同济大学计算机科学与技术系

编译原理课程综合实验一

院　　系　电子与信息工程学院

专　　业　计算机科学与技术

学　　号　1951566　1950084

姓　　名　贾仁军　陈泓仰

日　　期　2021.11.7

目录

1 需求分析

1.1 输入输出约定

需要由用户提供的输入位于 input 文件夹中，分别是源程序和产生式。

由词法分析器生成，供语法分析器使用的文件位于 intermediate 文件夹中。

由语法分析器生成的，供查阅的文件位于 output 文件夹中。

源程序输入

input\源程序.txt

直接使用 ppt 中的源程序输入即可。具体请见 1.3。

*ppt 中函数调用语句少了分号。 a=program(a,b,demo(c))

**为了测出并修改词法/语法分析器的 bug，我们构造了许多组具有特点的数据，详见 4.1。

文法规则输入

input\产生式.txt

文法规则的输入格式如下：

1. 一行只能有一个产生式；对于存在多种可能性的产生式如 S->a|b|c，需要分成若干个不同的产生式分别输入。
2. 产生式左右部用:分隔
3. 产生式右侧符号以空格分隔，右部为空可以加入$
4. 第一行产生式左部必须为初始符号。初始符号在整个产生式集合中只能出现这一次。

终结符和非终结符会自动区分。程序统计所有出现过的符号，并将在左部出现过的标记为非终结符，其余标记为终结符。

针对大作业，我们最终使用的语法规则示例如下：

```
sstart:start
start:external_declaration start
start:$
external_declaration:declaration
external_declaration:function_definition
type_specifier:char
type_specifier:int
```

……

完整语法规则见 1.3。

注：PPT 中的语法规则包含可选符号[]，需要手动将其转换；此外，貌似还有错误的规则，使我的程序无法正常接受待分析句，例如：

<内部声明> ::= 空 | <内部变量声明>{; <内部变量声明>}

因此，我们更换了语法规则。最终使用的规则如上，由[程森，高曾谊]小组提供。

词法分析输出

由词法分析器生成，供语法分析器使用的文件位于 intermediate 文件夹中。

文件	内容
processed_sourceCode.txt	将输入的源程序处理后，供语法分析器读入的代码。
names.txt	对 processed_sourceCode.txt 文件的补充。

以下是一个例子：

文件	内容
源程序.txt	`int program(int a, int b, int c){}`

processed_sourceCode.txt	int
	identifier
	(
	int
	identifier
	,
	int
	identifier
	,
	int
	identifier
	)
	{
	}

names.txt	program a b c

具体设计原因见 3.1。

语法分析输出

由语法分析器生成的，供查阅的文件位于 output 文件夹中。

本语法分析器（LR1）输出的文件清单为：

文件	内容
production.txt	根据输入的产生式生成的项目
first.txt	所有非终结符的 first 集，供求闭包时使用。
closure.txt	每个项目的编号、内容，以及其对应的闭包的编号。
xmjz.txt	从[S'→S,#]开始，根据闭包和 first 集求得的项目集族。
goto.txt	从[S'→S,#]开始，根据闭包和 first 集求得的转移函数 Go。
lr.txt	根据 Go 函数和项目集族求得的 Action/Goto 表。
analyze.txt	根据 A/G 表，对输入的句子进行移进/规约分析的过程（分析栈）。
语法树.html	根据每一步分析结果产生的语法树。

除了最终的语法树.html，其它输出文件主要供 debug 使用。

1.2 程序功能

词法分析器.cpp

词法分析器读入源程序，提取出源程序中的每一个符号，输出到两个文件：names.txt 和 processed_sourceCode.txt。

输入	内容
input\源程序.txt	见源程序输入
输出	
intermediate\processed_sourceCode.txt	见词法分析输出
intermediate\names.txt	见词法分析输出

语法分析器_LR1.py

语法分析器读入产生式，构造出 Action/Goto 表；

再读入句子，输出分析过程和语法树。

输入	内容
input\产生式.txt	见文法规则输入

intermediate\processed_sourceCode.txt	
intermediate\names.txt	
输出	见语法分析输出

1.3 测试数据

input\源程序.txt

```
int a;
int b;
int program(int a,int b,int c)
{
 int i;
 int j;
 i=0;
 if(a>(b+c))
 {
      j=a+(b*c+1);
 }
 else
 {
      j=a;
 }
 while(i<=100)
 {
      i=j*2;
 }
 return i;
}

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

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

input\产生式.txt

```
 sstart:start
 start:external_declaration start
 start:$
 external_declaration:declaration
 external_declaration:function_definition
 type_specifier:char
```

