
同济大学计算机科学与技术系

编译原理课程设计

设计说明书



作业项目 LR1 编译器

学号姓名 1950084 陈泓仰

专 业 计算机科学与技术

授课老师 卫志华

日 期 2022 年 5 月 20 日

目录

1 需求分析	4
1.1 输入输出约定	4
源程序输入	4
语法规则输入	4
词法分析器输出	5
语法语义分析器输出	6
目标代码输出	7
1.2 程序功能	7
词法分析器.cpp	7
语法语义分析器.py	8
目标代码生成器.py	8
1.3 测试数据	8
input\源程序.txt	8
input\产生式.txt	8
2 概要设计	9
2.1 任务分解及分工	9
2.2 数据类型定义	10
词法分析相关	10
语法分析相关	10
语义分析相关	10
目标代码生成相关	10
2.3 主程序流程	11
词法分析器	11
语法分析器	11
语义分析器	11
目标代码生成器	11
2.4 模块间的调用关系	11
3 详细设计	12
3.1 主要函数分析及设计	12
词法分析器 / 读入	12
词法分析器 / 过滤	12
词法分析器 / 分解	12
语法分析器 / 读入	13
语法分析器 / 求 First 集	14
语法分析器 / 求单个项目的闭包	14
语法分析器 / 求项目集族和 Go 转移函数	15
语法分析器 / 求 Action/Goto 表	15
语法分析器 / 分析输入的句子	16
语法分析器 / 构造语法树	16
语义分析器 / 数据结构和实用函数	16
语义分析器 / 详细产生式动作设计	18
目标代码生成器 / MIPS	21
目标代码生成器 / 内存存储变量、寄存器交互	22
目标代码生成器 / 数据结构和数据段格式	23
目标代码生成器 / 将四元式翻译为目标代码	24
1. 确定该中间代码的运算类型	24
2. 根据 y 所属类型进行不同操作	25

3. 执行核心指令。.....	26
4. 将运算结果（若有）存入内存。.....	26
目标代码生成器 / 核心指令.....	26
赋值.....	26
单/双目运算.....	26
跳转.....	28
目标代码生成器 / 函数调用.....	29
目标代码生成器 / 待用及活跃信息表.....	30
目标代码生成器 / 其它优化.....	30
3.2 函数调用关系.....	30
4 调试分析.....	31
4.1 测试数据及测试结果.....	31
类 C 文法（含过程调用）.....	31
input: 见 1.3.....	31
intermediate: 见 1.1.3.....	31
output / 过程输出.....	31
output / 最终输出.....	32
output / object_code.asm.....	33
类 C 文法（不含过程调用）.....	33
input / 源程序.txt.....	34
output / object_code.asm.....	34
测试文法强度.....	35
测试是否正确的处理了空产生式.....	35
测试 first 集是否计算正确.....	36
测试计算闭包时 $first(\beta a)$ 是否错误（非常强的数据）.....	36
4.2 调试过程存在的问题及解决方法.....	37
词法&语法分析器开发.....	37
从 C++ 到 python.....	37
如何 debug?.....	38
语义分析器开发.....	39
关于 identifier 和 number.....	39
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?.....	39
如何让我们的文法支持 <code>int a = 5, b, c = 4.321; ?</code>	39
如何处理 <code>program(a,b,demo(c)) ?</code>	40
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?.....	40
为什么移进 identifier 和移进 number 都被记录为 <code>attr["name"]</code> ?.....	40
我们支持哪些 warning 和 error?.....	40
关于 bool.....	41
目标代码生成器开发.....	41
关于 float.....	41
关于强转.....	42
关于 bool.....	42
5 总结与收获.....	42
词法&语法分析器开发.....	42
语义分析器开发.....	43
目标代码生成器开发.....	44
6 参考文献.....	44

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。

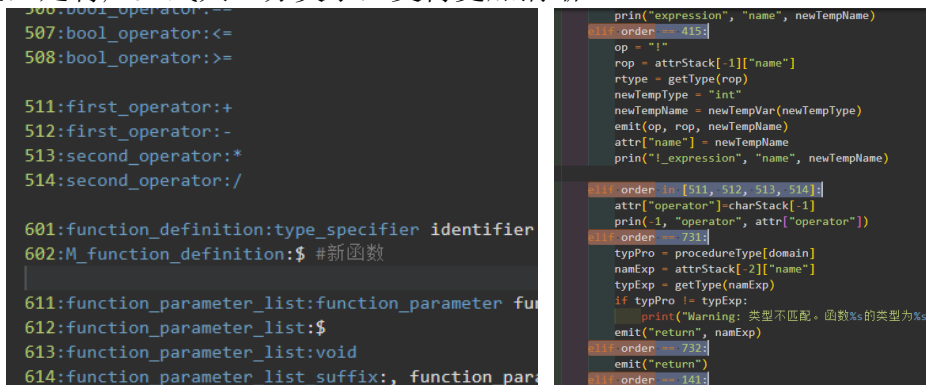
我们的文法规则的输入格式如下：

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 $S \rightarrow a|b|c$ ，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：

```
10 type_specifier:void
11 declaration:type_specifier declaration_parameter declaration_parameter_suffix ;
12 declaration_parameter:identifier declaration_parameter_assign
13 declaration_parameter_assign:= expression
14 declaration_parameter_assign:$
15 declaration_parameter_suffix:, declaration_parameter declaration_parameter_suffix
16 declaration_parameter_suffix:$
```

以下是第二次大作业（语义分析器）的产生式：

```
36 111:declaration:type_specifier declaration_parameter declaration_parameter_suffix ;
37 112:declaration_parameter:identifier # declaration_parameter declaration_parameter_assign # .type = {top-2}.type (其实可以不单这一步)
38 113: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匹配，匹配的话：就继续warning，不能继续就error! emit([-4].name := [-1].name)
39 114:declaration_parameter_assign:$ # 无动作
40 115:declaration_parameter_suffix:, # declaration_parameter_suffix declaration_parameter declaration_parameter_suffix # 无动作
41 116:declaration_parameter_suffix:$ # 无动作
42 117:M_declaration_parameter:$ # .name={top-1}.name dict[{top-1}.name].dict[{top-1}.name] = {type={top-2}.type is temp=false}
43 118: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。

词法分析器输出

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
<code>processed_sourceCode.txt</code>	将输入的源程序处理后，供语义分析器读入的代码。
<code>names.txt</code>	对 <code>processed_sourceCode.txt</code> 文件的补充。

以下是一个例子：

文件	内容
源程序.txt	<code>int program(int a, int b, int c){}</code>
processed_sourceCode.txt	<code>int identifier (int identifier , int identifier , int identifier)</code>

	{ }
names.txt	program a b c

具体设计原因见 [3.1](#)。

语法语义分析器输出

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。
xmjl.txt	从 $[S' \rightarrow S, \#]$ 开始，根据闭包和 first 集求得的项目集族。
goto.txt	从 $[S' \rightarrow S, \#]$ 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：

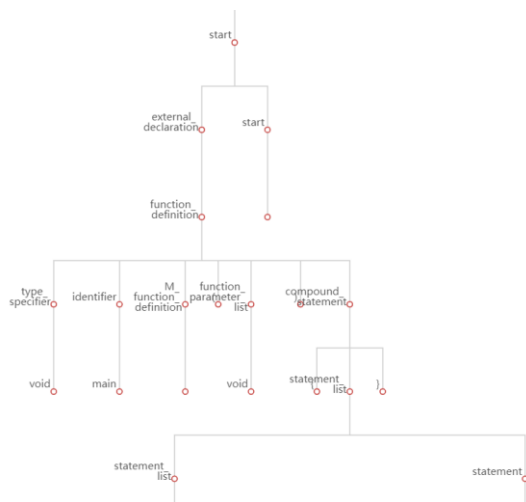
```

837 用724号产生式规约: assignment_expression -> identifier assignment_operator expression
838 identifier.name=i
839 expression.value=$7
840 expression.type=int
841 (1,program).value=$7
842 0 2 2 3 10 13 18 35 75 112 133 146 164 195 211 221 224 148 165 145 # external_declaration
843
844 用726号产生式规约: assignment_expression_list_suffix +
845 assignment_expression_list.name=[]
846 0 2 2 3 10 13 18 35 75 112 133 146 164 195 211 221 224 148 165 145 162 # external_declaration
847
848 用722号产生式规约: assignment_expression_list -> assignment_expression assignment_expression_list
849 assignment_expression_list.name=['i']
850 0 2 2 3 10 13 18 35 75 112 133 146 164 195 211 221 224 148 165 157 # external_declaration
851 0 2 2 3 10 13 18 35 75 112 133 146 164 195 211 221 224 148 165 157 193 # external_declaration
852
853 用721号产生式规约: expression_statement -> assignment_expression_list ;
854 0 2 2 3 10 13 18 35 75 112 133 146 164 195 211 221 224 148 165 155 # external_declaration
855
856 用713号产生式规约: statement -> expression_statement
857 0 2 2 3 10 13 18 35 75 112 133 146 164 195 211 221 224 148 165 152 # external_declaration
858
859 用711号产生式规约: statement_list -> statement_list statement
860 0 2 2 3 10 13 18 35 75 112 133 146 164 195 211 221 224 148 165 # external_declaration
861 0 2 2 3 10 13 18 35 75 112 133 146 164 195 211 221 224 148 165 196 # external_declaration
862
863 用701号产生式规约: compound_statement -> { statement_list }
864 0 2 2 3 10 13 18 35 75 112 133 146 164 195 211 221 224 147 # external_declaration
865
866 用717号产生式规约: statement -> compound_statement
867 0 2 2 3 10 13 18 35 75 112 133 146 164 195 211 221 224 238 # external_declaration
868
869 用743号产生式规约: iteration_statement -> while N_iteration_statement ( constant_expression )
870 回溯:产生式序号-12
871 回溯:值-16
872 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:

var.txt
output > var.txt

```

1 ('a', 0) {'type': 'int', 'is_temp': False}
2 ('b', 0) {'type': 'int', 'is_temp': False}
3 ('a', 'program') {'type': 'int', 'is_temp': False}
4 ('b', 'program') {'type': 'int', 'is_temp': False}
5 ('c', 'program') {'type': 'int', 'is_temp': False}
6 ('i', 'program') {'type': 'int', 'is_temp': False, 'value': '$7'}
7 ('j', 'program') {'type': 'int', 'is_temp': False, 'value': 'a'}
8 ('$1', 'program') {'type': 'int', 'is_temp': True}
9 ('$2', 'program') {'type': 'int', 'is_temp': True}
10 ('$3', 'program') {'type': 'int', 'is_temp': True}
11 ('$4', 'program') {'type': 'int', 'is_temp': True}
12 ('$5', 'program') {'type': 'int', 'is_temp': True}
13 ('$6', 'program') {'type': 'int', 'is_temp': True}
14 ('$7', 'program') {'type': 'int', 'is_temp': True}
15 ('a', 'demo') {'type': 'int', 'is_temp': False, 'value': '$1'}
16 ('$1', 'demo') {'type': 'int', 'is_temp': True}
17 ('$2', 'demo') {'type': 'int', 'is_temp': True}
18 ('a', 'main') {'type': 'int', 'is_temp': False, 'value': '$2'}
19 ('b', 'main') {'type': 'int', 'is_temp': False, 'value': 4}
20 ('c', 'main') {'type': 'int', 'is_temp': False, 'value': 2}
21 ('$1', 'main') {'type': 'int', 'is_temp': True}

