择以下两项之一作为分析算法的输入：**

   （1）直接输入根据已知文法人工构造的ACTION表和GOTO表。

   （2）输入已知文法，由程序自动生成该文法的ACTION表和GOTO表。

本次大作业以第二种输入文法为主，文法本身相对固定，存放在`Parser`文件夹的`grammar.txt`中，之前已经介绍过了，不再赘述。

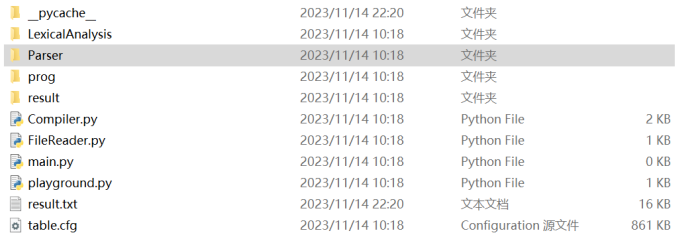
同时我们在此基础上也兼顾了第一种输入`ACTION`表和`GOTO`表的功能实现，这部分和之前的代码分析分开来，只在这里作思路上的说明

```
class LR1Parser:
    def __init__(self, rules, non_terminals, terminals, start_symbol,
load_from_file=None):
        self.rules = rules
        self.start_symbol = start_symbol
        self.non_terminals = non_terminals
        self.terminals = terminals
        self.action = {}
        self.goto = {}
        self.closures = []
        self.first_sets = {}

        if load_from_file:
            if not os.path.exists(load_from_file):
                self.construct_all()
```

```
                self.save_to_file(load_from_file)
            self.load_from_file(load_from_file)
        else:
            self.construct_all()

    def save_to_file(self, filename):
        data = {
            "action": self.action,
            "goto": self.goto,
            "closures": self.closures,
            "first_sets": self.first_sets
        }
        with open(filename, "wb") as f:
            pickle.dump(data, f)

    def load_from_file(self, filename):
        with open(filename, "rb") as f:
            data = pickle.load(f)
        self.action = data["action"]
        self.goto = data["goto"]
        self.closures = data["closures"]
        self.first_sets = data["first_sets"]
```

简单来说就是在用文法构造的同时也将`GOTO`等内容保存并输出到文件中，并可以将此处保存的文件进行加载然后执行，也就是相当于第一种输入方式的输入`ACTION`表和`GOTO`表来进行识别。

执行时使用保存下来的`table.cfg`就可以实现。




```
from Compiler import Compiler

compiler = Compiler("Parser/grammar.txt", load_from_file="table.cfg")
print(compiler.recognize("prog/test.c"))
```

需要注意的是，加载文件和导入文件用的是`rb`和`wb`，编码不是`utf-8`，所以并不能直观地看到内容，但是可以正常使用。

测试数据一（单个正确文件输入）

输入

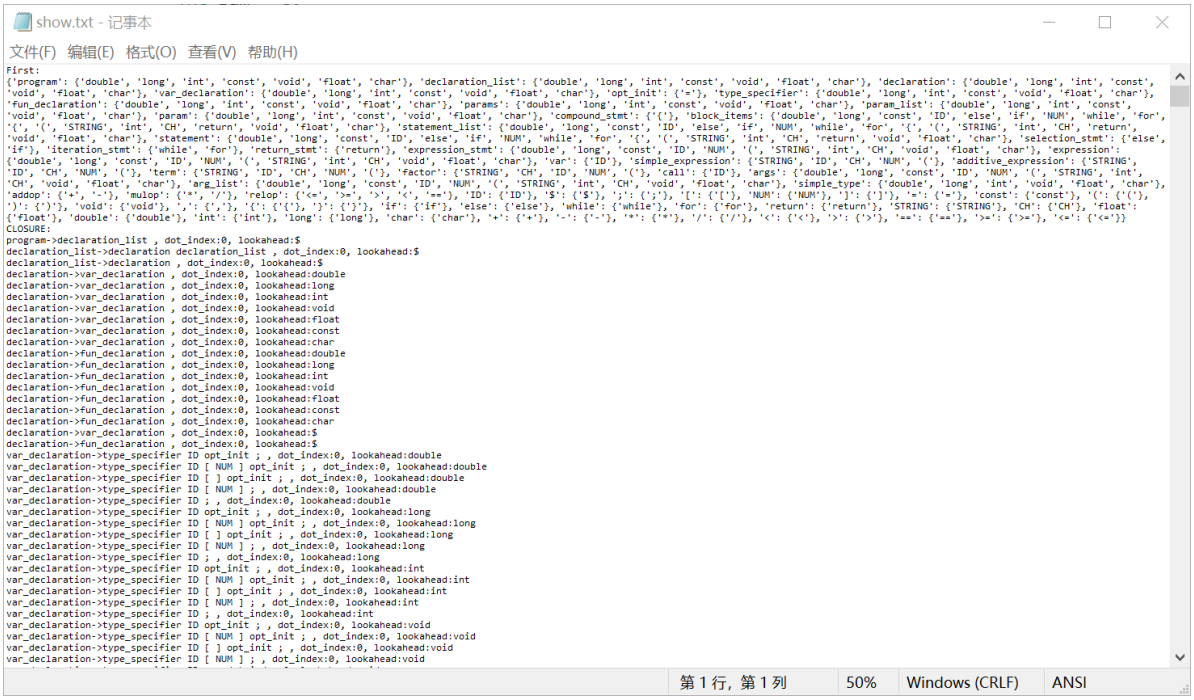
实际上我们在项目中给出了数个测试数据，但是为了展示我们功能的完整性，先直接展示其中最复杂的一个。

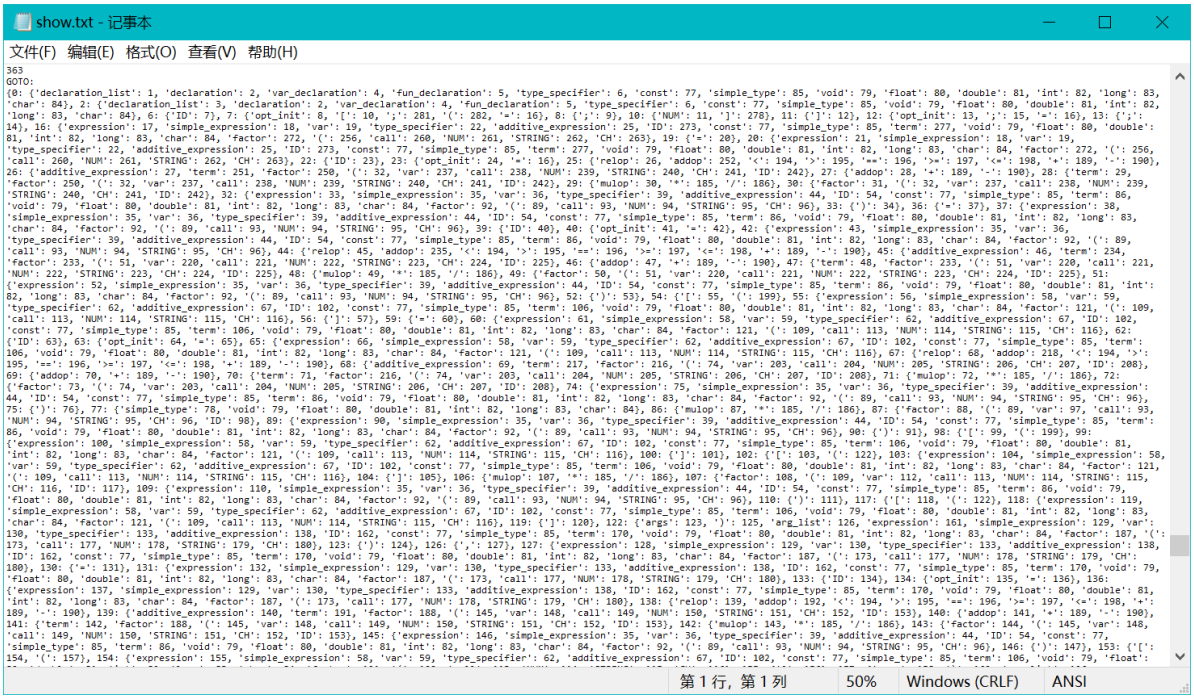
请注意，我们小组对C语言的语法分析相对来说已经是相当完善的了，除了指针没做以外，其他几乎都有实现，包括但不限于for循环、while循环、空大括号、函数等等，大致都可以在本例中直接看到