```
type_specifier:int
type_specifier:double
type_specifier:float
type_specifier:void
declaration:type_specifier declaration_parameter declaration_parameter_suffix ;
declaration_parameter:identifier declaration_parameter_assign
declaration_parameter_assign:= expression
declaration_parameter_assign:$
declaration_parameter_suffix:, declaration_parameter declaration_parameter_suffix
declaration_parameter_suffix:$
primary_expression:identifier
primary_expression:number
primary_expression:( expression )
operator:+
operator:-
operator:*
operator:/
operator:%
operator:^
operator:&
operator:<
operator:>
operator:!=
operator:==
operator:<=
operator:>=
assignment_operator:=
assignment_operator:+=
assignment_operator:-=
assignment_operator:*=
assignment_operator:/=
assignment_operator:%=
assignment_operator:^=
assignment_operator:&=
assignment_operator:|=
assignment_expression:identifier assignment_operator expression
assignment_expression_list_suffix:, assignment_expression assignment_expression_list_suffix
assignment_expression_list_suffix:$
assignment_expression_list:assignment_expression assignment_expression_list_suffix
assignment_expression_list:$
expression:constant_expression
expression:function_expression
constant_expression:primary_expression arithmetic_expression
arithmetic_expression:operator primary_expression arithmetic_expression
arithmetic_expression:$
function_expression:identifier ( expression_list )
expression_list_suffix:, expression expression_list_suffix
expression_list_suffix:$
expression_list:expression expression_list_suffix
expression_list:$
function_definition:type_specifier identifier ( function_parameter_list ) compound_statement
function_parameter_list:function_parameter function_parameter_list_suffix
function_parameter_list:$
function_parameter_list:void
function_parameter_list_suffix:, function_parameter function_parameter_list_suffix
function_parameter_list_suffix:$
function_parameter_list_suffix:void
function_parameter:type_specifier identifier
compound_statement:{ statement_list }
statement_list:statement statement_list
statement_list:$
```

```
statement:expression_statement
statement:jump_statement
statement:selection_statement
statement:iteration_statement
statement:compound_statement
statement:declaration
expression_statement:assignment_expression_list ;
jump_statement:continue ;
jump_statement:break ;
jump_statement:return expression ;
jump_statement:return ;
selection_statement:if ( expression ) statement else statement
iteration_statement:while ( expression ) statement
iteration_statement:for ( declaration expression ; assignment_expression ) statement
```

*输出结果见 4.1.1。

**为了测出并修改词法/语法分析器的 bug，我们还构造了许多组具有特点的数据，详见 4.1。

2 概要设计

2.1 任务分解及分工

见 1.2。

贾仁军：词法分析器。

陈泓仰：语法分析器、报告编写。

2.2 数据类型定义

词法分析相关

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

语法分析相关

```
start_symbol = ""  # 初始符号
symbol = set()  # 所有符号集合
terminal_symbol = set()  # 终结符集合
non_terminal_symbol = set()  # 非终结符集合

产生式 = []  # {'left': "S'", 'right': ['S']}
项目 = []  # {'left': "S'", 'right': ['S'], 'point': 0}
```

```
新项目 = []  # {'left': "S'", 'right': ['S'], 'point': 0,
"origin": 0, "accept": "#"}
首项 = {}  # 每个非终结符 B 的形如 B→·C 的产生式的序号 首项['S']={2,
5}

closure = []  # 每个项目的闭包 closure[0]={0, 2, 5, 7, 10}
closureSet = []  # 项目集族 closureSet[0]={0, 2, 5, 7, 10}

goto = [] # go[状态 i][符号 j] 该数组依次存储了不同内容，分别为:
# = Closure{项目 x, 项目 y}
# = {项目 x, 项目 y, 项目 z}
# = 状态 k
# 进入 Action/Goto 环节后，go 函数会被转换为 goto 函数
# goto[状态 i][符号 j]= 0:accept / +x:移进字符和状态 x（sx）/ -x:用
产生式 x 归约（rx）/ 无定义:err

first = {} # first['F']={'(', 'a', 'b', '^'}
first_empty = []  # first 集中含有空的非终结符集合 {"E'", "T'",
"F'"}

statusStack = [0] # 状态栈
charStack = ['#'] # 符号栈
inp = "" # 输入栈

nodeStack = [] # 语法树结点 nodeStack[cntNode]["name"]="123"
nodeStack[cntNode]["children"]=[1, 2, 3]
```

2.3 主程序流程

词法分析器

读入源程序
对源程序进行过滤
　　删除空行、多余空格、无用控制符、单行注释、多行注释。
加载保留字表、界符运算符表
源程序指针指向（过滤后的）源程序开头
Repeat
　扫描下一个符号
　判断是保留字、标识符还是数字，输出到不同文件
Until 指针尚未碰到程序尾

语法分析器

根据输入的产生式，得到终结符集合、非终结符集合，项目数组
求出所有非终结符的 first 集，供求闭包时使用。

求出每个项目的闭包
从[S'→S,#]开始，根据闭包和 first 集求得转移函数 Go 和项目集族
根据 Go 函数和项目集族求得 Action/Goto 表。
输入待分析句子，根据 A/G 表，对句子进行移进/规约分析。
根据每一步分析结果产生语法树，输出

可以看出，LR1 分析器的构造过程中涉及的步骤，和语法分析输出文件清单是相符的。

虽然词法分析器和语法分析器程序流程长度看起来差不多，但是前者涉及的设计细节、算法及优化非常多，详见 3.1。

2.4 模块间的调用关系

见 1.2。

3 详细设计

3.1 主要函数分析及设计

词法分析器 / 读入

采用 fstream，直接从文件读入到 string，非常好用

```
ifstream t("input\\源程序.txt");
string str((std::istreambuf_iterator<char>(t)), std::istreambuf_iterator<char>());
```

词法分析器 / 过滤

