装

订

线

同濟大學

TONGJI UNIVERSITY

《编译原理》

实验报告

实验名称	编译原理课程设计
姓名学号	王钧涛 2050254
学院（系）	电子与信息工程学院
专　　业	计算机科学与技术
任课教师	卫志华
日　　期	2023 年 5 月 18 日

目 录

装 订 线

1 需求分析

1.1 输入约定

（1） c 语言程序

需要用户输入 c 语言源程序文件，该文件默认命名为 input.c，可以通过添加默认输入［src］进行更改，具体方法见 1.3 节，其中存放的源 c 语言程序作为输入端提供给词法分析器。

（2）文法和语义规则

根据类 c 语言程序的要求，将产生式和保存在 base.json 中，每个元素是一个字典，包含两个 key，production 和 action。其中，production 是一个字典，包含 key：left 和 right，分别表示产生式的左边和右边；action 是一个列表，包含语义分析的语句。示例如下：

```
    {
      "production": {"left": "<函数名>", "right": ["$identifier"]},
      "action": [
                "checkFuncNotDefined(@r0.cont, @r0)",
                "ldict.func[@r0.cont]=0",
                "@l.cont=@r0.cont",
                "@l.debug_pos=@r0.debug_pos",
                "ldict.nowfunc = @r0.cont",
                "@l.quad=nextquad",
                "emit(@r0.cont+':','','','')"
            ]
```

下面是一些语法约定和处理上的细则：

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

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

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

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

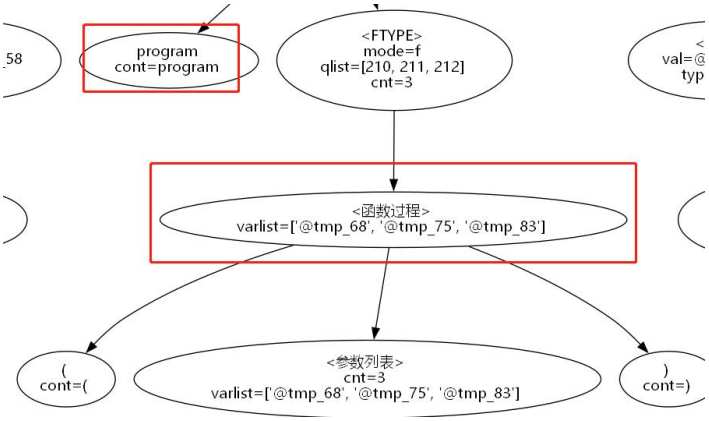
我的设计优点是将输入文法和其对应的语义动作放置在一起输入，并且为语义分

析设计了一个专用语言，不仅使得语法和语义可以更容易进行添加，同时也使得语义动作不再受制于具体的机内编写方式，配合 1.2 节中提到的语法分析树的直观输出，使得在我的编译器上进行文法语义开发可以以更贴近语法树的通用语言表达，而不需要考虑这些内容如何在语义分析器中进行存储和计算。这是由于本次课程设计采用 Python 这项解释性、弱类型语言开发所带来的独特优势。

举个例子：在设计函数调用的过程中，我想要需要在<函数过程>这个产生式被规约的时候获得所有调用变量的变量名，比如对如下语句：

```
1.    a[1][1]=program(a[0][0],a[0][1],demo(a[1][0]));
```

其语法树部分如下：



Promgram 函数传递了三个变量，分别是临时变量 68，临时变量 75 和临时变量 83。

需要注意的是，这里是专为函数过程设计的变长字符串数组列表，如果采用 c++ 进行开发，就需要在通用的节点类中专门定义一个 vector<string>的可选类型，命名为 varlist，并且设计专门的处理函数处理这条产生式，如果之后发现设计不合理、需求变更需要更换函数调用方法或者需要支持更多的调用方式（如 c 语言有的变长参数列表，以 .. 作为关键词），不仅需要重新修改节点类的定义，还需要新增一个处理函数，对敏捷开发十分不友好。

而在我的程序中，只需要在文法定义的同时输入希望执行的操作：

```
1. {
2.     "production":{"left": "<FTYPE>", "right": ["<函数过程>"]},
3.     "action": [
```

```
4.          "@l.mode='f'; @l.qlist=[]",
5.          "for j, i in enumerate(@r0.varlist):\n @l.qlist->append(nextquad)
    ;emit('par',i,j,''); UpdateVarUse(i, ldict.nowfunc);",
6.          "@l.cnt=len(@r0.varlist)"
7.      ]
8. },
9. {
10.        "production":{"left": "<函数过程>", "right": ["$(","<参数列
    表>","$)"]},
11.      "action": [
12.          "@l.varlist=@r1.varlist"
13.      ]
14. },
```

其中，@l 代指产生式左边的节点，@r[x]代指产生式右边的第 x 个节点（从 0 开始计算），在 action 内写一句 "@l.varlist=@r1.varlist" 就可以自动在<函数过程>这个节点中生成变长字符串数组列表，并将其值从<参数列表>这个节点中传递到该节点。同时，这样的写法也支持调用 python 内的函数，比如第五行的 for 循环调用 UpdateVarUse 内置函数，实现等价函数调用的编写，当不要这些功能的时候，无需更改任何语义分析器部分，只需将 json 配置文件中的这几行进行修改即可。

1.2 输出形式

输出形式如下：

（1）词法分析器：将结果保存在 base.py 中的 w_dict 里，w_dict 是一个列表，每个元素是一个元表，第一项 w_type 是该 word 的类型，包括数字类型（$digit_int），单词类型（$identifier 或保留字），第二项是单词 word，第三项是位置信息。

样例如下：

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

该类 c 语言程序所产生的 w_dict 如下：

```
1. ['$int', 'int', (1, 3)]
2. ['$identifier', 'main', (1, 8)]
```

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

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

（2）中间代码生成器：将结果保存在 base.py 中的 mid_code 里，mid_code 是一个列表，每个元素是一个元组，每个元组代表一个中间代码四元式。样例如下：

```
('j>=', 'tmp_0', '1', 107), ('j', '', '', 112), ('par', '1', '', '')
```

同时，将中间代码输出至文件 output.txt

```
中间代码:
100: ('binary:', '', '', '')
101: ('=', '1', '', '#eax')
102: ('ret', '', '', '')
103: ('main:', '', '', '')
104: ('+', '2', 'a', 'tmp_0')
105: ('j>=', 'tmp_0', '1', 107)
106: ('j', '', '', 112)
107: ('par', '1', '', '')
108: ('par', 'a', '', '')
109: ('call', 'main', '', '')
110: ('=', '#eax', '', 'a')
111: ('=', '1', '', 'cnt')
112: ('=', 'a', '', 'c')
113: ('=', 'a', '', 'd')
114: ('=', '1', '', 't')
115: ('=', 'cnt', '', '#eax')
  116: ('ret', '', '', '')
```

（3）代码变量表：

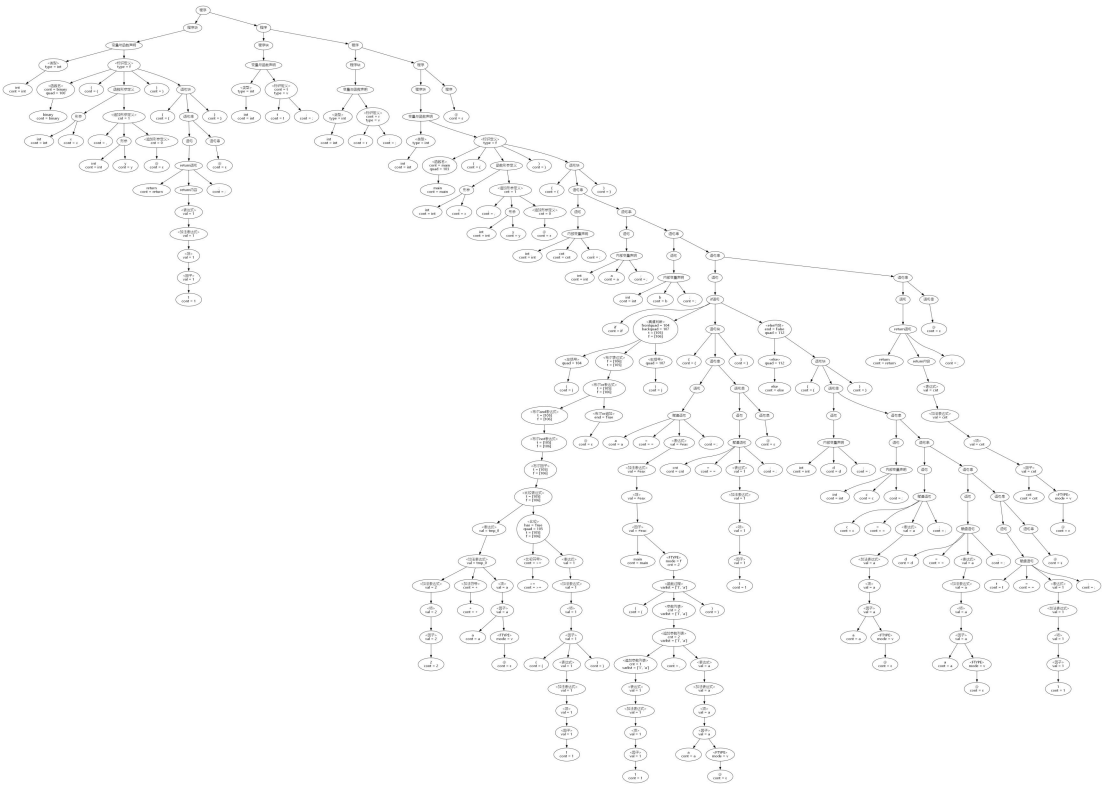
将结果同样保存在输出文件的 output.txt 下，代码变量表中会包含所有生成变量的变量名、所属函数、是否全局、变量 size，除此以外若是数组还会包括相应的

数组定义信息。

样例如下：

```
0: ('ans', 4, '$global', 'int', None, 54)
4: ('a', 400, '$global', 'arr', ['10', '10'], 43)
194: ('a', 4, 'max', 'int', None, 4)
198: ('b', 4, 'max', 'int', None, 5)
19c: ('ans', 4, 'max', 'int', None, 6)
1a0: ('@tmp_0', 4, 'main', 'int', None, 10)
1a4: ('@tmp_1', 4, 'main', 'main', None, 14)
1a8: ('@tmp_2', 4, 'main', 'int', None, 14)
```

（4） 语义分析树：利用 grammar_tree 画出语义分析树，生成 grammer_tree.png。（默认不启用该功能，需要在编译选项里添加 [-d 1] 命令才可以打开）语义树可以更为清晰直观的展示规约过程，也可也快速定位问题。生成的语义树图片如下：



语义树的节点信息如下（部分）：

```
    // Grammar Tree
    digraph {
        node0 [label="int
    cont = int
    " fontname="Microsoft YaHei"]
```

```
6.          node1 [label="<类型>
7.      type = int
8.      " fontname="Microsoft YaHei"]
9.          node1 -> node0
10.         node2 [label="binary
11.     cont = binary
12.     " fontname="Microsoft YaHei"]
13.         node3 [label="<函数名>
14.     cont = binary
15.     quad = 100
16.     " fontname="Microsoft YaHei"]
17.         node3 -> node2
18.         node4 [label="(
19.     cont = (
20.     " fontname="Microsoft YaHei"]
21.         node5 [label="int
22.     cont = int
23.     " fontname="Microsoft YaHei"]
24.         node6 [label="x
25.     cont = x
26.     " fontname="Microsoft YaHei"]
27.         node7 [label=<形参>
28.      fontname="Microsoft YaHei"]
29.         node7 -> node5
30.         node7 -> node6
31.         node8 [label=",
32.     cont = ,
33.     " fontname="Microsoft YaHei"]
34.         node9 [label="int
35.     cont = int
36.     " fontname="Microsoft YaHei"]
```

本次课程设计选用了 Python 作为实现语言，因此实现了包含中文的语义产生式，相应生成的语义分析树也更加直观易于理解。

（5） 最终代码

最终代码由 generator.py 生成，结果保存在 output.s 中，本次课程设计将中间代码翻译为 MIPS 指令集下的汇编代码，可在 MARS 模拟分析器中执行，也可以与同学期计算机系统结构课程设计的 89 条 CPU 大作业配合，在该 CPU 上执行。

样例如下：

```
1.  #0: (':', 'program', '', '')
```

```
program:

#1: ('=i', '0', '', '@tmp_0')
addi $4, $zero, 0

#2: ('=', '@tmp_0', '', 'i')
add $5, $4, $zero

#3: ('+', 'b', 'c', '@tmp_1')
lw $6, 268501036 #load <b> in func <program>
lw $7, 268501040 #load <c> in func <program>
add $8, $6, $7

#4: ('j>', 'a', '@tmp_1', 106)
lw $9, 268501032 #load <a> in func <program>
sw $5, 268501044 #save <i> in func <program>
bgt $9, $8, L106

#5: ('j', '', '', 112)
j L112

#6: ('*', 'b', 'c', '@tmp_2')
L106:
lw $4, 268501036 #load <b> in func <program>
lw $5, 268501040 #load <c> in func <program>
mul $6, $4, $5
```

装 订 线

1.3 程序功能

1. 项目整体功能

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

输入	**内容**
input.c	需要分析的类 c 语言源程序
输出	**内容**
词法分析结果	成功输出[Info]Lexical analysis

	success!，失败输出详细报错内容
语法分析过程	每一轮次的符号栈，状态栈，以及转移的方程
语法分析结果	成功输出[Info]Garmmar analysis success!，失败输出详细报错内容
程序运行结果	成功输出[Info]Compile success!，失败输出详细报错内容
GrammerTree.png	语法分析树
output.txt	中间代码和符号表
output.s	最终代码

装

订

线

2. 词法分析器 lexer.py

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

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

3. 语法分析器 gparser.py

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

输入	**内容**
w_dict	词法分析器结果
输出	**内容**
（[-debug 1]时）stdout	分析过程
grammer_tree	语法树
Action_goto	Action_goto 表
Project_set	Project 集合

