同济大学计算机科学与技术系

编译原理课程综合实验二

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　2021.12.23

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，供语义分析器使用的文件位于 intermediate 文件夹中。

由语义分析器生成的，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/语义分析器的 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

和语法分析器不同，语义分析器需要对不同的规约做出不同的动作。因此，语义分析器不允许自定义输入文法。

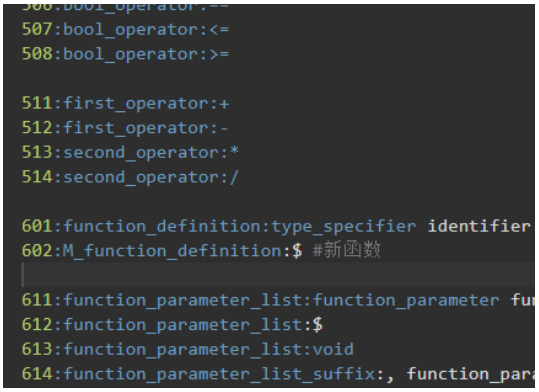
我们的文法规则的输入格式如下：

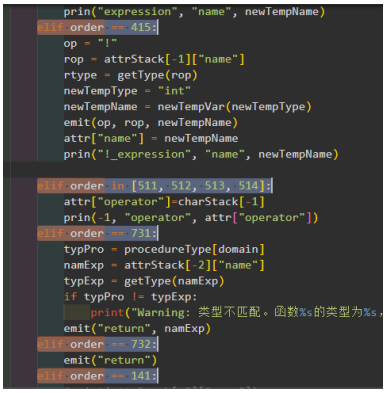
1. 一行只能有一个产生式；对于存在多种可能性的产生式如 S->a|b|c，需要分成若干个不同的产生式分别输入。
2. 产生式分为三段：标号、左部、右部。他们之间用冒号分隔。
3. 产生式右部不同符号以空格分隔，右部为空可以加入$
4. 第一行产生式左部必须为初始符号。初始符号在整个产生式集合中只能出现这一次。

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。

对不同产生式，仅仅用其输入所在的行数作为索引是很不健壮的。因为在编写过程中，可能会需要添加（空转移）产生式来完成一些语义动作。一旦中间插入新的产生式，则后续产生式的行号全部变化。

因此，我们在每条产生式左侧人为添加了序号，方便后续语义动作的索引。这样做还有一个好处，是将产生式人工分类了，变得更加清晰。





左侧为产生式，右侧为编写时的索引方式。

以下是第一次大作业（语法分析器）的产生式：

```
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:$
```

以下是第二次大作业（语义分析器）的产生式：

```
111:declaration:type_specifier declaration_parameter declaration_parameter_suffix ;
112:declaration_parameter:identifier M_declaration_parameter declaration_parameter_assign # .type = [ntop-2].type (其实可以不做这步)
113:declaration_parameter_assign:= expression  # dict[[top-4/-3].name].value=[top-1].name（这个name是个数值），判断是否和type冲突，冲突的话：能强转报warning，不能强转报error! emit([-4].name := [-1].name)
114:declaration_parameter_assign:$ # 无动作
115:declaration_parameter_suffix:, M_declaration_parameter_suffix declaration_parameter declaration_parameter_suffix  # 无动作
116:declaration_parameter_suffix:$ # 无动作
117:M_declaration_parameter:$ # .name=[top-1].name dic[变量名(.name),当前作用域] = {type=[top-2].type is_temp=False}
118:M_declaration_parameter_suffix:$ # .type = [top-2].type
```

我们在着手编写前，会先通过分析语法分析器生成的语法树，来确定语义分析器需要做的动作，标注在产生式右侧，方便编写。

针对大作业，我们最终使用的语法规则示例如下：

```
001:sstart:start
002:start:external_declaration start
003:start:$
101:external_declaration:declaration
102:external_declaration:function_definition
111:declaration:type_specifier declaration_parameter declaration_parameter_suffix ;
112:declaration_parameter:identifier M_declaration_parameter declaration_parameter_assign
113:declaration_parameter_assign:= expression
114:declaration_parameter_assign:$
115:declaration_parameter_suffix:,    M_declaration_parameter_suffix    declaration_parameter
declaration_parameter_suffix
116:declaration_parameter_suffix:$
117:M_declaration_parameter:$
118:M_declaration_parameter_suffix:$
……
```

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>}

同时，出于对后续语义分析实现的必然要求，我们对之前的语法规则进行了简化和修改，使得归约过程中的语义动作能够更加清晰、简洁地执行。

因此，我们更换了语法规则。最终使用的规则如上。

词法分析输出

由词法分析器生成，供语义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，供语义分析器读入的代码。
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。

语义分析输出

由语义分析器生成的，供查阅的文件位于 output 文件夹中。

本语义分析器由之前大作业的语法分析器升级得到，具体实现过程就是针对每一条产生式附加相应的语义动作，从而在归约语法树的过程中完成语义分析的任务。

本语义分析器（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/归约分析的过程（分析栈）。
语法树.html	根据每一步分析结果产生的语法树。
var.txt	语义分析过程中产生的所有变量表。
emit.txt	语义分析得到的中间代码（四元式）。

