《编译原理》课程设计

实验报告书

小组成员：

161730319吴扬俊（组长）

161730325荆顺吉

161730321武起龙

161730328张峥

161730301常为

指导老师： 李静 谢婉玲

课设地点： 实验楼104

课设时间： 2019/12/28-2019/1/3

2020年 01 月 06日

目录

[小组分工 3](#_Toc29222679)

[第一章 C-语言的语法图描述 3](#_Toc29222680)

[1.1 词法描述 4](#_Toc29222681)

[1.2 语法描述 4](#_Toc29222682)

[第二章 系统设计 5](#_Toc29222683)

[2.1 系统总体结构 5](#_Toc29222684)

[2.2 主要功能模块的设计 6](#_Toc29222685)

[2.2.1 词法分析器 6](#_Toc29222686)

[2.2.2 语法分析器 10](#_Toc29222687)

[2.2.3 符号表设计 17](#_Toc29222688)

[2.2.4 语义分析及四元式产生 18](#_Toc29222689)

[2.2.5 汇编生成器（新增功能） 20](#_Toc29222690)

[2.3 系统运行流程 22](#_Toc29222691)

[2.3.1 环境配置 22](#_Toc29222692)

[第三章 系统实现 22](#_Toc29222693)

[3.1 系统主要函数说明（主要功能、输入\输出、实现思想） 22](#_Toc29222694)

[3.1.1 词法分析 22](#_Toc29222695)

[3.1.2 语法分析 24](#_Toc29222696)

[3.1.3 语义分析 26](#_Toc29222697)

[3.1.4 汇编生成器 28](#_Toc29222698)

[3.1.5 通用接口 29](#_Toc29222699)

[3.2 系统代码（必须有注释） 30](#_Toc29222700)

[3.2.1 词法分析器 30](#_Toc29222701)

[3.2.2 语法分析和语义分析 32](#_Toc29222702)

[3.2.3 符号表 44](#_Toc29222703)

[3.2.4 汇编生成器 46](#_Toc29222704)

[3.2.5 UI界面 49](#_Toc29222705)

[第四章 系统测试 55](#_Toc29222706)

[4.1 正确性测试 56](#_Toc29222707)

[4.2 类型系统与作用域测试 58](#_Toc29222708)

[4.3 错误处理测试 59](#_Toc29222709)

[第五章 课程设计心得 60](#_Toc29222710)

[武起龙 60](#_Toc29222711)

[常为 60](#_Toc29222712)

[张峥 61](#_Toc29222713)

[吴扬俊 62](#_Toc29222714)

[第六章 建议和意见 63](#_Toc29222715)

[第七章 参考资料 63](#_Toc29222716)

# 小组分工

（注：新增汇编生成功能）

|  |  |  |
| --- | --- | --- |
| **成员名** | **负责工作** | |
| 常为 | 词法分析设计，编码，测试，文档撰写 | LEX源码分析 |
| 荆顺吉 | 界面设计，测试用例设计 |
| 武起龙 | 语法分析设计，编码，测试，文档撰写；YACC源码分析；改进算法效率 | UI设计实现；连接词法和语法 |
| 张峥 | 算法流程图绘制 |
| 吴扬俊 | 四元式，类型系统，汇编代码生成的设计，编码，测试，文档撰写；整合文档 | |

# C-语言的语法图描述

**C**-**Minus** (或简称为**C-**)，是一种适合编译器设计方案的语言，包括函数和数组。本质上它是C的一个子集，但省去了一些重要的部分。本章内容包括，首先，列出了语言惯用的词法，包括语言标记的描述。其次，给出了每个语言构造的BNF描述，同时还有相关语义的描述。

## 1.1 词法描述

C-语言共识别五类单词：关键字、运算符、界符、标识符和常量，每一类所识别的具体单词以枚举或正则表达式方式给出，具体见表1.1。

表 1.1 词法描述

|  |  |
| --- | --- |
| 单词种别 | 具体描述 |
| 保留字/关键字 | if else while int void return |
| 运算符 | + - \* / = |
| < > == != <= >= |
| 界符 | ( ) { } [ ] /\* \*/ |
| 标识符 | '[a-zA-Z\_][a-zA-Z\_0-9]\*'（正则表达式） |
| 常量（仅支持正整数） | '[0-9]+' （正则表达式） |

## 1.2 语法描述

语法描述满足基本的C语言规则，但是有几点值得注意：

* 变量定义时不可赋值。如int a = 1;就是非法语句。
* 变量每次只可定义一个。如int a,b；也是非法语句。
* 变量定义必须集中在花括号（作用域开始）的最前方。

其余具体描述见表1.2。

表 1.2 语法描述

|  |  |
| --- | --- |
| 1 | program -> declaration-list |
| 2 | declaration-list -> declaration-list declaration | declaration |
| 3 | declaration -> var-declaration | fun-declaration |
| 4 | var-declaration -> type-specifier ID; | type-specifier ID [ NUM ]; |
| 5 | type-specifier -> int | void |
| 6 | fun-declaration -> type-specifier ID ( params ) compound-stmt // ！！注意这里|去除了 |
| 7 | params -> params-list | void |
| 8 | params-list -> params-list, param | param |
| 9 | param -> type-specifier ID | type-specifier ID [ ] |
| 10 | compound-stmt -> { local-declarations statement-list } |
| 11 | local-declarations -> local-declarations var-declaration | empty |
| 12 | statement-list -> statement-list statement | empty |
| 13 | statement -> expression-stmt | compound-stmt | selection-stmt | iteration-stmt | return-stmt |
| 14 | expression-stmt -> expression ; | ; |
| 15 | selection-stmt -> if ( expression ) statement | if ( expression ) statement else statement |
| 16 | iteration-stmt -> while ( expression ) statement |
| 17 | return-stmt -> return ; | return expression; |
| 18 | expression -> var = expression | simple-expression |
| 19 | var -> ID | ID [ expression ] |
| 20 | simple-expression -> additive-expression relop additive-expression | additive-expression |
| 21 | relop -> <= | < | > | >= | == | != |
| 22 | additive-expression -> additive-expression addop term | term |
| 23 | addop -> + | - |
| 24 | term -> term mulop factor | factor |
| 25 | mulop -> \* | / |
| 26 | factor -> ( expression ) | var | call | NUM |
| 27 | call -> ID ( args ) |
| 28 | args -> arg-list | empty |
| 29 | arg-list -> arg-list, expression | expression |

# 第二章 系统设计

## 2.1 系统总体结构

本系统整体分为三个模块：编译器，文件处理，UI界面。其中编译器细分为词法分析器、语法分析器、语义分析与四元式产生、汇编生成器、符号表和错误处理六个子模块。每个模块之间的具体交互过程如图2.1所示。

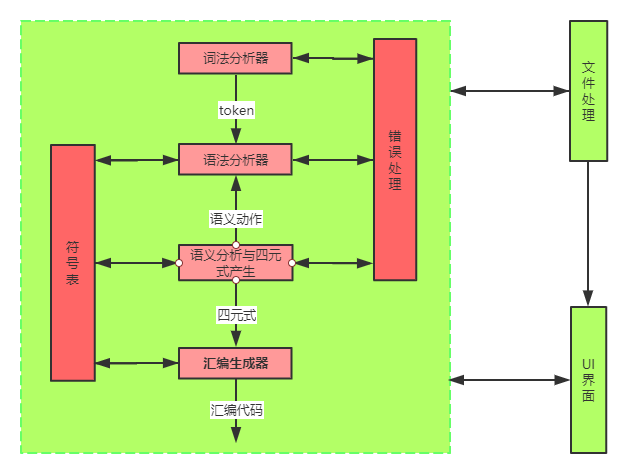


图 2.1 系统总体结构

## 2.2 主要功能模块的设计

### 2.2.1 词法分析器

本部分先给出PLY-LEX相对于手动实现、Unix下的LEX的优势，说明为何选用此工具，其次给出词法分析流程图，紧接着给出错误处理方案，最后阐明遇到的问题及解决方案。

#### 2.2.1.1 PLY-LEX与手动实现、Unix-LEX比较

限于篇幅，这里只详细给出PLY相对于手动实现、Unix-LEX的优点与缺点。具体见表2.1，也正是PLY的众多优点，让我们选择了它。

表 2.1 PLY相对于手动实现、LEX的优缺点

|  |  |  |
| --- | --- | --- |
| 比较项目 | PLY优点 | PLY缺点 |
| 手动实现 | 比起map拆分实现，采用lex更简洁明了，无需复杂的数据结构和type定义，只需字符串输入 | Lex所有规则都是正则表达式匹配而成，所以对正则表达式的书写要求非常严格，正则表达式书写不规范将直接影响字符匹配和词法分析 |
| 在对数据的处理上，手动需循环读取单个字符进行依次比较，在遇到难点例如”==”、”!=”等词时处理非常麻烦，而lex只需按照顺序书写规则，在处理时按照书写顺序或者正则表达式倒叙处理，非常简单地解决了复杂词的处理问题； |
| Lex内部生成LexToken()实例，无需用户定义，且内部提供了可供使用的接口token() |
| Unix-LEX | 传统lex需要用户将”.l”文件转化成”.c”文件才可以使用，但Ply.LEX无需转化，“.py”文件即可作为最终的词法分析器 |

#### 2.2.1.2 词法分析流程

词法分析的流程图见图2.2，以下对流程图关键步骤进行具体说明：

1. Ply.Lex通过Lexerreflect反射机制，将用户所有定义转化成用户数据字典，最终通过正则表达式匹配进行词法分析；
2. 首先，在tokens中写入待标记的列表(里面包含字符属性名称)，此列表将所有可能的标记告诉lex，以进行接下来的属性验证；
3. 在设计好标记列表后要给定所有标记的规则，每种标记用正则表达式给出，每个规则需要以”t\_”开头声明，其中”t\_”后面的定义与列表内容严格对应：对于简单的标记只要用”=”声明，复杂的标记需要定义一个方法以设计规则，采用方法的好处是既可以定义正则表达式，又可以处理语义动作；
4. 定义方法时，需要接受一个LexToken()实例的参数，该实例有四个属性值：t.type表示标记的类型名称，t.value表示标记值(匹配成功的字符串)，t.lineno表示当前输入串的行数，t.lexpos表示输入串的偏移；
5. 将输入传给input()方法进行匹配，lex用re模块处理匹配（匹配原理是有穷自动机DFA）：按照定义好的规则进行匹配，其中由方法定义的标记规则按照出现顺序依次加入；由字符串变量定义的规则按照正则表达式长度倒序依次加入；
6. lex通过token()方法作为接口获取标记序列；

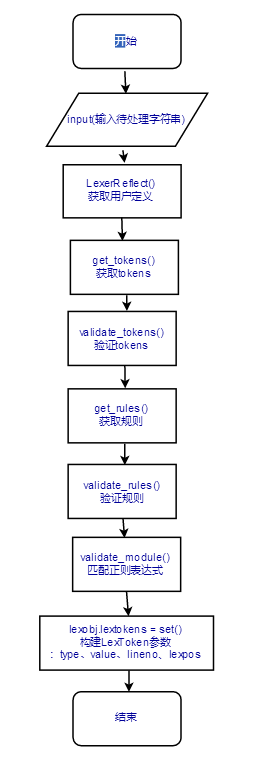


图2.2 词法分析流程图

#### 2.2.1.3 错误处理

LexToken()的参数 lineno（当前输入串的行数）和lexpos（当前输入串的偏移）表示当前token所在行数和偏移，因此可利用二者进行错误的行列号计算。