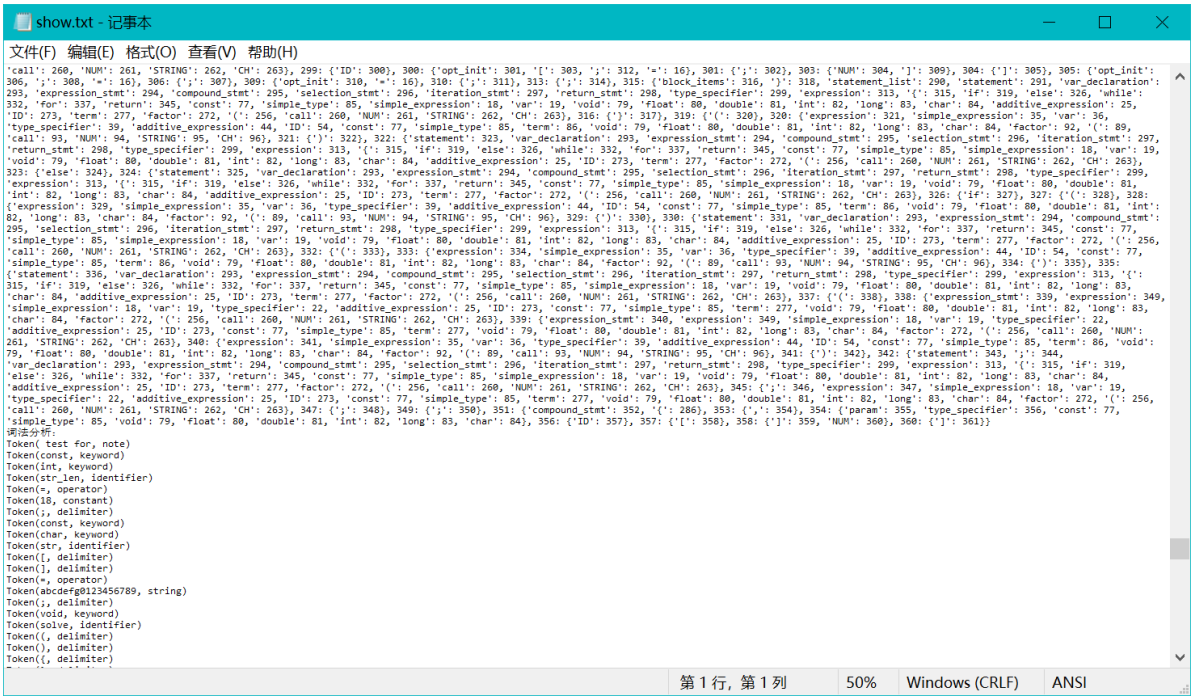
```
// test for

const int str_len = 18;
const char str[] = "abcdefg0123456789";

void solve() {
}

/*
    @is_digit
    @param ch
 */
const int is_digit(char ch) {
    int re = 0;
    if (ch >= '0' + 0) {
        if (ch <= '0' + 9) {
            re = 1;
        }
    }
    return re;
}

int is_letter(char ch) {
    int re = 0;
    if (ch >= 'a') {
        if (ch <= 'z') {
            re = 1;
        }
    }
    if (ch >= 'A') {
        if (ch <= 'Z') {
            re = 1;
        }
    }
    return re;
}

void input(void) {
    // input
}

long dpf(int n, int val[], int cost[]) {
    int dp[105];
    for (int i = 0; i < n; i = i + 1) {
        for (int j = 100; j >= val[i]; j = j - 1) {
            if (dp[j] < dp[j - cost[i]] + val[i]) {
                dp[j] = dp[j - cost[i]] + val[i];
            }
        }
    }
    return dp[100];
}

int main() {
    int cnt_digit = 0;
```

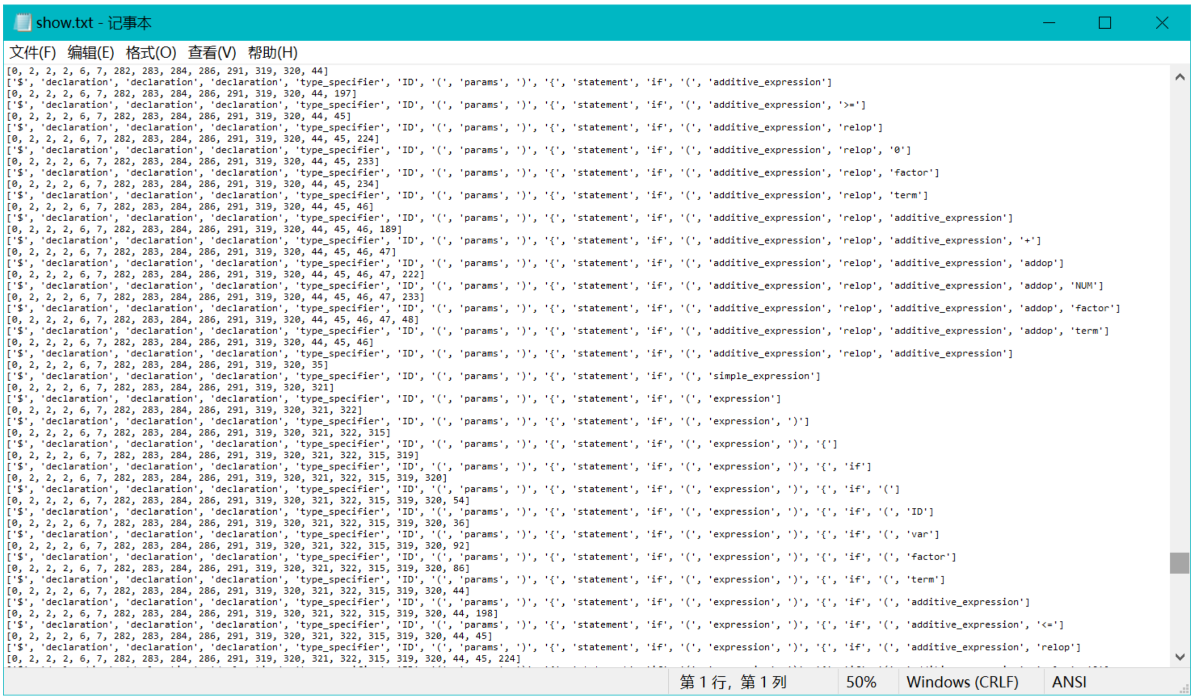
```
    int cnt_letter = 0;
    int i;
    for (i = 0; i < str_len; i = i + 1) {
        char ch = str[i];
        if (is_digit(ch)) {
            cnt_digit = cnt_digit + 1;
        }
        else if (is_letter(ch) == 1) {
            cnt_letter = cnt_letter + 1;
        }
    }

    int output;
    {
        output = cnt_digit * cnt_digit + (cnt_letter * cnt_letter);
        output = cnt_digit / cnt_digit * (cnt_digit - cnt_letter) - 1;
    }

    while (0) {
        int sum = 0;
        {
            for (int i = 0; i < 5; i = i + 1) {
                sum = sum + i;
            }
        }
        if (sum < 0) {
            return 1;
        }
    }

    return 0;
}
```