最后四个文件是真正的输出，其它文件主要供 debug 使用。

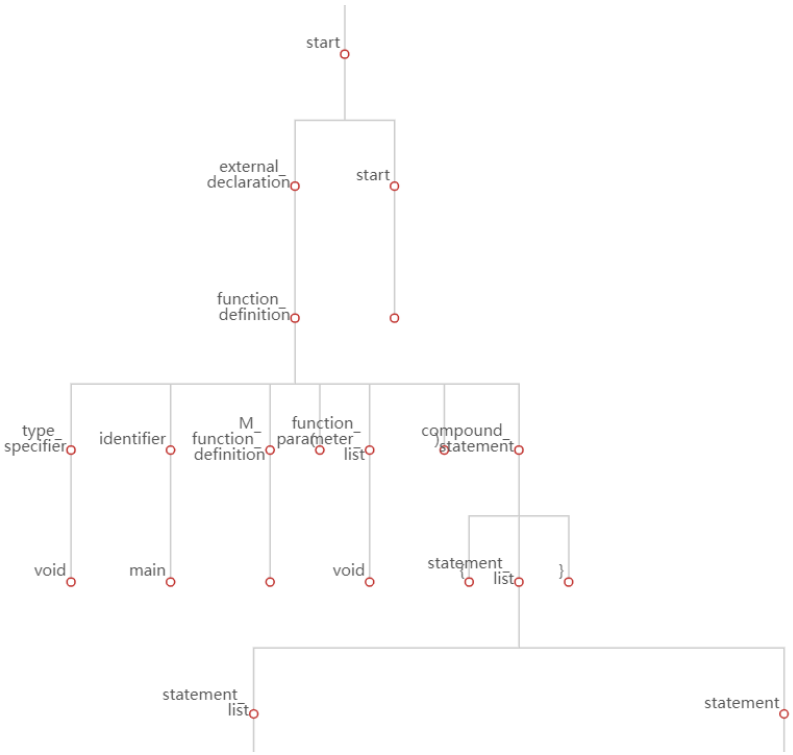
输出文件展示如下：

analyze.txt:

```
用724号产生式规约: assignment_expression → identifier assignment_operator expression
identifier.name=i
expression.value=$7
expression.type=int
(i,program).value=$7
0 2 2 3 10 13 18 35 75 112 133 146 164 195 211 221 224 148 165 145       # external_d

用726号产生式规约: assignment_expression_list_suffix →
assignment_expression_list.name=[]
0 2 2 3 10 13 18 35 75 112 133 146 164 195 211 221 224 148 165 145 162       # extern

用722号产生式规约: assignment_expression_list → assignment_expression assignment_expre
assignment_expression_list.name=['i']
0 2 2 3 10 13 18 35 75 112 133 146 164 195 211 221 224 148 165 157       # external_d
0 2 2 3 10 13 18 35 75 112 133 146 164 195 211 221 224 148 165 157 193       # extern

用721号产生式规约: expression_statement → assignment_expression_list ;
0 2 2 3 10 13 18 35 75 112 133 146 164 195 211 221 224 148 165 155       # external_d

用713号产生式规约: statement → expression_statement
0 2 2 3 10 13 18 35 75 112 133 146 164 195 211 221 224 148 165 152       # external_d

用711号产生式规约: statement_list → statement_list statement
0 2 2 3 10 13 18 35 75 112 133 146 164 195 211 221 224 148 165       # external_decla
0 2 2 3 10 13 18 35 75 112 133 146 164 195 211 221 224 148 165 196       # external_d

用701号产生式规约: compound_statement → { statement_list }
0 2 2 3 10 13 18 35 75 112 133 146 164 195 211 221 224 147       # external_declarati

用717号产生式规约: statement → compound_statement
0 2 2 3 10 13 18 35 75 112 133 146 164 195 211 221 224 238       # external_declarati

用743号产生式规约: iteration_statement → while N_iteration_statement ( constant_expres
回填.产生式序号=12
回填.值=16
0 2 2 3 10 13 18 35 75 112 133 158       # external_declaration external_declaration
```

每行为符号栈、状态栈和待输入栈。若产生规约，则输出规约所用产生式。若有规约时需要执行的相应的语义动作，也一并输出。是语义分析器部分最详细的输出信息。

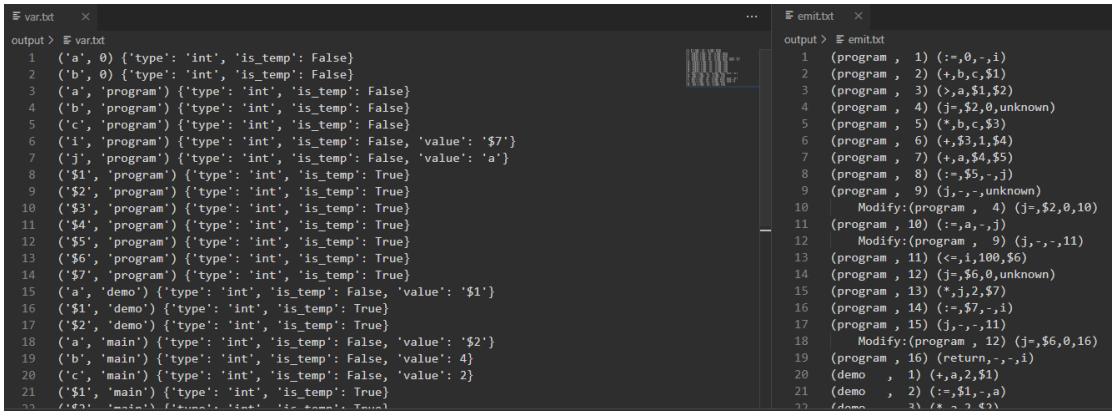
语法树.html:



更为清晰直观地展示了规约过程。(是第一次大作业的产物。)在 Debug 时，一般使用语法树快速定位问题，其次使用 analyze.txt 查看详细信息。

使用 python 的 pyecharts.charts.Tree 模块实现。支持拖动和缩放。

var.txt 和 emit.txt:



分别是所有产生的变量和四元式。其中四元式修改动作没有覆盖原四元式，而是添加了一个“Modify”修改标记，便于 Debug 观察。

其余文件输出内容见 4.1

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

语法语义分析器.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。

语义分析部分，我们选择直接嵌入在语法分析过程中，一次分析直接得到结果，并不分离。语义分析输出各函数内的变量表和四元式。此外还会对代码进行检测，给出一些初步的 warning 和 error。

