设计题目：一个简单文法的编译器实现

设计分工

摘　要

整个编译器划分为这几块，词法分析，语法分析，L 属性自顶向下翻译语义分析中间代码生成，生成的中间代码划分基本块优化后成目标代码生成，其中词法分析和语法分析过程完成符号表的初始化，并检测代码中存在的一些错误。

我们小组完成的课设实现了基于 LL1 分析将类 C 语言翻译成汇编语言的功能，并用网站的形式进行功能的使用展示。符号表中支持的数据类型是整型，整型数组，结构体（结构体中可以嵌入整型，整型数组），并且区分局部变量。文法支持函数定义，函数嵌套，支持 if else（elif）语句，支持 while 循环语句及嵌套，运算表达式。根据参数类型生成汇编时判断传递值或地址。支持多种代码错误检测包括：不符合文法规则，变量函数名重复定义，函数传参数量类型不匹配，变量使用未定义检测。

具体细节，词法分析部分将输入的源代码转换成 token 序列，接着根据文法规则计算 LL1 分析表，通过 LL1 分析法对 token 序列进行分析判定是否符合文法规则并填写符号表同时进行上一段中的几种错误机制检查，并且精确到错误的具体行数进行错误的显示。若代码没有错误则通过翻译文法并根据 LL1 分析表将其转换成相应的四元式。然后将四元式划分基本块进行优化，包括常数运算直接计算结果，删除多余运算，无用赋值等。填写活跃信息，跳转信息与符号表结合转换成目标的汇编代码。最终通过网站的形式对此课设进行展示网站链接：

http://justyan.top/compiler/index。

关键词：编译原理，前端，编译器后端，LL1，汇编语言

目　录

1 概述

在计算机上执行一个高级语言程序一般需要分为两步：第一步，用一个编译程序把高级语言翻译成机器语言程序；第二步，运行所得的机器语言程序求得计算结果。编译原理课设则从实践的角度让我们自己设计文法，将课本教授的理论知识用代码实现，实现执行高级程序的第一步。整个过程需要自己设计数据结构，设计流程，是一次锻炼团队的合作能力的机会与挑战。

本次我们尝试将自定义一个类 C 语言的文法通过自顶向下的 LL1 分析法翻译成汇编语言。设计的主要步骤如下：

1. 符合 LL1 的文法设计，包块循环，判断，函数，数组，结构体等常用语句。
2. 词法分析，识别关键字、界符、常数、标识符，生成包含位置信息的 token 序列。
3. 符号表设计，保存用户自定义变量，函数，结构体，偏移信息等内容
4. 语法分析，根据 token 序列判断输入的代码是否符合文法规则，并进行变量重复定义，未定义，函数传参等问题的检测。
5. 语义分析，设计翻译文法，将 token 序列转换成相应的四元式
6. 中间代码优化，先划分基本块，填写活跃信息，跳转信息，将临时变量在栈中开辟空间，检测变量是地址还是值，在符号表中添加相应信息。
7. 目标代码生成，生成汇编代码
8. 网站展示设计，将程序做成后端

2 课程设计任务及要求

2.1 设计目的

编译原理课程兼有很强的理论性和实践性，是计算机专业的一门非常重要的专业基础课程，在系统软件中占有十分重要的地位。编译原理课程设计是本课程重要的综合实践教学环节，是对平时实验的一个补充。通过编译器相关子系统的设计，使学生能够更好地掌握编译原理的基本理论和编译程序构造的基本方法和技巧，融会贯通本课程所学专业理论知识；培养学生独立分析问题、解决问题的能力，以及系统软件设计的能力；培养学生的创新能力及团队协作精神。

2.2 设计内容

在下列内容中任选其一：

1、一个简单文法的编译器前端的设计与实现。

2、一个简单文法的编译器后端的设计与实现。

3、一个简单文法的编译器的设计与实现。。

4、自选一个感兴趣的与编译原理有关的问题加以实现，要求难度相当。

2.3 设计要求

1、在深入理解编译原理基本原理的基础上，对于选定的题目，以小组为单位，先确定设计方案；

2、设计系统的数据结构和程序结构，设计每个模块的处理流程。要求设计合理；

3、编程序实现系统，要求实现可视化的运行界面，界面应清楚地反映出系统的运行结果；

4、确定测试方案，选择测试用例，对系统进行测试；

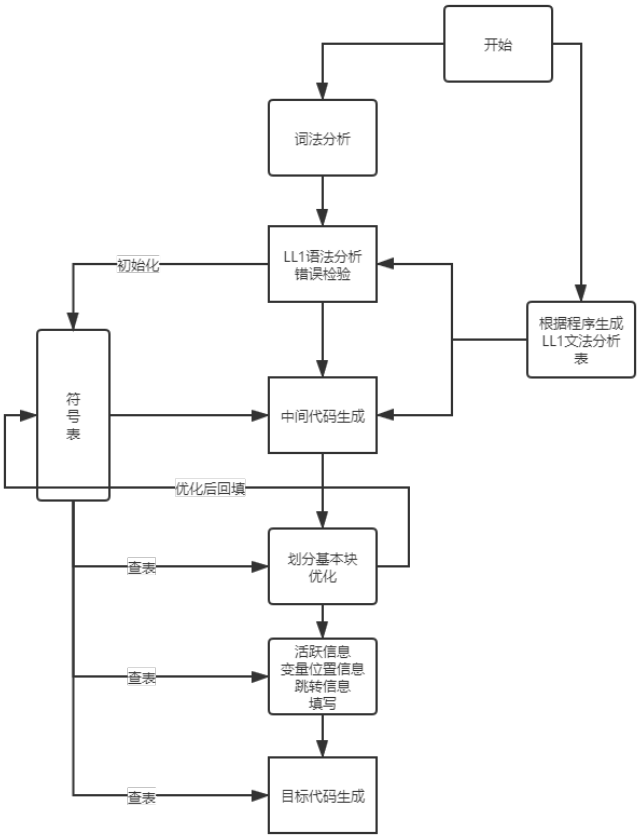
5、运行系统并要通过验收，讲解运行结果，说明系统的特色和创新之处，并回答指导教师的提问；