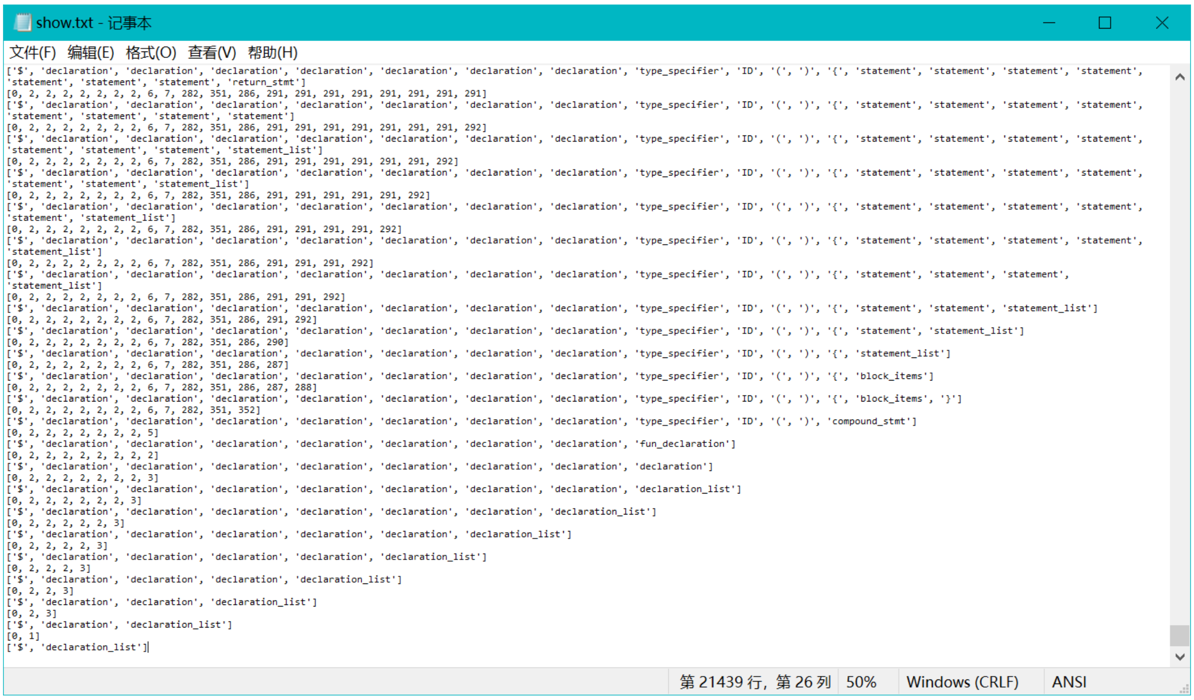
输出




show.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
363
GOTO:
{0: {'declaration_list': 1, 'declaration': 2, 'var_declaration': 4, 'fun_declaration': 5, 'type_specifier': 6, 'const': 77, 'simple_type': 85, 'void': 79, 'float': 80, 'double': 81, 'int': 82, 'long': 83, 'char': 84}, 2: {'declaration_list': 3, 'declaration': 2, 'var_declaration': 4, 'fun_declaration': 5, 'type_specifier': 6, 'const': 77, 'simple_type': 85, 'void': 79, 'float': 80, 'double': 81, 'int': 82, 'long': 83, 'char': 84}, 6: {'ID': 7}, 7: {'opt_init': 8, '[': 10, ';': 281, '(': 282, '=': 16}, 8: {';': 9}, 10: {'NUM': 11, ']': 278}, 11: {']': 12}, 12: {'opt_init': 13, ';': 15, '=': 16}, 13: {';': 14}, 16: {'expression': 17, 'simple_expression': 18, 'var': 19, 'type_specifier': 22, 'additive_expression': 25, 'ID': 273, 'const': 77, 'simple_type': 85, 'term': 277, 'void': 79, 'float': 80, 'double': 81, 'int': 82, 'long': 83, 'char': 84, 'factor': 272, '(': 256, 'call': 260, 'NUM': 261, 'STRING': 262, 'CH': 263}, 19: {'=': 20}, 20: {'expression': 21, 'simple_expression': 18, 'var': 19, 'type_specifier': 22, 'additive_expression': 25, 'ID': 273, 'const': 77, 'simple_type': 85, 'term': 277, 'void': 79, 'float': 80, 'double': 81, 'int': 82, 'long': 83, 'char': 84, 'factor': 272, '(': 256, 'call': 260, 'NUM': 261, 'STRING': 262, 'CH': 263}, 22: {'ID': 23}, 23: {'opt_init': 24, '=': 16}, 25: {'relop': 26, 'addop': 252, '<': 194, '>': 195, '==': 196, '>=': 197, '<=': 198, '+': 189, '-': 190}, 26: {'additive_expression': 27, 'term': 251, 'factor': 250, '(': 32, 'var': 237, 'call': 238, 'NUM': 239, 'STRING': 240, 'CH': 241, 'ID': 242}, 27: {'addop': 28, '+': 189, '-': 190}, 28: {'term': 29, 'factor': 250, '(': 32, 'var': 237, 'call': 238, 'NUM': 239, 'STRING': 240, 'CH': 241, 'ID': 242}, 29: {'mulop': 30, '*': 185, '/': 186}, 30: {'factor': 31, '(': 32, 'var': 237, 'call': 238, 'NUM': 239, 'STRING': 240, 'CH': 241, 'ID': 242}, 32: {'expression': 33, 'simple_expression': 35, 'var': 36, 'type_specifier': 39, 'additive_expression': 44, 'ID': 54, 'const': 77, 'simple_type': 85, 'term': 86, 'void': 79, 'float': 80, 'double': 81, 'int': 82, 'long': 83, 'char': 84, 'factor': 92, '(': 89, 'call': 93, 'NUM': 94, 'STRING': 95, 'CH': 96}, 33: {')': 34}, 36: {'=': 37}, 37: {'expression': 38, 'simple_expression': 35, 'var': 36, 'type_specifier': 39, 'additive_expression': 44, 'ID': 54, 'const': 77, 'simple_type': 85, 'term': 86, 'void': 79, 'float': 80, 'double': 81, 'int': 82, 'long': 83, 'char': 84, 'factor': 92, '(': 89, 'call': 93, 'NUM': 94, 'STRING': 95, 'CH': 96}, 39: {'ID': 40}, 40: {'opt_init': 41, '=': 42}, 42: {'expression': 43, 'simple_expression': 35, 'var': 36, 'type_specifier': 39, 'additive_expression': 44, 'ID': 54, 'const': 77, 'simple_type': 85, 'term': 86, 'void': 79, 'float': 80, 'double': 81, 'int': 82, 'long': 83, 'char': 84, 'factor': 92, '(': 89, 'call': 93, 'NUM': 94, 'STRING': 95, 'CH': 96}, 44: {'relop': 45, 'addop': 235, '<': 194, '>': 195, '==': 196, '>=': 197, '<=': 198, '+': 189, '-': 190}, 45: {'additive_expression': 46, 'term': 234, 'factor': 233, '(': 51, 'var': 220, 'call': 221, 'NUM': 222, 'STRING': 223, 'CH': 224, 'ID': 225}, 46: {'addop': 47, '+': 189, '-': 190}, 47: {'term': 48, 'factor': 233, '(': 51, 'var': 220, 'call': 221, 'NUM': 222, 'STRING': 223, 'CH': 224, 'ID': 225}, 48: {'mulop': 49, '*': 185, '/': 186}, 49: {'factor': 50, '(': 51, 'var': 220, 'call': 221, 'NUM': 222, 'STRING': 223, 'CH': 224, 'ID': 225}, 51: {'expression': 52, 'simple_expression': 35, 'var': 36, 'type_specifier': 39, 'additive_expression': 44, 'ID': 54, 'const': 77, 'simple_type': 85, 'term': 86, 'void': 79, 'float': 80, 'double': 81, 'int': 82, 'long': 83, 'char': 84, 'factor': 92, '(': 89, 'call': 93, 'NUM': 94, 'STRING': 95, 'CH': 96}, 52: {')': 53}, 54: {'[': 55, '(': 199}, 55: {'expression': 56, 'simple_expression': 58, 'var': 59, 'type_specifier': 62, 'additive_expression': 67, 'ID': 102, 'const': 77, 'simple_type': 85, 'term': 106, 'void': 79, 'float': 80, 'double': 81, 'int': 82, 'long': 83, 'char': 84, 'factor': 121, '(': 109, 'call': 113, 'NUM': 114, 'STRING': 115, 'CH': 116}, 56: {']': 57}, 59: {'=': 60}, 60: {'expression': 61, 'simple_expression': 58, 'var': 59, 'type_specifier': 62, 'additive_expression': 67, 'ID': 102, 'const': 77, 'simple_type': 85, 'term': 106, 'void': 79, 'float': 80, 'double': 81, 'int': 82, 'long': 83, 'char': 84, 'factor': 121, '(': 109, 'call': 113, 'NUM': 114, 'STRING': 115, 'CH': 116}, 62: {'ID': 63}, 63: {'opt_init': 64, '=': 65}, 65: {'expression': 66, 'simple_expression': 58, 'var': 59, 'type_specifier': 62, 'additive_expression': 67, 'ID': 102, 'const': 77, 'simple_type': 85, 'term': 106, 'void': 79, 'float': 80, 'double': 81, 'int': 82, 'long': 83, 'char': 84, 'factor': 121, '(': 109, 'call': 113, 'NUM': 114, 'STRING': 115, 'CH': 116}, 67: {'relop': 68, 'addop': 218, '<': 194, '>': 195, '==': 196, '>=': 197, '<=': 198, '+': 189, '-': 190}, 68: {'additive_expression': 69, 'term': 217, 'factor': 216, '(': 74, 'var': 203, 'call': 204, 'NUM': 205, 'STRING': 206, 'CH': 207, 'ID': 208}, 69: {'addop': 70, '+': 189, '-': 190}, 70: {'term': 71, 'factor': 216, '(': 74, 