

**编译原理课程设计报告**

学号：

姓名：

班级：

指导老师：

完成日期： 2021年6月15日

目录

[1 需求分析 3](#_Toc74590465)

[1.1 背景概述 3](#_Toc74590466)

[1.2 任务重述 3](#_Toc74590467)

[1.3 程序功能分析 4](#_Toc74590468)

[1.4 可行性分析 4](#_Toc74590469)

[1.4.1 理论可行性 4](#_Toc74590470)

[1.4.2 技术可行性 5](#_Toc74590471)

[1.5 任务输入及其范围 5](#_Toc74590472)

[1.6 程序的输出 6](#_Toc74590473)

[2 概要设计 7](#_Toc74590474)

[2.1 开发环境 7](#_Toc74590475)

[2.2 任务分解 7](#_Toc74590476)

[2.3 数据结构定义 8](#_Toc74590477)

[2.3.1 词法分析类（LexicalAnalyzer） 8](#_Toc74590478)

[2.3.2 文件读取类（CFG） 10](#_Toc74590479)

[2.3.3 语法分析器（Semantic） 10](#_Toc74590480)

[2.3.4 综合分析器（Semantic） 12](#_Toc74590481)

[2.3.5 目标代码生成模块 14](#_Toc74590482)

[2.3.6 图形化界面类 15](#_Toc74590483)

[2.4 主程序流程 16](#_Toc74590484)

[2.5 模块间的调用关系 16](#_Toc74590485)

[3 详细设计 17](#_Toc74590486)

[3.1 词法分析器 17](#_Toc74590487)

[3.1.1 源代码预处理 17](#_Toc74590488)

[3.1.2 Token识别 17](#_Toc74590489)

[3.1.3 流程图与状态转移图 18](#_Toc74590490)

[3.2 语法规则 19](#_Toc74590491)

[3.3 语法分析 22](#_Toc74590492)

[3.3.1 first集合 22](#_Toc74590493)

[3.3.2 LR0项目 24](#_Toc74590494)

[3.3.3 项目集闭包的构建 25](#_Toc74590495)

[3.3.4 构建 DFA 26](#_Toc74590496)

[3.3.5 LR1分析表 27](#_Toc74590497)

[3.3.6 token流的规约 28](#_Toc74590498)

[3.4 语义分析及中间代码生成 30](#_Toc74590499)

[3.4.1 符号表的查找与更新 31](#_Toc74590500)

[3.4.2 函数表的查找与更新 32](#_Toc74590501)

[3.4.3 一些属性 32](#_Toc74590502)

[3.4.4 语义分析 34](#_Toc74590503)

[3.5 目标代码生成 36](#_Toc74590504)

[3.5.1 寄存器分配与释放 36](#_Toc74590505)

[3.5.2 mips指令集特点 38](#_Toc74590506)

[3.5.3 生成目标代码 39](#_Toc74590507)

[3.6 图形化界面 44](#_Toc74590508)

[4 调试分析 45](#_Toc74590509)

[4.1 正确性测试 45](#_Toc74590510)

[4.2 语法错误测试 47](#_Toc74590511)

[4.3 静态语义错误测试 49](#_Toc74590512)

[4.3.1 变量重定义 49](#_Toc74590513)

[4.3.2 数组重定义 49](#_Toc74590514)

[4.3.3 使用未声明的变量 50](#_Toc74590515)

[4.3.4 使用未声明的数组 50](#_Toc74590516)

[4.3.5 使用未定义的函数 51](#_Toc74590517)

[4.3.6 函数形参和实参不匹配 52](#_Toc74590518)

[4.3.7 变量赋值时类型错误 53](#_Toc74590519)

[5 用户使用说明 53](#_Toc74590520)

[6 课程设计总结 56](#_Toc74590521)

[6.1 总结 56](#_Toc74590522)

[6.2 可改进的地方 56](#_Toc74590523)

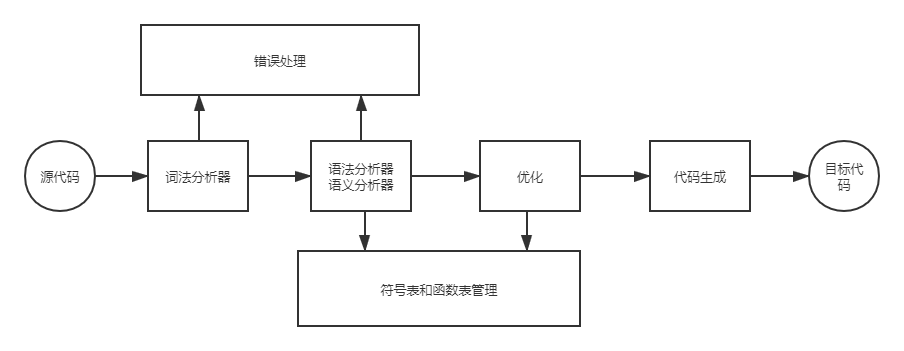
[6.3 感悟 58](#_Toc74590524)

