**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](#_Toc40117330)

[1.1. 目的 3](#_Toc40117331)

[1.2. 要求 4](#_Toc40117332)

[2. 需求分析 4](#_Toc40117333)

[2.1. 任务输入及其范围 4](#_Toc40117334)

[2.2. 输出形式 5](#_Toc40117335)

[2.2.1. 输出中间代码表示的程序 5](#_Toc40117336)

[2.2.2. 输出目标代码(可汇编执行)的程序 5](#_Toc40117337)

[2.3. 程序功能 6](#_Toc40117338)

[2.4. 测试数据 6](#_Toc40117339)

[2.4.1. 变量重定义 6](#_Toc40117340)

[2.4.2. 使用未声明的变量 6](#_Toc40117341)

[2.4.3. 使用未定义的函数 7](#_Toc40117342)

[2.4.4. 变量赋值时类型错误 7](#_Toc40117343)

[2.4.5. 函数形参和实参不匹配 7](#_Toc40117344)

[2.4.6. 寄存器是否正常选用 7](#_Toc40117345)

[3. 概要设计 8](#_Toc40117346)

[3.1. 任务的分解 & 数据类型的定义 8](#_Toc40117347)

[3.1.1. 语法文件读取与解析，词法分析 – CFG类 8](#_Toc40117348)

[3.1.2. 构建项目集规范族 – ItemSetSpecificationFamily类 9](#_Toc40117349)

[3.1.3. 语法、语义、中间代码生成 – SyntacticAnalyzer类 10](#_Toc40117350)

[3.1.4. 目标代码生成 – ObjCodeGenerator类 10](#_Toc40117351)

[3.2. 主程序流程 11](#_Toc40117352)

[3.3. 模块间的调用关系 11](#_Toc40117353)

[4. 详细设计 12](#_Toc40117354)

[4.1. 词法分析 13](#_Toc40117355)

[4.1.1. 实现思路 13](#_Toc40117356)

[4.1.2. 去除注释后进行词法单元的识别 13](#_Toc40117357)

[4.1.3. 词法规则的设计和读取 13](#_Toc40117358)

[4.2. LR1语法分析设计 13](#_Toc40117359)

[4.2.1. LR(1)原理 13](#_Toc40117360)

[4.2.2. 模块设计与分析 13](#_Toc40117361)

[4.3. 语义分析及中间代码生成设计 13](#_Toc40117362)

[4.3.1. S属性文法及自底向上扫描原理 13](#_Toc40117363)

[4.3.2. 更改为S属性文法 13](#_Toc40117364)

[4.3.3. 三地址代码和四元式 13](#_Toc40117365)

[4.3.4. 具体语句的语义规则 13](#_Toc40117366)

[4.3.5. 模块设计 13](#_Toc40117367)

[4.4. 目标代码生成设计 13](#_Toc40117368)

[4.5. 函数调用的中间代码生成和目标代码生成 13](#_Toc40117369)

[5. 调试分析 12](#_Toc40117370)

[5.1. 语法分析测试 13](#_Toc40117371)

[5.1.1. 时间复杂度分析 13](#_Toc40117372)

[5.1.2. 存在的问题与思考 13](#_Toc40117373)

[5.2. 静态语义测试 13](#_Toc40117374)

[5.2.1. 变量重定义 13](#_Toc40117375)

[5.2.2. 使用未声明的变量 13](#_Toc40117376)

[5.2.3. 使用未定义的函数 13](#_Toc40117377)

[5.2.4. 变量赋值时类型错误 13](#_Toc40117378)

[5.2.5. 函数形参和实参不匹配 13](#_Toc40117379)

[5.2.6. 时间复杂度分析 13](#_Toc40117380)

[5.2.7. 存在的问题与思考 13](#_Toc40117381)

[5.3. 目标代码生成测试 13](#_Toc40117382)

[5.3.1. 寄存器取用测试 13](#_Toc40117383)

[5.3.2. 复杂度分析 13](#_Toc40117384)

[5.4. 存在的问题，思考与解决 13](#_Toc40117385)

[5.4.1. 空串处理 13](#_Toc40117386)

[5.4.2. 判断是否该元素已存在该集合中 13](#_Toc40117387)

[5.4.3. 单个非终结符的First集的并发求解 13](#_Toc40117388)

[5.4.4. 语法更改，消除需要继承属性的语义规则 13](#_Toc40117389)

[6. 用户使用说明 12](#_Toc40117390)

[7. 课程设计总结 17](#_Toc40117391)

[8. 附录 18](#_Toc40117392)

[9. 参考文献 18](#_Toc40117393)

# 课程设计重述

## 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。

## 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

# 需求分析

## 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。

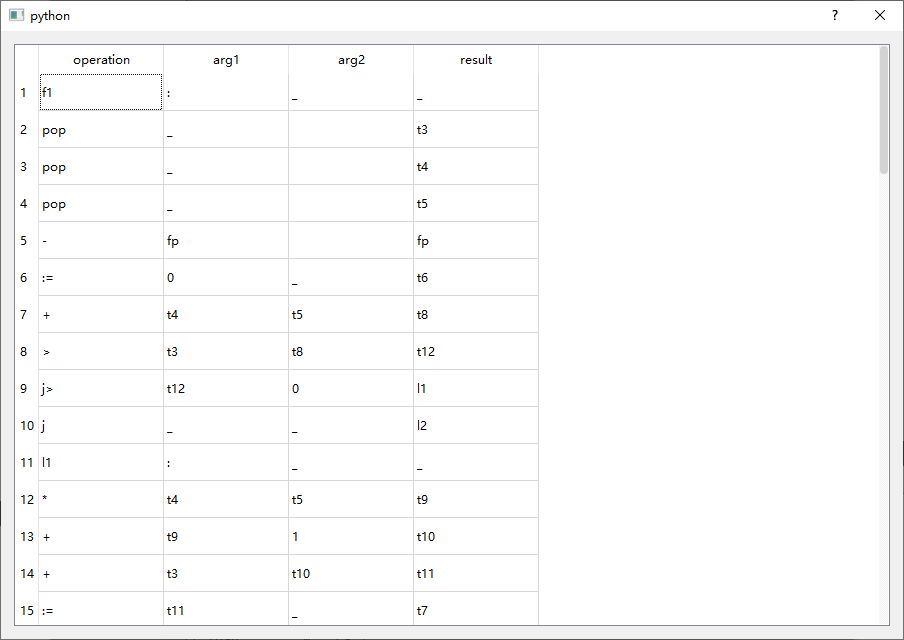
一段带过程调用的代码段实例如下：

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. int demo(int a)
23. {
24. a=a+2;
25. return a\*2;
26. }
28. void main(void)
29. {
30. int a;
31. int b;
32. int c;
33. a=3;
34. b=4;
35. c=2;
36. a=program(a,b,demo(c));
37. return ;
38. }

## 输出形式

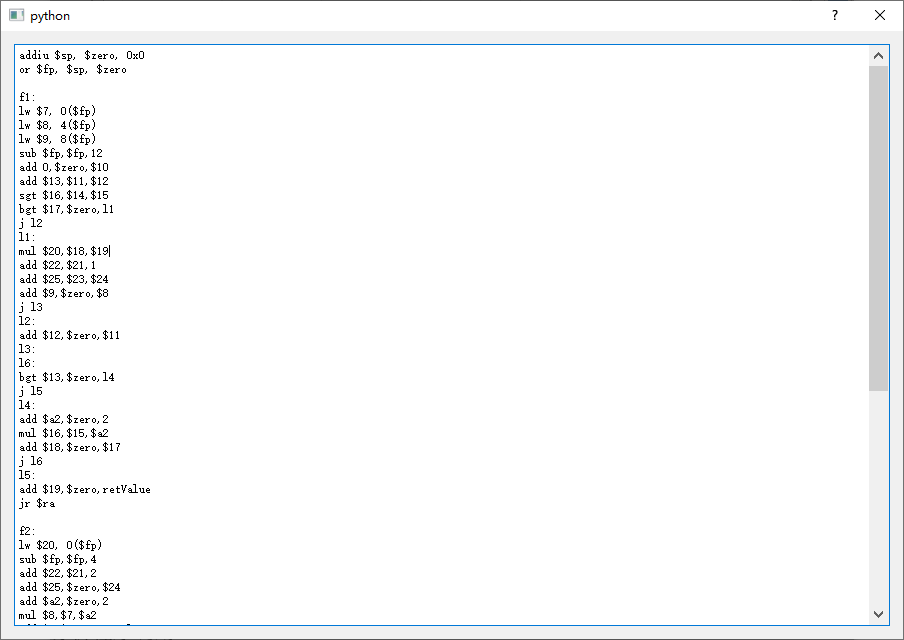
### 输出中间代码表示的程序

示例如下：



### 输出目标代码(可汇编执行)的程序

示例如下：



## 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。

## 测试数据

### 变量重定义

int a;

int a;