（1）行号计算

行号lineno每行+1即可；

遇到空行/连续空行使用“\n”正则表达式匹配；

遇到注释则通过检查注释中的“\n”个数来得知行数。

（2）列号计算

lexpos则是向左找到最近的“\n”，将当前偏移量减去“\n”处的偏移量即为列号。

#### 2.2.1.4 遇到的问题及解决方案

（1）消除注释

开始只能进行单行注释消除，改进后可以消除多行注释。

但发现会消除两个注释中间代码，原因是使用了正则表达式中的贪心匹配，改进后用“\*？”最短匹配解决了问题。

接下来发现表达式不能匹配中文的问题，原因是使用了“[a-zA-Z0-9 \_]”。

最终改进后的正则表达式为“(/\\*(.|\n)\*?\\*/)|(\/\/.\*)”。

### 2.2.2 语法分析器

本部分先对比LL（1）与LALR文法，接着给出基于PLY-YACC的语法分析器构建与分析思路，最后阐述错误处理方案。

#### 2.2.2.1 LL(1)与LALR对比

对于LL(1)与LALR文法，我们最关心它们的适用性程度，前者对文法要求极高，而后者适应性强，因此我们选择后者作为文法分析算法。其余详细差别见表2.2。

表2.2 LL（1）与LALR对比

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 文法名称 | 文法类型 | 分析过程 | 优缺点 | 适用情况 |
| LL(1) | 自上而下 | 根据产生式构造First、Follow集→写出预测分析表→进行预测分析 | 符合人的思维逻辑，但并不利于接下来的语义分析 | 仅进行语法分析时 |
| LALR | 自下而上 | 拓广文法→构造LR(1)自动机→构造文法分析表→进行文法分析 | 比较抽象，但方便语义分析 | 插入语义规则时和更复杂的语法 |

#### 2.2.2.2 分析表构建

（1）PLY-YACC参数列表分析，参见表2.3。

表 2.3 PLY-YACC参数列表

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数作用 | 默认值 |
| method | 设置分析方法 | LALR |
| debug | 默认为调试模式，调试模式会生成parser.out文件 | yaccdebug |
| module | 获取模块字典 | None |
| tabmodule | 设置parsetab.py文件名 | tab\_module |
| start | 设置起始产生式 | None |
| check\_recursion | 检查是否有不可达或无限递归 | True |
| optimize | 为true则使用已有的parsetab.py | False |
| write\_tables | 为false不生成分析表 | True |
| debugfile | 调试日志文件名 | debug\_file |
| outputdir | parsetab.py文件生成目录 | None |
| debuglog | 调试日志 | None |
| errorlog | 错误日志 | None |
| picklefile | parsetab.py文件路径 | None |

（2）分析表构建流程如图2.3所示，以下为关键步骤的详细解释。

1. 调用yacc()函数
2. 进行传递参数的处理
3. 若模块字典出错，停止构建parser
4. 判断文法分析表是否已构建，若已构建则读表，跳转至10
5. 根据获得的tokens和构造语法对象(Grammar类)，并设置优先级
6. 将产生式，起始符号添加到语法对象中
7. 验证语法对象正确性，不正确则停止构建parser
8. 查找未使用的tokens、未使用的产生式和不可达产生式
9. 构造文法分析表，并构造parsetab.py和parser.out文件
10. 构造parser(LRParser类)

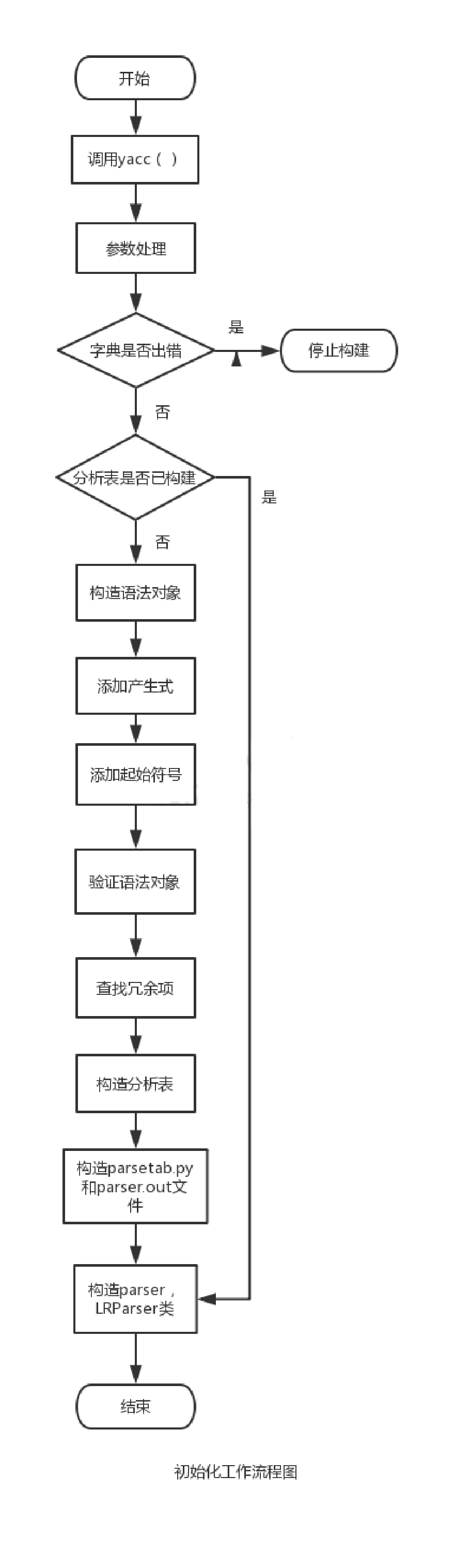


图2.3 分析表构建流程图

#### 2.2.2.3 主控程序

（1）分析接口parse()参数列表，详细见表2.4。

表 2.4 分析接口parse参数列表

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数作用 | 默认值 |
| input | 不为空则调用lexer.input(input) | None |
| lexer | 用于设置词法分析器 | None |
| debug | 为true会输出调试信息 | False |
| tracking | 为true会记录错误符号的终止行列号 | False |
| tokenfunc | 设置语法分析所用的token | None |

（2）主控分析流程如图2.4所示，以下为关键步骤具体描述。

1. 根据传入参数初始化
2. 进入语法分析，起始状态为0，构建一个状态栈
3. 判断状态是否为**默认状态**：（**默认状态**为有且仅有一个归约操作的状态）
   1. 如果不是默认状态，则检查是否设置了前向标志：
      1. 如果未设置，则读取下一个token，设置t(临时变量)为该token的action标号
      2. 如果设置，则使用其前向标志，设置t(临时变量)为该标志的action标号
   2. 如果是默认状态，则设置t(临时变量)为该状态的action标号
4. 判断t是否存在：
   1. 如果不存在，则进入错误处理：
      1. 如果状态栈中只有一个状态且当前token不为文件尾，则抛弃该token，设置状态为0.跳至步骤3
      2. 否则进入正常错误处理，如果有p\_error()则调用p\_error()输出错误信息，无便使用系统自带错误处理。
   2. 如果存在，则判断t的正负：
      1. 大于0，t进状态栈，进入移进操作，设置状态为移进后的状态
      2. 小于0，被归约状态出状态栈，归约后的状态进状态栈。进入归约操作，设置状态为规约后的状态
      3. 等于0，代表完成分析
5. 重复3,4直到完成分析

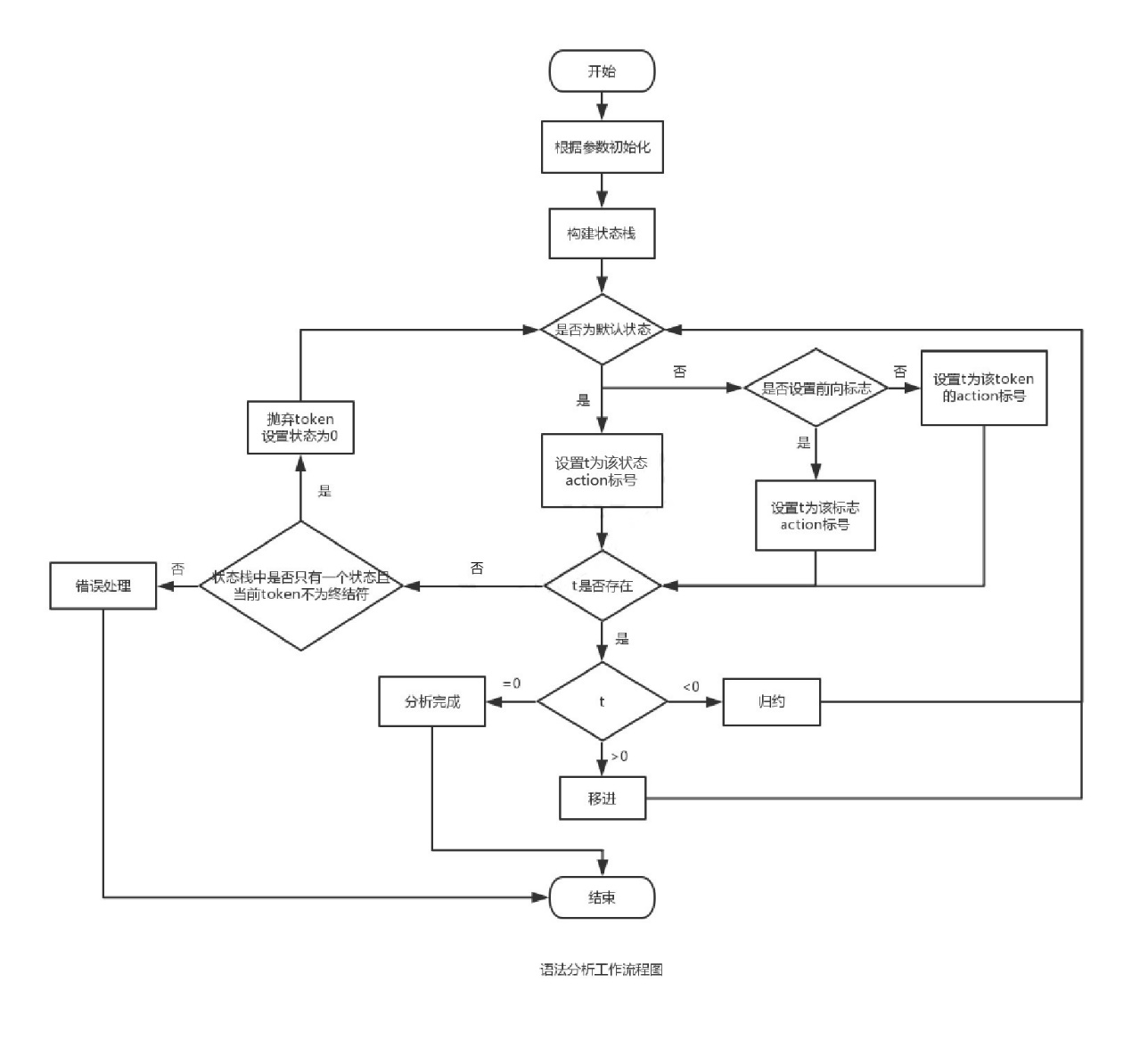


图 2.4 主控分析流程图

#### 2.2.2.4 错误处理

**（1）yacc自带错误处理**

处理方式：在用户未定义p\_error()错误处理函数时，当语法分析遇到错误时，会进入**错误恢复**模式，在此模式下，会输出错误信息，出错的token的类型会定义为error，随后会放弃整个分析栈，从错误token的下一个token重新开始进行进行分析，并不再输出错误信息，直到成功移进三个token，才会再次输出错误信息。

优缺点：该方法为系统自带错误处理方法，无需用户定义，但该方式处理方式会导致若语句块内任一语句，则整个语句块可能会被抛弃，也有可能会将原有语句块中的句子分析出错误的信息，故需要我们对其错误处理方式进行改进。

**（2）根据error规则和再同步**

处理方式：在语法规则中包含error标记，在语义动作中添加错误处理，如打印错误

* **def** p\_statement\_print\_error(p):  
   *'statement : PRINT error SEMI'*  
   print "Syntax error in print statement. Bad expression"