# **1 需求分析**

## 1.1 背景概述

编译是指将一种语言向另外一种语言转化的过程，编译这一功能是由编译器来完成的。编译器是编译系统的核心，主要负责解析源程序的语义，生成目标机器代码。一般情况下，编译流程包含词法分析、语法分析、语义分析和中间代码生成、优化和目标代码生成六个阶段，其中语法分析和语义分析通常同时进行。在编译的过程中，对符号表和函数表的管理以及对出现的错误的处理也是编译器的一个重要部分。

下图是一个编译器的组成模块和编译流程：



而在本次的《编译原理课程设计》中，我们将着手完成一个完整的包含数组、包含有过程调用的类 C 语言的编译器的开发。

## 1.2 任务重述

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

（1）使用高级程序语言作为实现语言，实现一个类 C 语言的编译器。编码实现编译器的组成部分。

（2）要求的类 C 编译器是个一遍的编译程序，词法分析程序作为子程序，需要的时候被语法分析程序调用。

（3）使用语法制导的翻译技术，在语法分析的同时生成中间代码，并保存到文件中。

（4）要求输入类 C 语言源程序，输出中间代码表示的程序;

（5）要求输入类 C 语言源程序，输出目标代码（可汇编执行）的程序。

（6）实现过程、函数调用的代码编译。

（7）拓展类 C 语言文法，实现包含数组的中间代码以及目标代码生成。

## 1.3 程序功能分析

1. 完整的功能，实现一个较为完善的编译器：
   1. 能够手动输入或选择文件读入已知文法和代码；
   2. 进行词法和语法分析，输出First集、Follow集、Action表、Goto表、LR1分析表以及代码解析过程等分析结果，并将其保存为文件；
   3. 拥有符号表和函数表，能够进行中间代码的生成；
   4. 能够根据生成的中间代码，在寄存器选择算法的基础上实现目标代码的生成；
   5. 能够将各功能清晰明确地反映在图形界面中，方便用户使用。
2. 良好的性能，能够支持各种容量、各种边界情形的输入。
3. 能够进行清洗有效正确的错误提示，方便用户对代码查错。
4. 能够支持源代码中含有数组和过程调用。
5. 能够可靠的运行，较低的故障率和维护要求。
6. 用户使用简单、维护容易。
7. 未来升级扩展容易，有良好的后续技术支持。

## 1.4 可行性分析

### **1.4.1 理论可行性**

鉴于在大三上学期系统的学习过《编译原理》这门课程，包含过程调用的含有数组的类 C 语言编译器有理论上实现的可能。

在词法分析部分，使用超前搜索的方法扫描一遍源程序，即可得到“符号串”；在语法分析部分，根据给出的语法规则，构造first集合和follow集合，构造LR(0)项目集，根据相关算法形成DFA，最后得到LR(1)分析表；在语义分析部分，根据语义规则，在进行LR1分析的过程中，逐步完善符号表和函数表，最终得到中间代码；在目标代码生成部分，根据寄存器分配算法分配可以使用的寄存器，最终得到可以在Mars上模拟运行的mips汇编指令。

综上所述，该课程设计项目在理论上有坚实的基础，具有理论可行性。

### **1.4.2 技术可行性**

该项目准备使用python进行开发。

项目开发者在数据结构课程设计中(大二下)就系统地学习了python这一门高级程序设计语言，经过一年的python项目开发(包括通识选修课深度学习、专业选修课机器学习、数据库原理课程设计以及一些比赛项目)，熟练的掌握了python语言的基本使用和众多库的使用。

在python强大的库函数和简洁便利的数据类型使用下，编译器的开发工作将具有可能性，因此，该课程设计项目技术可行性。

## 1.5 任务输入及其范围

本次编译器的项目给出了如下的测试用例，该测试用例包含了过程调用，包含了数组这一数据类型，能够将此类 C 语言源程序编译，得到正确的、能够运行的mips汇编语言代码，是该项目的编译器需要实现的目标。

|  |
| --- |
| int program(int a,int b,int c)  {      int i;      int j;      i=0;      if(a>(b+c))      {          j=a+(b\*c+1);      }      else      {          j=a;      }      while(i<=100)      {          i=j\*2;          j=j+1;      }      return i;  }  int demo(int a)  {      a=a+2;      return a\*2;  }  void main(void)  {      int a[2][2];      a[0][0]=3;      a[0][1]=a[0][0]+1;      a[1][0]=a[0][0]+a[0][1];      a[1][1]=program(a[0][0],a[0][1],demo(a[1][0]));      return;  } |