6、提交课程设计报告。

3 编译系统总体设计

3.1 编译器结构设计

结构图：



3.2 文法设计

```
Program->Struct Funcs
Funcs->FuncsHead { CodeBody } Funcs
FuncsHead->Type ID ( FormalParameters )
Funcs->$
FormalParameters->Type ID FormalParametersFollow
FormalParameters->$
```

```
FormalParametersFollow->, Type ID FormalParametersFollow
FormalParametersFollow->$
Type->int
Type->void
Type->float
Type->char
Type->st
CodeBody->$
CodeBody->LocalDefineList CodeList
LocalDefineList->LocalVarDefine LocalDefineList
LocalDefineList->$
LocalVarDefine->Type ID ;
CodeList->Code CodeList
CodeList->$
Code->NormalStatement
NormalStatement->ID NormalStatementFollow
NormalStatementFollow->= Operation ;
Operation->T A
A->M T A
A->$
T->F B
B->N F B
B->$
F->ID
```

```
F->NUM
F->( Operation )
M->+
M->-
N->*
N->/
Code->IfStatement
IfStatement->if ( JudgeStatement ) { CodeBody }
IFStatementFollow
IFStatementFollow->$
IFStatementFollow->ElseIFPart ElsePart
ElsePart->$
ElsePart->else { CodeBody }
ElseIFPart->elif ( JudgeStatement ) { CodeBody } ElseIFPart
ElseIFPart->$
JudgeStatement->Operation JudgeStatementFollow
JudgeStatementFollow->CompareSymbol Operation
JudgeStatementFollow->$
CompareSymbol->==
CompareSymbol-><=
CompareSymbol->>=
CompareSymbol-><
CompareSymbol->>
Code->LoopStatement
```

```
LoopStatement->while ( JudgeStatement ) { CodeBody }
Code->break ;
Code->continue ;
Code->return Operation ;
Code->FuncCall
FuncCall->CALL ID FuncCallFollow ;
FuncCallFollow->= ID ( Args )
FuncCallFollow->( Args )
Args->F ArgsFollow
ArgsFollow->, F ArgsFollow
Args->$
ArgsFollow->$
Struct->struct st { LocalDefineList } ; Struct
Struct->$
```

Name	Mean	Type
Program	程序入口	VN
Funcs	函数定义	VN
Type	类型	VN

Name	Mean	Type
ID	标识符（非关键字）	VT
FormalParameters	参数列表	VN
CodeBody	代码块	VN
LocalDefineList	变量定义	VN
CodeList	处理语句	VN
Code	语句块	VN
NormalStatement	变量赋值	VN
Operation	表达式语句	VN
IfStatement	If 语句	VN
JudgeStatement	判断	VN
IFStatementFollow	else or elif	VN

Name	Mean	Type
ElsePart	else	VN
ElseIFPart	elif	VN
CompareSymbol	比较符号	VN
LoopStatement	循环语句	VN
FuncCall	函数调用	VN
Args	参数	VN
Struct	结构体	VN
st	结构体变量定义的时候规约使用	VT

文法支持结构体定义，while 语句，if else 语句，函数定义，函数调用，算术表达式，不支持函数内定义函数，其中 while，if else 之间可以相互嵌套。一个函数，结构体是一个变量存在的基本块（与四元式中的基本块不同），结构体中只支持变量声明，函数中支持变量声明，变量赋值，

if，while 语句，调用函数等语句，所有的语句必须存在于声明的函数或者主函数中。

翻译文法：

Program->Struct Funcs

Funcs->FuncsHead { CodeBody } Funcs

FuncsHead->Type ID @PUSH_VAL @SAVE_FUN @GEQ_G

(FormalParameters)

Funcs->$

FormalParameters->Type ID FormalParametersFollow

FormalParameters->$

FormalParametersFollow->, Type ID FormalParametersFollow

FormalParametersFollow->$

Type->int

Type->void

Type->float

Type->char

Type->st

CodeBody->$

CodeBody->LocalDefineList CodeList

LocalDefineList->LocalVarDefine LocalDefineList

LocalDefineList->$

LocalVarDefine->Type ID ;

CodeList->Code CodeList

CodeList->$

```
Code->NormalStatement
NormalStatement->ID @PUSH_VAL NormalStatementFollow
NormalStatementFollow->= @SAVE_= Operation @GEQ_G ;
Operation->T A
A->M T @GEQ_G A
A->$
T->F B
B->N F @GEQ_G B
B->$
F->ID @PUSH_VAL
F->NUM @PUSH_VAL
F->( Operation )
M->+ @SAVE_+
M->- @SAVE_-
N->* @SAVE_*
N->/ @SAVE_/
Code->IfStatement
IfStatement->if @SAVE_if ( JudgeStatement ) @GEQ_G { CodeBody }
@GEQ_el IFStatementFollow @GEQ_ie
IFStatementFollow->$
IFStatementFollow->ElseIFPart ElsePart
ElsePart->$
ElsePart->else { CodeBody }
ElseIFPart->elif @SAVE_elif ( JudgeStatement ) @GEQ_G { CodeBody }
```

@GEQ_el ElseIFPart

ElseIFPart->$

JudgeStatement->Operation JudgeStatementFollow

JudgeStatementFollow->CompareSymbol Operation @GEQ_G

JudgeStatementFollow->$

CompareSymbol->== @SAVE_==

CompareSymbol-><= @SAVE_<=

CompareSymbol->>= @SAVE_>=

CompareSymbol->< @SAVE_<

CompareSymbol->> @SAVE_>

Code->LoopStatement

LoopStatement->while @GEQ_wh @SAVE_do (JudgeStatement)

@GEQ_G { CodeBody } @GEQ_we

Code->break @GEQ_break ;

Code->continue @GEQ_continue ;

Code->return @SAVE_return Operation @GEQ_G ;

Code->FuncCall

FuncCall->CALL ID @PUSH_VAL FuncCallFollow ;

FuncCallFollow->= ID @PUSH_VAL (Args) @SAVE_callr @GEQ_c1

FuncCallFollow->(Args) @SAVE_call @GEQ_c2

Args->F @SAVE_push @GEQ_G ArgsFollow

ArgsFollow->, F @SAVE_push @GEQ_G ArgsFollow

Args->$

ArgsFollow->$

```
Struct->struct st { LocalDefineList } ; Struct
Struct->$
```

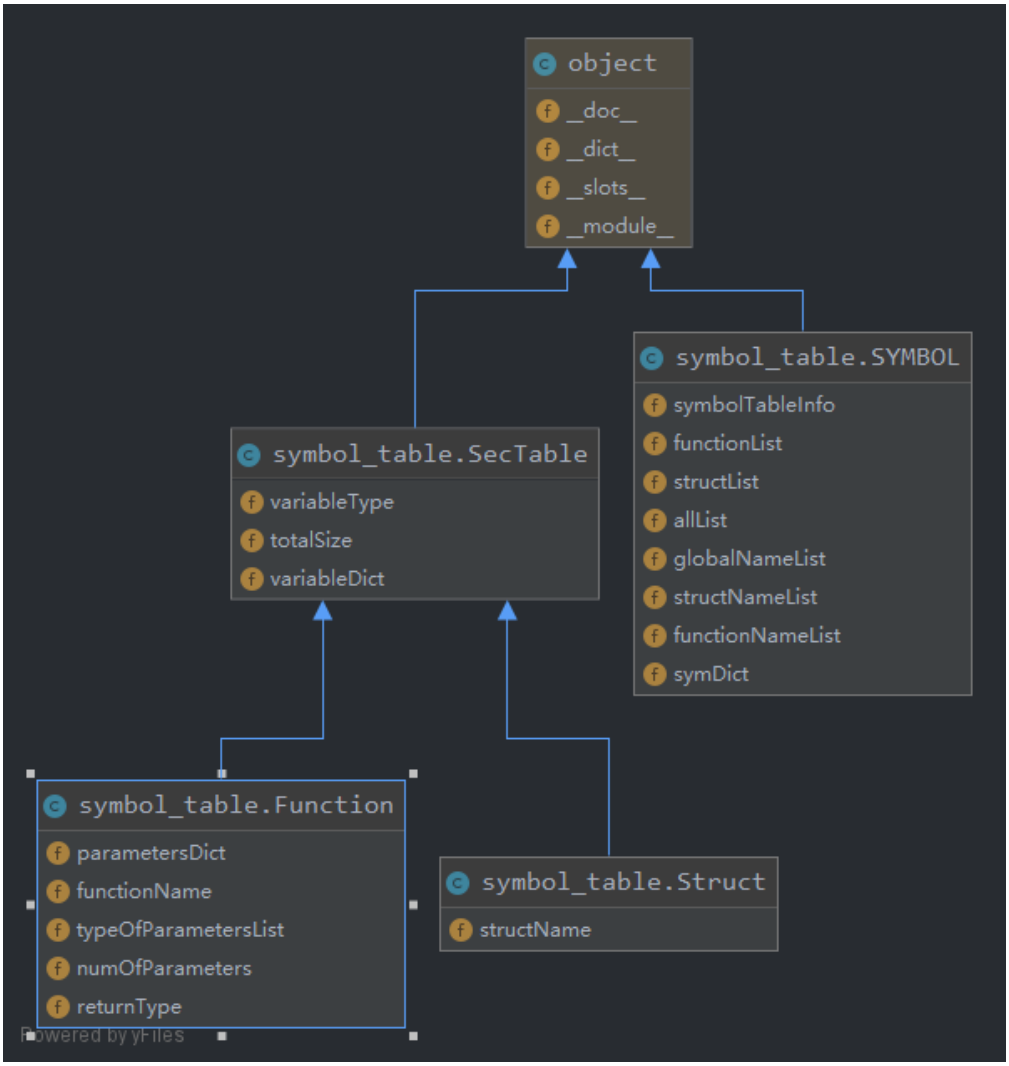
其中@SAVE 用于将于一动作加入 SYMBOL_STACK 中，@PUSH 用于将变量或数字加入 SEM_STACK 中，@GEQ 则根据 SYMBOL_STACK 栈顶的动作弹出，同时弹出 SEM_STACK 中对应数量的操作变量，组合生成四元式。