'var': 203, 'call': 204, 'NUM': 205, 'STRING': 206, 'CH': 207, 'ID': 208}, 71: {'mulop': 72, '*': 185, '/': 186}, 72: {'factor': 73, '(': 74, 'var': 203, 'call': 204, 'NUM': 205, 'STRING': 206, 'CH': 207, 'ID': 208}, 74: {'expression': 75, 'simple_expression': 35, 'var': 36, 'type_specifier': 39, 'additive_expression': 44, 'ID': 54, 'const': 77, 'simple_type': 85, 'term': 86, 'void': 79, 'float': 80, 'double': 81, 'int': 82, 'long': 83, 'char': 84, 'factor': 92, '(': 89, 'call': 93, 'NUM': 94, 'STRING': 95, 'CH': 96}, 75: {')': 76}, 77: {'simple_type': 78, 'void': 79, 'float': 80, 'double': 81, 'int': 82, 'long': 83, 'char': 84}, 86: {'mulop': 87, '*': 185, '/': 186}, 87: {'factor': 88, '(': 89, 'var': 97, 'call': 93, 'NUM': 94, 'STRING': 95, 'CH': 96, 'ID': 98}, 89: {'expression': 90, 'simple_expression': 35, 'var': 36, 'type_specifier': 39, 'additive_expression': 44, 'ID': 54, 'const': 77, 'simple_type': 85, 'term': 86, 'void': 79, 'float': 80, 'double': 81, 'int': 82, 'long': 83, 'char': 84, 'factor': 92, '(': 89, 'call': 93, 'NUM': 94, 'STRING': 95, 'CH': 96}, 90: {')': 91}, 98: {'[': 99, '(': 199}, 99: {'expression': 100, 'simple_expression': 58, 'var': 59, 'type_specifier': 62, 'additive_expression': 67, 'ID': 102, 'const': 77, 'simple_type': 85, 'term': 106, 'void': 79, 'float': 80, 'double': 81, 'int': 82, 'long': 83, 'char': 84, 'factor': 121, '(': 109, 'call': 113, 'NUM': 114, 'STRING': 115, 'CH': 116}, 100: {']': 101}, 102: {'[': 103, '(': 122}, 103: {'expression': 104, 'simple_expression': 58, 'var': 59, 'type_specifier': 62, 'additive_expression': 67, 'ID': 102, 'const': 77, 'simple_type': 85, 'term': 106, 'void': 79, 'float': 80, 'double': 81, 'int': 82, 'long': 83, 'char': 84, 'factor': 121, '(': 109, 'call': 113, 'NUM': 114, 'STRING': 115, 'CH': 116}, 104: {']': 105}, 106: {'mulop': 107, '*': 185, '/': 186}, 107: {'factor': 108, '(': 109, 'var': 112, 'call': 113, 'NUM': 114, 'STRING': 115, 'CH': 116, 'ID': 117}, 109: {'expression': 110, 'simple_expression': 35, 'var': 36, 'type_specifier': 39, 'additive_expression': 44, 'ID': 54, 'const': 77, 'simple_type': 85, 'term': 86, 'void': 79, 'float': 80, 'double': 81, 'int': 82, 'long': 83, 'char': 84, 'factor': 92, '(': 89, 'call': 93, 'NUM': 94, 'STRING': 95, 'CH': 96}, 110: {')': 111}, 117: {'[': 118, '(': 122}, 118: {'expression': 119, 'simple_expression': 58, 'var': 59, 'type_specifier': 62, 'additive_expression': 67, 'ID': 102, 'const': 77, 'simple_type': 85, 'term': 106, 'void': 79, 'float': 80, 'double': 81, 'int': 82, 'long': 83, 'char': 84, 'factor': 121, '(': 109, 'call': 113, 'NUM': 114, 'STRING': 115, 'CH': 116}, 119: {']': 120}, 122: {'args': 123, ')': 125, 'arg_list': 126, 'expression': 161, 'simple_expression': 129, 'var': 130, 'type_specifier': 133, 'additive_expression': 138, 'ID': 162, 'const': 77, 'simple_type': 85, 'term': 170, 'void': 79, 'float': 80, 'double': 81, 'int': 82, 'long': 83, 'char': 84, 'factor': 187, '(': 173, 'call': 177, 'NUM': 178, 'STRING': 179, 'CH': 180}, 123: {')': 124}, 126: {',': 127}, 127: {'expression': 128, 'simple_expression': 129, 'var': 130, 'type_specifier': 133, 'additive_expression': 138, 'ID': 162, 'const': 77, 'simple_type': 85, 'term': 170, 'void': 79, 'float': 80, 'double': 81, 'int': 82, 'long': 83, 'char': 84, 'factor': 187, '(': 173, 'call': 177, 'NUM': 178, 'STRING': 179, 'CH': 180}, 130: {'+': 131}, 131: {'expression': 132, 'simple_expression': 129, 'var': 130, 'type_specifier': 133, 'additive_expression': 138, 'ID': 162, 'const': 77, 'simple_type': 85, 'term': 170, 'void': 79, 'float': 80, 'double': 81, 'int': 82, 'long': 83, 'char': 84, 'factor': 187, '(': 173, 'call': 177, 'NUM': 178, 'STRING': 179, 'CH': 180}, 133: {'ID': 134}, 134: {'opt_init': 135, '=': 136}, 136: {'expression': 137, 'simple_expression': 129, 'var': 130, 'type_specifier': 133, 'additive_expression': 138, 'ID': 162, 'const': 77, 'simple_type': 85, 'term': 170, 'void': 79, 'float': 80, 'double': 81, 'int': 82, 'long': 83, 'char': 84, 'factor': 187, '(': 173, 'call': 177, 'NUM': 178, 'STRING': 179, 'CH': 180}, 138: {'relop': 139, 'addop': 192, '<': 194, '>': 195, '==': 196, '>=': 197, '<=': 198, '+': 189, '-': 190}, 139: {'additive_expression': 140, 'term': 191, 'factor': 188, '(': 145, 'var': 148, 'call': 149, 'NUM': 150, 'STRING': 151, 'CH': 152, 'ID': 153}, 140: {'addop': 141, '+': 189, '-': 190}, 141: {'term': 142, 'factor': 188, '(': 145, 'var': 148, 'call': 149, 'NUM': 150, 'STRING': 151, 'CH': 152, 'ID': 153}, 142: {'mulop': 143, '*': 185, '/': 186}, 143: {'factor': 144, '(': 145, 'var': 148, 'call': 149, 'NUM': 150, 'STRING': 151, 'CH': 152, 'ID': 153}, 145: {'expression': 146, 'simple_expression': 35, 'var': 36, 'type_specifier': 39, 'additive_expression': 44, 'ID': 54, 'const': 77, 'simple_type': 85, 'term': 86, 'void': 79, 'float': 80, 'double': 81, 'int': 82, 'long': 83, 'char': 84, 'factor': 92, '(': 89, 'call': 93, 'NUM': 94, 'STRING': 95, 'CH': 96}, 146: {')': 147}, 153: {'[': 154, '(': 157}, 154: {'expression': 155, 'simple_expression': 58, 'var': 59, 'type_specifier': 62, 'additive_expression': 67, 'ID': 102, 'const': 77, 'simple_type': 85, 'term': 106, 'void': 79, 'float':
第 1 行，第 1 列
50%
Windows (CRLF)
ANSI