```
string resProgram = filterResource(str);
```

若读取到空格，则将后续空格略过，只保留一个。（因为本分析器不接受字符串，可以不做判断）

若读取到//，将//以及该行剩余部分跳过。

若读取到/*，将/*以及后续内容跳过，直到遇到*/。*/后的字符正常接受。

注意“/*/”的情况

若读取到无用控制符(\n \t \v \r)，跳过。

必须先过滤//，后过滤\n

词法分析器 / 分解

从头开始读入字符。
若第一个字符为字母
    读入后续字符，直到非字母也非数字。
    判断是否为保留字（int、while 等）
    若是，输出到 processed_sourceCode.txt。
    若否
        输出到 names.txt
        向 processed_sourceCode.txt 输出“identifier”
 若第一个字符为数字
    读入后续字符，直到非数字。
    输出到 names.txt
    向 processed_sourceCode.txt 输出“number”
 若第一个字符为其它字符
    若为( ) * + , - / ; { }中的一个，接受。输出到 names.txt
    若为< = >中的一个
        向后 peek 一位。
        若为=，接受。输出<=或==或>=到 names.txt
        若否，不接受（不移动指针）。输出第一位到 names.txt
    若为!
        向后 peek 一位。
        若为=，接受。输出!=到 names.txt
        若否，不接受。报错。终止扫描。
    若不为以上字符，报错。终止扫描。

字符被区分成标识符、数字和保留字后，将被输出到两个文件：names.txt 和 processed_sourceCode.txt。其中，processed_sourceCode 中将每个符号输出到单独的一行。但：其中的标识符将被替换为 identifier，其中的数字将被替换为 number。

例如：int program(int a, int b, int c){}将被解析为：

```
int
identifier
(
int
identifier
,
int
identifier
,
int
identifier
```

```
)
{
}
```

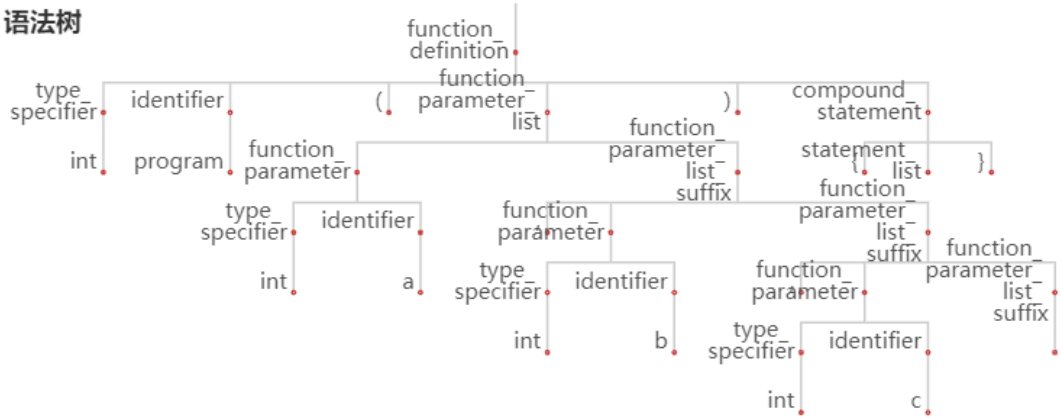
为什么要这样做？因为这样可以固定住产生式。否则，根据源程序的不同，输入语法分析器的产生式也要变化。例如在这段程序，如果不采用这种替换的方式，则需要额外加入产生式：

identifier→a|b|c|program

但这样做，又会带来一个新的问题：语法分析器只能分析到 identifier 这层，无法知道其最终对应的到底是哪个标识符。为此，词法分析器额外输出所有的标识符真实值到另一个文件 names.txt（也是每个符号一行）：

```
program
a
b
c
```

这样，当语法分析器在分析句子时，每当移进一个保留字（或数字），就从 names.txt 中当前的首行读取该保留字（或数字）的真实值，加入语法树中：



语法分析器 / 读入

本语法分析器会自动区分终结符和非终结符，以减少输入负担。

程序在读取产生式的过程中，统计所有出现过的符号，并将在左部出现过的标记为非终结符，其余标记为终结符。

由 json 格式而来的灵感，我对读入的产生式以 dict 格式存储：

```
{
    'left': 'sstart',
    'right': ['start'],
```

```
    'point': 0,
    'origin': 0,
    'isTer': False
}
```

加入接受的终结符后，便成为了新项目。新项目还记录了其原始对应的产生式编号，用于在构造 Action/Goto 表的规约(r)时，快速找到对应的产生式。

```
{
    'left': 'statement_list',
    'right': ['statement', 'statement_list'],
    'point': 1,
    'origin': 65,
    'isTer': False,
    'accept': '*'
}
```

此外，在生成项目的同时，预处理出所有圆点在最左侧的项目，按照左部为索引分类存储，以加快求取闭包的过程：

首项[左部符号'A'][接受符号'a'] = {项目 1，项目 2，项目 3}

语法分析器 / 求 First 集

对每个终结符，置其 First 集为自己。

对每个非终结符，置其 First 集初始为空。

首先，找出所有可以导出空字的非终结符。

> 复制一份所有产生式。
> 构造一个队列。对形如 A→ ε 的产生式，将其左部加入队列。
> 每次从队头取符号。遍历所有产生式。
> 若该符号出现在左部，将该产生式删除。
> 若该符号出现在右侧的列表中，将其从列表中删除。
> 若删除后列表为空，将该产生式左部加入队列。

由此，我们可以在 O(产生式中的所有符号)的时间里，求得所有可以导出空字的非终结符。将他们单独存储为一个集合，名为 first_empty。

那么，为何需要单独存储可以导出空字的非终结符？

因为，在之后求 first 集，以及闭包中计算 First(ABC)的值时，只需：