3.3 符号表设计

符号表分为三块，总表，函数块的符号表，结构体的符号表。总表中存储结构体和函数的实例。各个方法包含是否定义的检测功能。各个变量的声明只存在于其所处的函数块或者是结构体块中，各块内部变量定义不可以重复。同时符号表中有检测变量是否定义，调用函数时参数传递是否正确的方法。符号表的主要初始化完成部分在语法分析，语法分析根据 LL1 分析法，当遇到相应的 token 序列时先将符合要求的 token 传到符号表中进行检测检测没有问题则添加到符号表。主要函数：

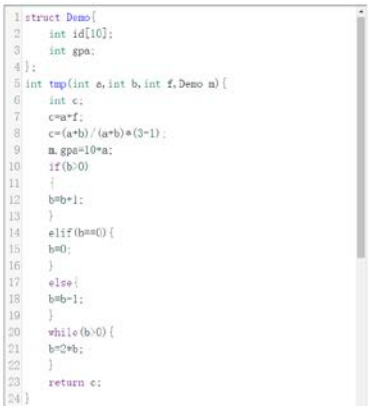
```
def addPaVariable(self, token, typ):    # 添加参数
def addVariable(self, token, typ, s):    # 添加变量
def checkHasDefine(self, token):    # 检查重复定义问题
def checkDoDefine(self, token):    # 检查是变量使用时是否定义
def addFunction(self, token, returnType):    # 添加函数
def addStruct(self, token):    # 添加结构体
```

```
def addVariableToTable(self, token, varType, doseParameter=False):  # 添加新定义的变量
def checkDoDefineInFunction(self, token):  # 检测函数中调用变量的时候变量是否存在
def checkFunction(self,RES_TOKEN, id):  # 调用函数时候检查参数传递是否正确
```



对于变量记录其类型，偏移地址，占用空间则通过偏移地址之差计算，函数中添加函数返回类型，参数数量参数类型这几个字段。总表则用于存放函数块结构体块。偏移地址为负表示参数，也是基于在实际在栈中的位置表示的。

结果展示如图 3-1：



```
struct Demo{
    int id[10];
    int gpa;
};
int tmp(int a,int b,int f,Demo m){
    int c;
    c=a+f;
    c=(a+b)/(a+b)*(3-1);
    m.gpa=10*a;
    if(b>0)
    {
    b=b+1;
    }
    elif(b==0){
    b=0;
    }
    else{
    b=b-1;
    }
    while(b>0){
    b=2*b;
    }
    return c;
}
```

Func/Struct	ReturnType/Var	NumOfParameters/Type	Size/Addr
tmp(function)	int	4	2
-	a	int	-8
-	b	int	-6
-	f	int	-4
-	m	Demo	-2
-	c	int	0
main(function)	int	0	32
-	d	int	0
-	c	int	2
-	de	Demo	4
-	m[3]	array	26
Demo(struct)	-	-	22
-	id[10]	array	0
-	gpa	int	20

Figure 3-1

4 编译前端

4.1 词法分析器

4.1.1 数据结构

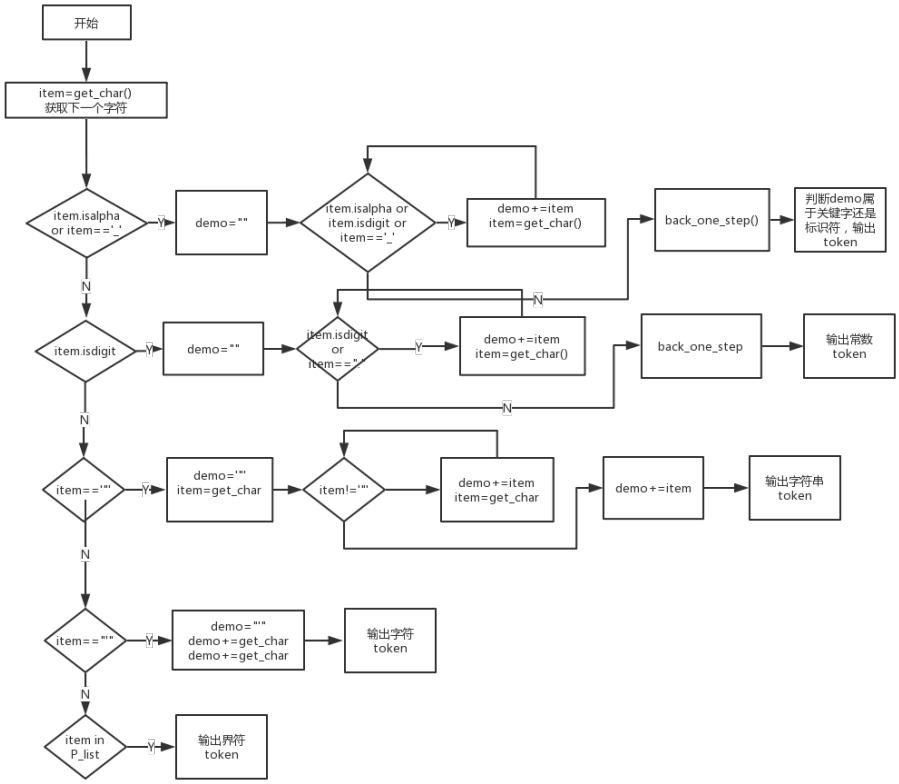
Token = namedtuple('Token', 'type index val cur_line id') 具名元组表示 token 序列 类型 type 中的 id 值 当前行数 序列中的索引。
DICT = { 'k': collections.OrderedDict(), # 关键字 'p': collections.OrderedDict(), # 界符 'con': collections.OrderedDict(), # 常数 'c': collections.OrderedDict(), # 字符 's': collections.OrderedDict(), # 字符串

```
        'i': collections.OrderedDict(),  # 标识符
    }
```