4. 语义分析器 analyser.py

语义分析器中定义语义分析所使用的功能函数以及规约是需要调用的分析函数。其中，功能函数包括故障处理函数，emit 函数，backFill 回填函数，newVar 定义新变量函数，makelist 生成布尔表达式的链函数，mergelist 将别人表达式的链合并函数；分析函数 analysis 在语法分析规约的时候被调用。

输入	**内容**
oper stream	语法分析器结果
输出	**内容**
（[-debug 1]时）stdout	语义分析过程
mocode	中间代码
ldict	符号表
Output.txt	中间代码和符号表

5. 汇编代码生成器 generator.py

汇编代码分析器使用语义分析器生成的中间代码和变量表。

输入	**内容**
mocode	中间代码
ldict	符号表
输出	**内容**
（[-debug 1]时）stdout	最终代码生成过程
ostr	最终代码表
Output.s	最终代码 MIPS 汇编文件

1.4 测试数据

测试数据共分为正确数据和错误数据两种，其中错误数据又分为词法语法错误和语义错误两者。针对错误数据，其文件名中包含了其错误原因。由于错误数据较多，此处不再全部展示，仅将问题区域进行选择性的展示以压缩篇幅。

正确数据：

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

词法语法错误数据：

（1）lack_of_).c

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

（2）lack_of_elseif.c

```
int main(){
    int d;
    int n;
    int m;
```

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

（3）lack_of_id.c

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

（4）lack_of_if.c

```
int main(){
    int d;
    int n;
    int m;
    int l;
    int r;
    int mid;
```

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

装

（5）lack_of_leftnum.c

```
=l+1;
```

（6）lack_of_procon.c

订

```
int binary_search()
```

（7）lack_of_proname.c

```
int (){
```

线

（8）lack_of_protype.c

```
binary_search(){
```

（9）lack_of_rightnum.c

```
l=;
cnt=;
```

（10）lack_of_sen.c

```
return
```

（11）lack_of_{.c

1. int binary_search()

语义错误数据类型概览如下：

（1）函数调用前没有声明

```
1. void main(void){
2.     int a;
3.     a = 1;
4.     print(a);
5.     return;
6. }
```

（2）函数重复定义

```
1. int test(int x, int y){
2.     return 1;
3. }
4. int binary(){
5.     return 2;
6. }
7. int t;
8. void main(void){
9.     return binary();
10. }
```

（3）变量调用前未声明

```
1. int test(int x, int y){
2.     return 1;
3. }
4. void main(void){
5.     x = 1;
6.     y = 1;
7. }
```

（4）变量重复定义

```
1. int test(int x, int y){
2.     return 1;
3. }
4. void main(void){
5.     int x;
6.     int x;
7.     x = 1;
8.     return;
9. }
```

（5）函数没有返回语句

```
1. int test(int x, int y){
2.     x = x + y;
3. }
4. void main(void){
5.     int x;
6.     int y;
7.     x = 0;
8.     y = 1;
9.     x = test(x, y);
10. }
```

（6）函数调用时参数数量不匹配

```
1. int test(int x, int y){
2.     x = x + y;
3.     return 1;
4. }
5. void main(void){
6.     int x;
7.     x = 0;
8.     x = test(xs);
9. }
```

（7）数组调用时维度错误

```
1. void main(void){
2.     int x[1][2];
3.     x[1] = 1;
4. }
```

（8）数组定义时长度错误

```
1. void main(void){
2.     int a[5][0];
3.     a[1][1] = 1;
4. }
```

具体的测试结果及结果分析在 4.1 节中阐释。

2 概要设计

2.1 任务分解

本次编译原理课程设计的任务要求如下：

使用高级程序语言实现一个类 C 语言的编译器，可以提供词法分析、语法分析、符号表管理、中间代码生成以及目标代码生成等功能。具体要求如下：

1. 使用高级程序语言作为实现语言，实现一个类 C 语言的编译器。编码实现编译器的组成部分。
2. 要求的类 C 编译器是个一遍的编译程序，词法分析程序作为子程序，需要的时候被语法分析程序调用。
3. 使用语法制导的翻译技术，在语法分析的同时生成中间代码，并保存到文件中。
4. 要求输入类 C 语言源程序，输出中间代码表示的程序；
5. 要求输入类 C 语言源程序，输出目标代码（可汇编执行）的程序。
6. 实现过程、函数调用的代码编译。
7. 拓展类 C 语言文法，实现包含数组的中间代码以及目标代码生成。

本次课程设计全部完成了以上所有功能。

据此，将任务拆分成：

1. 词法分析器 lexer.py
2. 语法分析器 gparser.py
3. 语义分析器 analyser.py
4. 文法和语义分析式 base.json
5. 基础工作 base.py
6. 目标代码生成器 generator.py
7. 模块集成 main.py

2.2 数据类型定义

1. 词法分析器

（1）保留字表，关键词：列表，每个元素为字符串

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

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

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

2. 语法分析器

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

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

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

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

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

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

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

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

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

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

3. 语义分析器

（1）中间代码 mid_code

mid_code 是一个列表，每个元素是一个元组，每个元组代表一个中间代码四元式（三地址表达式）。样例如下：

```
('j>=', 'tmp_0', '1', 107), ('j', '', '', 112), ('par', '1', '', '')
```

（2）list_dict

list_dict 是一个字典，含有四个 key，分别为 var、func、return。

var 的 value 是一个列表，列表中的每个元素是一个元表，元表中包含变量的名称、大小、作用域（属于的函数），var 用来保存语义分析过程中产生的新变量，新变量的命名方式为 tmp_{varnum}，var_num 为递增的全局变量，从而保证 var 的列表中保存的变量名不会重复。

```
1.     ldict['var'].append((name, 4, ldict['nowfunc']))
```

func 的 value 是一个字典，字典的 key 是函数的名称，字典的值保存函数的形参个数。用来检查函数未定义，函数重复定义，函数参数不符等错误。

```
1.     "ldict.func[ldict.nowfunc]=0"
2.     "ldict.func[ldict.nowfunc]=@r1.cnt+1"
```

return 的 value 是一个 int 值，保存是否函数进行了 return，用来检查没有 return 的错误。

```
1.     "ldict.return += 1"
```

4. 语义规则

列表，每一个元素是一个字典，字典包含两个 key，production 和 action，分别存储产生式和其对应的语义动作。

产生式 production 是一个字典，包含两个 key，left 和 right 分别表示产生式

的左部和右部。

语义动作 action 是一个列表，每个元素是一个字符串，包含用该产生式规约时需要执行的语义动作。

```
    {
      "production": {"left": "<函数名>", "right": ["$identifier"]},
      "action": [
                "checkFuncNotDefined(@r0.cont, @r0)",
                "ldict.func[@r0.cont]=0",
                "@1.cont=@r0.cont",
                "@1.debug_pos=@r0.debug_pos",
                "ldict.nowfunc = @r0.cont",
                "@1.quad=nextquad",
                "emit(@r0.cont+':','','','')"
            ]
     }
```

5. 语义分析树

语义树（grammar_tree）：列表，每个元素为自定义的字典，包含 sym、son、cont 和 debug_pos 四个 key。其中，sym 和 cont 的 value 是字符串，son 的 value 是整数列表，debug_pos 是一个元表。sym 与 cont 中存储的信息与语义树的节点输出信息相关，son 则为该节点的孩子节点，

```
    [{'sym': '$int', 'son': [], 'cont': 'int', 'debug_pos': (1, 2)}, {'sym': '<类型>', 'son': [0], 'cont': '', 'debug_pos': (1, -1), 'type': 'int'},
```

6. 目标代码生成 ostr

ostr 是一个列表，每个元素是一个字符串，包含所有输出的目标代码和生成注释。样例如下：

```
#1: ('=i', '0', '', '@tmp_0')
addi $4, $zero, 0

#2: ('=', '@tmp_0', '', 'i')
add $5, $4, $zero

#3: ('+', 'b', 'c', '@tmp_1')
lw $6, 268501036 #load <b> in func <program>
lw $7, 268501040 #load <c> in func <program>
add $8, $6, $7

```

```
#4: ('j>', 'a', '@tmp_1', 106)
lw $9, 268501032 #load <a> in func <program>
sw $5, 268501044 #save <i> in func <program>
bgt $9, $8, L106
```

2.3 主程序流程

1. main 函数

（1）根据参数决定输入文件

（2）打开文件，读取进入 str 字符串

（3）调用词法分析器，将分析结果存储在 w_dict 中（输出分析结果）

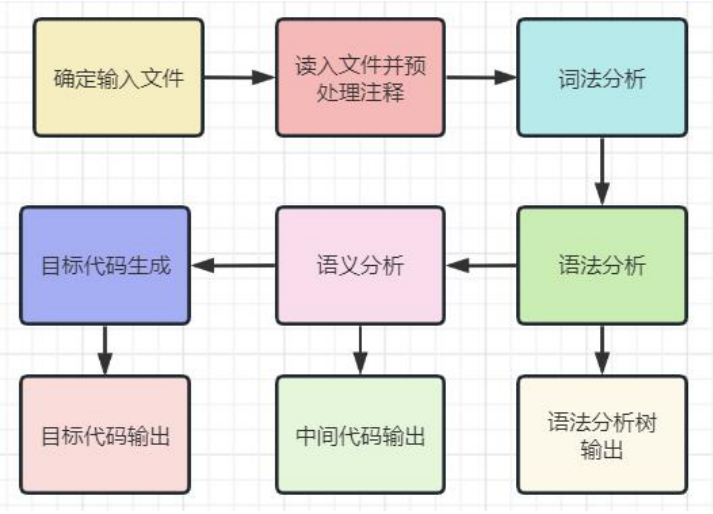
（4）调用语法分析器，在规约过程中同时生成中间代码，将语义树节点存储在 gramer_tree 中

（5）分析成功后，将中间代码保存至文件

（6）调用目标代码生成器，进行代码优化并生成目标代码

（7）生成成功后，将目标代码保存至文件

流程图如下：

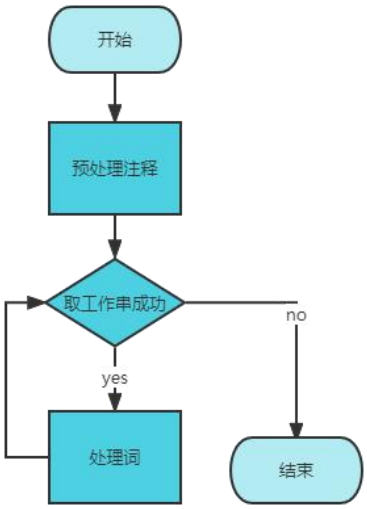


2. 词法分析器

（1）预处理注释

（2）循环获取工作串

（3）进行词语分割



3. 语法分析器

（1）求项目集合，产生 action/goto 表。

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

（3）每一步输出当前符号栈与状态栈。

（4）根据每一步结果生成语法树信息。

4. analysis 语义分析函数

（1）遍历语义动作

（2）判断语义是否是 pass

（3）翻译：将.val 翻译成['val']

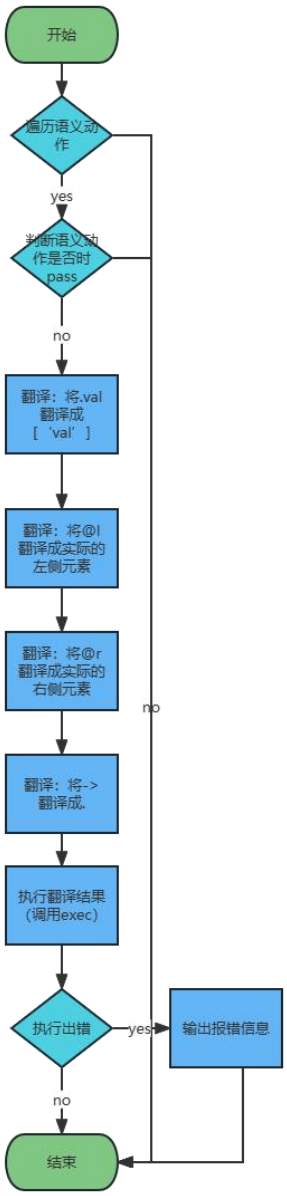
（4）翻译：将@l 翻译成实际的左侧元素

（5）翻译：将@r 翻译成实际的右侧元素

（6）翻译：将->翻译成.