优缺点：在语义分析中就已经把错误抛出，便于定位错误所在，但在已有产生式基础上修改添加新的产生式比较困难，要消耗大量时间

**（3）调用p\_error()函数**

处理方式：

1）抛弃错误标记：当检测到错误标记时，跳过当前标记，并设置为无错误，继续分析下面的文法，适用于文法的查错

* + parser = yacc.yacc()   
    **def** p\_error(p):   
     **global** error\_num   
     error\_num+=1   
     **if** p:   
     parser.errok()   
     print("Syntax error at '%s'" % p.value)   
     **else**:   
     print("Syntax error at EOF")

2）抛弃错误所在语句块：当检测到错误标记时，跳过当前语句块，抛弃当前整个分析栈，从下一个语句块的开头，重置分析器继续分析，适用于计算文法的属性

* + parser = yacc.yacc()   
    **def** p\_error(p):   
     **global** error\_num   
     error\_num+=1   
     tok   
     **if** p:   
     **while** 1:   
     tok = parser.token()   
     **if**(**not** tok **or** tok.type == 'INT' **or** tok.type == 'VOID' ):   
     break   
     print("Syntax error at '%s'" % p.value)   
     parser.restart()   
     **else**:   
     print("Syntax error at EOF")   
     **return** tok

优缺点：方便错误处理，但无法进行差异化的错误处理

**（4）最终采用的错误处理方法**

选用调用p\_error()的抛弃错误标记的方法，原因是较编写带错误处理的语义规则省时省力，而就错误语句块而言，能指出所有错误所在。

### 2.2.3 符号表设计

本部分先说明符号表表项属性设计，接着阐述其整体组织结构。

#### 2.2.3.1 符号表表项属性设计

符号表表项至少要包含符号名、类型及其作用域，其余重要属性见表2.5。

表 2.5 符号表表项主要属性

|  |  |
| --- | --- |
| 字段 | 说明 |
| Name | 符号名 |
| Attribution | 符号种属 |
| Type | 符号类型 |
| Scope | 作用域 |
| Offset | 位置信息 |
| Allocation | 存储分配信息 |
| Detail | 数组内情向量/函数参数列表 |

#### 2.2.3.2 符号表组织结构

符号表主要考虑插入和查询两种操作，而符号表有三种组织形式：线性表，搜索树，散列表。其中线性表易插入，难搜索；而搜索树与其相反互补；散列表兼顾插入和搜索，因此选用散列表方式组织符号表。

其中散列哈希函数使用符号的字典序计算，最后模一个很大的质数；使用链地址法解决冲突。

为了解决作用域问题，使用栈存储函数定义过程，结束后立即退栈。

### 2.2.4 语义分析及四元式产生

本部分主要阐明一遍扫描的翻译模式设计，再详细阐述类型检查系统设计。

#### 2.2.4.1 一遍扫描

不同语句翻译模式不同，所有的语句中条件和循环的翻译技巧性较强，需要用到回填技术完成一遍扫描，鉴于篇幅有限，此处仅给出if-then-else，if-then，while三者的翻译模式设计。

# if-then

selection\_stmt -> IF ( expression ) M statement N {

    backPatch(expression.truelist, M);

    selection\_stmt.nextlist = merge(expression.falseList,statement.nextlist);

    backPatch(selection\_stmt.nextlist, N)

}

# if-then-else

selection\_stmt -> IF ( expression ) M1 statement N ELSE M2 statement M3 {

    backPatch(expression.truelist, M1);

    backPatch(expression.falseList, M2);

    selection\_stmt.nextlist = merge(statement.nextlist, N, statement.nextlist);

    backPatch(selection\_stmt.nextlist, M3) # 结束后立刻回填

}

# while

iteration\_stmt -> WHILE M1 ( expression ) M2 statement M3

{

    backPatch(statement.nextlist, M1);

    backPatch(expression.truelist, M2);

    iteration\_stmt.nextlist = expression.falseList;

    backPatch(iteration\_stmt.nextlist, M3+1); # 增加一条while最后无条件跳转

}

# 插空，记录转移标号

M -> empty {

    M = nextQuad(); # 产生下一条四元式

}

# 插空，无条件跳转记录

N -> empty {

    N = nextQuad();

    emit(newLabel(), 'j') # 四元式跳转

}

#### 2.2.4.2 类型检查

类型检查系统主要包含作用域处理和引用错误，其中作用域处理尤为关键。

1. 作用域处理

结合符号表，设计全局静态区和局部区，每遇见一个过程开始就压入栈中，结束过程则相应退栈，在过程内部遇见局部作用域符号{、}分别执行进栈和出栈操作，以此来完成嵌套的作用域识别。

该部分可以识别符号是否定义，是否存在重定义等一系列问题。

1. 引用错误

在作用域实现的基础上，查询符号表顺序如下：

1. 从数组模拟的栈顶往回查询，一旦找到，立刻返回。否则进入全局区查找，若找到，进入b），否则进入c）。
2. 找到后判断其类型，若是数组，需检查其下标是否越界；若是函数，需检查参数列表个数、类型和返回值。
3. 提示相应错误。

### 2.2.5 汇编生成器（新增功能）

#### 2.2.5.1 指令选择

（1）指令集架构：IA-32

（2）汇编格式：NASM

#### 2.2.5.2 计算待用信息

以下为待用信息计算算法描述。

|  |
| --- |
| 算法名：计算待用信息  输入：基本块的四元式序列  输出：基本块内所有变量的待用信息和活跃信息  //初始化  1. 把基本块中每个变量在符号表登记项中的待用信息填为“非待用”。  2. 根据每个变量在出口后是否活跃，在活跃信息栏填上“活跃”或“非活跃”。  从基本块出口语句到入口语句由后向前依次向前处理每个中间代码。对每个形如i:A:=B op C 的代码，执行以下步骤：  2.1 把符号表中变量A的待用信息和活跃信息附加到中间代码i上  2.2 把符号表中变量A的待用信息栏和活跃信息栏分别设置为“非待用”和“非活跃”  2.3 把符号表中变量B和C的待用和活跃信息附加到中间代码i上  2.4 把符号表中变量B和C的待用信息设为i，活跃信息设置为“活跃” |

#### 2.2.5.3 寄存器分配

寄存器分配原则是加快代码运行速度，具体有以下三条：

1. 当生成某变量的目标代码时，尽可能让变量的值或计算结构驻留在寄存器中，除非该寄存器必须用来存放其它变量值。

2. 当到达基本块出口时，将变量值保存于内存中，便于后续使用。

3. 一个基本块后不再使用的变量所占用的寄存器需尽早释放，提高寄存器利用率。

#### 2.2.5.4 生成算法

以下为简单目标代码生成算法描述。

|  |
| --- |
| 算法：简单代码生成算法  输入：基本块BB[n]，每条中间代码形式为i:A:=B op C  输出：目标代码  For(j=1;j<=n;j++) {  R = getReg(BB[j]); // 调用寄存器选择函数  Breg = AVALUE[B];Creg=AVALUE[C]; // 获得B，C存放位置  If (Breg==R) {  生成目标代码op R, Creg  }  Else {  MOV Breg, R  Op R, Creg  }  // 修改寄存器描述数组和地址描述数组，释放B和C所占用寄存器，使A独占寄存器R  If(Breg == R) AVALUE[B]=AVALUE[B]-{R};  If(Creg == R) AVALUE[C]=AVALUE[C]-{R};  AVALUE[A]={R};RVALUE[R]={A};  // 若Breg或Creg不再被引用，就释放  If(Breg不再被引用) {  RVALUE[Ri]=RVALUE[Ri]-{B};AVALUE[B]=AVALUE[B]-{Ri}  }  If(Creg不再被引用) {  RVALUE[Ri]=RVALUE[Ri]-{C};AVALUE[C]=AVALUE[C]-{Ri}  }  } |

## 2.3 系统运行流程

### 2.3.1 环境配置

操作系统：window10

硬件配置：intel core i5

软件支持：Anaconda（python3.6，PyQt5）

运行步骤：

1. 源码用Spyder3打开，运行ui.py文件，出现界面。
2. 按照提示拖入测试文件即可。

# 第三章 系统实现

## 3.1 系统主要函数说明（主要功能、输入\输出、实现思想）

### 3.1.1 词法分析

**主要函数**

t\_annotation(t)：忽略注释

t\_LF(t)：忽略换行

t\_tab(t)：忽略tab

t\_NUMINT(t)：识别Int型

t\_NUMFLOAT(t)：识别float型

t\_ID(t)：识别标识符

t\_newline(t)：增加行数

t\_error(t)：错误处理：输出错误符号，行数，列数后跳过当前错误继续扫描

getToken()：获取Token的接口

**lex()输入与输出**

输入：字符串

输出：LexToken(str type, repr value, int lineno, int lexpos)：LexToken类型的实例，包括t.type(标记的类型名称)、t.value(标记值)、t.lineno(标记行号)和t.lexpos(标记列号)共四个属性值

**实现思想**

此词法分析器借助于ply.lex工具，ply.lex利用python的reflect反射机制，将用户所有定义转化成用户数据字典，最终通过正则表达式匹配进行词法分析，具体实现如下：

首先，定义tokens，在tokens中写入待标记的列表(里面包含字符属性名称)，此列表将所有可能的标记告诉lex，以进行接下来的属性验证。

在设计好标记列表后要给定所有标记的规则，每种标记用正则表达式给出，每个规则需要以”t\_”开头声明，其中”t\_”后面的定义与列表内容严格对应：对于简单的标记只要用”=”声明，复杂的标记需要定义一个方法以设计规则，采用方法的好处是既可以定义正则表达式，又可以处理语义动作。

定义方法时，需要接受一个LexToken()实例的参数，该实例有四个属性值：t.type表示标记的类型名称，t.value表示标记值(匹配成功的字符串)，t.lineno表示当前输入串的行数，t.lexpos表示输入串的偏移。

运行时，用户将输入传给input()方法进行匹配，lex用re模块处理正则表达式匹配（匹配原理是有穷自动机DFA）：lex按照定义好的规则（直接定义和函数定义）进行匹配，其中由方法定义的标记规则按照出现顺序依次加入；由字符串变量定义的规则按照正则表达式长度倒序依次加入，匹配成功后将匹配结果传给LexToken()的四个属性值，匹配出错则输出出错字符串以及对应行号、列号。

最后，用户通过token()方法作为接口获取标记序列，即LexToken，也是最终的输出结果。

### 3.1.2 语法分析

yacc\_ui.py的调用方法是先调用yacc()函数生成预测分析表，并返回一LRParser类，然后再调用该类的parse()函数并传递要分析的字符串即可。

#### Yacc()

该函数用于构造预测分析表。因为构造预测分析表要消耗大量的系统资源，所以函数会检测当前文件夹中是否已有相应文件，文件中的预测分析表的签名\_lr\_signature（即所有产生式）是否和需要构建预测分析表的产生式一致，不一致才会进行构建。构建完成后会生成两个文件parsetab.py和parser.out。

yacc()参数列表：

表3.1 yacc参数列表

| **参数名称** | **参数作用** | **默认值** |
| --- | --- | --- |
| method | 设置分析方法 | LALR |
| debug | 默认为调试模式，调试模式会生成parser.out文件 | yaccdebug |
| module | 获取模块字典 | None |
| tabmodule | 设置parsetab.py文件名 | tab\_module |
| start | 设置起始产生式 | None |
| check\_recursion | 检查是否有不可达或无限递归 | True |
| optimize | 为true则使用已有的parsetab.py | False |
| write\_tables | 为false不生成分析表 | True |
| debugfile | 调试日志文件名 | debug\_file |
| outputdir | parsetab.py文件生成目录 | None |
| debuglog | 调试日志 | None |
| errorlog | 错误日志 | None |
| picklefile | parsetab.py文件路径 | None |

parsetab.py包括：

表3.2 parsetab属性

|  |  |
| --- | --- |
| 名称 | 作用 |
| \_lr\_method | 记录是SLR还是LALR的预测分析表 |
| \_lr\_signature | 记录产生式 |
| \_lr\_action\_items | 记录各产生式的action |
| \_lr\_goto\_items | 记录各产生式的goto |
| \_lr\_productions | 记录内容依次为各产生式的内容，产生式左侧非终结符，子产生式的个数，所在py文件中的函数名，所在的py文件名，所在py文件中的行数 |