show.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
'call': 260, 'NUM': 261, 'STRING': 262, 'CH': 263}, 299: {'ID': 300}, 300: {'opt_init': 301, '[': 303, ';': 312, '=': 16}, 301: {';': 302}, 303: {'NUM': 304, ']': 309}, 304: {']': 305}, 305: {'opt_init': 306, ';': 308, '=': 16}, 306: {';': 307}, 309: {'opt_init': 310, '=': 16}, 310: {';': 311}, 313: {';': 314}, 315: {'block_items': 316, '}': 318, 'statement_list': 290, 'statement': 291, 'var_declaration': 293, 'expression_stmt': 294, 'compound_stmt': 295, 'selection_stmt': 296, 'iteration_stmt': 297, 'return_stmt': 298, 'type_specifier': 299, 'expression': 313, '{': 315, 'if': 319, 'else': 326, 'while': 332, 'for': 337, 'return': 345, 'const': 77, 'simple_type': 85, 'simple_expression': 18, 'var': 19, 'void': 79, 'float': 80, 'double': 81, 'int': 82, 'long': 83, 'char': 84, 'additive_expression': 25, 'ID': 273, 'term': 277, 'factor': 272, '(': 256, 'call': 260, 'NUM': 261, 'STRING': 262, 'CH': 263}, 316: {'}': 317}, 319: {'(': 320}, 320: {'expression': 321, 'simple_expression': 35, 'var': 36, 'type_specifier': 39, 'additive_expression': 44, 'ID': 54, 'const': 77, 'simple_type': 85, 'term': 86, 'void': 79, 'float': 80, 'double': 81, 'int': 82, 'long': 83, 'char': 84, 'factor': 92, '(': 89, 'call': 93, 'NUM': 94, 'STRING': 95, 'CH': 96}, 321: {')': 322}, 322: {'statement': 323, 'var_declaration': 293, 'expression_stmt': 294, 'compound_stmt': 295, 'selection_stmt': 296, 'iteration_stmt': 297, 'return_stmt': 298, 'type_specifier': 299, 'expression': 313, '{': 315, 'if': 319, 'else': 326, 'while': 332, 'for': 337, 'return': 345, 'const': 77, 'simple_type': 85, 'simple_expression': 18, 'var': 19, 'void': 79, 'float': 80, 'double': 81, 'int': 82, 'long': 83, 'char': 84, 'additive_expression': 25, 'ID': 273, 'term': 277, 'factor': 272, '(': 256, 'call': 260, 'NUM': 261, 'STRING': 262, 'CH': 263}, 323: {'else': 324}, 324: {'statement': 325, 'var_declaration': 293, 'expression_stmt': 294, 'compound_stmt': 295, 'selection_stmt': 296, 'iteration_stmt': 297, 'return_stmt': 298, 'type_specifier': 299, 'expression': 313, '{': 315, 'if': 319, 'else': 326, 'while': 332, 'for': 337, 'return': 345, 'const': 77, 'simple_type': 85, 'simple_expression': 18, 'var': 19, 'void': 79, 'float': 80, 'double': 81, 'int': 82, 'long': 83, 'char': 84, 'additive_expression': 25, 'ID': 273, 'term': 277, 'factor': 272, '(': 256, 'call': 260, 'NUM': 261, 'STRING': 262, 'CH': 263}, 326: {'if': 327}, 327: {'(': 328}, 328: {'expression': 329, 'simple_expression': 35, 'var': 36, 'type_specifier': 39, 'additive_expression': 44, 'ID': 54, 'const': 77, 'simple_type': 85, 'term': 86, 'void': 79, 'float': 80, 'double': 81, 'int': 82, 'long': 83, 'char': 84, 'factor': 92, '(': 89, 'call': 93, 'NUM': 94, 'STRING': 95, 'CH': 96}, 329: {')': 330}, 330: {'statement': 331, 'var_declaration': 293, 'expression_stmt': 294, 'compound_stmt': 295, 'selection_stmt': 296, 'iteration_stmt': 297, 'return_stmt': 298, 'type_specifier': 299, 'expression': 313, '{': 315, 'if': 319, 'else': 326, 'while': 332, 'for': 337, 'return': 345, 'const': 77, 'simple_type': 85, 'simple_expression': 18, 'var': 19, 'void': 79, 'float': 80, 'double': 81, 'int': 82, 'long': 83, 'char': 84, 'additive_expression': 25, 'ID': 273, 'term': 277, 'factor': 272, '(': 256, 'call': 260, 'NUM': 261, 'STRING': 262, 'CH': 263}, 332: {'(': 333}, 333: {'expression': 334, 'simple_expression': 35, 'var': 36, 'type_specifier': 39, 'additive_expression': 44, 'ID': 54, 'const': 77, 'simple_type': 85, 'term': 86, 'void': 79, 'float': 80, 'double': 81, 'int': 82, 'long': 83, 'char': 84, 'factor': 92, '(': 89, 'call': 93, 'NUM': 94, 'STRING': 95, 'CH': 96}, 334: {')': 335}, 335: {'statement': 336, 'var_declaration': 293, 'expression_stmt': 294, 'compound_stmt': 295, 'selection_stmt': 296, 'iteration_stmt': 297, 'return_stmt': 298, 'type_specifier': 299, 'expression': 313, '{': 315, 'if': 319, 'else': 326, 'while': 332, 'for': 337, 'return': 345, 'const': 77, 'simple_type': 85, 'simple_expression': 18, 'var': 19, 'void': 79, 'float': 80, 'double': 81, 'int': 82, 'long': 83, 'char': 84, 'additive_expression': 25, 'ID': 273, 'term': 277, 'factor': 272, '(': 256, 'call': 260, 'NUM': 261, 'STRING': 262, 'CH': 263}, 337: {'(': 338}, 338: {'expression_stmt': 339, 'expression': 349, 'simple_expression': 18, 'var': 19, 'type_specifier': 22, 'additive_expression': 25, 'ID': 273, 'const': 77, 'simple_type': 85, 'term': 277, 'void': 79, 'float': 80, 'double': 81, 'int': 82, 'long': 83, 'char': 84, 'factor': 272, '(': 256, 'call': 260, 'NUM': 261, 'STRING': 262, 'CH': 263}, 339: {'expression_stmt': 340, 'expression': 344, 'simple_expression': 18, 'var': 19, 'type_specifier': 22, 'additive_expression': 25, 'ID': 273, 'const': 77, 'simple_type': 85, 'term': 277, 'void': 79, 'float': 80, 'double': 81, 'int': 82, 'long': 83, 'char': 84, 'factor': 272, '(': 256, 'call': 260, 'NUM': 261, 'STRING': 262, 'CH': 263}, 340: {'expression': 341, 'simple_expression': 35, 'var': 36, 'type_specifier': 39, 'additive_expression': 44, 'ID': 54, 'const': 77, 'simple_type': 85, 'term': 86, 'void': 79, 'float': 80, 'double': 81, 'int': 82, 'long': 83, 'char': 84, 'factor': 92, '(': 89, 'call': 93, 'NUM': 94, 'STRING': 95, 'CH': 96}, 341: {')': 342}, 342: {'statement': 343, ';': 344, 'var_declaration': 293, 'expression_stmt': 294, 'compound_stmt': 295, 'selection_stmt': 296, 'iteration_stmt': 297, 'return_stmt': 298, 'type_specifier': 299, 'expression': 313, '{': 315, 'if': 319, 'else': 326, 'while': 332, 'for': 337, 'return': 345, 'const': 77, 'simple_type': 85, 'simple_expression': 18, 'var': 19, 'void': 79, 'float': 80, 'double': 81, 'int': 82, 'long': 83, 'char': 84, 'additive_expression': 25, 'ID': 273, 'term': 277, 'factor': 272, '(': 256, 'call': 260, 'NUM': 261, 'STRING': 262, 'CH': 263}, 345: {';': 346, 'expression': 347, 'simple_expression': 18, 'var': 19, 'type_specifier': 22, 'additive_expression': 25, 'ID': 273, 'const': 77, 'simple_type': 85, 'term': 277, 'void': 79, 'float': 80, 'double': 81, 'int': 82, 'long': 83, 'char': 84, 'factor': 272, '(': 256, 'call': 260, 'NUM': 261, 'STRING': 262, 'CH': 263}, 347: {';': 348}, 349: {';': 350}, 351: {'compound_stmt': 352, '{': 286}, 353: {',': 354}, 354: {'param': 355, 'type_specifier': 356, 'const': 77, 'simple_type': 85, 'void': 79, 'float': 80, 'double': 81, 'int': 82, 'long': 83, 'char': 84}, 356: {'ID': 357}, 357: {'[': 358}, 358: {']': 359, 'NUM': 360}, 360: {']': 361}}
词法分析：
Token(test for, note)
Token(const, keyword)
Token(int, keyword)
Token(str_len, identifier)
Token(=, operator)
Token(18, constant)
Token(;, delimiter)
Token(const, keyword)
Token(char, keyword)
Token(str, identifier)
Token([, delimiter)
Token(], delimiter)
Token(=, operator)
Token(abcdefg0123456789, string)
Token(;, delimiter)
Token(void, keyword)
Token(solve, identifier)
Token((, delimiter)
Token(), delimiter)
Token({, delimiter)
第 1 行，第 1 列
50%
Windows (CRLF)
ANSI