（7）执行翻译结果（调用 exec）

（8）若执行出错则报出错信息



5. codeGen 目标代码生成函数

（1）初始化寄存器管理器，进行基本块划分

（2）生成活跃待用信息表

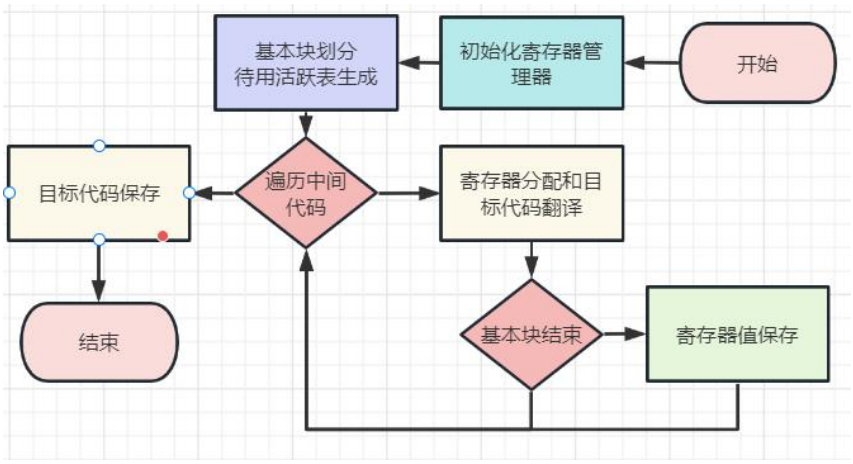
（3）遍历中间代码

（4）对每条中间代码，进行目标代码翻译：

（5）对内存量进行寄存器分配

（6）判断基本块是否结束，若结束清空寄存器管理器

（7）翻译完成后，保存目标代码



2.4 模块间调用关系

main 程序主要调用词法分析器和语法分析器以及绘制语法分析树的模块。

词法分析器调用了预处理注释，获取工作串以及词法分析的模块。

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

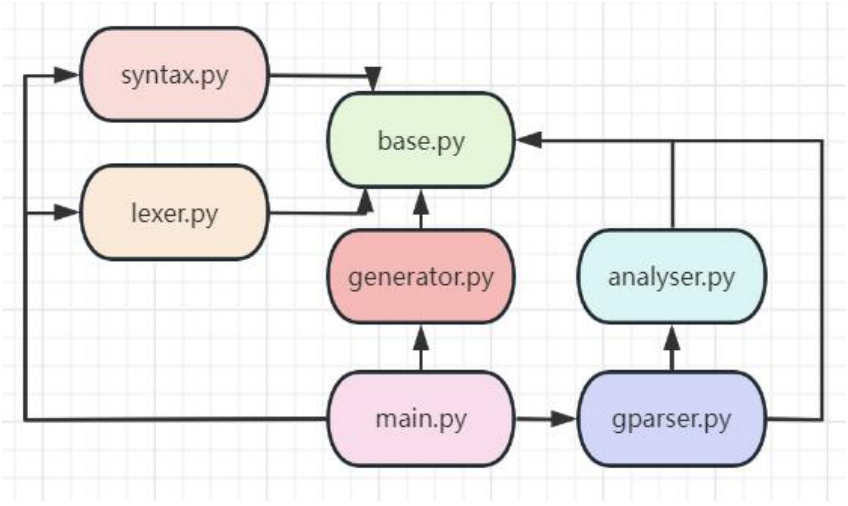
绘制语法分析树调用了 graphviz 中的 Digraph 模块。

语义分析器调用了 analyser.py 中的功能函数。

目标代码生成器调用了 generator.py 中的功能函数

其中，上述模块都会使用或改变 base.py 中的参数。

关系图如下：



3 详细设计

3.1 主要函数分析与设计

3.1.1 主函数 main

主函数负责将其他各个模块的主控函数耦合在一起进行工作：

```
1.    if __name__ == '__main__':
2.        try:
3.            with open(args.src, encoding='utf-8') as f:
4.                str = f.read()
5.                if Lexer(str) and GParser() and CodeGen():
6.                    midCodeSave(args.output)
7.                    print('\033[1;32;32m[Info]Compile success!\033[0m')
8.                    if(args.tree):
9.                        drawSyntaxTree()
10.       except FileNotFoundError as err:
11.           print('\033[1;31;31m[Error]#101\033[0m')
```

主函数较为简单，首先根据参数打开待分析的源程序文件 src，并通过 read 函数将其读入到字符串中，然后将其送入到词法分析器中产生词法分析器输出，并将该输出作为语法语义分析器的输入(在 base 文件中传递)，再将语法语义分析器的输出放到目标代码生成器中进行处理，判断这三者是否顺利完成工作，如果顺利完成代表输入文件通过了语法分析、语义分析、中间代码生成和目标代码生成，保存中间代码和目标代码，输出成功提示信息，如果画树参数被设置，则调用 drawSyntaxTree 函数绘制原始输入文件的语义分析树。如果文件打开失败，则输出错误提示信息。

3.1.2 词法分析器 Lexer

词法分析器负责从源文件生成机内输入序列，其主控函数如下：

```
1. def Lexer(i_str): #词法分析器入口函数，输出结果会放到 base 的 w_dict 中
2.     i_str = preProcessComment(i_str + '\r\n')#预处理，去除注释
3.     print('去除注解的源程序列表:', i_str)
4.     while getWorkString(i_str): #不断获取工作串
5.         try:
6.             dealWorkString() #进行分词解析
7.         except Exception as err: #捕获异常并进行异常输出
8.             print(str(err))
```

```
            return False
    print('单词机内表示序列:', w_dict) #词法分析器输出
    print("[Info]Lexical analysis success!")
    return True
```

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

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

词法分析器主控函数成功返回 True，失败返回 False

其中，注释处理函数如下：

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

此处的设计考虑到 C 语言的两种注释类型：

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

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

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

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

```
int mian(){
```

```
    int a;
    int b;
    if(a){
        return a;
        /* 这里是注释 a */
    }else if(b){
        return b;
        /* 这里是注释 b */
     }
```

显然，这里的注释 a 和注释 b 是两条语句，但是如果采用初始正则表达式会导致注释 a 开始符号匹配上注释 b 的结束符号，从而造成了无关代码的删除，因此在实际设计过程中需要从中间禁止’*/’符号的出现。

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

获取工作串函数如下：

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

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

其工作规则是：

若当前串的位置已经处于字符串的末尾，则返回 False

否则：

将行计数器增加一

从字符串的下一个\n 处进行切分

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

返回 True

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

```
def dealWorkString(): #词法分析
    global l_cnt, w_ptr, w_str, word, w_type
    ptr = 0 #句子的工作指针
    lw = len(w_str) #工作串的长度
    while ptr < lw: #当仍然有剩余的工作串
        ptr = ptr + re.search('[\x20\r\n]*', w_str[ptr:]).end() #过滤无用空格和回车
        if ptr == lw:
            return True
        if w_str[ptr].isdigit(): #处理数字
            ret = re.search('\\d+(\\.\\d+)?(e(\\+|-)?\\d+)?', w_str[ptr:])
            word = ret.group()
            if w_str[ptr+ret.end()].isalpha() or w_str[ptr+ret.end()]=='_':#数字匹配失败 抛出异常
                raise Exception('[Error]#101 in line {}, position {} : illegal num {}.'.format(l_cnt, ptr, word + w_str[ptr+ret.end()]))

            if '.' not in word and 'e' not in word: #根据其值区分整形数和浮点数
                w_type = '$digit_int'
            else:
                w_type = '$digit_int'#由于还没有实现浮点数，先当作整形数
        elif w_str[ptr].isalpha() or w_str[ptr] == '_': #处理单词
            ret = re.search('\\w*', w_str[ptr:])
            word = ret.group()
            w_type = '$identifier' if JudgeReverseWord(word) == -1 else JudgeReverseWord(word) #判断其是否为保留字
        else: #处理符号
            eptr = ptr + 1 #从下一个符号开始判断
            has_re = False
            while (not has_re and isSymbol(w_str[eptr - 1])) or JudgeReverseWord(w_str[ptr:eptr]) != -1: #匹配最长的是保留字的词
                if JudgeReverseWord(w_str[ptr:eptr]) != -1:
                    has_re = True
                eptr += 1
            word = w_str[ptr:eptr-1]
```

```
30.             if not has_re: #初始匹配失败 当前符号本身就不是关键字 抛出异常
31.                 raise Exception('[Error]#102 in line {}, position {} : illegal word {}.'.format(l_cnt, ptr, word))
32.             w_type = JudgeReverseWord(word)
33.         ptr += len(word)
34.         w_dict.append([w_type, word, (l_cnt, ptr)]) #将完成分割的词语放入输出序列中
```

该函数的工作原理如下：

首先设置 ptr 为当前工作位置指针，lw 为工作串总长度

循环判断 ptr 是否小于 lw，即还未分析结束：

则先去掉 ptr 之后的无用空格和回车

如果去掉后无剩余字符，则返回 True

否则判断第一个有用元素：

如果是字符或者下划线，进行【标识符分析】

如果是数字，进行【数字分析】

如果是符号，则进行【保留符号分析】

将分析结果[标识号，实际词语，原始位置]输入机内结果序列中

设置 ptr 指向当前词语的下一个位置

由于该函数较长，将其中部分分开进行叙述：

【标识符分析】：

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

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

使用正则表达式找出匹配的最大字符串 word

调用 JudgeReverseWord()函数判断其是否为保留字

如果是，则设置 w_type 为其对应的保留字标识

如果不是，设置 w_type 为$identitier

值得提及的是，标识符分析不会产生错误，因为其提取的过程中同时就满足了标

识符规则。

【数字分析】：

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

```
1. 1334
2. 764843241
3. 1314.0
4. 5485.44651
5. 125e4
6. 353.26e7
7. 135e+4
8. 643.777e-2
```

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

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

函数的工作原理如下：

应用正则表达式查找相应的符合要求的字符串，并赋值给 res

从 res 中提取字符串放入 word 中

若 word 之后跟的字符或者是下划线：

给出错误数字异常

否则通过正则表达式 d+判断是否是整形数

若是，设置标识符为整形数

否则，设置标识符为浮点数

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

【符号分析】：

符号分析的工作原理如下：

设置 has_re 变量用于统计当前是否已经找到了一个合适的符号串

设置当前工作指针 eptr 指向当前待分析的字符

循环执行：

如果当前串是保留字符，那么将 has_re 置为 True

直到已经出现过保留串并且当前字符不是保留串为止

判断是否找到了保留字串：

是：将其对应的标识号添加到 w_type

否：抛出符号错误异常

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

JudgeReverseWord 函数：

该函数用于判断输入串是否是保留字，并给其赋予标识符：

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

成功返回加上$的标识符号，如’$int，$return’，失败返回-1

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

3.1.3 语法分析器 GParser

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

语法分析器的主控程序如下：

```
def GParser():
```

```
    initProjectSet() #初始化项目集，构建 action 和 goto 表
     input_st = [['#', 'INPUT_END', w_dict[-1][2]]] + w_dict[::-1] #设置输入机内序列
     state_st, sym_st, id_st, son_st = [0], ['#'], [], [] #设置工作栈
    t_cnt=1 #轮次记录变量
    try:
        while len(input_st) > 0:
            ns = (state_st[-1], input_st[-1][0]) #记录当前状态的元组
            t = action_goto.get(ns) #从 action 和 goto 表中查找该元组
                print('#######\n 当前轮次:{}\n 当前符号栈:{},当前状态栈 :{}\n 当 前 读 入 字 符 :{}, 转 移 方 程为:{}'.format(t_cnt,sym_st,state_st,input_st[-1][0], ns))
             if t is None: #如果未找到，说明该串存在语法错误
                print(ns)
                raise Exception('[Error]#201 in line {}, position {} : Unexpected word \'{}\' after \'{}\'.'.format(*input_st[-1][2], input_st[-1][1], sym_st.pop()))
              if t[0] == 's' or t[0] == 'g': #移进或者 goto,两者代码相同
                it = input_st.pop()
                sym_st.append(it[0]) #将符号提取出符号栈中
                state_st.append(t[1]) #将当前的状态提取出放入状态栈
                 id_st.append(addGrammarTreeNode(sym_st[-1], son_st, it[1])) #存放语法树节点
                son_st = []
             elif t[0] == 'r': #规约
                  for i in range(len(productions[t[1]]['right'])): #如果不是从空串规约而来，设置子节点
                    sym_st.pop() #将被规约的子节点弹出
                    state_st.pop()
                    son_st.append(id_st.pop()) #并将其放入孩子栈中
                if len(productions[t[1]]['right']) == 0: #如果是空串,额外添加ε
                    son_st.append(addGrammarTreeNode('@', [], 'ε'))
                 input_st.append([productions[t[1]]['left'], '']) #将规约完的节点添加到 input 串中
             else:#规约成功 acc
                print("[Info]Garmmar analysis success!")
                break
             t_cnt += 1
    except Exception as err:#异常处理
        print(str(err))
        return False
    return True
```

其工作原理为：

首先求好有效项目集、action 表和 goto 表

循环直到 input 栈为空：

获取当前转移状态的元组

从 action_goto 表中查找下一步需要执行的状态

若为空，说明该串未找到，发出语法分析异常并给出异常位置

如果是移进或者 goto 操作，采用统一的操作方式：

将 input 栈的首元素弹出并将对应值放入符号栈和状态栈中

用该元素和孩子表作为参数注册语法树节点并将该节点放到 id 栈中

将孩子表设为空

如果是规约操作：

如果不是从空串规约而来：

对每一个孩子，将其从符号栈和状态栈中弹出并加入孩子表里

反之：

额外向孩子表中注册并添加 ε 节点

将规约完的节点添加到 input 栈中

如果都不是，那就是遇到了 acc 规约成功符号：

输出规约成功信息并退出

进行异常处理，捕获其中产生的异常并输出

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

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

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

```
1.  def initProjectSet():
2.      start = deepcopy(productions[0])
3.      start['dot'], start['accept'] = 0, '#'#生成初始集合
4.      c = findClosure([start]) #求解初始集合的闭包并加入有效项目集中
5.      project_set.append(c)
6.      top = 0 #递归调用计算
7.      while top < len(project_set): #采用广度优先搜索方式进行搜索
8.          for x in symbol_list: #对每一个正在被搜索的状态集:
9.              next_set = findGoto(project_set[top], x)
10.             if len(next_set) > 0 and not next_set in project_set:#将新构造的集合放入项目簇中
11.                 project_set.append(next_set)
12.             #if len(next_set) > 0: #调试语句
13.              #    print("#{}=<{},{}>:".format(project_set.index(next_set), top, x), next_set)
14.             if len(next_set) > 0:
15.                 if not isTerminalSymbol(x): #如果x不是终结符,那么填goto集
16.                     action_goto[(top,x)] = ['g', project_set.index(next_set)]
17.                 else: #否则填在移进集
18.                     action_goto[(top, x)] = ['s', project_set.index(next_set)]
19.         for wt in project_set[top]:
20.             if wt['dot'] == len(wt['right']):#遍历next集中的规约项目
21.                 ns = {'left':wt['left'], 'right':wt['right']} #将action_goto集的accept位设为规约
22.                 action_goto[(top, wt['accept'])] = ['r', productions.index(ns)] if ns['left'] != '<开始>' else ['acc']
23.         top += 1 #选取下一个集合进行操作
```