```

emit.txt
output > emit.txt

```

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

```

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。

详细文件输出内容见 [4.1.1 类 C 文法（含过程调用）](#)

目标代码输出

由目标代码生成器生成，可以直接放入 MARS 4.5 (MIPS32 汇编模拟器) 运行的 .asm 文件。该文件位于 output 文件夹中。

文件	内容
object_code.asm	可在 MARS 4.5 (MIPS32 汇编模拟器) 直接运行的 .asm 文件

输出内容详见 [类 C 文法（不含过程调用）](#)

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。

程序功能、算法设计、代码实现见 [3.1 / 词法分析器](#)

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。
程序功能、算法设计、代码实现见 [3.1 / 语法分析器/语义分析器](#)

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

目标代码生成器.py

输入变量表和四元式，输出 MIPS 目标汇编代码。
程序功能、算法设计、代码实现见 [3.1 / 目标代码生成器](#)

输入	内容
output\emit.txt	四元式
output\var.txt	变量表
输出	见 目标代码输出

*由于使用了相同的函数，该部分代码也被我放入语法语义分析器.py 中了
**实际上是直接从变量中读取四元式和变量表，并非从文件中读取。

1.3 测试数据

*输出结果见 [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:start: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:$

```

2 概要设计

2.1 任务分解及分工

任务分解：见 [1.2](#)。

申明：词法分析器编写、语法分析器编写、语义动作分析、语义分析器编写、目标代码生成器编写、报告编写，均由本人独立自主完成。产生式部分和语义动作分析部分（在上学期）有小组成员贾仁军参与，共同完成。

2.2 数据类型定义

词法分析相关

```
string wordList[] = {"!=", "(", ")", "*", "+", ";", "-", "/", "+=", "-=", "*=", "/=",  
"%=", "^=", "&=", "|=", ";", "<", "<=", "=", "==", ">", ">=", "else", "if", "int",  
"float", "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 $B \rightarrow \cdot C$ 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:

```
varStack = {} # varStack[name, 作用域] = {type:"int", is_temp=True, value=None}  
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
```

目标代码生成相关

见[目标代码生成器 / 数据结构和数据段格式](#)

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 $[S' \rightarrow S, \#]$ 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，且移进的是变量名 identifier 或数值 number：

输出移进消息

attrStack[-1]["name"]=变量名(string形式记录)或数值(number形式记录,使用时再分辨类型)

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。

目标代码生成器

见[目标代码生成器 / 将四元式翻译为目标代码](#)

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
)
{
}
```

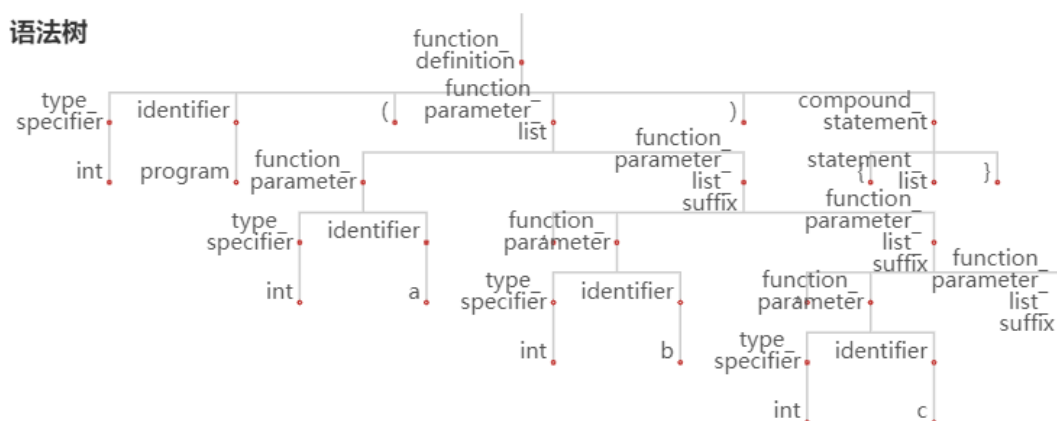
为什么要这样做? 因为这样可以固定住产生式。否则, 根据源程序的不同, 输入语法分析器的产生式也要变化。例如在这段程序, 如果不采用这种替换的方式, 则需要额外加入产生式:

`identifier` \rightarrow `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 $A \rightarrow \epsilon$ 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 $O(\text{产生式中的所有符号})$ 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 $\text{First}(ABC)$ 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 $A \rightarrow BCDE$ ，则 $\text{First}(A) \mid= \text{First}(BCDE)$ 。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:  $[A \rightarrow \alpha \cdot B\beta, a]$ 

```

```

For first( $\beta a$ ) 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} \cup 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一不必要，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' \rightarrow 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] \cup closure[5] \cup 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' \rightarrow 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。此不一一。

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“ $A \rightarrow (B)$ ”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。

语义分析器 / 数据结构和实用函数

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)

```

语义分析器 / 详细产生式动作设计

<p>编号: 117</p> <p>产生式: M_declaration_parameter:\$</p> <p>产生式用于: 读入新变量 (int a)</p> <p>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。</p> <p>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。</p>	
<p>编号: 617</p> <p>产生式: function_parameter:type_specifier identifier</p> <p>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。</p> <p>语义动作: 同 117</p>	
<p>编号: 118</p> <p>产生式: M_declaration_parameter_suffix:\$</p> <p>产生式用于: int a, b, c;中, 让 b 正确的获得申明类型 int。</p> <p>语义动作: attr["type"]=attrStack[-3]["type"]</p> <p>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)</p>	
<p>编号:</p> <p>121, 122, 131, 132, 401, 403, 405, 407, 412, 414, 416,</p> <p>产生式: primary_expression:identifier 等</p> <p>产生式用于: 将 name 一路传上去 (见右)</p> <p>语义动作: attr["name"]=attrStack[-1]["name"]</p> <p>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)。</p>	
<p>编号: 113</p> <p>产生式: declaration_parameter_assign:= expression</p> <p>产生式用于: int a = 5;中的 a=5 部分</p> <p>语义动作:</p> <p>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。</p> <p>最后产生赋值四元式(:=, 5, -, a)</p> <p>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。</p>	
<p>编号: 724</p>	

<p>产生式: <code>assignment_expression:identifier assignment_operator expression</code></p> <p>产生式用于: <code>a *= 5</code></p> <p>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 <code>tempa=a*5</code>,再让 <code>a=tempa</code>。产生的四元式也相应的要变成两条。</p>
<p>编号: 602</p> <p>产生式: <code>M_function_definition:\$</code></p> <p>产生式用于: <code>int main() {}</code>;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。</p> <p>语义动作:</p> <pre>domain = attrStack[-1]["name"] procedureType[domain] = attrStack[-2]["type"] midCode[domain] = [] print("新函数: %s type=%s"%(domain, procedureType[domain]))</pre> <p>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。</p>
<p>编号: 402, 404, 406, 411, 413</p> <p>产生式: <code>or_bool_expression:or_bool_expression or_operator and_bool_expression</code> <code>first_expression:first_expression first_operator second_expression</code> 等</p> <p>产生式用于: 各级表达式的计算。以 <code>5+a</code> 为例。</p> <p>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。)</p> <p>生成一个新临时变量 <code>tempa</code>,指向该表达式的结果。同时生成运算四元式(+, 5, a, <code>tempa</code>)。</p> <p>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。</p>
<p>编号: 415</p> <p>产生式: <code>third_expression:! primary_expression</code></p> <p>产生式用于: 单目运算符 not</p> <p>语义动作: 类似双目运算符:生成新临时变量、生成四元式。</p> <p>说明: 此处 not 指代 C++中的!,而非按位与~,因此也会被强转为 bool。</p>
<p>编号: 511, 512, 513, 514</p> <p>产生式: <code>first_operator:+ - * /</code></p> <p>产生式用于: 将符号规约</p> <p>语义动作: <code>attr["operator"]=charStack[-1]</code></p> <p>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。</p>
<p>编号: 731</p> <p>产生式: <code>jump_statement:return expression ;</code></p> <p>产生式用于: 函数返回值。<code>return a;</code></p> <p>语义动作: 产生四元式(return, a, -, -)</p> <p>说明: 判断返回的内容和变量类型是否匹配,若不匹配报 warning</p>
<p>编号: 732</p> <p>产生式: <code>jump_statement:return ;</code></p> <p>产生式用于: 函数返回,但是没有值。</p> <p>语义动作: 产生四元式(return, -, -, -)</p>

