装

订

线

同濟大學

TONGJI UNIVERSITY

《编译原理》

实验报告

实验名称	语法分析器大作业
姓名学号	王钧涛 2050254 张弛 2053394 陈奕澄 2053186
学院（系）	电子与信息工程学院
专　　业	计算机科学与技术
任课教师	卫志华
日　　期	2022 年 11 月 14 日

目 录

1 需求分析

1.1 输入约定

（1） c 语言程序

需要用户输入 c 语言源程序文件，该文件固定命名为 input.c，其中存放的源 c 语言程序作为输入端提供给词法分析器。

（2）文法规则

文法规则根据 ppt 中的约定，将类 c 语言的文法规则作为常量放在 base.py 中的 productions 中，该结构是一个列表，包含所有的产生式，每条产生式以字典的方式定义，字典包含 left 和 right 两个 key。

样例形态：

```
1. productions = [
2.     {'left': '<开始>', 'right': ['<程序>']},
3.     {'left': '<程序>', 'right': ['<类型>','$identifier','$(', '$)', '<语句块>']},
```

下面是一些语法约定和处理上的细则：

对于空产生式，right 的 value 是一个空列表，否则 value 列表中的每一项均代表该产生式中的一项。

对于表达式 A -> B | C，需要进行人工处理，产生两条产生式，即 A -> B 和 A -> C。

对于表达式 A -> B [C]，需要进行人工处理，将其拆分成两条产生式 A -> B 和 A -> B C。

对于表达式 A -> B {C}，需要进行人工处理，将其拆分成三条产生式 A -> B [D]、D -> C D 和 D -> ε 。

1.2 输出形式

输出形式如下：

（1）词法分析器：将结果保存在 base.py 中的 w_dict 里，w_dict 是一个列表，

每个元素是一个元表，第一项 w_type 是该 word 的类型，包括数字类型（$digit_int），单词类型（$identifier 或保留字），第二项是单词 word，第三项是位置信息。

样例如下：

```
1. int main() {
2.     int a;
3.     int b; // hello
4.     return 0;
5. }
```

该类 c 语言程序所产生的 w_dict 如下：

```
1. ['$int', 'int', (1, 3)]
2. ['$identifier', 'main', (1, 8)]
3. ['$(', '(', (1, 9)]
4. ['$)', ')', (1, 10)]
5. ['${', '{', (1, 12)]
6. ['$int', 'int', (2, 7)]
7. ['$identifier', 'a', (2, 9)]
8. ['$;', ';', (2, 10)]
9. ['$int', 'int', (3, 7)]
10.['$identifier', 'b', (3, 9)]
11.['$;', ';', (3, 10)]
12.['$return', 'return', (4, 10)]
13.['$digit_int', '0', (4, 12)]
14.['$;', ';', (4, 13)]
15.['$}', '}', (5, 1)]
```

其中，w_type 有可以看出$identifer 有 main，a，b；保留字；以及数值类型$digit_int。word 为单个词。以及位置信息。

（2）语法分析器：从 base.py 中导入 w_dict，将生成的集合闭包存入 base 中的 project_set 中，action_go 表写入 base 中的 action_go。生成的 GrammarTreeNode 写入 grammar_tree 中。

还是以（1）中的程序为例。

输出 project_set（部分），其中每一个元素是一个列表，列表中每一项为字典，包括项目集的产生式，产生式点的位置，以及需要接受的终结符。

```
1. [{'left': '<开始>', 'right': ['<程
   序>'], 'dot': 0, 'accept': '#'}, {'left': '<程序>', 'right': ['<类
   型>', '$identifier', '$(', '$)', '<语句
   块>'], 'dot': 0, 'accept': '#'}, {'left': '<类
```

```
型>', 'right': ['$int'], 'dot': 0, 'accept': '$identifier'}, {'left':
'<类型>', 'right': ['$void'], 'dot': 0, 'accept': '$identifier'}]
```

action_goto 表：状态 + 遇到的符号组成二元组作为字典的 key，将转移的信息作为 value：[’s’,1]表示 ACTION 表中 s1,即移进，入栈，将状态 1 移进状态栈，将该符号移进文法符号栈；[‘r’,2]表示 ACTION 表中符合产生式 2，将栈顶符号规约为产生式左部；[‘g’,3]则表示 GOTO 表中转移到状态 3。

```
{(0, '$int'): ['s', 1], (0, '$void'): ['s', 2], (0, '<程
序>'): ['g', 3], (0, '<类
型>'): ['g', 4], (1, '$identifier'): ['r', 2], (2, '$identifier'): ['r',
3], (3, '#'): ['acc']
```

每轮输出的信息（部分）：利用 action_go 表，将当前符号栈，当前状态栈，转移方程打印输出。

```
#######
当前轮次:1
当前符号栈:['#'],当前状态栈:[0]
当前读入字符:$int,转移方程为:(0, '$int')

#######
当前轮次:36
当前符号栈:['#', '<类型>', '$identifier', '$(', '$)', '${', '<内部声
明>', '$return', '<加法表达式>'],当前状态
栈:[0, 4, 5, 6, 7, 8, 11, 15, 31]
当前读入字符:$;,转移方程为:(31, '$;')

#######
当前轮次:56
当前符号栈:['#', '<程序>'],当前状态栈:[0, 3]
当前读入字符:#,转移方程为:(3, '#')

```

语法树的节点（部分）：

```
[{'sym': '$int', 'son': [], 'cont': 'int'}, {'sym': '<类
型>', 'son': [0], 'cont': ''}, {'sym': '$identifier', 'son': [], 'cont':
'main'},
```

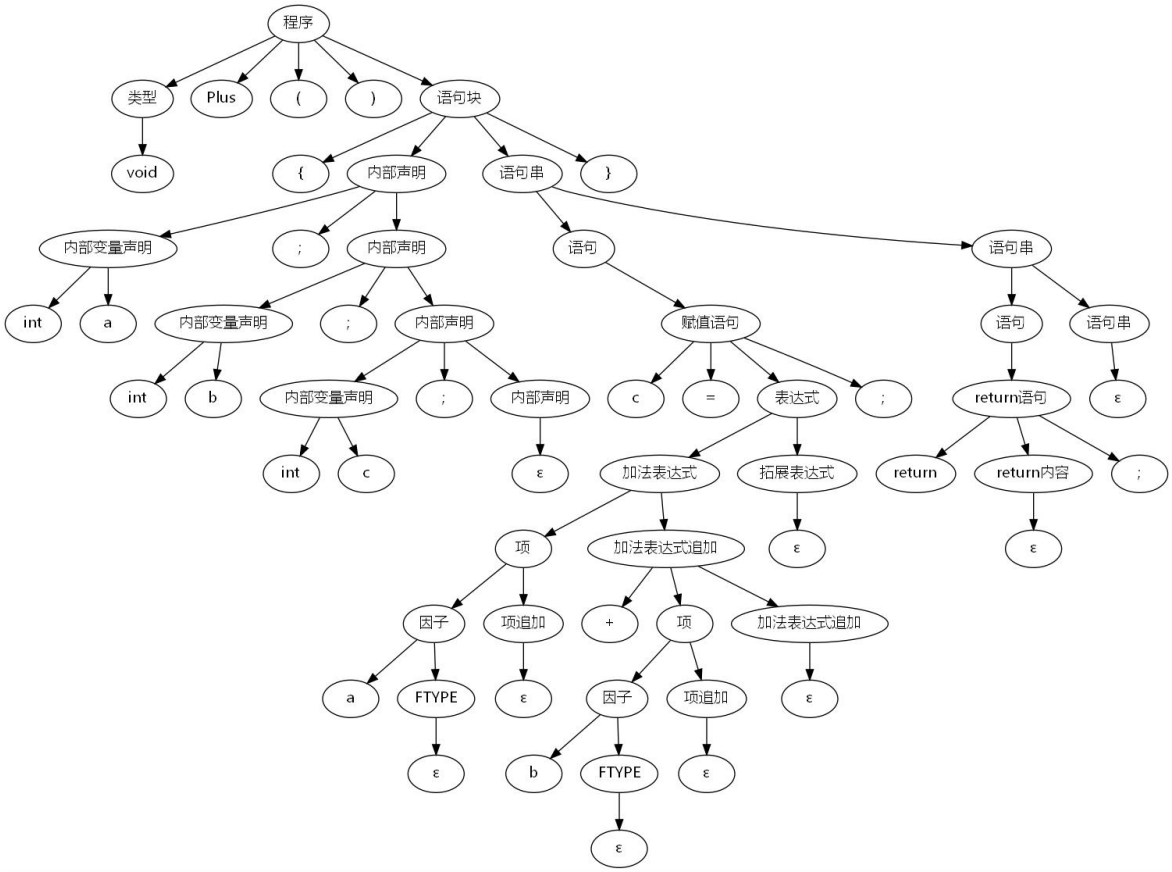
（3）语法分析树：利用 grammar_tree 画出语法分析树，生成 grammer_tree.png。