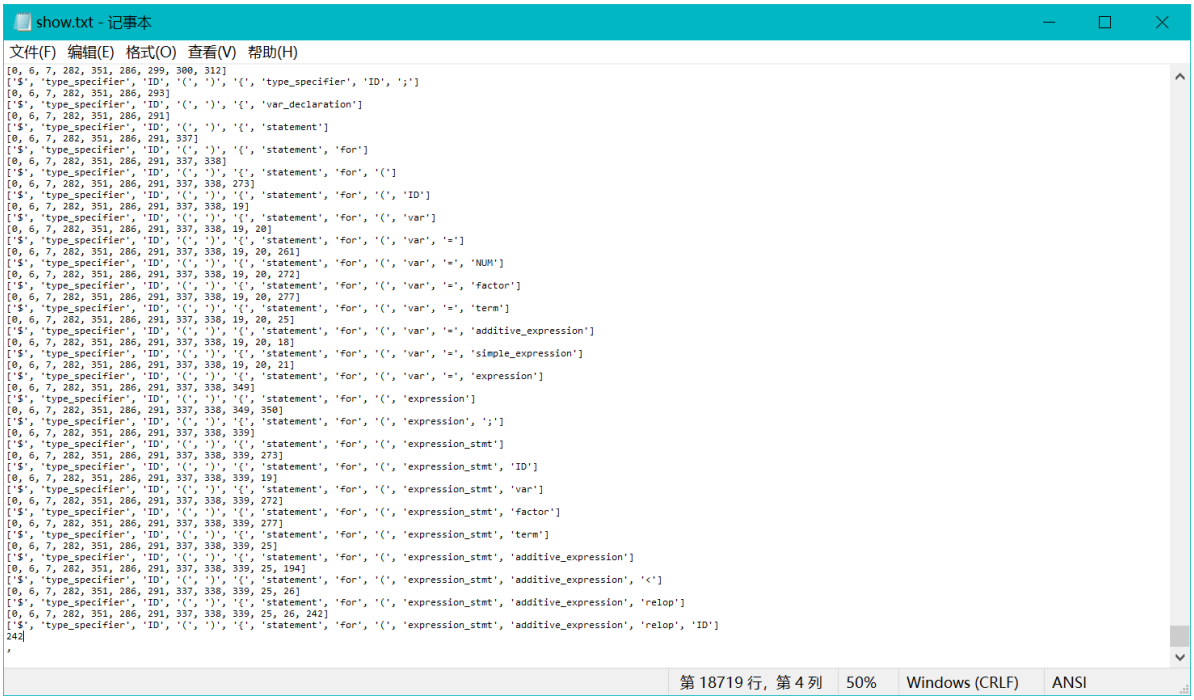
成功识别并归约到文法的开头

测试数据二（单个错误文件）

输入

```
int main() {
    int i;
    for (i = 0; i < n, i = i + 1) {
        i = i + 1;
    }
}
```

输出



识别失败，未能成功规约

测试数据三（同一文件夹下的一批文件）

输入

修改一下代码的输入部分

```
from Compiler import Compiler

compiler = Compiler("Parser/grammar.txt")
#compiler.recognize_files("prog/accepted", "result/accepted.txt")
compiler.recognize("prog/test.c")
compiler.lr1parser.construct_all
```

把单个文件换成文件夹

```
from Compiler import Compiler

compiler = Compiler("Parser/grammar.txt")
compiler.recognize_files("prog/accepted", "result/accepted.txt")
#compiler.recognize("prog/test.c")
compiler.lr1parser.construct_all
#####compiler.grammer.generate_syntax_tree("syntax_tree.png")#####
```

输出

accepted.txt - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

```
['$', 'declaration', 'declaration', 'declaration', 'declaration', 'declaration', 'declaration', 'type_specifier', 'ID', '(', ')', '{', 'block_items', '}']
[0, 2, 2, 2, 2, 2, 2, 6, 7, 282, 351, 352]
['$', 'declaration', 'declaration', 'declaration', 'declaration', 'declaration', 'declaration', 'type_specifier', 'ID', '(', ')', 'compound_stmt']
[0, 2, 2, 2, 2, 2, 2, 5]
['$', 'declaration', 'declaration', 'declaration', 'declaration', 'declaration', 'fun_declaration']
[0, 2, 2, 2, 2, 2, 2, 2]
['$', 'declaration', 'declaration', 'declaration', 'declaration', 'declaration', 'declaration']
[0, 2, 2, 2, 2, 2, 2, 3]
['$', 'declaration', 'declaration', 'declaration', 'declaration', 'declaration', 'declaration_list']
[0, 2, 2, 2, 2, 2, 3]
['$', 'declaration', 'declaration', 'declaration', 'declaration', 'declaration_list']
[0, 2, 2, 2, 2, 3]
['$', 'declaration', 'declaration', 'declaration', 'declaration_list']
[0, 2, 2, 2, 3]
['$', 'declaration', 'declaration', 'declaration_list']
[0, 2, 2, 3]
['$', 'declaration', 'declaration_list']
[0, 2, 3]
['$', 'declaration', 'declaration_list']
[0, 1]
['$', 'declaration_list']
final-test.c: 正确
词法分析:
Token(int, keyword)
Token(main, identifier)
Token((, delimiter)
Token(), delimiter)
Token({, delimiter)
Token(int, keyword)
Token(i, identifier)
Token(;, delimiter)
Token(int, keyword)
Token(n, identifier)
Token(;, delimiter)
Token(for, keyword)
Token((, delimiter)
Token(i, identifier)
Token(=, operator)
Token(0, constant)
Token(;, delimiter)
Token(i, identifier)
Token(<, operator)
```