### 使用未声明的变量

int main()

{

a = 1;

}

### 使用未定义的函数

int main()

{

int a;

a = demo(1);

return ;

}

### 变量赋值时类型错误

int main()

{

int a;

a = 1;

void b;

b = a;

return 1;

}

### 函数形参和实参不匹配

int demo()

{

return 1;

}

int main()

{

int c;

c = demo(1);

return 1;

}

### 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;

}

# 概要设计

## 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。

### 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 |
| 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 |
| 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：

这一部分是对于语法的处理。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。

### 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 |  |

### 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 |

### 目标代码生成 – ObjCodeGenerator类

模块设计思路与分析说明：

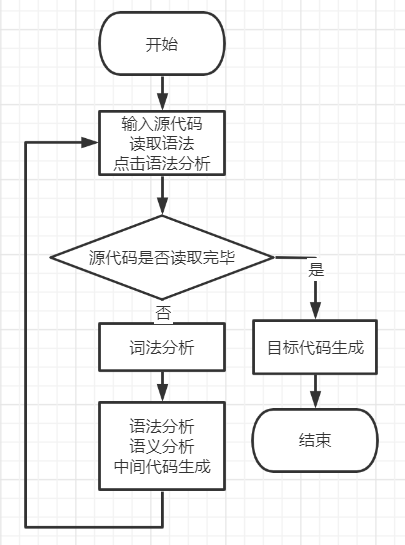
获取中间代码和其它语义分析结果，生成目标代码

|  |  |
| --- | --- |
| 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 |

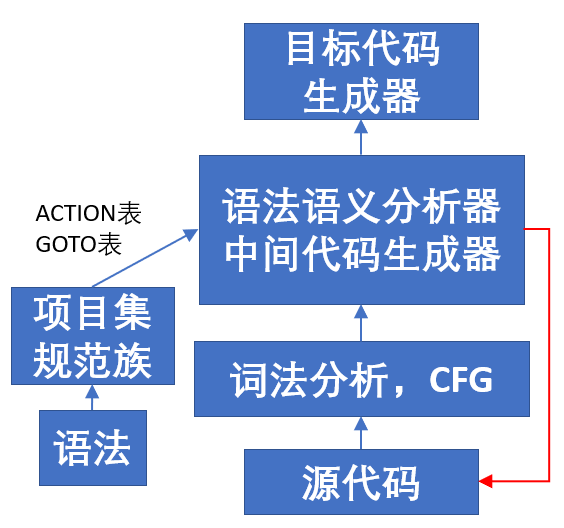
属性成员

|  |  |
| --- | --- |
| 属性名 | 描述 |
| mipsCode | 生成的Mips代码 |
| regTable | 计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 |

## 主程序流程



## 模块间的调用关系



# 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。

## 词法分析

### 实现思路

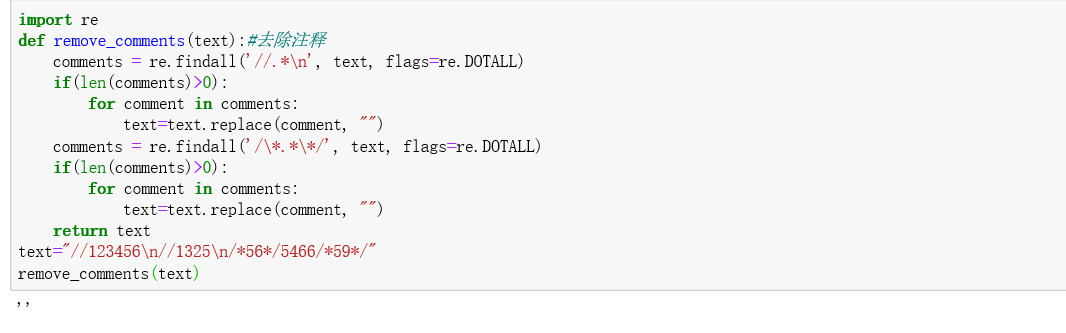
词法分析部分对词法符号的识别主要是通过python提供的正则表达式匹配功能。

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。



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“/\\*.\*?\\*/”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。

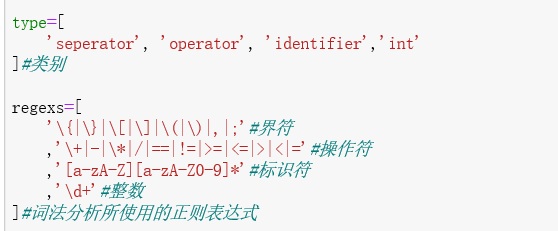
例如贪婪匹配用不加？的正则表达式匹配”//123\n int i=0\n”时的结果会覆盖整个字符串。



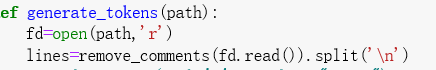
上图删除注释后的理想结果是5466，但由于进行的是贪婪匹配，结果/\*\*/匹配时将5466涉及了进去。所以应该进行非贪婪匹配。

### 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。



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。

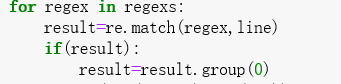


定义了scan\_line(line)函数将一行的词法单元识别，并将判别词法单元的结果存储到一个list作为返回结果中。定义scan（result）来读取并判别result的第一个词法单元。