## 1.6 程序的输出

程序的输出是一份mips汇编语言代码，这份汇编代码与上面那份类 C 语言程序代码相对应，是将类 C 语言代码编译得到的结果。这段汇编代码应该能够在Mars上模拟运行，并得到和源代码相匹配的结果。

|  |  |
| --- | --- |
| .data  .text      addiu $sp, $zero, 0x10018000      or $fp, $sp, $zero      jal  main      jal  programEnd  f1:      lw $7, 0($fp)      lw $8, 4($fp)      lw $9, 8($fp)      sub $fp,$fp,12      add $10,$zero,0      add $11,$8,$9      sgt $12,$7,$11      bgt $12,$zero,l1      j l2  l1:      mul $13,$8,$9      add $14,$13,1      add $15,$7,$14      add $16,$zero,$15      j l3  l2:      add $16,$zero,$7  l3:  l6:      add $a2,$zero,100      sgt $17,$10,$a2      xori $17,$17,1      bgt $17,$zero,l4      j l5  l4:      add $a2,$zero,2      mul $18,$16,$a2      add $10,$zero,$18      add $19,$16,1      add $16,$zero,$19      j l6 | l5:      add $v0,$zero,$10      jr $ra  f2:      lw $20, 0($fp)      sub $fp,$fp,4      add $21,$20,2      add $20,$zero,$21      add $a2,$zero,2      mul $22,$20,$a2      add $v0,$zero,$22      jr $ra  main:      add $23,$zero,3      add $24,$zero,4      add $25,$zero,2      sub $sp,$sp,4      sw $ra, 0($sp)      sub $sp,$sp,4      sw $ra, 0($sp)      add $fp,$fp,4      sw $25, 0($fp)      jal  f2      lw $ra, 0($sp)      add $sp,$sp,4      add $7,$zero,$v0      add $fp,$fp,12      sw $23, 0($fp)      sw $24, 4($fp)      sw $7, 8($fp)      jal  f1      lw $ra, 0($sp)      add $sp,$sp,4      add $8,$zero,$v0      add $23,$zero,$8      jr $ra  programEnd:      nop |

# **2 概要设计**

## 2.1 开发环境

本项目使用 python 3.7 进行开发，图形界面方面使用 pyqt 5.15.4 进行开发。

## 2.2 任务分解

根据需求分析，编译器应该分为如下模块：词法分析器、语法分析器、语义分析器、中间代码生成器、目标代码生成器和图形化界面模块。

但在实际的实现过程中，并不需要按照理论的模块划分来设计编译器，实际的任务分解如下：

* 词法分析是编译的第一步，它需要忽略源代码中的注释，采用超前搜索的方法对源文件进行扫描，得到Tokens流。因此，可以设置一个类来完成词法分析的工作。
* 所设计的编译器是一个一遍扫描的编译器，只需要对源程序进行一遍扫描，同时也只需对产生式进行一次读取。因此，可以设置一个类，以完成读取源程序生成Tokens流和读取、保存产生式的工作。
* 编译器想要进行编译的工作，核心是需要生成一张LR1分析表，而生成LR1分析表之前，需要利用产生式产生first集和follow集，需要利用产生式生成LR(0)项目，需要产生项目集规范族从而构造GOTO表和ACTION表(生成对应的DFA)，从而得到LR1分析表。因此，可以设置一个类来完成LR1分析表的生成。
* 由于分析过程使用的是LR1的分析方法，使用该方法，在进行语法分析的同时可以进行语义分析，语法分析的过程和语义分析的过程是一起进行的，而在语义分析时，需要针对每一条产生式指定语义规则，这其中需要使用到符号表与函数表。语义分析的结果是四元式，稍加改动就能生成中间代码因此，可以设置一个类来完成语法及语义分析以及中间代码的生成工作。
* 在得到中间代码之后，需要使用寄存器获得和寄存器释放算法来选择寄存器，从而完成由中间代码向目标代码的转化工作。因此，可以设置一个类来完成目标代码的生成工作。
* 编译器需要一个供用户使用的图形化界面，以便用户能够便捷地使用编译器。因此，可以设置一个类来实现人机交互的图形化界面。
* 最后需要用main来调用以上所有部分。

## 2.3 数据结构定义

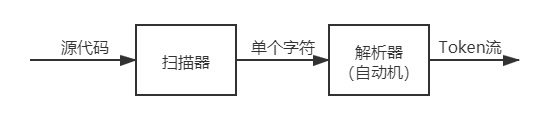
根据以上的任务划分，可以设计编译器的数据结构如下：

### **2.3.1 词法分析类（LexicalAnalyzer）**

（1）类的主要功能：

在编译器工作之前，需要将用高级语言书写的源程序作为输入。词法分析是编译器处理流程中的第一步，它顺序扫描源文件内的字符，通过与词法记号的有限自动机进行匹配，产生各式各样的词法记号。所谓词法记号（也称为终结符），反映在高级语言语法中就是对应的标识符、关键字、常量，以及运算符、逗号、分号等界符。