节点信息：

```
digraph {
```

```
    node0 [label=int fontname="Microsoft YaHei"]
    node1 [label=<类型> fontname="Microsoft YaHei"]
    node1 -> node0
    node2 [label=main fontname="Microsoft YaHei"]
    node3 [label="(" fontname="Microsoft YaHei"]
    node4 [label=")" fontname="Microsoft YaHei"]
    node5 [label="{" fontname="Microsoft YaHei"]
    node6 [label=int fontname="Microsoft YaHei"]
    node7 [label=a fontname="Microsoft YaHei"]
    node8 [label=<内部变量声明> fontname="Microsoft YaHei"]
    node8 -> node7
    node8 -> node6
```

最终产生的语法分析树图片如下：



本次实验选用了 Python 作为实现语言，因此实现了包含中文的语法产生式，相应生成的语法分析树也更加直观易于理解。

1.3 程序功能

1. 项目整体功能

本程序实现对类 c 语言程序的语法分析。对于源程序（input.c）的输入，构造 Action 和 Goto 表。经过词法分析和语法分析，输出词法分析和语法分析结果（[Info]Lexical analysis success! [Info]Garmmar analysis success!），输出分析的过程（即每一轮次的符号栈，状态栈，以及转移的方程），绘制语法分析树。

输入	内容
input.c	需要分析的类 c 语言源程序
输出	内容
词法分析结果	[Info]Lexical analysis success!
语法分析过程	每一轮次的符号栈，状态栈，以及转移的方程
语法分析结果	[Info]Garmmar analysis success!
程序运行结果	[Info]Compile success!
GrammerTree.png	语法分析树

2. 词法分析器.py

词法分析器读入源程序，从源程序中找到词的 w_type，word，和位置信息，输出到 base.py 中的 w_dict。

输入	内容
input.c	类 c 语言源文件
输出	内容
w_dict	分析结果

3. 语法分析器.py

生成 action goto 表以及项目集合，读入句子，输出分析过程和语法树。

输入	内容

w_dict	词法分析器结果
输出	内容
	分析过程
grammer_tree	语法树

1.4 测试数据

测试数据共分为正确数据和错误数据两种，针对错误数据，其文件名中包含了其错误原因。由于错误数据较多，此处不再全部展示，仅将问题区域进行选择性的展示以压缩篇幅。

正确数据：

```
int binary_search(){
    int cnt;
    int ans;
    int l;
    int r;
    cnt = 0;
    ans = 1;
    while(l<r) {
        int c;
        int t;
        if(c>12){
            l=print(r);
        }
        l=l+1;
        cnt=cnt+1;
        ans=ans*cnt;
    }
    return ans;
}
```

错误数据：

（1）lack_of_).c

```
int binary_search(){
    int cnt;
    int ans;
    int l;
    int r;
    cnt = 0;
    ans = 1;
    while(l<r {
        int c;
```

```
        int t;
        if(c>12){
            l=print(r);
        }
        l=l+1;
        cnt=cnt+1;
        ans=ans*cnt;
    }
    return ans;
}
```

（2）lack_of_elseif.c

```
int main(){
    int d;
    int n;
    int m;
    int l;
    int r;
    int mid;
    int ans;
    d = 5;
    n = 8;
    m = 2;
    l = 1;
    r = d;
    while (l < r){
        mid = (l+r) / 2;
        else if (mid < ans){
            ans = mid;
            l = mid + 1;
        }
        else if
            r = mid - 1;
        else
            l = mid;
    }
    return 0;
}
```

（3）lack_of_id.c

```
int binary_search(){
    it cnt;
    int ans;
    int l;
    nt r;
    cnt = 0;
    ans = 1;
    while(l<r) {
        int c;
        in t;
        if(c>12){
            l=print(r);
```

```
        }
        l=l+1;
        cnt=cnt+1;
        ans=ans*cnt;
    }
    return ans;
}
```

（4）lack_of_if.c

```
int main(){
    int d;
    int n;
    int m;
    int l;
    int r;
    int mid;
    int ans;
    d = 5;
    n = 8;
    m = 2;
    l = 1;
    r = d;
    while (l < r){
        mid = (l+r) / 2;
        if (mid < ans){
            ans = mid;
            l = mid + 1;
        }
        else
            r = mid - 1;
        else
            l = mid;
    }
    return 0;
}
```

（5）lack_of_leftnum.c

```
=l+1;
```

（6）lack_of_procon.c

```
int binary_search()
```

（7）lack_of_proname.c

```
int (){
```

（8）lack_of_protype.c

```
1. binary_search(){
```

（9）lack_of_rightnum.c

```
1. l=;
2. cnt=;
```

（10）lack_of_sen.c

```
1. return
```

（11）lack_of_{.c

```
1. int binary_search()
```

装

订

线

2 概要设计

2.1 任务分解

我们小组共计三人，其中一人负责词法分析器的构建，其余两人合力完成语法分析器的实现和语法分析树的绘制。

具体任务的分解如下：

（1） 词法分析器：lexer.py

（2） 语法分析器：gparser.py

a) 项目集合的求解等

b) 总控程序的实现

（3） 语法分析树的绘制：syntaxTree.py

2.2 数据类型定义

1. 词法分析器

（1）保留字表，关键词：列表，每个元素为字符串

```
reserve_word = [
    ",", "!=", "==", "<",  "<=", "=",  ">", ">=", "=",
    "*",  "+", "-", "/",  ";", "(",  ")", "{",  "}", ".", "&&",
    "else", "if", "int", "float", "return", "void", "while"
]
```

（2）词法分析表：列表，每个元素为自定义的结构体，即[w_type, word, (l_cnt, ptr)]

```
['$int', 'int', (1, 3)]
```

2. 语法分析器

（1）产生式表：列表，每个元素为自定义字典，包含 left 和 right 两个 key，分别代表产生式的左边和右边，左边的 value 是字符串，右边的 value 是字符串列表，可以包含 0 个（即空串）或多个字符串。

```
productions = [
    {'left': '<开始>', 'right': ['<程序>']},
```

（2）项目集（project_set）：列表，每个元素为自定义结构，即由多个字典组成的列表，每个字典用 left、right、dot、accept 这 4 个 key 来表示项目集合中的每一项。

```
2. [{'left': '<开始>', 'right': ['<程
   序>'], 'dot': 0, 'accept': '#'}, {'left': '<程序>', 'right': ['<类
   型>', '$identifier', '$(', '$)', '<语句
   块>'], 'dot': 0, 'accept': '#'}, {'left': '<类
   型>', 'right': ['$int'], 'dot': 0, 'accept': '$identifier'}, {'left': '<
   类型>', 'right': ['$void'], 'dot': 0, 'accept': '$identifier'}]
```

（3）action_goto 表：字典，状态 + 遇到的符号组成二元组作为字典的 key，将转移的信息作为 value。

```
2. {(0, '$int'): ['s', 1], (0, '$void'): ['s', 2], (0, '<程
   序>'): ['g', 3], (0, '<类
   型>'): ['g', 4], (1, '$identifier'): ['r', 2], (2, '$identifier'): ['r',
   3], (3, '#'): ['acc']
```

（4）是否为终结符的判断表（symbol_list）：字典，将所有字符所谓 key，终结符的 value 为 1，非终结符的 value 为 0。

```
{'$,': 1, '$!=': 1, '$==': 1, '$<': 1, '$<=': 1, '$=': 1, '$>': 1, '$>=': 1,
'$*': 1,
```

（5）first 集（first）：字典，产生所有非终结符的 first 集合（部分），所有非终结符作为 key，每个 value 是一个字符串列表，存储该非终结符的 first 集合。

```
1. {'#': ['#'], '$,': ['$,']
```

3. 语法树绘制

（1）语法树（grammar_tree）：列表，每个元素为自定义的字典，包含 sym、son 和 cont 三个 key。其中，sym 和 cont 的 value 是字符串，son 的 value 是整数列表。

```
1. [{'sym': '$int', 'son': [], 'cont': 'int'}, {'sym': '<类
   型>', 'son': [0], 'cont': ''}, {'sym': '$identifier', 'son': [], 'cont':
   'main'},
```

