目 录

一 实验要求

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。

要求语法分析程序要能够调用词法分析程序，并为后续调用语义分析模块做考虑。对输入的一个文法和一个单词串，程序能正确判断此单词串是否为该文法的句子，并要求输出分析过程和语法树。

二 需求分析

2.1 任务范围

本项目仅对编译器实现中词法和语法分析展开研究。头文件展开不在本项目的研究范围内。

2.1.1 词法分析的主要工作

词法分析是编译器中的一个重要阶段，它负责将源代码分解为一个个词法单元，也称为记号或词法符号。其主要工作是根据编程语言的词法规则，将输入的字符序列转换为具有语义意义的词法单元序列。这些词法单元可以是关键字、标识符、常量、运算符、界符等。通过词法分析，编译器能够将源代码转换为更易于处理的词法单元序列，为后续的语法分析和语义分析提供基础，词法分析的准确性和高效性对于编译器的整体性能至关重要。

基本步骤 其过程通常包括以下四个步骤：

1.扫描。逐个读取源代码字符，忽略空白符和注释，并将字符序列转换为记号序列；

2.识别。识别阶段会根据预定义的词法规则，将字符序列识别为特定的词法单元；

3.分类。将识别出的词法单元进行分类，例如将标识符和关键字区分开；

4.生成词法单元。生成词法单元序列，供后续的语法分析和语义分析使用。

DFA 构建 在词法分析中的一个关键步骤时构建 DFA。DFA 是一种用于识别正则语言的有限状态自动机，应用在词法分析中可以提高词法分析的效率、帮助简化和组织词法规则、提供良好的容错性和错误处理能力。构建 DFA 的详细步骤包括以下几个主要阶段：

1.根据给定的正则表达式，使用 Thompson 构造法生成 NFA；

2.通过子集构造法将 NFA 转换为等价的 DFA。在子集构造法中，根据 NFA 的状态和输入符号，构建 DFA 的状态转换表；

3.优化 DFA 以减少状态数，提高词法分析的效率。

2.1.2 语法分析的主要工作

语法分析负责对输入的源代码进行语法分析和解析，以确定其是否符合语法规则，并构建相应的语法树或其他中间表示形式。语法分析器的主要工作包括识别和分类符号、文法规则匹配、构建语法树、错误处理以及生成中间表示等任务。

基本步骤 其过程包括：构建 first 集、构建项目集、构造项目集之间的转移关系、构建项目集闭包和构建项目集族、构建 LR(1)分析表、构建语法树：

1.构建 first 集确定文法中每个非终结符的首个终结符集合，用于后续的项目集构建和分析表的生成。

2.构建项目集是将文法的产生式规则转化为项目的集合，其中每个项目表示规约或移进的可能性；

3.构造项目集之间的转移关系是为了建立项目集之间的转移路径，以便在语法分析过程中进行状态转移。同时构建项目集闭包是为了处理可扩展项目集，即包含闭包操作的项目集；

4.通过构建项目集族可以得到完整的 LR(1)分析表，其中包含状态和动作，用于解析输入符号串；

5.最后通过 LR(1)分析表的推导过程，采用自底向上的移进-归约分析方法，将输入的词法单元流进行匹配和推导，形成语法树的结构，至此获得语法分析的最终输出。

规约过程 在语法分析过程中，需要特别注意规约过程以及与之相关的栈变化。规约是指将项目集中的项目转化为产生式的过程，它是语法分析中的重要步骤之一。规约时，需要维护状态栈、符号栈和动作说明，以正确地跟踪和处理语法分析的状态变化（如下图）。状态栈用于存储语法分析器的状态，符号栈用于存储已识别的终结符和非终结符，而动作说明则指示在每个步骤中应该执行的操作，如移进、规约或接受。通过仔细跟踪栈的变化，可以确保语法分析器按照规定的语法规则正确地解析输入符号串。

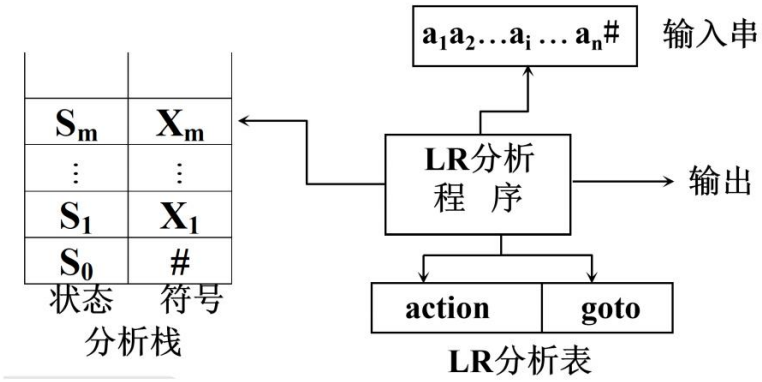


图 1 获得 LR(1)分析表与规约过程图解

语法树 构造语法树也是语法分析的关键任务，本作业中需要根据 LR(1)型文法最终构建语法分析树以直观地展示代码结构。语法树是一种树形结构，用于表示源代码的语法结构。在语法分析过程中，通过使用产生式规则，将输入符号串转换为语法树。语法树可以提供抽象语法结构的可视化表示，使得后续的语义分析更加方便和高效。它可以帮助语义分析器理解源代码的含义，进行类型检查、语法错误检测以及生成中间代码等任务。同时语法树还可以用于生成目标代码和进行代码优化，对编译器的整体性能和功能起到重要作用。

2.2 输入输出

根据题目要求，我们采用“由程序自动生成该文法的 ACTION 表和 GOTO 表”的形式实现语法分析过程。

系统输入输出 系统的输入为用户上传或在线编辑的类 C 语句。若分析过程出现错误，则返回报错信息。

词法分析 词法分析的主要任务是识别源程序中的单词符号，并将其映射为相应的词汇单元。该部分读入用户上传的类 C 语句，可以分析出以下内容：

1.标识符：标识符是用来命名变量、函数、常量等的字符串，例如变量、函数、常量等；

2.关键字：关键字是具有特殊含义的字符串，例如保留字、保留标识符等；

3.运算符：可以识别出运算符的类型和运算符的优先级；

4.界符：界符用于标记语句的开始和结束，以及表达式的分隔。

语法分析 语法分析的主要任务是根据语言的语法规则，将词法符号序列分解成各类语法单位。其输入是词法分析器划分生成的不同的词法单元（如关键字、标识符、运算符）的序列；其输出以下内容：

1.源程序的语法结构，例如语句的嵌套结构、表达式的运算顺序；

2.分析规约过程，包括符号栈、状态栈、动作说明；

3.LR(1)型语法分析表

4.语法树，根据 LR(1)分析表的推导过程可以获得语法树，直观地展示类 C 语句的语法结构。

2.3 系统功能

本系统的目标是实现类 C 语法和词法分析器，并创建用户界面来实现语法分析器与用户之间的交互。系统后台会预定义一个类 C 语法，用户可以将类 C 文件上传系统，系统可对输入的词法单元串进行语法分析。若分析成功，将展示相应的语法分析树以及详细的词法和语法分析过程；否则将提供相应的错误提示（如果是因上传文件不是 LR(1)型语法，系统可以进行甄别）。