其工作原理为：

从第一个项目开始生成初始集合

将初始集合的闭包加入有效项目集中

进行广度优先搜索，对有效项目集中每一个没有被搜索过的节点：

用符号集里的每一个元素计算其 goto 集，如果该 goto 集不为空：

判断其是否在有效状态集合中，如果不在则加入

计算其在有效状态集中的 id

根据 x 是否为终结符，将其填在 action 集或是 goto 集中

然后对该集合的规约项目，将其展望串放入 action 集中

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

```
1. def findGoto(proj, sym)->list: #通过输入的项目集和 sym 找其 goto 后的 closure 集
2.     c = [] #设置初始 goto 集为空
3.     for p in proj: #对于输入的每一个串,都尝试能否接受 sym 符号
4.         if p['dot'] != len(p['right']) and p['right'][p['dot']] == sym:
5.             np = deepcopy(p)
6.             np['dot'] += 1
7.             c.append(np) #如果可以接受,将其接受后的新串加入 goto 集中
8.     return findClosure(c) #计算计算后的 goto 集的闭包
```

该函数的工作原理是：

先设置初始 goto 集为空

对项目簇里面的每一个项目，都尝试其是否能接受输入的 sym 符号：

可以就先计算其接受后的新状态

让后将这个新状态加入到 goto 集中

最后返回该 goto 集的闭包

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

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

```
                                newp = deepcopy(p)
                                newp['dot'], newp['accept'] = 0, nas
                                if not newp in closure_set:#进行元组去重，避免重复元素添加
                                    closure_set.append(newp)
                        top += 1
                return closure_set
```

该函数的工作原理为：

将原项目簇进行拷贝以作为搜索的初始集

对每一个闭包里面没有被搜索过的项目：

若当前工作点后是非终结符：

将其之后的内容与其展望串合并，并赋值给 sym

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

对任意左部是遇到的非终结符的产生式：

分别将其变为新项目，其依次接受 First 集里的各展望串

如果该新项目还不在闭包中，将其添加入新项目

返回生成的项目集

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

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

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

该函数的工作原理如下：

令初始 first 集为空

依次遍历输入的符号串，对每一个符号 x：

计算他的 first 集

若其 first 集中含有 ε：

删除 ε，并将剩余符号加入 first 集中

否则：

并将计算结果加入 first 集中

终止遍历

返回生成的 first 集

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

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

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

```
                            if '@' in nf:#如果存在空集
                                nf.remove('@')#那么删除空集
                                f += nf
                            else:
                                f += nf
                                break
                        else:
                            f.append('@') #全部都含有空，将空集加入
        first[sym] = f #保存该元素值
        return f
```

其工作原理如下：

初始化 first 集为空

判断其是否已经搜索过，如果是直接返回结果

若其是终结符，直接返回

否则遍历所有产生式，寻找从他推出的产生式 p：

如果 p 能推出空，那么将 ε 加入

或者 p 能退出终结符，将该终结符加入

否则递归调用，依次计算其推出的非终结符串的 first 集 f：

若 f 中含有 ε：

删除 ε，并将剩余符号加入 first 集中

否则：

并将计算结果加入 first 集中

终止遍历

如果递归调用正常结束，将 ε 加入 first 集中

保存该结果

将计算获得的 first 集返回

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

3.1.4 语义分析器 analyser

语义分析器负责在语法分析器工作的时候被调用，根据提前设定的语义动作残生中间代码：

```
1.     def analysis(id, last_oper):
2.         for lst in prod_actions[last_oper]:
3.             if lst != 'pass':
4.                 if args.analysis_output: print(lst) #输出当前执行的语句
5.                  lst = re.sub(r'\.(\w+)',lambda x: "['{}']".format(x.group
   (1)), lst)#翻译 将.val 翻成['val']
6.                  lst = re.sub(r'@l',"grammar_tree[id]", lst) #翻译 将@l 翻
   成实际左边元素
7.                  lst = re.sub(r'@r(\d+)',lambda x: "grammar_tree[{}]".form
   at(grammar_tree[id]['son'][int(x.group(1))]), lst) #翻译 将@r1 翻译为实际
   右边元素
8.                 lst = re.sub(r'->','.', lst)#翻译 将-> 翻成.
9.                 try:
10.                    exec(lst)
11.                except Exception as err:
12.                    print(err)
13.                    return False
14.        return True
```

调用时机：在语法分析器 GParser()中，当该节点是规约产生时，则调用语义分析器，并将 last_oper 参数传给语义分析器，如果分析失败，则语法分析器直接返回。

```
1.     if last_oper != -1:#如果该节点是规约而来
2.         if not analysis(id_st[-1], last_oper): return False
3.         last_oper = -1
```

语义分析器的主控程序根据语法分析器传入的参数 last_oper 找到 prod_actions 结构中相应的语义动作。遍历该语义动作队列，如果不是 pass，则对该语句进行翻译操作，包括对.val 的翻译，对@1 的翻译，对@r 的翻译以及对->的翻译，最后调用 exec 函数，当作 python 语言执行。

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

语义分析器主控函数成功返回 True，失败返回 False

主控程序所调用的功能函数如下：

（1）故障处理函数-函数未定义错误：查找 ldict 数据结构（base.py 中的

list_dict）是否包含该函数名

```
    def checkFuncDefined(name, node):
        if name not in ldict['func']:
            raise Exception('\033[1;31;31m[Error]#301 in line {}, positio
n {}: Function name <{}> not defined.\n{}'.format(*node['debug_pos'], nam
e, getErrorCodeLine(*node['debug_pos'], node['cont'])))
```

（2）故障处理函数-函数重复定义错误：查找 ldict 数据结构（base.py 中的 list_dict）是否已经包含该函数名

```
    def checkFuncNotDefined(name, node):
        if name in ldict['func']:
            raise Exception('\033[1;31;31m[Error]#302 in line {}, positio
n {}: Function name <{}> has already defined.\n{}'.format(*node['debug_po
s'], name, getErrorCodeLine(*node['debug_pos'], node['cont'])))
```

（3）故障处理函数-变量未定义错误：检查 ldict 数据结构（base.py 中的 list_dict）中 global 类型或者该函数中是否含有该变量名

```
    def checkVarDefined(name,func,node):
        var_list = [x[0] for x in ldict['var'] if x[2] == '$global' or x[
2] == func]
        if name not in var_list:
            raise Exception('\033[1;31;31m[Error]#303 in line {}, positio
n {}: Variable name <{}> not defined.\n{}'.format(*node['debug_pos'], nam
e, getErrorCodeLine(*node['debug_pos'], node['cont'])))
```

（4）故障处理函数-变量重复定义错误：检查 ldict 数据结构（base.py 中的 list_dict）中 global 类型或者该函数中是否已经含有该变量名

```
    def checkVarNotDefined(name,func,node):
        var_list = [x[0] for x in ldict['var'] if x[2] == func]
        if name in var_list:
            raise Exception('\033[1;31;31m[Error]#304 in line {}, positio
n {}: Variable name <{}> has already defined.\n{}'.format(*node['debug_po
s'], name, getErrorCodeLine(*node['debug_pos'], node['cont'])))
```

（5）故障处理函数-函数返回错误：检查 ldict 数据结构（base.py 中的 list_dict）中是否有函数返回语句

```
    def checkHasReturn(fname, node):
        if ldict['return'] == 0:
```

```
3.             raise Exception('\033[1;31;31m[Error]#305 in line {}, positio
   n {}: Function <{}> doesn\'t have return statement.\n{}'.format(*node['de
   bug_pos'], fname, getErrorCodeLine(*node['debug_pos'], node['cont'])))
```

（6）故障处理函数-函数参数错误：检查 ldict 数据结构（base.py 中的 list_dict）中该函数参数个数是否正确

```
1.     def checkFuncParNum(fname, num, node):
2.         if ldict['func'][fname] != num:
3.             raise Exception('\033[1;31;31m[Error]#306 in line {}, positio
   n {}: Call <{}> mismatch parameters, need {} but give {}.\n{}'.format(*no
   de['debug_pos'], fname,ldict['func'][fname],num,getErrorCodeLine(*node['d
   ebug_pos'], node['cont'])))
```

（7）故障处理函数-类型转换警告：是否发生了类型的转换导致可能的数据丢失

```
1.     def WarningVarType(type1,type2,node):
2.         if maxType(type1, type1, 0) < maxType(type2, type2, 0):
3.             print('\033[1;31;35m[Warning]#101 in line {}, position {}: Po
   tential type downgrading <{}> to <{}>.\n{}'.format(*node['debug_pos'],typ
   e2,type1,getErrorCodeLine(*node['debug_pos'], node['cont'])))
```

（8）故障处理函数-空返回函数使用警告：是否发生了对返回值为空的函数的不正确使用

```
1.     def WarningVoid(node):
2.         print('\033[1;31;35m[Warning]#102 in line {}, position {}: Type <
   void> should not be used.\n{}'.format(*node['debug_pos'],getErrorCodeLine
   (*node['debug_pos'], node['cont'])))
```

（9）故障处理函数-函数维度错误：判断是否发生了调用的数组维度与声明的数组维度不匹配的情况

```
1.     def checkArrLength(name,length,func,node):
2.         v = getVar(name, func)
3.         if v is None or v[3] != 'arr' or len(v[4]) != length:
4.             raise Exception('\033[1;31;31m[Error]#307 in line {}, position {}:
    Wrong variable <{}> dimension: should be {} but given {} .\n {}'.format(*
   node['debug_pos'], name,len(v[4]) if v[3] == 'arr' else 0 , length , getEr
   rorCodeLine(*node['debug_pos'], node['cont'])))
```

（10）故障处理函数-函数声明时大小错误：判断声明完的变量大小是否在合理范围内

```
1.     def checkArrSize(size,node):
```

```
        if not (size > 0):
            raise Exception('\033[1;31;31m[Error]#308 in line {}, position {}: Wrong size <{}> given.\n {}'.format(*node['debug_pos'], size, getErrorCodeLine(*node['debug_pos'], node['cont'])))
```

（11）输出函数 emit：将全局变量 nextquad 自增 1，mid_code 变量中增加语句（emit 的传入参数）

```
    def emit(expr, t1, t2, s1):
        global nextquad
        nextquad += 1
        mid_code.append((expr,t1,t2,s1))
```

（12）回填函数 backFill：如果信息为 None 则使用 no 来回填，否则，保留原来信息。

```
    def backFill(no, tup):
        a = list(mid_code[no-100])
        for i in range(4):
            if tup[i] is not None:
                a[i] = tup[i]
        mid_code[no-100] = (a[0],a[1],a[2],a[3])
```

（13）新增变量函数 newVar：产生一个新的变量名，命名规则为 tmp_{var_num}，其中 var_num 是全局变量，每次调用自增 1，在 ldict 数据结构中增加该变量，包含名称，大小和所在函数信息

```
    def newVar(type): #新建一个普通变量
        global var_num
        name = 'tmp_' + str(var_num)
        var_num += 1
        ldict['var'].append((name, 4, ldict['nowfunc'], type))
        return name
```

（14）maxType 函数：返回 a 和 b 中更高级的变量类型，其中，变量类型包括 void，int 和 float，依次递增

```
    def maxType(a , b, tostr=True):
        type = ['void', 'int', 'float']
        return type[max(type.index(a), type.index(b))] if tostr else max(type.index(a), type.index(b))
```

（15）生成布尔表达式链函数 makelist：生成一个包含 i（传入参数）的列表

```
    def makelist(i): #生成布尔表达式的链
        return [i]
```

（16）合并链函数 mergelist：将 a 和 b 列表合并

```
    def mergelist(a, b): #将布尔表达式的两个链合并
        for i in b:
            if i not in a:
                a.append(i)
        return a
```

（17）batchlist 函数：将 li 中的语句进行串联

```
    def batchlist(li, no):#li：需要串联的链 no：链接上的语句号
        for i in li:
            backFill(i, (None,None,None, no))
```

此处的设计考虑到 python 语言的特点，可以直接在 base.json 中定义好产生式对应的语义动作，然后 analysis 根据一定的规则将语义动作语句进行翻译成 python 语句，再进行调用，该方法可以方便进行传参以及函数调用，使得语义规则的定义与设计不用考虑具体功能函数的实现，直接使用字符串表示。

3.1.5 语法语义树绘制函数 drawSyntaxTree

drawSynataxTree 函数用于将抽象的语法分析树绘制成 png 图片，从而直观的展示出来，实现过程调用 dot 库[1]，其代码如下：

```
    def drawSyntaxTree():
        dot = Digraph(comment='Grammar Tree') #生成语法图
        cnt = 0 #初始化节点统计变量
        for i in grammar_tree: #对所有语法树中的节点
            nodeName = 'node'+str(cnt) #设置节点名字
            #dot.node(nodeName, i['cont'] if len(i['cont']) else i['sym'], fontname="Microsoft YaHei") #添加节点
            show = ( i['cont'] if (i['sym'][0] == '$') else i['sym'] ) + '\n'
            for t in i.keys(): #绘制抽象语法树
                if t != 'sym' and t != 'son' and t != 'debug_pos' and ( t != 'cont' or t == 'cont' and len(i[t])):
                    show += t + ' = ' + str(i[t]) + '\n'
            dot.node(nodeName, show, fontname="Microsoft YaHei") #添加节点
```

```
            for j in i['son']: #设置孩子节点名
                sonName = 'node'+str(j)
                dot.edge(nodeName, sonName) #画自己到孩子节点的边
            cnt=cnt+1
        dot.render('grammar_tree', view=True, format='png') #生成图片
        os.remove('grammar_tree') #删除中间文件
```

