

---

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 编译器设计说明书

学号姓名 1751740 刘鲲

专 业 计算机科学与技术

授课老师 丁志军

日 期 2020/05/09

---

## 目 录

1.	课程设计重述 .....	3
1.1.	目的 .....	3
1.2.	要求 .....	4
2.	需求分析 .....	4
2.1.	任务输入及其范围 .....	4
2.2.	输出形式 .....	5
2.2.1.	输出中间代码表示的程序 .....	5
2.2.2.	输出目标代码(可汇编执行)的程序 .....	5
2.3.	程序功能 .....	6
2.4.	测试数据 .....	6
2.4.1.	变量重定义 .....	6
2.4.2.	使用未声明的变量 .....	6
2.4.3.	使用未定义的函数 .....	7
2.4.4.	变量赋值时类型错误 .....	7
2.4.5.	函数形参和实参不匹配 .....	7
2.4.6.	寄存器是否正常选用 .....	7
3.	概要设计 .....	8
3.1.	任务的分解 & 数据类型的定义 .....	8
3.1.1.	语法文件读取与解析, 词法分析 – CFG 类 .....	8
3.1.2.	构建项目集规范族 – ItemSetSpecificationFamily 类 .....	9
3.1.3.	语法、语义、中间代码生成 – SyntacticAnalyzer 类 .....	10
3.1.4.	目标代码生成 – ObjCodeGenerator 类 .....	10
3.2.	主程序流程 .....	11
3.3.	模块间的调用关系 .....	11
4.	详细设计 .....	12
4.1.	词法分析 .....	12
4.1.1.	实现思路 .....	12
4.1.2.	去除注释后进行词法单元的识别 .....	12
4.1.3.	词法规则的设计和读取 .....	14
4.2.	LR1 语法分析设计 .....	17
4.2.1.	LR(1)原理 .....	17
4.2.2.	模块设计与分析 .....	18
4.3.	语义分析及中间代码生成设计 .....	22
4.3.1.	S 属性文法及自底向上扫描原理 .....	22
4.3.2.	更改为 S 属性文法 .....	23
4.3.3.	三地址代码和四元式 .....	23

---

4.3.4. 具体语句的语义规则 .....	24
4.3.5. 模块设计 .....	27
4.4. 目标代码生成设计 .....	29
4.5. 函数调用的中间代码生成和目标代码生成 .....	32
5. 调试分析 .....	35
5.1. 语法分析测试 .....	35
5.1.1. 时间复杂度分析 .....	36
5.1.2. 存在的问题与思考 .....	36
5.2. 静态语义测试 .....	36
5.2.1. 变量重定义 .....	36
5.2.2. 使用未声明的变量 .....	37
5.2.3. 使用未定义的函数 .....	37
5.2.4. 变量赋值时类型错误 .....	38
5.2.5. 函数形参和实参不匹配 .....	39
5.2.6. 时间复杂度分析 .....	40
5.2.7. 存在的问题与思考 .....	40
5.3. 目标代码生成测试 .....	40
5.3.1. 寄存器取用测试 .....	40
5.3.2. 复杂度分析 .....	42
5.4. 存在的问题，思考与解决 .....	42
5.4.1. 空串处理 .....	42
5.4.2. 判断是否该元素已存在该集合中 .....	43
5.4.3. 单个非终结符的 First 集的并发求解 .....	43
5.4.4. 语法更改，消除需要继承属性的语义规则 .....	44
6. 用户使用说明 .....	45
7. 课程设计总结 .....	50
8. 附录 .....	51
9. 参考文献 .....	51

## 1. 课程设计重述

### 1.1. 目的

掌握使用高级程序语言实现一个一遍完成的、简单语言的编译器的方法；  
掌握简单的词法分析器、语法分析器、符号表管理、中间代码生成以及目标代码生成的实现方法；  
掌握将生成代码写入文件的技术。

## 1.2. 要求

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。

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；

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。

要求输入类 C 语言源程序，输出中间代码表示的程序；

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。

实现过程、函数调用的代码编译

## 2. 需求分析

### 2.1. 任务输入及其范围

输入上限：一段带过程调用的代码段。

输入下限：符合语法规则的语句。

一段带过程调用的代码段实例如下：

```
1.  int a;
2.  int b;
3.  int program(int a,int b,int c)
4.  {
5.  int i;
6.  int j;
7.  i=0;
8.  if(a>(b+c))
9.  {
10.  j=a+(b*c+1);
11. }
12. else
13. {
14.  j=a;
15. }
16. while(i<=100)
17. {
18.  i=j*2;
19. }
20. return i;
21.}
22.
23.int demo(int a)
```

```

24.{
25. a=a+2;
26. return a*2;
27.}
28.
29.void main(void)
30.{
31. int a;
32. int b;
33. int c;
34. a=3;
35. b=4;
36. c=2;
37. a=program(a,b,demo(c));
38. return ;
39. }

```

## 2.2. 输出形式

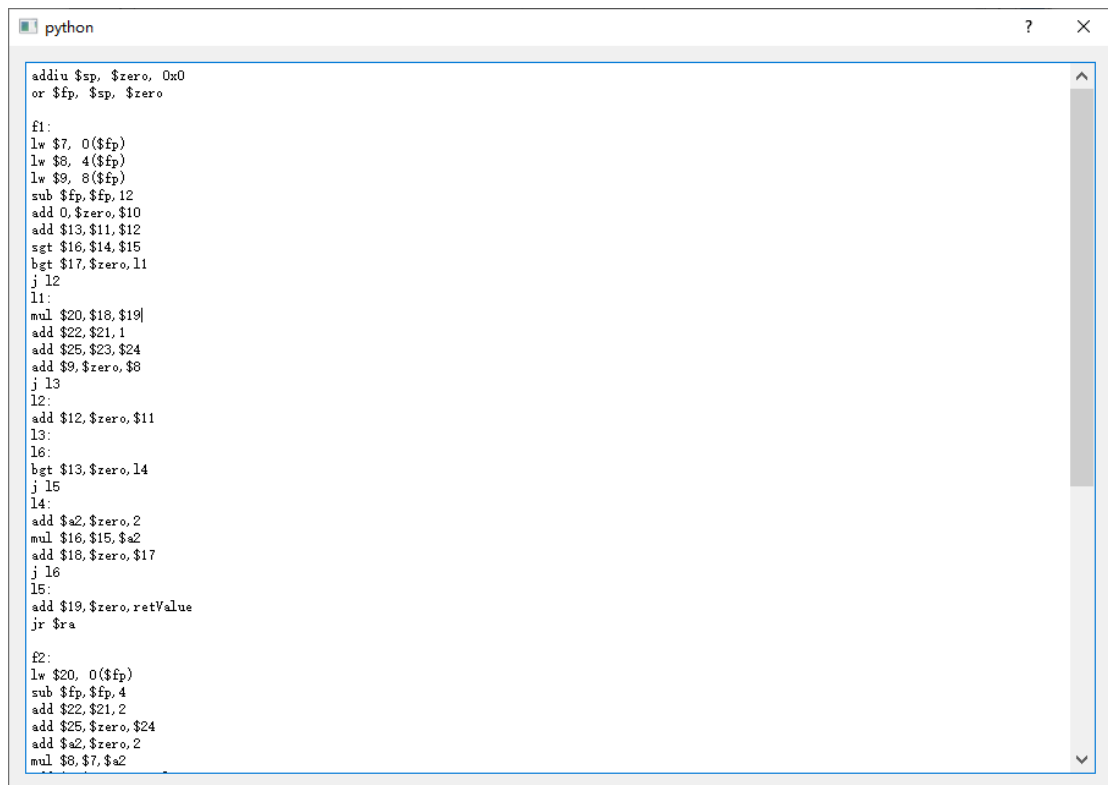
### 2.2.1. 输出中间代码表示的程序

示例如下：

	operation	arg1	arg2	result
1	f1	:	-	-
2	pop	-		t3
3	pop	-		t4
4	pop	-		t5
5	-	fp		fp
6	:=	0	-	t6
7	+	t4	t5	t8
8	>	t3	t8	t12
9	j>	t12	0	l1
10	j	-	-	l2
11	l1	:	-	-
12	*	t4	t5	t9
13	+	t9	1	t10
14	+	t3	t10	t11
15	:=	t11	-	t7

### 2.2.2. 输出目标代码(可汇编执行)的程序

示例如下：

A screenshot of a window titled 'python' containing MIPS assembly code. The code includes instructions for stack frame setup, function calls (f1, f2), arithmetic operations (add, sub, mul, sgt, bgt), and control flow (j, jr).

```
python
addiu $sp, $zero, 0x0
or $fp, $sp, $zero

f1:
lw $7, 0($fp)
lw $8, 4($fp)
lw $9, 8($fp)
sub $fp, $fp, 12
add 0, $zero, $10
add $13, $11, $12
sgt $16, $14, $15
bgt $17, $zero, 11
j 12
11:
mul $20, $18, $19
add $22, $21, 1
add $25, $23, $24
add $9, $zero, $8
j 13
12:
add $12, $zero, $11
13:
16:
bgt $13, $zero, 14
j 15
14:
add $a2, $zero, 2
mul $16, $15, $a2
add $18, $zero, $17
j 16
15:
add $19, $zero, retValue
jr $ra

f2:
lw $20, 0($fp)
sub $fp, $fp, 4
add $22, $21, 2
add $25, $zero, $24
add $a2, $zero, 2
mul $8, $7, $a2
```

## 2.3. 程序功能

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。

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。

要求输入类 C 语言源程序，输出中间代码表示的程序。

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。

实现过程、函数调用的代码编译。

## 2.4. 测试数据

### 2.4.1. 变量重定义

```
int a;
int a;
```

### 2.4.2. 使用未声明的变量

```
int main()
{
    a = 1;
}
```

---

#### 2.4.3. 使用未定义的函数

```
int main()
{
    int a;
    a = demo(1);
    return ;
}
```

#### 2.4.4. 变量赋值时类型错误

```
int main()
{
    int a;
    a = 1;
    void b;
    b = a;

    return 1;
}
```

#### 2.4.5. 函数形参和实参不匹配

```
int demo()
{
    return 1;
}
```

```
int main()
{
    int c;
    c = demo(1);
    return 1;
}
```

#### 2.4.6. 寄存器是否正常选用

```
int main(void)
{
    int a;int b;int c;int d;int e;int f;int g;int h;int i;int j;int k;int l;int m;int n;int o;
```

---

```
int p;int q;int r;int s;int t;int u;
```

```
a=0;b=1;c=2;d=3;e=4;f=5;g=6;h=7;i=8;j=9;k=10;l=11;m=12;n=13;o=14;p=15;q=16;r=17;s=18;t=19; u=20;
```

```
a=0;b=1;c=2;d=3;e=4;f=5;g=6;h=7;i=8;j=9;k=10;l=11;m=12;n=13;o=14;p=15;q=16;r=17;s=18;t=19;
```

```
return 1;
}
```

### 3. 概要设计

#### 3.1. 任务的分解 & 数据类型的定义

**注：根据任务设计书，这两点是分来的，但是如果是合并在一起说明更具连贯性。特此说明。**

一般来说，一个编译器主要有五个步骤：

词法分析

语法分析

语义分析

中间代码生成

目标代码生成

根据要求，一遍实现的编译器，LR(1)是适用的。在语法分析的过程中，语义分析、中间代码生成也在同步进行。故这三个部分可以合并。

同时，使用面向对象的方法，将每个阶段看作一个对象，对该部分的操作就是该对象的方法。这大大方便了程序的开发和维护。

另外还有 GUI。这部分和 PyQt 相关。

而对于每一部分来说，又需要不同的数据结构来维护，接下来将详细说明。

##### 3.1.1. 语法文件读取与解析，词法分析 – CFG 类

设置 CFG 类，做两件事：

1. 读取语法文件，进行解析，获取起始符号、终结符、变元、产生式，设置广义起始符，生成 LR1 需要的加点产生式，也就是项目 item
2. 对输入代码使用正则识别，生成 token 流

成员名称及类型	描述
---------	----



TerminalSymbols = []	终结符
StartSymbol	广义起始符
OriginStartSymbol	原语法的起始符
NonTerminalSymbols = []	变元
Reserved = {}	保留字
Items = []	加点的项目

方法的名称	功能描述
loadGrammar	读取文法
calFirstSet	计算 First 集
calNTFirstSet	计算非终结符的 First 集
getDotItems	将 item 加点
generateTokens	生成 tokens 流
scanLine	扫描代码的每一行，获取 token

模块设计思路与分析说明：

这一部分是对语法的处理。我们知道，LR(1)语法中的项目（Item）是带点的，读入产生式后，getDotItems 给所有产生式的所有位置加点。

对于单个所有符号（包括终结符和非终结符）都有 First 集。在之后的语法分析步骤中，需要计算一个字符串的 First 集，只需要根据规则调用单个字符的 First 集计算即可。“cal”指的是 calculate，计算，calFirstSet 将单个字符的 First 集结果保存供后续分析使用。

### 3.1.2. 构建项目集规范族 – ItemSetSpecificationFamily 类

模块设计思路与分析说明：

核心函数是 getLR1Closure，GO 和 buildFamily，三者通过 LR1 的算法构建项目集族的 DFA。

ItemSetSpecificationFamily 中的属性和方法描述如下：

方法的名称	功能描述
getLeftNT	获取某个非终结符的产生式
getLR1Closure	根据 LR1 的方法算 Closure 集
GO	状态转移函数
edge2str	将状态转移的边转为 string 方便比较
getFirstSet	获取字符串的 First 集
extendItem	根据 LR1 文法，将 item 进行终结符拓展
buildFamily	通过算法构建项目集族