系统注重用户界面的设计，以提供友好的交互体验。用户可以直观地上传文件，并在界面上查看各类分析结果，具体按钮和截图功能展示如下。

图 2 系统初始界面，未上传文件时，词法分析和语法分析按钮无法点击

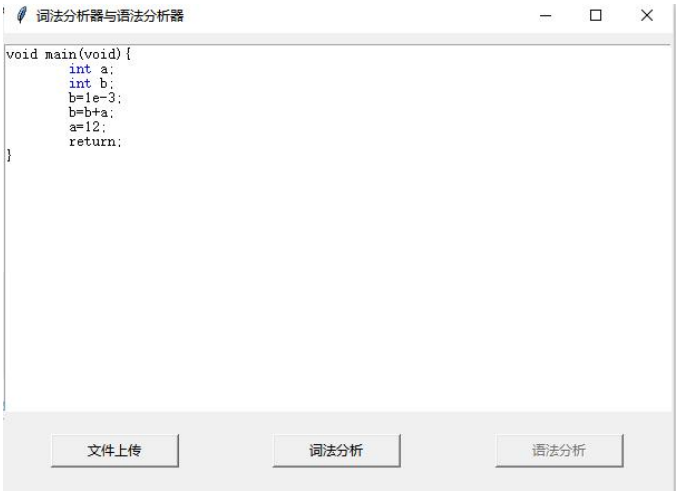


图 3 上传代码后，关键字高亮，词法分析按钮使能

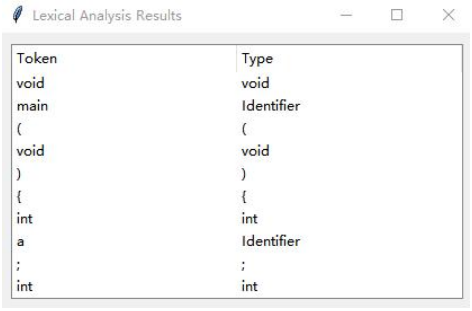


图 4 点击词法分析按钮后弹窗展示词法分析结果，同时主界面语法分析按钮使能

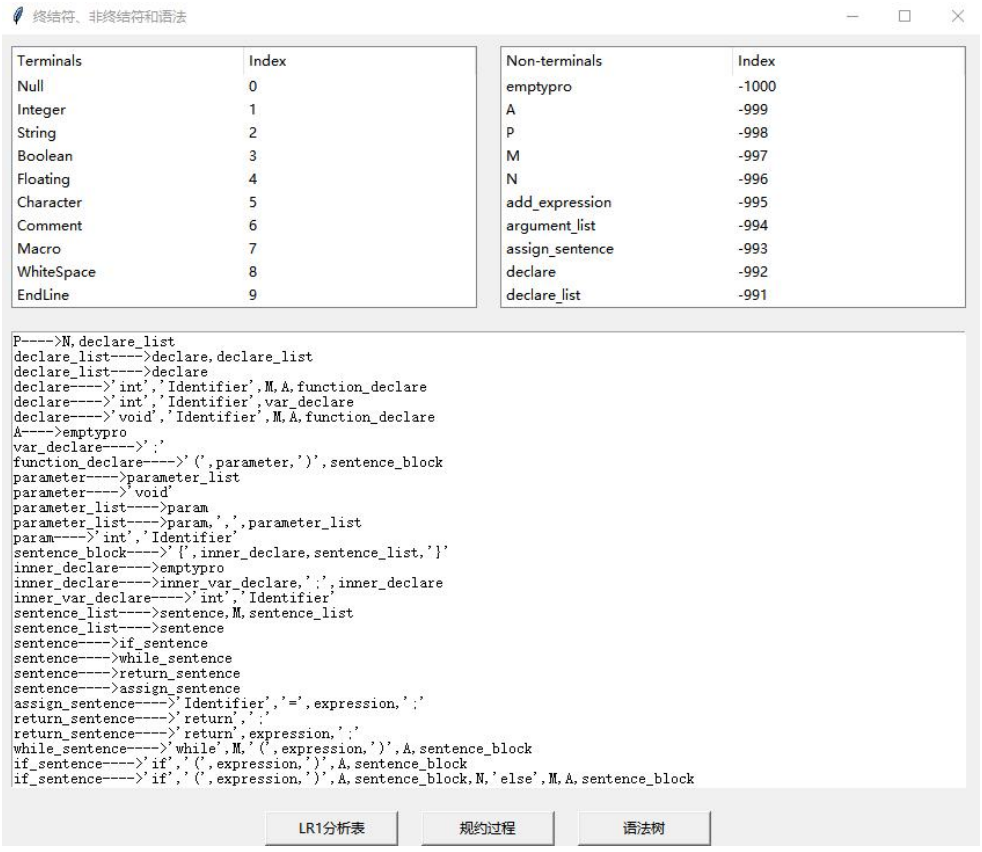


图 5 语法分析窗口，左上方为终结符，右上方为非终结符，下方为语法分析结果

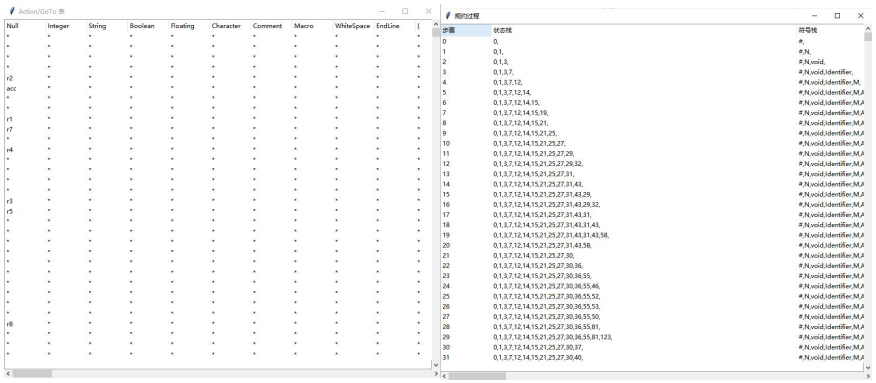


图 6　LR(1)分析表窗口与规约过程窗口

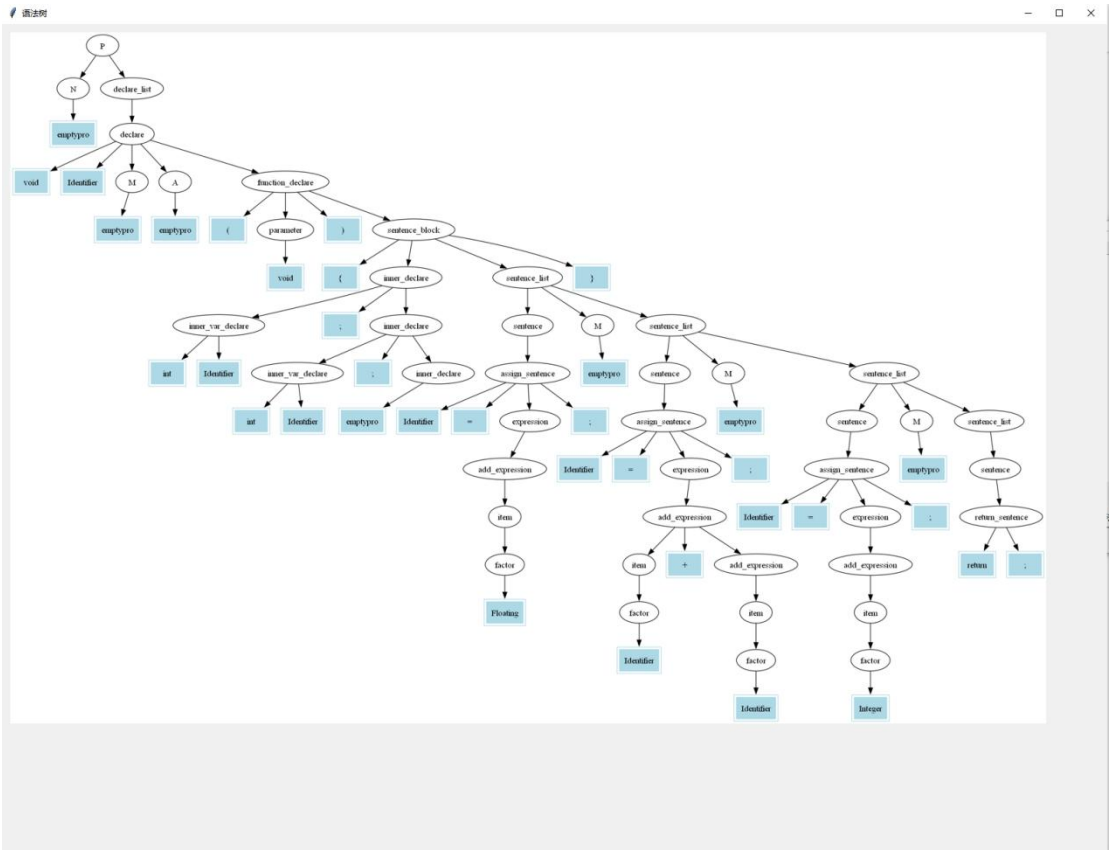


图 7　语法树分析窗口

2.4 测试数据

我们预先构建了五个测试文件，分三个单元进行测试：其中正确数据 1.c、2.cpp、3.txt 为正确数据，涵盖了从简单定义到浮点数到函数嵌套等逐级递增的复杂度；错误数据中 4.c 中有缺失分号的语法错误（不符合 LR(1)型文法）；5.cpp 中有错误表达式。

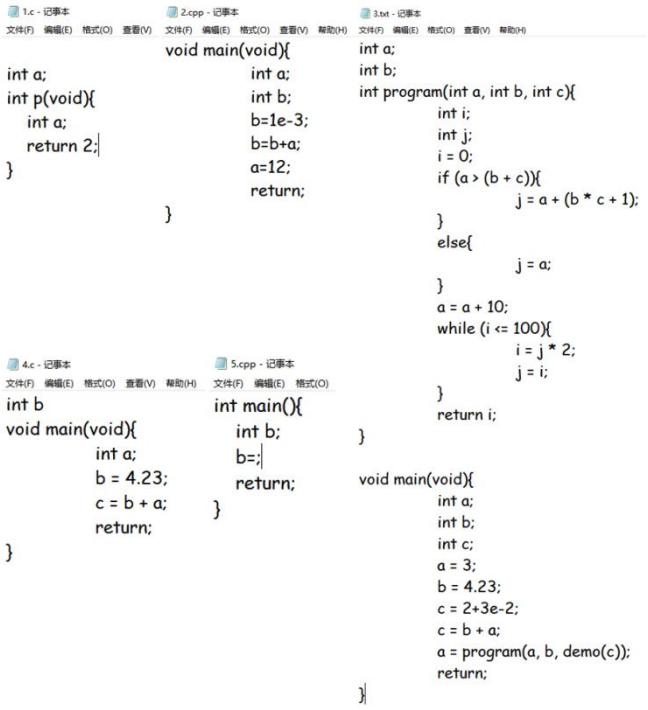


图 8　人为设置的 5 份测试数据内容

我们尽力通过这些测试文件涵盖了不同的情况，如包含了.c、.txt、.cpp 不同格式、存在额外空格回车制表符、同时存在整数浮点数等等。为了测试错误情况，我们还构建了错误的词法单元序列，如以上展示的 4.c、5.cpp。这些错误包括不符合 LR(1)文法的情况、错误的表达式等。通过这些测试文件，可以对词法分析器进行全面的测试和评估，检查其在不同情况下的准确性和鲁棒性。这将有助于发现和解决潜在的问题，并提高词法分析器的性能和可靠性。

三 概要设计

3.1 任务分解

本任务可以划分为词法分析、语法分析、系统呈现三大部分，整个系统逻辑结构如下。

3.1.1 词法分析

为实现一个高效、准确的词法分析器，我们将词法分析阶段的任务分为以下几个部分：

1.读入类 C 文件。若用户手动输入，则直接传入输入的字符串，若该上传的是一个文件，则将文件中内容的以字符串的形式读入变量中，传入该字符串；

2.分析字符、字符串、分隔符、关键字、运算符等词法单元。分隔符是用于标记代码中不同部分的特殊字符，如括号、分号等。本项目中我们以类似哈希表的数据结构预先定义了终结符 VT 与非终结符 NT 集，便于词法分析过程识别和生成相应的终结符分析，关键字是具有特殊含义的词汇（如 if、for、while 等），运算符用于执行各种操作（如加法、减法、赋值等）；

3.分析整数、浮点数、标识符和其他单词。本项目中构建了 3 个 DFA 来分别识别整数、浮点数、标识符；

4.输出词法分析结果。

构建 DFA 本阶段通过构建 DFA 来识别分析整数、浮点数、标识符。使用 DFA 的优点：

1.可以明确定义整数和浮点数的规则，例如数字的排列顺序、小数点的位置等。对于标识符，可以定义合法的字符集和命名规则，确保只有符合规范的标识符被识别。

2.利用有限状态的转移表进行匹配，可以快速地确定输入字符序列是否符合特定的词法规则，将词法分析的复杂度降低到线性时间复杂度，从而提高词法分析的速度。当然通过定义合适的状态转移，可以处理一些常见的错误情况，如非法字符、不完整的数字等。

构建 DFA 的详细步骤可概括为：

1.定义状态机的不同状态；

2.通过处理逻辑绘制状态转移图并构造 NFA（包含错误状态）；

3.通过子集构造法将 NFA 转换为等价的 DFA，在子集构造法中，根据 NFA 的状态和输入符号，构建 DFA 的状态转换表；

4.最后，优化 DFA 以减少状态数目，提高词法分析的效率。

字典树 词法分析的结果中包括类 C 文件中的关键字和运算符，在分析和输出过程中，涉及很多查找，为优化查找速度，我们采用字典树这种数据结构进行存储和分析。字典树（Trie）是一种用于高效查找的数据结构，可以帮助构建一个快速且灵活的词法分析器:通过一个字典树来存储关键字和运算符，字典树的每个节点表示一个字符，从根节点开始，通过不同的路径到达叶子节点，形成完整的单词。由此可以将关键字和运算符作为单词插入字典树中，每个字符对应一个节点。

前缀匹配 本项目使用字典树主要是考虑到其快速的前缀匹配功能：词法分析过程中，读取输入字符时，系统可以根据当前字符在字典树中进行匹配，以确定是否存在以该字符开头的关键字或运算符，由此大大提高词法分析的效率。应用过程中我们也意识到了字典树的其他功能，比如模糊匹配和自动补全：给定一个关键字时，可以使用字典树来查找与该部分匹配的结点之下的所有可能单词,从而提供自动补全的建议（当然对于本项目来说帮助不大，但是对很多输入法自动补全功能的实现等等给出了很多想法）。

时间与空间的权衡 在代码实现过程中，我们意识到：字典树的构建和维护可能需要一些额外的空间和时间，尤其是当关键字和运算符的数量很大时，使用时则需要开始权衡空间和时间的需求，并根据具体情况进行优化。

3.1.2 语法分析

为实现一个高效、准确的语法分析器，我们将语法分析阶段的任务分为以下几个部分：

1.**读取预先定义的语法文件。**首先需要读取包含语法规则的文件，该文件定义了文法的产生式和终结符、非终结符等信息。这些规则将用于后续的语法分析过程；

2.**获得词法分析结果。**在语法分析之前，需要进行词法分析等预处理，将用户输入的源代码转化为词法单元序列。词法单元可以是标识符、关键字、运算符等。利用词法分析器将源代码分割成词法单元，并将其作为输入提供给语法分析器；

3.**求 First 集和 Follow 集。**通过对语法规则进行分析可以计算终结符和非终结符的 First 集和 Follow 集。First 集表示一个符号串的首个终结符集合，Follow 集表示在某个上下文中紧随该符号之后的终结符集合。这些集合将在后续的语法分析中用于构建分析表和进行语法分析；

4.**求闭包和 LR(1)项目集族。**本作业中使用深度优先搜索（DFS）算法求解每个项目的闭包和构建 LR(1)项目集族。闭包操作将根据产生式的右部扩展项目，直到无法再扩展为止。然后使用 LR(1)项目集族构建状态转移图，表示不同项目集之间的转移关系。

5.**生成 LR(1)集族的状态转移信息，求得 Action 表和 Goto 表。**根据第 4 步中构建 LR(1)项目集族和状态转移图的过程可以生成状态转移信息，包括 Action 表和 Goto 表。Action 表用于处理终结符的移进和规约操作，Goto 表用于处理非终结符的转移操作。最终程序将结果写入文件 LR1Table.txt 中，存档并便于后续前端的读取展示；

6.**记录总规约过程。**借助生成的 Action 表和 Goto 表可以进行语法分析。语法分析过程中，根据输入的词法单元序列，分析器将根据 Action 表和 Goto 表执行相应的移进、规约和转移操作。本作业中我们将分析过程，包括操作栈、数字栈和行动说明写入文件 AnalyzeProcess.txt 中，以便查看分析器的状态和动作。

7.**获得语法树。**这是语法分析的最后阶段，规约过程将根据语法规则将栈中的符号进行规约，直到达到文法的起始符号，这个过程将生成语法树的节点信息，可以根据规约动作构建语法树，用于后续的语义分析和代码生成。

Graphviz 本作业在语法分析过程中，采用基于 Graphviz 库的 dot 文件绘制语法树，并转为 png 格式进行显示。Graphviz 根据脚本自动布局生成图表。我们主要使用 graphviz 进行 LR(1)语法树的绘制，利用其两个主要好处，一个是将用户从排版中解放出来，由工具自动处理这个过程，用户不必再关心如何布局，修改添加删除节点的时候也不用再对整个图的排版布局重新进行人工的整理;另一个好处是某些复杂的情况,比如代码的类图和调用图，dot 脚本可以使用其他工具自动生成。

3.1.3 系统呈现

程序运行时，必要的分析过程量会存储在相应变量中，而词法分析结果、LR(1)表、总规约过程等重要结果将存储在文件中。本项目通过 python 编写实现词法分析器和语法分析器，采用 tkinter 库完成图形化界面显示系统分析结果，向用户直观明了地展示执行信息。以下为任务分解阶段我们对界面主题内容、按钮交互、页面跳转等逻辑的设计。

tkinter 本作页使用 tkinter 进行前端开发。tkinter 是一个用于创建图形用户界面（GUI）的 Python 库,可以在多个操作系统上运行，使得开发的应用程序可以在不同平台上无缝运行。同时它也提供了丰富的功能和组件，如窗口、对话框、按钮、文本框、表格、菜单等，可以便捷地满足各种 GUI 应用程序的需求。从语言上来说，与基于 C++的 QT 相比，tkinter 具有 Python 语言简洁、易学、易读等优点，因此我们选用 tkinter 进行前端呈现。