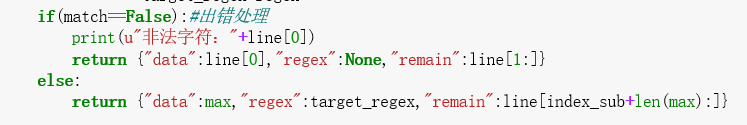
首先通过strip函数将一行字符串开头的空格和制表符去除。

C:\Users\HP\AppData\Local\Temp\1571645563(1).png

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。



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”“regx”“remain”下



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：

开始

输入初步判断类型和具体关键字

是否保留字

是否界符或分隔符

是否整数

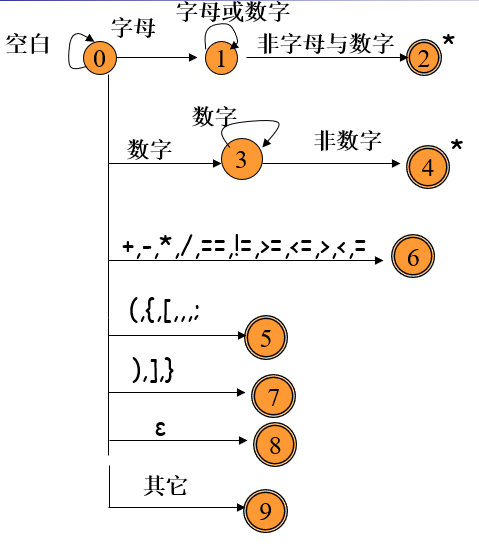
结束

处理保留字

处理界符和分割符

处理整数

其对应的状态转换图如下：



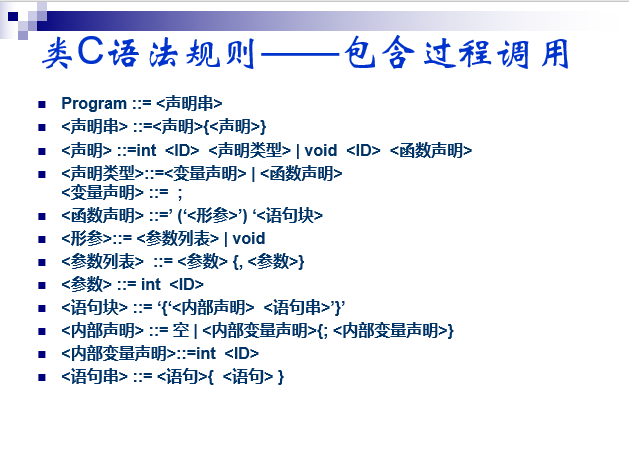
（由老师第三章课件改编而成）

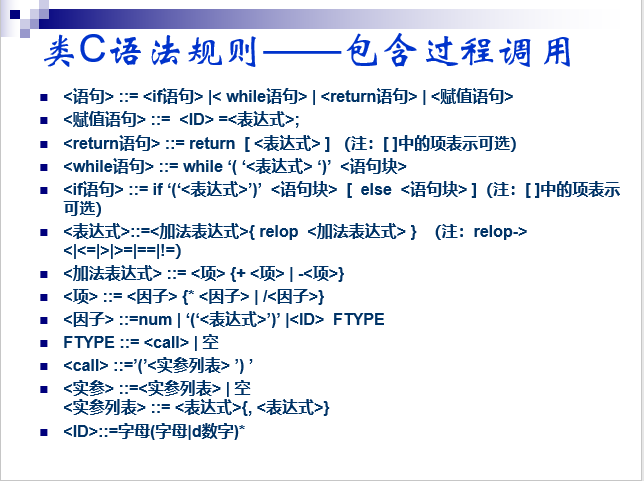
主要流程如下：

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

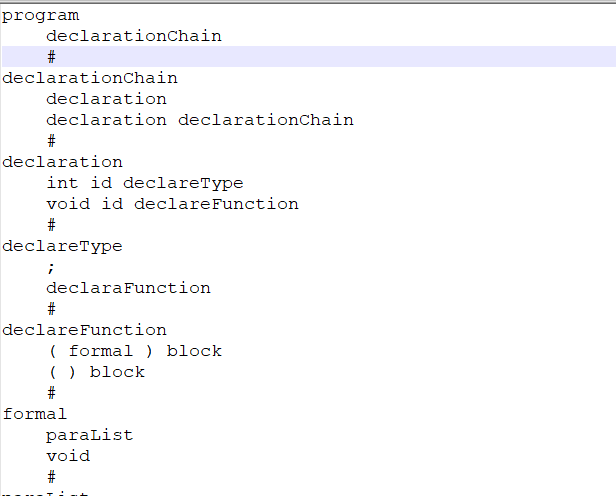
### 词法规则的设计和读取

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。





将语法规则按照一定规则存储在grammer\_final.txt中



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：



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。

## LR1语法分析设计

### 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→a.Bβ,a]属于CLOSURE(I)，则对于任何B→ξ，以及FIRST<βa>中的每一个终结符b，项目[β→.ξ,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＝{任何形如A→αX·β的项目| A→α· Xβ属于I}。

LR(1)项目集族的构建

初始项目集：I0

从I0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→·a, b]属于Ik且GO(Ik,a)＝Ij，a为终结符，则置ACTION[k, a]为“sj”。

(2)若项目[A→·a]属于Ik，则置ACTION[k,a]为“rj”；其中假定A→为文法G的第j个产生式。

(3)若项目[S→S·,#]属于Ik，则置ACTION[k,#]为“acc”。

(4)若GO(Ik，A)＝Ij，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: ( S0, #, a1a2…# )

某一步: (S0S1 . . . Sm, #X1X2 . . . Xm, ai ai+1…an #)

下一步: ACTION [Sm, ai ]

1.若 ACTION [Sm, ai ]为“移进”且GOTO [Sm, ai ] = S,则三元式为：

( S0S1…Sm S, #X1X2 …Xmai , ai +1…an# )

2.若 ACTION[Sm, ai]为“归约”{ A→β}, 且│β│= r ,β=Xm-r+1…Xm, GOTO[Sm-r, A]=S, 则三元式为：

(S0S1…Sm-r S, #X1X2…Xm-r A, ai ai+1…an # )

3.若ACTION[Sm, ai]为“接受”则结束

4.若ACTION[Sm, ai]为“报错”则进行出错处理

### 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：

这一部分是对于语法的处理。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。

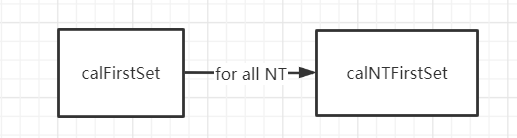
对于单个所有符号（包括终结符和非终结符）都有First集。在之后的语法分析步骤中，需要计算一个字符串的First集，只需要根据规则调用单个字符的First集计算即可。“cal”指的是calculate，计算，calFirstSet将单个字符的First集结果保存供后续分析使用。

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∈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**
9. is\_add = 1
10. **while**(is\_add):      # 必须！
11. is\_add = 0
12. **for** prod **in** prods:
13. hasEpsAllBefore = 0 # state 0
15. **for** right **in** prod.right:
16. # 2. 若X∈VN，且有产生式X→a…，a∈VT，
17. # 则 a∈FIRST(X)  X→ε,则ε∈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->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
25. **break**
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'])
34. # X→Y…是一个产生式且Y ∈VN  则把FIRST(Y)中的所有非空符号串ε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
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
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Yi) 能够推导出ε,(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
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。

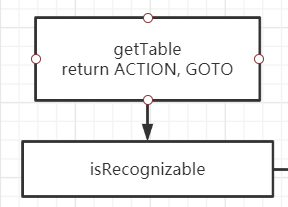
SyntacticAnalyzer中的属性和方法描述如下:

|  |  |
| --- | --- |
| 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。

## 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。

### 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。

### 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->integar | 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->integar |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。

### 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 OP Z

其中x，y，z为名字，常数或编译时产生的临时变量；op代表运算符号如定点运算符，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：

T1=y \* z

T2=x + T1

其中T1，T2位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。这四个域通畅被称为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 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：

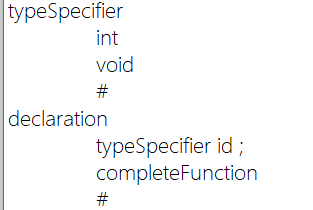
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **序号** | **op** | **arg1** | **arg2** | **result** |
| 0 | uminus | c |  | T1 |
| 1 | \* | b | T1 | T2 |
| 2 | uminus | c |  | T3 |
| 3 | \* | b | T3 | T4 |
| 4 | + | T2 | T4 | T5 |
| 5 | := | T5 |  | a |

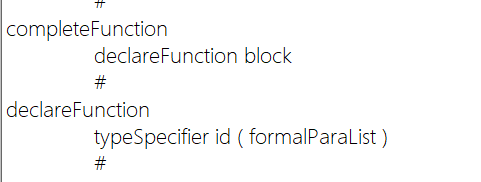
其中Ti（i=1,2..5）存放的是表达式运算得到的中间变量。

### 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





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;

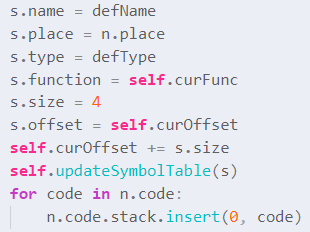
在declaration->typeSpecifier id 产生式对应语义动作

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。



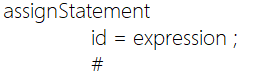
函数声明

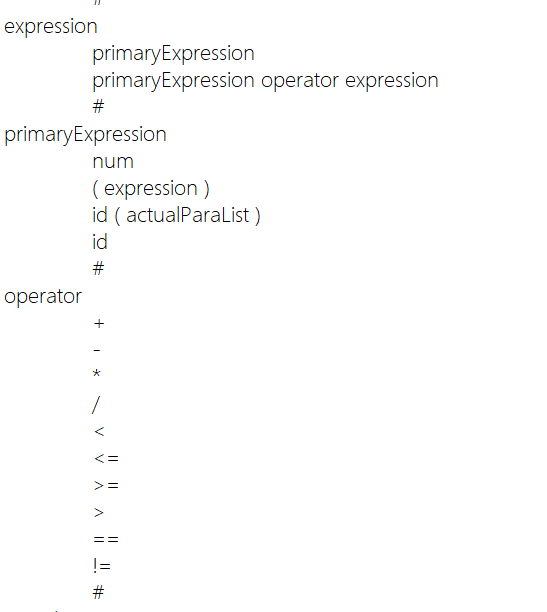
在函数声明的归约中，首先检查函数表是否存在着id.name的函数，如果已经存在则说明函数重定义，输出错误提示。否则将在函数表中插入相应的函数名称和参数列表。