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	见语义分析输出

1.3 测试数据

输出见 4.1

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

```
 001:sstart:start
 002:start:external_declaration start
 003:start:$

 101:external_declaration:declaration
 102:external_declaration:function_definition

 111:declaration:type_specifier declaration_parameter declaration_parameter_suffix ;
 112:declaration_parameter:identifier M_declaration_parameter declaration_parameter_assign
 113:declaration_parameter_assign:= expression
 114:declaration_parameter_assign:$
 115:declaration_parameter_suffix:,    M_declaration_parameter_suffix    declaration_parameter
 declaration_parameter_suffix
 116:declaration_parameter_suffix:$
 117:M_declaration_parameter:$
 118:M_declaration_parameter_suffix:$

 121:primary_expression:identifier
 122:primary_expression:number
 123:primary_expression:( expression )
```

```
131:expression:function_expression
132:expression:constant_expression

141:function_expression:identifier ( expression_list )

151:expression_list:expression expression_list_suffix
152:expression_list:$
153:expression_list_suffix:, expression expression_list_suffix
154:expression_list_suffix:$

201:assignment_operator:=
202:assignment_operator:+=
203:assignment_operator:-=
204:assignment_operator:*=
205:assignment_operator:/=
206:assignment_operator:%=
207:assignment_operator:^=
208:assignment_operator:&=
209:assignment_operator:|=

301:type_specifier:int
303:type_specifier:float
304:type_specifier:void

401:constant_expression:or_bool_expression
402:or_bool_expression:or_bool_expression or_operator and_bool_expression
403:or_bool_expression:and_bool_expression
404:and_bool_expression:and_bool_expression and_operator single_bool_expression
405:and_bool_expression:single_bool_expression
406:single_bool_expression:single_bool_expression bool_operator first_expression
407:single_bool_expression:first_expression

411:first_expression:first_expression first_operator second_expression
412:first_expression:second_expression
413:second_expression:second_expression second_operator primary_expression
414:second_expression:third_expression
415:third_expression:! primary_expression
416:third_expression:primary_expression

501:or_operator:||
502:and_operator:&&
503:bool_operator:<
504:bool_operator:>
505:bool_operator:!=
506:bool_operator:==
507:bool_operator:<=
508:bool_operator:>=

511:first_operator:+
512:first_operator:-
513:second_operator:*
514:second_operator:/

601:function_definition:type_specifier           identifier           M_function_definition
( function_parameter_list ) compound_statement
602:M_function_definition:$

611:function_parameter_list:function_parameter function_parameter_list_suffix
612:function_parameter_list:$
613:function_parameter_list:void
```

```
614:function_parameter_list_suffix:, function_parameter function_parameter_list_suffix
615:function_parameter_list_suffix:$
616:function_parameter_list_suffix:void
617:function_parameter:type_specifier identifier

701:compound_statement:{ statement_list }

711:statement_list:statement_list statement
712:statement_list:$
713:statement:expression_statement
714:statement:jump_statement
715:statement:selection_statement
716:statement:iteration_statement
717:statement:compound_statement
718:statement:declaration

721:expression_statement:assignment_expression_list ;
722:assignment_expression_list:assignment_expression assignment_expression_list_suffix
723:assignment_expression_list:$
724:assignment_expression:identifier assignment_operator expression
725:assignment_expression_list_suffix:,                                    assignment_expression
assignment_expression_list_suffix
726:assignment_expression_list_suffix:$

731:jump_statement:return expression ;
732:jump_statement:return ;

741:selection_statement:if ( constant_expression ) M_selection_statement statement else
N_selection_statement statement
742:selection_statement:if ( constant_expression ) M_selection_statement statement
743:iteration_statement:while     N_iteration_statement     (     constant_expression     )
M_selection_statement statement
751:M_selection_statement:$
752:N_selection_statement:$
753:N_iteration_statement:$
```

*输出结果见 4.1.1。

2 概要设计

2.1 任务分解及分工

任务分解：见 1.2。

分工：

贾仁军：产生式、语义动作分析、报告编写。

陈泓仰：语法分析器编写、语义分析器编写、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]
```

语义分析相关

在以上语法分析器的基础上，语义分析器额外使用的数据结构：

```
funStack = {} # funStack[过程名]= {code:四元式组, var=[接受的参数], ret_type="int"}
procedureType = {} # 每个过程的类型
domain = 0 # 一个全局变量记录当前作用域，初始为 int(0)，为全局；否则为过程名（string）
attrStack = [] # 每个栈里的符号的属性（不同类型的符号记录的属性不同），语义分析用
```

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/归约分析。
在每一步移进/归约的过程中，依照相应的语义规则分别生成符号表及四元式。
根据每一步分析结果产生语法树，输出

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

语义分析器

如果当前动作是移进，且移进的是 identifier 或 number：
　　输出移进消息

attrStack[-1]["name"]=变量名(string)或数值(double)
*为什么按 double 记录见 4.2
如果当前动作是规约：
```
print("用%d 号产生式归约:"%(item["order"]), item["left"], "→", " ".join(item["right"]))
```
attr = {} # 记录将要产生的规约节点的属性。规约完成后丢入属性栈
*attr 具体设计见 4.2
根据不同的产生式编号，做不同的语义动作。分为规约前执行的，和规约后执行的动作。

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

词法分析器和语法分析器部分见大作业一报告。不再重复。

本次语义分析器是在语法分析器的基础上完成的。

语义分析过程和构造语法树过程本质上是在同一个过程中完成。

在分析句子过程中对不同的产生式采用不同的操作。

新开了变量栈、函数栈、作用域、属性栈、过程类型栈用于存储结点信息。