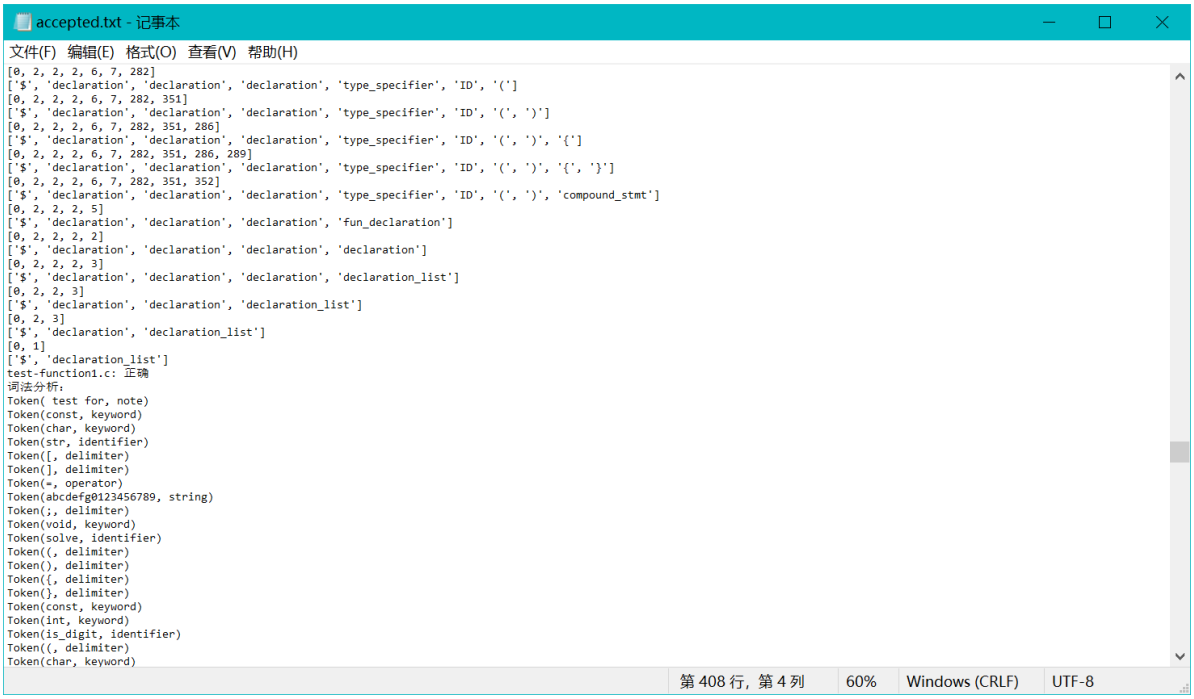
第 408 行，第 4 列 60% Windows (CRLF) UTF-8

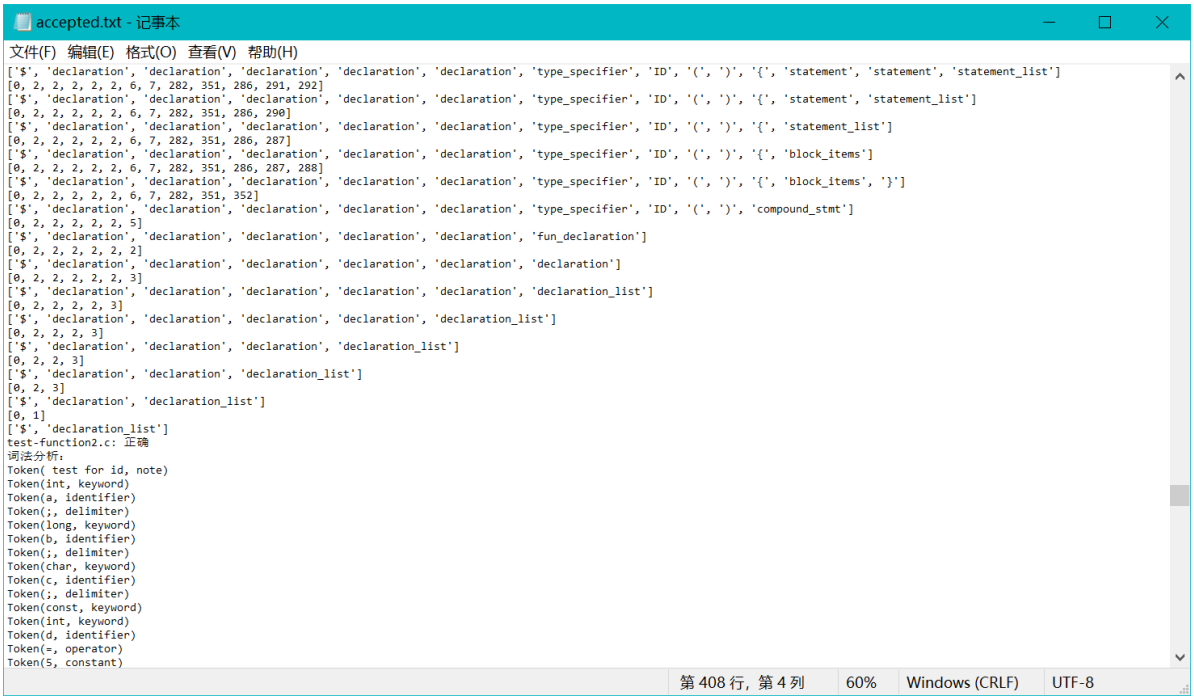
accepted.txt - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

```
['$', 'type_specifier', 'ID', '(', ')', '{', 'statement', 'statement', 'for', '(', 'expression_stmt', 'expression_stmt', 'expression', ')', '{', 'block_items', '}']
[0, 6, 7, 282, 351, 286, 291, 291, 337, 338, 339, 340, 341, 342, 295]
['$', 'type_specifier', 'ID', '(', ')', '{', 'statement', 'statement', 'for', '(', 'expression_stmt', 'expression_stmt', 'expression', ')', 'compound_stmt']
[0, 6, 7, 282, 351, 286, 291, 291, 337, 338, 339, 340, 341, 342, 343]
['$', 'type_specifier', 'ID', '(', ')', '{', 'statement', 'statement', 'for', '(', 'expression_stmt', 'expression_stmt', 'expression', ')', 'statement']
[0, 6, 7, 282, 351, 286, 291, 291, 297]
['$', 'type_specifier', 'ID', '(', ')', '{', 'statement', 'statement', 'iteration_stmt']
[0, 6, 7, 282, 351, 286, 291, 291, 291]
['$', 'type_specifier', 'ID', '(', ')', '{', 'statement', 'statement', 'statement']
[0, 6, 7, 282, 351, 286, 291, 291, 292]
['$', 'type_specifier', 'ID', '(', ')', '{', 'statement', 'statement', 'statement_list']
[0, 6, 7, 282, 351, 286, 291, 292]
['$', 'type_specifier', 'ID', '(', ')', '{', 'statement', 'statement_list']
[0, 6, 7, 282, 351, 286, 290]
['$', 'type_specifier', 'ID', '(', ')', '{', 'statement_list']
[0, 6, 7, 282, 351, 286, 287]
['$', 'type_specifier', 'ID', '(', ')', '{', 'block_items']
[0, 6, 7, 282, 351, 286, 287, 288]
['$', 'type_specifier', 'ID', '(', ')', '{', 'block_items', '}']
[0, 6, 7, 282, 351, 352]