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foramlParaList.stack来进行函数的创建。



赋值语句





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：

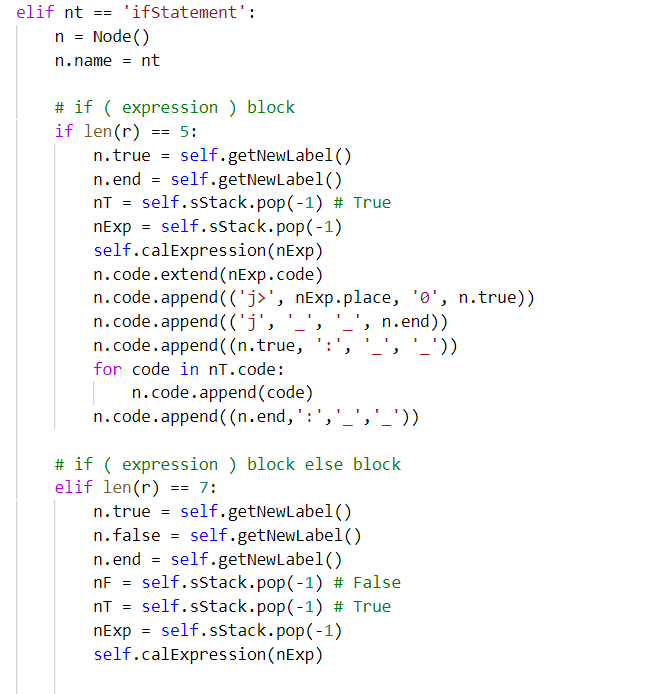
primaryExpression->id (actualParaList)

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：



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：



### 模块设计

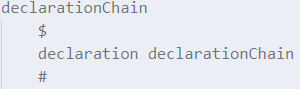
由于在LR1做规约的时候同时做语义分析和中间代码生成，将上文中提到的SyntacticAnalyzer进行扩充，并增加几个类辅助分析，主要成员和函数如下。

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，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：

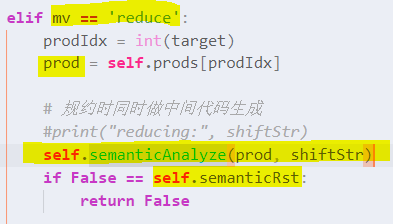
|  |  |
| --- | --- |
| 方法的名称 | 功能描述 |
| semanticAnalyze | 语义分析和中间代码生成 |
| findSymbol | 在符号表查找符号 |
| updateSymbolTable | 更新符号表 |
| updateFuncTable | 更新函数表 |
| findFuncSymbolByName | 查找函数表 |
| calExpression | 将表达式转换为中间代码 |

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，



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:

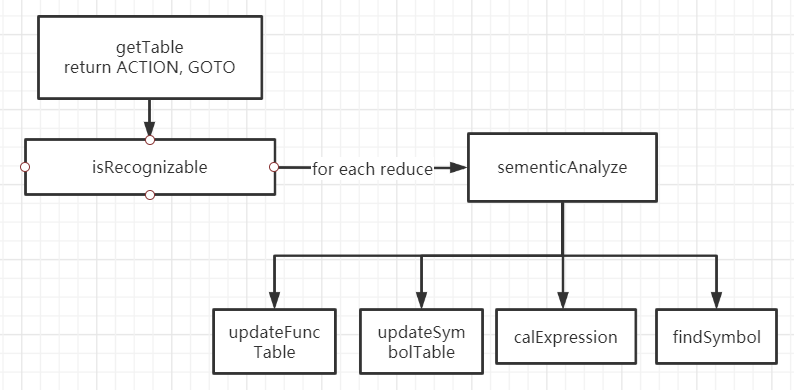
使用assign在左部的产生式对应的语义规则进行中间代码生成…

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：



## 目标代码生成设计

模块设计思路与分析说明：

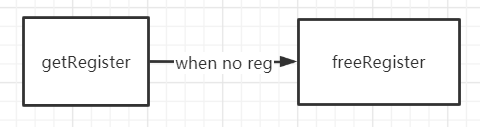
获取中间代码和其它语义分析结果，生成目标代码

|  |  |
| --- | --- |
| 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 | 计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. *#print(varRegUsed)*
6. *# 统计这些变量后续的使用情况*
7. varUsageCnts = {}
8. for code in codes:
9. *#print(code)*
10. for item in code:
11. *#print(item)*
12. tmp = str(item)
13. if tmp[0] == 't': *# 是个变量*
14. if tmp in varRegUsed:
15. if tmp in varUsageCnts:
16. varUsageCnts[tmp] += 1
17. else:
18. varUsageCnts[tmp] = 1
20. print('===\n', 'varUsageCnts:', varUsageCnts, '\n===\n')
22. sys.stdout.flush()
23. flag = False
24. *# 找出之后不会使用的变量所在的寄存器*
25. for var in varRegUsed:
26. if var not in varUsageCnts:
27. for reg in self.regTable:
28. if self.regTable[reg] == var:
29. self.regTable[reg] = ''
30. self.varStatus[var] = 'memory'
31. flag = True
32. if flag:
33. return
34. *# 释放最少使用的寄存器，*
35. sorted(varUsageCnts.items(), key=lambda x:x[1])
36. varFreed = list(varUsageCnts.keys())[0]
37. for reg in self.regTable:
38. if self.regTable[reg] == varFreed:
39. for item in self.symbolTable:
40. if item.place == varFreed: *# t1, t2, ...*
41. self.mipsCode.append('addi $at, $zero, 0x{}'.format(self.DATA\_SEGMENT))
42. self.mipsCode.append('sw {}, {}($at)'.format(reg, item.offset))
43. self.regTable[reg] = ''
44. self.varStatus[varFreed] = 'memory'
45. return
46. 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，所以需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*

## 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。



$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，这本不代表在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个参数。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。

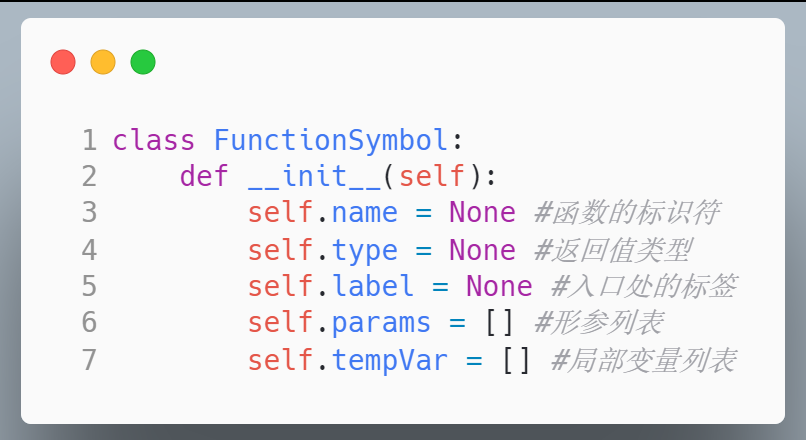
调用者不需要将返回值的位置压入栈中。寄存器 $v0 和 $v1 来保留返回值。当被调用者计算出返回值时，将其保存在寄存器 $v0（如果需要，和 $v1 中）。