<p>编号： 141</p> <p>产生式： <code>function_expression:identifier (expression_list)</code></p> <p>产生式用于： <code>main(int a, int b)</code></p> <p>语义动作： <code>for var in attrStack[-2]["name"]:</code> <code> emit("param", var, '-', '-')</code> <code> emit("call", nam, newVarNam)</code></p> <p>说明：函数调用。将传入的参数依次生成四元式（<code>emit(param, var, -, -)</code>）； 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（<code>call, 函数名, 返回变量名</code>（若无则为-），-）</p>
<p>编号： 151, 153, 722, 725</p> <p>产生式： <code>expression_list:expression expression_list_suffix</code> <code>assignment_expression_list_suffix:, assignment_expression</code> <code>assignment_expression_list_suffix 等</code></p> <p>产生式用于：逗号表达式规约</p> <p>语义动作： <code>attr["name"]=[attrStack[-2]["name"]] + attrStack[-1]["name"]</code></p> <p>说明：将 <code>expression</code> 逗号表达式依次规约到 <code>expression_list</code> 的过程中，也将 <code>express</code> 的值依次 <code>append</code> 到 <code>expression_list</code> 中。即： <code>expression</code> 中 [“name”] 存放的一个值（string 或 number），而 <code>expression_list</code> 中 [“name”] 存放的是一个 list。这样，141 就可以正确获取到所有值了。</p>
<p>编号： 201~209</p> <p>产生式： <code>assignment_operator:= += -= *=</code></p> <p>产生式用于：赋值符号规约</p> <p>语义动作： <code>attr["operator"] = charStack[-1]</code></p> <p>说明：类似 511，从符号栈中获取符号；被用于 724。</p>
<p>编号： 501~508</p> <p>产生式： <code>or_operator: and_operator:&& bool_operator:< != >=.....</code></p> <p>产生式用于：运算符规约</p> <p>语义动作：同 201，511。</p>
<p>编号： 123</p> <p>产生式： <code>primary_expression:(expression)</code></p> <p>产生式用于：括号表达式</p> <p>语义动作： <code>attr["name"] = attrStack[-2]["name"]</code></p> <p>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。</p>
<p>编号： 741, 742, 743</p> <p>产生式：</p> <p><code>selection_statement:if (constant_expression) M_selection_statement statement else</code> <code>N_selection_statement statement</code> <code>selection_statement:if (constant_expression) M_selection_statement statement</code> <code>iteration_statement:while N_iteration_statement (constant_expression)</code> <code>M_selection_statement statement</code></p> <p>产生式用于：if 和 while 语句</p> <p>语义动作： <code>if order == 743: emit("j", attrStack[-6]["pos"] + 1)</code> <code>midCode[domain][attrStack[-2]["pos"] - 1][3] = str(len(midCode[domain]) + 1)</code></p> <p>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。</p>

<p>编号： 751</p> <p>产生式： <code>M_selection_statement:\$</code></p> <p>产生式用于： <code>if then (else)</code>中 <code>then</code> 处</p> <p>语义动作：</p> <pre>emit("j=", attrStack[-2]["name"], 0, "unknown") attr["pos"] = len(midCode[domain])</pre> <p>说明： <code>if</code> 后的判断语句为 <code>false</code> 时需要跳转到 <code>else</code> 处(若有)或下一条语句(若无)。生成四元式。该四元式将在 752 (<code>if-then-else</code> 语句) 或 742 (<code>if-then</code> 语句) 处回填。具体见书。</p>
<p>编号： 752</p> <p>产生式： <code>N_selection_statement:\$</code></p> <p>产生式用于： <code>if then else</code> 中 <code>else</code> 处。</p> <p>语义动作：</p> <pre>emit("j", "unknown") midCode[domain][attrStack[-3]["pos"] - 1][3] = str(len(midCode[domain]) + 1) attr["pos"] = len(midCode[domain])</pre> <p>说明：</p> <ol style="list-style-type: none"> 1. 回填 <code>if</code> 判断为 <code>false</code> 时 (751) 应该跳转到的地方； 2. 用于 <code>then</code> 部分语句执行完毕后跳过 <code>else</code>，直接执行下一条语句。生成四元式。该四元式将由 741 回填。
<p>编号： 753</p> <p>产生式： <code>N_iteration_statement:\$</code></p> <p>产生式用于： <code>while</code> 语句的标记</p> <p>语义动作： <code>attr["pos"] = len(midCode[domain])</code></p> <p>说明： 打一个标记，用于指导 <code>while</code> 语句执行完后 (743) 跳转到的开头处到底是四元式的哪里。</p>
<p>编号： 301~304</p> <p>产生式： <code>type_specifier:int float double void</code></p> <p>产生式用于： 将 <code>type</code> 规约</p> <p>语义动作：</p> <pre>type_specifier = {301:"int", 302:"double", 303:"float", 304:"void"} typ = type_specifier[order] attr["type"] = typ</pre>
<p>编号： 601</p> <p>产生式： <code>function_definition:type_specifier identifier M_function_definition</code> <code>(function_parameter_list) compound_statement</code></p> <p>产生式用于： 整个函数规约完后，将当前作用域设为全局。</p> <p>语义动作： <code>domain = 0</code></p>

目标代码生成器 / MIPS

首先,我希望写出一份可以运行的基础代码,随后再加上函数调用,以及待用及活跃信息表优化。MIPS 汇编指令集与书上采用的教学用指令集相似,因此我的目标代码为 MIPS ASM。

MIPS 的寄存器用法如下:

REGISTER	NAME	USAGE
\$0	\$zero	常量 0(constant value 0)
\$1	\$at	保留给汇编器(Reserved for assembler)
\$2-\$3	\$v0-\$v1	函数调用返回值(values for results and expression evaluation)

\$4-\$7	\$a0-\$a3	函数调用参数(arguments)
\$8-\$15	\$t0-\$t7	暂时的(或随便用的)
\$16-\$23	\$s0-\$s7	保存的(或如果用, 需要 SAVE/RESTORE 的)(saved)
\$24-\$25	\$t8-\$t9	暂时的(或随便用的)
\$28	\$gp	全局指针(Global Pointer)
\$29	\$sp	堆栈指针(Stack Pointer)
\$30	\$fp	帧指针(Frame Pointer)
\$31	\$ra	返回地址(return address)

在这一部分, 我们只直接使用到其中的\$t0-\$t7, 以及 Coprocessor 1 (FPU)中的\$f1-\$f10。

MIPS 的常用指令如下:

表 4.5 MIPS 汇编语言示例列表

类别	指令名称	汇编举例	含 义	备 注
算术运算	add	add \$s1, \$s2, \$s3	\$s1 = \$s2 + \$s3	三个寄存器操作数
	subtract	sub \$s1, \$s2, \$s3	\$s1 = \$s2 - \$s3	三个寄存器操作数
存储访问	load word	lw \$s1, 100(\$s2)	\$s1 = Memory[\$s2+100]	从内存取一个字到寄存器
	store word	sw \$s1, 100(\$s2)	Memory[\$s2+100] = \$s1	从寄存器存一个字到内存
逻辑运算	and	and \$s1, \$s2, \$s3	\$s1 = \$s2 & \$s3	三个寄存器操作数,按位与
	or	or \$s1, \$s2, \$s3	\$s1 = \$s2 \$s3	三个寄存器操作数,按位或
	nor	nor \$s1, \$s2, \$s3	\$s1 = ~(\$s2 \$s3)	三个寄存器操作数,按位或非
	and immediate	andi \$s1, \$s2, 100	\$s1 = \$s2 & 100	寄存器和常数,按位与
	or immediate	ori \$s1, \$s2, 100	\$s1 = \$s2 100	寄存器和常数,按位或
	shift left logical	sll \$s1, \$s2, 10	\$s1 = \$s2<<10	按常数对寄存器逻辑左移
	shift right logical	srl \$s1, \$s2, 10	\$s1 = \$s2>>10	按常数对寄存器逻辑右移
条件分支	branch on equal	beq \$s1, \$s2, L	if(\$s1 == \$s2) go to L	相等则转移
	branch on not equal	bne \$s1, \$s2, L	if(\$s1 != \$s2) go to L	不相等则转移
	set on less than	slt \$s1, \$s2, \$s3	if(\$s2<\$s3) \$s1=1; else \$s1=0	小于则置寄存器为 1, 否则为 0,用于后续指令判 0
	set on less than immediate	slti \$s1, \$s2, 100	if(\$s2<100) \$s1=1; else \$s1=0	小于常数则置寄存器为 1, 否则为 0,用于后续指令判 0
无条件跳转	jump	j L	go to L	直接跳转至目标地址
	jump register	jr \$ra	go to \$ra	过程返回
	jump and link	jal L	\$ra=PC+4; go to L	过程调用

详细指令如下

[mips 指令与寄存器详解](#) [frozenshore 的博客-CSDN 博客](#) [mips sb](#)

完整指令如下

[Mips 汇编常用的基本操作指令讲解](#) [诸葛大钢铁的博客-CSDN 博客](#) [mips 汇编指令](#)

目标代码生成器 / 内存存储变量、寄存器交互

由于 MIPS 不具有真正意义上的栈, 因此此处我根据生成中间代码时同时得到的程序所使用到的所有变量(包括全局、局部、临时变量), 将他们所占用的所有空间求和, 于 data 段申明。这就是程序所可能用到的所有空间。

举例: 一个函数中有如下语句:

```
int a = 100;
float b = -1.2;
float c = a + b;
```

实际执行的代码为:

```
int a = 100;
float b = -1.2;
float c1 = (float)a + b;
float c = c1;
```