['$', 'type_specifier', 'ID', '(', ')', 'compound_stmt']
[0, 5]
['$', 'fun_declaration']
[0, 2]
['$', 'declaration']
[0, 1]
['$', 'declaration_list']
test-for1.c: 正确
词法分析:
Token( test for function, note)
Token(void, keyword)
Token(solve, identifier)
Token((, delimiter)
Token(), delimiter)
Token({, delimiter)
Token(}, delimiter)
Token(const, keyword)
Token(char, keyword)
Token(is_digit, identifier)
Token((, delimiter)
Token(char, keyword)
Token(ch, identifier)
Token(), delimiter)
Token({, delimiter)
```

第 408 行，第 4 列 60% Windows (CRLF) UTF-8





依次输出为正确

调试问题思考与解决

我们使用的文法是小组成员自己一点点修改完善出来的，不是网上直接搬来的，也不是老师给的。（当我们实现功能的时候老师还没有给我们提供文法）

实际上，文法的构造是及其耗费心力的。某种程度上来说，这甚至比代码的实现还要麻烦。

参考我们的输入测试一，可以看到我们的识别已经是相对来说完成度相当高了，但相对的，能识别这样复杂的测试代码本身对文法的要求也是相当高，因此在构造中也经过了相当多的修改调整，这里举一个例子类比我们遇到的一个问题及其解决。

这里仅仅是举例，并非当时的具体内容：

我们早期的文法中对有这样的语句：

```
statement_list -> statement_list statement | empty
```

其中`empty`表示的是空，也就是课上通常讲的`ϵ`。

使用这样的形式的文法，我们在识别和如下两句类似的情况时出现了问题：

```
int a = b + c;
int a;
```

最后的解决方法是将空消除掉，消除方式是将产生式右侧出现的空直接放到前一个用来产生这个产生式左侧的产生式中。

例如：

```
A -> B | C
B -> D | ϵ
```

就改成：

```
A -> C | D
```

按照此种方式就可以解决类似的问题。

五、总结与收获

收获

1. **深入理解编译原理原理：**

   实现LR(1)文法的词法和语法分析，需要对编译原理的相关理论有深入的理解。这使我们更加熟悉自底向上的语法分析方法、First和Follow集合、LR(1)项、LR(1)自动机等概念。

2. **实际应用能力：**

   实现编译器的部分功能是对编译原理理论的实际应用，这有助于将抽象的理论知识转化为实际可运行的程序。

3. **Python编程技能：**

   在编写编译器时，我们学习并熟悉和锻炼了许多Python编程技能，包括文件读写、数据结构的使用、递归、异常处理等。这有助于提升我们的编程能力。

对课程的认识

1. **实现了实践与理论结合：**

   编写编译器使我们更加深入理解编译原理的理论知识。这种实践与理论结合的方式使我们更好地理解课堂上学到的概念。

2. **拥有了对复杂性的认识：**

   编译器是软件工程中复杂的一部分。通过实现编译器，我们更加认识到软件工程中处理复杂问题的挑战。

3. **获得了深度学习机会：**

   通过编写编译器，我们有机会深入研究更高级的编译原理概念，例如优化、中间代码生成等。