```
def newTempVar(typ):
```
传入需要新生成的临时变量类型，由函数在当前域变量表中生成一个新临时变量，并将生成的变量名返回。
```
    cnt = 0
    if domain in tempVarCnt:
        cnt = tempVarCnt[domain]
    cnt += 1
    tempVarCnt[domain] = cnt
    cnt = "$%d"%cnt
    varStack[cnt, domain] = {"type":typ, "is_temp":True}
    print("新增临时变量：name=%s, 作用域=%s, type=%s, 临时变量
=%s"%(cnt, str(domain), typ, "True"), file=analyzeFile)
    return cnt

def getAttr(nam):
    根据传入的变量获取信息。
    global varStack
```

```
    global domain
    if (nam, domain) in varStack:
        return varStack[nam, domain]
    elif domain != 0 and (nam, 0) in varStack:
        return varStack[nam, 0]
    else:
        print("Error: undefined variable %s"%nam)
        a = [] # try catch 捕捉到错误，程序停止分析
        print(a[2])

def getNumType(num):
    获取数值的类型（int,float）
    使用 python 的 type()来获取，将放回的<class “int”>中的 int 截出来。
    return str(type(num))[8:-2]

def getType(nam):
    根据传入的内容，寻找其类型。
    如果传入的是数字，则使用 getNumType()以确定类型
    如果传入的是变量，则使用 getAttr()以获得变量信息，将其中的 Type 返回

def getDomain(nam):
    根据传入的内容，寻找其作用域。
        如果传入的是 number，返回当前函数域。
        如果传入的是 identifier：
            如果该变量名在当前域存在，返回当前域（string）
            否则，如果在全局存在，返回全局（0）
            否则报错

def emit(*args):
    生成新的四元式。允许传入 1-4 个变量。
    若只传入 1 个，则后面填三个’-’;若传入 2-3 个，则自动中间补齐。
    在当前函数域下加入一条新四元式，并输出到 emit.txt

def prin(num, typ, ss = 0):
    方便格式化输出到 analyze.txt 的函数。
```

分析过程中产生的中间信息输出到 analyze.txt 方便调试查看。

有了以上实用函数后，一个语义动作的代码示例如下：（以 724 号产生式为例）

```
724:assignment_expression:identifier assignment_operator expression
例：i=5.123
```

```
            elif order in [113, 724]:
                attr["name"]=attrStack[-3]["name"] # a
                prin(-3, "name") # identifier.name=i
                attr["value"]=attrStack[-1]["name"] # 5.123 "$1"
                prin(-1, "value") # expression.value=5.123
                attr["type"]=getType(attr["value"]) # float
                prin(-1, "type", attr["type"]) # expression.type=float
                op = "=" # 获取符号
                if order == 724:
                    op = attrStack[-2]["operator"]
                if op == "=":
                    op = ":="
                shouldType = getType(attr["name"]) # 左部变量的类型为 int
                if shouldType != attr["type"]: # 左右类型不同，报 warning
                    print("Warning: 类型不匹配。变量%s 的类型为%s，赋值%s 的类型
为%s"%(attr["name"], shouldType, str(attr["value"]), attr["type"]))
                varStack[attr["name"], getDomain(attr["name"])]["value"] = attr["value"]
                print("(%s,%s).value=%s"%(attr["name"], str(domain), attr["value"]),
file=analyzeFile) # 赋值并输出：(i,program).value=5.123
                emit(op, attr["value"], attr["name"]) # 生成四元式(:=,5.123,-,i)
```

详细产生式设计如下：

编号：117 产生式：M_declaration_parameter:$ 产生式用于：读入新变量（int a） 语义动作：从栈顶获取变量名，记录入变量表中：name=a， 作用域=main, type=int，临时变量=False。 说明：为了支持在申明时赋值，不能在整个(int a = 1;)读进后再将新变量 a 记录入变量表，而必须在等号之前就已经计入，否则无法赋值。为此使用此空产生式。
编号：617 产生式：function_parameter:type_specifier identifier 产生式用于：将函数传参（如 int program（int a））中的(int a)规约为 function_parameter。函数传参是不支持逗号表达式的（如 int a, b, c），必须是(int a, int b)。但仅对这条产生式而言，其语义动作和 117 相同。 语义动作：同 117
编号：118 产生式：M_declaration_parameter_suffix:$ 产生式用于：int a, b, c;中，让 b 正确的获得申明类型 int。

语义动作：attr["type"]=attrStack[-3]["type"]

说明：为了支持在一行中申明多个变量的逗号表达式（int a,b,c;）需要将 int 属性传递到 b 前面。这样无论是 b 还是 a，只要访问其前一个元素（栈顶）既可以获得类型（int）

编号：

121，122，

131，132，

401，403，405，407，

412，414，416，

产生式：primary_expression:identifier 等

产生式用于：将 name 一路传上去（见右）

语义动作：attr["name"]=attrStack[-1]["name"]

说明：有这么多级 expression 是为了支持算符优先级。在语法分析器中，我们没有做算符优先级支持，因为他不影响其它规约的正确性。在语义分析器中，我们补上了这一部分。若表达式中没有某一级的算符，则直接向上传递 name；若有，则根据算符生成四元式和新临时变量（见 402 等）。

编号：113

产生式：declaration_parameter_assign:= expression

产生式用于：int a = 5;中的 a=5 部分

语义动作：

分别获取运算符、运算符左部的变量及其类型、运算符右部的变量（或数值）及其类型。

最后产生赋值四元式(:=,5,-,a)

说明：若右侧类型与左侧不同，报 warning 并试图向左侧进行强制类型转换。

编号：724

产生式：assignment_expression:identifier assignment_operator expression

产生式用于：a *= 5

语义动作：前类似 113。但如果符号不是=而是*= +=之类的话，要先生成临时变量 tempa=a*5，再让 a=tempa。产生的四元式也相应的要变成两条。

编号：602

产生式：M_function_definition:$

产生式用于：int main(){};在进入大括号内的语句前更新当前函数域、记录函数名和返回类型。

语义动作：