2.3 主程序流程

1. main 函数

（1）定义输入文件

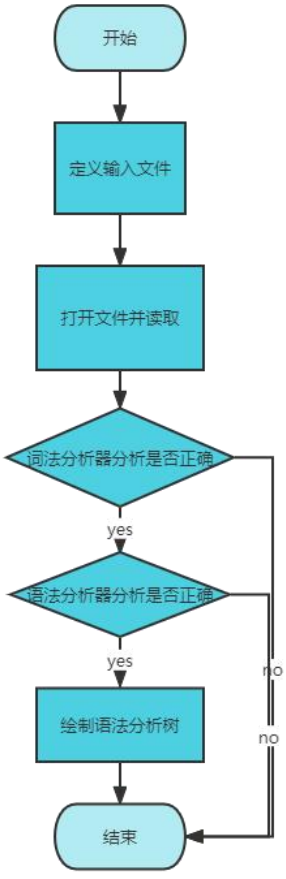
（2）打开文件，读取进入 str 字符串

（3）调用词法分析器，将分析结果存储在 w_dict 中（输出分析结果）

（4）调用语法分析器，输出分析过程和结果，将语法树节点存储在 gramer_tree 中

（5）调用绘制语法分析树

流程图如下：

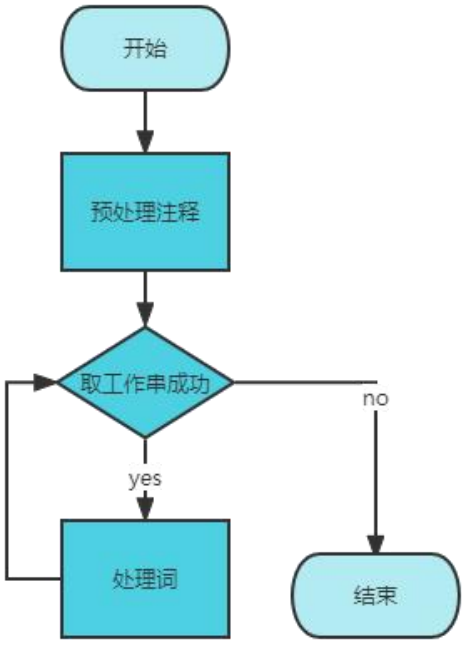


2. 词法分析器

（1）预处理注释

（2）循环获取工作串

（3）进行词语分割



3. 语法分析器

（1）求项目集合，产生 action/goto 表。

（2）从[S'→S,#]开始，输入的待分析句子，根据 A/G 表，对句子进行移进/归约。

（3）每一步输出当前符号栈与状态栈。

（4）根据每一步结果生成语法树信息。

2.4 模块间调用关系

main 程序主要调用词法分析器和语法分析器以及绘制语法分析树的模块。

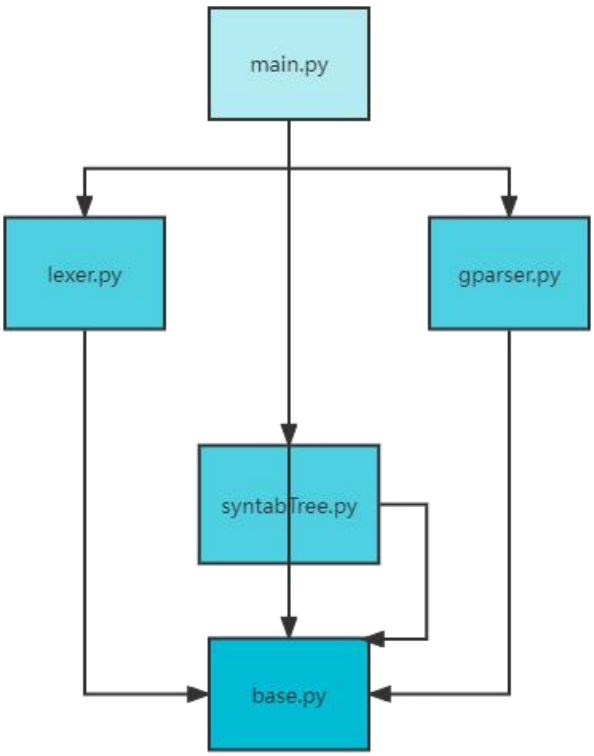
词法分析器调用了预处理注释，获取工作串以及词法分析的模块。

语法分析器调用了计算 first 集，计算闭包，计算项目集合，计算 action_go 表模块。

绘制语法分析树调用了 graphviz 中的 Digraph 模块。

其中，上述模块都会使用或改变 base.py 中的参数。

关系图如下：



装

订

线

3 详细设计

3.1 主要函数分析与设计

3.1.1 主函数 main

主函数负责将其他各个模块的主控函数耦合在一起进行工作：

```
if __name__ == '__main__':
    src = 'input.c'
    with open(src, encoding='utf-8') as f:
        str = f.read()
        if Lexer(str) and GParser():
            print('[Info]Compile success!')
            drawSyntaxTree()
```

主函数较为简单，首先将指定的待分析的源程序文件 src 并通过 read 函数读入到字符串中，然后将其送入到词法分析器中产生词法分析器输出，并将该输出作为语法分析器的输入(在 base 文件中传递)，最后判断这两者是否顺利完成工作，如果顺利完成代表输入文件通过了语法分析，这时候就调用 drawSyntaxTree 函数绘制原始输入文件的语法分析树。

3.1.2 词法分析器 Lexer

词法分析器负责从源文件生成机内输入序列，其主控函数如下：

```
def Lexer(i_str): #词法分析器入口函数，输出结果会放到 base 的 w_dict 中
    i_str = preProcessComment(i_str + '\r\n')#预处理，去除注释
    print('去除注解的源程序列表:', i_str)
    while getWorkString(i_str): #不断获取工作串
        try:
            dealWorkString() #进行分词解析
        except Exception as err: #捕获异常并进行异常输出
            print(str(err))
            return False
    print('单词机内表示序列:', w_dict) #词法分析器输出
    print("[Info]Lexical analysis success!")
    return True
```

词法分析器结构输入的源程序串，并且将其输入到 preProcessComment 函数中去除注释，然后不断调用 getWorkString 函数获取工作串，并且对每一个可能的工作串调用 dealWorkString 函数尝试进行分词处理。

一旦遇到处理失败的情况，dealWorkString 函数会 raise 出异常，此时主控程序会捕获异常并进行异常详细信息的输出。

词法分析器主控函数成功返回 True，失败返回 False

其中，注释处理函数如下：

```
1. def preProcessComment(str): #预处理注释
2.     str = re.sub('//[^\n]*\n', '\n', str) #处理单行注释
3.     str = re.sub('/\\*(.|\r|\n)*?\\*/[\x20]*', '\n', str) #处理多行注
   释
4.     return str #返回去除注释后的字符串
```

此处的设计考虑到 C 语言的两种注释类型：

1）// + 注释，这种表达方式一定是单行的，并且注释内容从 // 开始算起一直到行尾，因此可以用第二行的正则表达式来进行删除，其含义是从//开始中国经过不是回车的任意多个字符，最后以回车结尾的子字符串。值得注意的是，如果注释内容出现在最后一段，如：

```
1. int cnt(int b){
2.     int a = b * 2;
3.     return a;
4. } //some comment
```

此时产生式失效，应为文件末尾不存在回车，解决方法是在输入串的末尾强制添加一个回车，该回车不会污染语法分析器的最终结果。

2）/* + 注释 + */ 该种注释方式的特点是注释之间可以出现多行的情况，只要左注释符号和右注释符号闭合即可，因此可以用 /*[.*]*/正则表达式来描绘。但是，在实践中发现这样子进行匹配会默认匹配到最大的一种情况，例如：

```
1. int mian(){
2.     int a;
3.     int b;
4.     if(a){
5.         return a;
6.         /* 这里是注释 a */
7.     }else if(b){
8.         return b;
9.         /* 这里是注释 b */
10.     }
```

显然，这里的注释 a 和注释 b 是两条语句，但是如果采用初始正则表达式会导致

注释 a 开始符号匹配上注释 b 的结束符号，从而造成了无关代码的删除，因此在实际设计过程中需要从中间禁止’*/’符号的出现。

同时考虑注释嵌套的情况，由于这里采取的是删除操作，多个嵌套注释无论先后操作顺序删除，最外层的嵌套最终都一定会被删除，因此此处程序不用考虑删除的先后顺序，这里按先处理单行注释，再处理多行注释的方法进行操作。

获取工作串函数如下：

```
def getWorkString(str): #整行读取输入并返回分割后的字符串
    global w_ptr, l_cnt, w_str
    if w_ptr == len(str):#w_ptr: 指向当前工作位置的指针
        return False
    l_cnt += 1 #行计数器
    ret = re.search('.*\n', str[w_ptr:]) #找第一个出现的回车字符串，从此处切分
    w_ptr, w_str = ret.end() + w_ptr, ret.group() #设置新的工作串和工作指针
    return True
```

该函数的作用是切分出工作串，从而让的词法分析可以在一句句型的基础上进行分析，并且给出行计数器标识，使得之后的词法分割出来的单词能够包含其行信息以便于之后语法分析中遇到错误标识符时能够提供完整的错误处理位置定位。