这里使用到的 4 个变量, 总计 16 字节。则在汇编中, 我们申明:


```
.data
s: .space 16
```

之后，我们用 `s[0]` 来索引 `a`，`s[4]` 来索引 `b`，`s[8]` 来索引 `c1`，`s[12]` 来索引 `c`。

对应的汇编指令如下：

```
.data
s: .space 16
fpconst: .float -1.2
; MARS 不支持伪指令 li.s, 无法在 .text 中对寄存器或内存赋小数值，因此显式写成这样。
https://stackoverflow.com/questions/8074055/why-can-i-not-use-li-s-in-mars

.text
; int a = 100;
; (:=, 100, -, a)
addi $t1, $0, 100
sw $t1, s+0

; float b = -1.2;
; (:=, -1.2, -, b)
l.s $f1, fpconst
s.s $f1, s+4

; float c1 = (float)a + b;
; (+, a, b, c1)
l.s $f1, s+0
cvt.s.w $f1, $f1 ; 当识别到操作数类型不同时自动触发该转换
l.s $f2, s+4
add.s $f1, $f1, $f2
s.s $f1, s+8

; float c = c1;
; (:=, c1, -, c)
l.s $f1, s+8
s.s $f1, s+12
```

实际上，没有必要预先为所有变量分配内存。当寄存器不紧张时，很多临时变量在寄存器中被申明，结束其短暂的生命周期后就销毁——这就是编译原理第 11 章的知识内容：待用及活跃信息表。在寄存器不足时，我们将最近不会使用到的变量存入内存，并动态实时维护一个变量是否在寄存器中、在寄存器中的位置、是否在内存中、在内存中的位置。

但首先，我需要的是一个能跑的程序（用来交作业），而非完美的程序。因此在这个阶段，我们预先为每个变量安排好其在内存中的位置。在整个程序运行时将不会改变。在每一条指令执行前，我们从内存对应位置读取出需要的操作数，load 进寄存器中。在寄存器运算出结果后（若有），又将结果立刻写回内存中。可以发现，在绝大多数时候，我们只使用了 2-3 个寄存器。

现在，我们已经知晓如何在内存中保存数据，如何在寄存器中对各类型数据赋值和运算，如何在寄存器和内存间传值。

目标代码生成器 / 数据结构和数据段格式

数据段格式：

```
.data
s: .space <变量数*4>
fpconst_0: .float 0 # 一定有，方便 float 转 bool 用。
fpconst_1: .float 3.6
```

```
fpconst_2: .float -1.2
etc.
```

此时，data 段的数据结构如下：

```
float_const_num = 2 # 当前的浮点常量数
float_const_list = [3.6, -1.2] # 当前已生成的浮点数索引
当目标代码生成过程中，需要一个浮点型变量 y 时：
查询索引，如果在索引中第 k 个位置，返回 f'fpconst_{k+1}'
否则
float_const_num += 1
float_const_list.append(y)
object_code_data.append(f'fpconst_{float_const_num}: .float {float_num}')
return f'fpconst_{float_const_num}'
```

为了能将四元式逐行映射到代码段(.text)，我为四元式设计的数据结构如下：

```
{
  "domain1": [ # 全局(int(0)) 或当前四元式所属函数(string)
    {
      "form": [op1, op2, op3, op4] op 可以为变量、常量、保留字、None,
      "typ": 该四元式运算时采用的类型 (int / float / bool),
      "show_label": 是否要显示 label。默认 False。所有四元式生成后遍历。若此指令为函数中的
      首条指令，或可能为跳转目的地时，设为 True。若为 True，则在生成目标代码时加入 f'{domain}_{index}:'
    },
    {}, {}, ....., {}
  ],
  "domain2": [
    {第 1 条四元式}, {第 2 条四元式}, ...
  ]
}
```

当 debug_object_code = True 时，在每条四元式生成的目标代码前加上注释：

```
f'# ({domain} , {index:2d}) ({str(form)[1:-1]}) {typ}\n'
```

目标代码生成器 / 将四元式翻译为目标代码

为了方便说明，约定：

```
some_hash = f'{当前四元式所属函数}_{当前四元式是所处函数中的第几条指令}' # 例: "main_1"
```

以准备运算数 y 为例：假设 y 的内存位置 s[k]（若为变量），将要载入寄存器 \$s，对应的数字为 x。x 取值 1-8。每条指令中，每个变量对应的 x 必须唯一，因为一个变量可能同时占用 \$tx 和 \$fx。

1. 确定该中间代码的运算类型

运算类型可能为 int/float/bool。不同会导致目标代码指令不同，以及预处理方式不同。若变量类型为 int，则 s=f't{x}'；若为 float，则 s=f'f{x}'。

假设四元式为 (op, y, z, x)；x 的类型为 x_type，y 为 y_type，z 为 z_type，则：

op	运算类型 (formula_type)	运算结果类型 (x_actual_type)
:=	x_type	x_type
+ - *	int if x_type==y_type==int else float	formula_type
< > <= >= != ==	int if x_type==y_type==int else float	bool

/	float	float
j=	bool	/
^ &	int(若 x_type 或 y_type 为 float, 报错)	bool
&& !	bool	bool

运算类型分为 int, float, bool 三类。因此, 当 $x_type/y_type \neq formula_type$ 时, 需向 $formula_type$ 执行强转后再执行后续指令。(详细指令见第 2 点)

而运算结果类型只分 int/bool, float 两类, 因为 bool 实际上也是按 int 存入内存的。因此, 仅当 $x_actual_type == float$ and $x_type == int$ (或相反) 时, 需向 x_type 执行强转后再存入内存。(详细指令见第 4 点)

2. 根据 y 所属类型进行不同操作

2-1. 若 y 为 int 型常数:

addi $\{s\}$, $\$0$, $\{y\}$

2-2. 若 y 为 float 型常数:

根据上一节的方法, 调用函数, 返回一个 label

l.s $\{s\}$, {label}

2-3. 若 y 为 int 型变量:

ld $\{s\}$, $s+\{k\}$

2-4. 若 y 为 float 型变量:

l.s $\{s\}$, $s+\{k\}$

2-5. 若 y 类型为 float, 而运算类型为 int:

cvt.w.s $\$f\{x\}$, $\$f\{x\}$

mfcl, $\$t\{x\}$, $\$f\{x\}$

2-6. 若 y 类型为 float, 而运算类型为 bool:

l.s $\$f10$, fpconst_0

c.eq.s $\$f\{x\}$, $\$f10$ # =0 则 flag 为 1

bclt {some_hash}_1 # flag 为 1 则跳转

addi $\$t\{x\}$, $\$0$, 1

b {some_hash}_2

{some_hash}_1:

addi $\$t\{x\}$, $\$0$, 0

{some_hash}_2:

2-7. 若 y 类型为 int, 而运算类型为 float:

mtcl, $\$t\{x\}$, $\$f\{x\}$

cvt.s.w $\$f\{x\}$, $\$f\{x\}$

2-8. 若 y 类型为 int, 而运算类型为 bool:

sne $\$t\{x\}$, $\$t\{x\}$, 0

3. 执行核心指令。

针对不同四元式，需要执行的不同核心指令见下节。

4. 将运算结果（若有）存入内存。

若运算结果和目标变量类型不一致，使用 2-7 或 2-8 向目标类型强转。

假设（转换后的）运算结果存于寄存器 x1，目标变量对应内存位置为 k1：

sw \$t{x1}, s+{k1}（目标变量为 int）

s.s \$f{x1}, s+{k1}（目标变量为 float）

目标代码生成器 / 核心指令

（填入上节的第 3 部分）

其中 x, y, z 应被替换为该变量当前所属的寄存器 \$s(\$tx/\$fx)。

赋值

语义动作中生成的四元式： (:=, y, -, x)

说明：赋值 $x = y$

对应核心指令：

move x, y # 当四元式为 int 型。

mov.s x, y # 当四元式为 float 型，下同。

单/双目运算

语义动作中生成的四元式： (op, y, z, x)

说明：双目运算 $x = y \text{ op } z$ （单目运算 $x = \text{op } y$ ）

op == +

add x, y, z

add.s x, y, z

op == -

sub x, y, z

sub.s x, y, z

op == *

mul x, y, z

mul.s x, y, z

op == /

div x, y, z

div.s x, y, z

op == ^ # 按位异或

xor x, y, z # 只支持 int 型，float 报错，下同。

op == & # 按位与

and x, y, z

op == | # 按位或

or x, y, z

op == || # formula_type == bool, 下同。

or \$tx, \$ty, \$tz # 根据 2-6 和 2-8，无论运算数原始类型，其 bool 值应已存入 \$tx。

op == &&

and \$tx, \$ty, \$tz

op == <

int: # 当四元式为 int 型。

slt x, y, z

float: # 当四元式为 float 型, 下同。

c.lt.s y, z # f1<f2 则 flag 为 1

bclt {some_hash}_1 # flag 为 1 则跳转

addi x, \$0, 0

b {some_hash}_2

{some_hash}_1:

addi x, \$0, 1

{some_hash}_2:

op == >

int:

sgt x, y, z

float

c.le.s y, z # f1<=f2 则 flag 为 1

bclt {some_hash}_1 # flag 为 1 则跳转

addi x, \$0, 1

b {some_hash}_2

{some_hash}_1:

addi x, \$0, 0

{some_hash}_2:

op == <=

int:

sle x, y, z

float

c.le.s y, z # f1<=f2 则 flag 为 1

bclt {some_hash}_1 # flag 为 1 则跳转

addi x, \$0, 0

b {some_hash}_2

{some_hash}_1:

addi x, \$0, 1

{some_hash}_2:

op == >=

int:

sge x, y, z

float:

```

c.lt.s y, z # f1<f2 则 flag 为 1
bclt {some_hash}_1 # flag 为 1 则跳转
addi x, $0, 1
b {some_hash}_2
{some_hash}_1:
addi x, $0, 0
{some_hash}_2:

```

```

op == !=
int:
sne x, y, z

```

```

float:
c.eq.s y, z # f1==f2 则 flag 为 1
bclt {some_hash}_1 # flag 为 1 则跳转
addi x, $0, 1
b {some_hash}_2
{some_hash}_1:
addi x, $0, 0
{some_hash}_2:

```

```

op == ==
int:
seq x, y, z

```

```

float:
c.eq.s y, z # f1==f2 则 flag 为 1
bclt {some_hash}_1 # flag 为 1 则跳转
addi x, $0, 0
b {some_hash}_2
{some_hash}_1:
addi x, $0, 1
{some_hash}_2:

```

```

op == !
语义动作中生成的四元式: (!, y, -, x)
说明: x = !y # y 先转为 bool
对应目标代码:
xor $tx, $ty, 1

```

跳转

```

语义动作中生成的四元式: (j=, y, z, x)
说明: 当 y==z 时, 跳转到当前函数第 x 条语句。
对应目标代码: beq y, z, {当前四元式所属函数}_{x}

```

```

语义动作中生成的四元式: (j, -, -, x)

```

说明：跳转到当前函数第 x 条语句。

对应目标代码：b {当前四元式所属函数}_{x}

目标代码生成器 / 函数调用

根据编译原理第 9 章的知识，函数调用涉及现场的保存与恢复、参数传入和返回、sp 寄存器维护等。

当第二次(递归地)进入 F 后，DISPLAY 的内容是什么？当时整个运行栈的内容是什么？

19	12	display
18	0	display
17	n	形参单元
16	F	形参单元
15	2	参数个数
14	10	全局 display
13	返回地址	返回地址
第二次 F	12	4 老 sp(动态链)
11	4	display
10	0	display
9	n	形参单元
8	F	形参单元
7	2	参数个数
6	2	全局 display
5	返回地址	返回地址
第一次 F	4	0 老 sp(动态链)
3	K	简单变量
2	0	全局 display
1	返回地址	返回地址
main	0	0 老 sp(动态链)

实际可以不用如此复杂。因为作为“中间人”，我们拥有每个函数需要的参数个数以及各自的类型。我们可以：

1. 将四元式做一些手脚，保存更多信息。将 (param, var, -, -) 改为：(param, operand, index, funcName)
2. 每当开始执行一个新的函数：
 - a) 将传入的参数保存到内存。

```
param_cnt = len(procedure[domain]["param"]) - 1
for i, varName in enumerate(procedure[domain]["param"]):
    object_code += [f'lw $t1, {4*(param_cnt-i)}($sp)',
                   f'sw $t1, s+{getattr(varName)["pos"]}']
```

*因为是直接写到内存，因此不用在意参数类型是 int 还是 float。

- b) 保护返回地址

```
{domain}:
addi $sp, $sp, -4
sw, $ra, 0($sp)
```

语义动作中生成的四元式：(param, operand, index, funcName)

说明：调用函数前将参数从左到右挨个依次传入。

每当收到 param 四元式：

1. 将其从 \$t1 压入 sp 栈。（若其位于 \$f1，则先移入 \$t1）

* \$a 类寄存器只有 \$0~\$3 这 4 个。想要传入超过 4 个变量需要手动维护 sp。然而，MIPS 中没有真正意义上的栈，push 和 pop 需自己维护。

** 报 Warning 应在生成中间代码时处理，而非在生成目标代码时处理

对应目标代码：

将 operand 载入寄存器，向函数接受类型转换。

若函数接受类型为 float（即 operand 此时位于 \$f1），则先转入 \$t1

```
addi $sp, $sp, -4
```

```
sw, $t1, 0($sp) # PUSH 进栈
```

语义动作中生成的四元式：(call, funcName, -, -/varName)

说明：调用函数

对应目标代码：

```
jal funcName
addi $sp, $sp, {4*len(procedure[funcName] param)} # 恢复 sp
若有返回值，且有变量接收：
    move, $t1, $v0 # 从$v0 接收返回值到 t1
    mtcl, $t1, $f1 # 如果返回值类型为 float，转移到 f1
    若返回类型与接收变量类型不同，向接收变量类型强转
    写入接收变量对应内存
```

语义动作中生成的四元式：(return, -, -, -)

说明：从子函数不带值返回(return;)

对应目标代码：

```
lw $ra, 0($sp) # 从栈中恢复 ra (返回读职)
addi $sp, $sp, 4 # 恢复栈指针
jr $ra # 返回
```

语义动作中生成的四元式：(return, operand, -, -)

说明：从子函数带值返回(return operand;)

对应目标代码：

将 operand 载入寄存器，向函数类型转换。
若函数类型为 float（即 operand 此时位于 \$f1），则先转入 \$t1
move \$v0, \$t1 # 将结果转入 \$v0 ()
下同不带值返回 return

目标代码生成器 / 待用及活跃信息表

为了优化代码效率，可以使用编译原理中第 11 章的知识内容，不直接进行 Load 和 Store，除非变量不再使用或无空闲寄存器。

未实现。

目标代码生成器 / 其它优化

应用编译原理第 10 章内容，对程序进行进一步优化。包括：基本块划分、DAG 优化、循环优化（强度削弱、删除归纳变量）等。

未实现。

3.2 函数调用关系

词法分析器、语法语义分析器、目标代码生成器，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[主程序流程](#)。

4 调试分析

4.1 测试数据及测试结果

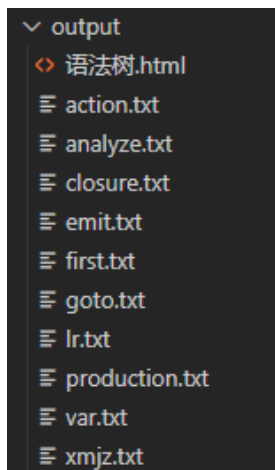
首先是本次作业所最终实现的类C文法的测试，后面是LR(1)针对的各个模块单独设计的小测试。

类C文法（含过程调用）

input: 见 [1.3](#)

intermediate: 见 [1.1.3](#)

output / 过程输出

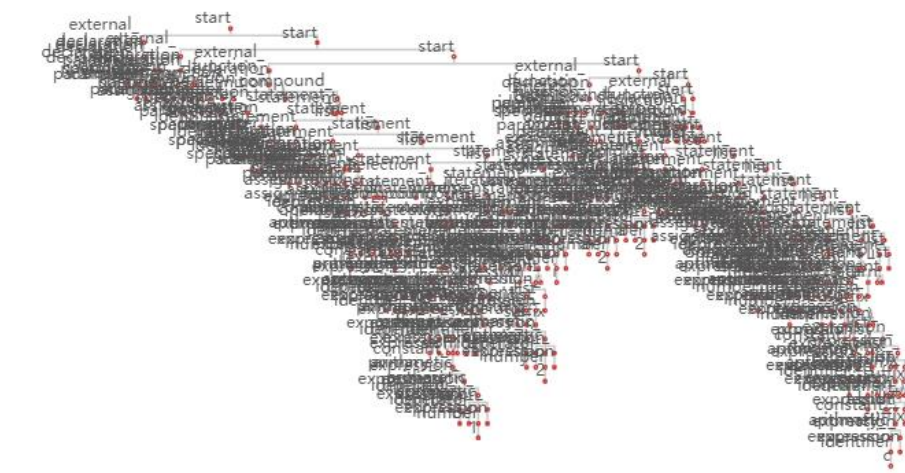
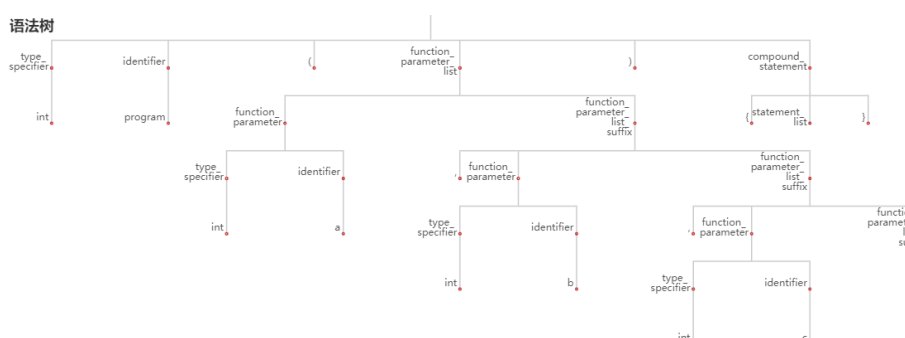


因为文件内容过大，以下列出几个关键指标，或者输出文件大致格式：

文件	内容
production.txt	共 238 个项目。 含不同接受符的项目共 $38 \times 238 = 9044$ 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	<pre>15564 7781 {'left': 'jump_statement', 'right': ['return', 'expression', ';'], 'point': 0, 'origin': 76, 'isTer': False, 'accept': '/='} 15565 {7781} 15566 7782 {'left': 'jump_statement', 'right': ['return', 'expression', ';'], 'point': 0, 'origin': 76, 'isTer': False, 'accept': '/' } 15567 {7782} 15568 7783 {'left': 'jump_statement', 'right': ['return', 'expression', ';'], 'point': 1, 'origin': 76, 'isTer': False, 'accept': 'while'} 15569 {1794, 1796, 1799, 1805, 1593, 1594, 4412, 1599, 4928, 1603, 1607, 1610, 1611, 1614, 1617, 1618, 1622, 1624, 1627, 1633, 7783, 1679, 1680, 15570 {7784 {'left': 'jump_statement', 'right': ['return', 'expression', ';'], 'point': 1, 'origin': 76, 'isTer': False, 'accept': 'identifier'} 15571 {1794, 1796, 1799, 1805, 1593, 1594, 4412, 1599, 4928, 1603, 1607, 1610, 1611, 1614, 1617, 1618, 1622, 1624, 1627, 1633, 7784, 1679, 1680, 15572 {7785 {'left': 'jump_statement', 'right': ['return', 'expression', ';'], 'point': 1, 'origin': 76, 'isTer': False, 'accept': 'X'} 15573 {1794, 1796, 1799, 1805, 1593, 1594, 4412, 1599, 4928, 1603, 1607, 1610, 1611, 1614, 1617, 1618, 1622, 1624, 1627, 1633, 7785, 1679, 1680,</pre>
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	<pre>1 Goto: 2 Goto(I0,external_declaration) = Closure([139]) = {517, 775, 5514, 139, 268, 5518, 272, 5521, 275, 5533, 5 3 Goto(I0,float) = Closure([732]) = {732} = { type_specifier->float,identifier } = I2 4 Goto(I0,start) = Closure([53]) = {53} = { sstart->start,# } = I3 5 Goto(I0,type_specifier) = Closure([5557, 5561, 5564, 5576, 5577, 5581, 913, 917, 920, 932, 933, 937]) = { 6 Goto(I0,char) = Closure([474]) = {474} = { type_specifier->char,identifier } = I5 7 Goto(I0,double) = Closure([646]) = {646} = { type_specifier->double,identifier } = I6 8 Goto(I0,declaration) = Closure([311, 315, 318, 330, 331, 335]) = {331, 311, 330, 315, 318, 335} = { extern 9 Goto(I0,function_definition) = Closure([397, 401, 404, 421, 416, 417]) = {416, 401, 417, 404, 421, 397} = 10 Goto(I0,void) = Closure([818]) = {818} = { type_specifier->void,identifier } = I9 11 Goto(I0,int) = Closure([560]) = {560} = { type_specifier->int,identifier } = I10</pre>