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。

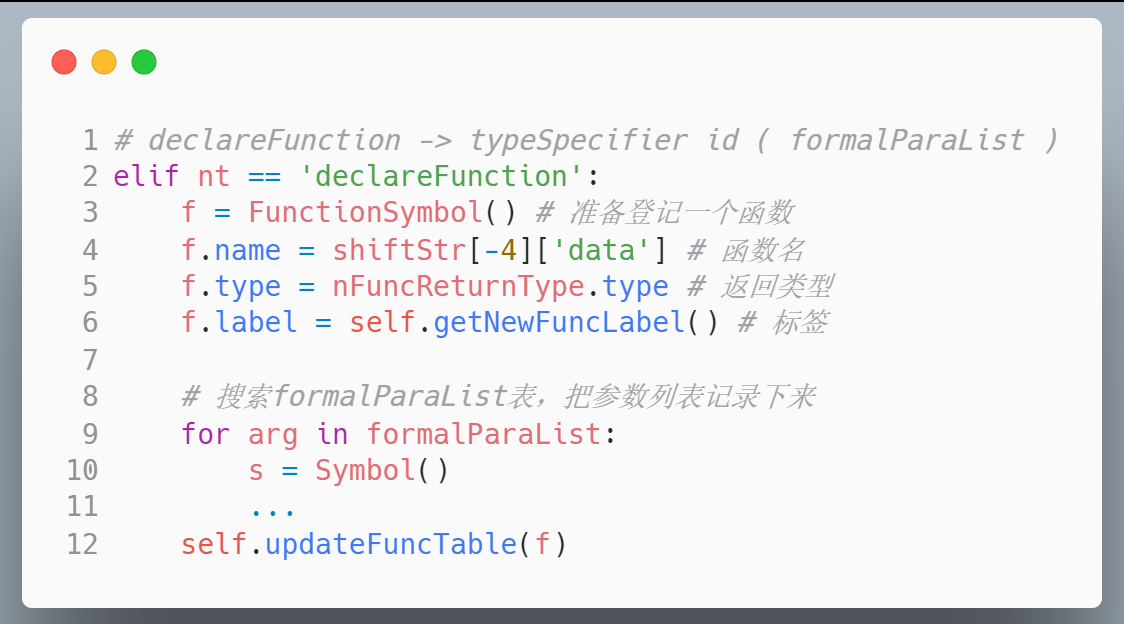
首先是过程调用的数据结构设计。



对此，需要给SyntacticAnalyzer新增方法

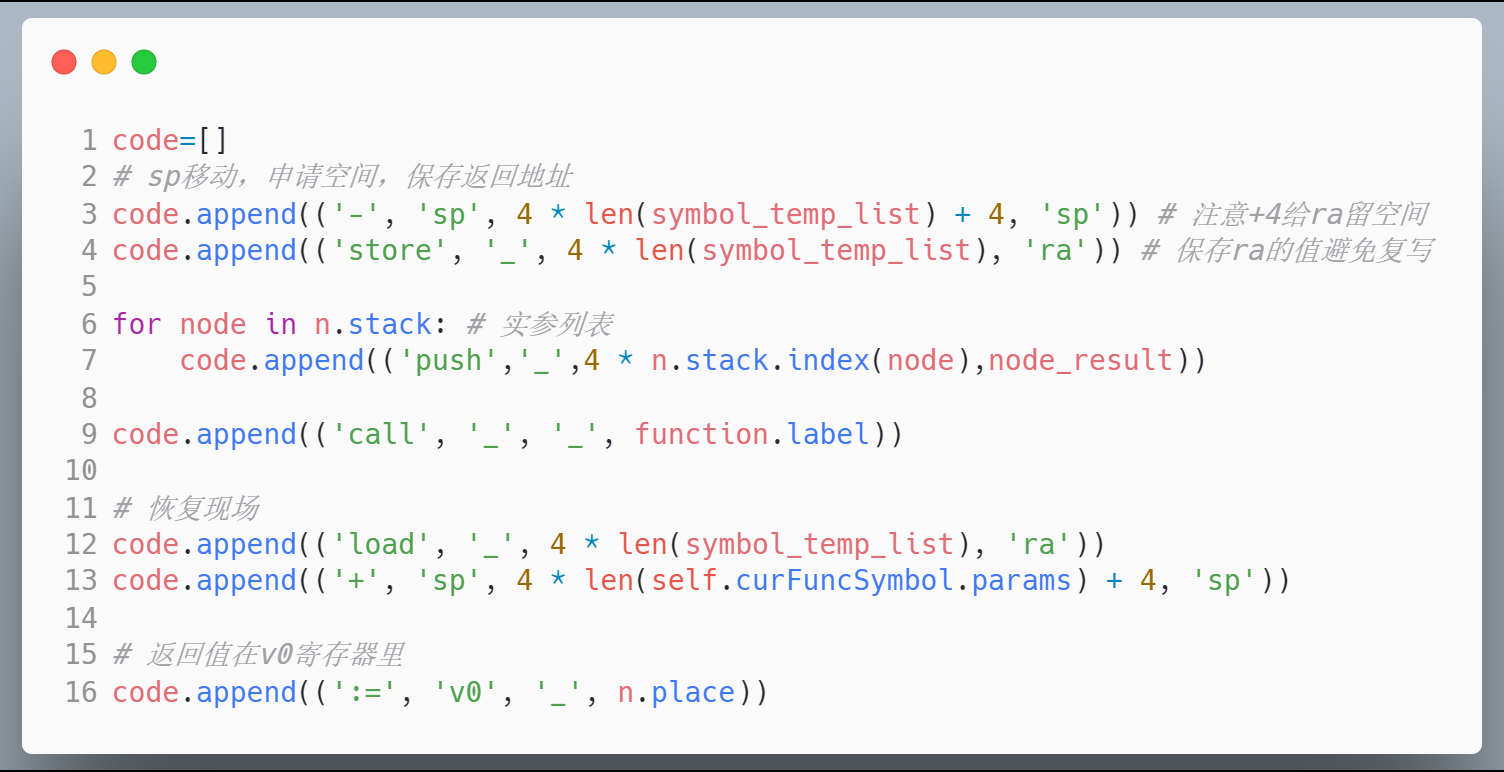
|  |  |
| --- | --- |
| 成员名 | 描述 |
| updateFuncTable | 更新函数表 |
| getNewFuncLabel | 获取函数符号 |
| findFuncSymbolByName | 由函数名获得函数符号 |