3.2 数据类型定义

词法分析和语法分析均需使用终结符的枚举类 VT、非终结符的枚举类 VN。包括数据类型、括号、运算符、关键字、空格、换行符、NULL 等，分别将其对应一个数字，再定义类似哈希表的结构，将名称与数字进行对应，便于存储与后续处理。数据结构定义如下：

```
class VT(Enum):
        VTNull = 0
        VTInteger = 1
        VTString = 2
        VTBoolean = 3
        VTFloating = 4
        VTCharacter = 5
        VTComment = 6

        ……

        VTUnion = 78
        VTUnsigned = 79
        VTVoid = 80
        VTVolatile = 81
        VTWhile = 82
        VTIdentifier = 83
        VTWord = 84
```

```
VT_names = [
    "Null",
    "Integer",
    "String",
    "Boolean",
    "Floating",
    "Character",
    "Comment",

    ……

    "union",
    "unsigned",
    "void",
    "volatile",
    "while",
    "Identifier",
    "Word"
]
```

3.2.1 词法分析

主要数据结构包括：分割组成类 LexSegment、字典树类 Trie（字典树结点 Res）、词法分析类 LexAnalyzer。

用以分割上传文件内容的类：

```
class LexSegment():
    def __init__(self, li=-1, ri=-1, ctype=VT.VTNull):
        self.li = li          # 原文中的相对偏移位置左端
        self.ri = ri          # 原文中的相对偏移位置右端
        self.ctype = ctype    # 终结符的种类
```

字典树结点存储信息的数据结构：

```
class Node:
    def __init__(self, val=VT.VTNull, begin=-1, end=-1):
        self.val = val
        self.begin = begin
        self.end = end
```

字典树类，便于查找关键字和运算符：

```
class Trie:
    def __init__(self):
        self.isLeaf = False   # 是否为叶结点
        self.next = [None for _ in range(128)]    # 指向下一个字典树的指针数组
        self.type = None
```

```
    # 插入字典树元素
    def insert(self, word, t):
    # 检查关键字或运算符是否匹配
    def match(self, word):
```

词法分析类 LexAnalyzer，主类，对外提供 2 个方法：

1. Analyze()进行词法分析的总过程；
2. Get_Result()传出词法分析的结果

```
class LexAnalyzer:
    def __init__(self):
        self.original_code = None          # 原始字符串
        self.code_length = -1              # 字符串长度
        self.inner_result = []             # 词法分析结果
        self.component_type = []           # 枚举类型记录？？？ #
        self.trie = Trie()                 # 字典树

    # 内部函数
        # 分析宏指令、字符、字符串、单行注释、多行注释
        def pre_analyze(self): ...
        # 分析分隔符，如空格和回车
        def analyze_spliters(self): ...
        # 分析关键字与运算符
        def analyze_keywords_and_operators(self): ...
        # 分析整数、浮点数、标识符等单词
        def analyze_words(self): ...
        # 储存分析结果
        def set_component(self, start_index, end_index, component): ...

    # 对外函数
        # 词法分析总函数
        def Analyze(self, code): ...
        # 获取结果接口
        def Get_Result(self): ...
```

3.2.2 语法分析

主要数据结构包括：用于记录项目集的基本项目结构体 Item、语法分析类 SyntaxAnalyzer。该部分也会调用 3.2.1 中的词法分析类 LexAnalyzer、终结符查找树 Trie。其中 SyntaxAnalyzer 为主类。

用于记录项目集的基本项目的结构体：

```
class Item:
    def __init__(self, nump=0, ppos=0, forward=[]):
        self.nump = nump
        self.ppos = ppos
```

```
        self.forward = forward

    def __eq__(self, other):
        return self.nump == other.nump \
               and self.ppos == other.ppos \
               and set(self.forward) == set(other.forward)
```

语法分析的主类：

```
class SyntaxAnalyzer:
    def __init__(self):
        self.VTs = []
        self.VNs = []
        # 记录规则的列表
        # 第一个元素是变元的标号，之后的元素是该变元推出的表达式
        # 如 [-200,-100,45,12] 代表 -200 => -100,45,12
               self.PATs = []

               self.InitVTs()
               self.InitVNs()
               self.InitPATs()
          # 初始化终结符集
            def InitVTs(self): ...
             # 初始化非终结符集
            def InitVNs(self): ...
          # 初始化规则向量
            def InitPATs(self): ...

    # 输出LR(1)分析表到LR1Table.txt 文件
    def LR1_Analyze_file(self, out): ...

    # 以下语法分析函数只给出定义，具体函数的实现将在4.2 中展开
    # 依据格式输入文法
    # 前一个参数输入，后三个参数输出
    def inputGrammar(file_path, get_num, get_string, production): ...
    # 合并左式相同的产生式
    # 前两个参数为输入，后一个参数为输出
    def getProduction(production, get_num, get_produce): ...
    # 通过DFS 算法计算每个文法符号的First 集
    def dfsGetFirst(production, getNum, getProduce, first, nv, nump, getfirst): ...
    def getFirst(production, getNum, getProduce, first): ...
    # 对上一个函数计算得到的First 集进行去重
    # 如First(A)={a,b,a}，则需要去重为{a,b}
    def change_production(production): ...
    def addedge(from1, to, w): ...
```

```
    # 项目与项目集的运算
    def itemInSet(a, b): ...
    def itemSetMerge(a, b):
    def itemSetEqual(a, b): ...
    def findItemSet(a, b): ...
    # 求解项目集闭包
    def getItemClosure(t, get_num, get_produce, production, first): ...
    # 获得项目集族
    def getItemSet(get_num, get_produce, production, first): ...
    # 打印项目集族
    def printItemSet(production, getString): ...
```

3.3 主程序流程

整个程序的整体流程词法分析、语法分析、结果展示等几个部分，其基本流程如下图所示：

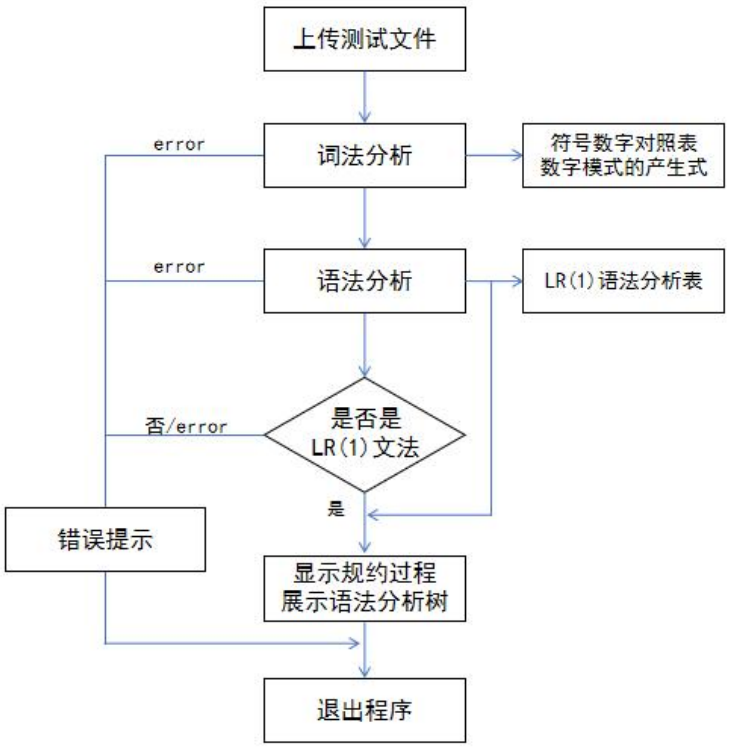


图 9　主程序流程图

系统中各部分调用逻辑如下图所示：左侧为“后端”框架，右侧为“前端”框架。

1.词法分析的具体输出为标准语法文件、预先定义的标准的标识符(VT、NT)映射表、用户上传文件中的标识符表；

2.语法分析输出的具体文件为规约过程、LR(1)分析表、语法树；这两部分在输出时均写入 txt 文件中保存；

3.系统界面负责统筹以上两大模块，将后端系统封装成“黑盒”展示给用户：一方面读取用户操作为词法分析提供输入，另一方面接受语法分析的结果展示给用户。

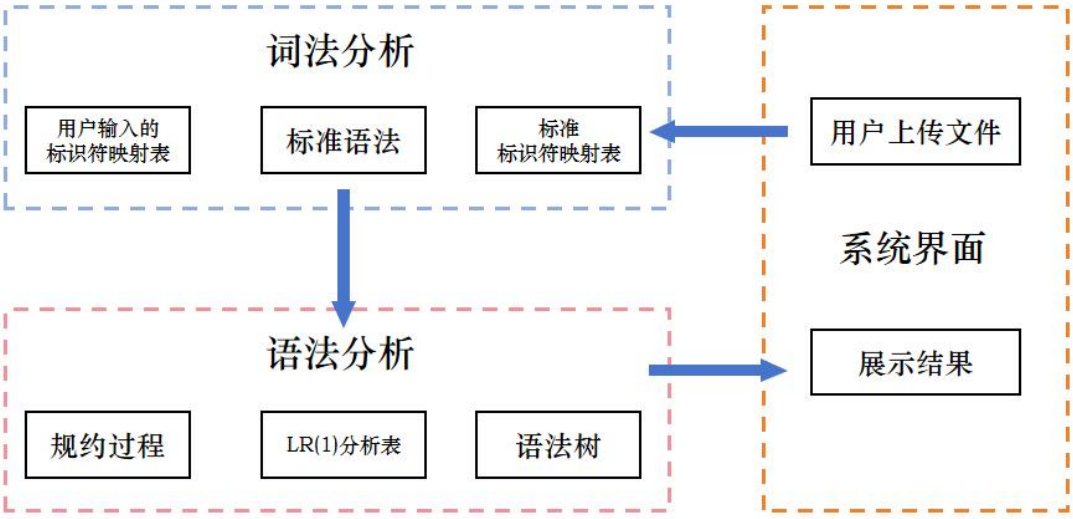


图 10　系统的模块间调用分析

四 详细设计

4.1 词法分析器设计方法

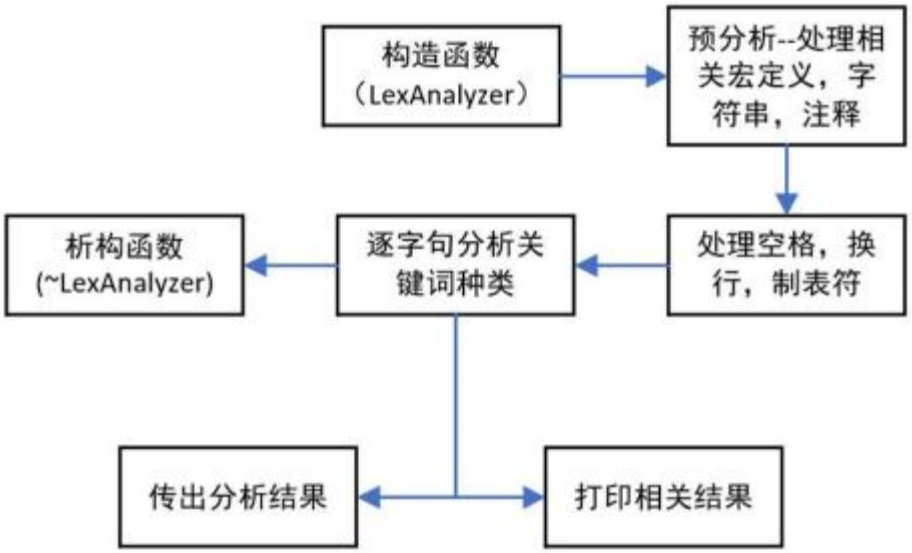


图 11　词法分析的主要流程

4.1.1 LexAnalayzer 类函数

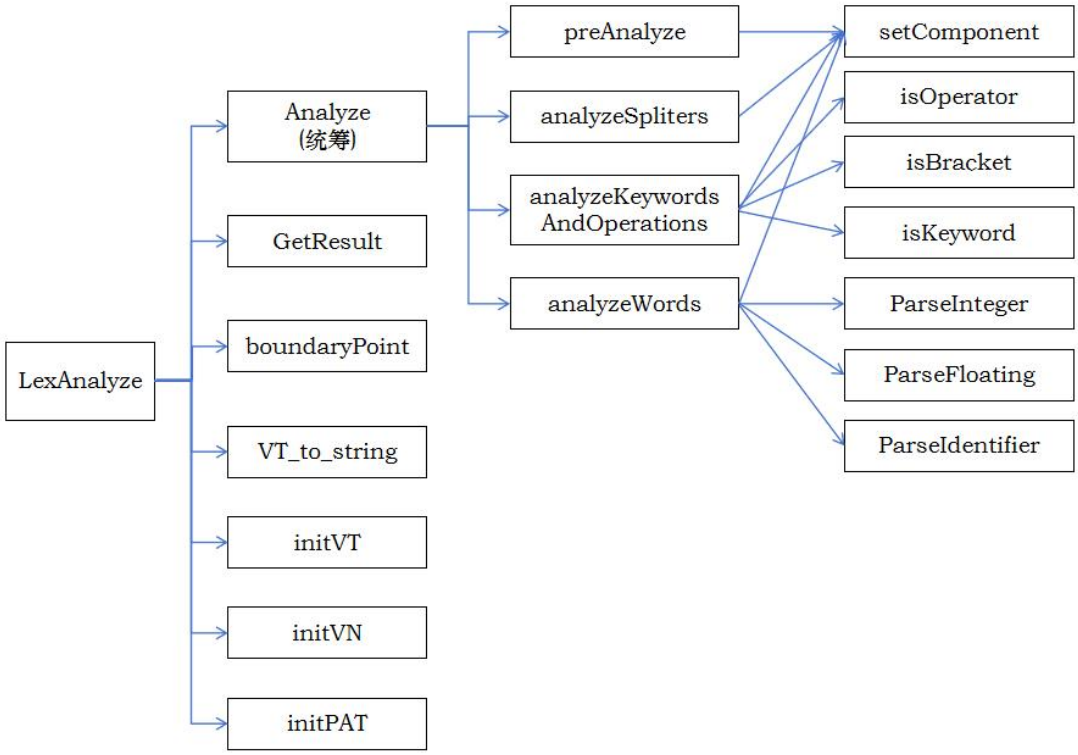


图 12　本作业词法分析部分的函数调用关系示意图

词法分析具体分为 4 步，该部分的函数调用逻辑如上图所示。该部分设计一个函数统筹全过程调度，在 Analyze 函数中分别执行，Analyze 函数内容如下。

```
def Analyze(self, code):
    self.__init__()
    self.original_code = bytearray(code, 'utf-8')
    self.code_length = len(self.original_code)
    self.component_type = [VT.VTNull for _ in range(self.code_length + 1)]
    self.pre_analyze()
    self.analyze_spliters()
    self.analyze_keywords_and_operators()
    self.analyze_words()
```