lr.txt	<pre> 1 LR(1)分析器: 2 [- # % & * += +, - - / /- ; < <- = == > >- ^ ~ bre cha con dou els flo 3 0 r2 4 1 r2 5 2 6 3 acc 7 4 8 5 9 6 10 7 r3 11 8 r4 12 9 13 10 14 11 r1 15 12 s14 r13 r15 s16 16 13 s17 17 14 r58 r11 r11 s5 s6 s2 18 15 s23 19 16 20 17 21 18 22 19 23 20 s34 24 21 r59 25 22 r61 s35 26 23 s38 27 24 r16 r16 r16 s45 r16 r16 r16 r16 r16 r16 r16 r16 r16 r16 r16 r16 r16 28 25 r12 29 26 r47 30 27 s56 s54 s58 s47 s49 r50 s52 s60 r50 s55 s53 s48 s46 s59 s50 31 28 r17 r17 r17 r17 r17 r17 r17 r17 r17 r17 r17 r17 r17 r17 r17 r17 </pre>
--------	--

analyze.txt	<p>共 417 行。</p> <pre> 406 0 1 1 1 1 4 12 14 20 34 63 94 139 # external_declaration external_declaration external_declaration external_declaration type_specifier 407 0 1 1 1 1 4 12 14 20 34 63 90 # external_declaration external_declaration external_declaration external_declaration type_specifier 408 0 1 1 1 1 4 12 14 20 34 63 90 126 # external_declaration external_declaration external_declaration external_declaration type_specifier 409 0 1 1 1 1 4 12 14 20 34 62 # external_declaration external_declaration external_declaration external_declaration type_specifier identifier 410 0 1 1 1 1 8 # external_declaration external_declaration external_declaration external_declaration function_definition # 411 0 1 1 1 1 1 # external_declaration external_declaration external_declaration external_declaration external_declaration # 412 0 1 1 1 1 1 11 # external_declaration external_declaration external_declaration external_declaration external_declaration start # 413 0 1 1 1 1 1 11 # external_declaration external_declaration external_declaration external_declaration external_declaration start # 414 0 1 1 1 1 1 # external_declaration external_declaration external_declaration start # 415 0 1 1 1 # external_declaration external_declaration start # 416 0 1 1 # external_declaration start # 417 0 3 # start # 418 Accepted </pre>
-------------	--

语法树.html	<p>支持缩放和拖动，以及子树展开与缩回。</p> <p>语法树</p>  <p>语法树</p> 
----------	--

output / 最终输出	<pre> 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} </pre>
---------------	---


```

('$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,-,-,-)

```

output / object_code.asm

在 output 文件夹下的 object_code.asm。太长了，不贴过来了。

类 C 文法（不含过程调用）

文件截图如下，分别为：源程序（其中内容为本次课设下发的代码的单函数版本）、生成的 asm 文件在 VSCode 中的部分截图、在 MARS 4.5 中的部分截图、运行结果。

可以看到，本程序支持声明时赋值（float c = 2）；支持+=、*=等；支持 float 和 int 两种类型，以及相互之间的运算和类型转换（隐式转换时会输出 Warning 提醒）。当设置为 a<b+c 时，最终结果为 144；当设置为 a>b+c 时，最终结果为 192。

```

if : true, accept : else
Warning: 类型不匹配。变量a的类型为float, 赋值3的类型为int
Warning: 类型不匹配。变量a的类型为float, 赋值2b的类型为int
Warning: 类型不匹配。变量c的类型为float, 赋值2b的类型为int
Warning: 类型不匹配。左操作数c的类型为float, 右操作数2的类型为int
Warning: 类型不匹配。左操作数b的类型为int, 右操作数c的类型为float
Warning: 类型不匹配。左操作数b的类型为int, 右操作数c的类型为float
Warning: 类型不匹配。左操作数4的类型为float, 右操作数1的类型为int
Warning: 类型不匹配。变量j的类型为int, 赋值$6的类型为float
Warning: 类型不匹配。变量j的类型为int, 赋值a的类型为float

```

object_code.asm X

```

output > ~ object_code.asm
1 .data
2 s: .space 44
3 fpconst_0: .float 0 # 一定有, 方便float转bool用。
4 .text
5 b main
6
7 main:
8 main_1:
9 # (main, 1) (':=', 3, None, 'a') type=float
10 addi $t1, $0, 3 # const int 3 载入 $t1
11 # int->float
12 mtc1, $t1, $f1
13 cvt.s.w $f1, $f1
14 mov.s $f3, $f1
15 s.s $f3, s+0 # float a 从 $f3 写回内存
16
17 # (main, 2) (':=', 4, None, 'b') type=int
18 addi $t1, $0, 4 # const int 4 载入 $t1
19 move $t3, $t1
20 sw $t3, s+4 # int b 从 $t3 写回内存
21
22 # (main, 3) (':=', 2, None, 'c') type=float
23 addi $t1, $0, 2 # const int 2 载入 $t1
24 # int->float
25 mtc1, $t1, $f1
26 cvt.s.w $f1, $f1
27 mov.s $f3, $f1
28 s.s $f3, s+8 # float c 从 $f3 写回内存

```

```

1.s $f1, s+0 # float a 从内存载入 $f1
# float->int
cvt.w.s $f1, $f1
mfcl, $t1, $f1
move $t3, $t1
sw $t3, s+16 # int j 从 $t3 写回内存
main_16:
# (main, 16) ('<=', 'j', 100, 'q7') type=int
ld $t1, s+16 # int j 从内存载入 $t1
addi $t2, $0, 100 # const int 100 载入 $t2
sle $t3, $t1, $t2
sw $t3, s+40 # int q7 从 $t3 写回内存
# (main, 17) ('j=', 'q7', 0, 20) type=bool
ld $t1, s+40 # int q7 从内存载入 $t1
sne $t1, $t1, 0 # int->bool
addi $t2, $0, 0 # const int 0 载入 $t2
sne $t2, $t2, 0 # int->bool
beq $t1, $t2, main_20
# (main, 18) ('+', 'j', 2, 'j') type=int
ld $t1, s+16 # int j 从内存载入 $t1
addi $t2, $0, 2 # const int 2 载入 $t2
mul $t1, $t1, $t2
sw $t1, s+16 # int j 从 $t1 写回内存
# (main, 19) ('j', None, None, 16) type=int
b main_16
main_20:
# (main, 20) ('return', None, None, 0) type=int
# 还未实现

```

pc		4194880	pc	4194880
hi		0	hi	0
lo		144	lo	192

input / 源程序.txt

```

float a;
int b;
int main() {
    a = 3;
    b = 4;
    float c = 2;
    c += 2;
    c = c * 2;
    int j;
    if (a < b + c)
        j = a + (b * c + 1);
    else
        j = a;
    while (j <= 100) {
        j *= 2;
    }
    return 0;
    // 最终结果: j=144(a<b+c)/192(a>b+c)
}

```

output / object_code.asm

```

.data
s: .space 44
fpconst_0: .float 0 # 一定有, 方便float转bool用。
.text
b main

main:
main_1:
# (main, 1) (':=', 3, None, 'a') type=float
addi $t1, $0, 3 # const int 3 -> $t1
# int->float
mtc1, $t1, $f1
cvt.s.w $f1, $f1
mov.s $f3, $f1 # a := 3
s.s $f3, s+0 # $f3 -> float a

# (main, 2) (':=', 4, None, 'b') type=int
addi $t1, $0, 4 # const int 4 -> $t1
move $t3, $t1 # b := 4
sw $t3, s+4 # $t3 -> int b

# (main, 3) (':=', 2, None, 'c') type=float
addi $t1, $0, 2 # const int 2 -> $t1
# int->float
mtc1, $t1, $f1
cvt.s.w $f1, $f1
mov.s $f3, $f1 # c := 2
s.s $f3, s+8 # $f3 -> float c

# (main, 4) ('+', 'c', 2, 'c') type=float
l.s $f1, s+8 # float c -> $f1
addi $t2, $0, 2 # const int 2 -> $t2
# int->float
mtc1, $t2, $f2
cvt.s.w $f2, $f2
add.s $f1, $f1, $f2 # c := c + 2
s.s $f1, s+8 # $f1 -> float c

# (main, 5) ('*', 'c', 2, 'c') type=float
l.s $f1, s+8 # float c -> $f1