对于函数定义，根据产生式，主体代码如下：





而对于函数调用，有：



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。

# 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**

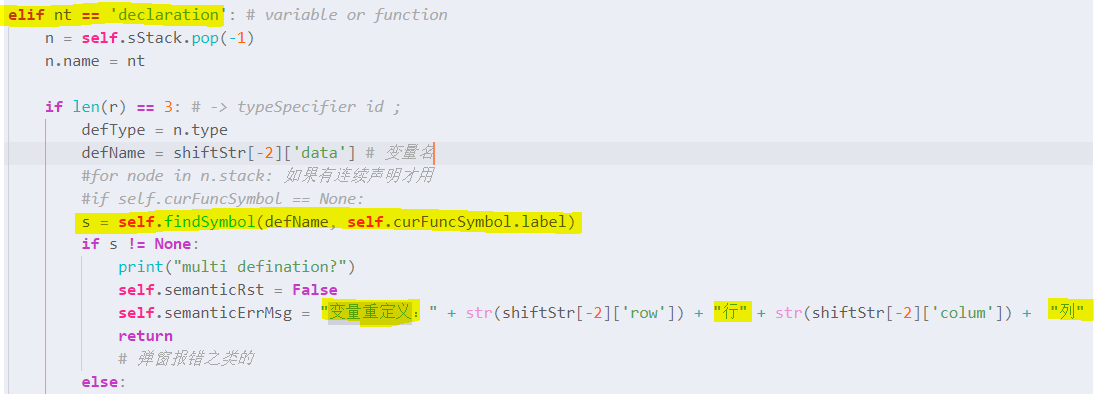
**使用未定义的函数**

**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：

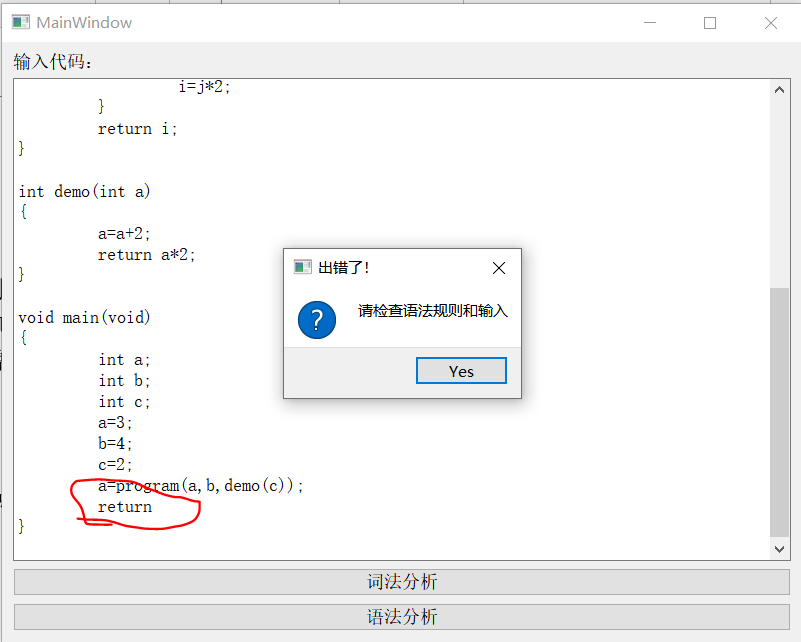


说明：在变量声明语句中，会根据变量名查找是否已经定义过该元素了。如果重定义，将会输出错误信息、行数列数，设置错误标记并返回。

其它的错误诊断和处理与这种模式类似。

## 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：



### 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。

### 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。

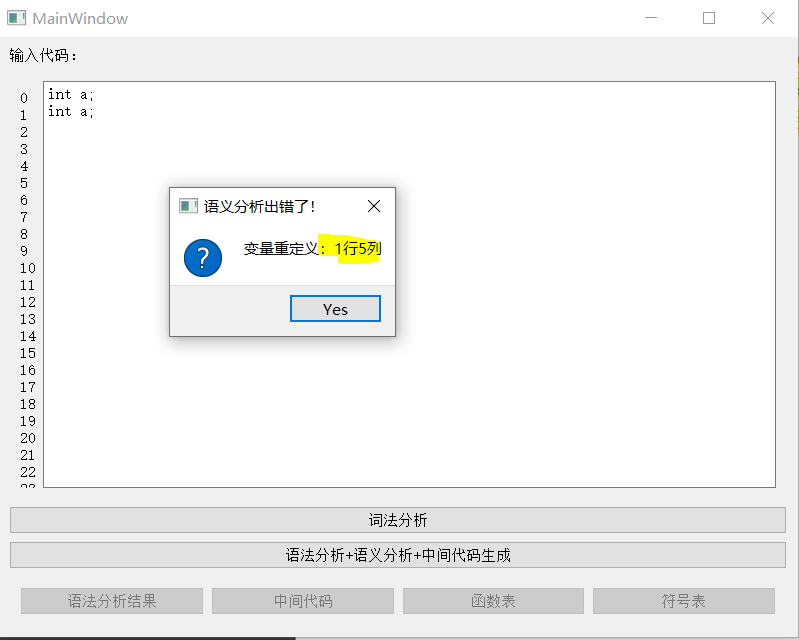
这一步其实相当重要，因为往往是在语义分析的时候才知道自己原来设计的语法有多烂，做语义分析有多别扭。好的语法设计是为语义分析省事。

## 静态语义测试

### 变量重定义

int a;

int a;



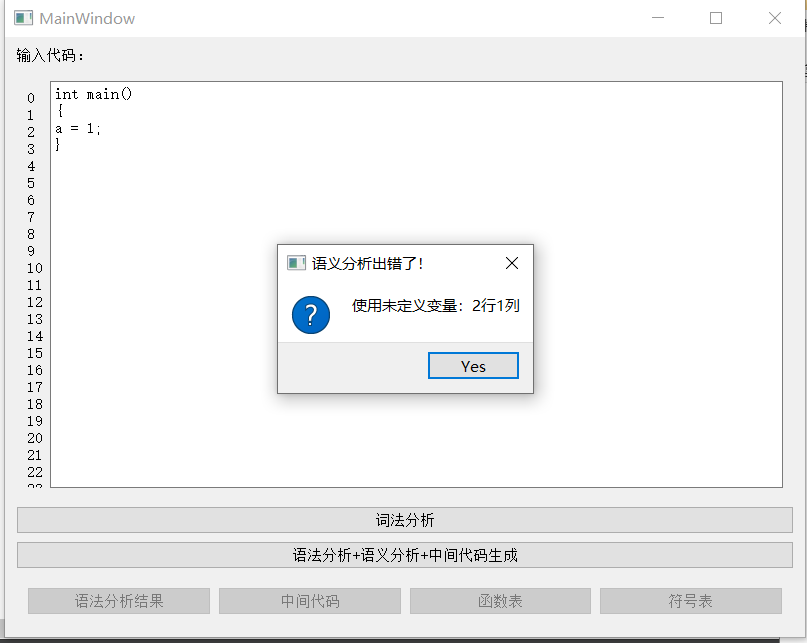
### 使用未声明的变量

int main()

{

a = 1;

}



### 使用未定义的函数

int main()

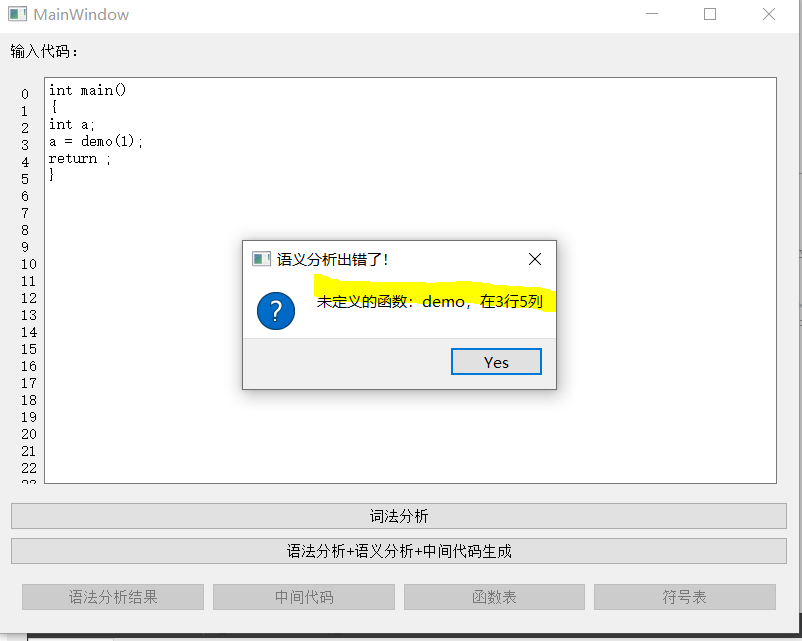
{

int a;

a = demo(1);

return ;

}



### 变量赋值时类型错误

int main()

{

int a;

a = 1;

void b;

b = a;

return 1;

}



### 函数形参和实参不匹配

int demo()

{

return 1;

}

int main()

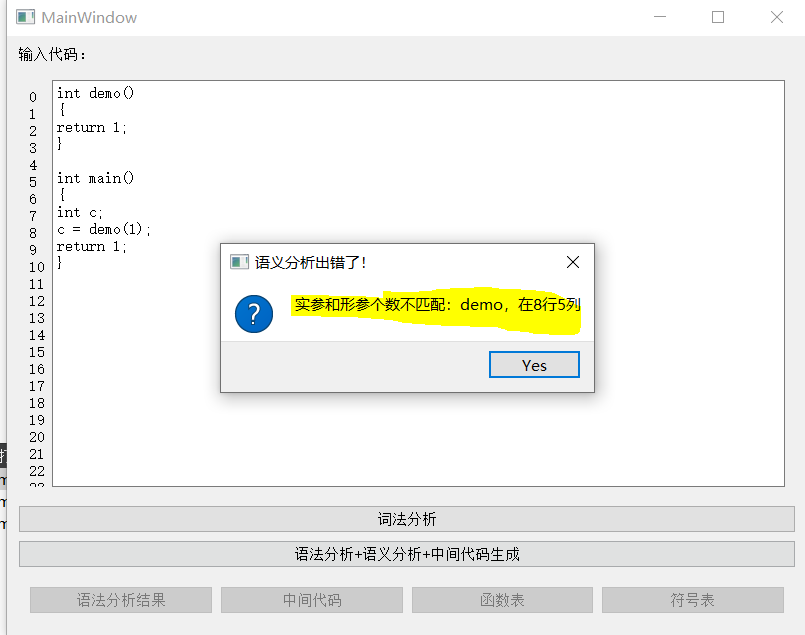
{

int c;

c = demo(1);

return 1;

}



### 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。

### 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。

## 目标代码生成测试

### 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。

### 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。

## 存在的问题，思考与解决

### 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和尝试，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->.ε和A->ε.等价于A->.