数据成员描述如下：

itemSets = []	项目集，也就是 DFA 的状态
prods=[]	项目集规范族的 DFA 的产生式
其它来自 CFG 类的语法数据	

### 3.1.3. 语法、语义、中间代码生成 – SyntacticAnalyzer 类

模块设计思路与分析说明：

核心函数是 getTables，即通过 ItemSetSpecificationFamily 中构建的 DFA 生成 ACTION 和 GOTO 表。通过这两张表就能对任意字符串给出是否符合 LR1 文法的判断。

核心函数是 isRecognizable，其中又有 semanticAnalyze 作语义分析。

SyntacticAnalyzer 中的属性和方法描述如下：

方法的名称	功能描述
item2prodIdx	给一个 item，返回该项目对应的产生式编号
getTables	计算 ACTION 和 GOTO 数组
isRecognizable	判断一个字符串是否能被识别
semanticAnalyze	语义分析

成员名称	描述
ACTION[s, a]	当状态 s 面临输入符号 a 时，应采取什么动作
GOTO[s, X]	状态 s 面对文法符号 X 时，下一状态是什么
symbolTable	符号表
funcTable	函数表
middleCode	中间代码

### 3.1.4. 目标代码生成 – ObjCodeGenerator 类

模块设计思路与分析说明：

获取中间代码和其它语义分析结果，生成目标代码

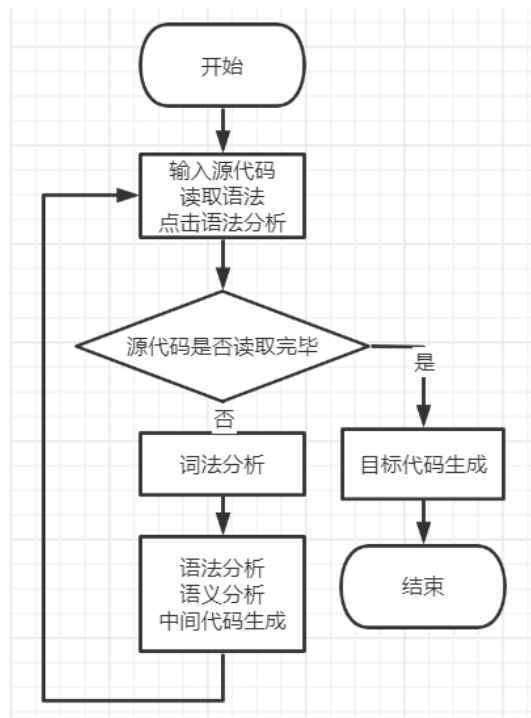
方法名称	描述
getRegister	获取一个寄存器
freeRegister	释放寄存器
genMips	生成 mips 代码

属性成员

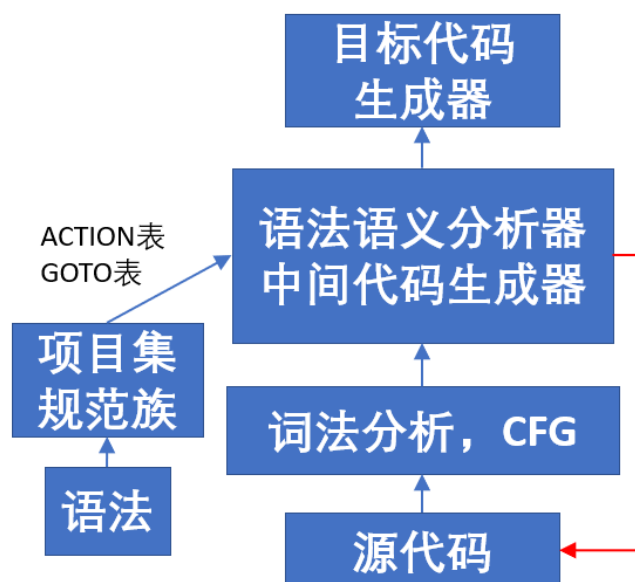
属性名	描述
mipsCode	生成的 Mips 代码

regTable	记录此时寄存器内部存的是哪个变量的值
varStatus	记录变量此时是在寄存器当中还是 memory

### 3.2. 主程序流程



### 3.3. 模块间的调用关系



---

## 4. 详细设计

要按照写程序的规则来编写。要结构清晰，重点函数的重点变量，重点功能部分要加上清晰的程序注释。画出函数调用图。

### 4.1. 词法分析

#### 4.1.1. 实现思路

词法分析部分对词法符号的识别主要是通过 python 提供的正则表达式匹配功能。首先通过如下所示的 `remove_comments()` 函数来先将源代码中的注释去除。

```
import re
def remove_comments(text):#去除注释
    comments = re.findall('//.*?\n', text, flags=re.DOTALL)
    if(len(comments)>0):
        for comment in comments:
            text=text.replace(comment, "")
    comments = re.findall('/\s*.*?\/', text, flags=re.DOTALL)
    if(len(comments)>0):
        for comment in comments:
            text=text.replace(comment, "")
    return text
```

我主要通过 “`//.*?\n`” 正则表达式调用 `re.findall()` 函数来找到所有匹配的注释行，通过 `re.replace()` 函数将找到的注释行消去。通过 “`/\s*.*?\/`” 正则表达式来匹配所有的注释段，用同样的替换将其消除。在这里需要注意的是用 `.*` 来匹配字符时需要加上 `?` 来进行非贪婪的匹配，否则如果有多个注释行匹配出的结果可能错误。

例如贪婪匹配用不加 `?` 的正则表达式匹配 “`//123\n int i=0\n`” 时的结果会覆盖整个字符串。

```
import re
def remove_comments(text):#去除注释
    comments = re.findall('//.*\n', text, flags=re.DOTALL)
    if(len(comments)>0):
        for comment in comments:
            text=text.replace(comment, "")
    comments = re.findall('/\s*.*\/', text, flags=re.DOTALL)
    if(len(comments)>0):
        for comment in comments:
            text=text.replace(comment, "")
    return text
text="//123456\n//1325\n/*56*/5466/*59*/"
remove_comments(text)
```

上图删除注释后的理想结果是 5466，但由于进行的是贪婪匹配，结果 `/**/` 匹配时将 5466 涉及了进去。所以应该进行非贪婪匹配。

#### 4.1.2. 去除注释后进行词法单元的识别

如下在 `regexs` 中构建四个正则表达式来分别对四类标识符：界符、操作符、标识

符、整数进行匹配。

```
type=[
    'separator', 'operator', 'identifier', 'int'
]#类别

regexs=[
    '\{|\}|\[|\]|\\(|\\)|,|;' #界符
    ,'\+|-|\*|/|=|!|=|>|=|<|=|<|=' #操作符
    ,'[a-zA-Z][a-zA-Z0-9]*' #标识符
    ,'\d+' #整数
]#词法分析所使用的正则表达式
```

具体实践上利用 python 文件操作 `open()` 和 `read` 函数读取源代码，通过 `split('\n')` 将源代码逐行分开，逐行识别词法单元。

```
def generate_tokens(path):
    fd=open(path, 'r')
    lines=remove_comments(fd.read()).split('\n')
```

定义了 `scan_line(line)` 函数将一行的词法单元识别，并将判别词法单元的结果存储到一个 list 作为返回结果中。定义 `scan(result)` 来读取并判别 `result` 的第一个词法单元。

首先通过 `strip` 函数将一行字符串开头的空格和制表符去除。

```
result = line.strip().strip('\t')
```

然后通过 `re.match()` 函数进行将 `regexs` 中的 4 个正则表达式逐个匹配 `result` 的开头，识别出关键词的类型。

```
for regex in regexs:
    result=re.match(regex, line)
    if(result):
        result=result.group(0)
```

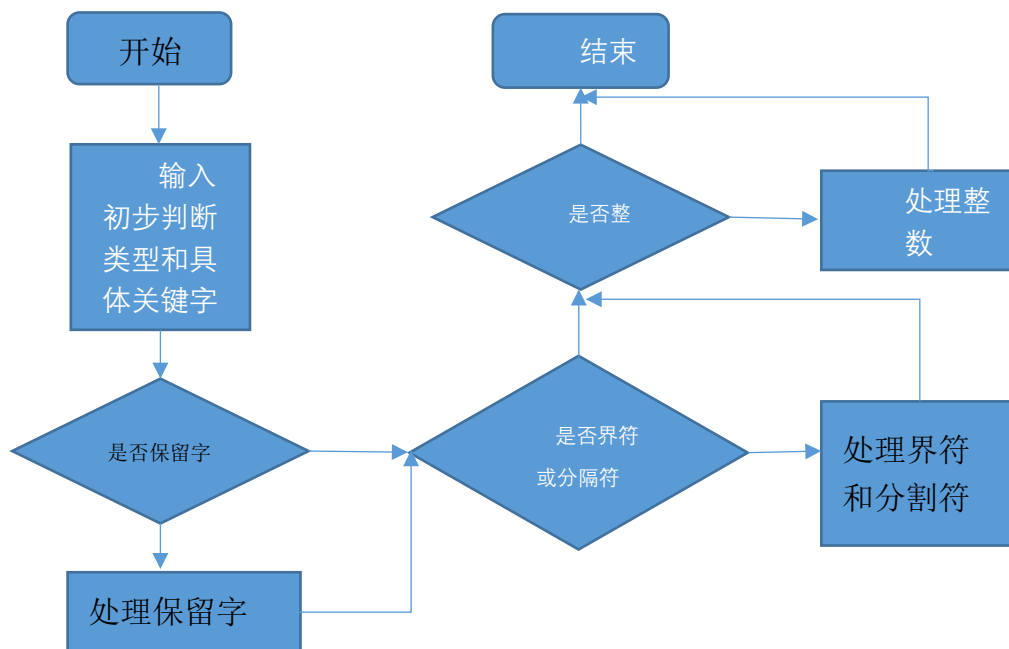
若匹配成功，返回匹配的具体关键词和初步判断该关键词的类型，和剩下的还未检索的字符串部分，分别存储在 “data” “regex” “remain” 下

```
if(match==False):#出错处理
    print(u"非法字符: "+line[0])
    return {"data":line[0], "regex":None, "remain":line[1:]}
else:
    return {"data":max, "regex":target_regex, "remain":line[index_sub+len(max):]}
```

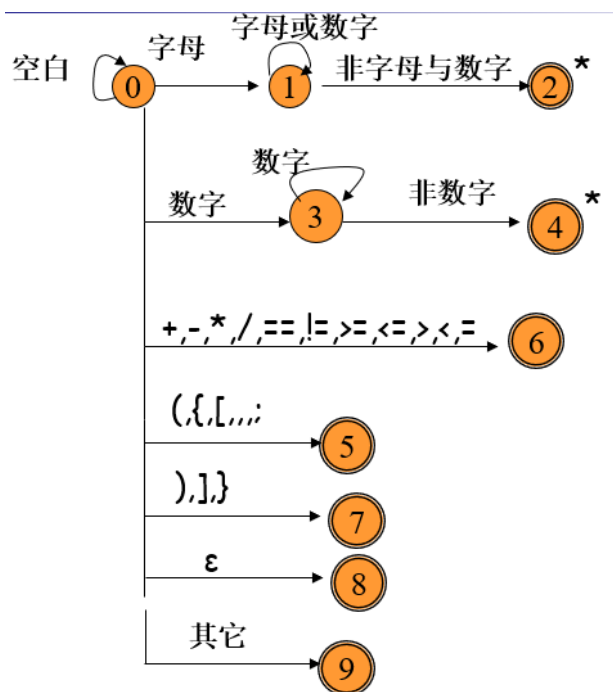
在 `scan_line` 中循环调用 `scan` 函数，可以识别出一行字符串的所有关键词和它们的类型。

随后，将根据 `scan` 初步划分的关键词类型来进一步划分关键词的类型。

状态转换图如下：



其对应的状态转换图如下：



（由老师第三章课件改编而成）

主要流程如下：

判断是否属于保留字→判断是否属于运算符或界符类型→判断是否属于整数类型

#### 4.1.3. 词法规则的设计和读取

参考 PPT 如下，进行了类 C 语法规则的设计和读取，为接下来的语法分析工作做准备。

## 类C语法规则——包含过程调用

- Program ::= <声明串>
- <声明串> ::= <声明>{<声明>}
- <声明> ::= int <ID> <声明类型> | void <ID> <函数声明>
- <声明类型> ::= <变量声明> | <函数声明>
- <变量声明> ::= ;
- <函数声明> ::= '(<形参>)' <语句块>
- <形参> ::= <参数列表> | void
- <参数列表> ::= <参数> {, <参数>}
- <参数> ::= int <ID>
- <语句块> ::= '{<内部声明> <语句串>}'
- <内部声明> ::= 空 | <内部变量声明>; <内部变量声明>}
- <内部变量声明> ::= int <ID>
- <语句串> ::= <语句>{ <语句> }

## 类C语法规则——包含过程调用

- <语句> ::= <if语句> | <while语句> | <return语句> | <赋值语句>
- <赋值语句> ::= <ID> = <表达式>;
- <return语句> ::= return [ <表达式> ] (注: [ ] 中的项表示可选)
- <while语句> ::= while '(<表达式> ' <语句块>
- <if语句> ::= if '(<表达式>)' <语句块> [ else <语句块> ] (注: [ ] 中的项表示可选)
- <表达式> ::= <加法表达式> { relop <加法表达式> } (注: relop-> <|<=|>|=|!=|)
- <加法表达式> ::= <项> {+ <项> | -<项>}
- <项> ::= <因子> { ' <因子> | /<因子> }
- <因子> ::= num | '(<表达式>)' | <ID> FTYPE
- FTYPE ::= <call> | 空
- <call> ::= '(<实参列表> )'
- <实参> ::= <实参列表> | 空
- <实参列表> ::= <表达式> {, <表达式> }
- <ID> ::= 字母(字母|d数字)\*