Parser.out即预测分析表。

#### LRParser.parse()

该函数用于根据传入参数debug和tracking的不同调用三种不同的分析函数parsedebug(会在控制台输出调试信息，debug=true时调用)、parseopt（会记录分析的tokens的尾坐标，debug=false,tracking=true时调用）、parseopt\_notrack(无特殊功能),这三个函数基本分析流程完全一致。将传入字符串进行语法分析和执行相应的语义动作。语法分析即根据预测分析表进行移进规约，出错则进入错误处理。该函数会根据预测分析表设置defaulted\_states列表，存储只用有且只有一个规约动作的状态，用于优化算法。

LRParser.parse()参数列表：

表 3.3 LRParser.parse参数列表

| **参数名称** | **参数作用** | **默认值** |
| --- | --- | --- |
| input | 不为空则调用lexer.input(input) | None |
| lexer | 用于设置词法分析器 | None |
| debug | 为true会输出调试信息 | False |
| tracking | 为true会记录错误符号的终止行列号 | False |
| tokenfunc | 设置语法分析所用的token | None |

#### sematic\_uiapi.py

该文件是为ui.py所特意编写的分析器。其接口为Analysis()函数。该函数用于接受ui.py传来的文件路径，将文件中的字符串进行分析，将分析结果返回。0代表分析成功，1代表分析出错，3代表文件打开失败。

该文件最终的分析过程会以相应的函数存储在相应的文件中。

### 3.1.3 语义分析

#### getID(ID)

根据ID查询符号表，存在则返回相应表项，否则返回None

先从当前过程的符号表开始逆序查询，找到第一个立即返回；若当前表不存在，到全局静态区继续找

参数为表示符ID

#### errNoDefine(messType, IDName, mess2='未定义',lineNo=0)

错误处理输出模板；

参数依次为错误消息，ID名称，自定义消息，行号

#### existDef(table=[], ID=None)

判断ID在当前作用域是否已定义，若是，返回相应item，否则返回空

参数：列表，ID名称

#### newLabel()

新建标号

#### nextQuad()

下一条四元式标号

#### newTmp()

新建中间变量

#### printQtua(labelNum=None, op='\_', arg1='\_', arg2='\_', result='\_')

打印四元式

参数：标号，操作符，参数1，参数2，结果

#### showQuad()

显示四元式，写到文件，同时打印控制台

#### backPatch(\_list=[], quad=0)

翻译if-then-else 和 while时的回填函数

参数：回填链表，目的标号

### 3.1.4 汇编生成器

#### getLabelDict()

返回值：基本块标签，函数标签位置

#### binop(code=[], asmCode=[], op='+')

二元算术转换，即四则运算

参数：四元式，汇编代码表，操作符

返回值：汇编代码

#### tranToAsm(quadCode)

将四元式转换为汇编代码；

参数：quadCode为四元式列表

返回值：转换后的汇编代码列表

#### outputASM(asmCode, fileName='output.asm')

输出汇编头部信息，代码，数据段

参数： asmCode：汇编代码列表； fileName：输出的文件名

### 3.1.5 通用接口

#### init()

初始化全局变量

#### setData(\_data)

设置输入串

参数：字符串

#### getData()

获取输入串

#### get\_Grammar()

语法分析接口

返回值：语法分析器

#### Analysis(file)

语法/语义分析接口，主要给ui界面调用

参数：file：文件名

返回：0：语法正确； 1：语法错误； 2：文件错误

## 3.2 系统代码（必须有注释）

### 3.2.1 词法分析器

# ====================================================================

# 文件名：c\_lexer.py

# 功能：基于python-lex的C-词法分析器

# 作者： 常为， 荆顺吉

# 时间：2019/12/29

#======================================================================

# \*接口说明\*

#   - getToken() : 获取下一个token

#   返回值：LexToken(str type, repr value, int lineno, int lexpos)

#         词法对象的参数依次为：类型，值，行号，在字符串中的偏移（从1开始）

#

# \*如何使用\*

#   - 作为模块使用时注释/关闭末尾的测试代码，调用getToken即可

#

# \*有用参考\*

#   - 英文文档：http://www.dabeaz.com/ply/ply.html （强烈推荐）

#   - 中文文档：https://www.cnblogs.com/P\_Chou/p/python-lex-yacc.html

#

# \*实现思路\*

#   三步走：1.定义tokens 2.对每个token定义正则表达式 3.输入字符串，调用lex构建分析器

# =====================================================================

error\_num = 0 #用于记录错误次数

errorfile = None # 错误文件

parser = 0 # lexer

# =====================================================================

# 1. tokens 列表定义

#   tokens必须要有，多于一个字符的终结符的命名写在这

#   里面的每个符号都要有对应正则定义,且以 t\_ 开头，详细见后文

# =====================================================================

tokens = (

    'INT','VOID','IF','ELSE','WHILE','RETURN','NUM','ID','GE','LE','EE','NE','ANNOTATION'

    )

# =====================================================================

# 1. literals 列表定义

#   一个字符的终结符的命名写在这

#   字符对应终结符，不需要正则定义

# =====================================================================

literals = ['=','+','-','\*','/', '(',')',';','<','>','{','}',',','[',']']

# =====================================================================

# 2.以下为各个token的正则表达式定义，采用 t\_tokenName 的命名方式

# 若要对其进行运算，可通过定义函数的方式

# =====================================================================

t\_INT = *r*'int'

t\_VOID = *r*'void'

t\_IF = *r*'if'

t\_ELSE = *r*'else'

t\_WHILE = *r*'while'

t\_RETURN = *r*'return'

t\_ID    = *r*'(?!int|void|if|else|while|return)[a-zA-Z\_][a-zA-Z0-9\_]\*'

t\_NUM = *r*'[0-9]+'

t\_GE = *r*'>='

t\_LE = *r*'<='

t\_EE = *r*'=='

t\_NE = *r*'!='

#定义忽略注释的正则表达式

*def* t\_ANNOTATION(*t*):

#    r'/\\*([a-zA-Z0-9 \_]|\r|\n|\t)\*\\*/' # 第一行写正则表达式

*r*'(/\\*(.|\n)\*?\\*/)|(\/\/.\*)'

    t.lexer.lineno += t.value.count('\n') # 累计行数

    pass # 表示忽略该token

# 定义忽略字符的正则表达式，忽略空格回车换行

t\_ignore = " \t\r"

#======================================================================

# 通过函数定义正则表达式，可以增加额外动作，比如给value重新赋值，或增加其余字段信息

#======================================================================

# 增加行数

*def* t\_newline(*t*):

*r*'\n+'

    t.lexer.lineno += t.value.count("\n")

    pass

# 错误处理：输出错误符号，行数，列数后跳过当前错误继续扫描

*def* t\_error(*t*):

    global error\_num

    global errorfile

    if(errorfile is None):

        errorfile = open("error.out",'w',*encoding*='UTF-8')

    tmp = t.lexpos - getData().rfind('\n',0,t.lexpos)

    errorfile.write("LEX : Illegal character : "+t.value[0]+"    line :"+*str*(t.lexer.lineno)+  '  col: '+*str*(tmp)+'\n')

    error\_num+=1

    t.lexer.skip(1)

import ply.lex as lex # 导入python lex模块

#lex.lex()# 调用Lex模块，构建词法分析器

### 3.2.2 语法分析和语义分析

#======================================================================

#产生式列表-语法分析过程中夹杂着语义分析（每个产生式下方插入的为语义动作）

#======================================================================

# 开始符号：program

*def* p\_program\_1(*p*):

    '''program : declaration\_list'''

# 声明列表

*def* p\_declaration\_list\_1(*p*):

    '''declaration\_list : declaration\_list declaration'''

*def* p\_declaration\_list\_2(*p*):

    '''declaration\_list : declaration'''

# 变量和函数声明

*def* p\_declaration\_1(*p*):

    '''declaration : var\_declaration'''

    # 全局变量重定义检查

    item = existDef(staticSymTab, p[1].name)

    if item: # 重定义

        errNoDefine('标识符',p[1].name,'重定义',*lineNo*=-1) # 行号多一行？

    staticSymTab[p[1].name] = p[1] # 全局静态符号表

    print('---staticVar----',p[1])

*def* p\_declaration\_2(*p*):

    '''declaration : fun\_declaration'''

    if p[1]:

        staticSymTab[p[1].name] = p[1] # 全局静态符号表

        print('----staticfunc----',p[1])

# 变量声明具体定义：普通变量；一维数组

*def* p\_var\_declaration\_1(*p*):

    '''var\_declaration : type\_specifier ID ';' '''

    p[0] = SymTabItem(*\_type*='var', *dataType*=p[1], *name*=p[2])

*def* p\_var\_declaration\_2(*p*):

    '''var\_declaration : type\_specifier ID '[' NUM ']' ';' '''

    p[0] = SymTabItem(*\_type*='array', *dataType*=p[1], *name*=p[2], *arrayLen*=p[4])

# 变量/函数声明类型：INT;VOID

*def* p\_type\_specifier\_1(*p*):

    '''type\_specifier : INT'''

    p[0] = 'INT'

*def* p\_type\_specifier\_2(*p*):

    '''type\_specifier : VOID'''

    p[0] = 'VOID'

# 函数声明：头部；过程体

*def* p\_fun\_declaration\_1(*p*):

    '''fun\_declaration : type\_specifier ID '(' params ')' '''

    p[0] = SymTabItem(*\_type*='func', *name*=p[2], *returnType*=p[1], *args*=p[4], *codeNum*=nextQuad())

    funcSymTab.append('$')

    if p[4] and p[4][0] != '$' and p[4][0] != 'VOID':

        for arg in p[4]: #保存函数形式参数与当前作用域

            funcSymTab.append(SymTabItem(*\_type*=arg[0], *dataType*=arg[1], *name*=arg[2]))

#            print('arg',arg)

*def* p\_fun\_declaration\_2(*p*):

    '''fun\_declaration : compound\_stmt'''

    print('\*\*\*\*')

    for i in funcSymTab:

        if i != '$':

            print('fd==',i)

    funcSymTab.clear() # 函数结束，清空当前过程的符号表

#def p\_E1(p): # 记录函数声明

#    '''E1 : '''

#

# 参数定义:二维数组存放[[type,dataType,name],[]]

*def* p\_params\_1(*p*):

    '''params : param\_list'''

    p[0] = p[1]

*def* p\_params\_2(*p*):

    '''params : VOID'''

    p[0] = ['VOID']

*def* p\_params\_empty(*p*):

    '''params : '''

*def* p\_param\_list\_1(*p*):

    '''param\_list : param\_list ',' param'''

    p[1].append(p[3]) # 不可直接赋值，append无返回值

    p[0] = p[1]

#    print('p1list', p[1], p[0])

*def* p\_param\_list\_2(*p*):

    '''param\_list : param'''

    p[0] = [p[1]]

*def* p\_param\_1(*p*):

    '''param : type\_specifier ID'''

    p[0] = ['var',p[1],p[2]]

*def* p\_param\_2(*p*):

    '''param : type\_specifier ID '[' ']' '''

    p[0] = ['array',p[1],p[2]]

# 函数体

*def* p\_compound\_stmt\_1(*p*):