函数的依赖：

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

函数的实现：

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

创建节点计数器 cnt，并初始化为 0。

遍历列表形式的 grammar_tree 中的每一个元素：

按照当前节点计数器的值，命名一个新节点。

定义显示信息 show：若节点是终结符，则将节点内容设置为该终结符；否则将节点内容设置为这个节点的类别。

绘制抽象语法树：遍历节点的 key，符合要求则加入 show 显示信息。

将新节点加入 dot 中。

遍历这个节点的所有儿子节点：

命名儿子节点。

在当前节点与儿子节点之间建边，加入 dot。

节点计数器自增。

生成 png 图片，展示语法树。

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

3.1.6 目标代码生成器 generator

目标代码生成器负责将语义分析器分析出的中间代码变成目标代码，其主控程序如下：

```
def Generator():
    varaddrInit()
    regMgr = RegManager()
    regMgr.allocInformationGenerate()
    bp = 4 * len(ldict['var'])
    sp = 0
    nowfunc = '$global'
    ostr.append('lui $ra, 0x40')
    ostr.append('addi $ra, $ra, 0xc')
     ostr.append('j main')
     ostr.append('j __END__')
     for i in range(len(mcode)):
         ostr.append('#' + str(i) + ': ' + str(mcode[i]))
         if i+100 in ldict['link_point']: #设置跳转点
             ostr.append('L' + str(i+100) + ':')
         code = mcode[i]
         translate(code)
     ostr.append('# compile finished.')
     ostr.append('__END__:')
     print("\033[1;32;32m[Info]Code generate success!\033[0m")
     return True
```

主控程序会对中间代码进行基本代码块的划分，并在此基础上实现待用及活跃信息表优化。目标代码生成器会将代码翻译成 MIPS 指令集[2]，此处我并没有遵循 MIPS 指令集对寄存器用法的分配，而是默认：

$0 作为零寄存器，其中永远存储 0。

$1 作为汇编器寄存器，保留。

$2 作为目标代码生成寄存器，保留。

$3 作为返回值存储器，功能类似于 x86 汇编下的 EAX 寄存器，保留。

$4 - $31 均为待用活跃信息表优化可以使用的寄存器，共 27 个。

上主控程序功能实现思路如下：

进行变量对应内存地址表的初始化

进行待用活跃信息表优化器的初始化

将当前工作函数设置为 main，并设置工作完后的返回函数

对每一句代码 code：

进行代码的翻译

输出翻译后的目标代码

下面介绍使用到的一些函数的设计：

(1) varAddrInit()：该函数负责从之前语义分析器分析后的符号表中提取出每个变量在内存栈中存放的位置，并且将其放入变量地址表 varaddr 中。

实现代码如下：

```
def varaddrInit():
    for i in ldict['var']:
        varaddr.append(i[1] + varaddr[-1] if varaddr else i[1])
    for i, item in enumerate(ldict['var']):
        varaddr[i] -= item[1]
```

(2) getVarAddress()：该函数负责利用到上面这个函数生成的变量地址表，让变量对应上其所属的变量地址。

实现代码如下：

```
def getVarAddress(name, func):
    save = 0
    for i in range(len(ldict['var'])):
        if ldict['var'][i][0] == name:
            if ldict['var'][i][2] == func:
                return varaddr[i]
            elif ldict['var'][i][2] == '$global':
                save = i
    return varaddr[save]
```

(3) getVarAddressFromId()：该函数与上面的函数非常相似，主要是为了处理函数调用时如何获取调用地址的问题。该问题如果是在类 C 语言设计的变量栈中则不会出现这样的问题，但是本次我的设计思路是在中间代码翻译的时候就进行变量地址的优化，因此每个变量在内存中的寄存器位置是给定的，传参数的时候就需要通过需要传参的函数名和其变量顺序进行传

递，因此设计了该函数，比如对一个包含两个参数 x 和 y 的函数 max，因为 x 和 y 定义在 max 函数声明头中，因此不会有任何变量的定义时间早于这两个形式参数，从而 x 在函数 max 中的变量 id 一定是 1，y 在函数 max 中的变量 id 一定是 2。在其他函数中调用该函数时，就只需要使用这个函数进行地址定位，然后将实参的值赋给形参即可。

函数实现如下：

```
def getVarAddressFromId(id, func):
    t = 0
    for i in range(len(ldict['var'])):
        if ldict['var'][i][2] == func:
            if t == id:
                return varaddr[i]
            t += 1
    return 0
```

(4) searchFuncBelongFromName()：用来搜索 name 变量实际从属的函数（比如，在 func 中引用了变量 a 而 a 未定义，就需要通过该函数查询其实际从属的函数）

函数实现如下：

```
def searchFuncBelongFormName(name, func):
    for i in range(len(ldict['var'])):
        if ldict['var'][i][0] == name:
            if ldict['var'][i][2] == func:
                return func
    else:
        return '$global'
```

(5) CutCodeBlock()：用来切分基本代码块，逻辑为遍历大妈短并寻找 j、call、ret、:等行和 j 跳转到的行进行切分。

函数实现如下：

```
def cutCodeBlock(self):
        self.cut = []
        self.cutfunc = []
        nowfunc = '$global'
        for i in range(len(self.vcode)):
            if self.vcode[i][0][0] == 'j' or self.vcode[i][0] == 'ca
ll' or self.vcode[i][0] == 'ret' or self.vcode[i][0] == ':':
                if self.vcode[i][0][0] == 'j':
```

```
                    self.cut.append(self.vcode[i][3] - 100)
                    self.cutfunc.append((self.vcode[i][3] - 100, now
func))
                if (i+1) not in self.cut:
                    self.cut.append(i+1)
                    self.cutfunc.append((i+1, nowfunc))
                if self.vcode[i][0] == ':':
                    nowfunc = self.vcode[i][1]
        self.cut.sort()
        self.cutfunc.sort()
        self.cut[-1] = self.cut[-1] + 1
```

(6) allocInformationGenerate()函数：用于生成待用活跃信息表

函数逻辑：对从后向前的每一个基本块内：

对从后向前的每一行代码：

将待用活跃信息表的对应项赋在其上

生成新的待用活跃信息

将这一待用活跃信息表的入口活跃变量赋给下一个初始待用活跃信息表

函数实现如下：

```
def allocInformationGenerate(self):
        self.cutCodeBlock()
        huoyue = dict()
        self.aInfrom = []
        nowfunc = '$global'
        def setHuoyue(chy, nt, j, n, load = False):
            name = searchFuncBelongFormName(self.vcode[j][n], nowfun
c) + '.' + self.vcode[j][n]
            if name not in huoyue: huoyue[name] = (-1, False)
            nt[name] = huoyue[name]
            huoyue[name] = (j, load) if load else (-1, False)
            if load: chy[name] = (-1, True)
            elif name in chy: chy.pop(name)
        for i in range(len(self.cut) - 1, -1, -1):
            nowfunc = self.cutfunc[i][1]
            chuoyue = dict()
            if i != 0: l = self.cutfunc[i-1][0]; r = self.cutfunc[i]
[0]
            else: r = l; l = 0
            tinform = []
            shuoyue = deepcopy(huoyue)
            for j in range(r-1, l-1, -1):
                nt = {}
                if self.vcode[j][0] == '=':
```

```
                    if self.vcode[j][1] != '#eax': setHuoyue(chuoyue, nt, j, 1, True)
                    if self.vcode[j][3] != '#eax': setHuoyue(chuoyue, nt, j, 3, False)
                elif self.vcode[j][0] == '=l':
                    setHuoyue(chuoyue, nt, j, 2, True)
                    setHuoyue(chuoyue, nt, j, 3, False)
                elif self.vcode[j][0] == '=r':
                    setHuoyue(chuoyue, nt, j, 1, True)
                    setHuoyue(chuoyue, nt, j, 2, False)
                elif self.vcode[j][0] == 'par':
                    setHuoyue(chuoyue, nt, j, 1, True)
                elif self.vcode[j][0] == '=i':
                    setHuoyue(chuoyue, nt, j, 3, False)
                elif self.vcode[j][0] in {'j<', 'j<=', 'j>', 'j>=', 'j==', 'j!='}:
                    setHuoyue(chuoyue, nt, j, 1, True)
                    setHuoyue(chuoyue, nt, j, 2, True)
                elif self.vcode[j][0] == 'jnz':
                    setHuoyue(chuoyue, nt, j, 1, True)
                elif self.vcode[j][0] in ('+', '-', '/', '*'):
                    setHuoyue(chuoyue, nt, j, 1, True)
                    if not isinstance(self.vcode[j][2], int): setHuoyue(chuoyue, nt, j, 2, True)
                    setHuoyue(chuoyue, nt, j, 3, False)
                tinform.append(nt)
            self.aInfrom += tinform
            huoyue = shuoyue
            for a in chuoyue:
                if a not in huoyue: huoyue[a] = chuoyue[a]
        self.aInfrom.reverse()
```

(7) GetReg()函数：负责为具体变量分配寄存器并控制其 load 和 store

以 LRU 方式分配寄存器，当需要释放原先的寄存器时，根据待用活跃信息表决定是否需要将其中的信息保存到内存中去。

函数实现如下：

```
def getReg(self, code, func, line, frommemory=True):
    if code == '#eax': return '$v0'
    if (code, func) in self.regs:
        self.ai_save[self.regs.index((code, func))] = self.aInfrom[line][searchFuncBelongFormName(code, func)+'.'+code]
        self.m_save[self.regs.index((code, func))] &= frommemory
        return self._i2n(self.regs.index((code, func)))
    savei = None
    hasNone = (None in self.regs)
```

```
    for i in range(len(self.regs)):
        if self.regs[i] == None or (hasNone and self.regs[i] == (code, func)):
            savei = i
            break
    else:
        savei = randint(0, len(self.regs) - 1)
    if self.regs[savei] != None:
        self._save(self.regs[savei][0], self.regs[savei][1], savei)
    self.regs[i] = (code, func)
    self.ai_save[i] = self.aInfrom[line][searchFuncBelongFormName(code, func)+'.'+code]
    self.m_save[i] = frommemory
    if frommemory: self._load(code, func, savei)
    return self._i2n(savei)
```

(8) Clear()函数：负责释放所有寄存器

函数实现如下：

```
def clear(self):
    for i in range(len(self.regs)):
        if self.regs[i] != None:
            self._save(self.regs[i][0], self.regs[i][1], i)
            self.regs[i] = None
```

3.2 产生式语义规则

产生式和其对应的语义动作存放在 base.json 中，在 base.py 中被加载存入变量 productions 和 prod_actions 中，被其它模块调用。

整体而言，本次课程大作业使用的产生式在课堂所涉及的范围内，严格与课程所教学的语义动作保持一致，只是在课程没有涉及的地方，如故障处理等区域，增设了前文所示的一系列函数用来进行错误诊断和 Warning 警告。

1. 说明语句

说明部分中把定义性出现的标识符与类型等属性相关联,从而确定它们在计算机内部的表示法、取值范围及可对其进行的运算。为了产生有效地可执行目标代码，对于说明部分的翻译，不仅仅把与标识符相关联的类型等属性填入符号表中，还必须考虑到标识符所标记的对象的存储分配问题。

把等号右边的类型赋值给等号左边条目，在 ldict 中填入相关信息。

例如，变量的声明：

```
{
    "production": {"left": "<声明类型>", "right": ["<类型>"]},
    "action": [
        "@l.type = @r0.type",
        "ldict.t_type = @r0.type"
    ]
}
```

2. 赋值语句

进行变量的检查和类型的检查，产生中间代码

例如：

```
{
    "production":{"left": "<赋值语句>", "right": ["$identifier", "$=", "<表达式>", "$;"]},
    "action": [
        "checkVarDefined(@r0.cont,ldict.nowfunc, @r0)",
        "WarningVarType(getVarType(@r0.cont), @r2.type, @r0)",
        "emit('=', @r2.val, '', @r0.cont)"
    ]
}
```

3. 布尔表达式

布尔表达式：用布尔运算符把布尔量、关系表达式联结起来的式子，包括布尔运算符：and，or，not 和关系运算符 $<, \leqslant, =, \neq, >, \geqslant$

布尔表达式的两个基本作用：用于逻辑演算，计算逻辑值；用于控制语句的条件式。

例如，布尔 or 表达式如下：

```
{
    "production":{"left": "<布尔 or 表达式>", "right": ["<布尔 and 表达式>", "<布尔 or 追加>"]},
    "action": [
        "if @r1.end: @l.t=@r0.t; @l.f=@r0.f;",
        "if not @r1.end: @l.f=@r1.f; @l.t=mergelist(@r0.t, @r1.t);",
        "if not @r1.end: batchlist(@r0.f, @r1.quad)"
    ]
}
```

4. 控制语句

根据 if-then，if-else，while 控制语句的翻译模式，分别进行翻译。

例如，while 语句如下：

```
1.     {
2.             "production":{"left": "<while 语句>", "right": ["$while", "<真值判断>", "<语句块>"]},
3.             "action": [
4.                 "batchlist(@r1.t, @r1.backquad)",
5.                 "batchlist(@r1.f, nextquad)",
6.                 "emit('j', '', '', @r1.frontquad)"
7.             ]
8.         }
```

WW

5. 过程调用

过程调用主要对应两种事：传递参数；转子（过程）

传地址：把实在参数的地址传递给相应的形式参数。调用段预先把实在参数的地址传递到被调用段可以拿到的地方；程序控制转入被调用段之后，被调用段首先把实在参数的地址抄进自己相应的形式单元中；过程体对形式参数的引用与赋值被处理成对形式单元的间接访问。

例如：

```
1.     {
2.         "production": {"left": "<函数名>", "right": ["$identifier"]},
3.         "action": [
4.             "checkFuncNotDefined(@r0.cont, @r0)",
5.              "ldict.func[@r0.cont]=0; ldict.return=0; ldict.functype[@r0.cont]=ldict.t_type",
6.             "@l.cont=@r0.cont",
7.             "@l.debug_pos=@r0.debug_pos",
8.             "ldict.nowfunc = @r0.cont",
9.             "@l.quad=nextquad",
10.             "emit(@r0.cont+':','','','')"
11.         ]
12.     }
```