```
domain = attrStack[-1]["name"]
procedureType[domain] = attrStack[-2]["type"]
midCode[domain] = []
print("新函数：%s type=%s"%(domain, procedureType[domain]))
```

说明：函数传参，以及函数内的变量和语句等会要求获取当前所在的作用域 domain（此处为 main）。因此，等整个函数规约完后再获取 domain 名称为时已晚，需要添加一个空转移。该空转移在大括号前，及时通知到语义分析器，我们进入了一个新的函数体，以及该函数体的名字、返回类型。

编号：402,404,406,411,413

产生式：or_bool_expression:or_bool_expression or_operator and_bool_expression

first_expression:first_expression first_operator second_expression 等

产生式用于：各级表达式的计算。以 5+a 为例。

语义动作：承接 401 等。若表达式中无该级运算符则上传 name，若有则执行运算。首先获取运算符、左部变量（或数值）及其类型、右部变量（或数值）及其类型。（注意和 724 不同，724 为赋值语句，左侧一定为变量。）

生成一个新临时变量 tempa，指向该表达式的结果。同时生成运算四元式(+,5,a,tempa)。

说明：若左右部类型不同，则报 warning，并且试图向高等级强转（如 int 向 float 强转）。

若表达式为 single(指代<、>=等符号)，and（此处指 C 中的&&，而非按位与&），or（||），则新变量的类型被强制为 bool，而非向高等级强转。

编号：415

产生式：third_expression:! primary_expression

产生式用于：单目运算符 not

语义动作：类似双目运算符：生成新临时变量、生成四元式。

说明：此处 not 指代 C++中的！，而非按位与~，因此也会被强转为 bool。

编号： 511,512,513,514

产生式： first_operator:+|-|*|/

产生式用于： 将符号规约

语义动作： attr["operator"]=charStack[-1]

说明： 和其它产生式不同之处在于，符号是从符号栈拿的，不像变量从状态栈拿的。因为变量会先被规约成 identifier，因此在状态栈中有他的名字。

编号： 731

产生式： jump_statement:return expression ;

产生式用于：函数返回值。return a;

语义动作：产生四元式(return, a, -, -)

说明：判断返回的内容和变量类型是否匹配，若不匹配报 warning

编号： 732

产生式： jump_statement:return ;

产生式用于： 函数返回，但是没有值。

语义动作：产生四元式(return, -, -, -)

编号： 141

产生式： function_expression:identifier (expression_list)

产生式用于： main(int a, int b)

语义动作：for var in attrStack[-2]["name"]:

emit("param", var, '-', '-')

emit("call", nam, newVarNam)

说明：函数调用。将传入的参数依次生成四元式（emit(param,var,-,-)）；随后生成一个新变量（若有返回）用于接收返回值。最后生成四元式(call, 函数名, 返回变量名(若无则为-), -)
编号： 151,153,722,725 产生式： expression_list:expression expression_list_suffix assignment_expression_list_suffix:, assignment_expression assignment_expression_list_suffix 等 产生式用于：逗号表达式规约 语义动作：attr["name"]=[attrStack[-2]["name"]] + attrStack[-1]["name"] 说明：将 expression 逗号表达式依次规约到 expresion_list 的过程中，也将 express 的值依次 append 到 expression_list 中。即：expression 中[“name”]存放的一个值（string 或 number），而 expression_list 中[“name”]存放的是一个 list。这样，141 就可以正确获取到所有值了。
编号： 201~209 产生式： assignment_operator:=|+=|-=|*=|…… 产生式用于： 赋值符号规约 语义动作： attr["operator"] = charStack[-1] 说明：类似 511，从符号栈中获取符号；被用于 724。
编号： 501~508 产生式： or_operator:|| and_operator:&& bool_operator:<|!=|>=…… 产生式用于： 运算符规约 语义动作： 同 201，511。
编号： 123 产生式： primary_expression:(expression) 产生式用于： 括号表达式 语义动作： attr["name"] = attrStack[-2]["name"] 说明：通过各种优先级表达式，运算顺序已经可以被正确解析，因此括号表达式不需要生成四元式，只需要将值（["name"]）从属性栈中上传即可。

编号： 741,742,743

产生式：

```
selection_statement:if ( constant_expression ) M_selection_statement statement else N_selection_statement statement
selection_statement:if ( constant_expression ) M_selection_statement statement
iteration_statement:while N_iteration_statement ( constant_expression ) M_selection_statement statement
```

产生式用于： if 和 while 语句

语义动作： if order == 743: emit("j", attrStack[-6]["pos"] + 1)

```
midCode[domain][attrStack[-2]["pos"] - 1][3] = str(len(midCode[domain]) + 1)
```

说明：if 和 while 规约完毕后的回填。while 语句执行到最后，一定是跳转到 while 头（去进行判断），因此还多了一个 jump（以及其对应的 emit）。

具体见书，对着书写的。

编号： 751

产生式： M_selection_statement:$

产生式用于： if then (else)中 then 处

语义动作：

```
emit("j=", attrStack[-2]["name"], 0, "unknown")
```

```
attr["pos"] = len(midCode[domain])
```

说明：if 后的判断语句为 false 时需要跳转到 else 处(若有)或下一条语句（若无）。生成四元式。该四元式将在 752（if-then-else 语句）或 742（if-then 语句）处回填。具体见书。

编号： 752

产生式： N_selection_statement:$

产生式用于： if then else 中 else 处。

语义动作：

```
emit("j", "unknown")
```

```
midCode[domain][attrStack[-3]["pos"] - 1][3] = str(len(midCode[domain]) + 1)
```