关键函数

def get_char() #获取下一个字符
def back_one_step() #将标识返回到上一个字符
def scanner() #每次生成一个 token 这个函数的流程图如上
def analyse() #调用 scanner 分析

4.1.2 算法设计


开始
item=get_char()
获取下一个字符
item.isalpha
or item=='_'
Y
demo=""
item.isalpha or
item.isdigit or
item=='_'
Y
demo+=item
item=get_char()
back_one_step()
判断demo属
于关键字还是
标识符，输出
token
N
N
item.isdigit
Y
demo=""
item.isdigit
or
item=="."
Y
demo+=item
item=get_char()
back_one_step
输出常数
token
N
item=='"'
Y
demo='"'
item=get_char
item!='"'
demo+=item
item=get_char
demo+=item
输出字符串
token
N
item=="'"
Y
demo="'"
demo+=get_char
demo+=get_char
输出字符
token
N
item in
P_list
Y
输出界符
token

4.1.3 实验结果

```
Token(type='k', index=0, val='struct', cur_line=0, id=0)
Token(type='i', index=0, val='Demo', cur_line=0, id=2)
Token(type='p', index=0, val='{', cur_line=0, id=4)
Token(type='k', index=1, val='int', cur_line=1, id=6)
Token(type='i', index=1, val='id[10]', cur_line=1, id=8)
Token(type='p', index=1, val=';', cur_line=1, id=10)
Token(type='k', index=1, val='int', cur_line=2, id=12)
Token(type='i', index=2, val='gpa', cur_line=2, id=14)
Token(type='p', index=1, val=';', cur_line=2, id=16)
Token(type='p', index=2, val='}', cur_line=3, id=18)
```

4.2 语法分析

4.2.1 数据结构

```
class GrammarParser:
    def __init__(self,path=None):
        self.GRAMMAR_DICT = {}          #文法规则存储，{VN:XXX}
        self.P_LIST = []                #文法规则列表
        self.FIRST = []                 #first 集合
        self.FIRST_VT = {}              # {A:a|A->a...}
        self.FOLLOW = {}                #follow 集合
        self.VN = set()
        self.VT = set()
        self.Z = None
        self.SELECT = []                #select 集合
```

```
self.analysis_table = {}        #分析表
```

这个部分主要计算分析表，后面的语法分析和中间代码生成都继承于这个类。

4.2.2 算法设计

计算 FIRST 集合前先通过递归建立每个 VN 的 FIRST_VT 即 {A：a,b|A->a..,A->b..；A∈VT，a，b∈VN}，接着根据这个字典建立 FIRST 集合。FOLLOW 集合的计算同样也使用 FIRST_VT 这个集合，同样通过递归和一个状态表控制递归。

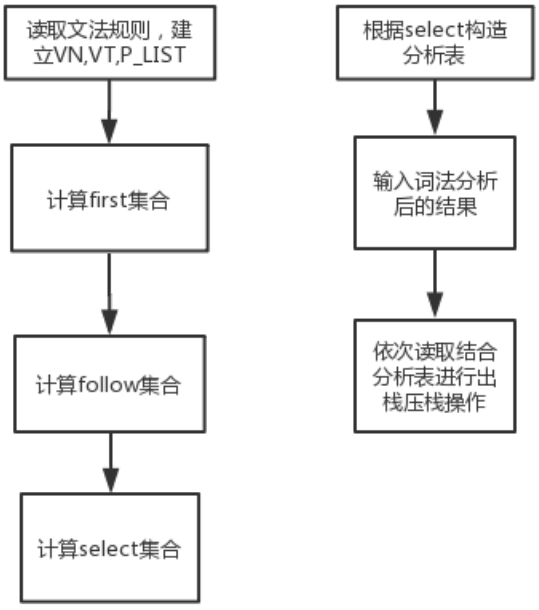
```
class GrammarParser:    主要函数
    def __calFirstvt(self):
    def __calFirst(self):
    def __calFollow(self):
    def __calSelect(self):
    def __calAnalysisTable(self):
class LL1(GrammarParser):
    def analyzeInputString(self): #分析输入的 token 序列
    def editSymTable(self, x, w, token): #编辑符号表
```

LL1 分析部分，从左到右扫描，在分析表中查看栈顶符号和当前符号是否存在，若不存在或是步匹配则转向错误处理，同时这部分对符号表进行初始化，当遇到 token 序列表示变量定义/调用，函数定义/调用的

时候则根据不同的情况转向符号表中相应的函数对符号表进行填写/检查。

具体的错误检测当分析出源程序的错误时可以返回错误类型、出错位置和错误信息。本程序一共包含五种错误类型，分别是：Identifier expected、Unknown identified、Duplicate identified、syntax error、函数参数数量不对。

结构流程图：



4.2.3 实验结果

分析表部分截取展示：后面的数字代表在文法规则列表中的第几个，-1 表示没有关联。

```
"A": {
    "=": -1,
    "return": -1,
    "<": 25,
    "char": -1,
    "st": -1,
    "==": 25,
    "(": -1,
    ">": 25,
    "CALL": -1,
    "else": -1,
    "while": -1,
    ",": -1,
    "ID": -1,
    "}": -1,
    "int": -1,
    ">=": 25,
    "break": -1,
    "-": 24,
    "<=": 25,
```

```
"*": -1,
```

错误检测部分：

错误类型-Identifier expected

```
5   int tmp(int a,int b,int f,Demo m){
6       int c
7       c=a+b;
8       c=(a+b)/(a+b)*(3-1);
9       m.gpa=10+a;
10      if(b>0)
```

（源程序第六行缺少一个分号）

```
Message(ErrorType='Identifier Expected', Location='Location:line 6', ErrorMessage="expect ';' before 'ID'")
```

错误类型—Unknown identified

```
5   int tmp(int a,int b,int f,Demo m){
6
7       c=a+b;
8       c=(a+b)/(a+b)*(3-1);
9       m.gpa=10+a;
```

（源程序第 6 行 c 没有声明）

```
Message(ErrorType='Identifier Expected', Location='Location:line 6', ErrorMessage="expect ';' before 'ID'")
```

错误类型—syntax error

```
5   int tmp(int a,int b,int f,Demo m){
6
7       c=a+b;
8       c=(a+b)/(a+b)*(3-1);
9       m.gpa=10+a;
10      if(b>0
11      {
12      b=b+1;
13      }
```

（源程序第 10 行缺少一个右括号）

```
Message(ErrorType='syntax error', Location='Location:line 10', ErrorMessage="error:expect ['/', '*', '<=', '>', '-', '==', '+', '>=', ')', ';', '<'] after '0' token ")
```

错误类型—Duplicate identified

```
5   int tmp(int a,int b,int f,Demo m){
6       int c;
7       int c;
8       c=a+b;
9       c=(a+b)/(a+b)*(3-1);
10      m.gpa=10+a;
```