6. 数组

数组主要考虑两件事：数组的声明和数组的调用。

在声明过程中，需要将数组的各维度信息进行保存，并逐一相乘以计算出数组

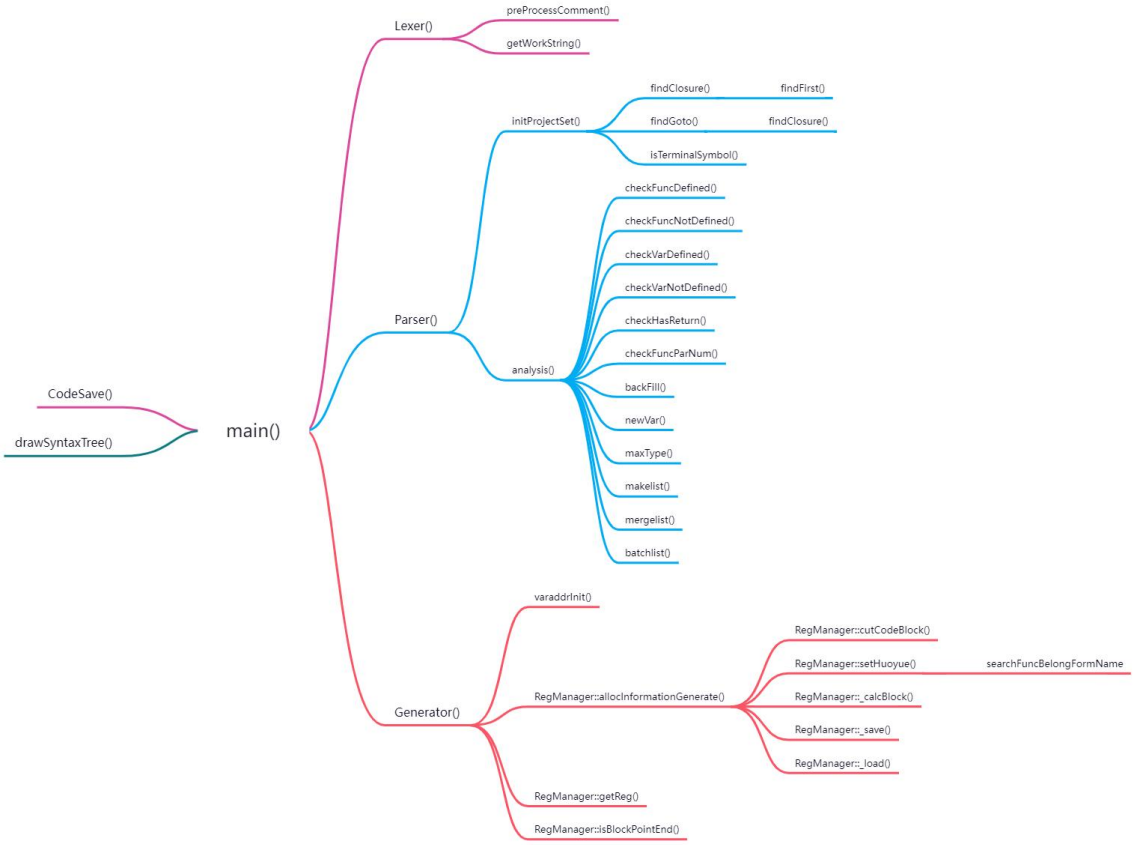
最终的长度，这一步还需要保存数组的维度，以供之后检查数组维度是否正确；而在调用过程中，需要先计算出数组各个维度的表达式值信息并赋予数组以计算，然后将计算出来的结果根据现在 len 的长度计算之前维度需要乘的维度大小信息，例如声明 a[4][5][6]后调用 a[1][2][3]，就需要在处理[2]时将前面的 1 乘以 5=5 再加 2 得 7，然后在处理[3]时将前面 7 乘以 6 再加 3 得 45，这是其最终的偏移值。当然，在该偏移值处理的过程中，还需要再将其乘以 4 以获得针对 int 型 4 字节的偏移值，即上述 45 的偏移值，其实际偏移位置为 45*4 字节=180。

例如：

```
{
        "production": {"left": "<数组>", "right": ["<ID>","$[","<表达式>","$]"]},
       "action": [
          "@l.len=1; @l.vname=@r0.cont; @l.cont=@r0.cont",
          "@l.offset = newVar(ldict.nowfunc); emit('*', @r2.val, 4, @l.offset); UpdateVarUse(@r2.val, ldict.nowfunc);",
          "@l.debug_pos = @r0.debug_pos"
       ]
    },
    {
       "production": {"left": "<数组>", "right": ["<数组>","$[","<表达式>","$]"]},
       "action": [
          "@l.len = @r0.len + 1; @l.vname = @r0.vname; @l.cont=@r0.cont",
          "ntmp=newVar('int'); emit('*', @r0.offset, int(getVar(@l.vname, ldict.nowfunc)[4][@r0.len-1]), ntmp)",
          "atmp = newVar(ldict.nowfunc); emit('*', @r2.val, 4, atmp); UpdateVarUse(@r2.val, ldict.nowfunc);",
          "@l.offset=newVar('int'); emit('+', ntmp, atmp, @l.offset); UpdateVarUse(ntmp, ldict.nowfunc); UpdateVarUse(atmp, ldict.nowfunc);",
          "@l.debug_pos = @r0.debug_pos",
          "UpdateVarUse(@r0.offset, ldict.nowfunc); UpdateVarUse(@l.vname, ldict.nowfunc); UpdateVarUse(@r2.val, ldict.nowfunc); UpdateVarUse(ntmp, ldict.nowfunc)"
       ]
    }
```

3.3 总函数调用图

整体程序的总函数调用图如下：



整个程序分为七个文件，分别为 base.py、lexer.py、gparser.py、analyser.py、syntaxTree.py、generator.py 和 main.py。

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

4 调试分析

4.1 测试数据与测试结果

1、语法词法测试：

1) #101 词法分析器::非法数字错误：

测试数据 1：

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

输出：

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

测试数据 2（漏乘号）：

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

输出：

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

2) #102 词法分析器::非法符号错误

测试数据 1：

```
1. int binary_search(){
2.     int cnt;
3.     cnt ^= 3;
4.     return cnt;
5. }
```

输出：

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

测试数据 2：

```
1. int binary_search(){
2.     int cnt;
3.     cnt == 3;
4.     return cnt;
5. }
```

输出：

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

词法分析器正确识别到 == 而非两个 = 号

3) #201 语法分析器::非预期的符号错误

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

测试数据 1：if 语句缺乏左括号

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

```
19.  }
```

程序输出：

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

正确识别异常，分析栈无误。

测试数据 2：main 函数左大括号失配

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

程序输出：

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

正确识别异常，分析栈无误。

测试数据 3：左值异常（部分代码）

```
1.           int c;
2.           int t;
3.           if(c>12){
4.               l=print(r);
```

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

程序输出：

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

正确识别异常，分析栈无误。

测试数据 4：函数名缺失（部分代码）

```
int (){
    int cnt;
    int ans;
    int l;
    int r;
    cnt = 0;
```

程序输出：

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

正确识别异常，分析栈无误。

测试数据 5：分号确实（部分代码）

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

```
8.              l=print(r)
9.          }
10.         l=l+1;
11.         cnt=cnt+1;
12.         ans=ans*cnt;
```

程序输出：

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

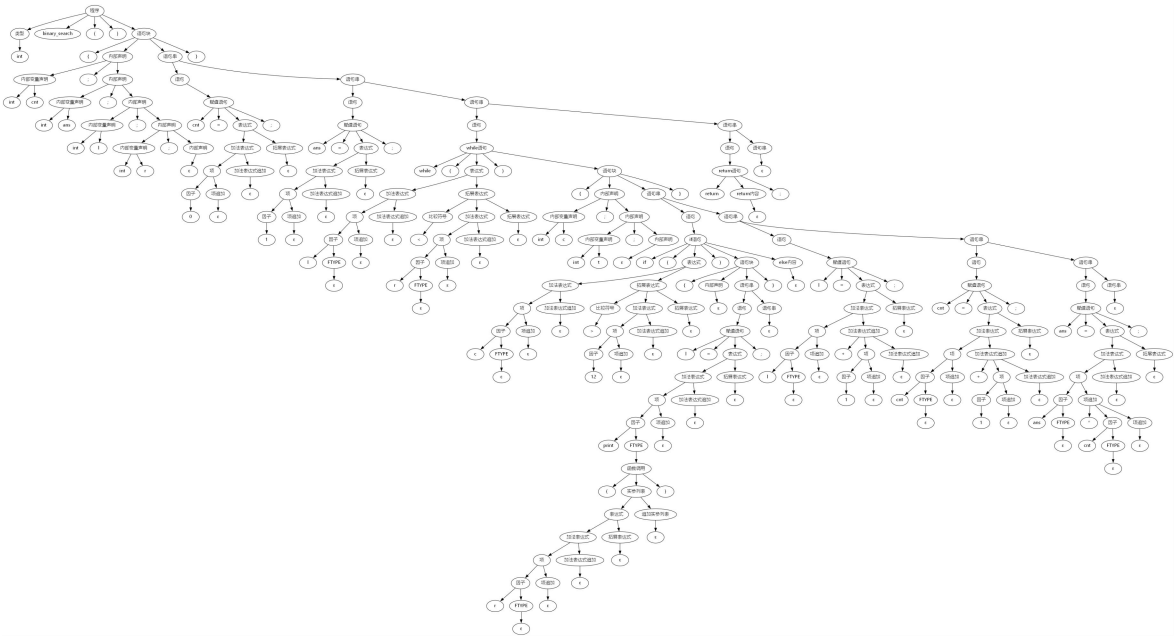
正确识别异常，分析栈无误。

测试数据 6：正确代码

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

程序顺利规约完毕，绘制出分析树：

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



2、语义测试

4) error#301 中间代码生成器::函数调用前未声明

在下面的代码中可以看到，在尝试调用名为 binary 的函数，但是这个函数并没有定义在代码段当中，因此应该程序应该会提示说函数未定义。

```
6.int main(int x, int y){
7.    int cnt;
8.    int a;
9.    int b;
10.     if(2+a>=(1)){
11.         a = binary(1,a);
12.         cnt = 1;
13.     } else {
14.         int d;
15.         int c;
16.         c = a;
17.         d = a;
18.         t = 1;
19.     }
20.     return cnt;
21. }
```

在终端中实际运行结果如下：

```
PS E:\_courses\编译\Complier\PigCompiler> python main.py ./examples/func_undefined.c
[Info]Lexical analysis success!
[Error]#301 in line 8, position 17: Function name <binary> not defined.
8:        a = binary(1,a);
```

可以看到，程序识别出了未定义的函数，并在相应位置报错。

5) error#302 中间代码生成器::函数重复定义

在下面的代码中可以看到，名为 binary 的函数被重复定义了两次，因此程序应该会提示 binary 函数被重复定义。

```
1.  int binary(int x, int y){
2.      return 1;
3.  }
4.
5.  int binary(){
6.      return 1;
7.  }
8.
9.  int t;
10.
11. int main(int x, int y){
12.     int cnt;
13.     int a;
14.     int b;
15.     if(2+a>=(1)){
16.         a = binary(1,a);
17.         cnt = 1;
18.     } else {
19.         int d;
20.         int c;
21.         c = a;
22.         d = a;
23.         t = 1;
24.     }
25.     return cnt;
26. }
```

实际运行结果如下：

```
PS E:\_courses\编译\Complier\PigCompiler> python main.py ./examples/func_undefined.c
[Info]Lexical analysis success!
[Error]#301 in line 8, position 17: Function name <binary> not defined.
8:          a = binary(1,a);
```

程序在函数第二次定义的时候报错。

6) error#303 中间代码生成器::变量调用前未声明

在下面的代码中，使用了未定义过的变量，程序应该会给出对应的错误提示。

```
1.  int binary(int x, int y){
2.      return 1;
3.  }
```

```
4.
5.  int main(int x, int y){
6.      int cnt;
7.      int a;
8.      int b;
9.      if(2+a>=(1)){
10.         a = binary(1,a);
11.         cnt = 1;
12.     } else {
13.         int d;
14.         int c;
15.         c = a;
16.         d = a;
17.         t = 1;
18.     }
19.     return cnt;
20. }
```

实际运行结果如下：

```
PS E:\_courses\编译\Complier\PigCompiler> python main.py ./examples/golvar_undefined.c
[Info]Lexical analysis success!
[Error]#303 in line 17, position 8: Variable name <t> not defined.
17:         t = 1;
```

程序在调用了未定义的变量 t 的时候报错。

7) error#304 中间代码生成器::变量重复定义

在下面的代码中，重复定义了 c 和 d 两个自动变量，因此程序应该会在变量第一次重定义的地方提示出错。

```
1.  int binary(int x, int y){
2.      return 1;
3.  }
4.
5.  int t;
6.
7.  int main(int x, int y){
8.      int cnt;
9.      int a;
10.     int b;
11.     int c;
12.     int d;
13.     if(2+a>=(1)){
14.         a = binary(1,a);
15.         cnt = 1;
16.     } else {
17.         int d;
18.         int c;
19.         c = a;
20.         d = a;
```

```
21.         t = 1;
22.     }
23.     return cnt;
24. }
25.
26. int t;
```

实际运行结果如下：

```
PS E:\_courses\编译\Complier\PigCompiler> python main.py ./examples/var_muldefined.c
[Info]Lexical analysis success!
[Error]#304 in line 17, position 12: Variable name <d> has already defined.
17:         int d;
```