```
attr["pos"] = len(midCode[domain])
```

说明：

1. 回填 if 判断为 false 时（751）应该跳转到的地方；

2. 用于 then 部分语句执行完毕后跳过 else，直接执行下一条语句。生成四元式。该四元式将由 741 回填。
编号： 753 产生式： N_iteration_statement:$ 产生式用于： while 语句的标记 语义动作： attr["pos"] = len(midCode[domain]) 说明：打一个标记，用于指导 while 语句执行完后（743）跳转到的开头处到底是四元式的哪里。
编号： 301~304 产生式： type_specifier:int|float|double|void 产生式用于： 将 type 规约 语义动作： type_specifier = {301:"int", 302:"double", 303:"float", 304:"void"} typ = type_specifier[order] attr["type"] = typ
编号： 601 产生式： function_definition:type_specifier identifier M_function_definition (function_parameter_list) compound_statement 产生式用于： 整个函数规约完后，将当前作用域设为全局。 语义动作： domain = 0

3.2 函数调用关系

无论词法分析器还是语法语义分析器，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，仅在语法语义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）。

具体程序流程见 2.3。

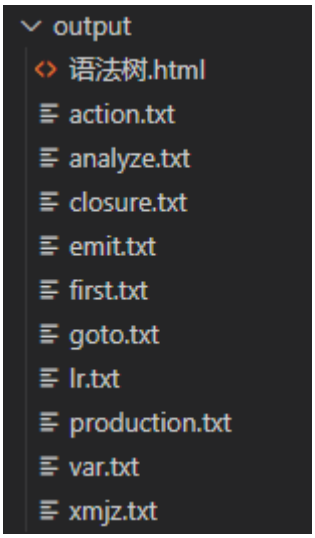
4 调试分析

4.1 测试数据及测试结果

input：见 1.3

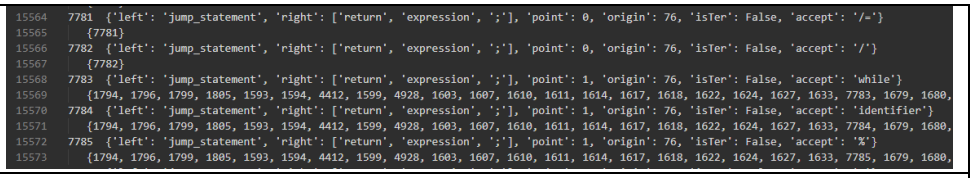
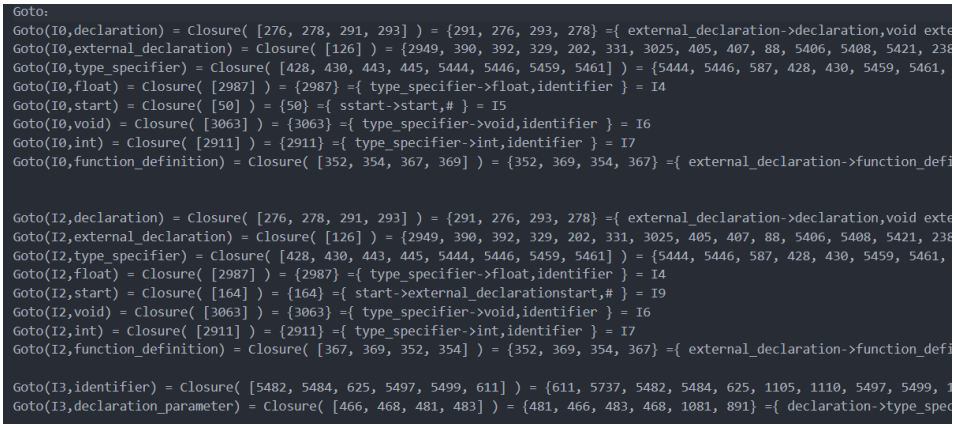
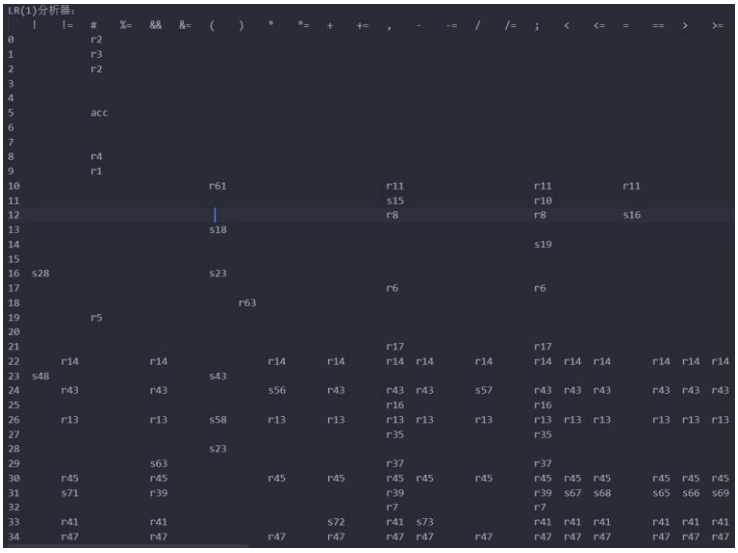
intermediate:见 1.1.3

output / 过程输出



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 238 个项目。 含不同接受符的项目共 38×238=9044 个。 其中，圆点在开头的产生式共 3450 个。
first.txt	共 17 个非终结符的 first 集中含有空字。 ['start','declaration_parameter_assign','declaration_parameter_suffix', 'M_declaration_parameter','M_declaration_parameter_suffix','expression_list', 'expression_list_suffix','M_function_definition','function_parameter_list', 'function_parameter_list_suffix','statement_list','assignment_expression_list', 'assignment_expression_list_suffix','M_selection_statement', 'N_selection_statement','N_iteration_statement', 'sstart']

closure.txt	
xmjz.txt	共有 265 个项目集。0 号项目集有 22 个项目。 （接受符不同算做不同项目）
goto.txt	
lr.txt	

output / 最终输出

var.txt

```
('a', 0) {'type': 'int', 'is_temp': False}
('b', 0) {'type': 'int', 'is_temp': False}
('a', 'program') {'type': 'int', 'is_temp': False}
('b', 'program') {'type': 'int', 'is_temp': False}
('c', 'program') {'type': 'int', 'is_temp': False}
('i', 'program') {'type': 'int', 'is_temp': False, 'value': '$7'}
('j', 'program') {'type': 'int', 'is_temp': False, 'value': 'a'}
('$1', 'program') {'type': 'int', 'is_temp': True}
('$2', 'program') {'type': 'int', 'is_temp': True}
('$3', 'program') {'type': 'int', 'is_temp': True}
```