将语法规则按照一定规则存储在 grammer\_final.txt 中

```

program
    declarationChain
    #
declarationChain
    declaration
    declaration declarationChain
    #
declaration
    int id declareType
    void id declareFunction
    #
declareType
    ;
    declaraFunction
    #
declareFunction
    ( formal ) block
    ( ) block
    #
formal
    paraList
    void
    #

```

其中每一个推导式的左边非终结符在记录前一行开头，右边推导结果记录在下一行，且以\t制表符开始，每一种可能的右端推导结果记录在不同行。以#结束该推导式，用\$表示空。不同关键词间必须间隔空格。

编写如下函数进行语法规则推导式的读取：

```

while 1:
    line=fd.readline().replace('\n','')
    if not line:
        break
    token1=[]
    token3=[]
    token1.append({'type':line,'class':'NT','name':line})
    while 1:
        token2=[]
        line=fd.readline().replace('\n','')
        if not line:
            break
        if line[0]!='\t':
            line=line.strip('\t').split(' ')
            if line[0]!='#':
                tokens.append({'left':token1,'right':token3})
                break
        for item in line:
            match=0
            for regex in regexs[0:2]:
                result=re.match(regex,item)
                if result:
                    match=1
                    break
            if(match==1):
                token2.append({'type':type[regexs.index(regex)].upper(),'class':'T','name':item})
            elif(item in reserved):
                token2.append({'type':item,'class':'T','name':item})
            elif(item=='id'):
                token2.append({'type':'IDENTIFIER','class':'T','name':'IDENTIFIER'})
            elif(item=='null'):
                token2.append({'type':item,'class':'T','name':item})
            else:
                token2.append({'type':item,'class':'NT','name':item})
        token3.append(token2)

```



---

首先读取推导式的左端，记录其 **type** 和 **name** 相等，为具体的非终结符，类别 **class** 记录是否是终结符。将以上三个属性按照对应的索引存储在字典中，由 **token1** 存储该字典。随后读取推导式的右端分支，在 **token2** 中按顺序存储一个推导分支的终结符和非终结符，在 **token3** 中存储推导式右端的各个分支。需要注意的是，在判断关键词的类型时，要仿照词法分析判断关键词是否属于保留字或 **identifier**，将其 **class** 设置为 **T**，表示终结符。

## 4.2. LR1 语法分析设计

### 4.2.1. LR(1)原理

#### LR(1)文法定义

**LR 文法**: 对于一个文法，如果能够构造一张 **LR** 分析表，使得它的每个入口均是唯一确定，则该文法称为 **LR** 文法。在进行自下而上分析时，一旦栈顶形成句柄，即可归约。

**LR(k)文法**: 对于一个文法，如果每步至多向前检查 **k** 个输入符号，就能用 **LR** 分析器进行分析。则这个文法就称为 **LR(k)**文法。

大多数适用的程序设计语言的文法不能满足 **LR(0)** 文法的条件，因此使用 **LR(0)** 规范族中冲突的项目集（状态）用向前查看一个符号的办法进行处理，以解决冲突，即 **LR(1)**。

#### LR(0)项目集规范族

构成识别一个文法活前缀的 **DFA** 的项目集（状态）的全体称为文法的 **LR(0)**项目集规范族。

#### 项目集 **I** 的闭包 **CLOSURE(I)**

- (1) **I** 中的所有项目都属于 **CLOSURE(I)**;
- (2) 若项目  $[A \rightarrow a.B\beta, a]$  属于 **CLOSURE(I)**，则对于任何  $B \rightarrow \xi$ ，以及 **FIRST** $\langle \beta a \rangle$  中的每一个终结符 **b**，项目  $[\beta \rightarrow \cdot \xi, b]$  也属于 **CLOSURE(I)**;
- (3) 重复执行 (1) (2) 直至 **CLOSURE(I)** 不再增大为止。

#### 状态转换函数 **GO(I, X)**

**GO**: 状态转换函数，**I**: 项目集，**X**: 文法符号。

**GO(I, X) = CLOSURE(J)**,

其中  $J = \{ \text{任何形如 } A \rightarrow \alpha X \cdot \beta \text{ 的项目} \mid A \rightarrow \alpha \cdot X \beta \text{ 属于 } I \}$ 。

## LR(1)项目集族的构建

初始项目集： $I_0$

从  $I_0$  开始，对于所有项目集  $I$ ，对于  $I$  的每个项目  $X$ ，求  $I' = GO(I, X)$ ，若  $I'$  之前不曾出现过：

1. 则将  $I'$  加入项目集族中，并添加为 DFA 新的状态
2. 为 DFA 添加一条边  $(I, X, I')$

循环此操作直到项目集族不再增大为止，此时获得了一个 DFA，即代表了文法  $G$  的 LR(1) 项目集族。

## LR(1)预测表的构建

- (1) 若项目  $[A \rightarrow \cdot a, b]$  属于  $I_k$  且  $GO(I_k, a) = I_j$ ， $a$  为终结符，则置  $ACTION[k, a]$  为 “sj”。
- (2) 若项目  $[A \rightarrow \cdot a]$  属于  $I_k$ ，则置  $ACTION[k, a]$  为 “rj”；其中假定  $A \rightarrow$  为文法  $G$  的第  $j$  个产生式。
- (3) 若项目  $[S \rightarrow S \cdot, \#]$  属于  $I_k$ ，则置  $ACTION[k, \#]$  为 “acc”。
- (4) 若  $GO(I_k, A) = I_j$ ，则置  $GOTO[k, A] = j$ 。
- (5) 分析表中凡不能用规则 1 至 4 填入信息的空白栏均填上 “出错标志”。

## 字符串识别

三元式（栈内状态序列，移进归约串，输入串）的变化：

开始：（  $S_0$ ,  $\#$ ,  $a_1 a_2 \dots \#$  ）

某一步：（ $S_0 S_1 \dots S_m$ ,  $\#X_1 X_2 \dots X_m$ ,  $a_i a_{i+1} \dots a_n \#$ ）

下一步： $ACTION[S_m, a_i]$

1. 若  $ACTION[S_m, a_i]$  为 “移进” 且  $GOTO[S_m, a_i] = S$ ，则三元式为：  
（ $S_0 S_1 \dots S_m S$ ,  $\#X_1 X_2 \dots X_m a_i$ ,  $a_{i+1} \dots a_n \#$ ）
2. 若  $ACTION[S_m, a_i]$  为 “归约”  $\{A \rightarrow \beta\}$ ，且  $|\beta| = r$ ， $\beta = X_{m-r+1} \dots X_m$ ， $GOTO[S_{m-r}, A] = S$ ，则三元式为：  
（ $S_0 S_1 \dots S_{m-r} S$ ,  $\#X_1 X_2 \dots X_{m-r} A$ ,  $a_i a_{i+1} \dots a_n \#$ ）
3. 若  $ACTION[S_m, a_i]$  为 “接受” 则结束
4. 若  $ACTION[S_m, a_i]$  为 “报错” 则进行出错处理

### 4.2.2. 模块设计与分析

Item 中的属性和方法描述如下:

方法的名称	功能描述
Self.right	产生式右部
Self.left	产生式左部
Self.dotPos	点的位置
Self.terms	终结符串
ToString	将这个 item 转为字符串
NextItem	将点向后移动一位

模块设计思路与分析说明:

LR(1)中的 1 就体现在 self.terms 中, 本质是往前看一位; self.dotPos 是 int 型, 指明项目的点在哪里。ToString 是将 item 转为一串字符方便比较; nextItem 是当读入某个字符后, 点就要随之向后移动。

CFG 中的方法描述如下:

方法的名称	功能描述
loadGrammer	读取文法
calFirstSet	计算 First 集
calNTFirstSet	计算非终结符的 First 集
getDotItems	将 item 加点

模块设计思路与分析说明:

这一部分是对语法的处理。我们知道, LR(1)语法中的项目 (Item) 是带点的, 读入产生式后, getDotItems 给所有产生式的所有位置加点。

对于单个所有符号 (包括终结符和非终结符) 都有 First 集。在之后的语法分析步骤中, 需要计算一个字符串的 First 集, 只需要根据规则调用单个字符的 First 集计算即可。“cal”指的是 calculate, 计算, calFirstSet 将单个字符的 First 集结果保存供后续分析使用。

核心的一个计算 First 集函数如下:

```
1. def calNTFirstSetImprove(self, symbol):
2.     eps = {'class': 'T', 'name': '', 'type': self.Epsilon}
3.     # 若  $X \in VT$ , 则  $FIRST(X) = \{X\}$ 。
4.     hasEpsAllBefore = -1
5.     prods = [prod for prod in self.prods if prod.left == symbol]
6.     if len(prods) == 0:
7.         return
8.
9.     is_add = 1
10.    while(is_add):      # 必须!
11.        is_add = 0
```

```

12.         for prod in prods:
13.             hasEpsAllBefore = 0 # state 0
14.
15.             for right in prod.right:
16.                 # 2. 若  $X \in VN$ , 且有产生式  $X \rightarrow a...$ ,  $a \in VT$ ,
17.                 # 则  $a \in FIRST(X)$   $X \rightarrow \epsilon$ , 则  $\epsilon \in FIRST(X)$ 
18.                 if right['class'] == 'T' or \
19.                     (right['type'] == self.Epsilon and len(prod.right) == 1): #不是
#随便加的那种 eps, 即  $A \rightarrow \epsilon$ 
20.                     #有就加
21.                     if right['type'] not in self.firstSet[symbol]:
22.                         self.firstSet[symbol].append(right['type'])
23.                         is_add = 1
24.
25.                     break
26.
27.                 # 3. 对 NT
28.                 # 之前已算出来过
29.                 # 但有可能是算到一半的
30.                 if len(self.firstSet[right['type']]) == 0:
31.                     if right['type'] != symbol: #防止陷入死循环
32.                         self.calNTFirstSetImprove(right['type'])
33.
34.                 #  $X \rightarrow Y...$  是一个产生式且  $Y \in VN$  则把  $FIRST(Y)$  中的所有非空符号串  $\epsilon$  元素都加入
#到  $FIRST(X)$  中。
35.                 if self.Epsilon in self.firstSet[right['type']]:
36.                     # 状态机
37.                     if hasEpsAllBefore == 1:
38.                         hasEpsAllBefore = 1
39.                     elif hasEpsAllBefore == 0:
40.                         hasEpsAllBefore = 1
41.
42.                 for f in self.firstSet[right['type']]:
43.                     if f != self.Epsilon and f not in self.firstSet[symbol]:
44.                         self.firstSet[symbol].append(f)
45.                         is_add = 1
46.
47.                 # 到这里说明整个产生式已遍历完毕 看是否有始终能推出 eps
48.                 # 中途不能退出 eps 的已经 break 了
49.                 # 所有 right(即  $Y_i$ ) 能够推导出  $\epsilon$ , ( $i=1,2,...n$ ), 则
50.                 if hasEpsAllBefore == 1:
51.                     if self.Epsilon not in self.firstSet[symbol]:
52.                         self.firstSet[symbol].append(self.Epsilon)
53.                         is_add = 1

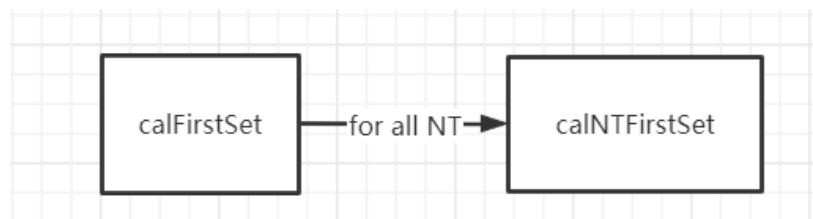
```

54.