词法分析器的功能图和结构图如下：



词法分析器的输入是源代码文本文件内一长串的文本内容，为了从文本串中分析出每个词法记号，我们引入了有限自动机。

有限自动机能够解析并识别词法记号，比如识别标识符的有限自动机、识别常量的有限自动机等。有限自动机从开始状态启动，读入一个字符作为输入，并根据该字符选择进入下一个状态。继续读入新的字符，直到遇到结束状态为止，读入的所有字符序列便是有限自动机识别的词法记号。

本词法分析器可以识别的词法记号如下：

|  |  |
| --- | --- |
| 关键字 | int void if else while return |
| 界符 | ; |
| 分割符 | , |
| 注释号 | // /\* \*/ |
| 括号 | ( ) { } [ ] |
| 赋值号 | = |
| 运算符 | + - \* / % |
| 比较运算符 | == > < != >= <= |
| 数字 |  |
| 变量 |  |

（2）类的数据结构：

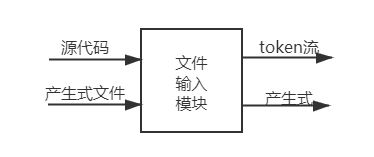
|  |  |
| --- | --- |
| 类成员 | 说明 |
| currentLine | 当前扫描到的行 |
| pInputStr | 当前扫描到的列 |
| reserved={} | 保留字 |
| type=[] | token 类型 |
| regexs=[] | 界符、操作符、变量和数字 |
| 类方法 | 说明 |
| removeAnnotation | 移除源代码中的注释 |
| scanLine | 扫描一行中的字符 |
| scan | 扫描源代码中的字符 |
| getTokensOfOneLine | 对一行识别 token 流 |
| genTokensFromInputBox | 对源文件识别 token 流 |

### **2.3.2 文件读取类（CFG）**

（1）类的主要功能：

在拥有词法分析器，能够识别 token 流的基础上，调用词法分析器，获得 token流。同时需要把准备好的语法产生式进行读入。文件读取类完成的就是该工作。文件读取类是编译器的输入模块，编译器所有需要的输入都通过该模块读入。

文件读入模块的功能图和结构图如下：



文件读取类得到了token流和产生式，这将使用到编译的后续过程中。

（2）类的数据结构：

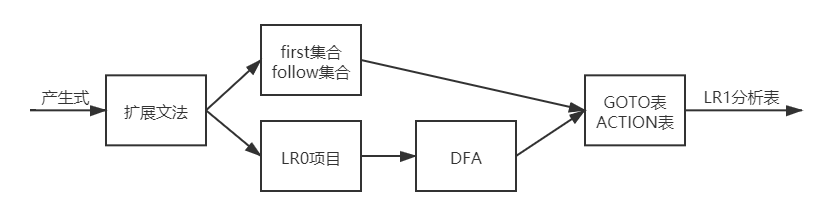
|  |  |
| --- | --- |
| 类成员 | 说明 |
| prods=[] | 产生式 |
| startProd | 广义文法起始符 |
| EndSymbol | 文法终结符 |
| TerminalSymbols=[] | 终结符表 |
| NonTerminalSymbols=[] | 非终极符表 |
| 类方法 | 说明 |
| readGrammerFile | 读取产生式 |
| readCodeFile | 读取 token 流 |

### **2.3.3 语法分析器（Semantic）**

（1）主要功能

该语法分析器模块与一般意义上的语法分析并不相同，一般的语法分析是指使用 token 流进行规约，得到文法开始符的过程，而这里的语法分析器主要是指得到语法分析的核心LR1分析表的过程。该语法分析器处理读入的产生式，首先将文法处理为扩展文法S’，然后利用扩展文法生成first集和follow集，同时生成LR0项目集，通过计算项目族闭包生成DFA，获得GOTO表和ACTION表，即得到LR1分析表，这是该语法分析器的核心流程。

语法分析器模块的功能图和结构图如下：



（2）数据结构

LR0项目集类（Item）

|  |  |
| --- | --- |
| 类成员 | 说明 |
| right=[] | LR0 项目的左侧 |
| left | LR0 项目的右侧 |
| dotPos | LR0 项目点的位置 |
| 类方法 | 说明 |
| nextItem | 返回下一个 LR0 项目 |

DFA 的一个状态（ItemSet）

|  |  |
| --- | --- |
| 类成员 | 说明 |
| name | 该状态的编号 |
| items | 该状态包含的项目 |

DFA 类（ItemSetSpecificationFamily）

|  |  |
| --- | --- |
| 类成员 | 说明 |
| itemSets=[] | DFA 不同的状态 |
| edges | 项目集之间的转移 |