（源程序第七行变量 c 重复定义）

```
Message(ErrorType='Duplicate identified', Location='Location:line 7', ErrorMessage="variable 'c' duplicate definition")
```

错误类型—函数参数的数量不对

```
50      }
51      CALL c=tmp(c,d,d);
52      return 0;
53  }
```

（51 行参数只有 3 个，需要 4 个）

```
Message(ErrorType='function argument number', Location='Location:line 50', ErrorMessage="too many or too few arguement in function 'tmp'")
```

4.3 中间代码生成

这部分主要是将之前的 LL1 文法改写成翻译文法，中间算法与语法分析的 LL1 算法基本相同，都继承了 grammarParser，不同的是从左向右读 token 序列时进行若遇到语义信息进行相关操作。

4.3.1 数据结构

```
class QtGen(GrammarParser): #四元式生成
    def __init__(self,syn):
        self.t_id = 0
        self.qt_res = []
```

继承 GrammarParser 的数据结构使用其中的符号分析表，添加 qt_res 用来存储四元式，t_id 用于区分临时变量。

4.3.2 算法设计

翻译文法中的语义信息包括 SAVE,PUSH,GEQ，基于 LL1 自顶向下翻译成四元式。

1. SAVE 用于将遇到的动作符号（+，-，*，/，<,等）放入符号栈 SYMBOL_STACK

2. PUSH 用于将变量，数字等被操作的量放入 SEM_STACK
3. GEQ 表示生成四元式，将 SYMBOL_STACK 栈顶的动作弹出，同时从 SEM_STACK 中去除相应的被操作量形成四元式。
4. 主要操作如下

```
def catch(x,val):
    deal,symbol=x.split('_')
    if deal=='@SAVE':
        SYMBOL_STACK.append(symbol)
    if deal=='@PUSH':
        SEM_STACK.append(val)
    if deal=='@GEQ':
        if symbol in ["el","ie","wh","we","break","continue"]:
            qtList.append([symbol,'_','_','_'])
        else:
            s=SYMBOL_STACK.pop()
            if s=='=' or s=='callr':
                tmp2 = SEM_STACK.pop()
                tmp1 = SEM_STACK.pop()
                qtList.append([s,tmp2,'_',tmp1])
            elif s in ['if','elif','do','FUN','return','push','call']:
                tmp=SEM_STACK.pop()
                qtList.append([s,tmp,'_','_'])
            else:
                tmp2 = SEM_STACK.pop()
```

```
tmp1 = SEM_STACK.pop()
t='@t'+str(self.t_id)
self.t_id+=1
SEM_STACK.append(t)
qtList.append([s,tmp1,tmp2,t])
```

具体的部分语句四元式生成：

函数声明	FUN FUNNAME __ __
If elif else	If XX __ __ el __ __ __ elif XX __ __ el __ __ __ ie __ __ __
while	wh __ __ __ do __ __ __ we __ __ __
函数调用	callr FUNNAME __ __ XX 有返回值 call FUNNAME __ __ __ 无返回值
返回语句	Return __ __ __

4.3.3 实验结果

```
int main(){
  int a;
  int b;
  a=12;
  b=13;
  a=(12-3)*(b-a);
  if (a>0){
    a=a-1;
  }
  else{
   b=b+2;
  }
  while(a<0){
   a=(a-2)*3;
  }
}
```

```
FUN     main    _       _
=       12      _       a
=       13      _       b
-       12      3       @t0
-       b       a       @t1
*       @t0     @t1     @t2
=       @t2     _       a
>       a       0       @t3
if      @t3     _       _
-       a       1       @t4
=       @t4     _       a
el      _       _       _
+       b       2       @t5
=       @t5     _       b
ie      _       _       _
wh      _       _       _
<       a       0       @t6
do      @t6     _       _
-       a       2       @t7
*       @t7     3       @t8
=       @t8     _       a
we      _       _       _
```

5 编译后端

5.1 基本块划分与 DAG 优化

本部分主要内容将以函数为基本单位的四元式块继续划分成基本块进行优化，各个基本块之间临时变量不互相冲突可以共用空间，同时方便目标代码的生成。

5.1.1 基本块划分算法算法

以函数为基本单位根据生成的四元式中的特殊标识符进行基本块的划分，主要标识为：

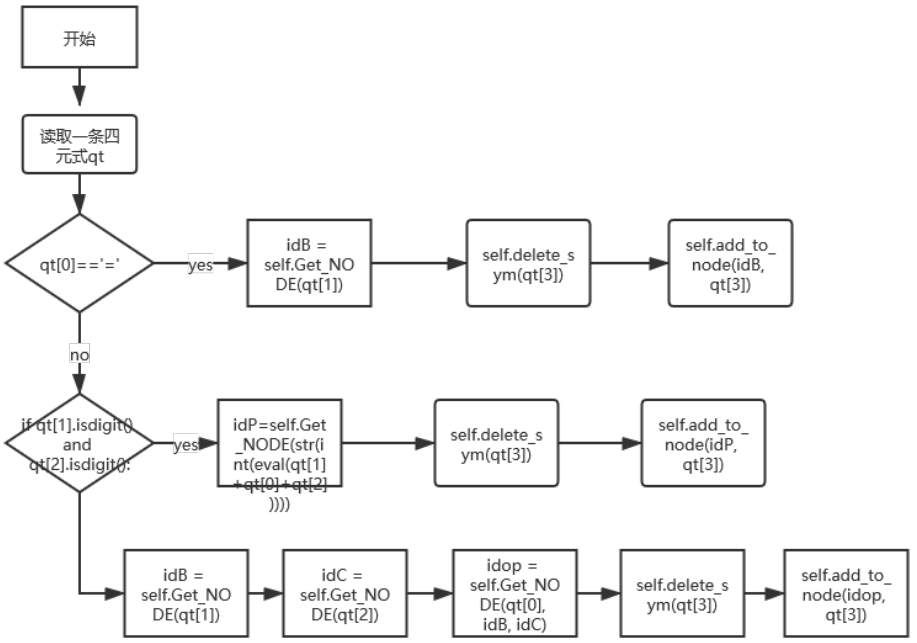
```
['do','we','if','el','elif','ie','return','continue','break','wh']
```

5.1.2 DAG 优化

数据结构：结点用一个数组顺序存储，每一个结点用一个具名元组定义

```
self.NODE = namedtuple('NODE', 'op ID signs leftNodeID rightNodeID')
```

算法流程：



关键函数：

def Get_NODE(self, sym, BofLeftNodeID=None, CofRightNodeID=None): "如果存在一个结点含有 sym 则返回结点 id；如果不存在，创一个新结点，对其编号并且返回它的 id(sym 可能是操作数或操作符)"
def delete_sym(self, A): "DOG 中有 A，且 A 不是主标记，那么删除 A"
def generate_new_qt(self, NODES): "由 DAG 生成新的四元式"
def add_to_node(self, nodeID, sym): "将 sym 附加到结点"