Pre_analyze 用于分析字符、字符串、单行注释、多行注释。首先从源程序中去除一些无用的注释以及需要替换的宏指令和一些具有干扰性的字符、字符串，其实现主要由几个状态判断 bool 变量来完成。

```
def pre_analyze(self):
    is_single_line_comment = False  # 单行注释
    is_multiline_comment = False  # 多行注释
    is_string = False  # 字符串
    is_char = False  # 字符
    start_index = 0
    for i in range(len(self.original_code)):
        cch = self.original_code[i]
        if is_macro:  # 宏指令
            if not cch or cch == '\n':
                self.set_component(start_index, i, VT.VTMacro)
```

```
                    is_macro = False
            elif is_single_line_comment:  # 单行注释
                if not cch or cch == '\n':
                    self.set_component(start_index, i, VT.VTComment)
                    is_single_line_comment = False
            elif is_multiline_comment:  # 多行注释
                if cch == '/' and self.original_code[i - 1] == '*':
                    self.set_component(start_index, i + 1, VT.VTComment)
                    is_multiline_comment = False
            elif is_string:  # 字符串处理
                if cch == '\"' and self.original_code[i - 1] != '\\':
                    self.set_component(start_index, i + 1, VT.VTString)
                    is_string = False
            elif is_char:  # 字符处理
                if cch == '\'' and self.original_code[i - 1] != '\\':
                    self.set_component(start_index, i + 1, VT.VTCharacter)
                    is_char = False
            elif self.original_code[i:i + 2] == '/*' and i != 0:
                is_multiline_comment = True
                start_index = i
            elif self.original_code[i:i + 2] == '*/' and i != 0:
                is_multiline_comment = False
            elif self.original_code[i:i + 2] == '//' and i != 0:
                is_single_line_comment = True
                start_index = i - 1
            elif self.original_code[i] == '#':
                is_macro = True
                start_index = i
            if not cch:
                return
```

• Analyze_spliters 用于分析分隔符，这一部分较为简单，主要功能是将源程序中的分割符统一，便于之后其他关键字、标识符等词语的分析。

```
def analyze_spliters(self):
    start_index = 0
    is_spliter = False
    tmp_code = [chr(c) for c in self.original_code]
    for i in range(len(tmp_code)):
        if self.component_type[i].value and not is_spliter:
            continue
        cch = tmp_code[i]
        if cch == ' ' or cch == '\t':
            if not is_spliter:
                is_spliter = True
                start_index = i
        else:
            if is_spliter:
                self.set_component(start_index, i, VT.VTWhiteSpace)
                is_spliter = False
        if cch == '\n':
            self.set_component(i, i + 1, VT.VTEndLine)
```

```
        elif not cch:
            break
```

• Analyze_keywords_and_operators 函数分析关键字与运算符。主要采用字典树匹配关键字与运算符，这样可以避免回退，降低时间复杂度。

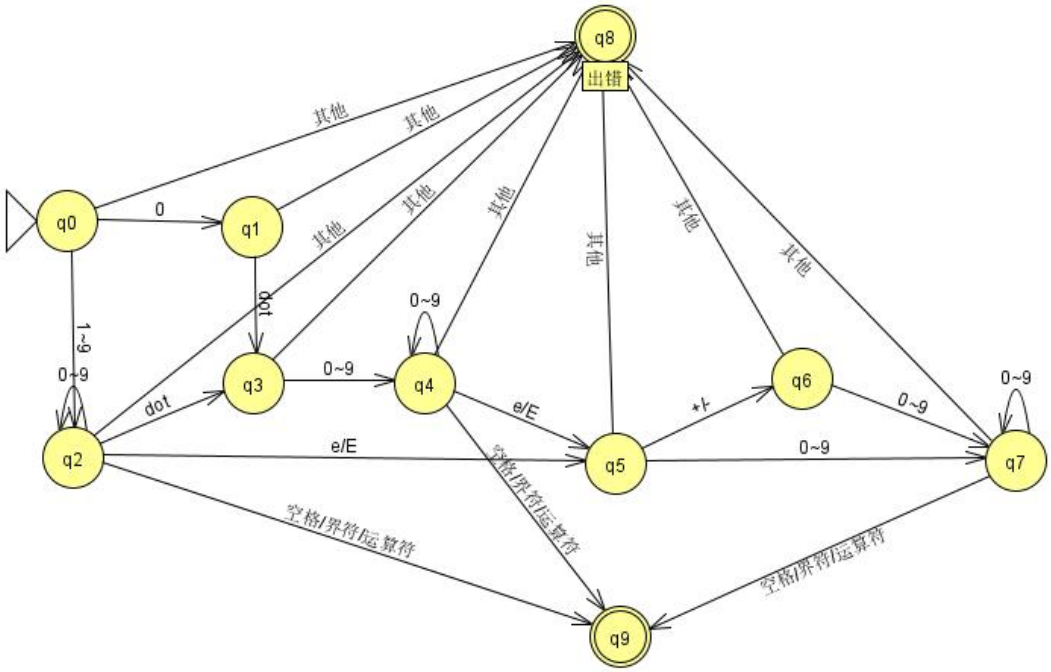
```
def analyze_keywords_and_operators(self):
    # 37: the last printable character
    tmp_code = [chr(c) if c > 37 else ' ' for c in self.original_code]
    # print(tmp_code)
    res = self.trie.match(tmp_code)
    # print(res[4].val)
    # print([VT_to_string(res[i].val) for i in range(len(res))])
    for seg in res:
        if is_operator(seg.val.value):
            self.set_component(seg.begin, seg.end, seg.val)
    for seg in res:
        if is_bracket(seg.val.value):
            self.set_component(seg.begin, seg.end, seg.val)
    for seg in res:
        if is_keyword(seg.val.value):
            if (seg.begin == 0 or self.component_type[seg.begin - 1]) \
              and (seg.end == len(tmp_code) \
              or self.component_type[seg.end]):
                                self.set_component(seg.begin, seg.end, seg.val)
```

• Analyze_words 分析整数、浮点数、标识符和其他单词。

```
def analyze_words(self):
    start_index = -1
    tmp_code = "".join([chr(c) if c > 37 else ' ' for c in self.original_code])
    for i in range(len(tmp_code)):
        if start_index < 0 and (tmp_code[i] and not self.component_type[i].value):
            start_index = i
        elif start_index >= 0 and (not tmp_code[i] or self.component_type[i].value):
            code = tmp_code[start_index:i] + ' '
            if code == "true " or code == "false ":
                self.set_component(start_index, i, VT.VTBoolean)
            elif ParseInteger(code):
                self.set_component(start_index, i, VT.VTInteger)
            elif ParseFloating(code):
                self.set_component(start_index, i, VT.VTFloating)
            elif ParseIdentifier(code):
                self.set_component(start_index, i, VT.VTIdentifier)
            else:
                self.set_component(start_index, i, VT.VTWord)
            start_index = -1
```

4.1.2 Parse_Float

本程序在设计词法分析时还针对无符号浮点数设置了 parse 函数，可以识别 1.2 普通形式和 1e-3 科学计数形式的浮点数。DFA 转移图如下：



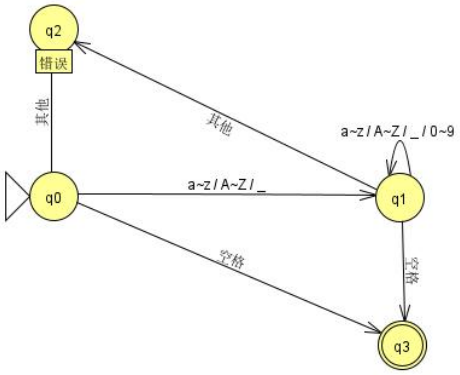
代码实现如下：

```
def ParseFloating(string):
    state = '0'  # 初始状态
    i = 0
    n = len(string)
    while state != '9' and state != '8' and i < n:
        ch = string[i]
        if state == '0':
            if ch >= '1' and ch <= '9':
                state = '1'  # 1-9，转状态 1
            elif ch == '0':
                state = '2'  # 0，转状态 2
            else:
                state = '8'  # 其他，转状态 8
        elif state == '1':
            if ch >= '0' and ch <= '9':
                state = '1'  # 0-9，转状态 1
            elif ch == 'e' or ch == 'E':
                state = '5'  # e ，转状态 5
            elif ch == '.':
                state = '3'  # . ，转状态 3
            elif ch == ' ':
                state = '9'  # 结束，转 9
            else:
                state = '8'  # 其他，转状态 8
        elif state == '2':
            if ch == '.':
                state = '3'  # . ，转状态 3
            else:
                state = '8'  # 其他，转状态 8
        elif state == '3':
            if ch >= '0' and ch <= '9':
                state = '4'  # 0-9，转状态 4
            else:
```

```
                state = '8'  # 其他，转状态 8
        elif state == '4':
            if ch >= '0' and ch <= '9':
                state = '4'  # 0-9，转状态 4
            elif ch == 'e' or ch == 'E':
                state = '5'  # e ，转状态 5
            elif ch == ' ':
                state = '9'  # 结束，转 9
            else:
                state = '8'  # 其他，转状态 8
        elif state == '5':
            if ch >= '0' and ch <= '9':
                state = '7'  # 0-9，转状态 7
            elif ch == '+' or ch == '-':
                state = '6'  # +/-，转状态 6
            else:
                state = '8'  # 其他，转状态 8
        elif state == '6':
            if ch >= '0' and ch <= '9':
                state = '7'  # 0-9，转状态 7
            else:
                state = '8'  # 其他，转状态 8
        elif state == '7':
            if ch >= '0' and ch <= '9':
                state = '7'  # 0-9，转状态 7
            elif ch == ' ':
                state = '9'  # 结束，转 9
            else:
                state = '8'  # 其他，转状态 8
        i += 1
    return state == '9'
```

4.1.3 Parse_Identifier

本程序在设计的标识符处理：



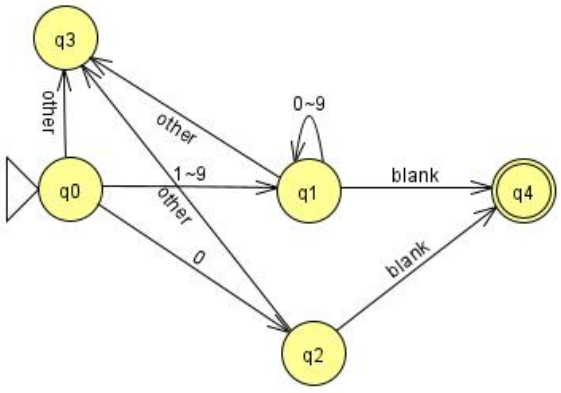
代码实现如下：

```
def ParseIdentifier(string):

    state = '0'  # 初始状态

    i = 0
```

```
    n = len(string)
    while state != '3' and state != '2' and i < n:
        ch = string[i]
        i += 1
        if state == '0':
            if ch >= 'a' and ch <= 'z' or ch >= 'A' and ch <= 'Z' or ch == '_':
                state = '1'  # 转状态 1
            elif ch == ' ':
                state = '3'  # 转状态 3
            else:
                state = '2'
        elif state == '1':
            if ch >= 'a' and ch <= 'z' or ch >= 'A' and ch <= 'Z' or ch >= '0' and ch <= '9' or ch == '_':
                state = '1'  # 转状态 1
            elif ch == ' ':
                state = '3'  # 转状态 3
            else:
                state = '2'
    return state == '3'
```

4.1.3 Parse_Integer

本作业设计的识别整数的 DFA 如下：



代码实现如下：

```
def ParseInteger(string):
    state = '0'  # 初始状态
    i = 0
    n = len(string)
    while state != '3' and state != '4' and i < n:
        ch = string[i]
        if state == '0':
            if '1' <= ch <= '9':
                state = '1'  # 1-9，转状态 1
            elif ch == '0':
                state = '2'  # 0，转状态 2
            else:
                state = '3'  # 其他，转状态 3
        elif state == '1':
            if '0' <= ch <= '9':
                state = '1'  # 0-9，转状态 1
            elif ch == ' ':
                state = '4'  # 结束，转 4
            else:
                state = '3'  # 其他，转状态 3
        elif state == '2':
            if ch == ' ':
                state = '4'  # 结束，转 4
            else:
                state = '3'  # 其他，转状态 3
        i += 1
    return state == '4'
```