语法分析类（SyntacticAnalyzer）

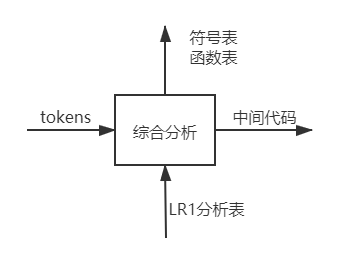
|  |  |
| --- | --- |
| 类成员 | 说明 |
| cfg | 文件读入类实例化 |
| family | LR0项目集 |
| ACTION | ACTION表 |
| GOTO | GOTO表 |
| FIRST | first集 |
| FOLLOW | follow集 |
| 类方法 | 说明 |
| calFirstSet | 生成 first 集 |
| calFollowSet | 生成 follow 集 |
| getDotItems | 生成 LR0 项目集 |
| getLR1Closure | 计算 LR1 闭包 |
| buildFamily | 构造项目集规范族 |
| GO | 生成状态转移集 |
| item2prodIdx | 返回产生式序号 |
| getTables | 获得 LR1 分析表 |
| isRecognizable | 对 token 流进行语法分析 |

### **2.3.4 综合分析器（Semantic）**

（1）主要功能

综合分析类的主要功能是利用文件读入类得到的 token 流和语法分析类得到的 LR1 分析表进行编译的核心工作——语义分析。进行一遍扫描的语义分析时，需要生成语法树，语法树的节点附有不同的属性。在语义分析的过程中，需要不断使用和更新变量表和函数表，变量表和函数表也需要定义相应的类。在语义规则的指导下，依据 LR1 分析表，对 token 流进行规约，即可得到中间代码。

综合分析器模块的功能图和结构图如下：



（2）数据结构

语法树结点类（Node）

|  |  |
| --- | --- |
| 类成员 | 说明 |
| place | 语句块入口中间变量 |
| code | 中间代码 |
| stack | 临时栈 |
| name | 语句块标识符 |
| arrname | 数组名字 |
| type | 结点数据类型 |
| data | 结点携带的数据 |
| dims | 数组结点维度 |
| position | 符号元素所在的数组位置 |
| begin | 循环入口 |
| end | 循环出口 |
| true | 条件为真时的出口 |
| false | 条件为假时的出口 |

符号表类（Symbol）

|  |  |
| --- | --- |
| 类成员 | 说明 |
| name | 符号的标识符 |
| type | 类型 |
| size | 占用字节数 |
| offset | 内存偏移量 |
| place | 对应的中间变量 |
| function | 所在函数 |
| dims | 数组的维度 |

函数表类（FunctionSymbol）

|  |  |
| --- | --- |
| 类成员 | 说明 |
| name | 符号的标识符 |
| type | 类型 |
| label | 入口标签 |
| params=[] | 形参列表 |

语义分析类（SemanticAnalyser）

|  |  |
| --- | --- |
| 类成员 | 说明 |
| sStack=[] | 语义分析栈 |
| symbolTable=[] | 符号表 |
| funcTable=[] | 函数表 |
| curTempId | 中间变量序号名 |
| curLabelId | label入口序号 |
| curFuncId | 函数入口序号 |
| curOffset | 当前偏移量 |
| curFuncSymbol | 当前函数 |
| middleCode=[] | 生成的中间代码 |
| 类方法 | 说明 |
| semanticAnalyze | 根据语义规则进行规约 |
| findSymbol | 在符号表里查找符号 |
| updateSymbolTable | 更新符号表 |
| findFuncSymbolByName | 在函数表里查找函数 |
| updateFuncTable | 更新函数表 |
| getNewTemp | 获得新的结点 |
| getNewLabel | 获得新的标签 |
| getNewFuncLabel | 获得新的函数标签 |
| calExpression | 生成代码emit |
| saveMidCodeToFile | 将中间代码保存到文件中 |

### **2.3.5 目标代码生成模块**

（1）主要功能

在获得中间代码后，可以将中间代码转化为mips指令，其中寄存器的分配是根据寄存器分配算法进行的。这样，即可得到目标代码。

目标代码生成模块的功能图和结构图如下：



（2）数据结构

目标代码生成器类（ObjectCodeGenerator）

|  |  |
| --- | --- |
| 类成员 | 说明 |
| mipsCode | 目标代码 |
| regTable | 寄存器状态表 |
| varStatus | 变量状态表 |
| DATA\_SEGMENT | 数据段偏移量 |
| STACK\_OFFSET | 堆栈偏移量 |
| 类方法 | 说明 |
| getRegister | 申请一个寄存器 |
| freeRegister | 释放一个寄存器 |
| genMips | 生成mips目标代码 |

### **2.3.6 图形化界面类**

（1）主要功能

图形界面是编译器的输入输出接口。在图形界面中，可以对待编译的源代码进行编辑。成功编译代码以后，可以查看词法分析、语法分析的规约过程、中间代码、符号表函数表以及生成的mips汇编指令。此外，若编译出错，图形界面也能够将出错信息提供给用户以便进行错误更正。

（2）数据结构

|  |  |
| --- | --- |
| 类成员 | 说明 |
| …… | pyqt5各种组件的定义 |
| 类方法 | 说明 |
| compile | 进行源代码的编译 |