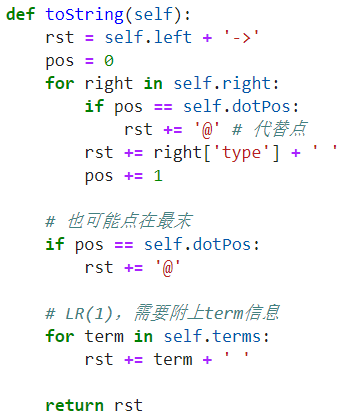
最后reduce的时候，规约栈要压入A，状态栈压入GO(I,A)所到达的状态

经过反复尝试，做到以上三点，LR(1)即具备了较强的能力。

### 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：



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



每次要加入新的item时，就用它的string在记录中比对：



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。

### 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->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->Aa的first求解陷入死循环，并且对于求解它的first没有意义：

A -> Aa

First(A) = First(A),无意义

3. 考虑：可能存在A->eps，使得First集能进一步扩大，且顺序有一定影响

A->eps

A->AB

或

A->AB

A->eps

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->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。所以就需要while多次，直到没有一个NT的first集能再扩大为止。

### 语法更改，消除需要继承属性的语义规则

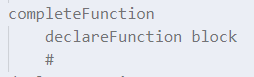
我们知道，S属性文法是只有综合属性的文法。而在语法分析的时候，还对语义分析和中间代码生成没有比较好的认识，故需要重构语法。

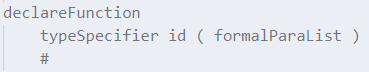
重构方法在于，每次都用相同的代码测试，改动一点语法就运行一次，确保每次改动都是正确的。

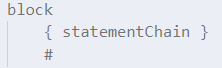
首先，正确的LR（1）文法是不会重复填写GOTO表和ACTION表的，故需要在每次填写两张表的时候判断填写时是否为空，若非空则报错，放弃这次语法的改动。当然这个要确保First集的计算是绝对无误的，GOTO和ACTION的填写是绝对无误的（某种意义上这个是伪命题）。

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：



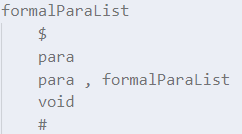


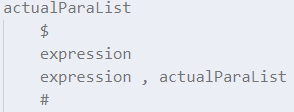


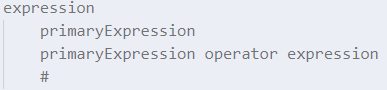
这三个语法完成一个函数的定义。但是为什么不一个产生式完事，非要分成函数返回值（typeSpecifier）、函数名（id）、形参列表（formalParaList）一个产生式，和花括号的语句块一个产生式呢？

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：





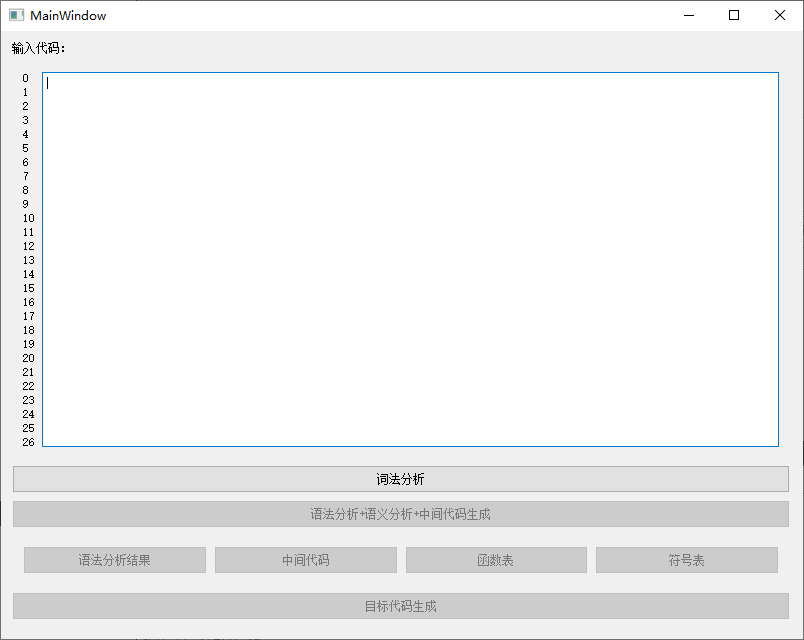


将它们的中间结果存在Node.stack里，等待formalParaList被它的上一级产生式，如declareFunction -> typeSpecifier id ( formalParaList )规约时，再进行formalParaList的翻译。将para从formalParaList节点的stack中取出来，逐个翻译。