    '''compound\_stmt : '{' E1 local\_declarations statement\_list '}' '''

#    '''compound\_stmt : '{' local\_declarations statement\_list '}' '''

    p[0] = p[4]

    l = len(funcSymTab)

    for i in range(l-1, -1, -1): # 逆序查找，直到$出现，删除这之间所有变量（模拟栈）

        print('          ---localFunc[i]',funcSymTab[i])

        if funcSymTab[i] == '$':

            funcSymTab.pop()

            break

        else:

            funcSymTab.pop()

*def* p\_E1(*p*): # 作用域控制标记，{}

    ''' E1 : '''

    funcSymTab.append('$')

    print('$$$$$$$$$$$$$$$$$$$$$$$$$')

*def* p\_local\_declarations\_1(*p*):

    '''local\_declarations : local\_declarations var\_declaration'''

    # 局部变量重定义检查

    item = existDef(funcSymTab, p[2].name)

    if item: # 重定义

        errNoDefine('标识符',p[2].name,'重定义', -1) # 行号减一？？

    # 填入符号表

    funcSymTab.append(p[2])

*def* p\_local\_declarations\_empty(*p*):

    '''local\_declarations : '''

# 语句列表

*def* p\_statement\_list\_1(*p*):

    '''statement\_list : statement\_list statement'''

    p[0] = p[1]

*def* p\_statement\_list\_empty(*p*):

    '''statement\_list : '''

    p[0] = Node('statement\_list', 'state\_list')

# 表达式；符合语句；选择；循环；返回

*def* p\_statement\_1(*p*):

    '''statement : expression\_stmt'''

    p[0] = p[1]

*def* p\_statement\_2(*p*):

    '''statement : compound\_stmt'''

    p[0] = p[1]

*def* p\_statement\_3(*p*):

    '''statement : selection\_stmt'''

    p[0] = p[1]

*def* p\_statement\_4(*p*):

    '''statement : iteration\_stmt'''

    p[0] = p[1]

*def* p\_statement\_5(*p*):

    '''statement : return\_stmt'''

    p[0] = p[1]

    if p[1] == 'VOID':

        printQtua(newLabel(), 'Return', '\_', '\_', '\_')

    else:

        printQtua(newLabel(), 'Return', p[1].name, '\_', '\_')

    p[0] = Node('statement', 'name')

# 表达式语句

*def* p\_expression\_stmt\_1(*p*):

    '''expression\_stmt : expression ';' '''

    p[0] = p[1]

*def* p\_expression\_stmt\_2(*p*):

    '''expression\_stmt : ';' '''

# IF语句

# N必须同时加，否则出现reduce/reduce冲突

*def* p\_selection\_stmt\_1(*p*):

    '''selection\_stmt : IF '(' expression ')' M statement N'''

    backPatch(p[3].trueList, p[5])

    p[0] = Node('selection\_stmt', 'selec\_stmt')

    code.pop() # 删除N推入多余的记录

    global labelNum

    labelNum -= 1 # 标号位置也要回退1

    p[0].nextList = p[3].falseList + p[6].nextList

    backPatch(p[0].nextList, p[7][0])

#    backPatch(p[3].falseList, p[4])

# 第三个M用来回填nextList，即跳过S2

*def* p\_selection\_stmt\_2(*p*):

    '''selection\_stmt : IF '(' expression ')' M statement N ELSE M statement M'''

    backPatch(p[3].trueList, p[5])

    backPatch(p[3].falseList, p[9])

    p[0] = Node('selection\_stmt\_2', 'selec\_stmt\_2')

    # S.nextList = merge(S1.nextList+N.nextList+S2.nextList)

    p[0].nextList = p[6].nextList+ p[7] + p[10].nextList

    backPatch(p[0].nextList, p[11])

# 增加空产生式，作为标记

*def* p\_M(*p*): # 下一个语句开始

    '''M :'''

    p[0] = nextQuad()

*def* p\_N(*p*): # 无条件跳转

    '''N :'''

    p[0] = [nextQuad()] # N.nextlist = makeList(nextquad)

    printQtua(newLabel(), 'j')

# WHILE语句:增加两个M标号

*def* p\_iteration\_stmt\_1(*p*):

    '''iteration\_stmt : WHILE M '(' expression ')' M statement M'''

    backPatch(p[7].nextList, p[2])

    backPatch(p[4].trueList, p[6])

    p[0] = Node('while', 'while')

    p[0].nextList = p[4].falseList

    printQtua(newLabel(), 'j', *result*=p[2])

    backPatch(p[0].nextList, p[8]+1) # 增加一条while最后无条件跳转

# RETURN语句

*def* p\_return\_stmt\_1(*p*):

    '''return\_stmt : RETURN ';' '''

    p[0] = 'VOID'

*def* p\_return\_stmt\_2(*p*):

    '''return\_stmt : RETURN expression ';' '''

    p[0] = p[2]

# 表达式具体分解

*def* p\_expression\_1(*p*):

    '''expression : var '=' expression'''

    tmp = newLabel()

    printQtua(tmp, p[2], p[3].name, '\_', p[1].name) # 先产生代码

    p[0] = Node('expression', p[1].name, *children*=[p[1],p[3]])

*def* p\_expression\_2(*p*):

    '''expression : simple\_expression'''

    p[0] = p[1]

# 变量：普通变量或数组

*def* p\_var\_1(*p*):

    ''' var : ID'''

    p[0] = Node('var', p[1])

    item = getID(p[1])

    if not item:

        errNoDefine('变量', p[1])

    elif item.\_type != 'var':

        if item.\_type == 'array':

            errNoDefine('变量', p[1], '已定义为 数组')

        else:

            errNoDefine('变量', p[1], '已定义为 函数，不可作为左值')

# 一维数组引用：a[m] => T = a[m\*dataType.width]，[]中填偏移量

*def* p\_var\_2(*p*):

    ''' var : ID '[' expression ']' '''

    tmp1 = newTmp()

    printQtua(newLabel(), '\*', p[3].name, '4', tmp1) # 偏移：T1 = expression.val \* 4

    tmp2 = newTmp()

    p[0] = tmp2 # 其实并不需要记录tmp2的标号内有什么，只要记住他的索引即可

    printQtua(newLabel(), '[]', p[1], tmp1, tmp2) # 引用：T2 = ID[T1]

    # 建树

    p[0] = Node('var', tmp2)

    # 检查数组是否定义，是否越界

    item = getID(p[1])

    if not item:

        errNoDefine('数组', p[1])

    elif item.\_type != 'array':

        errNoDefine('变量', p[1], '不是数组类型')

    elif p[3].ttype == 'NUM' and *int*(item.arrayLen) <= *int*(p[3].name):

        errNoDefine('数组', p[1], '下标越界'+' 最大长度:'+*str*(item.arrayLen))

#=====================================

# 以下为表达式定义：优先级：关系符 < 加减 < 乘除

# 简单表达式：关系表达式

*def* p\_simple\_expression\_1(*p*):

    '''simple\_expression : additive\_expression relop additive\_expression'''

#    p[0] = RelOpExpr('simple\_expression', newTmp(), children=[p[1],p[3]])

    p[0] = Node('simple\_expression', newTmp(), *children*=[p[1],p[3]])

    p[0].trueList.append(nextQuad())

    p[0].falseList.append(nextQuad()+1)

    printQtua(newLabel(), 'j'+p[2], p[1].name, p[3].name, 0)

    printQtua(newLabel(), 'j', '\_', '\_', 0)

*def* p\_simple\_expression\_2(*p*):

    '''simple\_expression : additive\_expression'''

    p[0] = p[1]

# 6种关系符号

*def* p\_relop\_1(*p*):

    '''relop : LE'''

    p[0] = p[1]

*def* p\_relop\_2(*p*):

    '''relop : '<' '''

    p[0] = p[1]

*def* p\_relop\_3(*p*):

    '''relop : '>' '''

    p[0] = p[1]

*def* p\_relop\_4(*p*):

    '''relop : GE'''

    p[0] = p[1]

*def* p\_relop\_5(*p*):

    '''relop : EE'''

    p[0] = p[1]

*def* p\_relop\_6(*p*):

    '''relop : NE'''

    p[0] = p[1]

# 四则运算：已考虑四则运算优先级

*def* p\_additive\_expression\_1(*p*):

    '''additive\_expression : additive\_expression addop term'''

    p[0] = newTmp()

    printQtua(newLabel(), p[2], p[1].name, p[3].name, p[0])

    p[0] = Node('additive\_expression', p[0], *children*=[p[1],p[3]])

*def* p\_additive\_expression\_2(*p*):

    '''additive\_expression : term'''

    p[0] = p[1]

# 关系符直接传递

*def* p\_addop\_1(*p*):

    '''addop : '+' '''

    p[0] = p[1]

*def* p\_addop\_2(*p*):

    '''addop : '-' '''

    p[0] = p[1]

# 生成代码

*def* p\_term\_1(*p*):

    '''term : term mulop factor'''

    p[0] = newTmp() # 新建标号

    printQtua(newLabel(), p[2], p[1].name, p[3].name, p[0])

    p[0] = Node('term', p[0], *children*=[p[1],p[3]])

*def* p\_term\_2(*p*):

    '''term : factor'''

    p[0] = p[1]

# 关系符直接传递

*def* p\_mulop\_1(*p*):

    ''' mulop : '\*' '''

    p[0] = p[1]

*def* p\_mulop\_2(*p*):

    ''' mulop : '/' '''

    p[0] = p[1]

# factor可以是表达式、简单变量，函数调用，常数

# 不产生代码

*def* p\_factor\_1(*p*):

    '''factor : '(' expression ')' '''

    p[0] = Node('factor', p[2].name)

*def* p\_factor\_2(*p*):

    '''factor : var'''

    p[0] = Node('factor', p[1].name)

*def* p\_factor\_3(*p*):

    '''factor : call'''

    p[0] = Node('factor', p[1].name)

*def* p\_factor\_4(*p*):

    '''factor : NUM'''

    p[0] = Node('NUM', p[1])

# 调用函数

*def* p\_call\_1(*p*):

    ''' call : ID '(' args ')' '''

    if p[3]: # 存在实参

        for arg in p[3].name: # 遍历实参，生成param代码

            printQtua(newLabel(), 'Param', arg[1], '\_', '\_')

    tmpLabel = newTmp()

    plen = 0 # 参数个数

    if p[3]: # 处理无参情况

        plen = len(p[3].name)

    printQtua(newLabel(), 'Call', p[1], plen, tmpLabel) # 需输出函数名和参数个数

    # 建树

    p[0] = Node('call', tmpLabel)

    # 参数个数检查与类型检查

    item = getID(p[1])

    if not item: # 函数未定义

        errNoDefine('函数', p[1])

    else:

        # 个数检查

        if not (p[3] != None and item.args != None and len(item.args) == len(p[3].name) or (item.args != None and item.args[0]=='VOID' and p[3]==None)):

            errNoDefine('函数参数个数不一致', '\_','\_')

*def* p\_args\_1(*p*):

    '''args : arg\_list'''

    p[0] = p[1]

*def* p\_args\_empty(*p*):

    '''args : '''

# 实参拼接

*def* p\_arg\_list\_1(*p*):

    ''' arg\_list : arg\_list ',' expression'''

    p[1].name.append((p[3].ttype, p[3].name))

    p[0] = Node('arg\_list', *name*=p[1].name)

*def* p\_arg\_list\_2(*p*):

    ''' arg\_list : expression'''

    p[0] = p[1]

    p[0].name = [(p[1].ttype, p[0].name)]