5.1.2 结果展示

基本块之间用空行隔开

```
FUN     main    _       _
=       12      _       a
=       13      _       b
+       a       b       @t0
=       @t0     _       a
>       a       0       @t1
if      @t1     _       _
-       a       1       @t2
=       @t2     _       a
el      _       _       _
==      a       0       @t3
elif    @t3     _       _
=       1       _       a
el      _       _       _
+       b       2       @t4
=       @t4     _       b
ie      _       _       _
wh      _       _       _
<       a       0       @t5
do      @t5     _       _
-       a       2       @t6
*       @t6     3       @t7
=       @t7     _       a
we      _       _       _
```

```
FUN     main    _       _

=       13      _       b
+       12      13      a
>       a       0       @t1
if      @t1     _       _

-       a       1       a
el      _       _       _

==      a       0       @t3
elif    @t3     _       _

=       1       _       a
el      _       _       _

+       b       2       b
ie      _       _       _


wh      _       _       _
<       a       0       @t5
do      @t5     _       _

-       a       2       @t6
*       @t6     3       a
we      _       _       _
```

具体的优化前：

```
['+', 'a', 'b', '@t0']
['=', '@t0', '_', 'c']
['+', 'a', 'b', '@t1']
['+', 'a', 'b', '@t2']
['/', '@t1', '@t2', '@t3']
['-', '3', '1', '@t4']
['*', '@t3', '@t4', '@t5']
['=', '@t5', '_', 'c']
['+', '10', 'a', '@t6']
['=', '@t6', '_', 'm.gpa']
['>', 'b', '0', '@t7']
['if', '@t7', '_', '_']
```

优化后：

```
['+', 'a', 'b', 'c']
['/', 'c', 'c', '@t3']
['*', '@t3', '2', 'c']
['+', '10', 'a', 'm.gpa']
['>', 'b', '0', '@t7']
['if', '@t7', '_', '_']
```

5.2 信息填写与目标代码生成

本部分包含，活跃信息填写，跳转信息填写，目标代码，其中跳转信息是在目标代码生过程过程中填写的。

5.2.1 数据结构：

信息填写部分

qtx = namedtuple('qtx', 'val actInfo addr1 addr2 dosePointer ')	变量名 活跃信息 一级地址 相对于函数 二级地址相对于结构体数组 是否是指针
actTable = {}	活跃信息表
t_table={}	临时变量表用于扩充符号表中不存在的临时变量

具名元组将四元式中的每一个量结合符号表扩展信息，用于目标代码生成。

目标代码生成：

```
op2asm = { #一些运算符号的对照其中比较符号填写相反的操作的汇编指令。
    '+': 'ADD',
```

```
        '-': 'SUB',
        '*': 'MUL',
        '/': 'DIV',
        '<': 'JAE',
        '>': 'JBE',
        '>=': 'JB',
        '<=': 'JA',
        '==': 'JNE',
}

def __init__(self, sym, allCode):
        self.symTable = sym    # 符号表传入
        self.allCode = allCode    # 优化后的代码传入
        self.allAsmCode=[]        # 目标代码的头部，基于符号表生成
        self.funcsAsmCode=[]      # 所有函数块的目标代码
        self.mainAsmCode=[]             # 主函数目标代码
```

最终代码由最后这三块平凑而成

5.2.2 算法设计

1. 首先顺序遍历传入的四元式基本块，生成相应的活跃信息表，并将临时变量在栈中的空间填写入临时变量表中。

2. 接着倒序遍历传入的优化后的四元式基本块查表并根据其在四元式中位置填写活跃信息并修改活跃信息表。

3. 倒序遍历的同时查找符号表，将遍历的偏移地址填入四元式的变量中，具体的根据变量类型判断变量是值还是地址，若是地址则添加两层地址，即数据结构 qtx 中的 addr1，addr2，对于传值来说 addr1 就是变量所在的值的地址，对于传地址来说 addr1 是变量中存放的地址的地址，addr2 是变量的实际位置的偏移地址，这么做的目的是使得生成汇编代码时方便区分传值和引用的情况。

在汇编代码中的体现：

函数中:

I:

ID.X=10 这种情况其中 ID 是函数的参数且是定义的结构体，这种情况往函数中传的是地址

MOV BX , SS:[BP-XX] #XX 是 ID 的参数相对函数的偏移地址在栈中

MOV AX , 10

MOV DS:[BX+XX] , AX #XX 是结构体中变量的偏移地址在内存中

若 ID 是函数中定义的结构体变量

MOV AX , 10

MOV SS:[BP+IDX+XX] , AX

II:

VAR=ID.X 这种情况 VAR 是函数中的变量 ID 是参数结构体

MOV BX , SS:[BP-XX]

MOV AX , DS:[BX+XX]

MOV SS:[BP+XX] , AX

若 ID 是函数中的变量

MOV AX , SS:[BP+IDX+XX]

MOV SS:[BP+XX] , AX

III:

FUN(ID) 这种情况调用函数 ID 是结构体

MOV AX , OFFSET ID

PUSH AX

5. 汇编代码的生成部分，将一般函数与主函数的翻译区分开来，我们将主函中的定义的变量放在 data 段所以在主函数翻译的汇编代码可以直接根据变量名获取变量的值，主函数中的临时变量是存在于堆栈段中的，函数中所有的变量除了传入的引用外其余均处于堆栈段。
6. 在获取数据的时候使用的通用寄存器是 AX 和 BX,AX 寄存器主要用于值的存放，BX 寄存器主要用于地址的存放。另外使用 BP 寄存器在函数调用时保存栈顶位置，同时用于基于偏移量求函数中的值在栈中的位置。
7. 跳转信息填写生成汇编代码的时候进行跳转信息的填写，首先将遇到的转向信息压栈接着，在遇到对应的跳转信息时将保存的信息弹出，修改，从而完成跳转信息的填写。基于此我们的课设支持 while 循环和 if 条件句的相互嵌套。
8. 在函数调用部分对 bp 指针进行了保护，和计算函数中使用的栈空间函数结束后内平栈保证了栈平衡。
9. 具体的部分汇编代码生成函数：

```
def LD(x): #用于加载数据
    res=[]
```

```
    if x.val in list(funcTable.variableDict.keys()):       #特殊情况传地址的时候使用
        if funcTable.variableDict[x.val].type in self.symTable.structNameList:
            res.append("MOV AX,OFFSET %s"%x.val)
            return res

    if x.val.isdigit():   # x 是立即数
        res.append("MOV AX,%s" % x.val)
    elif x.val.startswith('@'):
        res.append("MOV AX,SS:[BP-%d]"%x.addr1)
    elif '[' in x.val:
        t,n=x.split('[')
        n,_=n.split(']')
        n=eval(n)
        res.append("MOV BX,OFFSET %s"%t)
        res.append("MOV AX,DS:[BX+%d]"%(n*2))
    else:
        res.append("MOV AX,%s"%x.val)
    return res
```

10. 具体的一些设计样例：

循环结构 wh __ __ __	While: mov ax,b

> b _ 0 @t1
do @t1 __ __
*2 b b
We __ __ __
	cmp ax,10
	jae endwhile
	mov ax,b
	mul 2
	mov b,ax
	jmp while
	endwhile