## 2.4 主程序流程

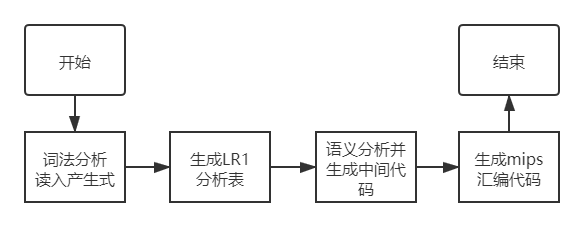
整个主程序的流程如下：

首先，程序的输入为源代码文件和产生式文件，由文件读入模块完成对两者的读入，其中对源代码文件的读入需要调用词法分析模块。经历这一步后，得到了 token 流和产生式。

其次，在读入产生式的基础上，调用语法分析模块，依次生成 first 集， follow 集， LR0项目，再计算项目蔟闭包生成 DFA ，从而获得 LR1 分析表。

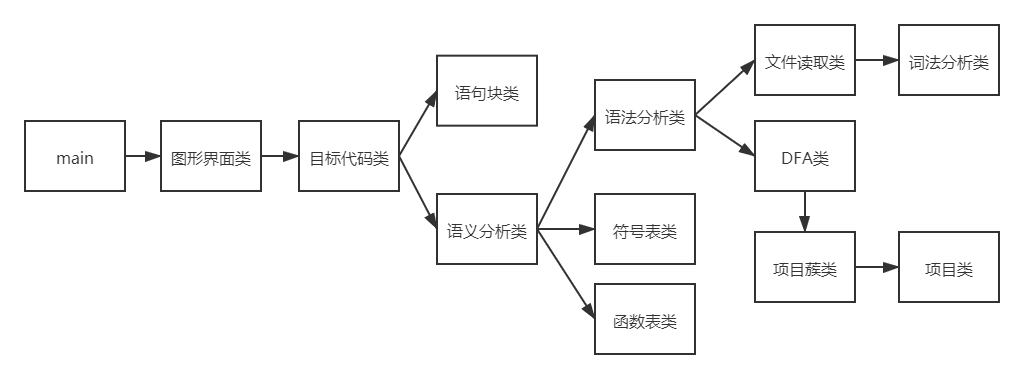
然后，在拥有 LR1 分析表后，根据语义规则，对 token 流进行规约，生成中间代码，并在此过程中完善符号表和函数表。

最后，根据寄存器分配规则分配寄存器，在中间代码的基础上生成mips汇编代码。



## 2.5 模块间的调用关系

各个类之间的调用关系如下：



# **3 详细设计**

## 3.1 词法分析器

### **3.1.1 源代码预处理**

词法分析器的实现主要依赖于 python 中提供的正则表达式库 re 来匹配 token 字符。

对于源代码文件，第一步是要去除源代码文件中的注释。主要通过调用 findall() 函数来实现。具体来说，对于正则表达式“//.\*?\n”调用 findall() 来找到所有匹配的注释行，对于正则表达式“/\\*.\*?\\*/”调用 findall() 来找到所有匹配的段注释，用 replace() 将其替换为空白，这样就完成对于源代码的预处理工作。

以下是对源代码进行预处理的函数：

|  |
| --- |
| def removeAnnotation(self, codes):      annotation = re.findall('//.\*?\n', codes, flags=re.DOTALL)      if len(annotation) > 0:          codes = codes.replace(annotation[0], "")      annotation = re.findall('/\\*.\*?\\*/', codes, flags=re.DOTALL)      if len(annotation) > 0:          codes = codes.replace(annotation[0], "")      ret = codes.strip()      return ret |

### **3.1.2 Token识别**

对于预处理后的源代码中的语法单词的识别需要使用线性有界自动机。同样，这里可以使用 python 的 re 库里的 findall() 函数去识别所有符合的语法单词。

首先需要定义可能出现的语法单词的正则表达式，定义四种可能出现在源代码中的语法单词的正则表达式，分别是界符、操作符、标识符和整数。

|  |
| --- |
| self.regexs = [      '\{|\}|\[|\]|\(|\)|,|;'          # 界符      , '\+|-|\\*|/|==|!=|>=|<=|>|<|='  # 操作符      , '[a-zA-Z][a-zA-Z0-9]\*'         # 标识符      , '\d+'                          # 整数  ] |

其中，标识符包括变量和保留字。

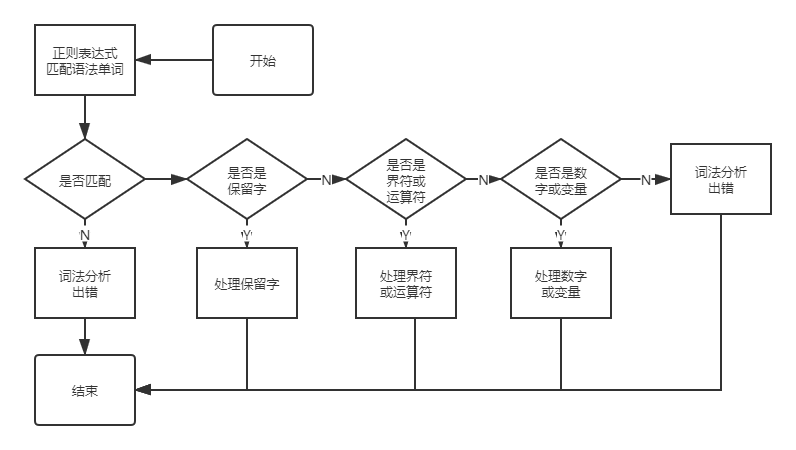
将通过正则表达式匹配读取到的语法单词根据类别进行分类，填写该语法单词的类别、是否是终结符、该语法单词的值、该语法单词在源代码文件中的位置。