程序在第一次重定义的时候报错。

值得一提的是，这里支持了不同嵌套深度下变量的重复定义，也就是根据嵌套深度来保存变量的生命周期，这与过程嵌套语言保持一致。

在下面的代码中，首先定义了一个全局变量 t，随后又在 main 函数中定义了一个同名的变量 t，期望程序可以正确运行。

```
1.  int binary(int x, int y){
2.      return 1;
3.  }
4.
5.  int t;
6.
7.  int main(int x, int y){
8.      int cnt;
9.      int a;
10.     int b;
11.     int t;
12.     if(2+a>=(1)){
13.         a = binary(1,a);
14.         cnt = 1;
15.     } else {
16.         int d;
17.         int c;
18.         c = a;
19.         d = a;
20.         t = 1;
21.     }
22.     return cnt;
23. }
```

实际运行结果如下：

```
PS E:\_courses\编译\Complier\PigCompiler> python main.py
[Info]Lexical analysis success!
[Info]Garmmar analysis success!
[Info]Compile success!
中间代码已输出到同级文件夹下..
```

检查中间代码的变量区，应该存在两个生命周期不同的同名变量。

```
变量表:
0: ('x', 4, 'binary', 'int')
4: ('y', 4, 'binary', 'int')
8: ('t', 4, '$global', 'int')
12: ('x', 4, 'main', 'int')
16: ('y', 4, 'main', 'int')
20: ('cnt', 4, 'main', 'int')
24: ('a', 4, 'main', 'int')
28: ('b', 4, 'main', 'int')
32: ('t', 4, 'main', 'int')
36: ('tmp_0', 4, 'main', 'int')
40: ('d', 4, 'main', 'int')
44: ('c', 4, 'main', 'int')
```

发现的确存在不同的两个 t。

8) error#305 中间代码生成器::函数没有返回语句

在下面的代码中，的 binary 函数没有定义 return 语句，测试程序是否能正确处理这一问题。

```
1.  int binary(int x, int y){
2.      x = 2;
3.  }
4.
5.  int t;
6.
7.  int main(int x, int y){
8.      int cnt;
9.      int a;
10.     int b;
11.     if(2+a>=(1)){
12.         a = binary(1,a);
13.         cnt = 1;
14.     } else {
15.         int d;
16.         int c;
17.         c = a;
18.         d = a;
19.         t = 1;
```

```
20.     }
21.     return cnt;
22. }
```

实际运行结果如下：

```
PS E:\_courses\编译\Complier\PigCompiler> python main.py ./examples/lack_of_return.c
[Info]Lexical analysis success!
[Error]#305 in line 1, position 9: Function <binary> doesn't have return statement.
1:int binary(int x, int y){
```

可以看到成功检测出了没有 return 语句的函数。

9) error#306 中间代码生成器::函数调用时参数错误

下面测试调用过程出错。这里主要的错误就是调用的参数与函数对不上。

```
1.  int binary(int x, int y){
2.      return 1;
3.  }
4.
5.  int t;
6.
7.  int main(int x, int y){
8.      int cnt;
9.      int a;
10.     int b;
11.     if(2+a>=(1)){
12.         a = binary();
13.         cnt = 1;
14.     } else {
15.         int d;
16.         int c;
17.         c = a;
18.         d = a;
19.         t = 1;
20.     }
21.     return cnt;
22. }
```

实际运行结果如下：

```
PS E:\_courses\编译\Complier\PigCompiler> python main.py ./examples/func_useerr.c
[Info]Lexical analysis success!
[Error]#306 in line 12, position 17: Call <binary> mismatch parameters, need 2 but give 0.
12:         a = binary();
```

可以看到，程序找到了调用错误的函数，并且指出了应该给的参数个数和实际给的参数个数。

此外，再对调用的实参没有声明做一个测试：

```
1.  int binary(int x, int y){
2.      return 1;
3.  }
4.
5.  int t;
6.
7.  int main(int x, int y){
8.      int cnt;
9.      int a;
10.     int b;
11.     if(2+a>=(1)){
12.         a = binary(e, f);
13.         cnt = 1;
14.     } else {
15.         int d;
16.         int c;
17.         c = a;
18.         d = a;
19.         t = 1;
20.     }
21.     return cnt;
22. }
```

程序运行结果如下：

```
PS E:\_courses\编译\Complier\PigCompiler> python main.py ./examples/func_useerr.c
[Info]Lexical analysis success!
[Error]#303 in line 12, position 19: Variable name <e> not defined.
12:         a = binary(e, f);
```

可以看出，这里本质上就是一个变量调用没有声明的错。

10) warning#201 中间代码生成器::类型错用

这里主要是 int 类型，float 类型还有 void 类型的混用问题。可以通过实例来说明。

在下面的代码中，定义了一个返回类型为 void 的函数 binary，但是这个返回值却被用来与 int 类型相加，这是不被允许的行为。

```
1.  void binary(int x, int y){
2.      return 1;
3.  }
4.
5.  int t;
6.
7.  int main(int x, int y){
8.      int a;
9.      float b;
10.     a = 0;
11.     b = 1;
12.     a = a + binary(2,0);
13.     return a;
```

```
14. }
```

实际运行结果如下：

```
PS E:\_courses\编译\Complier\PigCompiler> python main.py ./examples/func_paraerr.c
[Info]Lexical analysis success!
[Warning]#102 in line 12, position 17: Type <void> should not be used.
12:     a = a + binary(2,0);
[Info]Garmmar analysis success!
[Info]Compile success!
中间代码已输出到同级文件夹下..
```

可以看到，这样的类型冲突，不会实际影响到语义和中间代码的生成，但是从程序的角度来看显然是有问题的，因此这里选择保留了一个 warning，标识出什么地方的什么类型可能会出现问题。

另外，可以再演示一下 float 和 int 类型的转换。下面是与刚才大致相同的代码，只不过将调用的 binary 函数换成了变量 b。

```
1.  void binary(int x, int y){
2.      return 1;
3.  }
4.
5.  int t;
6.
7.  int main(int x, int y){
8.      int a;
9.      float b;
10.     a = 0;
11.     b = 1;
12.     a = a + binary(2,0);
13.     return a;
14. }
```

实际运行结果如下：

```
PS E:\_courses\编译\Complier\PigCompiler> python main.py ./examples/func_paraerr.c
[Info]Lexical analysis success!
[Warning]#101 in line 12, position 4: Potential type downgrading <float> to <int>.
12:     a = a + b;
[Info]Garmmar analysis success!
[Info]Compile success!
中间代码已输出到同级文件夹下..
```

可以看出，在处理这样的类型冲突时，最终保留的是右值的数据类型，也就是 int，这与右操作数 b 的类型 float 不符，因此在这里保留了一个 warning，说明这个地方存在从 float 到 int 的转换。

正确结果测试：

4.2 时间复杂度分析

1. 词法分析器

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

1. 语法分析器

显然，不可否定的是语法分析器的时间复杂度与出现的符号个数和所应用的产生式表的长度有关，这里先记产生式表的长度为 m，显然符号个数不会超过 m 的常数倍，也即 O(m) 个。先考虑 findSymbolFirst 函数的时间复杂度贡献，由于采用了记忆化存储的方法，对每个符号 findSymbolFirst 的有效执行次数不会超过一次，因此这部分的贡献独立于调用函数，单独调用的均摊贡献为 O(1)，总和有 O(m) 的时间复杂度贡献。

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

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

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

= O(m)。

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

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

语义分析和中间代码生成的操作是附属于语法分析系统的，在语法分析执行到规约操作后，调用语义分析模块 Analyser 并对其进行分析。在大部分的情况下，这种规约操作都是线性 O(1)的，因为大部分的规约操作都只是在抽象语法树上进行给定的属性设置和处理，不会牵涉到循环问题，下面分析下少数几个不为线性的操作。

1. 变量和函数使用时的 check 查询操作：

Check 操作需要遍历整个符号表，对变量和函数定义而言，其需要从符号表中查找当前层次下是否存在相同标识符，而对变量访问和函数调用而言，其需要从符号表中查找当前层次和全局层次下是否存在相同标识符，该操作没有索引，采用散列索引的方式执行，考虑到访问交叠的情况下平均访问性能为 O(log n)。

2. 函数调用时的调用链查询：

在函数调用中，确定用户的实参后需要完成实参向形参的转换，在这一步中就需要查询实参和形参是否一一对应，从而需要访问函数符号表的调用链部分，这种访问与函数长度相关，显然不超过符号数即 O(n)。

3. 数组调用时的 size 链查询：

在数组编写中，需要查询数组维度是否对应和数组每个维度位置声明的大小，从而需要访问符号表的数组维度大小表部分，这种访问与数组维度长度相关，

一般认为不超过 O(n)。

4. 控制语句循环调用的控制链传递：

控制链在整个控制语句当中都需要传递，在 python 中我们采用数组的方式实现，因此整个传递过程可能会产生 O(n^2)的开销，均摊到每条指令上为 O(n)的级别。

因此整体而言，每次规约操作最多会带来 O(m)的额外开销，在语法分析主程序中该循环规约的复杂度为 O(m)，因此整体复杂度为 O(n^2)，从而有整个中间代码生成器的复杂度为 O(m^3 log m + n^2)。需要注意的是，O(n^2)并不是一般程序代码生成的时间复杂度，只是在考虑到极端代码情况下的一个不严格的上界，在一般程序中占主要地位的是标识符的查询和写入操作，函数的参数个数不会很长（以一般程序员的能力而言），控制语句也不会写出 O(n)级别的嵌套调用，因此在处理一般程序的情况下，整个中间代码生成器的平均复杂度为 O(m^3 log m + n log n)。

最后的目标代码生成器中，基本块划分、待用活跃信息表设计和代码翻译只对中间代码进行遍历，平均复杂度为 O(m)，开销较少。

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

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

装 订 线

4.3 调试中遇到的一些问题

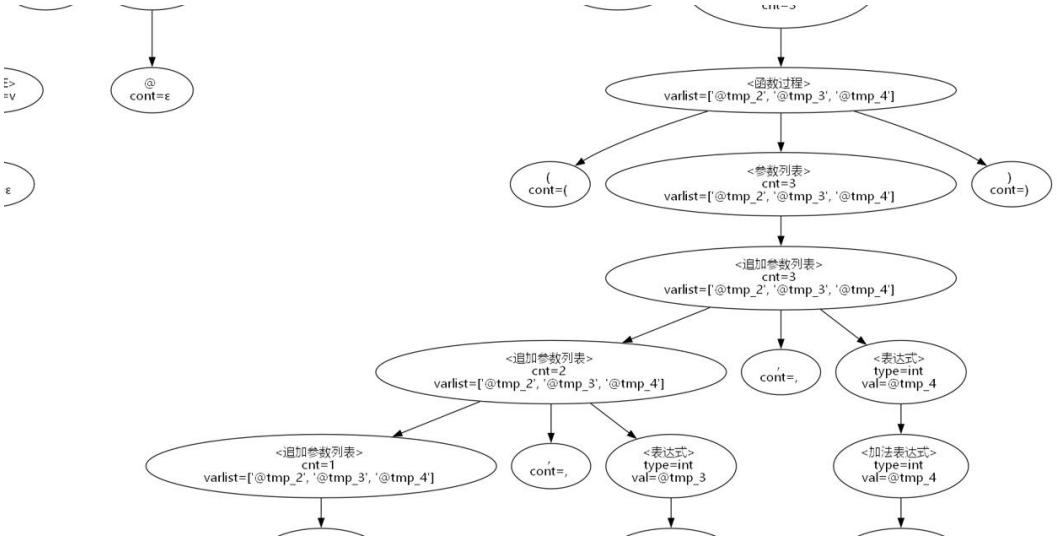
本次课程设计的实现还是比较顺利的，主要归功于在上学期进行大作业设计时较为认真，对编译器的底层原理有了较为充分的理解，这学期在其上进行重新构建、优化并添加数组调用和目标代码生成就显得十分顺利，没有碰到什么大的挫折。

主要的一些问题有两点，一是不是很清楚 python 的赋值原则所引起的调用链错误，例如分析以下代码：

```
1. int test(int a,int b,int c){
2.     return a+b+c;
3. }
4.
5. void main(void){
```

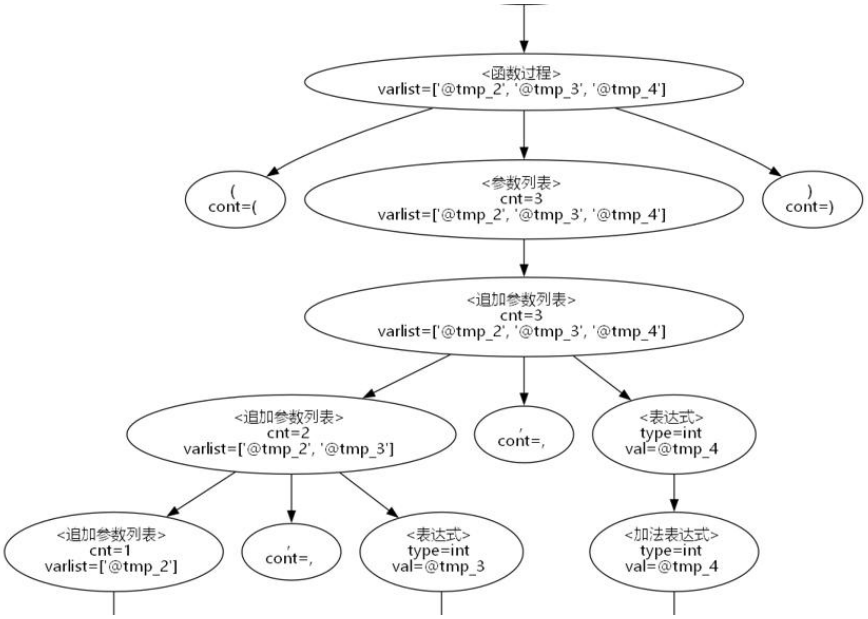
```
6.      int a;
7.      a = test(1,2,3);
8.      return;
9. }
```

解析 test 的过程中，语义树出现：



也即发现，在追加参数列表计算到 1 时，varlist 就已经获知了剩余的两个参数的位置，而这显然是不可能的，虽然这并没有影响到函数过程最终的结果，但是有时候没有体现在结果上的一些错误，我认为更值得我们警惕。