```
for symbol in ['A', 'B', 'C']:
    symbolFirst |= first[symbol]
    if symbol not in first_empty:
        break
```

现在有了 first_empty，如何求每个非终结符的 first 集？

```
f = 1
while f:
    f = 0
```

```
    for item in 产生式:
        for sym in item["right"]:
            if not first[item["left"]].issuperset(first[sym]):
                f = 1
                first[item["left"]] |= first[sym]
            if sym not in first_empty:
                break
```

如上，不断遍历所有产生式，例如对 A→BCDE，则 First(A)|=First(BCDE)。对每次遍历完毕，只要有 1 个非终结符的 First 集出现了变动，则再次遍历。

语法分析器 / 求单个项目的闭包

由于项目很多，如果还采用上面求 First 集的方法求闭包，会很慢。因此，我们采用带记录的 dfs 的方式。

```
for i, item in zip(range(len(新项目)), 新项目):
    closure[i].add(i)
    closure[i] = find_closure(i, i)

def find_closure(当前处理的项目 i, 本次待求闭包的项目 ini):
    令当前项目 i 表达为：[A→α·Bβ,a]
    For first(βa) 中的每个终结符号 b:
        For 首项[B][b] 中的每个项目 j:
            若 j 不在 closure[ini]中:
                closure[ini] += {j}
                closure[i] += {j}
                closure[i] |= find_closure(j, ini)
    return closure[i]
```

可以看到，我们程序预处理了 首项[B][b]，因此不需要每次从所有项目里找出“左部为 B，原点在右部最左侧，且接受符号为 b”的项目；此外，first(βa)可以如上所述快速求取。

假如当前要求 closure[项目 0]。首先找到项目 1 属于该闭包，则：

closure[0] = {0} ∪ closure[1]

假如 closure[1] = {1, closure[2]}。本程序快就快在，在求出 closure[0] = {0, 1, closure[2]}的同时，也将 closure[1]= {1, closure[2]}记录了（黄色荧光部分）。等到求 closure[1]的闭包时，就不必再把 closure[2]再求一次了。

另外，灰色荧光的语句乍一看不必要，但是漏掉的话会让程序陷入死循环。原因易证。

那么，为何需要提前求出每个项目的闭包？

已经对每个项目求出其闭包后，之后想要计算某个状态的闭包时，只需要把这个状态包含的所有项目的闭包做集合或运算即可：

```
    for itemOrd in goto[i][j]:
        newSet |= closure[itemOrd]
```

语法分析器 / 求项目集族和 Go 转移函数

初始化：closureSet[0]=closure[0]。即：置 0 号项目集初始包含的项目为[S’ →S,#]的闭包。

然后依次求每个项目集（状态）的 Go 转移函数：