```
('$4', 'program') {'type': 'int', 'is_temp': True}
('$5', 'program') {'type': 'int', 'is_temp': True}
('$6', 'program') {'type': 'int', 'is_temp': True}
('$7', 'program') {'type': 'int', 'is_temp': True}
('a', 'demo') {'type': 'int', 'is_temp': False, 'value': '$1'}
('$1', 'demo') {'type': 'int', 'is_temp': True}
('$2', 'demo') {'type': 'int', 'is_temp': True}
('a', 'main') {'type': 'int', 'is_temp': False, 'value': '$2'}
('b', 'main') {'type': 'int', 'is_temp': False, 'value': 4}
('c', 'main') {'type': 'int', 'is_temp': False, 'value': 2}
('$1', 'main') {'type': 'int', 'is_temp': True}
('$2', 'main') {'type': 'int', 'is_temp': True}
```

emit.txt

```
(program ,  1) (:=,0,-,i)
(program ,  2) (+,b,c,$1)
(program ,  3) (>,a,$1,$2)
(program ,  4) (j=,$2,0,unknown)
(program ,  5) (*,b,c,$3)
(program ,  6) (+,$3,1,$4)
(program ,  7) (+,a,$4,$5)
(program ,  8) (:=,$5,-,j)
(program ,  9) (j,-,-,unknown)
    Modify:(program ,  4) (j=,$2,0,10)
(program , 10) (:=,a,-,j)
    Modify:(program ,  9) (j,-,-,11)
(program , 11) (<=,i,100,$6)
(program , 12) (j=,$6,0,unknown)
(program , 13) (*,j,2,$7)
(program , 14) (:=,$7,-,i)
(program , 15) (j,-,-,11)
    Modify:(program , 12) (j=,$6,0,16)
(program , 16) (return,-,-,i)
(demo   ,  1) (+,a,2,$1)
(demo   ,  2) (:=,$1,-,a)
(demo   ,  3) (*,a,2,$2)
(demo   ,  4) (return,-,-,$2)
(main   ,  1) (:=,3,-,a)
(main   ,  2) (:=,4,-,b)
(main   ,  3) (:=,2,-,c)
(main   ,  4) (param,c,-,-)
(main   ,  5) (call,demo,-,$1)
(main   ,  6) (param,a,-,-)
(main   ,  7) (param,b,-,-)
(main   ,  8) (param,$1,-,-)
(main   ,  9) (call,program,-,$2)
(main   , 10) (:=,$2,-,a)
(main   , 11) (return,-,-,-)
```

4.2 调试过程存在的问题及解决方法

1. 数值的 type 为什么一律按 double 读入？

数值的 type 在被使用时才判断，因为向上传递的时候 identifier 的 type 是不确定的。而 identifier 和 number 共用一条传递路径。因此，在传递时，我们只需要能区分是数值还是变量即可。在使用时，变量从变量信息处获取类型，而数值另外判断。

2. 为什么需要临时 attr？为什么语义动作要分为规约前执行和规约后执行？

所有动作都可以在规约前执行，但是有些动作写在规约后执行会更为直观。例如：
601:function_definition:type_specifier identifier M_function_definition (function_parameter_list) compound_statement

当产生 601 规约时，说明一个函数执行完毕。函数执行完毕后，需要将当前作用域置为全局：