经过调查，我才发现原因来源于 python 的所有传值其实都是传引用，并且=号对数组这样的复杂变量而言默认都是浅拷贝，从而在第二个追加参数列表产生的时候，其将追加参数列表 1 与自己合并并形成自己的列表，此时列表 1、列表 2 是共享了同一个列表了。解决方法是用 python 的 deepcopy 函数在传递过程中进行修饰，最终修正后的语义图如下，输出了正确的结果：



此外，在设计寄存器分配的过程中，我之前想当然地认为有 27 个寄存器，因此对临时变量而言一定是够用的，第一次设计就没有为临时变量分配内存，并且对于所有的临时变量在寄存器清空时都一定不会将其保存到内存中。后来我才发现我把问题想简单了，因为临时变量不仅仅可能会超出 27 个寄存器的限制，并且很多临时变量在寄存器出口依旧是活跃的。

这两点造成的原因都是因为函数调用。第一点是因为虽然本次课程设计只有 5 个不同的符号优先级，理论在一条语句内最多只可能产生六个临时变量，但因为函数调用和函数递归的存在，可能一条语句会产生六个以上的函数调用栈，此时就会造成寄存器被占满。同时，由于函数递归的存在也会导致同一条计算语句产生的计算表达式会在两个甚至多个不同的基本块中间，从而需要在基本块出口存储临时变量。为此，我重构了整个符号表生成器，在其中为每一个临时变量计算并分配地址，从而实现了合理恰当的寄存器管理器。

5 用户使用说明

5.1 编译器命令行使用方式

编译器采用命令行方式使用，具体的命令可以通过 pcc -help 或 pcc -h 查看：

```
usage: main.py [-h] [-debug DEBUG] [-output OUTPUT] [-tree TREE] [-analysis_output ANALYSIS_OUTPUT] [src]

Pig Compiler for MIPSx54

positional arguments:
  src                   Input File.

optional arguments:
  -h, --help            show this help message and exit
  -debug DEBUG          Activate debug output.
  -output OUTPUT        Output File.
  -tree TREE            Show Syntax Tree.
  -analysis_output ANALYSIS_OUTPUT
                        Show Analysis Output.
```

其中 src 为可选项，若默认执行 pcc，则会默认编译同级文件夹下的 input.c 文件，想指定编译其他文件可采用 pcc <filename> 的方式进行。

其他 Optional 命令有：

-debug 1：启用 debug 输出，会将词法器的机内序列，语法器的 ACTION_GOTO 集，递归过程和其中间信息进行输出。

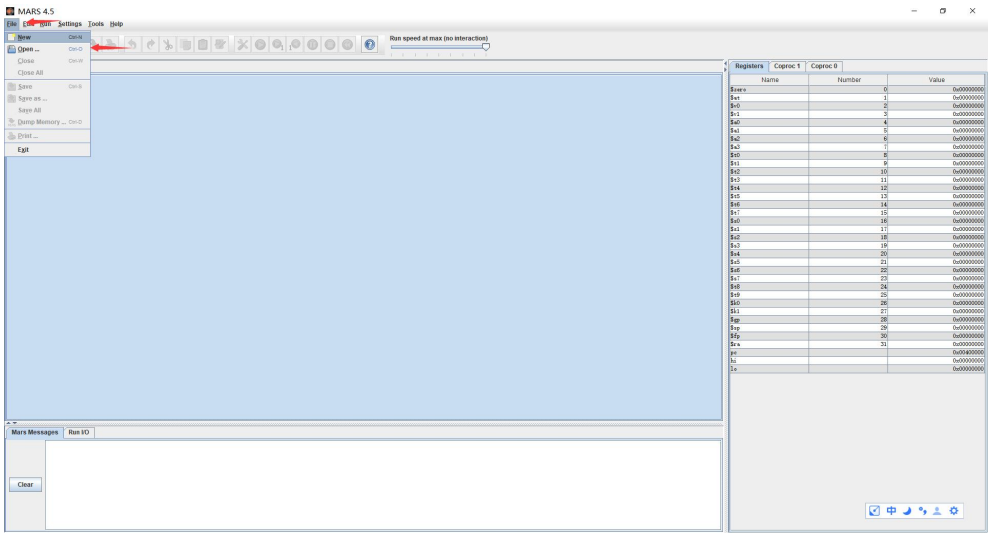
-tree 1：启用语法语义树输出，会将语法语义树生成到 grammar_tree.png 中并打开。

-output <filename> ：设定输出文件为<filename>

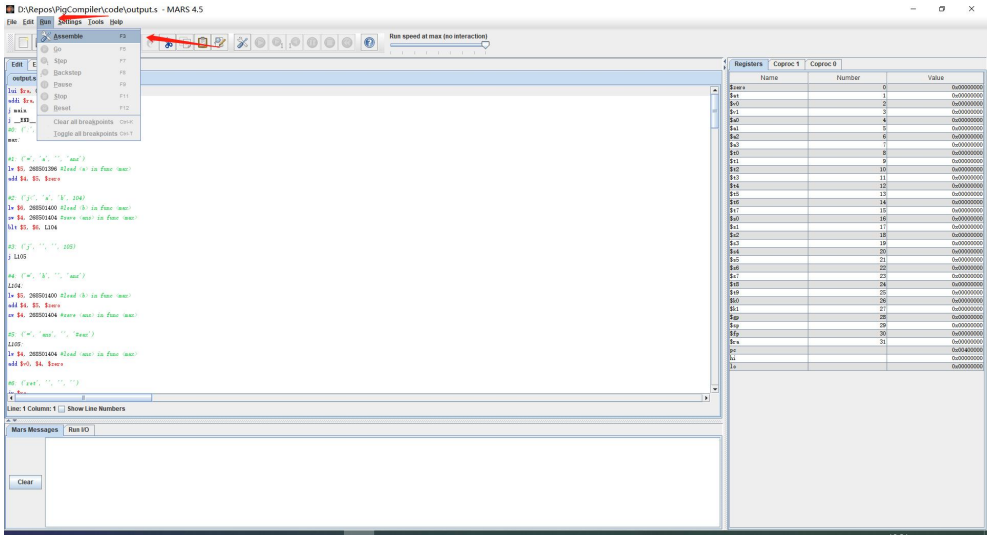
-analysis_output 1：启用语义分析输出，会将执行的每条语义指令输出。

5.2 目标代码的汇编执行与结果观察方式

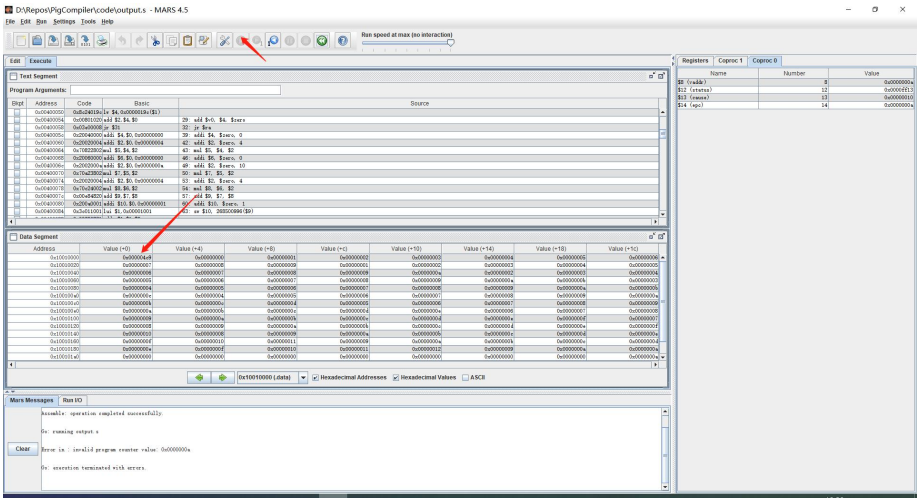
目标代码默认会保存在 Output.s 文件中，其目标指令集为 MIPS 指令集，需要下载 MARS 汇编器[3]：https://download.csdn.net/tagalbum/1712704



点击 File - Open，打开编译器生成的汇编文件。



点击 Run - Assemble，进行汇编，然后点击 Run - Go 执行：



可以从下方（内存显示）和右侧（寄存器显示）两个部分观察代码执行的结果。

5.3 要求的程序实例的翻译及其执行执行结果

要求的程序实例如下：

```
1. int program(int a,int b,int c)
2. {
3.     int i;
4.     int j;
5.     i=0;
6.     if(a>(b+c))
7.     {
8.         j=a+(b*c+1);
9.     }
10.     else
11.     {
12.         j=a;
13.     }
14.     while(i<=100)
15.     {
16.         i=j*2;
17.     }
18.     return i;
19. }
20.
21. int demo(int a)
22. {
23.     a=a+2;
24.     return a*2;
25. }
26.
27. void main(void)
28. {
29.     int a[2][2];
30.     a[0][0]=3;
31.     a[0][1]=a[0][0]+1;
32.     a[1][0]=a[0]+a[1];
33.     a[1][1]=program(a[0][0],a[0][1],demo(a[1][0]));
34.     return;
35. }
```

但是这份代码是有误的，在第 32 行：

```
a[1][0]=a[0]+a[1];
```

上，后面的 a[0]和 a[1]缺少维度，因此这里将其改为 a[0][0]和 a[0][1]；

其次在 15 行和 16 行中，

```
1. while(i<=100)
2.     {
3.         i=j*2;
4.     }
```

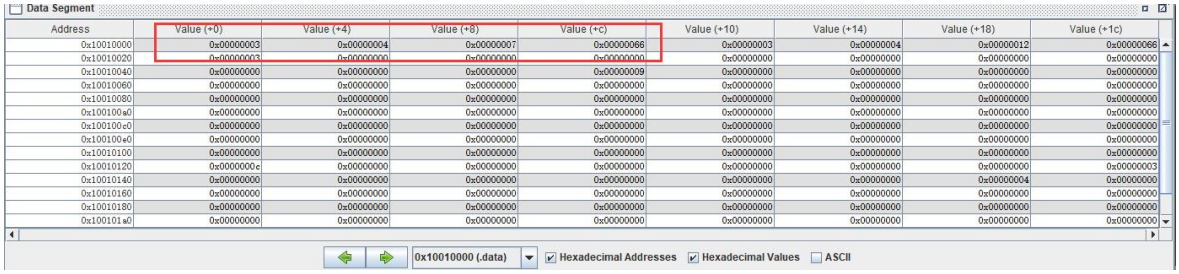
该循环为死循环，永远无法跳出，更改为 i = i + j * 2;

此外，将 a 数组变为全局变量提到最前，使得观察实验结果更方便，最终代码如下：

```
1. int a[2][2];
2. int program(int a,int b,int c)
3. {
4.     int i;
5.     int j;
6.     i=0;
7.     if(a>(b+c))
8.     {
9.         j=a+(b*c+1);
10.     }
11.     else
12.     {
13.         j=a;
14.     }
15.     while(i<=100)
16.     {
17.         i=i+j*2;
18.     }
19.     return i;
20. }
21.
22. int demo(int a)
23. {
24.     a=a+2;
25.     return a*2;
26. }
27.
28. void main(void)
29. {
30.     a[0][0]=3;
31.     a[0][1]=a[0][0]+1;
32. a[1][0]=a[0][0]+a[0][1];
33.     a[1][1]=program(a[0][0],a[0][1],demo(a[1][0]));
34.     return;
```

```
35. }
```

进行编译，并执行：



可以观察出，a[0][0] = 3，a[0][1] = 4，a[1][0] = 7，a[1][1] = 0x66 = 102，这与代码计算的结果相符，整体计算过程正确。

装 订 线

6 课程设计总结

6.1 收获、思考与感想

从上学期10月写下第一行词法分析器Lexer的代码以来也有八九个月了，这个课设项目私以为是跨度最长的一个课程设计，也是我倾注心血最长的一个项目之一，最终看着汇编出来的代码能够成功跑在MARS模拟器上的那一刻真的有一种看着自己的孩子长成大人的感觉。

编译器是一个很复杂的系统，整个实现的过程也切实让我感受到了debug的痛苦，可能还是我的工程能力不足吧，写完之后回顾之前的代码，确实还是感觉到了有很多很多地方在当初设计的时候其实是不完善的，之后也没有进行完美的修改，而是在出现问题时通过一个又一个的补丁修修改改，这台引擎出厂之时其实已经有一种“拖拉机”的感觉了（笑）。

除此以外还有一点遗憾的是没有采用同一种语法去写编译器。私以为编译器的核心在于自展，也就是能够用自己的编译器去编译出自己编译器的代码，本届也看到了一位同学朝着这个目标前进并实现了部分，心里其实是有一股不服输的冲劲的。但是考虑到C++写编译器的难度，还是采用了Python作为开发语言，算是临阵脱逃的怯弱之举吧（哭）。整体而言还是对编译器很感兴趣的，最近也有很多ai4编译的方向产生，希望有朝一日能把现在未尽的遗憾实现吧。

课设主要参考的书籍为经典的龙书[4]和我们用到的编译原理[5]教材，龙书的详细图解真的对调试和分析有很大的帮助，也感谢陈奕澄同学在最后测试中寄存器管理器出错的时候帮我一条一条语句查询并进行故障定位，以及陈泓仰学长在课设开始之初给我了诸多建议，少走了很多弯路，节省了大量时间。

7 参考文献

[1] Graphviz 的使用指南[EB/OL] 2020-1, https://blog.csdn.net/a6661314/article/details/122656342.

[2] MIPS 汇编详细指令[EB/OL] 2017-5, https://blog.csdn.net/wxc971231/article/details/108032595

[3] Mars 的使用[EB/OL]. 2019-2, https://blog.csdn.net/weixin_41863129/article/details/83069288

[4] 编译系统透视 图解编译原理[B]. 新设计团队著. ISBN: 9787111498582

[5] 编译原理（第三版）[B]陈意云，张昱，高等教育出版社. ISBN：9787040178128