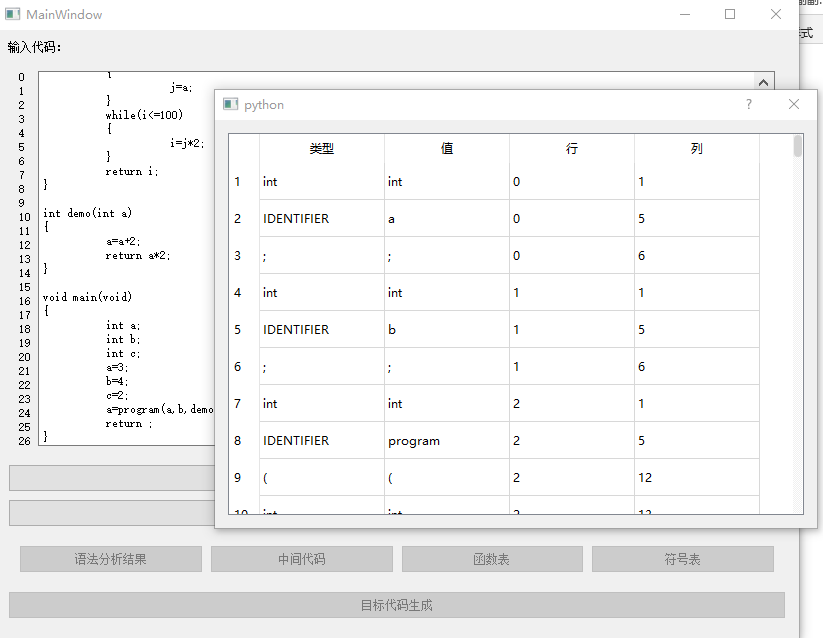
以上就是主要的语法更改。最后语法精简到18个非终结符。

# 用户使用说明

初始界面如下：

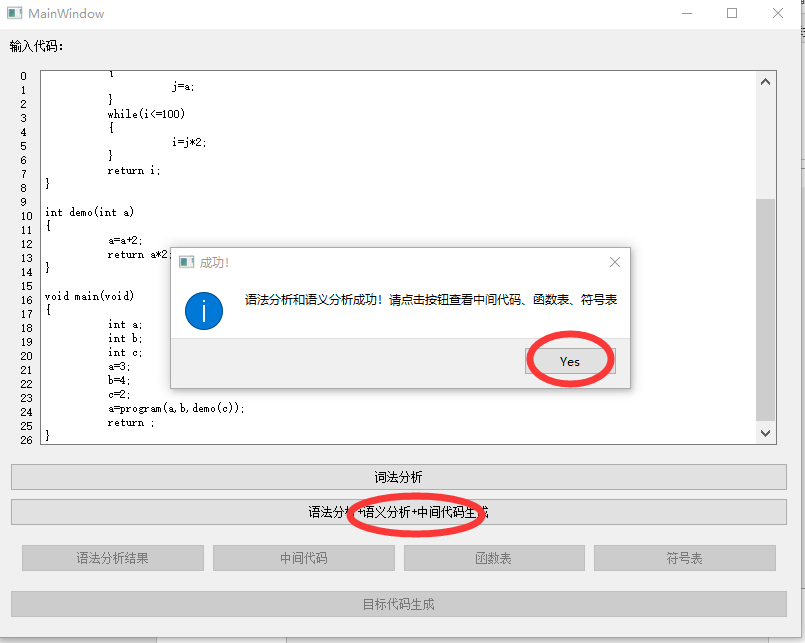


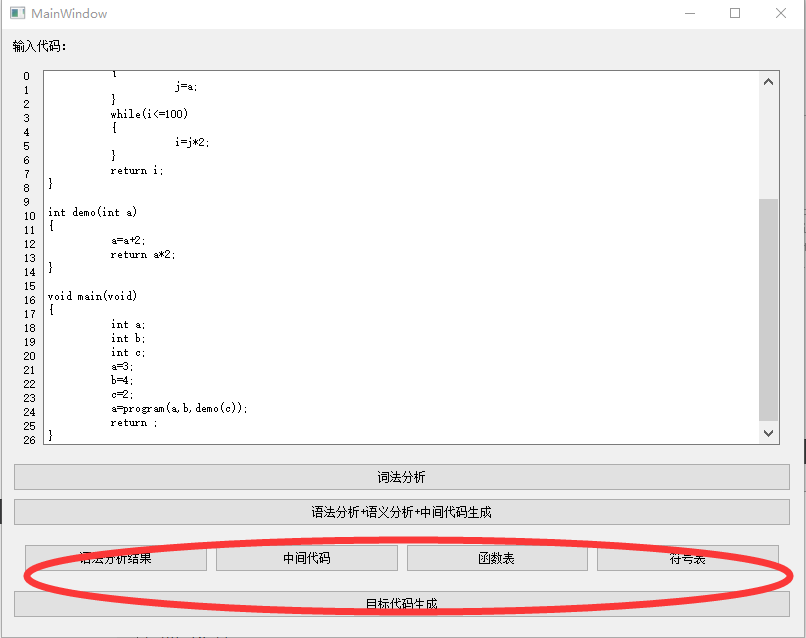
然后放入带过程调用的代码：



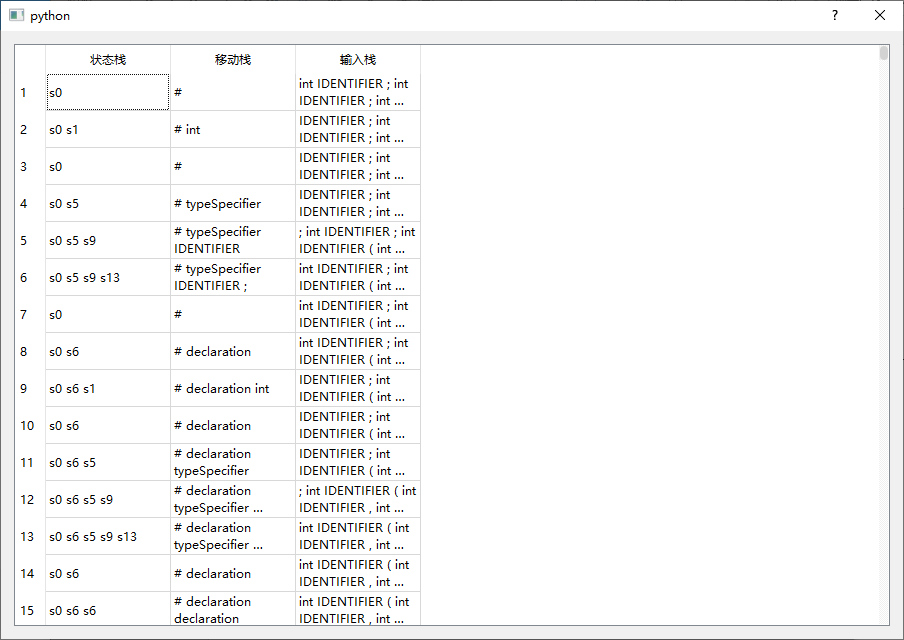
点击词法分析的按钮，弹出词法分析结果：

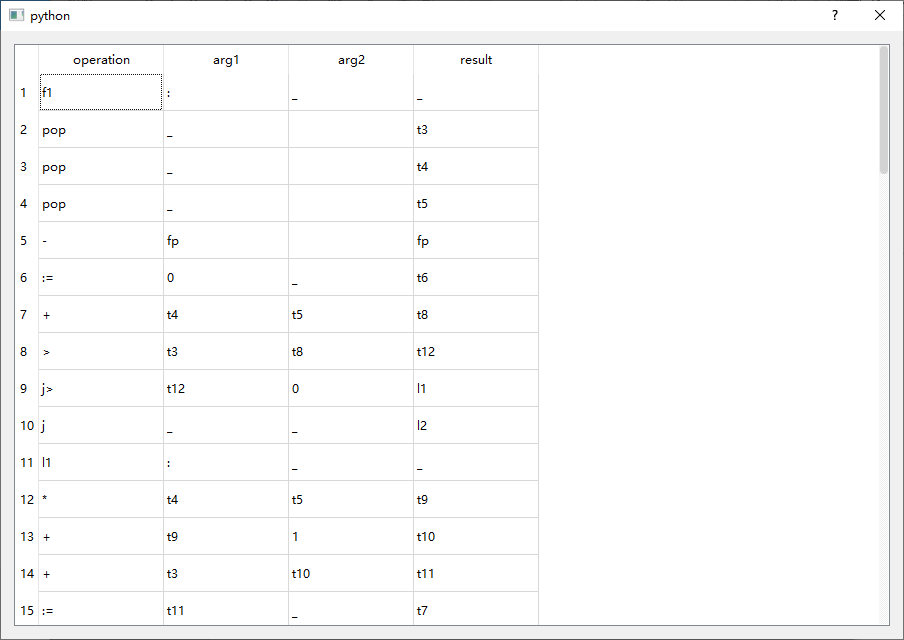
点击“语法分析+语义分析+中间代码生成”，释放下面四个按钮的使用





然后点击四个新生成的按钮，可以查看结果，从左到右依次：

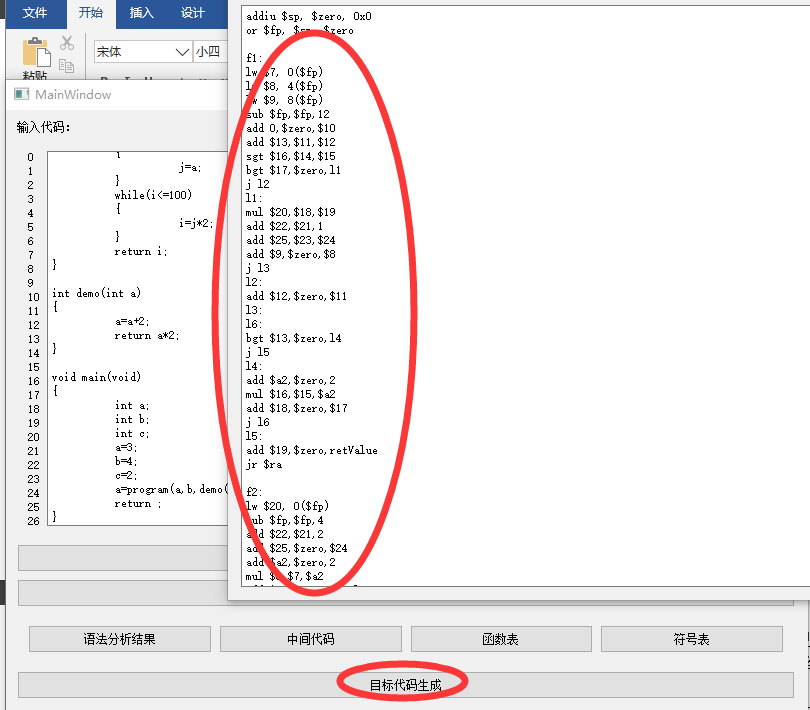




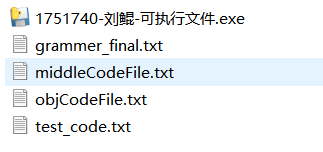




点击目标代码生成，生成代码：



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。



# 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。

# 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

# 参考文献

1. 自底向上分析 (下)[EB/OL]. <https://pandolia.net/tinyc/ch12_buttom_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,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.