|  |
| --- |
| if result == "":      break  before = result  result = self.scan(result)  if result['regex']:      token = {}      token['class'] = "T"      token['row'] = self.currentLine      token['colum'] = origin.find(before) + 1      token['name'] = self.type[regexs.index(result['regex'])].upper()      token['data'] = result['data']      token['type'] = token['name']      if result['data'] in self.reserved:          token['name'] = self.reserved[result['data']].lower()          token['type'] = token['name']      if token['name'] == "operator".upper() or token['name'] == "seperator".upper():          token['type'] = token['data']      if token['name'] == "INT":          token['data'] = token['data']      tokens.append(token) |

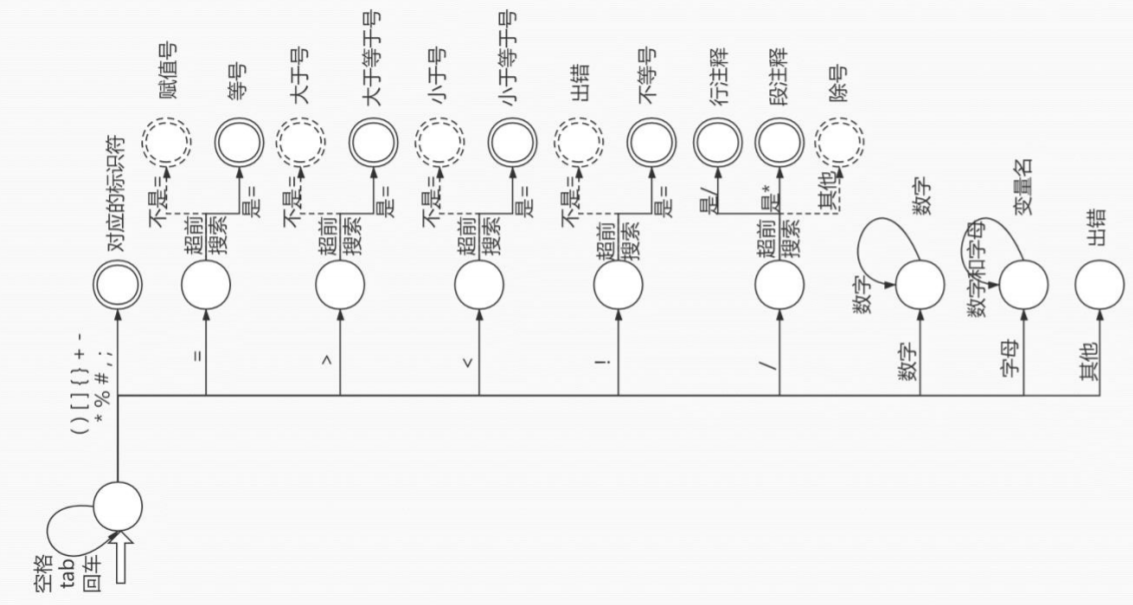
最后将识别的语法单词逐一加入到列表 tokens 中，即可完成词法分析的工作。

### **3.1.3 流程图与状态转移图**

词法分析器的详细流程图如下：



识别语法单词的状态转移图如下：



## 3.2 语法规则

含有数组的语法规则如下：

|  |
| --- |
| Program ::= <声明串>  <声明串> ::=<声明>{<声明>}  <声明> ::=int <ID> <声明类型> | void <ID> <函数声明>  <声明类型>::=<变量声明> | <函数声明> | <数组声明>  <变量声明> ::= ;  <函数声明> ::= ‘(’<形参>‘)’ <语句块>  <数组声明> ::= ‘[’ (数字) + ‘]’{‘[’ (数字) + ‘]’}  <形参>::= <参数列表> | void  <参数列表> ::= <参数> {, <参数>}  <参数> ::= int <ID>  <语句块> ::= ‘{’<内部声明> <语句串>‘}’  <内部声明> ::= 空 | <内部变量声明>;{<内部变量声明>;}  <内部变量声明>::=int <ID>  <语句串> ::= <语句>{ <语句> }  <语句> ::= <if 语句> |< while 语句> | <return 语句> | <赋值语句>  <赋值语句> ::= <ID>‘=’<表达式>; | <数组>‘=’<表达式>;  <return 语句> ::= return [ <表达式> ] ;  <while 语句> ::= while ‘( ’<表达式> ‘)’ <语句块>  <if 语句> ::= if ‘(‘<表达式>‘)’ <语句块> [ else <语句块> ]  <表达式>::=<加法表达式>{ relop <加法表达式> }  <加法表达式> ::= <项> {+ <项> | -<项>}  <项> ::= <因子> {\* <因子> | /<因子>}  <因子> ::=num | ‘(’<表达式>‘)’ |<ID> FTYPE |<数组>  FTYPE ::= <call> | 空  <call> ::=‘(’<实参> ‘) ’  <数组>::=<ID> ‘[’<表达式>‘]’ | <数组> ‘[’<表达式>‘]’  <实参> ::=<实参列表> | 空  <实参列表> ::= <表达式>{, <表达式>}  <ID>::=字母(字母|数字)\*  （注：[ ]中的项表示可选, { }表示可以重复任意次） |