#错误处理，输出错误所在单词

*def* p\_error(*p*):

    global error\_num

    global errorfile

    global parser

    if(errorfile is None):

        errorfile = open("error.out",'w',*encoding*='UTF-8')

    if p :

        parser.errok()

        errorfile.write("Syntax :  error at `" + p.value+"`   line : "+*str*(p.lineno)+

                        '\nSkip this token and Step into Normal Mode\n')

    else:

        errorfile.write("Syntax : error at EOF \n")

    error\_num+=1

import yacc\_ui as yacc

#import ply.yacc\_ui as yacc

# 对输出字符串的获取与设置

data = '' # 存储输入字符串

*def* setData(*\_data*):

    global data

    data = \_data

*def* getData():

    return data

# =====================================================================

# 初始化全局变量

# =====================================================================

*def* init():

    global staticSymTab

    staticSymTab.clear()

    global funcSymTab

    funcSymTab.clear()

    global labelNum

    labelNum = 0

    global tmpNum

    tmpNum = 0

    global code

    code = [0]

# =====================================================================

# 语法/语义分析接口，主要给ui界面调用

# 输入：file：文件名

# 返回：0：语法正确； 1：语法错误； 2：文件错误

# =====================================================================

*def* Analysis(*file*):

    init()

    lexer = lex.lex()# 调用Lex模块，构建词法分析器

    global parser

    global errorfile

    global error\_num

    errorfile = None

    error\_num = 0

    parser = yacc.yacc()

    try:

        with open(file, *encoding*='UTF-8')as f:

            contents = f.read()

#            data = contents # 词法错误列号处理

            setData(contents)

        parser.parse(contents,*debug*=1)

        #语法正确

        if(error\_num==0):

            showQuad()

            return 0

        #语法错误

        else:

            errorfile.close()

            return 1

    except *EOFError*:

        return 2

# =====================================================================

# 一遍扫描的回填实现；抽象语法树的建立

# 四元式的标号，中间变量控制

# =====================================================================

labelNum = 0 # 四元式标号记录

tmpNum = 0 # 中间变量数量累计

# 新建标号

*def* newLabel():

    global labelNum

    labelNum += 1

    return labelNum

# 下一条四元式标号

*def* nextQuad():

    return labelNum+1

# 新建中间变量

*def* newTmp():

    global tmpNum

    tmpNum += 1

    return 'T'+*str*(tmpNum)

code = [0] # 存放四元式代码

# 打印四元式:标号，操作符，参数1，参数2，结果

*def* printQtua(*labelNum*=None, *op*='\_', *arg1*='\_', *arg2*='\_', *result*='\_'):

    code.append([op, arg1, arg2, result])

# 显示四元式，写到文件，同时打印控制台

*def* showQuad():

    with open("code.out","w", *encoding*='UTF-8') as f:

        f.write(*str*(len(code))+'\n')

        for i in range(1,len(code)):

            f.write(*str*(code[i][0])+' '+ *str*(code[i][1])+' '+ *str*(code[i][2])+' ' +*str*(code[i][3])+'\n')

    for i in range(1,len(code)):

        print(*str*(i)+') : ', code[i])

# 翻译if-then-else 和 while时的回填函数

*def* backPatch(*\_list*=[], *quad*=0):

    for i in \_list:

        code[i][3] = quad

# AST节点

*class* Node:

*def* \_\_init\_\_(*self*,*\_type*,*name*,*children*=None,*leaf*=None):

        self.ttype = \_type # 类型

        self.name = name # 名字

        if children: # 孩子

            self.children = children

        else:

            self.children = [ ]

        self.leaf = leaf # 叶子结点

        # 条件语句

        self.trueList = []

        self.falseList = []

        # S.nextList：语句

        self.nextList = []

### 3.2.3 符号表

# =====================================================================

# 符号表

# =====================================================================

*class* SymTabItem:

*def* \_\_init\_\_(*self*, *\_type*=None, *dataType*=None, *name*=None, *arrayLen*=None, *args*=None, *returnType*=None, *codeNum*=0):

        self.\_type = \_type # ID类型，变量，数组，函数

        self.dataType = dataType # 变量和数组的数据类型

        self.name = name # ID名字

        self.arrayLen = arrayLen # 数组长度

        self.args = args # 函数参数列表

        self.returnType = returnType # 函数返回值类型

        self.codeNum = codeNum # 四元式开始的位置

*def* \_\_str\_\_(*self*): # 打印输出

        if self.\_type == 'var':

            ret = [('type',self.\_type), ('dataType', self.dataType), ('name', self.name)]

        elif self.\_type == 'array':

            ret = [('type',self.\_type), ('dataType', self.dataType), ('name', self.name), ('arrayLen', self.arrayLen)]

        else :

            ret = [('type',self.\_type), ('returnType', self.returnType), ('name', self.name), ('args', self.args), ('codeNum', self.codeNum)]

        return *str*(ret)

staticSymTab = *dict*() # 全局符号表，存放静态声明的变量，数组，函数

funcSymTab = [] # 过程的符号表，不同作用域以$分割

# =====================================================================

# 根据ID查询符号表，存在则返回相应表项，否则返回None

# 先从当前过程的符号表开始逆序查询，找到第一个立即返回；若当前表不存在，到全局静态区继续找

# =====================================================================

*def* getID(*ID*):

    for i in range(len(funcSymTab)-1, -1, -1):

        t = funcSymTab[i]

        if t != '$' and t.name == ID:

            return t

    if staticSymTab.get(ID):

        return staticSymTab[ID]

    return None

# 错误处理输出模板

*def* errNoDefine(*messType*, *IDName*, *mess2*='未定义',*lineNo*=0):

    global error\_num

    error\_num += 1

    global errorfile

    if(errorfile is None):

        errorfile = open("error.out",'w',*encoding*='UTF-8')

    errorfile.write('Sematic : Error'+*str*(error\_num)+'   line:'+*str*(lex.lexer.lineno+lineNo) + '   '+*str*(messType)+': 【'+*str*(IDName)+'】'+ *str*(mess2)+'\n')

    print('Error'+*str*(error\_num)+'   行号:'+*str*(lex.lexer.lineno+lineNo) + '   '+messType+': 【'+IDName+'】'+ mess2)

# 判断ID在当前作用域是否已定义，若是，返回相应item，否则返回空

*def* existDef(*table*=[], *ID*=None):

    if *type*(table) == *dict*:

        print('dict',table.get(ID))

        return table.get(ID)

    for i in range(len(table)-1,-1,-1):

        if table[i] == '$':

            return None

        if table[i].name == ID:

            return table[i]

    return None

### 3.2.4 汇编生成器

# =====================================================================

# 四元式转换为汇编代码

# =====================================================================

# 返回基本块标签，函数标签位置

*def* getLabelDict():

    labelDict = {} # 记录需要表示的跳转标签，jmp,jrop,

    funcDict={} # 函数标签；call

    for c in code:

        if c != 0:

            if c[0][0] == 'j': # 有条件/无条件跳转

#                print('j',c)

                labelDict[c[3]] = '$Label'+*str*(c[3]) # 生成跳转标签

    for item in staticSymTab.keys():

        tmp = staticSymTab.get(item)

        if tmp.\_type == 'func':

            funcDict[tmp.codeNum] = tmp.name

    return labelDict, funcDict

# 关系符映射

relopDict = {

        'j<': 'JL',

        'j>': 'JG',

        'j>=': 'JGE',

        'j<=': 'JLE',

        'j==': 'JE',

        'j!=': 'JNE'

        }

# 二元算术转换，即四则运算

*def* binop(*code*=[], *asmCode*=[], *op*='+'):

    if code == []:

        return asmCode

    asm = 'MOV EAX, '+ code[1]

    asmCode.append(asm)

    asm = 'MOV EBX, '+ code[2]

    asmCode.append(asm)

    asm = op+' EAX, EBX'

    asmCode.append(asm)

    asm = 'MOV '+code[3]+', EAX'

    asmCode.append(asm)

    return asmCode

# 将四元式转换为汇编代码；

# quadCode为四元式列表；返回值为转换后的汇编代码列表

*def* tranToAsm(*quadCode*):

    labelDict , funcDict = getLabelDict()

    asmCode = []

    i = 0

    for code in quadCode:

        if code == 0:

            continue

        asm = ''

        i += 1

        # 函数或跳转标签存在，先打印

        funLabel = funcDict.get(i)

        label = labelDict.get(i)

        if funLabel != None:

            asmCode.append(funLabel+':')

        if label != None:

            asmCode.append(label+':')

#        print(code)

        # 继续处理四元式，根据动作类型进入不同处理

        if code[0][0] == 'j':

            if code[0] == 'j': # 无条件跳转

                asm = 'JMP '+labelDict[code[3]] # 无条件跳转

                asmCode.append(asm)

            else: # 分情况跳转

                # 先产生cmp，然后根据flag跳转

                asm = 'CMP '+ code[1]+' ,'+code[2]

                asmCode.append(asm)

                asm = relopDict[code[0]]+' '+labelDict[code[3]]

                asmCode.append(asm)

        # 参数压栈：PUSH、EAX

        elif code[0] == 'Param':

            asm = 'MOV EAX, '+ code[1]

            asmCode.append(asm)

            asm = 'PUSH EAX'

            asmCode.append(asm)

        # 调用函数

        elif code[0] == 'Call':

            asm = 'CALL '+code[1]

            asmCode.append(asm)

        # 函数返回

        elif code[0] == 'Return':

            asm = 'RET\n'

            asmCode.append(asm)

        elif code[0] == '+':

            asmCode = binop(code, asmCode, 'ADD')

        elif code[0] == '-':

            asmCode = binop(code, asmCode, 'SUB')

        elif code[0] == '/':

            asmCode = binop(code, asmCode, 'DIV')

        elif code[0] == '\*':

            asmCode = binop(code, asmCode, 'MUL')

        # 赋值语句

        elif code[0] == '=':

            asm = 'MOV EAX, '+code[1]

            asmCode.append(asm)

            asm = 'MOV '+code[3]+', EAX'

            asmCode.append(asm)

        # 数组引用求值

        elif code[0] == '[]':

            asm = "MOV EBX, "+ code[1]

            asmCode.append(asm)

            asm = 'MOV ECX, ' + code[2]

            asmCode.append(asm)

            asm =  'MOV EDX, ' + '[EBX + ECX\*4]'

            asmCode.append(asm)

            asm = 'MOV '+code[3]+', EDX'

            asmCode.append(asm)

    return asmCode

# 输出汇编头部信息，代码，数据段

# asmCode：汇编代码列表； fileName：输出的文件名

*def* outputASM(*asmCode*, *fileName*='output.asm'):

    with open(fileName, 'w', *encoding*='utf-8') as f: # 打开文件

        f.write('org 100H\n') # 起始偏移

        # 数据段

        f.write('\nSECTION data\n')

        for key in staticSymTab.keys():

            var = staticSymTab.get(key)

            if var.\_type == 'var':

                f.write('\t'+var.name+' dd 0\n') # 初始化变量

            elif var.\_type == 'array': # 数组初始化

                f.write('\t'+var.name+' times '+ *str*(var.arrayLen) +' dd 0\n')

        f.write('\tT'+' times '+ *str*(len(code)) +' dd 0\n')

        # 代码段

        f.write('\nSECTION code\n')

        f.write('\tMOV EAX, CS\n')

        f.write('\tMOV ESP, EAX\n') # 堆栈设置

        for asm in asmCode:

            if asm != 0:

                if asm.find(':') != -1:

                    f.write(asm+'\n')

                else:

                    f.write('\t'+asm+'\n')

# =====================================================================

# 接口：运行时生成相应的parse.out和parsetab.py文件供之后使用

# =====================================================================

#import yacc\_ui as yacc # 包含界面处理的yacc

*def* get\_Grammar():

    yacc.yacc()