4.2 语法分析器设计方法

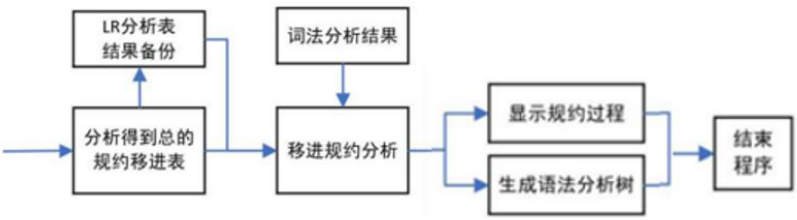


图 13　语法分析器的基本设计思路

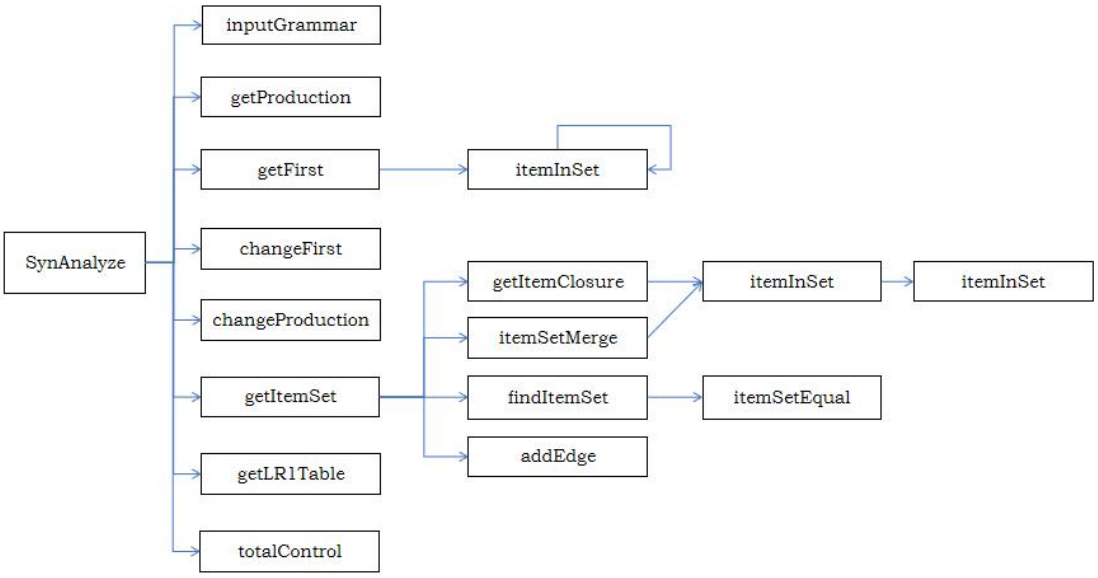


图 14　本作页中的语法分析模块的函数调用分析

4.2.1 SyntexAnalyzer 类函数

文法处理模块与语法分析流程如上图所示。而整个语法分析的函数内部调用网展示如下。

```
def SyntexAnalyzer():
# 变量初始化
    init()
    # 输入终结符、非终结符、产生式
    inputGrammar("./NEW_FOLDER/4.SyntaxGrammar.txt", getNum, getString, production)
    # 输入终结符、非终结符的哈希对应关系
    with open("./NEW_FOLDER/0.LexSynCheckList.txt", "r") as in1:
        for line in in1:
            symbol, index = line.split()
            V[index] = symbol
    # 合并左侧相同的产生式
    getProduction(production, getNum, getProduce, getString)
    # 得到 First 集
    getFirst(production, getNum, getProduce, first, getString)
    # 非终结符去重
    change_production(production)
    # 特殊处理空产生式
    getItemSet(getNum, getProduce, production, first)
    # 得到项目集族
    printItemSet(production, getString)

    # 获得 LR1 分析表（包括 ACTION 表和 GOTO 表）
    LR1_info_path = "./NEW_FOLDER/LR1_info_tuple.txt"
    if not getLR1Table(production):
        print("This Grammar is not a LR(1) Grammar !")
        return -2
```

```
    printLR1Table(getString, LR1_info_path)

    # 输出中间结果到 txt 文件中
        LRTableBackup1(folder + "backup_Table.txt", len(ItemSet) + 1, num + 2)
        LRTableBackup2(folder + "backup_tb_s_r.txt", len(ItemSet) + 1, num + 2)
        LRProductionBackup(folder + "backup_production.txt", production)
        # 读取词法分析结果利用总控程序分析句子
    # 分析结束后将产生整个规约过程的文件、语法树图片的 dot 文件
        with open(folder + "2.LexResultList.txt", "r") as inpu:
            for line in inpu:
                if int(line) < 0:
                    break
                word.append(int(line[:-1]))
            word.append(2147483647)
        if not totalControl(getString, getNum, production, printTreeTerminal):
            print("error!")
            return -1
        else:
            return 0
```

4.2.2 inputGrammar

在进行文法处理分析之前需要先依据格式读入预定文法。即这里的 inputGrammar 函数。该函数中前一个参数输入，后三个参数输出。

```
def inputGrammar(file_path, get_num, get_string, production):
    global numvt, num, nume, word, V, head, ItemSet, edge, tb_s_r, table
    n1, n2, n3 = 0, 0, 0  # 分别记录终结符、非终结符、产生式的数目
    n = 0
    get_num['2147483647'] = n  # 注意 get_num 的 key 是 str!!!!!!!!!!
    get_string[n] = '2147483647'  # INT_MAX 代表#号
    n += 1
    with open(file_path, 'r') as f:
        n1, n2, n3 = map(int, f.readline().split())
        numvt = n1 + 1
        num = n1 + n2 + 1

        # terminals
        terminals = f.readline().split()
        for i in range(len(terminals)):
            get_num[terminals[i]] = n
            get_string[n] = terminals[i]
            n += 1;

        get_num['-1000'] = n
```

```
        get_string[n] = -1000
        n += 1

        # non-terminals
        non_terminals = f.readline().split()
        for i in range(len(non_terminals)):
            get_num[non_terminals[i]] = n
            get_string[n] = non_terminals[i]
            n += 1;

        # grammars
        for i in range(n3):
            file_line_buf = list(map(int, f.readline().split()))
            grammar_tmp = file_line_buf[1:]
            production.append(grammar_tmp)
```

4.2.3 获取 First 集的函数

GetFirst 算法原理如下，使用以下规则，直至每一个非终结符的 FIRST 集合不在增大为止，则可以获得相应的 First 集：

1.若 X 属于 VT，则 FIRST（X）= {X}；

2）若 X 属于 VN，且有产生式 X -> a…，则把 a 加入 FIRST；若 S -> ε也是产生式，则把 ε 也加入到 FIRST（X）中；

3）若 X -> Y……是一个产生式且 Y 属于 VN，则把 FIRST（Y）中所有的非空元素都加入到 FIRST（X）中，若 X -> Y1Y2…Yk 是一个产生，Y1,Y2,…,Yi-1 都是非终结符，而且对于任何满足 1 <= j <= i - 1，First（Yj）都含有 ε ，则把 FIRST（Yi）中所有的非 ε 元素都加入到 FIRST（X）中，特别的，若所有 FIRST（Yj）对于 j = 1，2，…, k 都含有 ε ，则将 ε 也加入到 FIRST(X) 中。

该部分包括 dfsGetFirst、getFirst、changeProduction 共三个函数：

• dfsGetFirst 函数通过 DFS 算法计算每个文法符号的 First 集

```
def dfsGetFirst(production, getNum, getProduce, first, nv, nump, getfirst):

    global numvt, num, nume, word, V, head, ItemSet, edge, tb_s_r, table

    temp = getNum[str(production[nump][1])]

    getfirst[nump] = True

    if temp <= numvt:

        first[nv].append(temp)

    else:
```

```
        for j in range(1, len(production[nump])):
            temp = getNum[str(production[nump][j])]
            for i in range(len(getProduce[temp])):
                if production[nump][0] == production[nump][1]:
                    continue
                dfsGetFirst(production, getNum, getProduce, first, temp, getProduce[temp][i], getfirst)
            if numvt not in first[temp]:   # numvt -----> \epsilon
                first[nv] = first[nv] + first[temp]
                break
            tmp_first_temp = cp.deepcopy(first[temp])
            tmp_first_temp.remove(numvt)
            first[nv] = first[nv] + tmp_first_temp
            if j == len(production[nump]) - 1:
                first[nv].append(numvt)
```

- GetFirst 函数调用 dfsGetFirst 获取所有终结符的 First 集

```
def getFirst(production, getNum, getProduce, first):
    global numvt, num, nume, word, V, head, ItemSet, edge, tb_s_r, table
    getfirst = [False for _ in range(MAX_N)]  # 初始化
    for i in range(1, numvt + 1):  # 终结符 first 集合是它自己。 first 集合通过终结符的编号来存储
        first[i].append(i)
    for i in range(len(production)):
        if (production[i][0] == production[i][1]):
            continue
        if (getfirst[i]):
            continue
        temp = getNum[str(production[i][0])]
        dfsGetFirst(production, getNum, getProduce, first, temp, i, getfirst)

    for i in range(len(first)):
        first[i] = list(set(first[i]))
```

• changeProduction 函数对上一个函数计算得到的 First 集进行去重如 First(A)={a,b,a}，则需要去重为{a,b}

```
def change_production(production):

    global numvt, num, nume, word, V, head, ItemSet, edge, tb_s_r, table

    for i in range(len(production)):

        if production[i][1] == -1000:

            production[i].remove(-1000)
```

4.2.4 构造项目集闭包

假定 I 是一个项目集，它的闭包 CLOSURE（I）可按如下方式构造：

（1） I 的任何项目都属于 CLOSURE（I）

（2）若项目[A -> α,ß] 属于 CLOSURE（I），B -> ϒ是一个产生式，那么，对于 FIRST（ß）中的每个终结符 b，如果[B -> ϒ, ß]原来不在 CLOSURE（I）中，则把它加进去；

（3）重复执行步骤（2）直至 CLOSURE（I）不再增大为止。

在实际代码编写中，我们主要采用广度优先搜索算法 BFS 求解项目的闭包，在确定项目的展望字符时，需要用到之前计算的 FIRST 集。主要通过 getItemClosure 函数实现，其功能为求解并构造项目集闭包

```
def getItemClosure(t, get_num, get_produce, production, first):
    global numvt, num, nume, word, V, head, ItemSet, edge, tb_s_r, table
    temp = [t]
    q = Queue()
    q.put(t)
    while not q.empty():
        cur = q.get()
        if cur.ppos == len(production[cur.nump]):
            continue
        tt = get_num[str(production[cur.nump][cur.ppos])]
        if tt <= numvt:
            continue
        for i in range(len(get_produce[tt])):
            c = Item(ppos=1, nump=get_produce[tt][i])

            if len(production[cur.nump]) - cur.ppos == 1:
                c.forward = cur.forward + c.forward
            else:
                for j in range(1, len(production[cur.nump])):
                    tttnum = get_num[str(production[cur.nump][cur.ppos + j])]

                    if numvt not in first[tttnum]:
                        c.forward = first[tttnum] + c.forward
                        break
```

```
                    tmp_first_tttnum = cp.deepcopy(first[tttnum])
                    tmp_first_tttnum.remove(numvt)
                    c.forward = c.forward + tmp_first_tttnum
                    if cur.ppos + j == len(production[cur.nump]) - 1:
                        c.forward = cur.forward + c.forward
                        break
            if not itemInSet(c, temp):
                q.put(c)
                temp.append(c)
    return temp
```

4.2.4 构造项目集族

构造有效的 LR（1）项目集族的办法本质上和构造 LR（0）项目集规范族的办法是一样的。都需要两个函数 CLOSURE 和 GO，上一小节已经构造出 CLOSURE 函数，因此还需要构造 GO 函数，定义如下：

令 I 是一个项目集，X 是一个文法符号，函数 GO（I, X）定义为 GO（I, X）= CLOSURE（J）。关于文法 G 的 LR（1）项目集族的构造算法为：

```
BEGIN:
    C := {Closure([S→•S,#])};
    REPEAT:
        FOR C中的每一个项目集I和G的每个符号X
            IF Go(I,X)非空且不属于C, THEN
                把Go(I,X)加入C中
    UNTIL C不再增大
END
```

其具体代码实现如下。该部分包括两个函数：getItemSet 和 printItemSet。具体实现代码时，我们仍采用 BFS 算法求解项目集族，实际上最后构造的项目集转移图也是一个 DFA，我们采用记录图的方式来记录这个 DFA。