```
# 归约后执行的
elif order == 601:
    print("%s 函数结束，domain=0"%(domain), file=analyzeFile)
    domain = 0
```

如果直接往栈里写入，那么需要向产生式左部符号记录的属性需要在规约后执行；而规约后，栈里已经不存在产生式右部的符号，属性也相应的被丢弃了。为此，我们需要 attr 变量，临时记录右部需要向左部传递的所有信息。并在产生式规约后，将左部 push 入符号栈后，语义动作也完成后，再将 attr push 入属性栈。

3. 如何让我的文法支持 int a = 5, b, c = 4.321; ？

为了支持在一行中申明多个变量的逗号表达式（int a,b,c;）需要将 int 属性传递到 b 前面。这样无论是 b 还是 a，只要访问其前一个元素（栈顶）既可以获得类型（int）。我们设计了 118 号空转移来完成

为了支持在申明时赋值，不能在整个(int a = 1;)读进后再将新变量 a 记录入变量表，而必须在等号之前就已经计入，否则无法赋值。为此使用此空产生式。我们设计了 117 号空转移来完成。

4. 如何处理 program(a,b,demo(c)) ？

一开始，我们使用了一个全局的 list 来记录值：如果遇到逗号，则把 name 加入 list；如果遇到分号，则把 list 清空。然而这个“取巧”的方法遇到 f1(a,b,f2(c))就出错了：因为没有遇到分号，所以调用 f2 的时候不只传入了 c，而是传入了 a，b，c；获取 f2(c)的值（假设为 d）以后，调用 f1 的时候传入了 a,b,c,d，而实际上应该传入 a,b,d。为此，我们在 151、153、722、725 产生式中解决了该问题。

将 expression 逗号表达式依次规约到 expresion_list 的过程中，也将 express 的值依次 append 到 expression_list 中。即：expression 中[“name”]存放的一个值（string 或 number），而 expression_list 中[“name”]存放的是一个 list。这样，就可以正确获取到所有值了。

此外，经测试，我们的程序不支持直接过程调用。因为函数调用被认为一定会有一个赋值。因此想要直接调用，必须写(void)func();然而我们的程序不支持显式类型转换。

5. 同时支持 3 和 4 这两种逗号表达式，会有文法冲突吗？文法依然是 LR(1)的吗？

(1)为了支持 demo(a,b,c) 我们做了 exp 级别的逗号表达式

(2)为了支持 int a=1, b=2, c=3，我们做了 ari_exp 级别的逗号表达式

(3)并且，我们的 exp 会被规约到 ari_exp

因此，我们的文法支持逗号表达式 d=(a=1,b=2,c=3)，或者是 program(a,b,c)；但不支持 a=(1,2,3) a=1,2,3。因为这需要 exp 允许直接转移到 ari_exp(4)

(1)(2)(3)(4)构成 LR(1)冲突。因此我们只支持(1),(2),(3)。

6. 为什么移进 identifier 和移进 number 都被记录为 attr[“name”]？

因为表达式的值可能是变量也可能是数值。因此在传递时采用相同的标记可以节省大量判断，只用存储类型来分辨（identifier 按 string 存，number 按 float 存，利用了 python 弱变量类型的特性）。当需要运算时，再去判断 attr[“name”]中的是 identifier 还是 number。如果是 identifier，从变量表中获取其类型；如果是 number，再进一步判断是 float 还是 int。这一切都封装进了 getType()函数。在语义分析过程中，我只需要知道 name 和 type 就够了。

7. 我们支持哪些 warning 和 error？

赋值时，若左右部类型不相同，报 warning，并按左部类型强转。

运算时，若左右部类型不相同，报 warning，并按规则强转（例如 int 和 float 运算按 float 存结果）。

变量被使用时检查是否被赋值（即是否有[“value”]记录），没有的话报错，并赋值 0，随后参与运算。

检测未被使用的变量（一个过程块结束后没有有[“value”]记录的局部变量，和整个输入程序结束后未被使用的全局变量）。

8. 关于 bool

在 402、404、406 产生式语义动作中可以看到，若表达式为 single(指代<、>=等符号)，and（此处指 C 中的&&，而非按位与&），or（||），则新变量的类型被强制为 bool，而非向高等级强转。然而实际上，在我们程序中，bool 被记录成了 int，因为我们想避免过多的函数类型，让强转逻辑更加复杂；但是我们确实考虑到了这一点，做了特别的判断。

细心的同学老师可能发现，我们的赋值总是从 primary_expression（指代感叹号，即 not）开始，一路经过加减乘除运算，随后是 and/or/single_exp，最后来到 exp。那是不是我们的赋值总是会被错误的强转成 bool？实际是不会的。因为只有出现 and/or/not 算符，即经过了 402/404/406 产生式，最后 expression 获得的结果才是 bool；而通过 401 等产生式直接上传的值，是不会被强转的。

5 总结与收获

将之前的语法分析器进化为语法语义分析器的过程中，我们认识到在产生式中间添加 M、N 的空转移产生式的重要之处。它能够有效解决语义动作的滞后问题，让语义动作及时地发生在产生式的任意你需要的地方。

在编写语义分析器的时候，我们充分感受到了小组作业中同学间的相辅相成、默契与配合。在分析产生式的语义动作时，一位同学思路中的错误总是能由另一位同学发现。编写代码时，在一旁观察的同学总能即时指出代码中的 bug。如果

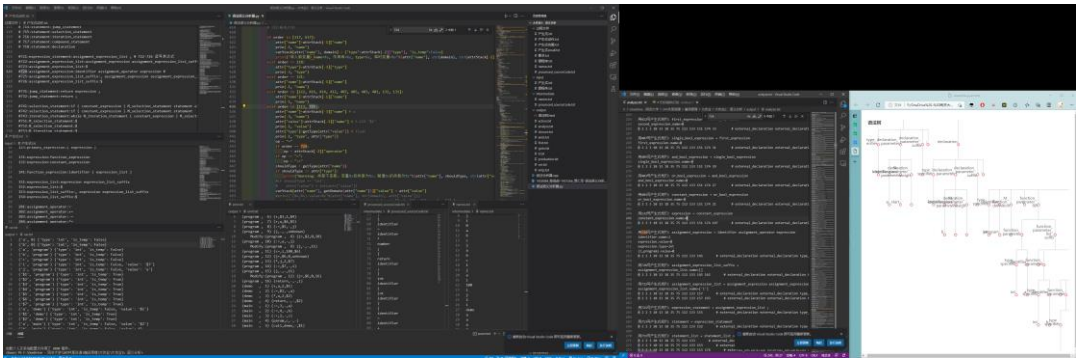
这次作业仅仅由一个人来完成的话，相信我会遇到更多的问题，花出成倍的时间来 debug，并感受到相当的挫败。

实现中，一个好的封装是非常必要的。如果某段代码我认为可能会被复用到，我会留下标记；若在后续编写时发现可以复用，则封装为函数。在 expression 语句中，从属性栈中传入的“name”，无论是数字还是变量，均可通过 getType()获得其类型，通过 getDomain 获得其所在域。此外，在输出四元式和输出详细分析过程中，也做了封装，让代码简洁易懂。

本次作业（语义分析器部分）共耗时约 14 小时。其中报告约 4 小时。语义分析部分核心代码（针对不同产生式的语义动作）共约 160 行（实际上有接近一半还是用于向 analyze.txt 输出详细信息的代码）。计函数封装共约 240 行。

有同学问：你们作业独立自主完成，为了让作业“更完整”，能“充分展示水平”要不要做一个 UI？我我们对“卷 UI、卷报告”没有特别强烈的感受，只希望把自己想要做的事做到最好，而这之中最基本的一件事就是不抄袭。

至于 UI 方面，我们全程使用 VSCode 作为编写和测试的平台，VSCode 可以自定义各程序框占比，一件运行，以及实时更新发生变化的文件，满足了我们的一切需求。



*编写过程窗口截图实例。大屏左侧摆放产生式和变量表，右上摆放代码，右下摆放四元式窗口和输入的代码。小屏左侧摆放详细分析数据，右侧摆放语法树。

强行去做一个比 VSCode 原生体验更差的 UI，好比画蛇添足；而做一个非常强大的 UI，又超出编译原理这门课的范畴。最终我们决定，将精力只放在完成词法、语法和语义分析上。

6 参考文献

无，全是自己写的。