### 3.2.5 UI界面

# =====================================================================

# 文件名：ui.py

# 功能：基于PyQt5的Cmini语法分析图形界面

# 作者： 武起龙

# 时间：2020/1/1

#======================================================================

#

# \*如何使用\*

#   - 运行ui.py后将分析文件拖入即可

#

# \*所需文件\*

#   - 语义分析器接口:sematic\_uipai.py

#   - sematic\_uipai.py所引用的特殊yacc文件:yacc\_ui.py

#

# \*功能\*

#   - 语法正确，则显示词法分析结果，语法分析过程和三地址代码，汇编代码

#   - 语法错误，则显示出错类型和位置

#   - 文件错误，则提示文件不匹配

#

# =====================================================================

# \*实现思路\*

#   1.构建主界面,出错界面和结果界面

#   2.在主界面中接受文件后将调用sematic\_uipai.py进行词法，语法，语义分析

#   3.根据分析结果调用相应界面

# =====================================================================

import sematic\_uiapi as Gram

import sys

from PyQt5.QtWidgets import \*

from PyQt5.QtGui import\*

openfile = None

#主界面

*class* MainWidget(*QDialog*):

*def* \_\_init\_\_(*self*):

*super*().\_\_init\_\_()

        self.initUI()

    #界面设置

*def* initUI(*self*):

        #规定大小和标题

        self.resize(600,400)

        self.setMinimumSize(600,400)

        self.setMaximumSize(600,400)

        self.setWindowTitle('CMCompiler演示程序')

        #允许拖入文件

        self.setAcceptDrops(1)

        #提示信息

        la = QLabel(self)

        la.setGeometry(0,100,600,200)

        la.setContentsMargins(120,0,0,0)

        ft = QFont()

        ft.setPointSize(20)

        la.setFont(ft)

        la.setText("请将要分析的C语言文件拖入")

        self.la = la

        #拖入文件不符合标准时的提示信息

        lae = QLabel(self)

        lae.setGeometry(0,100,600,200)

        lae.setContentsMargins(120,0,0,0)

        ft = QFont()

        ft.setPointSize(20)

        lae.setFont(ft)

        lae.setText("该文件无效，请拖入有效文件")

        self.lae = lae

        lae.hide()

    #重写拖入文件事件

*def* dragEnterEvent(*self*,*event*):

        if(event.mimeData().hasUrls()):

            event.acceptProposedAction()

        else: event.ignore()

    #重写文件放下事件

*def* dropEvent(*self*,*event*):

        mimedata = event.mimeData()

        urllist = mimedata.urls()

        filename = urllist[0].toLocalFile()

        global openfile

        #判断文件是否为C语言文件

        isC = filename.endswith(".c")

#        if(isC):

        if(True):

            openfile = filename

            res = Gram.Analysis(filename)

            #文件打开失败

            if(res==2):

                self.la.hide()

                self.lae.show()

            #文件分析出错，调出出错界面

            elif(res==1):

                e = ErrorWidget()

                self.hide()

                self.la.show()

                self.lae.hide()

                e.exec()

                self.show()

            #文件分析成功，调出结果界面

            else:

                r = ResultWidget()

                self.hide()

                self.la.show()

                self.lae.hide()

                r.exec()

                self.show()

        else:

            self.la.hide()

            self.lae.show()

#出错界面

*class* ErrorWidget(*QDialog*):

*def* \_\_init\_\_(*self*):

*super*().\_\_init\_\_()

        self.initUI()

*def* initUI(*self*):

        #规定大小和标题

        self.resize(900,400)

        self.setMinimumSize(900,400)

        self.setMaximumSize(900,400)

        #

        global openfile

        txt = QTextEdit(self)

        txt.setGeometry(590,20,290,360)

        f = open(openfile,'r',*encoding*='UTF-8')

        lines = f.readlines()

        strline = ''

        i=0

        for line in lines:

            strline+=(('{*:<3*}'.format(*str*(i+1)))+':  '+line)

            i+=1

        txt.setText(strline)

        #设为只读模式

        txt.setReadOnly(1)

        self.setWindowTitle('文件语句错误')

        #错误信息显示

        te = QTextEdit(self)

        te.setGeometry(20, 20, 480, 360)

        f = open("error.out",'r',*encoding*='UTF-8')

        lines = f.readlines()

        strline = ''

        for line in lines:

            strline+=(line)

        te.setText(strline)

        #设为只读模式

        te.setReadOnly(1)

        #右下返回按钮

        btn = QPushButton("分析其他文件",self)

        btn.setGeometry(500,360,90,20)

        #建立按钮被点击和btnClicked函数的连接

        btn.clicked.connect(self.btnClicked)

*def* btnClicked(*self*):

        self.close()

#结果界面

*class* ResultWidget(*QDialog*):

*def* \_\_init\_\_(*self*):

*super*().\_\_init\_\_()

        self.initUI()

    #界面设置

*def* initUI(*self*):

        #规定大小和标题

        self.resize(900,400)

        self.setMinimumSize(900,400)

        self.setMaximumSize(900,400)

        self.setWindowTitle('代码分析结果')

        #

        global openfile

        txt = QTextEdit(self)

        txt.setGeometry(590,20,290,360)

        f = open(openfile,'r', *encoding*='utf-8')

        lines = f.readlines()

        strline = ''

        i=0

        for line in lines:

            strline+=(('{*:<3*}'.format(*str*(i+1)))+':  '+line)

            i+=1

        txt.setText(strline)

        #设为只读模式

        txt.setReadOnly(1)

        #左侧列表信息

        leftlist = QListWidget(self)

        leftlist.insertItem(0, '词法分析结果')

        leftlist.insertItem(1, '语法分析流程')

        leftlist.insertItem(2, '四元式结果')

        leftlist.insertItem(3, '汇编结果')

        leftlist.setGeometry(20, 20, 90, 140)

        leftlist.setCurrentRow(0)

        #左下返回按钮

        btn = QPushButton("分析其他文件",self)

        btn.setGeometry(20,360,90,20)

        #右侧窗口信息

        Stack = QStackedWidget(self)

        Stack.setGeometry(110, 20, 470, 360)

        stack1 = QWidget()

        stack2 = QWidget()

        stack3 = QWidget()

        stack4 = QWidget()

        Stack.addWidget(stack1)

        Stack.addWidget(stack2)

        Stack.addWidget(stack3)

        Stack.addWidget(stack4)

        #窗口stack1显示

        tes1 = QTextEdit(stack1)

        tes1.resize(470,360)

        f = open("tokensanalysis.out",'r')

        lines = f.readlines()

        strline = ''

        for line in lines:

            strline+=(line)

        tes1.setText(strline)

        #设为只读模式

        tes1.setReadOnly(1)

        #窗口stack2显示

        tes2 = QTextEdit(stack2)

        tes2.resize(470,360)

        f = open("grammaranalysis.out",'r')

        lines = f.readlines()

        strline = ''

        for line in lines:

            strline+=(line)

        tes2.setText(strline)

        #设为只读模式

        tes2.setReadOnly(1)

        #窗口stack3显示

        f = open("code.out",'r')

        lines = f.readlines()

        tws3 = QTableWidget(*int*(lines[0]),4,stack3)

        tws3.resize(470,360)

        tws3.horizontalHeader().setSectionResizeMode(QHeaderView.Stretch)

        tws3.setHorizontalHeaderLabels(['Op','arg1','arg2','result'])

        tws3.setEditTriggers(QAbstractItemView.NoEditTriggers)

        i=1

        while i < *int*(lines[0]):

            temp = lines[i].split()

            tws3.setItem(i-1,0,QTableWidgetItem(temp[0]))

            tws3.setItem(i-1,1,QTableWidgetItem(temp[1]))

            tws3.setItem(i-1,2,QTableWidgetItem(temp[2]))

            tws3.setItem(i-1,3,QTableWidgetItem(temp[3]))

            i+=1

        #窗口stack4显示

        tes4 = QTextEdit(stack4)

        tes4.resize(470,360)

        f = open("output.asm",'r')

        lines = f.readlines()

        strline = ''

        for line in lines:

            strline+=(line)

        tes4.setText(strline)

        #设为只读模式

        tes4.setReadOnly(1)

        #建立按钮被点击和btnClicked函数的连接

        btn.clicked.connect(self.btnClicked)

        #建立leftlist和Stack连接

        leftlist.currentRowChanged.connect(Stack.setCurrentIndex)

    #按钮被点击事件

*def* btnClicked(*self*):

        self.close()

if \_\_name\_\_ == '\_\_main\_\_':

    app = QApplication(sys.argv)

    m = MainWidget()

    m.exec()

    sys.exit(app.exec\_())

# 第四章 系统测试

本部分包含系统使用说明与每个功能的黑盒测试截图说明。

## 4.1 正确性测试

本部分使用完全符合词法、语法和语义规则的测试用例《test\_完全正确.c》进行测试，验证程序的正确性处理。以下为具体步骤及截图。

1. 配置好如2.3节所要求的环境后，运行ui.py，出现主界面，如图4.1所示。



图 4.1 主界面

1. 拖入测试文件《test\_完全正确.c》，得到分析结果，默认展示词法分析结果，其最左侧为分析结果选项，中间面板为分析结果展示，最右侧为输入的测试源文件，具体如图4.2所示。

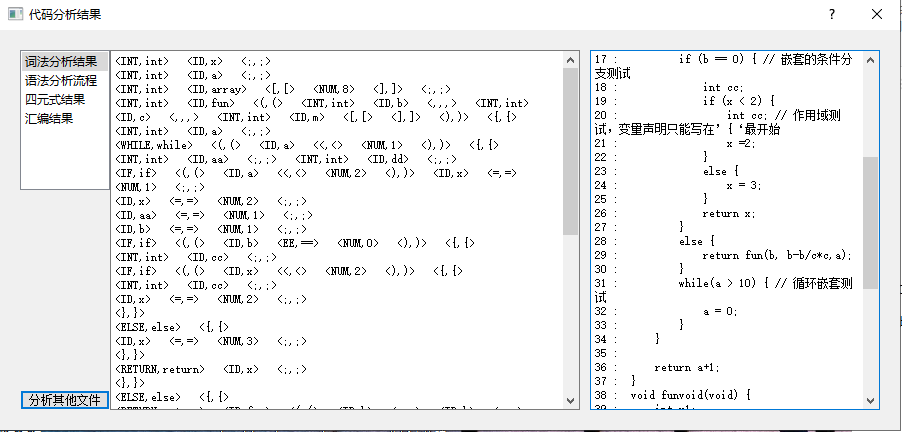


图 4.2 词法分析结果

语法分析结果如图4.3所示，体现了每一步的状态转移。

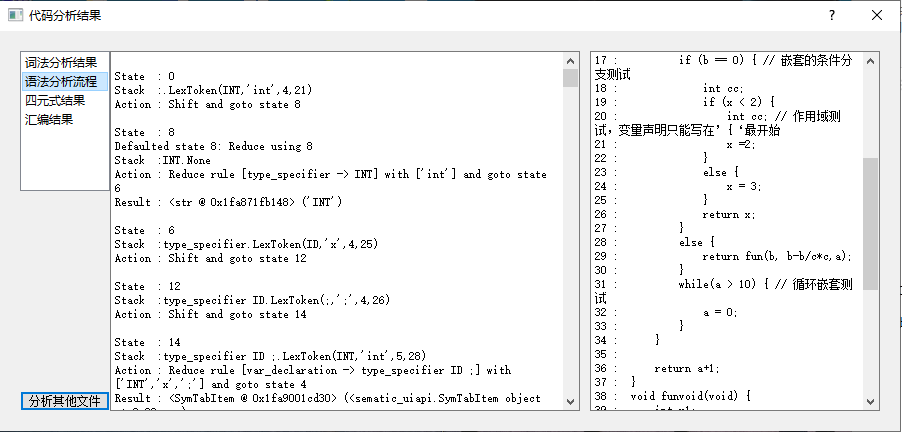


图 4.3 语法分析结果

四元式如图4.4所示，使用表格形式呈现。



图 4.4 四元式

汇编代码如图4.5所示。

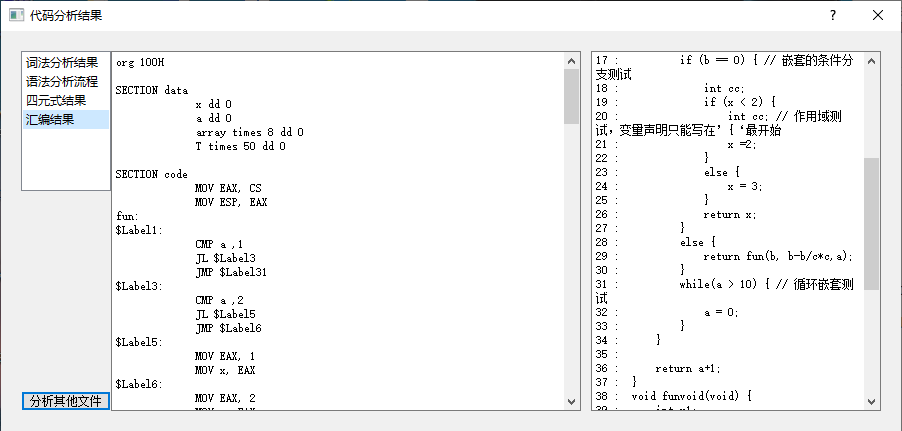


图 4.5 汇编代码

## 4.2 类型系统与作用域测试

使用存在语义错误—类型和作用域问题的测试用例《test\_类型检查\_作用域.c》进行测试。其错误种类包括标识符重定义、未定义，函数参数列表不一致，函数引用作为左值，数组下标越界，非数组使用下标索引等等，其余检测如图4.6所示。