- getItemSet 函数通过 BFS 获得获得项目集族

```
def getItemSet(get_num, get_produce, production, first):
    global numvt, num, nume, word, V, head, ItemSet, edge, tb_s_r, table
    global q, t
    temp = []
    t = Item(nump=0, ppos=1, forward=[0])
    temp = getItemClosure(t, get_num, get_produce, production, first)
    q = Queue()
    q.put(temp)
    ItemSet.append(temp)  # S -> .BB, #

    while not q.empty():
        cur = q.get()
        for i in range(1, num + 1):  # 所有符号
            if i == numvt:
                continue  # 空字符
```

```
            temp = []
            for j in range(len(cur)):
                if cur[j].ppos == len(production[cur[j].nump]):  # 是规约项目，无法再读入
                    continue
                tt = get_num[str(production[cur[j].nump][cur[j].ppos])]
                if tt == i:
                    tempt = Item(ppos=cur[j].ppos + 1, nump=cur[j].nump, forward=cur[j].forward)
                    temp = itemSetMerge(temp, getItemClosure(tempt, get_num, get_produce, production, first))
            if len(temp) == 0:
                continue  # 该符号无法读入
            num_cur = findItemSet(cur, ItemSet)  # 当前节点标号
            tttnum = findItemSet(temp, ItemSet)  # 新目标标号
            if tttnum == -1:  # 新的项目集
                ItemSet.append(temp)
                q.put(temp)
                addedge(num_cur, len(ItemSet) - 1, i)  # 添加边，权为读入的符号
            else:
                addedge(num_cur, tttnum, i)
# 打印项目集族
def printItemSet(production, getString):
    global numvt, num, nume, word, V, head, ItemSet, edge, tb_s_r, table
    for i in range(len(ItemSet)):
            j = head[i]
            while j != -1:
                # print("   " + getString[edge[j][2]] + "\n" + str(i) + "--->" + str(edge[j][0]))
                j = edge[j][1]
```

4.2.5 获得 LR(1)分析表

从文法的 LR（1）项目集族 C 构造分析表的算法如下：假定 C＝{I0，I1，…，In}，令每一个 Ik 的下标 k 为分析表的状态。令那个含有[S’ -> .S,#]的 Ik 的 k 作为分析器的初态。动作 ACTION 和状态转换 GOTO 可构造如下：

（1）若项目[A -> α, ß]属于 Ik，且 GOTO（Ik, a）＝Ij（其中 a 为终结符），则置 ACTION[k, a]为“把状态 j 和符号 a 移进栈”，简记为“aj”。

（2）若项目[A -> α, ß]属于 Ik，则置 ACTION[k, a]为“用产生式 A -> ”规约”，简记为“rj”；其中假定 A -> 为文法 G’的第 j 个产生式。

（3）若项目[S’ -> S., #]属于 Ik，则置 ACTION[k, #]为“接受”，简记为“acc”

（4）若 GO（Ik, a）＝Ij，则置 GOTO[k, A]＝j。

• 具体实现的 python 函数如下，其中 table[i][j] = w:状态 i --> j, 读入符号 W：

```
def getLR1Table(production):

    global numvt, num, nume, word, V, head, ItemSet, edge, tb_s_r, table

    print(f"Size of ItemsSet: {len(ItemSet)}")
```

```
    for i in range(len(ItemSet)):  # 遍历图
        j = head[i]
        while j != -1:  # 遍历边
            if table[i][edge[j][2]] != -1:
                return False  # 多重入口，报错.
            table[i][edge[j][2]] = edge[j][0]
            tb_s_r[i][edge[j][2]] = -1  # 移近项-1
            j = edge[j][1]
    for i in range(len(ItemSet)):  # 遍历所有项目
        for j in range(len(ItemSet[i])):
            if ItemSet[i][j].ppos == len(production[ItemSet[i][j].nump]):  # 归约项
                for k in range(len(ItemSet[i][j].forward)):
                    if table[i][(ItemSet[i][j].forward)[k]] != -1:
                        return False  # 多重入口，报错.
                    if (ItemSet[i][j].forward)[k] == 0 and ItemSet[i][j].nump == 0:
                        table[i][(ItemSet[i][j].forward)[k]] = -3  # 接受态
                    else:
                        table[i][(ItemSet[i][j].forward)[k]] = ItemSet[i][j].nump
                        tb_s_r[i][(ItemSet[i][j].forward)[k]] = -2  # 归约态
    return True
```

- 此外，我们还额外定义了 LR(1) 分析表的打印函数

```
def printLR1Table(getString, LR1_info_path):
    global numvt, num, nume, word, V, head, ItemSet, edge, tb_s_r, table
    LR1_info_file = open(LR1_info_path, 'w')
    with open("./NEW_FOLDER/5.LR1Table.txt", "w") as out:
        out.write(f"{len(ItemSet)} {num}\n")
        for j in range(num + 1):
            if j == numvt:
                continue
            out.write(f"    {getString[j]}")
        out.write('\n')
```

```
        for i in range(len(ItemSet)):
            for j in range(num + 1):
                if j == numvt:
                    continue
                msg = ""
                if table[i][j] == -3:
                    msg = "acc "
                elif table[i][j] == -1:
                    msg = "* "
                elif tb_s_r[i][j] == -1:
                    msg = f"s{table[i][j]} "
                elif tb_s_r[i][j] == -2:
                    msg = f"r{table[i][j]} "
                out.write(msg)
                if msg != "* ":
                    LR1_info_file.write(f"{i} {j} {msg}\n")
            out.write("\n")
    LR1_info_file.close()
```

4.2.6 生成语法树

语法树绘制的难点在于语法树的构造，因为 LR（1）分析是自下而上的分析，而由下而上的构造一棵树是比较困难的，我们的解决办法是在总控程序进行规约的时候，把规约用到的产生式存到一个栈里面，规约成功后再逐个取出，从根节点 S 开始构造语法树。因为 LR（1）是最左规约的，对应着最右推导，所以每次只需要取最右边的非终结符节点进行推导展开，将该非终结符节点作为根节点构造子树，如此完成最终语法树的构造。
最终绘制语法树。

本作也使用 dot 语言和 GraphViz 库生成了比较美观的语法树图片，如下。

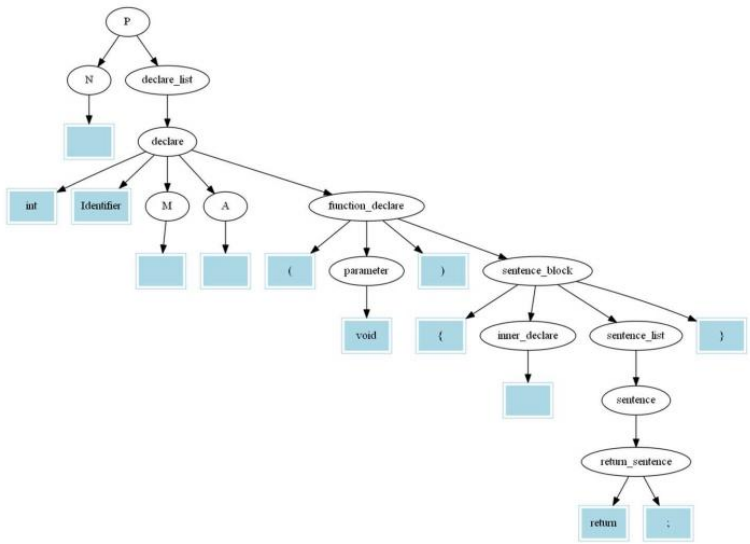


图 15　Graphviz 读取 dot 文件作出的语法树图片

Graphviz 即 Graph Visualization Software，是一个开源的图形可视化软件，它能够从简单的文本文件描述中生成复杂的图形和网络。它使用一种名为 DOT 的描述语言来定义图形，使得用户可以专注于内容而非布局和设计。其主要特点和用途包括：可以生成 PNG、

PDF、SVG 等多种格式的图形；用户只需在 dot 中定义图的元素和它们之间的关系，函数库可以实现自动布局；使用于 Web 服务、生成报告、其他软件的集成等，泛用性强。

• 本作业中具体代码写入语法树的 dot 文件的代码如下：

```
def totalControl(getString, getNum, production, printTree):
    with open("./D_NEW_FOLDER/6.AnalyzeProcess.txt", "w") as out:
        regularset = deque()  # 规约顺序
        print("步骤", "状态栈", "符号栈", "输入串", "动作说明", sep=" ", file=out)
        state = deque()  # 状态栈和符号栈
        state.append(0)

        wd = deque()  # '#'
        wd.append(0)
        count = 0
        i = 0
        while True:
            cur = state[-1]
            if table[cur][getNum[str(word[i])]] == -1:  # 空白，报错误
                return 0
            if table[cur][getNum[str(word[i])]] == -3:  # 接受态
                regularset.append(production[0])  # 添加规约顺序
                state_ = cp.deepcopy(state)
                wd_ = cp.deepcopy(wd)
                printCurState(count, state_, wd_, i, getString,out)
                count += 1
                out.write("acc!\n")
                getTree(regularset, printTree)
                return 1
            if tb_s_r[cur][getNum[str(word[i])]] == -1:  # 移进项
                state_ = cp.deepcopy(state)
                wd_ = cp.deepcopy(wd)
                printCurState(count, state_, wd_, i, getString,out)
                count+=1
                print("action[", cur, ",", V[str(getNum[str(word[i])])], "]=", table[cur][getNum[str(word[i])]],
",", "状态", table[cur][getNum[str(word[i])]], "入栈", sep=" ", file=out)
                wd.append(getNum[str(word[i])])
                state.append(table[cur][getNum[str(word[i])]])
                i += 1
            elif tb_s_r[cur][getNum[str(word[i])]] == -2:  # 归约
                state_ = cp.deepcopy(state)
                wd_ = cp.deepcopy(wd)
                printCurState(count, state_, wd_, i, getString, out)
                count += 1
                numpro = table[cur][getNum[str(word[i])]]
```

```
                len_ = len(production[numpro]) - 1
                for j in range(len_):
                    state.pop()
                    wd.pop()
                wd.append(getNum[str(production[numpro][0])])
                cur1 = state[-1]  # ---------------------------------
                print("用", V[str(production[numpro][0])], "->", sep=" ", end="", file=out)
                for j in range(1, len_ + 1):
                    print(V[str(production[numpro][j])], end=",", file=out)
                regularset.append(production[numpro])  # 添加规约顺序
                print("进行归约,", "goto[", cur1, ",", V[str(getNum[str(word[i])])], "]=",
table[cur1][getNum[str(production[numpro][0])]], ",", "入栈", sep=" ", file=out)
                state.append(table[cur1][getNum[str(production[numpro][0])]])
        return 1
```

• 本作业中将决策树 dot 文件可视化为. png 文件的函数如下：

```
def dot2png(dot_file_path=None, img_path=None):
    if not dot_file_path:
        raise Exception(".dot file is not given.")
    elif not dot_file_path.endswith('.dot'):
        raise Exception("file provided is not '.dot' type.")
    DOT_PATH = check_valid_path(dot_file_path)
    if not img_path:
        img_path = 'dt_png.png'
    elif not img_path.endswith('.png'):
        raise Exception("image file not end with '.png'.")
    IMG_PATH = img_path
    cmd_args = ['dot', '-Tpng', DOT_PATH, '-o', IMG_PATH]
    cmd_pro = subprocess.Popen(args=cmd_args, stdout=subprocess.PIPE, stderr=subprocess.STDOUT)
    retval = cmd_pro.stdout.read().decode('gbk')
    if retval == '':
        print("successfully create file " + IMG_PATH)
    else:
        print("The program encountered some error: ")
        print(retval)
```

4.3 系统呈现

由于 python 本身就有 tkinter 这类前端组件库，直接在代码中使用可以省去编写后端的麻烦，减少了开发成本，因此我们选用 tkinter 库搭建前端框架，形成整套语法分析系统。