55. `return`

具体说明在注释里很详尽。

这一部分的函数调用图如下：



ItemSetSpecificationFamily 中的属性和方法描述如下：

方法的名称	功能描述
getLeftNT	获取某个非终结符的产生式
getLR1Closure	根据 LR1 的方法算 Closure 集
GO	状态转移函数
edge2str	将状态转移的边转为 string 方便比较
getFirstSet	获取字符串的 First 集
extendItem	根据 LR1 文法，将 item 进行终结符拓展
buildFamily	通过算法构建项目集族

模块设计思路与分析说明：

核心函数是 getLR1Closure，GO 和 buildFamily，三者通过 LR1 的算法构建项目集族的 DFA。

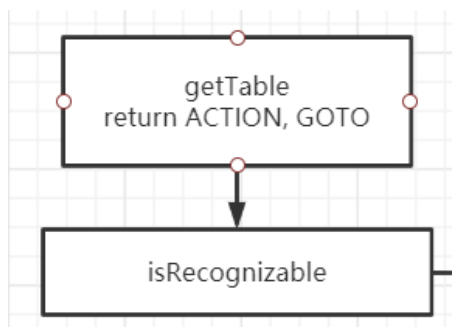
SyntacticAnalyzer 中的属性和方法描述如下：

方法的名称	功能描述
item2prodIdx	给一个 item，返回该项目对应的产生式编号
getTables	计算 ACTION 和 GOTO 数组
isRecognizable	判断一个字符串是否能被识别

模块设计思路与分析说明：

核心函数是 getTables，即通过 ItemSetSpecificationFamily 中构建的 DFA 生成 ACTION 和 GOTO 表。通过这两张表就能对任意字符串给出是否符合 LR1 文法的判断。

这部分函数调用图如下：



上文说到，语法、语义是一起做的，故完整的函数调用图在语义分析当中，也就是下一节。

### 4.3. 语义分析及中间代码生成设计

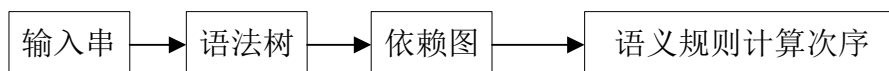
由于已经实现了 LR(1)这一自底向上的语法分析器，那么不需要语法树，考虑一遍扫描的 S 属性文法。

#### 4.3.1. S 属性文法及自底向上扫描原理

目前在实际应用中比较流行的的语义描述和语义处理方法主要是属性文法和语法制导翻译方法。我的语义分析正是基于仅包含综合属性的 S 属性文法和伴随着语法分析进行的从底向上扫描翻译的原理。

属性文法是在上下文文法的基础上，为每个文法符号配备若干相关的值，这些属性代表与文法符号相关信息，例如它的类型、值、代码序列、符号表内容等等。这些属性与变量一样，可以进行计算和传递。属性加工的过程就是语义处理的过程。

这其中的属性主要分为两类：综合属性和继承属性。综合属性用于自下而上的传递信息，继承属性用于自上而下地传递信息。



通常意义上基于属性文法的处理过程是这样的:对单词符号进行语法分析、构造语法分析树，根据输入串遍历语法树并在语法树的个节点按语义规则进行计算。这种由源程序的语法结构所驱动的处理办法就是语法制导翻译。语义规则的计算可能产生代码、在符号表中存放信息，给出错误的信息或执行其他动作。对输入符号串的翻译就是根据语义规则进行计算的结果。

然而在一些情况下并不一定要画出语法树和依赖图，可用一遍扫描实现属性文法的语义规则计算。具体实现起来就是在语法分析的同时进行语法规则的计算，无须明显地构造语法树或构造属性之间的依赖图。在自下而上的语法分析中，当一个产生式被用于归约时，此产生式相应的语义规则就被计算，完成有关的语义分析和代码产生的工作。要采用的 S 属性文法正是适用这种情况的。

S 属性文法是只含有综合属性的属性文法。而综合属性可以在分析输入符号串的同时由下而上的分析器计算。分析器可以保存预展中文法符号有关的综合属性值，每

当进行归约时，新的属性值由栈中正在归约的产生式右边符号的属性值来计算。

#### 4.3.2. 更改为 S 属性文法

在实际处理程序的过程中，由于一开始设计的文法需要的语法规则并不符合 S 属性文法的要求，因此对原有文法进行了调整改变基础文法从而避免继承属性。

例如原文法：

```
D->L:T
T->integer | char
L->L.id | id
```

其中标识符由 L 产生而类型不在 L 的子树中（由 T 决定），不能仅使用综合属性就把类型与标识符联系起来。最终得到的属性文法并不是 S 属性文法。因此将原文法做如下变换：

```
D->id L
L->,id L | : T
T->integer | char
```

这样，原文法与变换后的文法等价，但是类型信息可以通过综合属性 L.type 从 T 开始传递。可以产生相应的 S 属性文法。

#### 4.3.3. 三地址代码和四元式

语义分析最终需要的结果是中间代码，而源程序的中间表示方法包括：后缀式，三地址代码（包括三元式，四元式，间接四元式），DAG 图表示。在本实验中，使用四元式来进行中间代码的表示，语义分析程序最终生成的四元式和函数表，变量表也可以在 GUI 界面中点击对应按钮查看。

三地址代码由下面一般形式的语句构成的语句序列：

$$X:=Y \text{ OP } Z$$

其中 x, y, z 为名字，常数或编译时产生的临时变量；op 代表运算符如定点运算符，浮点运算符，逻辑运算符等等。每个语句的右边只能有一个运算符。例如，源语言表达式  $x+y*z$  可以被翻译为如下语句序列：

$$\begin{aligned} T1 &= y * z \\ T2 &= x + T1 \end{aligned}$$

其中 T1, T2 为编译时产生的临时变量。

四元式属于三地址语句的一种，一个四元式通常是一个带有四个域的记录结构。这四个域通常被称为 op, arg1, arg2, result。域 op 包含一个代表运算符的内部码，三地址语句  $x:=y \text{ op } z$  可表示为：将 y 置于 arg1 域，z 置于 arg2 域，x 置于 result 域，:= 则是运算符。

例如赋值语句  $a:=b*-c+b*-c$  可表示为如下的四元式：

序号	op	arg1	arg2	result
0	uminus	c		T1
1	*	b	T1	T2

2	uminus	c		T3
3	*	b	T3	T4
4	+	T2	T4	T5
5	:=	T5		a

其中  $T_i$  ( $i=1,2..5$ ) 存放的是表达式运算得到的中间变量。

#### 4.3.4. 具体语句的语义规则

语义分析具体涉及到的语句包括：变量声明语句、赋值语句、循环语句、条件语句、函数调用语句等，对不同的语句都需要设计不同的语义生成规则分别讨论。

##### 变量声明语句

```

typeSpecifier
    int
    void
    #
declaration
    typeSpecifier id ;
    completeFunction
    #
    #
completeFunction
    declareFunction block
    #
declareFunction
    typeSpecifier id ( formalParaList )
    #

```

如上图通过 **declaration** 推导出 **int** 类型变量的声明。

在语法分析的过程中，**typeSpecifier**->**int** 产生式对应着语义动作

**typeSpecifier.type=int;**

在 **declaration**->**typeSpecifierid** 产生式对应语义动作

**declaration.type=int;**

**declaration.size=4;**

为了防止变量重定义，需要在符号表中检查是否存在同名的变量。

最后调用 **updateSymbolTable** 在符号表中新创建一个数据项，保存该变量的名称，类型，大小，值。



```

s.name = defName
s.place = n.place
s.type = defType
s.function = self.curFunc
s.size = 4
s.offset = self.curOffset
self.curOffset += s.size
self.updateSymbolTable(s)
for code in n.code:
    n.code.stack.insert(0, code)

```

## 函数声明

在函数声明的归约中，首先检查函数表是否存在着 `id.name` 的函数，如果已经存在则说明函数重定义，输出错误提示。否则将在函数表中插入相应的函数名称和参数列表。

其中参数列表在归纳 `formalParaList` 的时候存放到 `formalParaList.stack` 中，在进行 `completeFunction` 归纳的时候获取 `formalParaList.stack` 来进行函数的创建。

```

# 搜索formalParaList表，把参数列表记录下来
self.prtNodeStack(n)
for arg in n.stack:
    s = Symbol()
    s.name = arg.data # 在处理para时，变量名放在data里
    s.place = arg.place # 此时是None
    #print('debug arg.place: ', arg.place)
    s.type = arg.type
    s.function = f.label
    s.size = 4
    s.offset = self.curOffset
    self.curOffset += s.size
    self.updateSymbolTable(s)
    #newPara =
    f.params.append((arg.data, arg.type, arg.place))

n.data = f.label
self.updateFuncTable(f)
self.stack = [] # 可以清空了
self.curFuncSymbol = f
self.sStack.append(n)

```

## 赋值语句

```

assignStatement
    id = expression ;
    #

"
expression
    primaryExpression
    primaryExpression operator expression
    #
primaryExpression
    num
    ( expression )
    id ( actualParaList )
    id
    #

```

在赋值语句的语义分析中，需要首先检查表达式左端的变量名是否在符号表中，

---

如果不存在报错：未声明的变量。

然后，会根据归纳的中间变量 `expression` 的 `type` 属性和 `place` 属性是否为空来判断变量是常数还是标识符（本质上区分常量和标识符是在词法分析的时候完成）。

同时在归纳 `assignment` 的时候进行中间代码的生成：

1. 找到赋值语句左端的变量 `s`
2. 新建一个中间变量 `n`
3. `n.code.append(expression.code)`
4. `n.code.append(':=', n.place, '_', s.place)`
5. `push(n)`

而赋值语句右端的简单运算会依照左递归的语法推导式：

`expression -> primaryExpression`  
`| primaryExpression operator expression`

在归约的时候创建中间变量，产生相应的四元式存放在 `expression.code` 上。

### 函数调用语句

把调用函数的语句也放到了 `primaryExpression` 中：

`primaryExpression->id(actualParaList)`

此时会从 `actualParaList` 的 `stack` 属性中读取输入参数，在函数表中寻找名字是 `id` 的函数，比对输入参数的 `type` 和数目与函数表中存放的函数声明信息是否符合。

而如何将 `actualParaList` 的参数传递到子程序中呢？程序会在该推导式归约的过程中，将 `actualParaList` 的 `stack` 属性存放的参数值置入符号表中，并注明变量所在的函数。子函数可以随时从符号表中取得变量。

### 条件语句和循环语句

`if` 条件语句可能有 `else`，也可能没有 `else`。对于没有 `else` 的情况，只需使用 `true` 条件为真的标签和 `end` 条件语句结束的标签即可；对于有 `else` 的，需要使用到 `true` 条件为真、`false` 条件为假、`end` 条件语句出口三个标签即可，代码如下：

```

elif nt == 'ifStatement':
    n = Node()
    n.name = nt

    # if ( expression ) block
    if len(r) == 5:
        n.true = self.getNewLabel()
        n.end = self.getNewLabel()
        nT = self.sStack.pop(-1) # True
        nExp = self.sStack.pop(-1)
        self.calExpression(nExp)
        n.code.extend(nExp.code)
        n.code.append(('j>', nExp.place, '0', n.true))
        n.code.append(('j', ' ', ' ', n.end))
        n.code.append((n.true, ':', ' ', ' '))
        for code in nT.code:
            n.code.append(code)
        n.code.append((n.end, ':', ' ', ' '))

    # if ( expression ) block else block
    elif len(r) == 7:
        n.true = self.getNewLabel()
        n.false = self.getNewLabel()
        n.end = self.getNewLabel()
        nF = self.sStack.pop(-1) # False
        nT = self.sStack.pop(-1) # True
        nExp = self.sStack.pop(-1)
        self.calExpression(nExp)

```

## 循环语句

对于循环语句，有四个成员，true、false、begin、end，分别是循环为真时调整位置、循环为假时跳转位置，循环入口，循环出口。循环语句可以识别 break 和 continue，分别跳转到 false 循环条件为假、begin 循环入口处。代码如下：

```

elif nt == 'iterStatement':
    n = Node() #生成新节点
    n.name = nt
    n.true = self.getNewLabel()#四个分支的入口
    n.false = self.getNewLabel()
    n.begin = self.getNewLabel()
    n.end = self.getNewLabel()

    if r[0]['type'] == 'while':
        statement = self.sStack.pop(-1)
        expression = self.sStack.pop(-1)
        self.calExpression(expression)
        n.code.append((n.begin, ':', ' ', ' '))
        for code in expression.code:
            n.code.append(code)
        n.code.append(('j>', expression.place, '0', n.true))
        n.code.append(('j', ' ', ' ', n.false))
        n.code.append((n.true, ':', ' ', ' '))
        for code in statement.code:
            if code[0] == 'break':
                n.code.append(('j', ' ', ' ', n.false))
            elif code[0] == 'continue':
                n.code.append(('j', ' ', ' ', n.begin))
            else:
                n.code.append(code)
        n.code.append(('j', ' ', ' ', n.begin))
        n.code.append((n.false, ':', ' ', ' '))

```

### 4.3.5. 模块设计

由于在 LR1 做规约的时候同时做语义分析和中间代码生成，将上文中提到的 SyntacticAnalyzer 进行扩充，并增加几个类辅助分析，主要成员和函数如下。

#### Node 类

成员	描述
----	----

Name	名称
Type	数据类型
Data	具体数据
Place	占位符（中间变量）
Code=[]	中间代码
Stack=[]	临时栈
Begin	循环的入口
End	循环的出口
True	控制语句为真的跳转位置
False	控制语句为假的跳转位置

说明：

其中着重说明 stack。由于很多非终结符在产生式左边和右边都有出现，如：

```
declarationChain
    $
    declaration declarationChain
    #
```

declarationChain->\$ | declaration declarationChain，那么需要等到所有的 declaration 都规约完毕了，再进行中间代码生成，故需要用 stack 存储这些 declaration 节点。

还需要说明的是 place 和 data 的关系。如果是一个常数 1，那么 data=1，place=None，而如果这个节点表示的是一个中间变量 T，那么 data=None，place=T。

给 SyntacticAnalyzer 进行扩充：

方法的名称	功能描述
semanticAnalyze	语义分析和中间代码生成
findSymbol	在符号表查找符号
updateSymbolTable	更新符号表
updateFuncTable	更新函数表
findFuncSymbolByName	查找函数表
calExpression	将表达式转换为中间代码

说明：

着重说明的是 calExpression，它将一个名为 Expression 的表达式转换为中间代码，这个函数是无意中提炼出来的，大大地降低了我的代码行数。

语义分析和中间代码生成的主函数是 semanticAnalyze，它的调用位置如下：

在 isRecognizable 函数中，

```

elif mv == 'reduce':
    prodIdx = int(target)
    prod = self.prods[prodIdx]

    # 规约时同时做中间代码生成
    #print("reducing:", shiftStr)
    self.semanticAnalyze(prod, shiftStr)
    if False == self.semanticRst:
        return False

```

说明：如果某一步骤为规约，那么将产生式和当前规约栈传给该函数，进行处理。

这个函数内部结构如下：

```
def semanticAnalyze(prod, shiftStr)
```

```
    noneTerminal = prod.left
```

```
    if noneTerminal == 'statement':
```

使用 statement 在左部的产生式对应的语义规则进行中间代码生成...

```
    elif noneTerminal == 'assign:
```

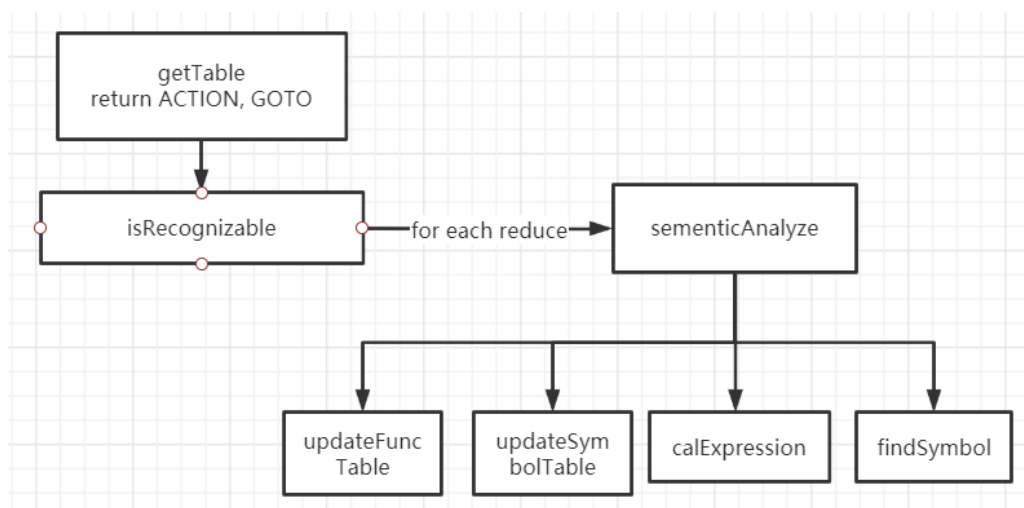
使用 assign 在左部的产生式对应的语义规则进行中间代码生成...