其工作规则是：

若当前串的位置已经处于字符串的末尾，则返回 False

否则：

将行计数器增加一

从字符串的下一个\n 处进行切分

将工作指针 w_ptr 置为分割串的末尾指针，将工作串 w_str 置为当前分割出来的字符串

返回 True

在完成工作串切割后，该工作串会被输入到 dealWorkString 函数中进行词法分析，其函数代码如下：

```
def dealWorkString(): #词法分析
    global l_cnt, w_ptr, w_str, word, w_type
    ptr = 0 #句子的工作指针
    lw = len(w_str) #工作串的长度
    while ptr < lw: #当仍然有剩余的工作串
        ptr = ptr + re.search('[\x20\r\n]*', w_str[ptr:]).end() #过滤无用空格和回车
        if ptr == lw:
            return True
        if w_str[ptr].isdigit(): #处理数字
            ret = re.search('\\d+(\\.\\d+)?(e(\\+|-)?\\d+)?', w_str[ptr:])
            word = ret.group()
            if w_str[ptr+ret.end()].isalpha() or w_str[ptr+ret.end()]=='_':#数字匹配失败 抛出异常
                raise Exception('[Error]#101 in line {}, position {} : illegal num {}.'.format(l_cnt, ptr, word + w_str[ptr+ret.end()]))
            if '.' not in word and 'e' not in word: #根据其值区分整形数和浮点数
                w_type = '$digit_int'
            else:
                w_type = '$digit_int'#由于还没有实现浮点数，先当作整形数
        elif w_str[ptr].isalpha() or w_str[ptr] == '_': #处理单词
            ret = re.search('\\w*', w_str[ptr:])
            word = ret.group()
            w_type = '$identifier' if JudgeReverseWord(word) == -1 else JudgeReverseWord(word) #判断其是否为保留字
        else: #处理符号
            eptr = ptr + 1 #从下一个符号开始判断
            has_re = False
            while (not has_re and isSymbol(w_str[eptr - 1])) or JudgeReverseWord(w_str[ptr:eptr]) != -1: #匹配最长的是保留字的词
                if JudgeReverseWord(w_str[ptr:eptr]) != -1:
                    has_re = True
                eptr += 1
            word = w_str[ptr:eptr-1]
            if not has_re: #初始匹配失败 当前符号本身就不是关键字 抛出异常
                raise Exception('[Error]#102 in line {}, position {} : illegal word {}.'.format(l_cnt, ptr, word))
            w_type = JudgeReverseWord(word)
        ptr += len(word)
        w_dict.append([w_type, word, (l_cnt, ptr)]) #将完成分割的词语放入输出序列中
```

该函数的工作原理如下：

首先设置 ptr 为当前工作位置指针，lw 为工作串总长度

循环判断 ptr 是否小于 lw，即还未分析结束：

则先去掉 ptr 之后的无用空格和回车

如果去掉后无剩余字符，则返回 True

否则判断第一个有用元素：

如果是字符或者下划线，进行【标识符分析】

如果是数字，进行【数字分析】

如果是符号，则进行【保留符号分析】

将分析结果[标识号，实际词语，原始位置]输入机内结果序列中

设置 ptr 指向当前词语的下一个位置

装

由于该函数较长，将其中部分分开进行叙述：

【标识符分析】：

此处已经知道起始的开头符号是字符，因此该标识符只需要符合标识符规则“以字符或者下划线开始，只包含字符、下划线和数字的字符串”即可。

因此，构造正则表达式’w*’就可以表达出标识符的规则，该部分的工作原理如下：

订

使用正则表达式找出匹配的最大字符串 word

调用 JudgeReverseWord()函数判断其是否为保留字

如果是，则设置 w_type 为其对应的保留字标识

如果不是，设置 w_type 为$identitier

线

值得提及的是，标识符分析不会产生错误，因为其提取的过程中同时就满足了标识符规则。

【数字分析】：

数字分析是上述三个分析中比较困难的一项，这里我们实现了额外要求 2，即将整常数扩充为实常数，这就使得数字不再局限于只是由数字字符组成的字符串了，为此，我们对各类实常数进行了考虑：

```
1334
764843241
1314.0
5485.44651
125e4
353.26e7
135e+4
643.777e-2
```

一个数字可以是由整数组成，也可以是整数部分.小数部分，同时在末尾还可以加上 e 和指数部分，其中后两者为浮点数，前者为整形数。

从而，设计出正则表达式：`'\\d+(\\.\\d+)?(e(\\+|-)?\\d+)?'`

函数的工作原理如下：

应用正则表达式查找相应的符合要求的字符串，并赋值给 res

从 res 中提取字符串放入 word 中

若 word 之后跟的字符或者是下划线：

给出错误数字异常

否则通过正则表达式 d+判断是否是整形数

若是，设置标识符为整形数

否则，设置标识符为浮点数

这里没有采用先将最大字符串读入，再判断是否是数字的方法，主要是因为 e 之后支持读入+和-作为有符号数的开始，这就导致无法区分+和-是否是符号标识符的开始还是有符号数的开始（在起始初也类似，这里不采用先读入-号作为有符号数开始的方法，而是将-作为一元运算符，下放到语法和语义分析中进行处理）

【符号分析】：

符号分析的工作原理如下：

设置 has_re 变量用于统计当前是否已经找到了一个合适的符号串

设置当前工作指针 eptr 指向当前待分析的字符

循环执行：

如果当前串是保留字符，那么将 has_re 置为 True

直到已经出现过保留串并且当前字符不是保留串为止

判断是否找到了保留字串：

是：将其对应的标识号添加到 w_type

否：抛出符号错误异常

符号分析的分析方法与其余两者不太一致，主要是由于一个保留字符号的前缀串不一定是符号串，因此不能盲目的使用拓展算法，需要保存最大的保留字符号。

JudgeReverseWord 函数：

该函数用于判断输入串是否是保留字，并给其赋予标识符：

```
reserve_word = [
    ",", "!=", "==", "<",  "<=", "=",  ">", ">=", "=",
    "*",  "+", "-", "/",  ";", "(",  ")", "{",  "}", ".", "&&",
    "else", "if", "int", "float", "return", "void", "while"
]
def JudgeReverseWord(w):
    return '$'+w if w in reserve_word else -1
```

成功返回加上$的标识符号，如’$int, $return’，失败返回-1

其中，这里实现了词法分析器的额外要求 1，通过向 reserve_word 里添加和删除保留字，可以实现增加或者删除保留字，一个简单的例子就是此处额外添加的用于表示浮点数的‘float’关键字。

3.1.3 语法分析器 GParser

语法分析器以词法分析器所产生的词语机内表示序列作为输入，利用给出的产生式不断对其进行规约操作，完成检测语法错误和构建语法树的工作。

语法分析器的主控程序如下：

```
def GParser():
    initProjectSet() #初始化项目集，构建 action 和 goto 表
    input_st = [['#', 'INPUT_END', w_dict[-1][2]]] + w_dict[::-1] #设置输入机内序列
    state_st, sym_st, id_st, son_st = [0], ['#'], [], [] #设置工作栈
    t_cnt=1 #轮次记录变量
    try:
        while len(input_st) > 0:
            ns = (state_st[-1], input_st[-1][0]) #记录当前状态的元组
            t = action_goto.get(ns) #从 action 和 goto 表中查找该元组
```

```
                print('#######\n 当前轮次:{}\n 当前符号栈:{},当前状态栈 :{}\n 当 前 读 入 字 符 :{}, 转 移 方 程为:{}'.format(t_cnt,sym_st,state_st,input_st[-1][0], ns))
            if t is None: #如果未找到，说明该串存在语法错误
                print(ns)
                raise Exception('[Error]#201 in line {}, position {} : Unexpected word \'{}\' after \'{}\'.'.format(*input_st[-1][2], input_st[-1][1], sym_st.pop()))
             if t[0] == 's' or t[0] == 'g': #移进或者 goto,两者代码相同
                it = input_st.pop()
                sym_st.append(it[0]) #将符号提取出符号栈中
                state_st.append(t[1]) #将当前的状态提取出放入状态栈
                 id_st.append(addGrammarTreeNode(sym_st[-1], son_st, it[1])) #存放语法树节点
                son_st = []
            elif t[0] == 'r': #规约
                 for i in range(len(productions[t[1]]['right'])): #如果不是从空串规约而来，设置子节点
                    sym_st.pop() #将被规约的子节点弹出
                    state_st.pop()
                    son_st.append(id_st.pop()) #并将其放入孩子栈中
                if len(productions[t[1]]['right']) == 0: #如果是空串,额外添加ε
                     son_st.append(addGrammarTreeNode('@', [], 'ε'))

                 input_st.append([productions[t[1]]['left'], '']) #将规约完的节点添加到 input 串中
            else:#规约成功 acc
                print("[Info]Garmmar analysis success!")
                break
            t_cnt += 1
    except Exception as err:#异常处理
        print(str(err))
        return False
    return True
```