1. 欢迎界面+主界面

欢迎界面包括欢迎字符画与“文件上传”“词法分析”“语法分析”三个按钮。未上传文件时，词法分析和语法分析不允许点击，避免误触和重复计算等情况。

图 16 主页面示意图

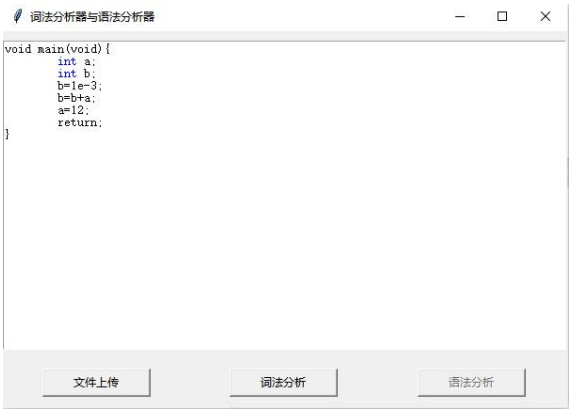


图 17 上传代码后的主页面示意图

2. 词法分析展示

上传完毕后词法分析按钮生效，点击后弹窗通过表格展示分析结果，直观清晰。

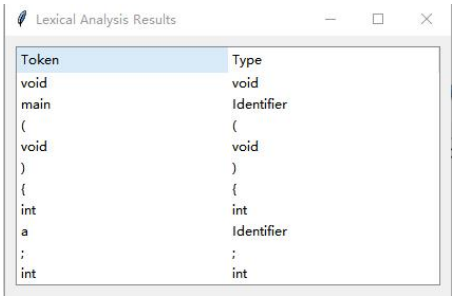


图 18 词法分析结果示意图

3. 语法元素展示

词法分析完毕后语法分析按钮生效，展示语法元素和语法推导规则。

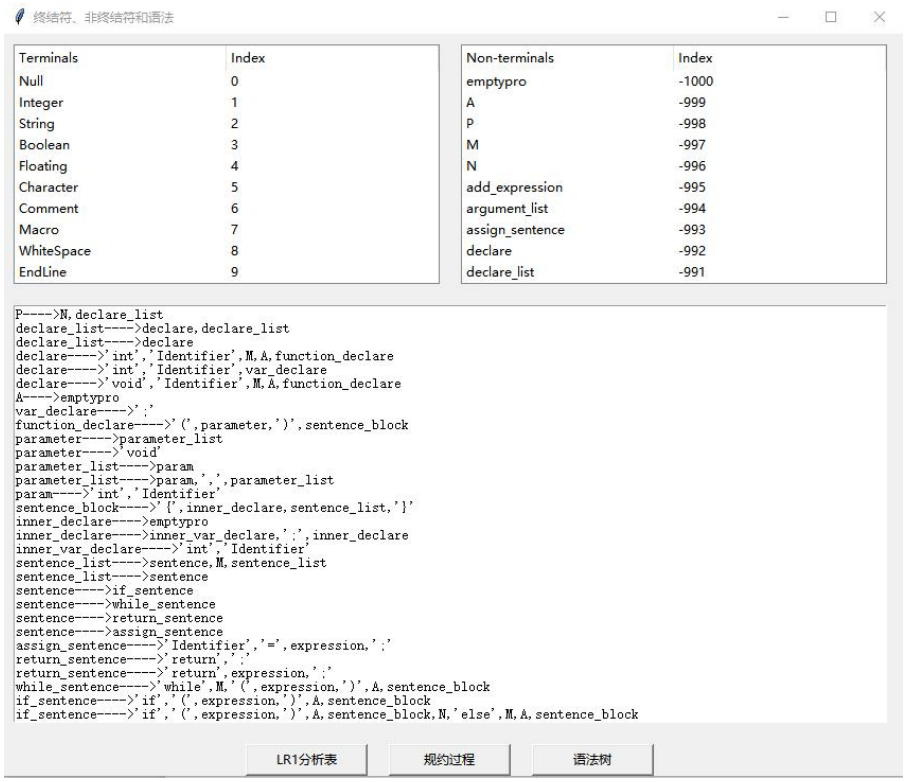


图 19 语法分析结果示意图

4. LR1 表、移进规约过程、语法树展示

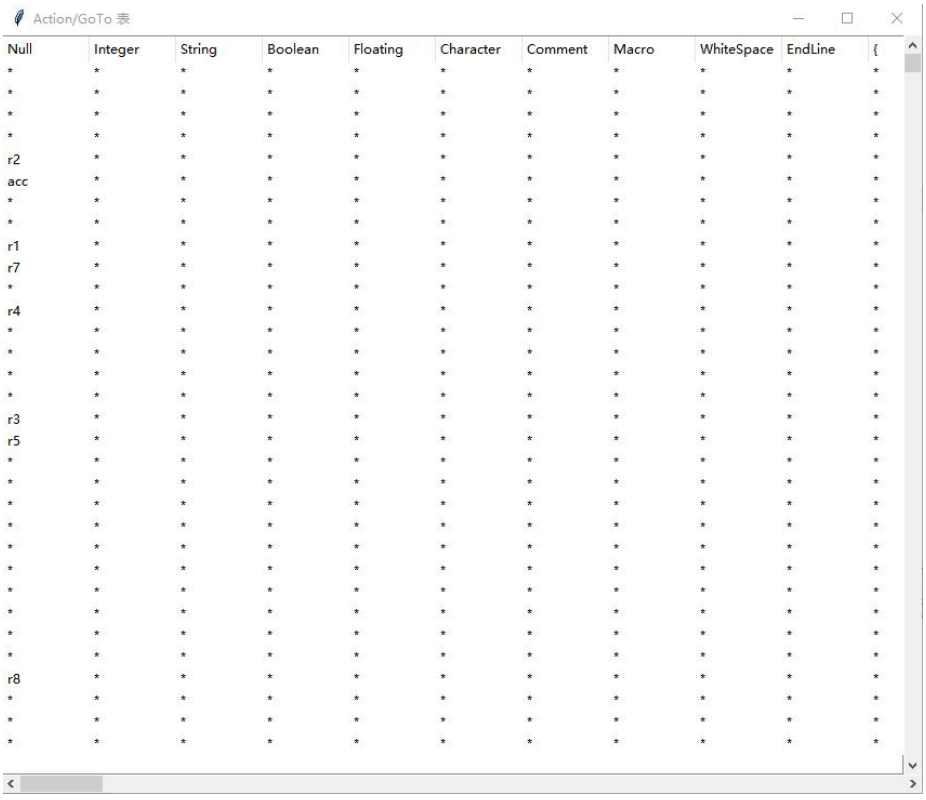


图 20 LR(1)分析表示意图

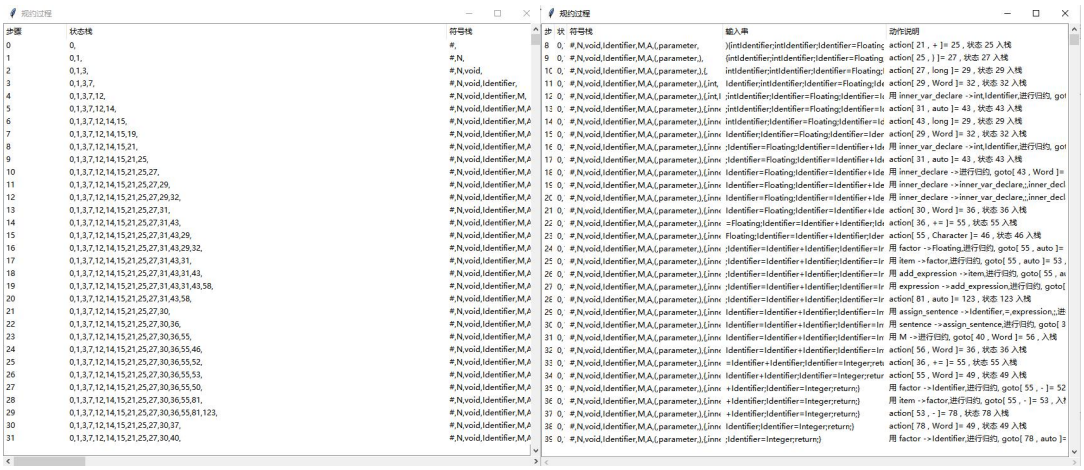


图 21　总规约过程示意图，包括状态栈、符号栈、操作说明

五 调试分析

5.1 测试数据

准备的类 C 代码的正确输入、错误输入示例在 2.4 中已展示。

5.1.1 正确数据分析

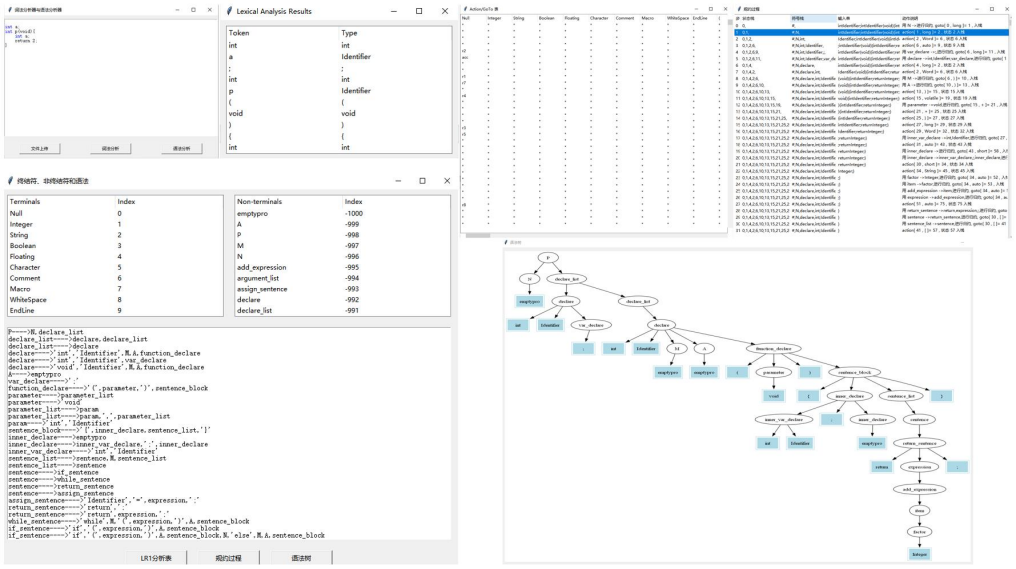


图 22　1.c 文件上传后进行相关词法语法分析示意图

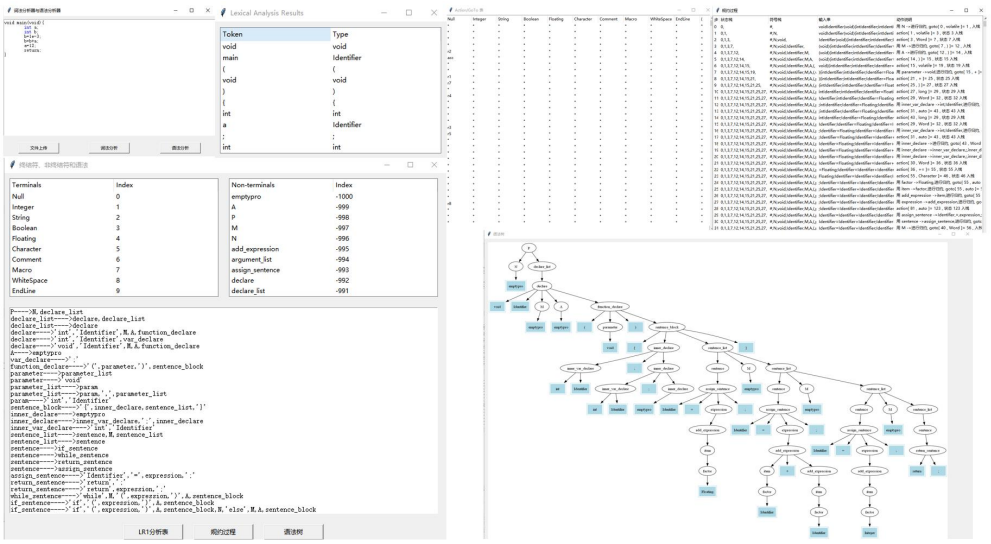


图 23　2.cpp 文件上传后进行相关词法语法分析示意图

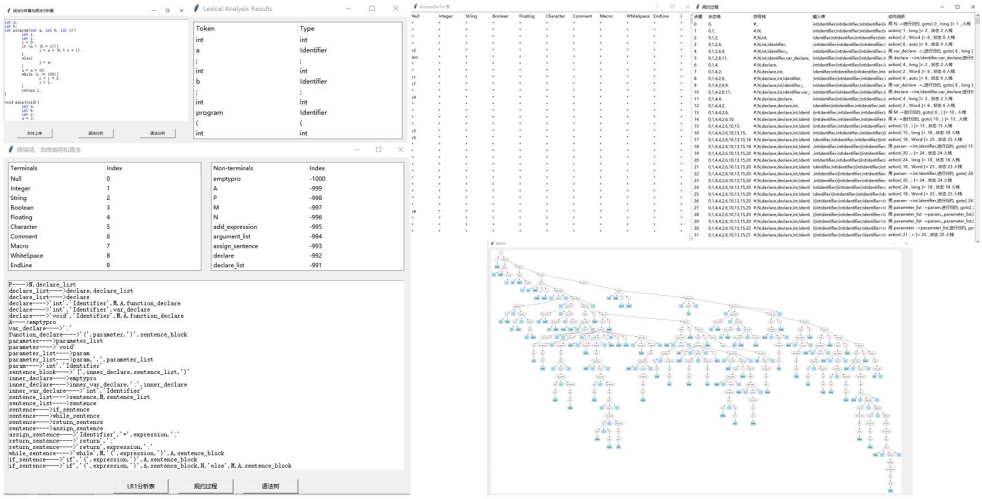


图 24　3.txt 文件上传后进行相关词法语法分析示意图

5.1.2 错误数据分析

4.c 中设置错误为缺少分号

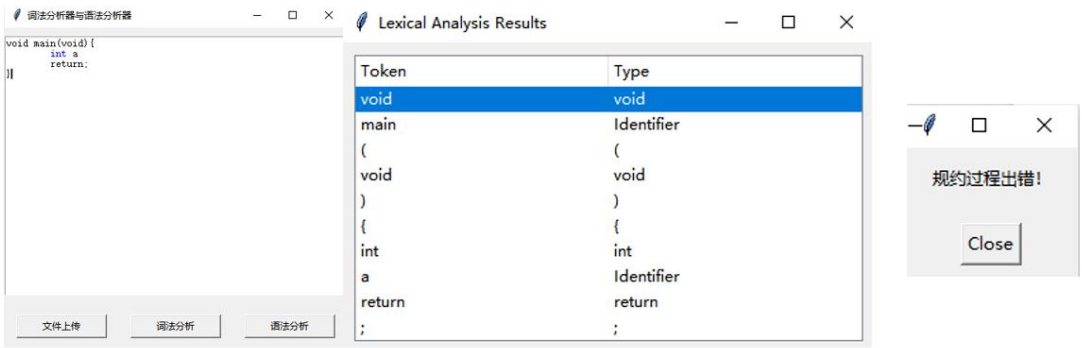


图 25　4.c 文件上传后进行词法分析的结果与报错示意图

5.cpp 中设置错误为赋值语句的语法错误

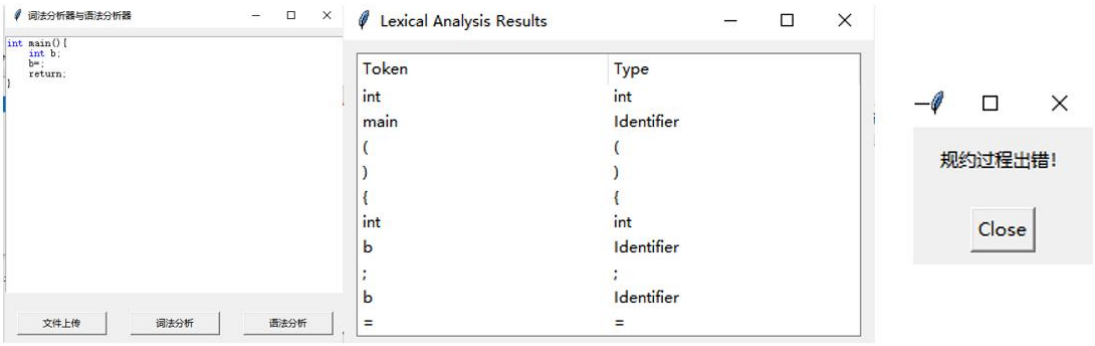


图 26　5.cpp 文件上传后进行词法分析的结果与报错示意图

5.2 复杂度分析

5.2.1 词法分析

假设不存在词法错误。词法分析过程，采用先识别，再提取的方式，相当于对整个文件进行 2 次扫描。其中，第一次扫描借助自动机完成。因此，对于包含 n 个转移的自动机，总长度为 x 的文件，总体时间复杂度为$O(x \cdot log\ n + x)$，可视为$O(x \cdot log\ n)$。

5.2.2 Action Goto 表构建

该过程涉及多个步骤，涉及的映射表结构全部使用哈希表，以降低时间复杂度。语法展开的时间复杂度与语法所含子产生式数量呈线性。

First 集合构造的复杂度相对符号总数呈线性。初状态构建的耗时操作发生在求闭包环节。求闭包的复杂度与产生式总数呈线性，但用时远不及线性时间。状态转移前，寻找转移符号的复杂度为状态内产生式的数量。转移过程复杂度与状态内产生式数量相同。之后则是求闭包。

构建状态表理论上复杂度可以达到指数级，程序在这部分停留的时间也最多，可以达到数秒甚至十几秒。

构建转移表相对简单，其复杂度与所有状态内所有产生式的数量总和呈线性关系。

5.2.3 语法分析

借助构建完毕的语法分析表，即可完成对符号串的分析。由于语法分析表采用类似于哈希表的结构存储，查询复杂度为常数级别。因此，语法分析所需时间复杂度与符号串长度呈线性。

5.2.4 整体开销

如果从文法定义文件构建 LR(1)分析表，带来的时间开销主要与状态转移计算有关。该过程十分缓慢，需要花费数秒时间。

5.3 调试时遇到的问题与思考

5.3.1 字典树的应用

在进行词法分析的过程中，各类字符、关键字等每次读取时都需要进行匹配，速度较慢。联想到数据结构课上所学，我们在这方面应用字典树来提高查找效率，不用每读入一个字符都在列表里查匹配，只要判断是否到达叶结点即可。应用字典树后，分析关键字与运算符过程中减少了无谓字符串的比较，避免了字符串匹配的回退，使得关键字匹配的效率大大提高。

不过字典树属于“以空间换时间”的数据结构，由于一般来说单个文件中的代码行数不会上千上万，因此本处使用字典树是合理的。

5.3.2 缓存思想引入

由于本作业中存在模块间的互相调用，如语法分析会调用词法分析模块，前端呈现将调用两大分析模块。此外，在语法分析内部，光产生式就可能有超过一千条，状态集更是超过一千五百个，整个分析过程十分缓慢。

考虑到对于同样的语法规则，构建得到的 Action Goto 表应该是相同的。而对于相对复杂的语法规则，构建 Action Goto 表涉及的查询次数非常多。于是在构建 First 集的过程中，我们选择在单次构建后，将该表存储到本地文件。后续启动时，首先尝试加载缓存，仅当加载失败才重新构建，以此实现瞬间启动。

5.3.3 文法产生式的空串处理

在预先设计时我们并未考虑到空串的情况，但实际测试的时候发现 ε 其实是一个比较头疼的问题，如果输入的产生式里面包含空产生式的话，对于求 FIRST 集和求项目集来说都是一个头疼的问题.

我们采用的处理方法是:求 FIRST 集时 ε 显式表示，即求得的 FIRST 集可能包含 ε，而在求项目集族的时候 ε 隐式表示，即空产生式从一开始就是一个规约项目，应该直接表示成“S-> .”而不是“S->.ε”，因为最后根据算法生成项目之间的转移关系图是一个 DFA，不可能有空转移的存在。最后经过实际测试，这种处理方法也是完全可行的。

5.3.4 Python 语法等的问题