```
i = 0
while (i < len(closureSet)):
    find_Goto(i)
          i += 1
```

对每个项目集 I,终结符 x,先找到 I 中可以接受 x 的项目,假设为{1, 2 ,3}；他们分别接受 x 后项目的序号变为{4, 5, 6}

则 Goto[I][x]=closure[4] ∪ closure[5] ∪ closure[6](假设={7,8,9})

由于已经得到每个项目的闭包集合，因此 Goto[I][x]可以由此直接计算出来。

最后，判断 closureSet 中是否已有该状态。

若有，假设为 10 号状态，则 Goto[I][x]=10；

若无，则新建一个状态。假设 closureSet 中当前已有状态[0~11]，：

closureSet[12] = Goto[I][x] (={7,8,9})

Goto[I][x]=12

这里也可以看出 python 的灵活性，一个 Goto 先后存了不同类的数据。

语法分析器 / 求 Action/Goto 表

这部分比较简单。若 go[I][x]=k，则 goto[I][x]=sk;（移进）

再遍历项目集族中所有状态 I，找到将其中的终结项目 t（原点在右部最右侧的项目）。假设该项目 t 接受的符号为 y，对应的产生式序号为 k，则：

goto[I][y]=rk（规约）

特别的，若 y == #, k == 0（S’→S），则：

Goto[I][#]=accept

语法分析器 / 分析输入的句子

开一个符号栈，一个状态栈，一个输入栈，根据 A/G 表分析即可。

这部分也比较简单，麻烦的是输出的格式。不一一。

语法分析器 / 构造语法树

新开一个节点栈，在分析句子的同步运行。

每当移进一个字符时新建一个（叶子）节点，节点名字即为移进的符号。将该节点压入节点栈。

每当用产生式 t 规约时，新建一个节点，节点名字为 t 的左部符号，节点的孩子为节点栈最顶端的 k 个。K 为 t 的右部的长度（即符号数量）。

这部分原理比较简单，但是实现的时候有一些细节：

1. 不可以直接用状态作为节点的编号，因为状态可以多次进入，并不唯一。
2. 节点的孩子加入后要逆序排列一次，因为是从结点栈里弹出来的。不做逆序最后画出来的节点树会左右倒过来，也就是“A→(B)”会变成“)B(”。
3. 向空产生式规约时，最好处理为额外新建一个名字为空的节点，将先新建的节点指向该“空节点”。（即一共新建 2 个节点）
4. 若规约的产生式左部为“identifier”或“number”时，同样额外新建一个节点，名字从 names.txt 中读入，将先新建的节点指向该节点。

3.2 函数调用关系

见 2.3。

无论词法分析器还是语法分析器，整个程序流程总体均为从上到下，由 main 函数依次调用。例如，词法分析器中，由 main 函数依次调用读入函数、过滤函数和分解函数。

各部分的实现中没有子函数相互调用，仅在语法分析器的求单个项目的闭包时，使用了（带记录的）dfs，存在自调用（递归）

4 调试分析

4.1 测试数据及测试结果

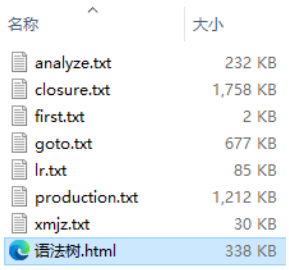
首先是本次作业所最终实现的类 C 文法的测试，后面是 LR(1)针对的各个模块单独设计的小测试。

类 C 文法（含过程调用）

input：见 1.3

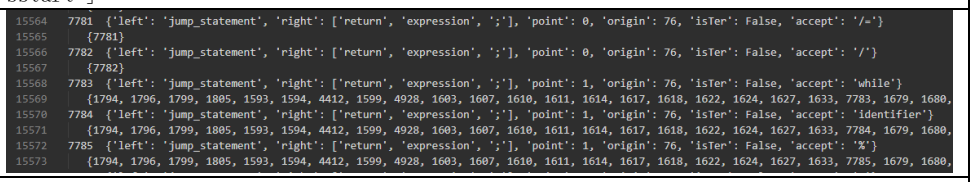
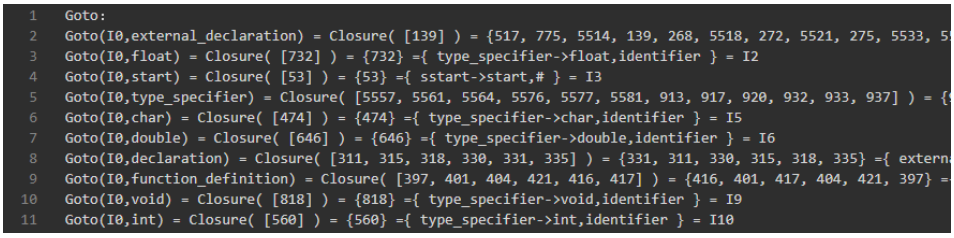
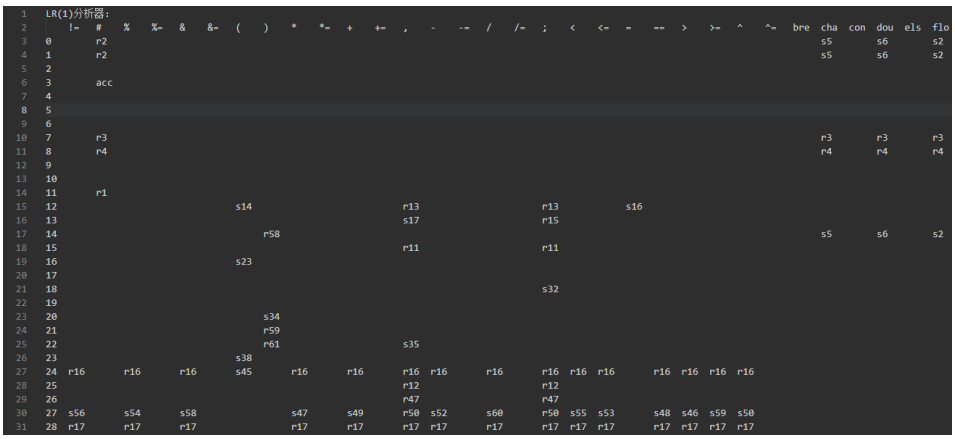
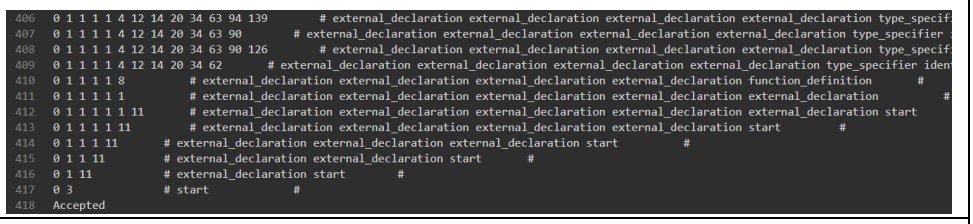
intermediate:见 1.1.3

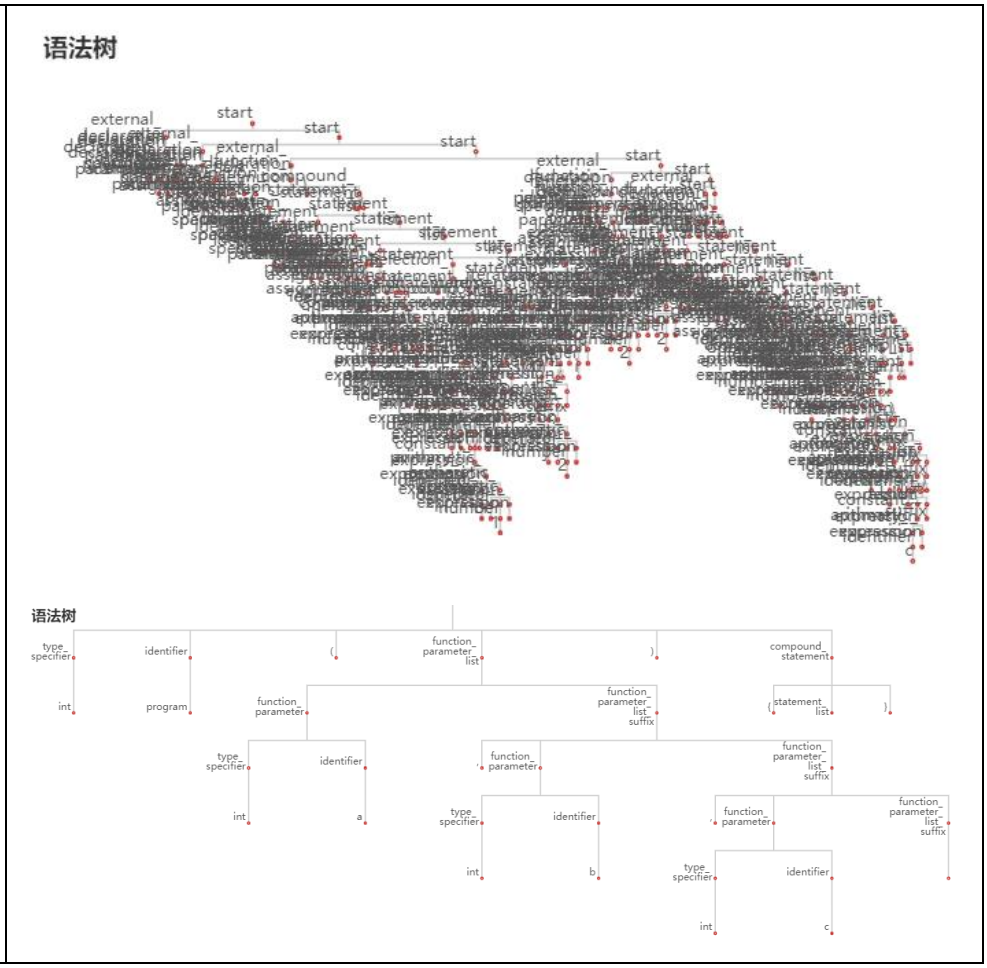
output：



因为文件内容过大，以下列出几个关键指标，或者输出文件大致格式：

文件	内容
production.txt	共 210 个项目。

	含不同接受符的项目共 43×210=9030 个。 其中，圆点在开头的产生式共 3383 个。
first.txt	共 12 个非终结符的 first 集中含有空字。 ['start','declaration_parameter_assign','declaration_parameter_suffix', assignment_expression_list_suffix','assignment_expression_list', arithmetic_expression','expression_list_suffix','expression_list', function_parameter_list','function_parameter_list_suffix','statement_list', sstart']
closure.txt	
xmjz.txt	共有 233 个项目集。0 号项目集有 32 个项目。 （接受符不同算做不同项目）
goto.txt	
lr.txt	
analyze.txt	共 417 行。 
语法树.html	支持缩放和拖动，以及子树展开与缩回。



测试文法强度

产生式：S:A A:eA|ε

源程序：eee

该文法的 I0 中同时含有移进项和规约项。LR0 无法处理，LR1 方能处理。