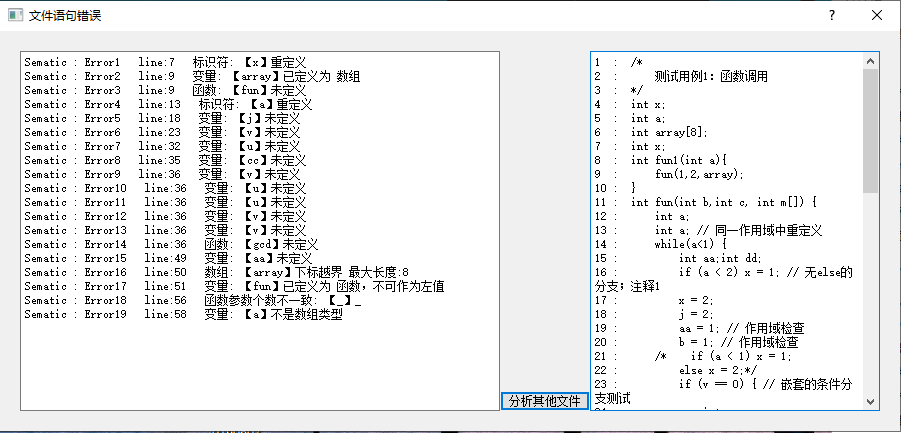


图 4.6 类型检查与作用域测试

## 4.3 错误处理测试

使用存在词法、语法错误的测试用例《test\_词法语法错误\_恢复测试.c》进行测试。词法错误可准确输出行列号，然后跳过一定数量字符继续处理；语法错误可准确输出行号，然后进入错误恢复状态，如图4.7所示。

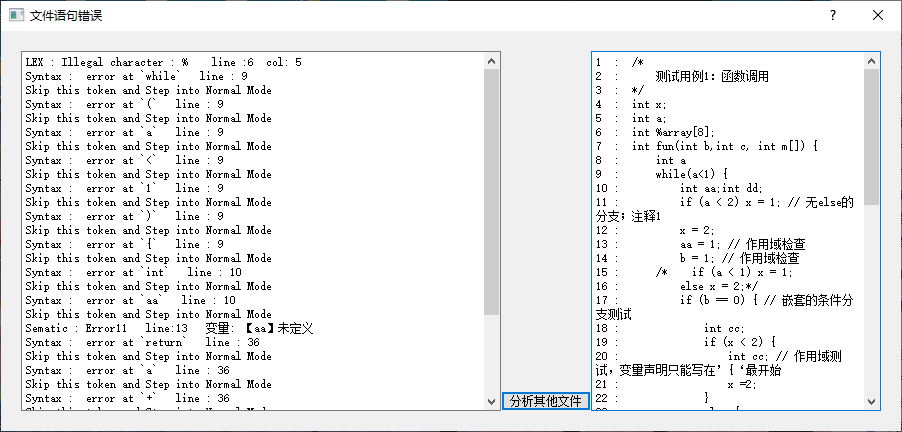


图 4.7 词法语法错误处理与恢复测试

# 第五章 课程设计心得

## 武起龙

本次编译原理课设中我负责语法分析和UI界面实现。

语法分析方面因为使用了YACC，所以实现起来并没有遇到什么困难。但是知其然更要知其所以然，我对yacc的源代码进行了较为深入的研读。

首先了解到yacc的构造分析表函数yacc()的实现过程和特殊之处——因为构造预测分析表要消耗大量的系统资源，所以函数会检测当前文件夹中是否已有相应文件，文件中的预测分析表的签名（即所有产生式）是否和需要构建预测分析表的产生式一致，不一致才会进行构建。

其次是yacc的分析函数parse()，其会根据传递的参数不同调用相应的函数，这些函数分析过程一致，不过在输出等方面会有差异。正因如此，我们对yacc进行了修改，为parse()函数添加了一个函数，该函数会进行正常的分析，但是会记录分析过程，以便在UI结果界面中展示。

最后便是yacc的错误处理模块，我们对其进行了一定的修改，具体的方法在上面的报告中已经描述，这里便不再赘述。

UI界面采用了PyQt5实现，具体过程没啥创新，算是强化了自己对于Qt的使用。

通过本次课设，我再次体会到了阅读源代码的头疼，也体会到了有一个好的注释对他人阅读代码时的巨大帮助。所以在功能实现之后我对代码进行了详尽的注释，一方面是再次走一遍自己的思路，查漏补缺，另一方面也是为了供以后他人在借鉴我代码时的方便。

最重要的心得，就是对于编译过程实现的深入了解。俗话说坐而论道不如起而行之，理论与实践相结合，将书本上的知识具象化，使我这种抽象能力较差的学生也能全面深入的理解编译原理。

## 常为

这次课设我负责词法分析器的部分，由于之前没有接触过ply.lex，所以大部分时间都花在了源文档的翻译和理解上。开始写的时候非常不顺利，其实就是因为文档中的内容没有理解透，有关词法分析的用法没有深刻理解，导致写起来很困难，等把文档多看几遍后再写起来就顺利很多。

除去文档中语法规则的特殊写法外，最困难的就是正则表达式的写法，由于ply.lex是严格依照正则表达式匹配来完成词法分析的，所以在写正则表达式的时候遇到了不少问题。当然，在完成后，我对正则表达式的理解也深刻了许多。

在做这一部分我另一个很深刻的体会是：读文档很重要。有时候甚至不用查找其他资料，文档中就有详尽的介绍、语法和代码，所以把文档读懂、理解透能使工作事半功倍。就像上学期计组的课设，我开始就是因为没有认真读i386的文档，导致实验写起来很困难。

另一个体会是代码的注释要详尽，小组合作，自己负责部分的注释要写得简练易懂，这样其他组员也容易理解。同时接口的统一也很重要，我的词法分析器要供语法分析时调用，统一的接口可以避免后续修改代码等麻烦情况。

## 张峥

最初了解到课设的任务时，觉得和之前实验内容相仿，没有那么难。但实际做的时候，无论是python还是使用的lex和yacc都需要从头了解学习，不过好在组长安排两个人一组进行语法分析，一方面可以减少编程中的细节错误，另一方面在查错和学习过程中能互补，因此在完成安排的任务时，也能更轻松，学到的东西也更多。

一开始，可能连python的字符串表达中单引号的使用都要测试好几次，到后来也可以通过读源码优化查错报错等，直到完成也是很有成就感。后来从github上看到，另一组他们的词法分析源码，里面的注释和命名都比我们要精致，也是比对着他们的注释，进行命名修改和注释的添加，虽然繁琐但也学会了耐心完成，以增加代码可读性。

最后一项工作是对比本次LALR与LL(1)文法有什么不同和阅读学习yacc完整的工作流程，也是在这个过程中，我们总结了几种错误处理方法和优缺点；而对阅读源码我并不擅长，是在武起龙同学的阅读讲解下，一起归纳了整个工作流程，然后绘制流程图。

总之，这次课程设计与其他课程设计最大的不同就是组队完成，除了组长的大量工作和合理安排外，其他同学的工作也不可或缺，另外在整个完成过程中，每个同学的成果和帮助都让我学到很多，也很感谢老师对课程设计的安排。

## 吴扬俊

（1）时间安排及项目分工：合理安排时间，管理项目进度（甘特图）；知人善用，根据每个同学的特点分配任务（一般分配超出其现有能力一点的任务，这样既可以完成任务，又有极大成就感）；协调组内关系。

虽然李老师提前两周布置了课程设计，但恰逢考试周，大家都没有精力去准备，真正开始写编译器仅4天时间。身为组长，加上对于编译器的陌生和时间上的紧张，感到巨大压力。考完试一结束，立刻组织讨论，根据每个人擅长的部分分配工作，同时采用结对编程方式，两两一组，提高效率。为了完成任务，跨年夜和元旦那天大家依旧在寝室调试bug到凌晨三点钟，好在大家效率比较高，经过两天学习和不断测试，讨论，迭代，总算完成了词法和语法分析部分，稍微松了口气，谁知道更可恶的语义分析还在后头。

（2）资料查找与分析对比：善用搜索引擎，图书馆和老师建议；主动分析对比已有方法优劣，选择合适自己的方案。

关于词法分析和语法分析是纯手写还是借用工具实现，借用工具用哪个工具好呢等问题，我们都经过仔细考量。

鉴于实验已经手动实现过词法和语法分析，而且目前时间紧迫，觉得没有必要手动再实现一遍，于是采用工具。现在关于lex和yacc工具很多，综合对比分析后，觉得采用PLY实现，其最适合我们目前状态，于是进入快速学习阶段，大家抱着英文文档啃。

（3）代码未动，测试先行：测试直接和需求挂钩，可见其重要性，别蒙头做到最后连是对是错都无法验证。而测试通常是我们忽略的环节，测试可以纠正完善我们在设计环节出现的漏洞，考虑多种边界情况。

（4）总体收获：个人编码能力提高是必然的，同时体会到把握整个系统的艰辛，整个团队进行项目开发的经历更受益匪浅，特别是一些非技术因素，通常这些是决定成败的关键。也明白一个道理，知识是死的，而方法论是活的，以后要常分析对比，常总结归纳。

# 第六章 建议和意见

（1）李老师和谢老师都很用心负责，问问题基本秒回，这令人很感动。

（2）健全评分机制，加强抄袭查处。有一个怪现象，直接从github或往届学长学姐拿来的代码通常比自己做的代码分数高，导致抄袭之风蔓延，更奇怪的是抄袭居然可以混过去，并且获取高分，希望能够在验收时加大监察力度，或者每年更换题目。

（3）时间太过紧张，建议错开同学们的考试周。

# 第七章 参考资料

[1]陈火旺.编译原理[M].国防工业出版社:北京,2010:123.

[2]Alfred V. Aho , Monica S.Lam , Ravi Sethi ,etc.编译原理(龙书）[M].机械工业出版社:北京,2008:120.

[3]David M. Beazley.PLY (Python Lex-Yacc)[EB/OL].http://www.dabeaz.com/ply/ply.html,2019-10-10.

[4]Tutorials.compiler\_design[EB/OL].https://www.tutorialspoint.com/compiler\_design/index.htm,2010-4-3.