If 判断结构
>d 0 t1
If t2 __ __
+ d 2 d
el __ __ __
== d 0 t2
Elif t2 __ __
-d 1 d
el __ __ __
+ d 1 c
Ie __ __ __
	cmp d,0
	jbe next1
	mov ax,d
	add ax,2
	mov d,ax
	jmp endif
	next1:
	cmp d 0
	jne next2
	mov ax,d
	sub ax,1
	mov d,ax
	jmp endid
	next2
	mov ax,d
	add ax,1
	mov c,ax

	endif

11. 最终代码拼接，先基于符号表，生成汇编代码的头部，如结构体的定义等部分,将主函数中的变量在数据段声明。

5.2.3 结果展示

输入的源代码：

```
struct Demo{
    int id[10];
    int gpa;
};
int tmp(int a,int b,int f,Demo m){
    int c;
    c=a+f;
    c=(a+b)/(a+b)*(3-1);
    m.gpa=10+a;
    if(b>0)
    {
    b=b+1;
    }
    elif(b==0){
    b=0;
    }
    else{
    b=b-1;
    }
    while(b>0){
    b=2*b;
    }
    return c;
}
int main(){
    int d;
    int c;
    Demo de;
    int m[3];
    m[1]=10;
    de.id=2;
    d=2;
    while(d<10){
        d=d+1;
        if(d>0){
        d=d-10;
        continue;
        }
        d=d-2;
        d=d-3;
    }
    if(d>0){
        c=(d+2)/4;
    }
    elif (d==0){
    c=d-1;
    }
    else {
    c=d+1;
    }
    CALL c=tmp(c,d,d,de);
    return 0;
}
```

目标代码：

```
demo struct
id dw 10 dup (0)
gpa dw ?
demo ends
dseg segment
d dw ?
c dw ?
de demo <?>
m dw 3 dup (0)
dseg ends
sseg segment stack
stk db  40 dup (0)
sseg ends
cseg segment
assume cs:cseg,ds:dseg,ss:sseg
main:
mov bp,sp
sub sp,20
mov ax,10
mov bx,offset m
mov ds:[bx+2],ax
mov ax,2
mov de.id,ax
mov ax,2
while5:
mov ax,d
cmp ax,10
jae endwhile8
mov ax,d
add ax,1
mov d,ax
cmp ax,0
jbe next6
mov ax,d
sub ax,10
mov d,ax
jmp while5
jmp endif7
next6:
endif7:
mov ax,d
sub ax,2
mov d,ax
mov ax,d
sub ax,3
jmp while5
endwhile8:
mov ax,d
cmp ax,0
jbe next9
mov ax,d
add ax,2
mov ss:[bp-2],ax
div 4
mov c,ax
jmp endif11
mov ax,d
cmp ax,0
jne next10
mov ax,d
sub ax,1
mov c,ax
jmp endif11
next10:
mov ax,d
add ax,1
mov c,ax
endif11:
mov ax,c
push ax
mov ax,d
push ax
mov ax,d
push ax
mov ax,offset de
push ax
call tmp
mov c,ax
mov sp,bp
mov ax,4c00h
int 21h
cseg ends
end main
tmp proc near
push bp
mov bp,sp
sub sp,20
mov ax,ss:[bp+10-0]
add ax,ss:[bp+6-0]
mov ax,ss:[bp+10-0]
add ax,ss:[bp+8-0]
mov ss:[bp-4-0],ax
div ss:[bp-4-0]
mov ss:[bp-6-0],ax
mul 2
mov ss:[bp-2-0],ax
mov ax,10
add ax,ss:[bp+10-0]
mov bx,ss:[bp+4]
mov ds:[bx+20],ax
mov ax,ss:[bp+8-0]
cmp ax,0
jbe next0
mov ax,ss:[bp+8-0]
add ax,1
mov ss:[bp+8-0],ax
jmp endif2
next0:
mov ax,ss:[bp+8-0]
cmp ax,0
jne next1
mov ax,0
mov ss:[bp+8-0],ax
mov ax,0
mov ss:[bp+8-0],ax
jmp endif2
next1:
mov ax,ss:[bp+8-0]
sub ax,1
mov ss:[bp+8-0],ax
endif2:
while3:
mov ax,ss:[bp+8-0]
cmp ax,0
jbe endwhile4
mov ax,2
mul ss:[bp+8-0]
jmp while3
endwhile4:
mov ax,ss:[bp-2-0]
mov sp,bp
pop bp
ret 8
tmp endp
```