```
LR(1)分析器:
    #    +    A
0   r2   s1   2
1   r2   s1   3
2   acc
3   r1
```

```
['+', '+', '+', '#'] 的分析栈:
0          #          + + + #
0 1              # +                + + #
0 1 1            # + +              + #
0 1 1 1          # + + +            #
0 1 1 1 3        # + + + A          #
0 1 1 3          # + + A            #
0 1 3            # + A              #
0 2              # A                #
Accepted
```

测试是否正确的处理了空产生式

直接查看项目（production.txt）。

对产生式 D→ ε ，以下是正确的：

```
{'left': 'D', 'right': [], 'point': 0, 'origin': 5, 'isTer': True}]
```

以下是错误的：

```
{'left': 'D', 'right': [], 'point': 0, 'origin': 5, 'isTer': True}]
{'left': 'D', 'right': [], 'point': 1, 'origin': 5, 'isTer': True}]
```

以下也是错误的：

```
{'left': 'D', 'right': ['$'], 'point': 0, 'origin': 5, 'isTer': True}]
```

产生式：S:A A:bBc B: ε

源程序：bc

```
LR(1)分析器：
    #     b     c     A     B
0         s1          2
1                r2          3
2   acc
3                s4
4   r1
```

```
['b', 'c', '#'] 的分析栈：
0          #          b c #
0 1              # b                c #
0 1 3            # b B              c #
0 1 3 4          # b B c            #
0 2              # A                #
Accepted
```

测试 first 集是否计算正确

产生式：

$$E \rightarrow TE'$$
$$E' \rightarrow +E \mid \varepsilon$$
$$T \rightarrow FT'$$
$$T' \rightarrow T \mid \varepsilon$$
$$F \rightarrow PF'$$
$$F' \rightarrow * F' \mid \varepsilon$$
$$P \rightarrow (E) \mid a \mid b \mid \Lambda$$

源程序：((a))

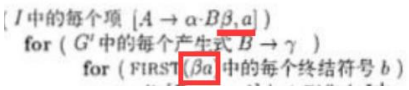
First 集：

非终结符	First 集	是否包含$