```

```

addi $t2, $0, 2 # const int 2 -> $t2
# int->float
mtc1, $t2, $f2
cvt.s.w $f2, $f2
mul.s $f3, $f1, $f2 # $1 := c * 2
s.s $f3, s+16 # $f3 -> float $1

# (main, 6) ('<=', '$1', None, 'c') type=float
l.s $f1, s+16 # float $1 -> $f1
mov.s $f3, $f1 # c := $1
s.s $f3, s+8 # $f3 -> float c

# (main, 7) ('+', 'b', 'c', '$2') type=float
ld $t1, s+4 # int b -> $t1
# int->float
mtc1, $t1, $f1
cvt.s.w $f1, $f1
l.s $f2, s+8 # float c -> $f2
add.s $f3, $f1, $f2 # $2 := b + c
s.s $f3, s+20 # $f3 -> float $2

# (main, 8) ('<', 'a', '$2', '$3') type=float
l.s $f1, s+0 # float a -> $f1
l.s $f2, s+20 # float $2 -> $f2
# $3 := a < $2
c.lt.s $f1, $f2 # f1<f2 则flag为1
bc1t main_8_1 # flag为1则跳转
addi $t3, $0, 0
b main_8_2
main_8_1:
addi $t3, $0, 1
main_8_2:
sw $t3, s+24 # $t3 -> int $3

# (main, 9) ('j=', '$3', 0, 15) type=bool
ld $t1, s+24 # int $3 -> $t1
sne $t1, $t1, 0 # int->bool
addi $t2, $0, 0 # const int 0 -> $t2
sne $t2, $t2, 0 # int->bool
beq $t1, $t2, main_15 # if $3 == 0 goto main_15

# (main, 10) ('+', 'b', 'c', '$4') type=float
ld $t1, s+4 # int b -> $t1
# int->float
mtc1, $t1, $f1
cvt.s.w $f1, $f1

```

```

1: s $2, s+8 # float c -> $f2
main: $f3, $f2, $f2 # $4 := b * c
s.s $f3, s+28 # $f3 -> float $4

# (main, 21) ('+', '54', 1, '55') type=float
1: s $f1, s+8 # float a -> $f1
addi $f2, $0, 1 # const int 1 -> $f2
# int->float
mtc1 $f2, $f2
cvt.s.w $f2, $f2
addi $f3, $f1, $f2 # $5 := $4 + 1
s.s $f3, s+32 # $f3 -> float $5

# (main, 22) ('+', 'a', '55', '56') type=float
1: s $f1, s+8 # float a -> $f1
1: s $f2, s+32 # float $5 -> $f2
addi $f3, $f1, $f2 # $6 := a + $5
s.s $f3, s+36 # $f3 -> float $6

# (main, 23) ('-', '56', None, 'j') type=int
1: s $f1, s+36 # float $6 -> $f1
# float->int
cvt.m.s $f1, $f1
mtc1 $f1, $f1
move $t3, $t1 # j := $6
sw $t3, s+16 # $t3 -> int j

# (main, 24) ('j', None, None, 16) type=int
b main_16 # goto main_16

main_16:
# (main, 25) ('<=', 'a', None, 'j') type=int
1: s $f1, s+8 # float a -> $f1
# float->int
cvt.m.s $f1, $f1
mtc1 $f1, $f1
move $t3, $t1 # j := a
sw $t3, s+16 # $t3 -> int j

```

main_16:

```

# (main, 16) ('<=', 'j', 100, '$7') type=int
ld $t1, s+16 # int j -> $t1
addi $t2, $0, 100 # const int 100 -> $t2
sle $t3, $t1, $t2 # $7 := j <= 100
sw $t3, s+40 # $t3 -> int $7

```

```

# (main, 17) ('j=', '$7', 0, 20) type=bool
ld $t1, s+40 # int $7 -> $t1
sne $t1, $t1, 0 # int->bool
addi $t2, $0, 0 # const int 0 -> $t2
sne $t2, $t2, 0 # int->bool
beq $t1, $t2, main_20 # if $7 == 0 goto main_20

```

```

# (main, 18) ('*', 'j', 2, 'j') type=int
ld $t1, s+16 # int j -> $t1
addi $t2, $0, 2 # const int 2 -> $t2
mul $t1, $t1, $t2 # j := j * 2
sw $t1, s+16 # $t1 -> int j

```

```

# (main, 19) ('j', None, None, 16) type=int
b main_16 # goto main_16

```

```

main_20:
# (main, 20) ('return', None, None, 0) type=int

```

测试文法强度

产生式: $S:A \quad A:eA \mid \varepsilon$

源程序: eee

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

					1 ['+', '+', '+', '#'] 的分析栈:				
					2 0 # + + + #				
					3 0 1 # + + + #				
1	LR(1)分析器:				4 0 1 1 # + + + #				
2		#	+	A	5 0 1 1 1 # + + + #				
3	0	r2	s1	2	6 0 1 1 1 3 # + + + A #				
4	1	r2	s1	3	7 0 1 1 3 # + + A #				
5	2	acc			8 0 1 3 # + A #				
6	3	r1			9 0 2 # A #				
					10 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 $D \rightarrow \varepsilon$, 以下是正确的:

```

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

```

以下是错误的:

```

15 | {'left': 'D', 'right': [], 'point': 0, 'origin': 5, 'isTer': True}
16 | {'left': 'D', 'right': [], 'point': 1, 'origin': 5, 'isTer': True}
17

```

以下也是错误的:

```

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

```

产生式: $S:A \quad A:bBc \quad B:\varepsilon$

源程序: bc

1	LR(1)分析器:						1	['b', 'c', '#'] 的分析栈:					
2		#	b	c	A	B	2	0	#	b	c	#	
3	0		s1		2		3	0 1		# b		c #	
4	1			r2		3	4	0 1 3		# b B		c #	
5	2	acc					5	0 1 3 4		# b B c		#	
6	3			s4			6	0 2		# A		#	
7	4	r1					7	Accepted					

测试 first 集是否计算正确

$$\begin{aligned} E &\rightarrow TE' \\ E' &\rightarrow + E \mid \epsilon \\ T &\rightarrow FT' \\ T' &\rightarrow T \mid \epsilon \\ F &\rightarrow PF' \\ F' &\rightarrow * F' \mid \epsilon \end{aligned}$$

产生式: $P \rightarrow (E) \mid a \mid b \mid \wedge$

源程序: ((a))

First 集:

非终结符	First 集	是否包含\$
E	(, a, b, ^	
E'	+	\$
T	(, a, b, ^	
T'	(, a, b, ^	\$
F	(, a, b, ^	
F'	*	\$
P	(, a, b, ^	

测试计算闭包时 $\text{first}(\beta a)$ 是否错误 (非常强的数据)

[I 中的每个项 $[A \rightarrow \alpha B \beta, a]$
 for (G' 中的每个产生式 $B \rightarrow \gamma$)
 for (FIRST(βa) 中的每个终结符号 b)

假如项目为 $[B \rightarrow b \cdot CDc, b]$, 则 $\text{first}(\beta a) = \text{first}(Dcb)$, 而非 $\text{first}(Db)$ 。若写错, 程序将会在以下文法中出错:

产生式: $S: A \quad A: BA \mid \epsilon \quad B: bCDc \quad C: a \quad D: \epsilon$

源程序: bacbac

```

1  Goto:
2  Goto(I0,b) = Closure( [28, 30] ) = {28, 30, 47} = { B->bCDc, # B->bCDc, b C->a, c } = I1
3  Goto(I0,A) = Closure( [4] ) = {4} = { S->A, # } = I2
4  Goto(I0,B) = Closure( [12] ) = {20, 8, 24, 26, 12} = { A->, # A->BA, # B->bCDc, # B->bCDc, b A->BA, # } = I3
5
6  Goto(I1,a) = Closure( [51] ) = {51} = { C->a, c } = I4
7  Goto(I1,C) = Closure( [32, 34] ) = {32, 34, 55} = { B->bCDc, # B->bCDc, b D->, c } = I5
    
```

$\text{bac} \rightarrow \text{bCc} \rightarrow \text{bCDc} \rightarrow \text{B} \rightarrow \text{BA} \rightarrow \text{A} \rightarrow \text{S}$

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

```

1  LR(1)分析器:
2  | #   c   a   b   A   B   C   D
3  0  r2           s1  2   3
4  1           s4           5
5  2  acc
6  3  r2           s1  6   3
7  4           r4
8  5           r5           7
9  6  r1
10 7           s8
11 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

词法&语法分析器开发

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：

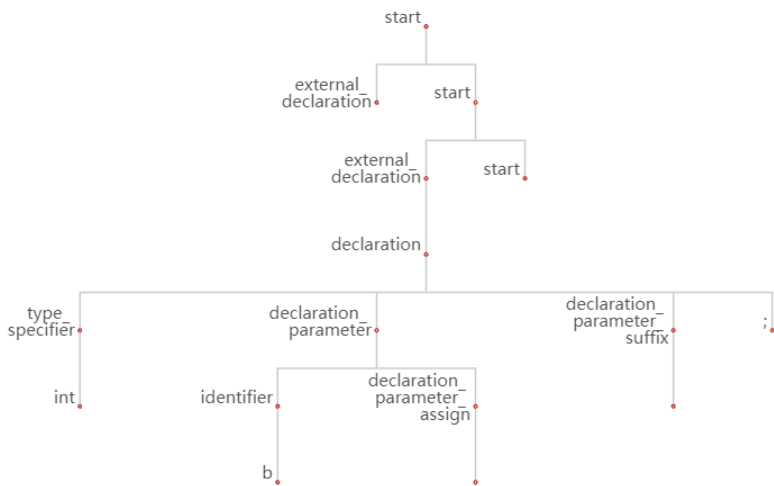
```
if newSet in closureSet:
    goto[i][j] = closureSet.index(newSet)
```

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 pycharts.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"))
```



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。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。

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 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 4.1.6 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：

```

1  ['b', 'a', 'c', '#'] 的分析栈:
2  0          #          b a c #
3  0 1          # b          a c #
4  0 1 5        # b a          c #
5  error

```

随后，观察 goto.txt。

假设 $Go(I_0, b) = I_1$ $Go(I_1, a) = I_4$ ，则问题一定出现在 I_0 、 I_1 、 I_4 中。

看 $Go(I_0, b)$ 展示的项目是否正确：

$Goto(I_0, b) = Closure([28, 29]) = \{28, 29, 47\} = \{B \rightarrow bCDc, b B \rightarrow bCDc, \# C \rightarrow a, c\} = I_1$

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。

语义分析器开发

关于 identifier 和 number

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。

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。

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。

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 `expression_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。

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`
 - (4) 若还想支持 `a=(1, 2, 3)` 或 `a=1, 2, 3`, 则需要 `exp` 允许直接转移到 `ari_exp`。这实际为不规范的写法, 但被 C 编译器支持。
- (1) (2) (3) (4) 构成 LR(1) 冲突。因此我们编写的编译器只支持 (1), (2), (3)。即: 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`。