其它非终结符...

```
    return
```

最后只会剩下根节点的 Node，其中的 code 就是该程序完整的中间代码。

与语法分析的函数图如下：



#### 4.4. 目标代码生成设计

模块设计思路与分析说明：

获取中间代码和其它语义分析结果，生成目标代码

方法名称	描述
getRegister	获取一个寄存器
freeRegister	释放寄存器
genMips	生成 mips 代码

### 属性成员

属性名	描述
mipsCode	生成的 Mips 代码
regTable	记如此时寄存器内部存的是哪个变量的值
varStatus	记录变量此时是在寄存器当中还是 memory

主要函数是 freeRegister 和 getRegister，函数调用图如下：



具体的实现如下：

```

1.  # 释放一个寄存器，可以优化
2.  def freeRegister(self, codes):
3.      # 提取出使用了 reg 的变量，形式如 t1, t2, ...
4.      varRegUsed = list(filter(lambda x:x != '', self.regTable.values(
5.      )))
6.      # print(varRegUsed)
7.      # 统计这些变量后续的使用情况
8.      varUsageCnts = {}
9.      for code in codes:
10.         # print(code)
11.         for item in code:
12.             # print(item)
13.             tmp = str(item)
14.             if tmp[0] == 't': # 是个变量
15.                 if tmp in varRegUsed:
16.                     if tmp in varUsageCnts:
17.                         varUsageCnts[tmp] += 1
18.                     else:
19.                         varUsageCnts[tmp] = 1
20.         print('===\n', 'varUsageCnts:', varUsageCnts, '\n===\n')
21.
22.         sys.stdout.flush()
  
```

```

23.         flag = False
24.
25.         # 找出之后不会使用的变量所在的寄存器
26.         for var in varRegUsed:
27.             if var not in varUsageCnts:
28.                 for reg in self.regTable:
29.                     if self.regTable[reg] == var:
30.                         self.regTable[reg] = ''
31.                         self.varStatus[var] = 'memory'
32.                         flag = True
33.         if flag:
34.             return
35.
36.         # 释放最少使用的寄存器,
37.         sorted(varUsageCnts.items(), key=lambda x:x[1])
38.         varFreed = list(varUsageCnts.keys())[0]
39.         for reg in self.regTable:
40.             if self.regTable[reg] == varFreed:
41.                 for item in self.symbolTable:
42.                     if item.place == varFreed: # t1, t2, ...
43.                         self.mipsCode.append('addi $at, $zero, 0x{}'.format(
mat(self.DATA_SEGMENT)))
44.                         self.mipsCode.append('sw {}, {}($at)'.format(reg
, item.offset))
45.                         self.regTable[reg] = ''
46.                         self.varStatus[varFreed] = 'memory'
47.                 return
48.
49.         return

```

具体的说明见注释。注意这里的第二个参数 `code`。因为是需要看最远未使用的寄存器，所以要把当前翻译的四元式的之后的四元式都看一遍。这里可以用记忆化方法优化。

在 `genMipsCode` 中，一般的格式和模板如下如下：

```

1.     elif code[0] == '*': # 四元式第一个是运算符
2.         # 获取寄存器名
3.         arg1 = self.getRegister(code[1], dc)
4.         arg2 = self.getRegister(code[2], dc)
5.         arg3 = self.getRegister(code[3], dc)
6.         mc.append("mul {}, {}, {}".format(arg3, arg1, arg2)) # 添加到
目标代码

```

#### 4.5. 函数调用的中间代码生成和目标代码生成

单独把函数调用挑出来说明。

对于函数调用的语义分析，有多种四元式的翻译方法。

对于不同的四元式，有不同的目标代码翻译方法。

这两者最好一起设计！

由于之前有写过 mips 架构的 CPU，故这里使用较为了解的 mips 架构汇编的规则。

一般来说，对于 mips 的寄存器使用，有着一定的规定。

寄存器编号	命名	使用说明
\$0	zero	总是为0
\$1	at	MIPS汇编时使用的临时寄存器
\$2~\$3	v0~v1	子程序返回值存放寄存器
\$4~\$7	a0~a3	子程序参数传递寄存器
\$8~\$15	t0~t7	子程序可以使用这些寄存器，并且在使用之前不需要保存这些寄存器中旧的值
\$24~\$25	t8~t9	
\$16~\$23	s0~s7	子程序在使用这些寄存器之前必须保存寄存器中旧的值
\$26~\$27	k0,k1	中断、陷阱程序使用
\$28	gp	全局指针寄存器
\$29	sp	堆栈指针寄存器
\$30	s8/fp	帧指针寄存器
\$31	ra	返回地址寄存器

\$29 被用作堆栈指针寄存器（sp），保存堆栈的栈顶地址。该寄存器和 x86 的地址指针寄存器中的堆栈指针寄存器 SP，ARM 的堆栈指针寄存器 r13（sp）作用一致。

\$30 被用作帧指针寄存器（s8/fp）

\$31 被用作返回地址寄存器（ra，return address），相当于 ARM 的链接寄存器（lr），保存调用子程序的返回地址

在 MIPS 中，第九个通用寄存器\$8，又叫做帧指针（frame pointer,fp），在 X86 和 ARM 中都没有使用这样一个名字的寄存器。但是，这并不代表在 X86 和 ARM 中就没有相应功能的一个寄存器。在 X86 中，使用的是通用地址寄存器中的机制指针寄存器 BP 当做帧指针。

MIPS 不提供 push 和 pop，要自己实现，如：

push \$ra 等价于

sub \$sp, \$sp - 4

sw \$ra, 0(\$sp)

pop \$ra 等价于

lw \$ra, 0(\$sp)



---

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

对于参数传递，调用者将参数保存在寄存器 \$a0 - \$a3 中。其总共能保存 4 个数。如果有更多的参数，或者有传值的结构，其将被保存在栈中。这里的话为了简便，统一使用栈传参。

调用者不需要将返回值的位置压入栈中。寄存器 \$v0 和 \$v1 来保留返回值。当被调用者计算出返回值时，将其保存在寄存器 \$v0（如果需要，和 \$v1 中）。

被调用者从寄存器中访问参数和返回值。

在没有 BP(base pointer) 寄存器的目标架构中，进入一个函数时需要将当前栈指针向下移动 n 比特，这个大小为 n 比特的存储空间就是此函数的 stack frame 的存储区域。此后栈指针便不再移动，只能在函数返回时再将栈指针加上这个偏移量恢复栈现场。由于不能随便移动栈指针，所以寄存器压栈和出栈都必须指定偏移量，这与 x86 架构的计算机对栈的使用方式有着明显的不同。

(<https://blog.csdn.net/do2jiang/article/details/5404566>)

由此，设计函数调用语义分析。

首先是过程调用的数据结构设计。



```
1 class FunctionSymbol:
2     def __init__(self):
3         self.name = None #函数的标识符
4         self.type = None #返回值类型
5         self.label = None #入口处的标签
6         self.params = [] #形参列表
7         self.tempVar = [] #局部变量列表
```

对此，需要给 SyntacticAnalyzer 新增方法

成员名	描述
updateFuncTable	更新函数表
getNewFuncLabel	获取函数符号
findFuncSymbolByName	由函数名获得函数符号

对于函数定义，根据产生式，主体代码如下：

```

1 # declareFunction -> typeSpecifier id ( formalParaList )
2 elif nt == 'declareFunction':
3     f = FunctionSymbol() # 准备登记一个函数
4     f.name = shiftStr[-4]['data'] # 函数名
5     f.type = nFuncReturn.type # 返回类型
6     f.label = self.getNewFuncLabel() # 标签
7
8     # 搜索formalParaList表, 把参数列表记录下来
9     for arg in formalParaList:
10         s = Symbol()
11         ...
12     self.updateFuncTable(f)

```

```

1 # completeFunction -> declareFunction block
2 elif nt == 'completeFunction':
3     code = []
4     code.append(("函数名", ':', '_', '_')) # 函数名
5
6     for arg in argList:
7         # 拿参数

```

而对于函数调用，有：

```

1 code=[]
2 # sp移动, 申请空间, 保存返回地址
3 code.append(('-', 'sp', 4 * len(symbol_temp_list) + 4, 'sp')) # 注意+4给ra留空间
4 code.append(('store', '_', 4 * len(symbol_temp_list), 'ra')) # 保存ra的值避免复写
5
6 for node in n.stack: # 实参列表
7     code.append(('push', '_', 4 * n.stack.index(node), node_result))
8
9 code.append(('call', '_', '_', function.label))
10
11 # 恢复现场
12 code.append(('load', '_', 4 * len(symbol_temp_list), 'ra'))
13 code.append(('+', 'sp', 4 * len(self.curFuncSymbol.params) + 4, 'sp'))
14
15 # 返回值在v0寄存器里
16 code.append((':=', 'v0', '_', n.place))

```

解释可见注释。

对于目标代码，则强调 call 语句

elif code[0] == 'call':

objCode.append('jal {}'.format(code[3]))

jal 是会把返回地址保存在 ra 寄存器中的。其它的目标代码和四元式差不多。

## 5. 调试分析

测试数据，测试输出的结果，时间复杂度分析，和每个模块设计和调试时存在问题的思考（问题是哪些？问题如何解决？）。

首先测试数据分为正常数据和异常数据。正常数据这里不做过多说明，因为如果正常数据不通过是无法进行下一步代码编写的，但目标代码生成需要关注一个寄存器的分配和释放机制是否正常。

另外，目前来看源代码的错误都能在词法分析、语法分析和语义分析中得到检查。

关注点还有异常数据，对于一些常见错误有一定鉴别能力和提示。

具体的语义错误诊断和处理实现了以下几种：

变量重定义

使用未声明的变量

使用未定义的函数

变量赋值时类型错误

函数形参和实参不匹配

必须说明的是，静态语义错误的诊断和处理与语法设计有关。由于是在规约的时候进行语义分析和中间代码生成，故如果规约都不成立那么就会报错。

以变量重定义为例，进行说明：

```
elif nt == 'declaration': # variable or function
    n = self.sStack.pop(-1)
    n.name = nt

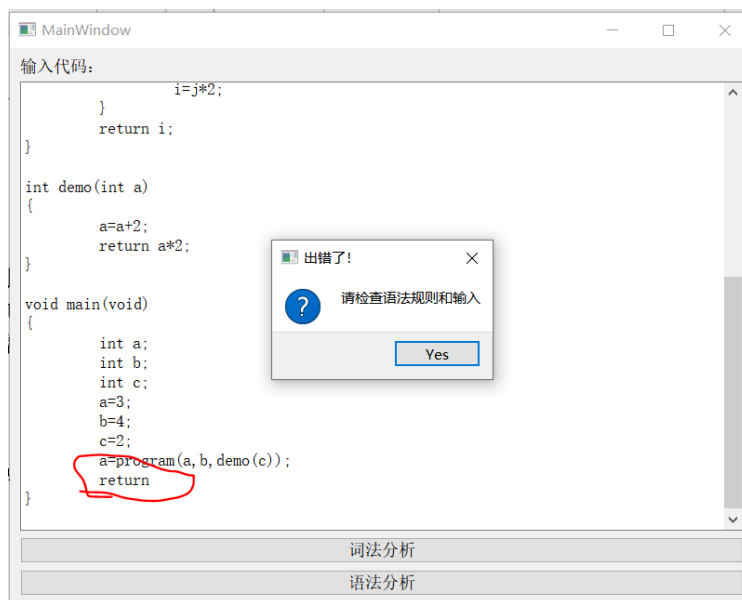
    if len(r) == 3: # -> typeSpecifier id ;
        defType = n.type
        defName = shiftStr[-2]['data'] # 变量名
        #for node in n.stack: 如果有连续声明才用
        #if self.curFuncSymbol == None:
        s = self.findSymbol(defName, self.curFuncSymbol.label)
        if s != None:
            print("multi defination?")
            self.semanticRst = False
            self.semanticErrMsg = "变量重定义: " + str(shiftStr[-2]['row']) + "行" + str(shiftStr[-2]['column']) + "列"
            return
            # 弹窗报错之类的
        else:
```