首先是 python 中赋值引用与 C++存在区别，列表在导致生成 First 集时出现了残缺，后续进行规约过程中语法栈符号栈的实现也出了不小的问题。其次是应用全局变量后未及时更新。没有连接前端测试时，由于只测试一次，处理结果是正确的，但是接入前端后系统相当于变成了不断等待输入、处理后继续等待下一阶段输入的循环体。但由于没有每次调用模块时都对全局变量初始化，导致每次启动前端只有第一次分析结果正常，后续阶段都会因为直接在第一次分析变量的基础上添加新值而不断报错，找了很久才意识到是全局变量的问题。

六 实验总结

6.1 成果概述

本项目中，我们使用 C 语言完成词法分析和语法分析功能，它是整个编译器系统的最基础部分之一。在课程的理论学习基础之上，我们积极进行动手实践，实现了一个具有交互功能 LR(1)语法分析器：

1.词法分析器模块 我们将词法分析并生成单词流、求给定文法中终结符、非终结符。

2.语法分析器模块 我们调用词法分析器的分析结果，计算产生式的 First 集、计算 LR(1)项目簇、构建对应 ACTION 表和 GOTO 表等诸多课程理论知识进行代码实现，完成了类 C 语言的词法分析和语法分析过程。

3.模块整合连接 我们小组将以上两大模块的代码进行了整合和连接，确保它们能够协同工作，也方便前端封装。我们设计了清晰的接口和数据结构，使得模块之间的通信和数据传递变得简单而高效。通过仔细的代码审查和测试，我们确保了整个系统的稳定性和正确性。

4.前端展示模块 为了提供一个良好的用户体验，我们小组也设计了前端展示部分。综合考虑 vue 的部署较复杂、HTML 界面不够美观、QT 代码逻辑较复杂等因素后，我们决定使用 tkinter 库提供了一个直观且易于使用的界面，使得用户能够方便地输入代码和查看分析结果。我们还考虑了用户的需求，提供了一些额外的功能，如代码高亮显示、错误提示等，以帮助用户更好地理解和分析代码。

本实验完整的完成了语法分析器，能够正确地根据给定类 C 语言的文法对 类 C 语言程序进行词法和语法分析，并能正确输出 LR（1）分析表、规约过程和语法树。程序设计与编写结束后撰写了完整的实验报告，对本次实验的过程以及所设计的程序进行了清晰全面的说明，完成了课程作业的要求。

6.2 课程认识

《编译原理》作为计算机系的一大重要课程，在整个教学体系内扮演不可替代的作用。在本门课以前我们似乎已经习惯于使用各种编译器编译自己的 C++/Java/ Python 代码，将编译器的这份工作认为是理所应当，也从来没有想过自己也有能力编写一个这样的类 C 语言分析方式。因此深深感受到，作为计算机系的学生，学习这门课是研究计算机工作原理时不可少的一部分。

在实践中，我们了解到词法分析和语法分析是编译原理中两个重要的阶段。词法分析的任务是将源程序中的字符序列转换为词法单元序列，而语法分析的任务是将词法单元序列转换为抽象语法树。

词法分析是编译过程的第一阶段，它将源程序中的字符序列转换为词法单元序列。词法单元是具有语义意义的最小单位，例如标识符、数字、运算符等。词法分析器使用正则表达式或有限状态自动机来识别词法单元。语法分析是编译过程的第二阶段，它将词法单元序列转换为抽象语法树。抽象语法树是源程序的语法结构的树形表示。语法分析器使用上下文无关文法来分析源程序。

总的来说，这次大作业既是对课堂所学到知识方法的一次学习与巩固；又是一次宝贵的设计实践活动，我们将理论应用到实践，巩固课堂所学的知识，收获颇丰。

6.3 将来的拓展工作

基于本次词法分析语法分析综合大作业，我们小组也考虑了未来的拓展：

1.新的语法特性支持：编译器可以拓展以支持新的语法特性，以适应不断演变的编程语言和编程范式。例如，可以添加对新的关键字、表达式、语句或语法结构的支持。这样可以提高编译器的适用性和灵活性，使其能够处理更多类型的源代码。

2.优化和性能改进：编译器可以进行更多的优化和性能改进，以提高生成的目标代码的效率和执行速度。例如，可以引入更高级的优化技术，如基于数据流分析的优化、循环优化、内联函数优化等。这些改进可以使生成的代码更紧凑、更快速，并减少资源消耗。

3.编译全过程的剩余步骤：如果要继续将该程序扩展实现一个编译器，还需要加入语义分析、中间代码生成、目标代码优化等部分，这些会在后续借助课程大作业进行优化添加。整个界面与系统预期在寒假继续完善并开源，作为我们的代表项目。

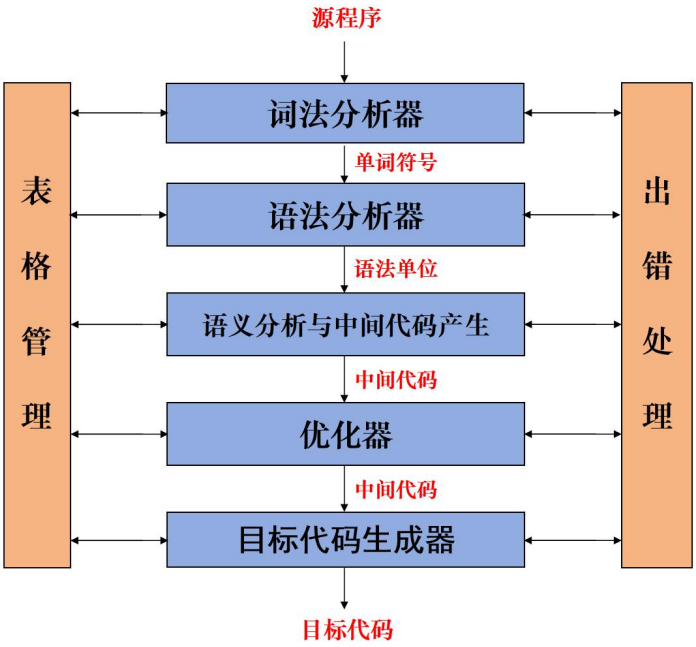


图 27　编译程序的结构示意图

6.3 项目收获

在本项目中，我们不仅需要设计好读取用户输入的词法分析器，还需要考虑后续语法分析器的调用，这段时间内，我们过的充实而紧张，在这份大作业中我们也收获了许多。

从项目结构构思角度来说，实验之前我们进行了充分的讨论和研究，以确定整个项目的结构和框架。我们首先了解了词法分析和语法分析的基本概念和原理，然后根据实验要求和我们的理解，共同设计了项目的整体结构，在明确了各个模块的功能和相互之间的依赖关系之后，项目推进更有目的性。

从代码实现与调试过程来说，我们也遵循了良好的编码规范，并注重代码的可读性和可维护性，这样更方便成员之间的协作与整合。我们也进行了详尽的测试，确保代码在许多预设的错误情景下都能正确地运行。同时，我们也进行了代码的交流和讨论，互相帮助和解决遇到的问题。

从前端交互构思角度来说，在实验中，我们实现了一个具有前端交互功能的词法分析器和语法分析器，不过还需要通过一个前端将代码封装为按钮和图形界面暴露给用户。我们首先一起讨论并具体分析了题中要求，确认了按钮和界面的逻辑，为之后的开发确定方向；然后在分头调研后，最终决定选择 tkinter 库进行呈现，实现了一个用户友好的界面。于是前端部分也在有条不紊的节奏中展开。

通过这个实验，我们深入理解了词法分析和语法分析的原理和实践。我们学会了如何设计和实现一个完整的词法分析语法分析模块，并在团队合作中提高了沟通和协作能力。这个实验为我们今后继续开发中间代码生成或者整个简易的编译器的工作打下了坚实的基础。