其工作原理为：

首先求好有效项目集、action 表和 goto 表

循环直到 input 栈为空：

获取当前转移状态的元组

从 action_goto 表中查找下一步需要执行的状态

若为空，说明该串未找到，发出语法分析异常并给出异常位置

如果是移进或者 goto 操作，采用统一的操作方式：

将 input 栈的首元素弹出并将对应值放入符号栈和状态栈中

用该元素和孩子表作为参数注册语法树节点并将该节点放到 id 栈中

将孩子表设为空

如果是规约操作：

如果不是从空串规约而来：

对每一个孩子，将其从符号栈和状态栈中弹出并加入孩子表里

反之：

额外向孩子表中注册并添加 ε 节点

将规约完的节点添加到 input 栈中

如果都不是，那就是遇到了 acc 规约成功符号：

输出规约成功信息并退出

进行异常处理，捕获其中产生的异常并输出

该函数最后会将所有的规约信息都输出到 base 模块下的 grammar_tree 中，以供之后绘制语法树调用。

值得一提的是，该函数在设计的过程中没有区分 action 表和 goto 表，这是因为在实际编写代码的过程中我发现 action 表的移进操作和 goto 表的移进操作的内容其实是相同的，如果在规约状态中不将非终结符符号直接放入已经分析的状态栈，而是放入 input 栈中，那么在遇到移进操作和转移操作时两者基本没有区别，除了一个是终结符一个是非终结符。

该函数调用了 initProjectSet 函数用于求解有效项目集、action 表和 goto 表，该函数的代码如下：

```
1.  def initProjectSet():
2.      start = deepcopy(productions[0])
3.      start['dot'], start['accept'] = 0, '#'#生成初始集合
4.      c = findClosure([start]) #求解初始集合的闭包并加入有效项目集中
5.      project_set.append(c)
6.      top = 0 #递归调用计算
7.      while top < len(project_set): #采用广度优先搜索方式进行搜索
8.          for x in symbol_list: #对每一个正在被搜索的状态集：
9.              next_set = findGoto(project_set[top], x)
```

```
            if len(next_set) > 0 and not next_set in project_set:#将新构造的集合放入项目簇中
                project_set.append(next_set)
            #if len(next_set) > 0: #调试语句
             #    print("#{}=<{},{}>:".format(project_set.index(next_set), top, x), next_set)
            if len(next_set) > 0:
                if not isTerminalSymbol(x): #如果x不是终结符,那么填goto集
                    action_goto[(top,x)] = ['g', project_set.index(next_set)]
                else: #否则填在移进集
                    action_goto[(top, x)] = ['s', project_set.index(next_set)]
        for wt in project_set[top]:
            if wt['dot'] == len(wt['right']):#遍历next集中的规约项目
                ns = {'left':wt['left'], 'right':wt['right']} #将action_goto集的accept位设为规约
                action_goto[(top, wt['accept'])] = ['r', productions.index(ns)] if ns['left'] != '<开始>' else ['acc']
        top += 1 #选取下一个集合进行操作
```

其工作原理为：

从第一个项目开始生成初始集合

将初始集合的闭包加入有效项目集中

进行广度优先搜索，对有效项目集中每一个没有被搜索过的节点：

用符号集里的每一个元素计算其 goto 集，如果该 goto 集不为空：

判断其是否在有效状态集合中，如果不在则加入

计算其在有效状态集中的 id

根据 x 是否为终结符，将其填在 action 集或是 goto 集中

然后对该集合的规约项目，将其展望串放入 action 集中

该函数共调用了两个函数 findGoto 和 findClosure，先介绍 findGoto 函数，其代码如下：

```
def findGoto(proj, sym)->list: #通过输入的项目集和sym找其goto后的closure集
    c = [] #设置初始goto集为空
    for p in proj: #对于输入的每一个串,都尝试能否接受sym符号
```

```
        if p['dot'] != len(p['right']) and p['right'][p['dot']] == sym:
            np = deepcopy(p)
            np['dot'] += 1
            c.append(np) #如果可以接受,将其接受后的新串加入 goto 集中
    return findClosure(c) #计算计算后的 goto 集的闭包
```

该函数的工作原理是：

先设置初始 goto 集为空

对项目簇里面的每一个项目，都尝试其是否能接受输入的 sym 符号：

可以就先计算其接受后的新状态

让后将这个新状态加入到 goto 集中

最后返回该 goto 集的闭包

同样的，findGoto 函数中也需要计算闭包，而实现该功能的 findClosure 函数如下：

```
def findClosure(parent:list)->list: #输入一个项目簇,将其扩充成其 closure 闭包
    top = 0
    closure_set=deepcopy(parent) #先将闭包初始设置为输入的项目簇
    while top < len(closure_set): #不断对闭包里的每一个元素进行遍历
        nowc = closure_set[top]
        pos = nowc['dot'] #记录当前项目的 dot 位置
        if pos < len(nowc['right']) and not isTerminalSymbol(nowc['right'][pos]): #如果是非终结符，进行拓展
            follow_symbol = deepcopy(nowc['right'][pos+1:]) if pos < len(nowc) - 1 else [] #构造后面用于求 first 集的子串
            follow_symbol.append(nowc['accept']) #再将展望串放在最后
            new_accept_set = findFirst(follow_symbol) #计算新的展望串
            for p in productions:
                if p['left'] == nowc['right'][pos]: #求到新拓展项目，添加入 closure 集中
                    for nas in new_accept_set: #遍历展望串各元素，都要添加入 closure 集
                        newp = deepcopy(p)
                        newp['dot'], newp['accept'] = 0, nas
                        if not newp in closure_set:#进行元组去重，避免重复元素添加
                            closure_set.append(newp)
        top += 1
    return closure_set
```

该函数的工作原理为：

将原项目簇进行拷贝以作为搜索的初始集

对每一个闭包里面没有被搜索过的项目：

若当前工作点后是非终结符：

将其之后的内容与其展望串合并，并赋值给 sym

调用 findFirst 计算 sym 的 First 集，其为新项目的展望串

对任意左部是遇到的非终结符的产生式：

分别将其变为新项目，其依次接受 First 集里的各展望串

如果该新项目还不在闭包中，将其添加入新项目

返回生成的项目集

值得注意的是，这样子的程序可能会产生很多的重复集合，这里采用的一种较好的方法是在对一个产生式求一个项目的同时将其加入访问字典，以供下一次查询时可以直接获得其结果。

该函数调用了 findFrist 函数用于求解 first 集，其代码如下：

```
1. def findFirst(sym_list): #找到一个符号串的 first 集
2.     f=[]
3.     for x in sym_list: #依次遍历符号串的每个元素
4.         nf = findSymbolFirst(x)
5.         if '@' in nf:#如果存在空集
6.             nf.remove('@')
7.             f += nf #将空集删除并将其加入 first 集，继续遍历
8.         else:
9.             f += nf #将其加入 first 集并终止遍历
10.             break
11.     return f #返回计算得到的 first 集
```

该函数的工作原理如下：

令初始 first 集为空

依次遍历输入的符号串，对每一个符号 x：

计算他的 first 集

若其 first 集中含有 ε：

删除 ε，并将剩余符号加入 first 集中

否则：

并将计算结果加入 first 集中

终止遍历

返回生成的 first 集

这里需要强调的是，一个正常 first 集的计算方式与此处的代码是不竟相同的，因为一个正常的符号串其 first 集有可能为空，但是调用这条指令是在 findclosure 中，从而输入的符号串的结尾肯定是某一个项目的展望符号，而这个展望符号一定是终结符，从而就导致了 findFirst 函数所找到的 first 集一定是不会包含空元素的，从而这里就没有对 ε 进行特殊处理。

在 findFirst 符号中，还调用了 findSymbolFirst 函数用于寻找每个符号的 first 集，该函数的代码如下：

```
first = {'#':['#']} #所有非终结符的 first 集，空集用'@'表示
def findSymbolFirst(sym):
    global first
    f=[] #初始化 first 集为空
    if sym in first: #如果它已经计算完毕，则直接返回
        f=deepcopy(first[sym])
        return f
    if isTerminalSymbol(sym): #如果他是终结符，直接返回它自己
        f.append(sym)
    else:
        for p in productions:
            if p['left'] == sym:#遍历每条从 sym 开始的产生式
                if len(p['right']) == 0: #如果它能推出空
                    f.append('@') #将空集加入
                elif isTerminalSymbol(p['right'][0]): #或者他是终结符
                    f.append(p['right'][0]) #将该终结符加入
                else:
                    for x in p['right']:
                        nf = findSymbolFirst(x) #递归求解其元素
                        if '@' in nf:#如果存在空集
                            nf.remove('@')#那么删除空集
                            f += nf
                        else:
                            f += nf
                            break
```

```
                    else:
                        f.append('@') #全部都含有空，将空集加入
    first[sym] = f #保存该元素值
    return f
```