说明：在变量声明语句中，会根据变量名查找是否已经定义过该元素了。如果重定义，将会输出错误信息、行数列数，设置错误标记并返回。

其它的错误诊断和处理与这种模式类似。

### 5.1. 语法分析测试

如果一些分析不符合语法，比如 `return` 没有加 “;”，显示如下：



### 5.1.1. 时间复杂度分析

这一部分是及时分析的，所以是一遍就出了。

### 5.1.2. 存在的问题与思考

在语法分析的错误中，还有一个问题是语法本身就有问题。

仔细分析，对于语法本身设计的错误，体现在生成 ACTION 和 GOTO 表的时候会复写！

所以我在写 ACTION 和 GOTO 的时候判断是否已经有值，如下：

```
if self.ACTION[I.name][item.terms[0]] != '':  
    print('rewrite error!!!')
```

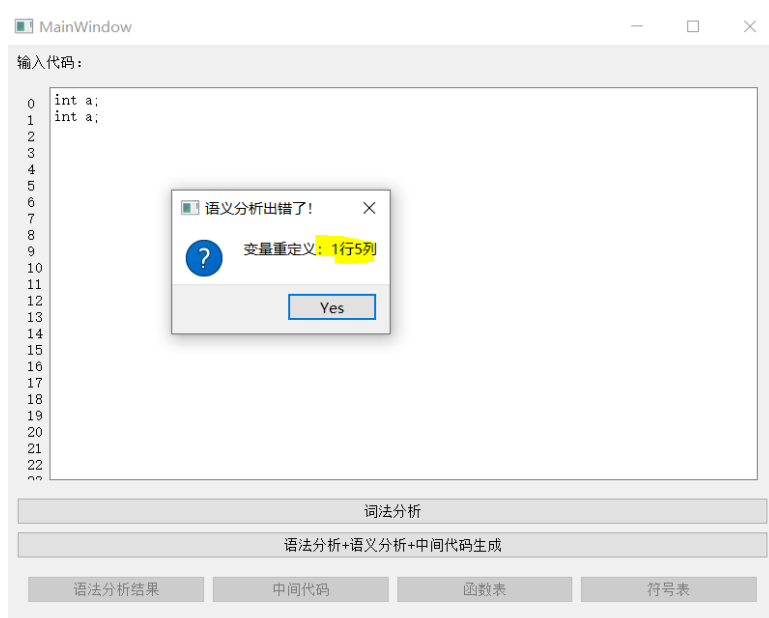
通过这种方式查错，设计了合理的语法。

这一步其实相当重要，因为往往是在语义分析的时候才知道自己原来设计的语法有多烂，做语义分析有多别扭。好的语法设计是为语义分析省事。

## 5.2. 静态语义测试

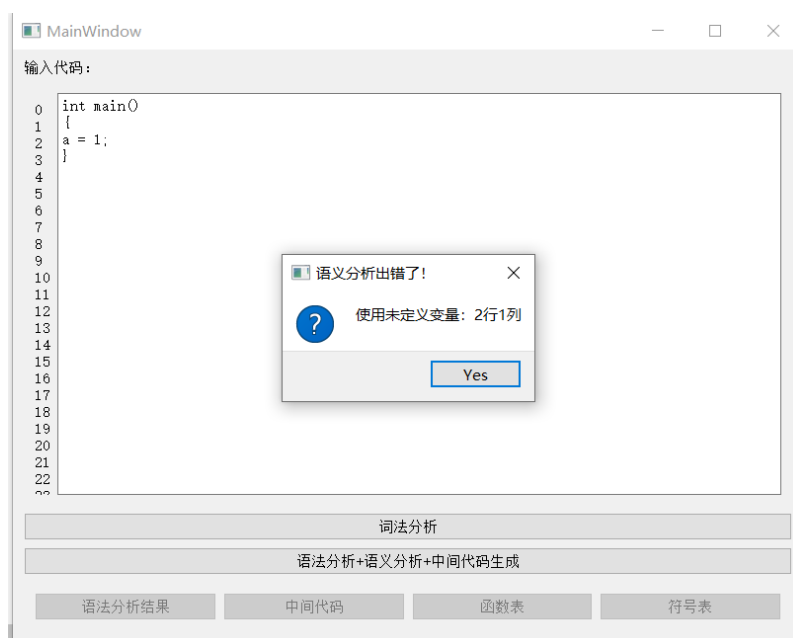
### 5.2.1. 变量重定义

```
int a;  
int a;
```



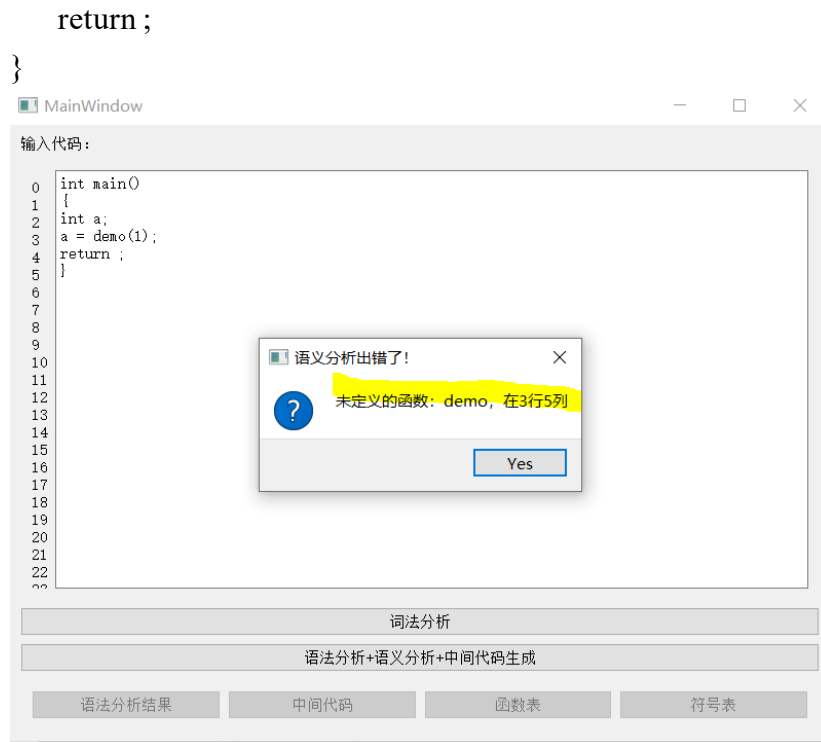
### 5.2.2. 使用未声明的变量

```
int main()
{
    a = 1;
}
```



### 5.2.3. 使用未定义的函数

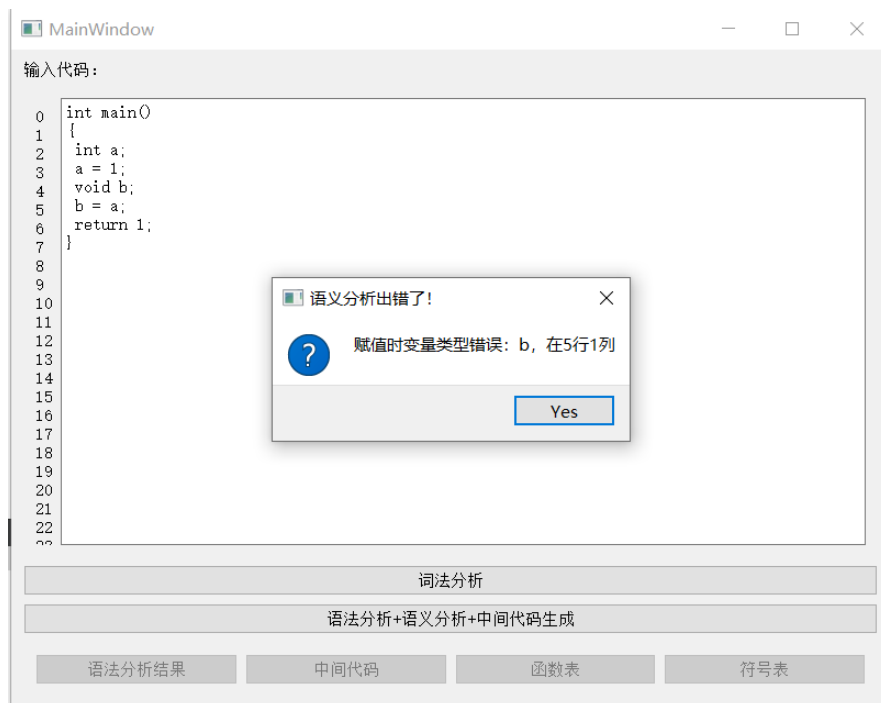
```
int main()
{
    int a;
    a = demo(1);
}
```



#### 5.2.4. 变量赋值时类型错误

```
int main()
{
    int a;
    a = 1;
    void b;
    b = a;

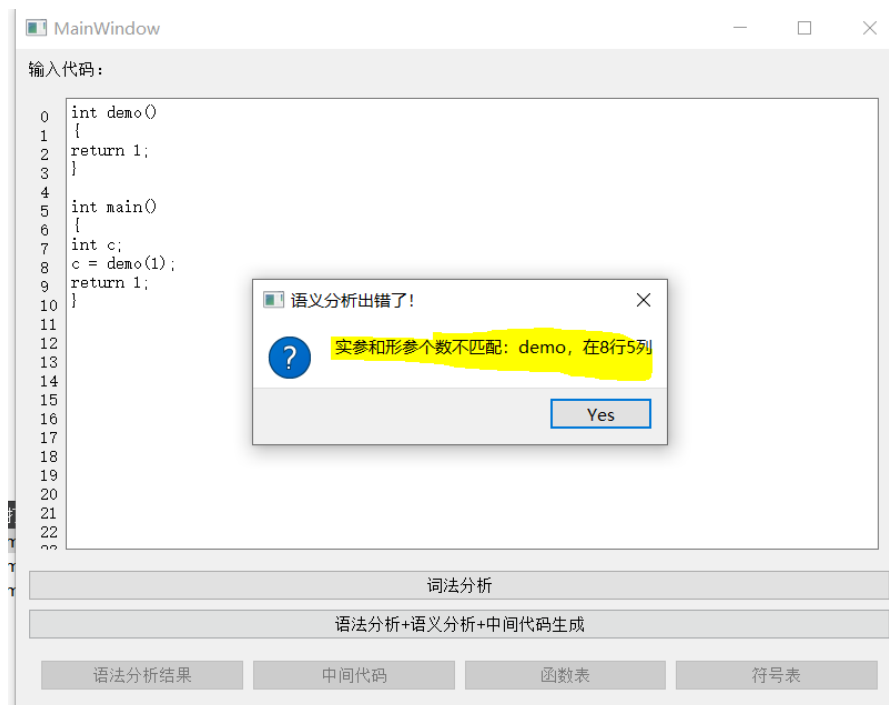
    return 1;
}
```



#### 5.2.5. 函数形参和实参不匹配

```
int demo()  
{  
    return 1;  
}
```

```
int main()  
{  
    int c;  
    c = demo(1);  
    return 1;  
}
```



### 5.2.6. 时间复杂度分析

以上的错误都是一遍过程中找错。但细节上不太一样。比如：

变量、函数重定义/未定义需要遍历符号表和函数表；

形参和实参的匹配看语法的设计。

我这里的形参列表的产生式如下：

paraList →

para

paraList, para

实际上这里看语法的设计是否合理。如果是如上的设计，那就是压栈，判断个数是否相同、类型是否相同，依次出栈即可。

### 5.2.7. 存在的问题与思考

这一部分的错误都是很常见的错误，具体的识别较为简单不再赘述。

这里有一个小问题，是如何区分各类错误的问题。可以看到，错误除了错误类型，还有所在行列和变量名等信息。

我这里原本想的是用结构化数据返回，类似于错误码机制。但后来发现错误需要的内容往往不一样，结构化数据反而不灵活，于是还是选择返回字符串。

劣势就在于不够灵活，错误提示内嵌于分析过程，耦合度太高，维护不够方便。

## 5.3. 目标代码生成测试

### 5.3.1. 寄存器取用测试



---

测试代码:

```
int main(void)
```

```
{
```

```
    int a;int b;int c;int d;int e;int f;int g;int h;int i;int j;int k;int l;int m;int n;int o;
```

```
    int p;int q;int r;int s;int t;int u;
```

```
    a=0;b=1;c=2;d=3;e=4;f=5;g=6;h=7;i=8;j=9;k=10;l=11;m=12;n=13;o=14;p=15;q=16;r=17;s=18;t=19; u=20;
```

```
    a=0;b=1;c=2;d=3;e=4;f=5;g=6;h=7;i=8;j=9;k=10;l=11;m=12;n=13;o=14;p=15;q=16;r=17;s=18;t=19;
```

```
    return 1;
```

```
}
```