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。

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。

```
Warning: 类型不匹配。变量b的类型为float, 赋值4的类型为int
```

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)。

```
Warning: 类型不匹配。左操作数c的类型为float, 右操作数2的类型为int
Warning: 类型不匹配。左操作数b的类型为int, 右操作数c的类型为float
```

过程调用时传入参数个数错误

```
637 print_error, file_analyzer
638 raise e
发生异常: IndexError x
Error: 函数demo接受1个参数, 传入0个。
File "C:\OneDrive - 同济大学\247共享资源\
raise e
File "C:\OneDrive - 同济大学\247共享资源\
raise e
```

过程调用时传入参数类型不匹配

```
Warning: 函数program的第2个参数类型为int, 传入了float
```

`void` 类型的过程反回了值, 或非 `void` 类型的过程返回不带值。


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

发生异常: TypeError ×
Error: 过程demo类型为int, 需要返回值。

```
void main(void)
{
    int a;
    float b;
    int c;
    a = 3;
    b = 4;
    c = 2;
    a = program
    return 1;
}
```

发生异常: TypeError ×
Error: 过程main类型为void, 不需要返回值。
File "C:\OneDrive - 同济大学\247共享资源\3.2\编译器

过程没写 return

```
except:
    raise RuntimeError(f"{domain} {formula} 跳转定位错误。请检查函数是否缺少了return。")
```

使用未定义变量

```
else:
    raise RuntimeError("Error: undefined variable %s" % nam)
```

在按位与/或/异或中使用了非整型操作数

```
elif op in ["^", "&", "|"]:
    if y_type != "int":
        raise TypeError(f'Error: 四元式 {form} 中 {y} 的类型为 {y_type}')
    if z_type != "int":
        raise TypeError(f'Error: 四元式 {form} 中 {z} 的类型为 {z_type}')
    formula_type = "int"
```

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。

目标代码生成器开发

关于 float

MIPS 中的浮点运算被放在了协处理器 FPU 中。FPU 的能力完全比不上 CPU。以下举例说明:

在整型之间比较时, 都有现成的指令(sne/sle/slt 等 \$t1, \$t2, \$t3)。然而, 在 MIPS 中, 比较结果会被存入 flag 中。flag 值不可直接被读取, 只有一条以 flag 值作为判定依据的跳转指令。因此, float 的比较要写成这样:

```
# (main, 8) ('<', 'a', '$2', '$3') type=float
l.s $f1, s+0 # float a -> $f1
l.s $f2, s+20 # float $2 -> $f2
# $3 := a < $2
c.lt.s $f1, $f2 # f1<f2 则 flag 为 1
bc1t main_8_1 # flag 为 1 则跳转
addi $t3, $0, 0
b main_8_2
main_8_1:
addi $t3, $0, 1
main_8_2:
sw $t3, s+24 # $t3 -> int $3
```

此外, 在 MIPS 中, float 不可以直接被赋小数值。若你想给整数寄存器赋值 123 时, 只需要 addi \$t1, \$0, 123。而你想给浮点数寄存器赋值 1.23 时, 需要:

```
.data
```

```
fpconst_1: .float 1.23
```

```
.text  
l.s $f1, fpconst_1
```

要么根据 IEEE 标准，将 1.23 的 32 位值算出来；要么使用一个专门的数据结构去维护。无论哪种做法，都让生成的代码显得不那么“友好”。

关于强转

我们的产生式没有类型转换，改起来又会导致很多额外的问题（用语法分析器检验修改后的产生式是否符合 LR1、修改词法分析器、修改语义分析器）等等。后来我发现，由于编写语义分析器时，我做了对应的检验，因此可以在报 warning 的同时，生成一些额外的四元式，来完成转换。例如：

```
int a = 5.123;
```

这种情况下，应该在语义分析时直接报 warning，随后生成 $(:=, 5, -, a)$ ；

```
int a = b; 且 b 为 float 类型。
```

这种情况下，应该报 warning，随后生成 $((float), b, -, b1) (:=, b1, -, a)$

```
float c = a * b; 其中 a 为 int, b 为 float
```

这种情况下，应该报 warning，随后生成 $((float), a, -, a1) (:=, a1, b, c)$

```
void demo(float a) {}; demo(b); 且 b 为 int 类型。
```

这种情况下，应该报 warning，随后生成 $((int), b, -, b1) (param, b1, -, -)$

以上是正确的写法。然而，因为当时的偷懒（只报 warning 没做强转），我这次在生成目标代码时才实现了类型转换。也就是说，针对每条四元式，我都要去获取一次各操作数的类型，当出现类型不匹配的情况时，根据操作符的不同，向不同的目标对齐类型。

关于 bool

由于支持 float 已经花费了太多的时间，因此没有编写对 bool 的支持。这直接导致了代码的冗长。每次判断时，

以 `if (a || b && c) {}`；为例：在我的程序中，会先将 b 和 c 分别转为 bool，计算出结果后，以 int 形式存入 d 中；随后再将 a 和 d 分别转为 bool，计算出结果后，执行条件跳转。

此外，想要将 float 转为 bool，不可以先转为 int（否则会出现 $0.45 \rightarrow 0 \rightarrow \text{False}$ ），而是需要直接和 0.0 去比较。上面已经说到，在 MIPS 中，float 的比较需要长达 7 行。因此，当涉及 float 时，情况将变得更加糟糕。

5 总结与收获

词法&语法分析器开发

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\(\$\beta a\$ \)](#)）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？

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。
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。
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。

语义分析器开发

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。

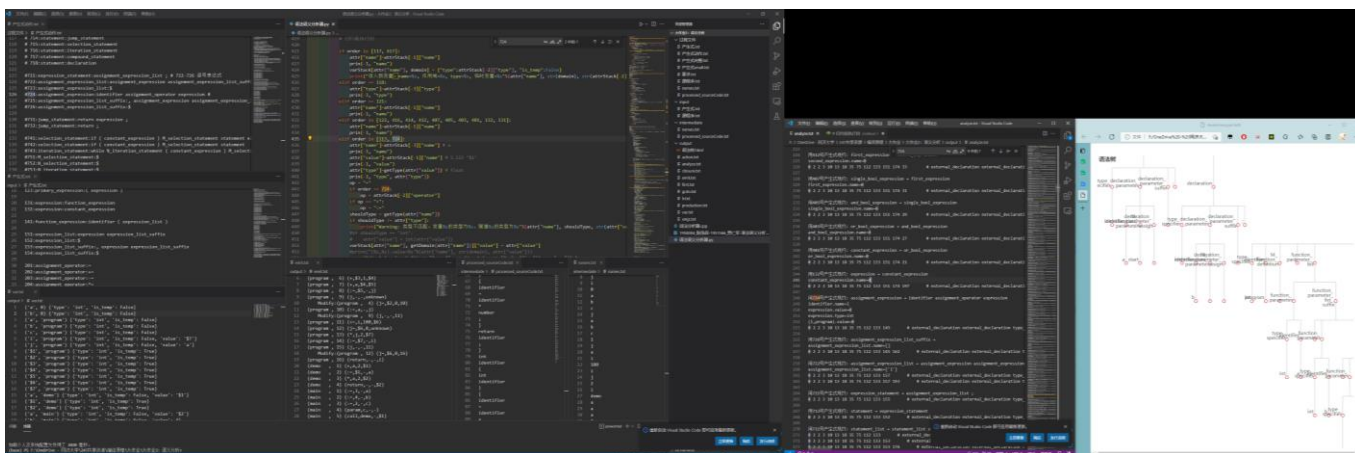
在编写语义分析器的时候，我们充分感受到了小组作业中同学间的相辅相成、默契与配合。在分析产生式的语义动作时，一位同学思路中的错误总是能由另一位同学发现。编写代码时，在一旁观察的同学总能即时指出代码中的 bug。如果这次作业仅仅由一个人来完成的话，相信我会遇到更多的问题，花出成倍的时间来 debug，并感受到相当的挫败。

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。

目标代码生成器开发

在编写目标代码生成器时，也得以纠正以前代码中的一些错误。例如：`+=`、`*=`等运算符应该被自动拆成 运算 和 赋值 两个四元式，而非直接生成`+=`符号的四元式。（相应的，词法分析器也要增加对这些符号的支持。）又例如，之前没有维护函数的参数列表，只保存了函数的类型。函数传参被当成普通的局部变量存入了变量表。这些错误都被纠正。

我选择了 MIPS32 目标代码语法进行开发。一来，MIPS 汇编指令集与书上采用的教学用指令集相似；二来，MIPS 的寄存器多，我觉得会更加便于编写。

事实证明，这是我“脑子一热”的想法——越多的选择代表优化越复杂。

在上学期编写词法语法语义分析器时，并没有要求支持多类型（其实这学期也没有）。当时写到表达式时，我觉得比较符号生成的结果不是 `bool` 而是 `int`，有些“不优雅”。这次编写目标代码生成器，写到除法时，我又觉得“`312/123=2`”“不优雅”（然而除了 `python`，这是符合主流语言的结果的）。于是我想：我尝试写一下多类型的支持吧。

这一想，就掉进了大坑。在过程调用方面，MIPS 没有原生的 `POP` 和 `PUSH`。当要传的参数过多，寄存器放不下时，需手动维护 `SP`。此外，MIPS 对浮点数的支持也堪称灾难。（吐槽见[关于 float](#)）

编写多类型支持花费时间远超预期，最终花费 4 天完成本次课设。最终代码长度（含过程调用和多类型支持的目标代码生成器）共 260 行。

6 参考文献

词法分析器、语法分析器 LR0、语法分析器 LR1：

只参考了编译原理教材，自己写的。

语法树可视化：

[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>

目标代码生成器：

MIPS 汇编详细指令

<https://blog.csdn.net/wxc971231/article/details/108032595>

计算机组成原理—— MIPS 指令系统（RSIC）

<https://blog.csdn.net/wxc971231/article/details/108032595>