其工作原理如下：

初始化 first 集为空

判断其是否已经搜索过，如果是直接返回结果

若其是终结符，直接返回

否则遍历所有产生式，寻找从他推出的产生式 p：

如果 p 能推出空，那么将 ε 加入

或者 p 能退出终结符，将该终结符加入

否则递归调用，依次计算其推出的非终结符串的 first 集 f：

若 f 中含有 ε：

删除 ε，并将剩余符号加入 first 集中

否则：

并将计算结果加入 first 集中

终止遍历

如果递归调用正常结束，将 ε 加入 first 集中

保存该结果

将计算获得的 first 集返回

该函数的设计采用递归调用的思想，大大简化了代码编写的难度，同时通过结果保存的方法节省了代码的计算时间复杂度

3.1.4 语法树绘制函数 drawSyntaxTree

drawSynataxTree 函数用于将抽象的语法分析树绘制成 png 图片，从而直观的展示出来，其代码如下：

```
def drawSyntaxTree():
    dot = Digraph(comment='Grammar Tree') #生成语法图
    cnt = 0 #初始化节点统计变量
```

```
    for i in grammar_tree: #对所有语法树中的节点
        nodeName = 'node'+str(cnt) #设置节点名字
        dot.node(nodeName, i['cont'] if len(i['cont']) else i['sym'], fontname="Microsoft YaHei") #添加节点
        for j in i['son']: #设置孩子节点名
            sonName = 'node'+str(j)
            dot.edge(nodeName, sonName) #画自己到孩子节点的边
         cnt=cnt+1
    dot.render('grammar_tree', view=True, format='png') #生成图片
    os.remove('grammar_tree') #删除中间文件
```

函数的依赖：

首先在函数中引进 python 的 graphviz 库中的 Digraph 类，从 base.py 中引进已经按照列表形式给出的 grammar_tree。

函数的实现：

创建一个 Diagraph 对象 dot，图名设置为 Grammar Tree。

创建节点计数器 cnt，并初始化为 0。

遍历列表形式的 grammar_tree 中的每一个元素：

按照当前节点计数器的值，命名一个新节点。

将新节点加入 dot 中。若节点是终结符，则将节点内容设置为该终结符；否则将节点内容设置为这个节点的类别。

遍历这个节点的所有儿子节点：

命名儿子节点。

在当前节点与儿子节点之间建边，加入 dot。

节点计数器自增。

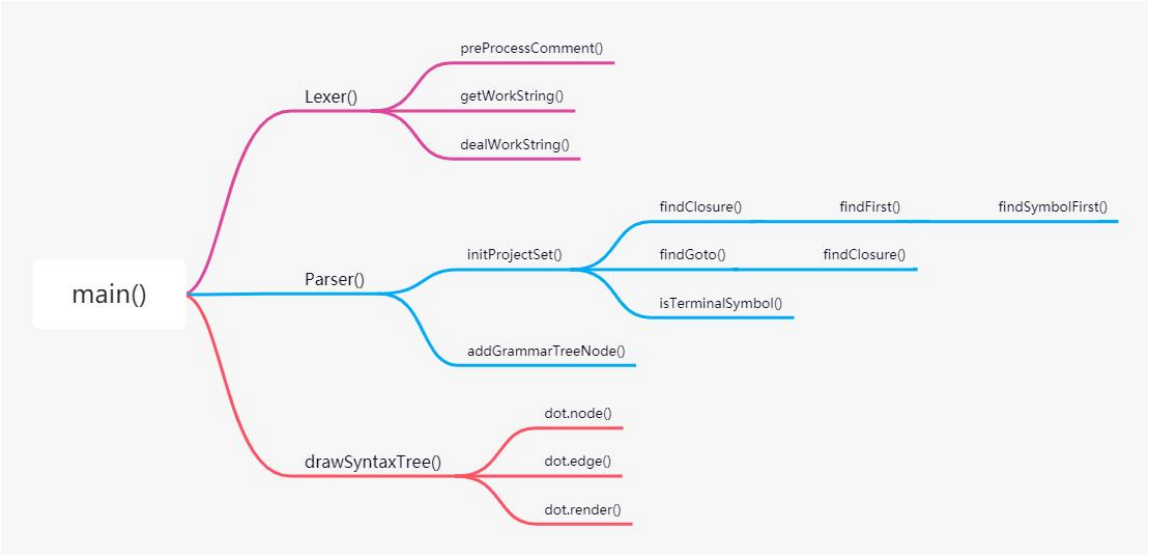
生成 png 图片，展示语法树。

该函数的主要设计考虑是在命名儿子节点的时候，如何确定儿子节点的节点名。事实上，在列表形式的 grammar_tree 中，我们可以通过下标定位每一个元素，也就是说所有的节点实际上天然已经有了一个序号，可以用字符串”node”加上这个下标来组成节点的名字，我们在命名新节点的时候，正是使用了这种方法。而在串联儿子节点的时候，由于在 grammar_tree 中，储存儿子节点使用的记号是儿子节点的下标，因此只需要取出儿子节点这个元素的内容，就可以得到下标，所以可以直接定位到儿

子节点。

3.2 总函数调用图

整体程序的总函数调用图如下：



整个程序分为五个文件，分别为 base.py、lexer.py、gparser.py、syntaxTree.py 和 mian.py。

其中，main.py 是整个程序的入口，存放主函数 main，lexer.py 中存放词法分析器的相关函数，gparser.py 中存放语法分析器的相关函数，syntaxTree.py 存放绘制语法树的相关函数，base.py 作为基本文件，存放保留字表、产生式表和词法语法分析器的工作全局变量，包括词语机内表示序列、action 集、goto 集和语法树 grammar_tree 等并提供基本的操作和判断函数。

4 调试分析

4.1 测试数据与测试结果

本次实验测试了多种不同的语法分析和词法分析错误，均得到了预期的测试效果：

1) #101 词法分析器::非法数字错误：

测试数据 1：

```
1.  int binary_search(){
2.      int cnt;
3.      int ans;
4.      cnt = 1.2e+3e4;
5.      ans = 1;
6.      return ans;
7.  }
```

输出：

```
[Error]#101 in line 4, position 10: illegal num 1.2e+3e.
```

测试数据 2（漏乘号）：

```
1.  int binary_search(){
2.      int b, c;
3.      cnt = 1.2+3.4+5ans+6;
4.      ans = 1;
5.      return ans;
6.  }
```

输出：

```
[Error]#101 in line 4, position 18: illegal num 5a.
```

2) #102 词法分析器::非法符号错误

测试数据 1：

```
1. int binary_search(){
2.     int cnt;
3.     cnt ^= 3;
```

```
    return cnt;
}
```

输出：

```
[Error]#102 in line 3, position 8: illegal word ^=.
```

测试数据 2：

```
int binary_search(){
    int cnt;
    cnt == 3;
    return cnt;
}
```

输出：

```
单词机内表示序列: [['$int', 'int', (1, 3)], ['$identifier', 'binary_search', (1, 17)], ['$(', '(', (1, 18)], ['$)', ')', (1, 19)], ['${', '{', (1, 20
)], ['$int', 'int', (2, 7)], ['$identifier', 'cnt', (2, 11)], ['$;', ';', (2, 12)], ['$identifier', 'cnt', (3, 7)], ['$==', '==', (3, 10)], ['$digit_
int', '3', (3, 12)], ['$;', ';', (3, 13)], ['$return', 'return', (4, 10)], ['$identifier', 'cnt', (4, 14)], ['$;', ';', (4, 15)], ['$}', '}', (5, 1)]
]
[Info]Lexical analysis success!
```

词法分析器正确识别到 == 而非两个 = 号

3) #201 语法分析器::非预期的符号错误

我们共针对目前所采用的语法产生式构造了五类错误数据，进而判断输出结果是否与预期相符。

测试数据 1：if 语句缺乏左括号

```
int binary_search(){
    int cnt;
    int ans;
    int l;
    int r;
    cnt = 0;
    ans = 1;
    while(l<r) {
        int c;
        int t;
        if c>12){
            l=print(r);
        }
        l=l+1;
        cnt=cnt+1;
        ans=ans*cnt;
```

```
    }
    return ans;
}
```

程序输出：

```
#######
当前轮次:138
当前符号栈:['#', '<类型>', '$identifier', '$(', '$)', '${', '<内部声明>', '<语句>', '<语句>', '$while', '$(', '<表达式>', '$)', '${',
当前读入字符:$identifier,转移方程为:(14, '$identifier')
(14, '$identifier')
[Error]#201 in line 11, position 12: Unexpected word 'c' after '$if'.
```