```
["E'", "T'", "F'"]
F
a ( ^ b

T'
a ( ^ b  $

T
a ( ^ b

E
a ( ^ b

E'
+  $
```

E	(, a, b, ^	
E’	+	$
T	(, a, b, ^	
T’	(, a, b, ^	$
F	(, a, b, ^	
F’	*	$
P	(, a, b, ^	

测试计算闭包时 first(βa)是否错误（非常强的数据）



假如项目为[B→b·CDc，b],则 first(βa)=first(Dcb)，而非 first(Db)。若写错，程序将会在以下文法中出错：

产生式：S:A　A:BA|ε　B:bCDc　C:a　D:ε

源程序：bacbac

```
Goto:
Goto(I0,b) = Closure( [28, 30] ) = {28, 30, 47} ={ B->bCDc,# B->bCDc,b C->a,c } = I1
Goto(I0,A) = Closure( [4] ) = {4} ={ S->A,# } = I2
Goto(I0,B) = Closure( [12] ) = {20, 8, 24, 26, 12} ={ A->,# A->BA,# B->bCDc,# B->bCDc,b A->BA,# } = I3

Goto(I1,a) = Closure( [51] ) = {51} ={ C->a,c } = I4
Goto(I1,C) = Closure( [32, 34] ) = {32, 34, 55} ={ B->bCDc,# B->bCDc,b D->,c } = I5
```

bac→bCc→bCDc→B→BA→A→S

当面临 c 时，a 需要被规约为 C。若 first(βa)被错误的视为 first(Db)=b（first(D)=$）,则无法正确规约。

```
LR(1)分析器:
    #    c    a    b    A    B    C    D
0   r2             s1   2    3
1                  s4                  5
2   acc
3   r2             s1   6    3
4        r4
5        r5                            7
6   r1
7        s8
8   r3             r3
```

该测试数据同时可以测试文法强度、空产生式。

4.2 调试过程存在的问题及解决方法

4.2.1 从 C++到 python

如第一部分展示，词法分析器后缀为 cpp，而语法分析器后缀为 py。

实际上，我们一开始打算均用 C++语言来编写整个项目。在词法分析器阶段，遇到的问题不大；但在语法分析器阶段出现了极大的问题：

读入

C++读入产生式很麻烦，而 python 可以：

```
if line=="":
    break;
line = line.strip().replace('$', '')
right = line.split(':')[1]
item["right"]=list(filter(lambda x: x != "", right.split(' ')))
```

短短几行，可以滤过空行、滤除多余空格、处理空产生式、将产生式右部加入项目。

存储

这是我对每个项目的存储方式：

```
{'left': 'E', 'right': ['T', "E'"], 'point': 0, 'origin': 0, 'isTer': False, 'accept': '#'}
```

在 C++，这需要开辟一个 struct，struct 中还要包含 vector。难以随时更改和变动。

又譬如，在 Go 转移计算中，求出一个新的集合时，想要比对该集合是否出现过（即，是否是一个已有的状态），python：

```
if newSet in closureSet:
        goto[i][j] = closureSet.index(newSet)
```

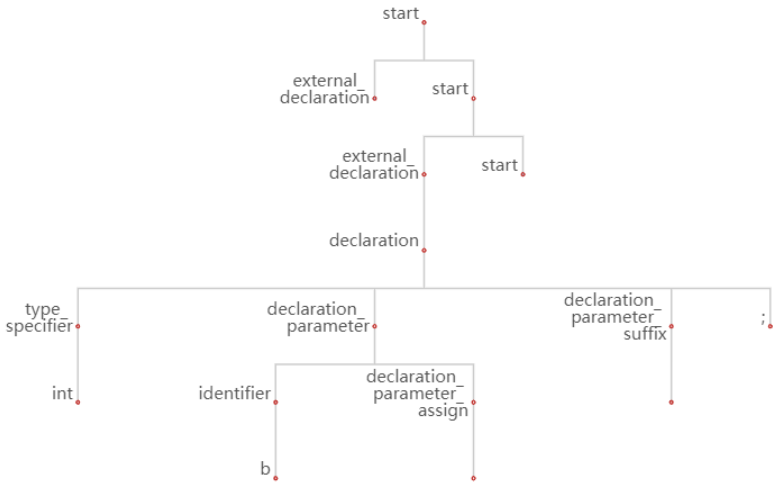
说到集合，python 对集合的运算非常符合直觉，可以使用+=、|=等。

stl/依赖

在读入句子，分析并生成语法树时，python 可以通过 pyecharts.charts.Tree 模块，一键生成非常漂亮的语法树。C++没有这种便利。