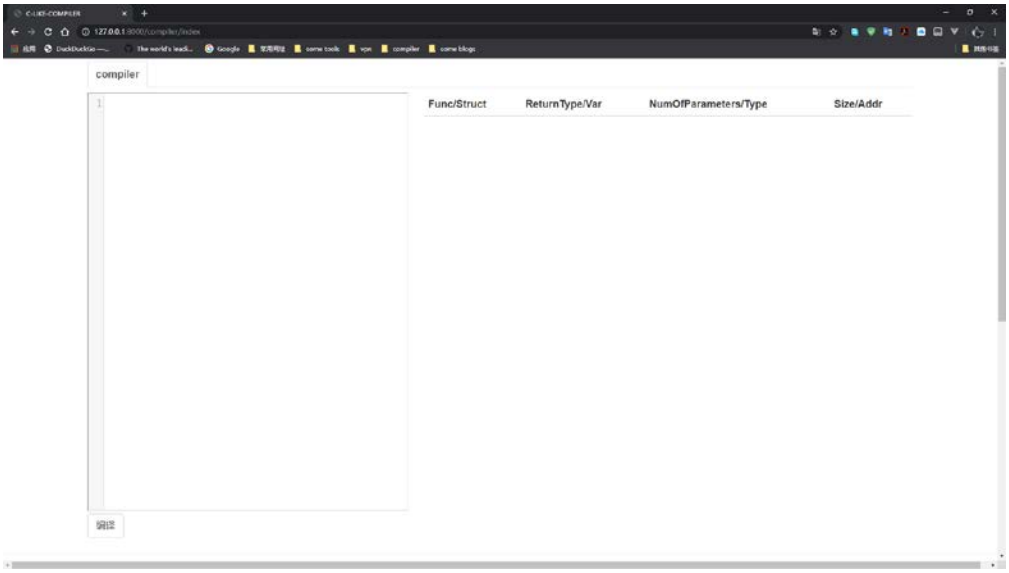
6 可视化界面

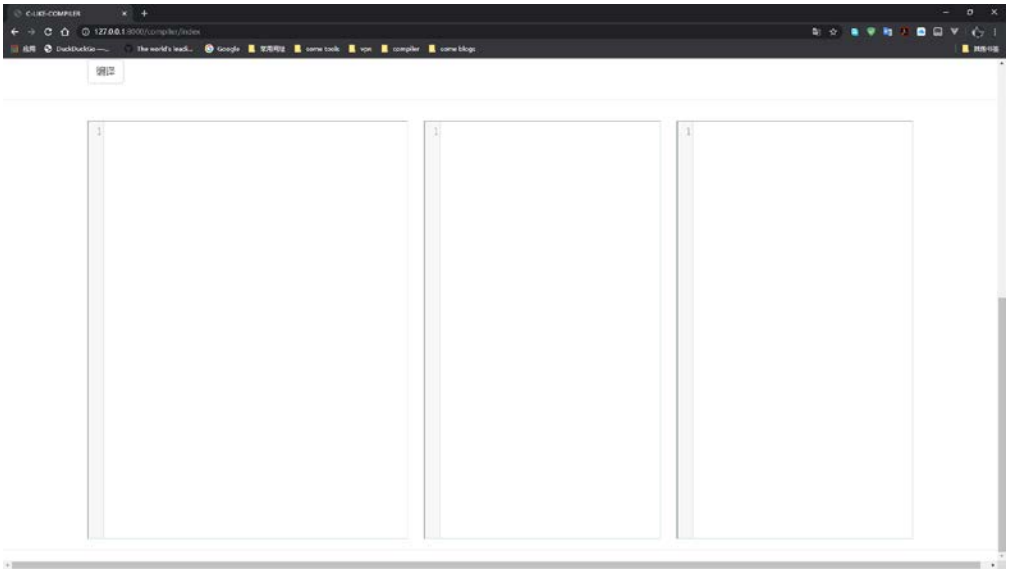
6.1 界面设计

前端基于 html jquary bootstrap 具有代码高亮的功能，后端使用基于 python 的 django 框架，由于本课设是基于 python 编写的后端可以直接调用。目前项目已经放到了个人的网站可以登陆访问使用。网址：

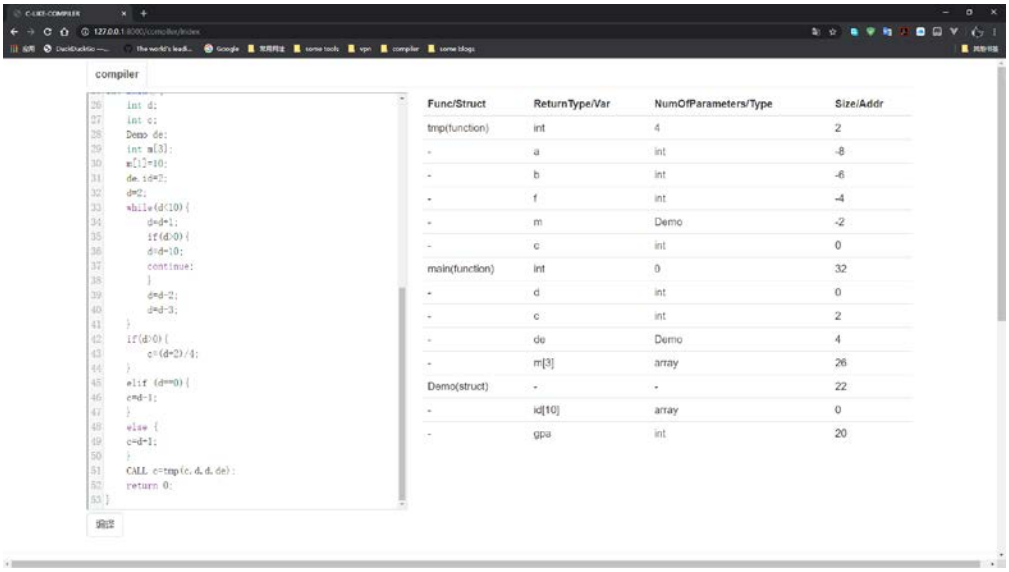
http://justyan.top/compiler/index

展示分为上下两个部分，上半部分是代码输入，和符号表，下半部分依次是目标代码，未优化的四元式，优化后的四元式。打开的页面如下。

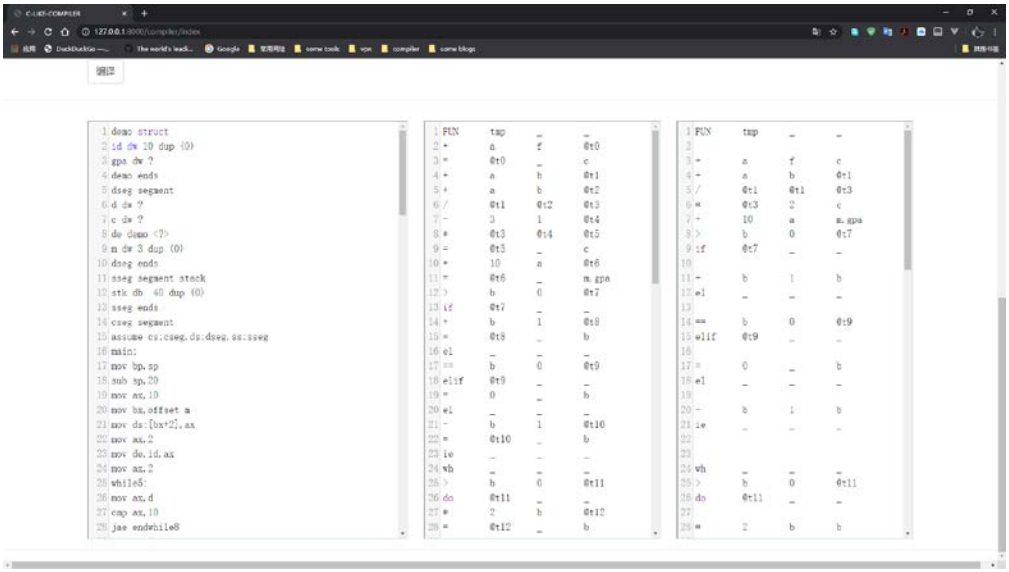




输入代码后点击编译：



Func/Struct	ReturnType/Var	NumOfParameters/Type	Size/Addr
tmp(function)	int	4	2
-	a	int	-8
-	b	int	-6
-	f	int	-4
-	m	Demo	-2
-	c	int	0
main(function)	int	0	32
-	d	int	0
-	c	int	2
-	de	Demo	4
-	m[3]	array	26
Demo(struct)	-	-	22
-	id[10]	array	0
-	gpa	int	20



7 结论

这次我们设计实现了一种基于 LL1 自顶向下将一种自己设计的类 C 语言翻译成汇编语言的简单编译器，并且用网站的形式制作了交互简单的使用界面。设计符合 LL1 的文法，支持识别结构体定义，整型数，数组，字符串，函数定义调用，if while 互相嵌套，整个文法结构清晰，并且可以随意修改支持自动计算分析表。符号表的设计部分保证了变量定义只存在于其作用域内，区分了局部变量。词法分析部分将输入的字符串转换成 token 序列方便后续操作，并且记录了位置信息用于检测代码中的错误位置。语法分析过程进行了整个流程中重要的几个步骤，填写符号表，发现错误信息及时返回，四元式生成部分基于原本的 LL1 文法添加了语义动作构造了翻译文法，与原文法的分析表相结合进行四元

式的生成，接着将生成的四元式划分基本块，并进行优化，最终对基本块中四元式进行信息填写，生成目标汇编代码。

8 参考文献

1、陈火旺.《*程序设计语言编译原理*》(*第3版*). 北京：国防工业出版社.2000.

2、美 Alfred V.Aho Ravi Sethi Jeffrey D. Ullman 著.李建中，姜守旭译.《*编译原理*》.北京：机械工业出版社.2003.

3、美 Kenneth C.Louden 著.冯博琴等译.《*编译原理及实践*》.北京：机械工业出版社.2002.

4、金成植著.《*编译程序构造原理和实现技术*》. 北京：高等教育出版社. 2002.