结果:

```
addiu $sp, $zero, 0x0
```

```
or $fp, $sp, $zero
```

```
main:
```

```
add 0,$zero,$7
```

```
add 1,$zero,$8
```

```
add 2,$zero,$9
```

```
add 3,$zero,$10
```

```
add 4,$zero,$11
```

```
add 5,$zero,$12
```

```
add 6,$zero,$13
```

```
add 7,$zero,$14
```

```
add 8,$zero,$15
```

```
add 9,$zero,$16
```

```
add 10,$zero,$17
```

```
add 11,$zero,$18
```

```
add 12,$zero,$19
```

```
add 13,$zero,$20
```

```
add 14,$zero,$21
```

```
add 15,$zero,$22
```

```
add 16,$zero,$23
```

```
add 17,$zero,$24
```

```
add 18,$zero,$25
```

```
addi $at, $zero, 0x0
```

```
sw $7, 0($at)
```

---

```
add 19,$zero,$7
addi $at, $zero, 0x0
sw $8, 4($at)
add 20,$zero,$8
add 0,$zero,$8
add 1,$zero,$8
add 2,$zero,$8
add 3,$zero,$9
add 4,$zero,$8
add 5,$zero,$9
add 6,$zero,$10
add 7,$zero,$11
add 8,$zero,$8
add 9,$zero,$9
add 10,$zero,$10
add 11,$zero,$11
add 12,$zero,$12
add 13,$zero,$13
add 14,$zero,$14
add 15,$zero,$15
add 16,$zero,$8
add 17,$zero,$9
add 18,$zero,$10
add 19,$zero,$11
add 1,$zero,retValue
jr $ra
```

分析：

我设定的寄存器一共 26 个，在 26 个以后，根据寄存器的规则，需要把最远使用的寄存器的值存入内存，也就是有 sw 的操作。可以看到这里的测试结果是正确的。

### 5.3.2. 复杂度分析

由于是找最远使用做替换，所以需要遍历之后所有的代码。但完全可以使用一个哈希桶，先一次遍历，记录所有变量出现的位置，用链表尾插法记录出现行数。

这样每翻译一行就把这一行用到的变量的链表头删去。这样一次遍历即可。

## 5.4. 存在的问题，思考与解决

### 5.4.1. 空串处理

---

在龙书和课本上的 LR(1)都没有提到对于空产生式要如何处理。经过大量地查找资料 and 尝试，主要要做到如下：

GO 函数不把空串当作任何字符 (NT/T) 处理

$A \rightarrow \epsilon$  和  $A \rightarrow \epsilon \cdot$  等价于  $A \rightarrow \cdot$ 。

最后 reduce 的时候，规约栈要压入 A，状态栈压入 GO(I,A)所到达的状态

经过反复尝试，做到以上三点，LR(1)即具备了较强的能力。

#### 5.4.2. 判断是否该元素已存在该集合中

如果是普通元素，如数字，可以通过 `list(set(a))`，将 a 的重复元素去除；除此之外还有 list 的各种方法可以使用，大不了用循环。

但在本项目中，比如将项目 item 加入到项目集 itemSet 的时候，如何判断该 item 是否已经存在于该 itemSet。Item 是自定义的类，不是普通元素，上面的 `list(set(a))` 的方法不适用。解决方法是给 item 写一个 string 方法，如下：

```
def toString(self):
    rst = self.left + '->'
    pos = 0
    for right in self.right:
        if pos == self.dotPos:
            rst += '@' # 代替点
            rst += right['type'] + ' '
            pos += 1

    # 也可能点在最末
    if pos == self.dotPos:
        rst += '@'

    # LR(1), 需要附上term信息
    for term in self.terms:
        rst += term + ' '

    return rst
```

然后每添加一个 item 到一个 itemSet，就用 Python 字典记录这个 string：

```
setStrings[tempItemSet.name] = tempItemSet.toString()
```

每次要加入新的 item 时，就用它的 string 在记录中比对：

```
if tempItemSet.toString() in setStrings.values():
```

虽然对于某一个 item，它的 string 不唯一，但是 item 是在前一个步骤确定了的、固定好了的，在之后的处理过程中是只读的，所以这种方法是合理的。

同理，itemSet 也可以写一个这样的 toString 方法，来判断 itemSet 是否重复。

#### 5.4.3. 单个非终结符的 First 集的并发求解

原来的写法是这样子的：

For 所有非终结符

---

## 计算单个非终结符

但后来发现不太行，因为有些情况下会陷入死循环，比如：

$A \rightarrow AB$

主要问题分析如下：

1. 必须并发计算 first 集，即按照笔算的方法，能算出几个字符，每个字符算出几个就算几个，直到不再增加，原因：不能一次性算出一个 NT 的 first 集，遇到递归即使跳过了依然可能永远算不了它；

2. 由于是递归函数，对于左递归  $A \rightarrow Aa$  的 first 求解陷入死循环，并且对于求解它的 first 没有意义：

$A \rightarrow Aa$

$\text{First}(A) = \text{First}(A)$ , 无意义

3. 考虑：可能存在  $A \rightarrow \epsilon$ ，使得 First 集能进一步扩大，且顺序有一定影响

$A \rightarrow \epsilon$

$A \rightarrow AB$

或

$A \rightarrow AB$

$A \rightarrow \epsilon$

于是选用一种偏并发的算法，就如同人在算 first 集一样，能算出几个字符，每个字符算出几个就算几个，直到不再增加。这样算出来的 first 一定是完备的。如果连这种方法都算不出来 first，那么它必然是有问题的。

每当计算一个非终结符时，进入了这个计算单个非终结符的 first 函数 `calNTFirstSet`，它能利用的 first 集可能是不完整的，例如： $A \rightarrow AB$ ，而 A 的 first 集因为左递归的原因退出了函数，没有继续计算下去，但可能它的 first 集还没有算完。所以需要 while 多次，直到没有一个 NT 的 first 集能再扩大为止。

### 5.4.4. 语法更改，消除需要继承属性的语义规则

我们知道，S 属性文法是只有综合属性的文法。而在语法分析的时候，还对语义分析和中间代码生成没有比较好的认识，故需要重构语法。

重构方法在于，每次都用相同的代码测试，改动一点语法就运行一次，确保每次改动都是正确的。

首先，正确的 LR (1) 文法是不会重复填写 GOTO 表和 ACTION 表的，故需要在每次填写两张表的时候判断填写时是否为空，若非空则报错，放弃这次语法的改动。当然这个要确保 First 集的计算是绝对无误的，GOTO 和 ACTION 的填写是绝对无误的（某种意义上这个是伪命题）。

并且，由于文法过于复杂的情况下，人工去判断是否是 LR (1) 语法（算 follow 集）基本不可能（设计的语法 18 个非终结符，已经 130 个状态了）。

那么如何将继承属性消除？实际上这个要在具体的语义规则的撰写时才会发现。

例如：

```
completeFunction
| declareFunction block
| #

declareFunction
| typeSpecifier id ( formalParaList )
| #

block
| { statementChain }
| #
```

这三个语法完成一个函数的定义。但是为什么不是一个产生式完事，非要分成函数返回值（**typeSpecifier**）、函数名（**id**）、形参列表（**formalParaList**）一个产生式，和花括号的语句块一个产生式呢？

原因在于，当处理 **statementChain** 时，需要为里面的临时变量进行登记，但如果是在一条产生式中完成整个函数的定义，由自底向上文法的规则，就会发现在处理语句块的时候，函数还没有登记过。所以将函数头声明的语句先完成，赋予函数的登记的语义规则，那么接下来的语句块的变量等等都有地方登记。

在者是一些产生式的精简。因为对于每一条产生式都要配语义规则，越少的产生式工作量越少，尤其是期末特别忙的时候。有些不定长的语句，语义规则的翻译等待长度确定时再进行，如形参列表、实参列表、表达式：

```
formalParaList
| $
| para
| para , formalParaList
| void
| #

actualParaList
| $
| expression
| expression , actualParaList
| #

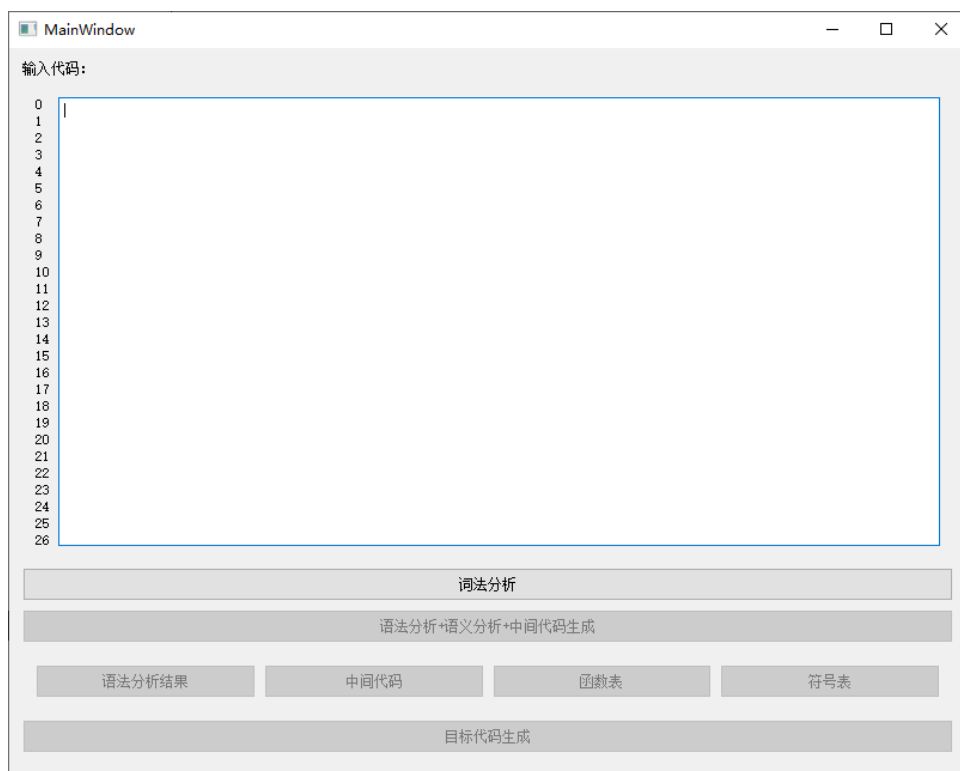
expression
| primaryExpression
| primaryExpression operator expression
| #
```

将它们的中间结果存在 **Node.stack** 里，等待 **formalParaList** 被它的上一级产生式，如 **declareFunction -> typeSpecifier id ( formalParaList )** 规约时，再进行 **formalParaList** 的翻译。将 **para** 从 **formalParaList** 节点的 **stack** 中取出来，逐个翻译。