```
c = (Tree().add(
    "",
    treeData,
        orient="TB",
        initial_tree_depth = -1,
        #collapse_interval=10,
        symbol_size = 3,
        is_roam=True,
        edge_shape="polyline",
        #is_expand_and_collapse=False,
        label_opts=opts.LabelOpts(
            position="top",
            horizontal_align="right",
            vertical_align="middle",
            #rotate='15',
            font_size=15
        )
).set_global_opts(title_opts=opts.TitleOpts(title="语法树")).render("output\语法
树.html"))
```

语法树



第一版的，用 C++写的语法分析器长达四百行。经过两个小时的 debug，发现其对递归和空串的处理存在 bug。此外，在读入产生式时，我们是按照标准格式“S->aB|c”读入。这意味着，符号必须为单个字符（不可为- > |），以及要手动输入终结符集和非终结符集。这需要词法分析器提前读入产生式、将每个符号映射为单字符、语法分析器的中间结果难以使用（如同乱码）、最终结果要再映射回一般字符串、能接受的不同符号数最多不超过 100（126-33+1-3）。C++对

程序的模块化、封装、前期数据结构的确定，都提出了很高的要求（若做不到，则难以进行 debug；并且程序一旦写死，会变得极难修改）。

最终，我们克服了沉没成本，毅然决定重新设计输入输出约定，用 python 重写语法分析器，并配套修改词法分析器。从此开始计算，实现各模块使用的时间为：

词法分析　2H
可读入的产生式（消除左递归）　1H
语法分析 LR(0)　4H
求 first 集　1H
把 LR(0)升级为 LR(1)　3H
Debug　2H
语法树　2H

用 python 写的 LR(0)，除了输入处理、输出处理之外，核心算法仅 100 行（吹一波自己的实现能力）；升级为 LR(1)后，也不到 200 行。得益于 python 数据结构的自由性和丰富的库函数，新的语法分析器不仅可读性更高，数据结构也更加简介清晰。且对于之后绘制语法树部分也有很大的帮助。而还有一个，对本项目而言最大的好处，是便于 debug。

4.2.2 如何 debug？

当程序跑通，小的测试数据通过后，将类 C 文法的产生式输入，总难以一次就生成正确的 Action/Goto 表。我的 debug 流程如下：

首先，当存在冲突时（即 A/G 表中某格已经存在内容，却同时想往里填入规约符时）：

```
if item["isTer"] == True:
    if item["accept"] in goto[i]:
        print("error!", "%d 号项目集族的\t%s\t 符号冲突，冲突的产生式为\t%d\t" % (i,
item["accept"], k), 新项目[k])
```

由此，可以得到发生冲突的产生式为哪一条。随后，保留该产生式，尽量删除其它无关产生式。删一次运行一次；如果依然能复现该冲突，就继续删除其它产生式。直到能触发该错误所需的最少产生式。将非终结符（例如 sstart,start,declaration 等）替换为 A、B、C……；将终结符替换为 a、b、c……。事实上，4.1.2-4.1.5 中所罗列的测试数据，均是由此方法得来，用于解决程序中出现的 bug 的。

以 4.1.5 为例：

产生式：S:A　A:BA|ε　B:bCDc　C:a　D:ε
源程序：bac

首先，手动计算推导：bac→bCc→bCDc→B→BA→A→S

其次，观察 analyze.txt：

```
['b', 'a', 'c', '#'] 的分析栈：
0          #          b a c #
0 1              # b                  a c #
0 1 5            # b a                c #
error
```

随后，观察 goto.txt。

假设 Go(I0,b)=I1　Go(I1,a)=I4，则问题一定出现在 I0、I1、I4 中。

看 Go(I0,b)展示的项目是否正确：

```
Goto(I0,b) = Closure( [28, 29] ) = {28, 29, 47} ={ B->bCDc,b B->bCDc,# C->a,c } = I1
```

若正确，则下推。由此，可以锁定问题状态。

若问题不是出在求取 Go 函数，则根据问题状态，进一步往前查看求闭包过程(closure.txt)。

通过这样的“自下而上”的 debug，我成功解决了所有的 bug；即使仍然没有找出问题，由于已经剔除了无关产生式，变成简单的文法，你也可以“自上而下”，跟着程序一起手动做一遍“这道题”，总能成功找到并解决问题。

言而总之，在过程输出文件足够完备的前提下，只要能成功“抽象”出产生式，一定能通过上述方法解决问题。

5 总结与收获

经过这次大作业，我们巩固了第四第五章学习的所有知识。经过编写程序，和大量的数据测试，有一些一直以来埋藏着的，理解上的误区（比如 first(βa)）得以发现和纠正。程序跑通的那一刻，我感觉

“Nobody knows LR(1) better than me.”

此外，我们锻炼了代码实现能力。本作业是我第一次使用 python 编写复杂的数据结构和算法。从前，因为 python 慢，“看不起”python；然而现在我意识到，python 的简洁，有时更能让我把思考集中在优化算法本身的时间复杂度。例如上面提到的：如何快速求单个项目的闭包？为什么要提前求出这个？如何找出所有可以导出空字的非终结符？为什么要专门存储这个？

本程序从输入类 C 文法，计算，输出所有过程数据，共用 500ms；到生成语法树网页，总计 800ms。

当算法足够优秀，面对大型数据，没有任何编译器能弥补一个次方的时间复杂度差距。

6 参考文献

词法分析器

只参考了编译原理教材，自己写的。

语法分析器 LR0

只参考了编译原理教材，自己写的。

语法分析器 LR1

只参考了编译原理教材，自己写的。

语法树生成：

[pyecharts 学习笔记]——系统配置项（LabelOpts 标签配置项
https://blog.csdn.net/qq_42374697/article/details/105694022

pyecharts 自适应,跟随屏幕大小变化大屏显示
https://blog.csdn.net/weixin_42262769/article/details/118192412

「Python 数据可视化」使用 Pyecharts 制作 Tree（树图）详解
https://zhuanlan.zhihu.com/p/365906408