正确识别异常，分析栈无误。

测试数据 2：main 函数左大括号失配

```
int binary_search(){
    int cnt;
    int ans;
    int l;
    int r;
    cnt = 0;
    ans = 1;
    while(l<r) {
        int c;
        int t;
        if(c>12){
            l=print(r);
        }
        l=l+1;
        cnt=cnt+1;
        ans=ans*cnt;
    return ans;
```

程序输出：

```
#######
当前轮次:359
当前符号栈:['#', '<类型>', '$identifier', '$(', '$)', '${', '<内部声明>', '<语句>', '<语句>', '$while', '$(', '<表达式>', '$)', '${', '<内部声明>', '
<语句>', '<语句>', '<语句>', '<语句>', '$return', '<return内容>', '$;'],当前状态栈:[0, 4, 5, 6, 7, 8, 11, 19, 19, 16, 34, 65, 94, 98, 127, 19, 19, 19
, 19, 15, 29, 50]
当前读入字符:#,转移方程为:(50, '#')
(50, '#')
[Error]#201 in line 17, position 15: Unexpected word 'INPUT_END' after '$;'.
```

正确识别异常，分析栈无误。

测试数据 3：左值异常（部分代码）

```
        int c;
        int t;
```

```
3.          if(c>12){
4.              l=print(r);
5.          }
6.          3=l+1;
7.          cnt=cnt+1;
8.          ans=ans*cnt;
9.      }
10.     return ans;
11. }
```

程序输出：

```
#######
当前轮次:230
当前符号栈:['#', '<类型>', '$identifier', '$(', '$)', '${', '<内部声明>', '<语句>', '<语句>', '$while', '$(', '<表达式>', '$)', '${', '<内部声明>', '
$if', '$(', '<表达式>', '$)', '${', '<内部声明>', '<语句串>', '$}'],当前状态栈:[0, 4, 5, 6, 7, 8, 11, 19, 19, 16, 34, 65, 94, 98, 127, 14, 25, 42, 71
, 98, 127, 143, 152]
当前读入字符:$digit_int,转移方程为:(152, '$digit_int')
(152, '$digit_int')
[Error]#201 in line 14, position 9: Unexpected word '3' after '$}'.
```

正确识别异常，分析栈无误。

测试数据 4：函数名缺失（部分代码）

```
1.  int (){
2.      int cnt;
3.      int ans;
4.      int l;
5.      int r;
6.      cnt = 0;
```

程序输出：

```
#######
当前轮次:2
当前符号栈:['#', '$int'],当前状态栈:[0, 1]
当前读入字符:$(,转移方程为:(1, '$(')
(1, '$(')
[Error]#201 in line 1, position 5: Unexpected word '(' after '$int'.
```

正确识别异常，分析栈无误。

测试数据 5：分号确实（部分代码）

```
1.      int r;
2.      cnt = 0;
3.      ans = 1;
4.      while(l<r) {
5.          int c;
```

```
6.          int t;
7.          if(c>12){
8.              l=print(r)
9.          }
10.         l=l+1;
11.         cnt=cnt+1;
12.         ans=ans*cnt;
```

程序输出：

```
#######
当前轮次:202
当前符号栈:['#', '<类型>', '$identifier', '$(', '$)', '${', '<内部声明>', '<语句>', '<语句>', '$while', '$(', '<表达式>', '$)', '${', '<内部声明>', '
$if', '$(', '<表达式>', '$)', '${', '<内部声明>', '$identifier', '$=', '$identifier', '$(', '<实参列表>', '$)'],当前状态栈:[0, 4, 5, 6, 7, 8, 11, 19,
 19, 16, 34, 65, 94, 98, 127, 14, 25, 42, 71, 98, 127, 17, 35, 27, 47, 88, 119]
当前读入字符:$},转移方程为:(119, '$}')
(119, '$}')
[Error]#201 in line 13, position 9: Unexpected word '}' after '$)'.
```

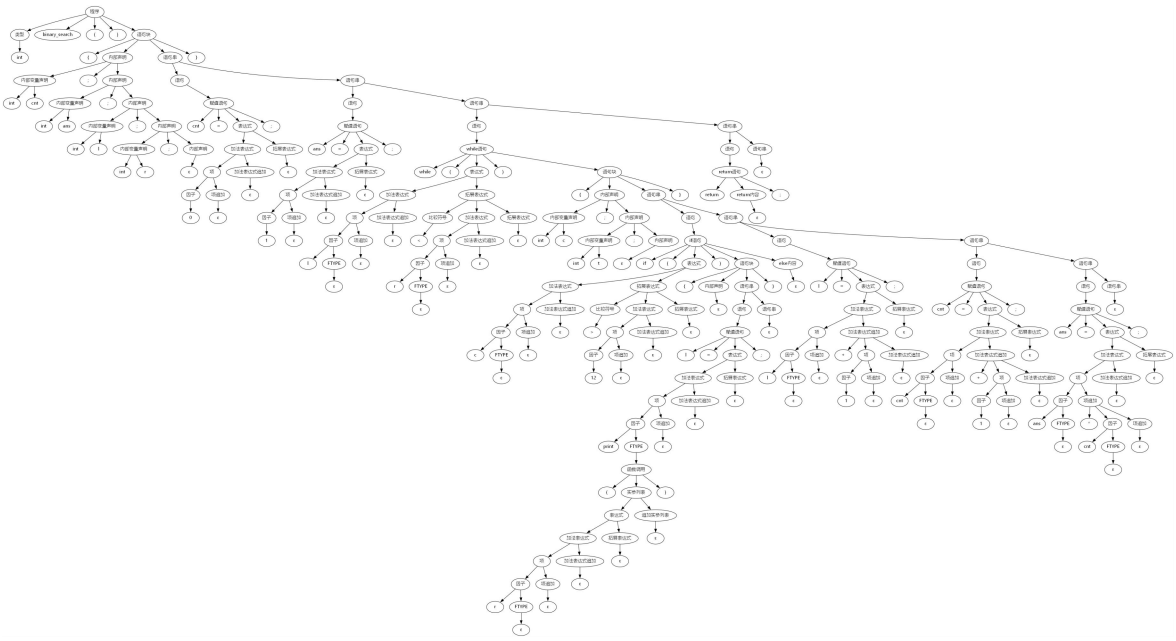
正确识别异常，分析栈无误。

测试数据 6：正确代码

```
1.  int binary_search(){
2.      int cnt;
3.      int ans;
4.      int l;
5.      int r;
6.      cnt = 0;
7.      ans = 1;
8.      while(l<r) {
9.          int c;
10.         int t;
11.         if(c>12){
12.             l=print(r);
13.         }
14.         l=l+1;
15.         cnt=cnt+1;
16.         ans=ans*cnt;
17.     }
18.     return;
19. }
```

程序顺利规约完毕，绘制出分析树：

```
########
当前轮次:378
当前符号栈:['#', '<程序>'],当前状态栈:[0, 3]
当前读入字符:#,转移方程为:(3, '#')
[Info]Garmmar analysis success!
[Info]Compile success!
```



4.2 时间复杂度分析

1. 词法分析器

词法分析器进行的操作主要与读入的文件长度有关，在预处理注释的时候会对文件进行一次遍历，在进行分词的时候又会对文件读入的字符串进行一次遍历，该层循环的开销为 O(2*file_len) = O(file_len)。在每个词被分好后，词法分析器需要查阅该词是否在保留字表内，由于保留字表采用字典方式存储，故而该处查询的期望复杂度为 O(1)，因此整个词法分析器的时间复杂度为 O(file_len)。我们不妨记文件的长度为 n，最终的期望时间复杂度为 O(n)。

2. 语法分析器

显然，不可否定的是语法分析器的时间复杂度与出现的符号个数和所应用的产生式表的长度有关，这里先记产生式表的长度为 m，显然符号个数不会超过 m 的常数倍，也即 O(m) 个。先考虑 findSymbolFirst 函数的时间复杂度贡献，由于采用了记忆

化存储的方法，对每个符号 findSymbolFirst 的有效执行次数不会超过一次，因此这部分的贡献独立于调用函数，单独调用的均摊贡献为 O(1)，总和有 O(m) 的时间复杂度贡献。

于是，对于 findFirst 函数，由于其每次执行都只有常数次 findSymbolFirst 的调用，其单独调用的均摊贡献也为 O(1)。

接下来考虑 findClosure 函数，findClosure 函数需要对每个项目都生成闭包，采用记忆化存储的方法后，一个项目簇的闭包可以从该项目簇中每个元素的闭包叠加而来，考虑到生成项目中推出的子项目同时也是我们需要计算的对象，也因此每个项目的贡献度都只是在其他项目上简单叠加，再考虑到每个产生式只包含有限个右项，也即总项目的个数不会超过 O(m)。因此求所有项目集的闭包的总贡献度不会超过 O(m^2)【其中的一个 m 是由于项目集簇之间所需的深拷贝时间而引入的】，查询时间均摊为 O(m)。