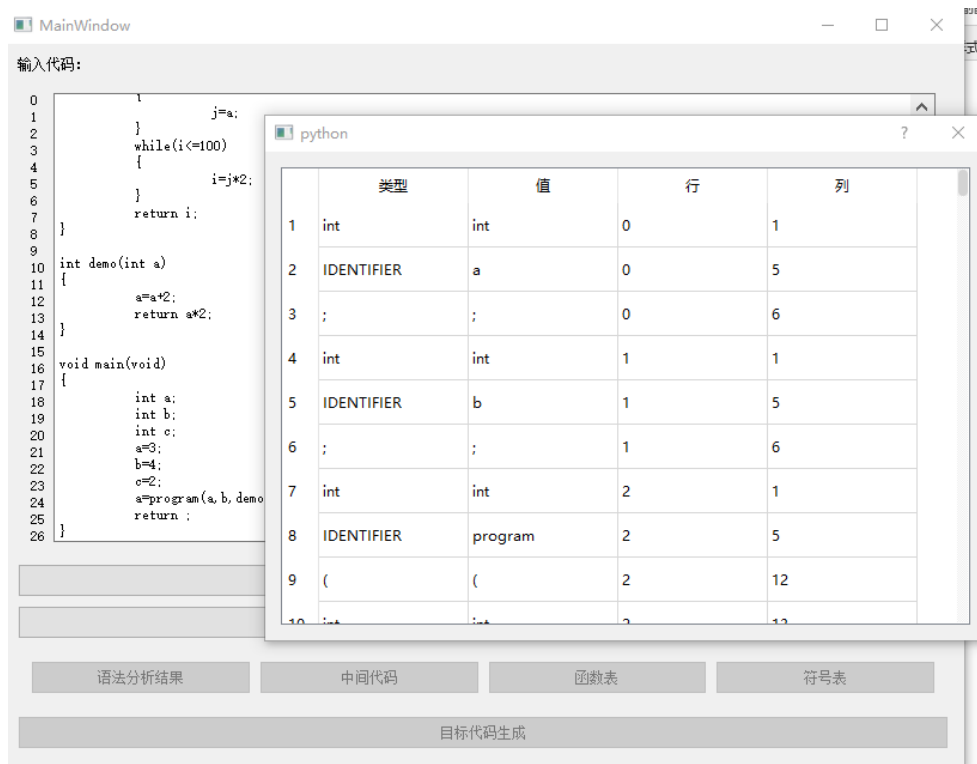
以上就是主要的语法更改。最后语法精简到 18 个非终结符。

## 6. 用户使用说明

初始界面如下：

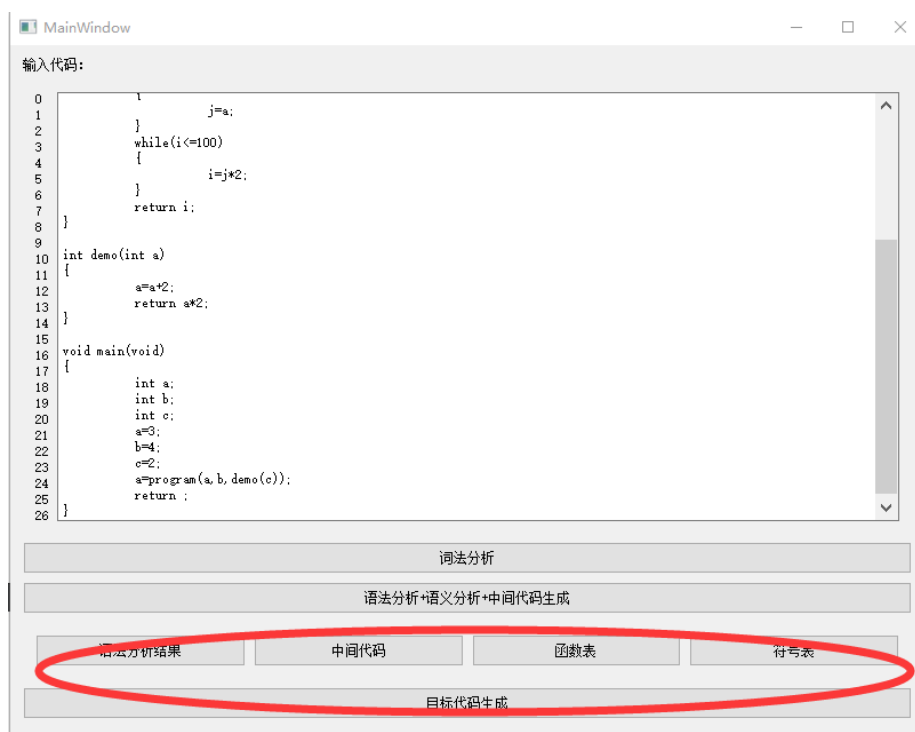
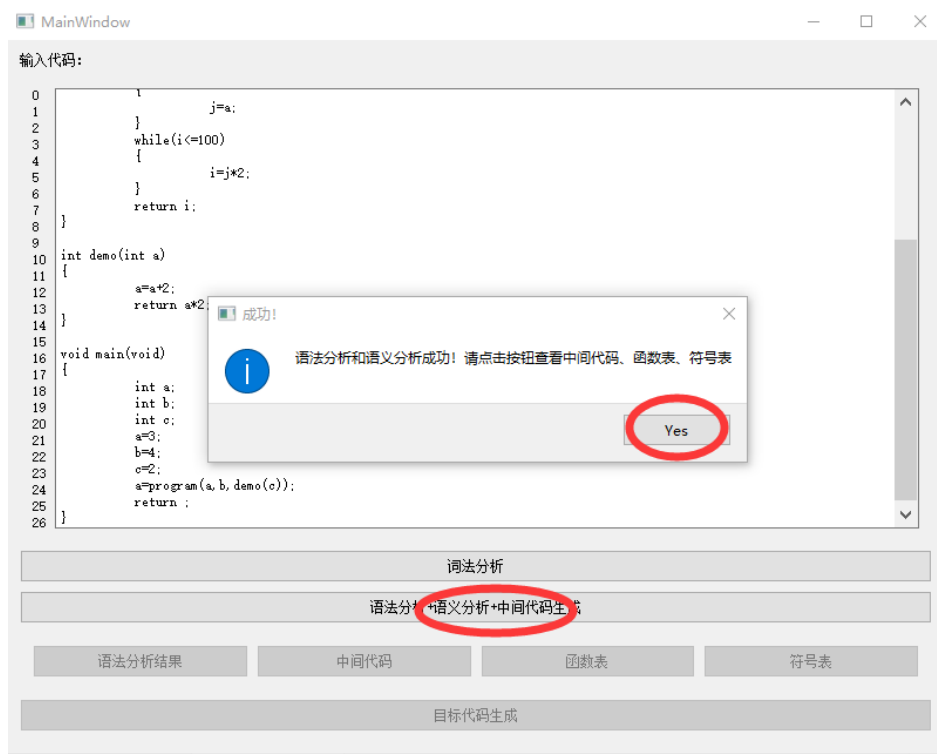


然后放入带过程调用的代码：



点击词法分析的按钮，弹出词法分析结果：

点击“语法分析+语义分析+中间代码生成”，释放下面四个按钮的使用



然后点击四个新生成的按钮，可以查看结果，从左到右依次：

	状态栈	移动栈	输入栈
1	s0	#	int IDENTIFIER ; int IDENTIFIER ; int ...
2	s0 s1	# int	IDENTIFIER ; int ...
3	s0	#	IDENTIFIER ; int ...
4	s0 s5	# typeSpecifier	IDENTIFIER ; int ...
5	s0 s5 s9	# typeSpecifier IDENTIFIER	; int IDENTIFIER ; int IDENTIFIER ( int ...
6	s0 s5 s9 s13	# typeSpecifier IDENTIFIER ;	int IDENTIFIER ; int IDENTIFIER ( int ...
7	s0	#	int IDENTIFIER ; int IDENTIFIER ( int ...
8	s0 s6	# declaration	int IDENTIFIER ; int IDENTIFIER ( int ...
9	s0 s6 s1	# declaration int	IDENTIFIER ; int IDENTIFIER ( int ...
10	s0 s6	# declaration	IDENTIFIER ; int IDENTIFIER ( int ...
11	s0 s6 s5	# declaration typeSpecifier	IDENTIFIER ; int IDENTIFIER ( int ...
12	s0 s6 s5 s9	# declaration typeSpecifier ...	; int IDENTIFIER ( int IDENTIFIER , int ...
13	s0 s6 s5 s9 s13	# declaration typeSpecifier ...	int IDENTIFIER ( int IDENTIFIER , int ...
14	s0 s6	# declaration	int IDENTIFIER ( int IDENTIFIER , int ...
15	s0 s6 s6	# declaration declaration	int IDENTIFIER ( int IDENTIFIER , int ...

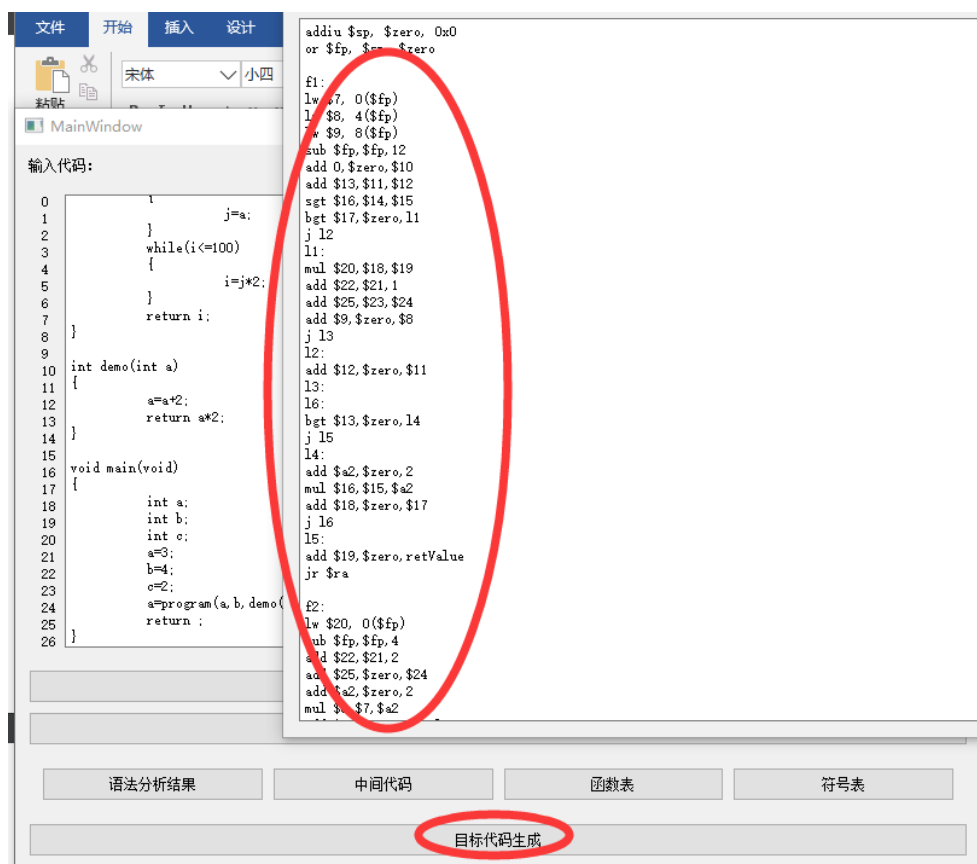
	operation	arg1	arg2	result
1	f1	:	-	-
2	pop	-	-	t3
3	pop	-	-	t4
4	pop	-	-	t5
5	-	fp	-	fp
6	:=	0	-	t6
7	+	t4	t5	t8
8	>	t3	t8	t12
9	j>	t12	0	l1
10	j	-	-	l2
11	l1	:	-	-
12	*	t4	t5	t9
13	+	t9	1	t10
14	+	t3	t10	t11
15	:=	t11	-	t7



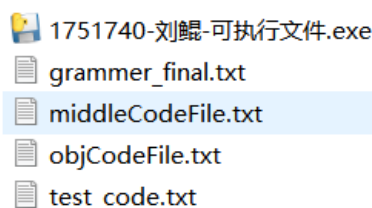
	函数的标识符	返回值类型	入口处的标签	形参列表
1	global		global	[]
2	program	int	f1	[('a', 'int', 't3'), ('b', 'int', 't4'), ('c', 'int', ...
3	demo	int	f2	[('a', 'int', 't15')]
4	main	void	main	[]

	符号的标识符	类型	占用字节数	内存偏移量	对应的中间变量	所在函数
1	a	int	4	0	t1	global
2	b	int	4	4	t2	global
3	a	int	4	8	t3	f1
4	b	int	4	12	t4	f1
5	c	int	4	16	t5	f1
6	i	int	4	20	t6	f1
7	j	int	4	24	t7	f1
8	a	int	4	28	t15	f2
9	a	int	4	32	t18	main
10	b	int	4	36	t19	main
11	c	int	4	40	t20	main

点击目标代码生成，生成代码：



然后可以看到多了两个新的文件：middleCodeFile 和 objCodeFile，点开就是上面的弹窗显示的代码。



## 7. 课程设计总结

可以包括课程设计过程的收获、遇到问题、遇到问题解决问题过程的思考、程序调试能力的思考、对这门课程的思考、在课程设计过程中对课程的认识等内容。

设计类的时候，我在对语法分析的功能进行充分的了解后，以功能为分类标准，将语法分析分成三个部分，彼此有关联，但耦合性又足够低，也避免了全局变量满天飞的情况。

对于语法分析，遇到的最大的困难就在于空产生式的处理，那时候真的是把所有的资料都看过一遍了，课本、龙书、stackoverflow...，找到别人说的语焉不详的话去尝试，实在是痛苦。

---

对于中间代码，看着 ppt 的原理讲解发懵，并且我们的课本是抄龙书的，翻来覆去都一回事，看了等于没看。后来，我把目标分割，一步一步来，先完成最简单也最基础和必要的变量声明，然后是赋值语句和函数调用。在写变量声明的时候，发现语法设计得很不合理，导致 S 属性文法的翻译真的很不顺手。于是我开始简化语法，也就是这个时候才算开窍了，路子走得顺畅了起来，苦尽甘来。

在这两次大作业中，我锻炼了资料查找能力和代码编写能力。对着教程写代码很有程序员的感觉。就是根据文档写代码的能力。但是在编写过程中，只是隐隐约约理解了原理，在老师后续上课的解释后，我才真正明白一些方法背后的原理，而不是单纯地死板地实现语法分析和语义分析。

我认为，减少写代码的痛苦的最好方式之一就是找一点好的 example。比如我看课件、课本和龙书，例子都讲得很正确，可是我该怎么用？“你说得都对，然后呢？”我只好看着代码这里改改那里加加，逼着自己写出东西来。有时候动手开始做了才会发现盲点，全部都搞懂了再下手写代码当然很快，但想必这种好日子越来越少了。

在目标代码生成的部分，我起初对于 MIPS 的函数调用并不了解。而且中间代码和目标代码生成是关联的（尽管我原来的目的是解耦，使得两者互不干扰）。这里更改一下，那里就要动好多。而且 MIPS 不同的编译器似乎都不太一样，于是我干脆选择写起来最顺手的那一种。

在这次课设中，我极大地锻炼了自己的各项能力，是一个有模有样的小项目。可以感受到，纸上得来终觉浅，绝知此事要躬行。面对具体的问题，有具体的解决方案，需要不断地测试和尝试。

## 8. 附录

本课设附有：

- 1.源程序
- 2.报告
- 3.演示视频
- 4.答辩 ppt
- 5.exe 可执行文件

## 9. 参考文献

- [1] 自底向上分析（下）[EB/OL].  
[https://pandolia.net/tinyc/ch12\\_bottom\\_up\\_parse\\_b.html](https://pandolia.net/tinyc/ch12_bottom_up_parse_b.html), 2008-10-05/2019-09-25.
- [2] 陈火旺, et al. 程序设计语言编译原理(第 3 版) [M]. 北京:国防工业出版社, 1992.
- [3] PyQt5 笔记 (03): 弹出窗口大全[EB/OL].  
<https://www.cnblogs.com/hhh5460/p/5174266.html>, 2017-11-07/2019-09-26.

- 
- [4] GLR 解析器. [EB/OL]. [https://en.wikipedia.org/wiki/GLR\\_parser](https://en.wikipedia.org/wiki/GLR_parser), 2015-11-27/2019-11-30.
  - [5] Edward D. Willink. Meta-Compilation for C++[D]. Surrey: University of Surrey, 2001: 147.
  - [6] MIPS 汇编. [EB/OL]. <https://blog.csdn.net/do2jiang/article/details/5404566>
  - [7] Alfred V. Aho, et al. 编译原理[M]. 北京:机械工业出版社, 2003.