然后考虑 findGoto 函数，该函数在调用 findClosure 函数的基础上引入了尝试进行 dot 前进的工作，该层循环的长度不超过 O(m)，整体循环的事件复杂度为 O(m) + O(m) = O(m)。

再者考虑 initProjectSet 函数，该函数通过不断调用 findGoto 函数进行计算来获取完整的有效项目集。对于不同产生式而言，该部分函数的时间复杂度的区别较大，这里考虑平均的情况，所有状态下推的项目集数量相对均匀，此时对任意一个有效项目簇，其包含的子产生式的 dot 位置一定小于其父产生式的 dot 位置，从而有产生的状态数为 O(mlogm) 级，同时遍历符号表的复杂度为 O(m)，对这些状态都进行 goto 调用的复杂度为 O(m)，因此整个遍历循环的时间复杂度为 O(m^3log m)。尽管还需在其中生成 action_goto 表，但是由于采用哈希字典的存储方式，其存储平均复杂度为 O(1)，该函数的综合时间复杂度为 O(m^3 log m)。

对于 GParser 而言，其遍历 input 表并规约的时间复杂度不超过 O(机内词语长度)，这部分虽然不好评估，但是显然不超过 O(n)，因此可以用 O(n)取代，每次遍历中状态的转移是 O(1)的，因此对 GParser 而言整体的时间复杂度为 O(m^3 log m + n)。

绘制语法树的部分，由于是直接将产生的树变形后调用 graphviz 三方库进行绘制，此部分的时间复杂度不做评估。

因此，对于整个编译器而言，总时间复杂度为 O(m^3 log m + n) + O(n) = O(m^3 log m + n)。

4.3 调试中遇到的一些问题

本次词法语法分析器的开端是比较顺利的，这或许是因为词法分析器的 DFA 状态机都是手动编写，因而在调试过程中遇到问题能够很快的想通并且纠正过来。

我们在语法分析器阶段遇到的问题主要是对于符号的处理考虑，因为在某次测试中我们添加了 %= 符号，却意外发现无法识别。我们通过手动模拟的方式，才发现我们程序在处理符号的过程中，没有考虑到一个可接受串的前缀不一定是一个可接受串的问题，从而造成了程序的异常。在修正后的测试中，我们的程序均运行正常，开发较为顺利。

然而当我们希望将这种顺利的状态延续到语法分析器当中时，我们发现事情开始慢慢偏离我们的预期，我们写完后第一次运行程序就直接进入了死循环，而我们不知道如何解决。在之前的计算机课程的学习过程中，我们一直习惯于使用 Debug 功能来查询程序的状态。然而这次调试的程序状态集是如此的大，以至于我们的 Debug 调试器的整个屏幕都无法容纳下数据的输出，为此我们采用了在生成有效项目簇的同时就进行打印的手段观察 FindGoto 函数返回的闭包，从而发现问题出现在大约项目号 50 以后的项目簇之中，这些项目簇的大小意外的变大，并且当中存在有相当数量的重复项目。我们尝试对 FindGoto 函数的返回先进行去重之后，我们的程序才终于能够在没有错误的情况下顺利的运行结束。

可是之后的测试让我们深受打击。在我们提前编写好的样例上，我们的程序没有通过任何一个样例，并且每次都是在一个十分靠前的位置提示 action_goto 集中不存在相应的转移。为此，我们将整个 BFS 过程中发生的状态转移打印出来，并且针对相应的转移方程手算闭包。我们最终发现，这是由于 findGoto 集中的返回值出现了问题导致的，某条规约项目本来应该有四个展望字符，而 findGoto 集中却只返回了一条包含其中一个展望字符的项目。

因此我们认定可能是 FindClosure 闭包出现了一些问题，可是当我们尝试定位这个问题的时候，我们惊讶的发现 findClosure 闭包在计算的过程中是完完整整的计算

出了四个展望字符的！

经过了许久的调试，我们最终定位到了下面这段代码：

```
1.   for nas in new_accept_set: #遍历展望串各元素，都要添加入 closure 集
2.       newp = deepcopy(p)
3.       newp['dot'], newp['accept'] = 0, nas
4.       if not newp in closure_set:#进行元组去重，避免重复元素添加
5.           closure_set.append(newp)
```

高亮标记的 deepcopy 函数在我们初版的语法分析器中并不存在，我们最初直接使用等号作为赋值。使用 C++的惯性让我们认为这条语句的作用就是将 p 中的对象给 newp 进行一次拷贝，可是在 Python 中对于一个列表的拷贝不会将其内容全部再生成一边并赋给另一个对象，而只是将这个对象的地址赋给另一个对象，这就导致我们修改 newp 的 accept 属性的时候，其实是在原来的那个项目上进行的修改——这就导致 closure_set 对象里永远只有一个对象的实例存在，我们的问题终于得到了解释，在添加上 deepcopy 函数后，我们的语法分析器运行正常。

在最终的测试过程中，我们还遇到了名为“string out of index”的错误，其在所有规约过程中都工作正常，出现条件仅仅是如果程序前面都顺利规约完成，却在最后对栈底的’#’尝试进行规约又失败后才会产生，可谓是一个非常难以检索到的小问题。

经过我们的调试，这是因为我们在完成了 input 栈的升级之后（这项升级用于在词法分析的过程中同时容纳位置信息，从而使得语法分析的错误故障能够给用户提出对应的位置定位）产生的。这次故障的排出相对来说比较轻松，我们很快定位到了 input 栈的初始化代码

```
1.   def GParser():
2.       initProjectSet() #初始化项目集，构建 action 和 goto 表
3.       input_st = [['#', 'INPUT_END', w_dict[-1][2]]] + w_dict[::-1] #设置输入机内序列
4.       state_st, sym_st, id_st, son_st = [0], ['#'], [], [] #设置工作栈
5.       t_cnt=1 #轮次记录变量
```

其中黄色部分为我们修复 bug 所添加的变量，这个问题的产生其实就是因为我们没有采用初始化函数的方式去生成 input 栈所产生的。我们添加了位置信息后，却没有去考虑到栈底的’#’终结符应该填充什么样的位置（最后我们决定将机内词汇序列的最后一个元素的位置填入）。这样固然带给我们快速的开发速度，但也相对牺牲了

一些东西——这样一个非常细小的错误，如果不是恰好测到了前文中的样例 4（函数体缺少右括号）的问题是不可能被发现的。我们意识到，在一个对外编译器中，如果存在一个细小的问题，可能就会对一个非常重要的项目产生致命性的影响，这也要求着我们开发者尽可能针对自己的代码构造可能存在的边界问题，努力提升代码的质量。

5 总结与收获

5.1 收获、思考与感想

经过本次大作业，我们巩固了第四第五章的知识。通过对于 FIRST 集合求解，对 ACTION 表和 GOTO 表的求解，CLOSURE 闭包的求解以及 LR(1)过程的具体实现，更好地促进了我们对于理论知识的理解与掌握。有些知识上的误区，可能如果不经历真正的实操开发，只靠老师的讲解和笔头几道练习题的训练是完全不能得到纠正的。一个非常具体的例子就是上课讲解 LR(1)的 ppt 中的规范推导部分：

- 形式上我们说一个**LR(1)项目[A→α·β, a]对于活前缀γ是有效的**，如果存在规范推导

$$S \overset{*}{\Rightarrow} \delta A \omega \Rightarrow \delta\alpha\beta\omega$$

其中，1) γ=δα；2) a是ω的第一个符号，或者a为#而ω为ε。

我们三位同学在上课的时候都认为自己已经理解了这个 α ・β，然而当我们坐在电脑面前时，我们三个人同时面面相觑，同时选择再去温习 ppt 和课本。我们发自内心的认为，没有任何一种学习编译器的方式能够代替亲自动手写一个编译器！

此外，这次实验同时也是我们组三位同学第一次选择使用 Python 作为一个大项目的实现语言，在此之前我们的所有项目均使用 C++进行开发。其实我们本来也尝试过使用 C++进行这次程序的开发，我们也确实写了一个 C++版本的词法分析器，但是当我们进一步尝试使用 C++进行语法分析器的开发时，我们组感到了难以描述的迷茫。C++所营造的严谨的数据定义固然让其成为了效率很高的一门语言，但是与此同时，也迫使我们浪费大量的时间去考虑如何更好地设计我们的存储结构、访问结构和成员体和成员操作上——这并不是我们组学习编译原理想加深理解的东西。Python 牺牲效率换取的是敏捷性开发的自由，我们可以专注于如何实现我们的编译器、如何将课堂上讲的 LR1 分析方式落实到代码上、以及如何设计出更优化、更高效的算法，这是我们认为这次实现词法语法分析器让我们感到动力满满的一件事，我们期待着下一次中间代码生成大作业的到来！我们信心满满！