根据含有数组的语法规则，设计语法产生式如下：

|  |
| --- |
| program\_ --> program  program --> declarationChain  declarationChain --> $  declarationChain --> declaration declarationChain  typeSpecifier --> int  typeSpecifier --> void  declaration --> typeSpecifier IDENTIFIER ;  declaration --> completeFunction  declaration --> typeSpecifier IDENTIFIER arrayDeclaration ;  arrayDeclaration --> [ INT ]  arrayDeclaration --> [ INT ] arrayDeclaration  completeFunction --> declareFunction block  declareFunction --> typeSpecifier IDENTIFIER ( formalParaList )  formalParaList --> $  formalParaList --> para  formalParaList --> para , formalParaList  formalParaList --> void  para --> typeSpecifier IDENTIFIER  block --> { statementChain }  statementChain --> $  statementChain --> statement statementChain  statement --> declaration  statement --> ifStatement  statement --> iterStatement  statement --> returnStatement  statement --> assignStatement  array --> IDENTIFIER [ expression ]  array --> array [ expression ]  assignStatement --> IDENTIFIER = expression ;  assignStatement --> array = expression ;  returnStatement --> return expression ;  returnStatement --> return ;  iterStatement --> while ( expression ) block  ifStatement --> if ( expression ) block  ifStatement --> if ( expression ) block else block  expression --> primaryExpression  expression --> primaryExpression operator expression  primaryExpression --> INT  primaryExpression --> ( expression )  primaryExpression --> IDENTIFIER ( actualParaList )  primaryExpression --> IDENTIFIER  primaryExpression --> array  operator --> +  operator --> -  operator --> \*  operator --> /  operator --> <  operator --> <=  operator --> >=  operator --> >  operator --> ==  operator --> !=  actualParaList --> $  actualParaList --> expression  actualParaList --> expression , actualParaList |

而文件读入类就是将上述产生式读入，并储存在相应的数据结构中。

## 3.3 语法分析

语法分析是编译过程的核心部分。它的任务是在词法分析识别出单词符号串的基础上，分析并判定程序的语法结构是否符合语法规则。语言的语法结构是用上下文无关文法描述的。因此，语法分析器的工作本质上就是按文法的产生式，识别输入字符串是否为一个句子。这里所说的输入字符串是指由单词符号(文法的终结符)组成的有限序列。要识别符号串是否是文法的一个句子，就是要建立一棵与输入串相匹配的语法分析树。

LR 方法的基本思想是：在规范归约的过程中，一方面记住已经移进和归约出的整个字符串，即记住“历史”，另一方面根据所用的产生式推测未来可能碰到的输入符号，即对未来进行“展望”。对于一串呈现在分析栈的顶端的符号串，使用所记载的“历史”和“展望”以及“现实”的输入符号等三方面的材料来确定栈顶的符号串是否构成相对产生式的某一句柄。

语法分析完成的工作较为零散与繁杂。

### **3.3.1 first集合**

LR1分析方法需要所有产生式左侧句子的 first 集合，首先需要得到所有非终结符的 first 集合。计算单个NT的 first 集合的算法如下：

使用如下规则，直至每一个非终结符的 FIRST 集合不在增大为止：

1）若 X 属于 VT，则 FIRST（X）={X}

2）若 X 属于 VN，且有产生式 X→a…，则把 a 加入 FIRST；若 S→e 也是产生式，则把e 也加入到 FIRST(X)中

3）若 X→Y……是一个产生式且 Y 属于 VN，则把 FIRST(Y)中所有的非空元素都加入到FIRST（X）中，若 X→Y1,Y2…Yk 是一个产生式，Y1,Y2,…Yi-1 都是非终结符，而且对于任何满足 1<=j<=i-1，first(Yj)都含有 e，则把 FIRST（Yi）中所有的非 e 元素都加入到 FIRST（X）中，特别的，若所有 FIRST(Yj)对于 j=1，2，…,k 都含有 e，则将 e 也加入到 FIRST(X)中。 基于以上算法，先实现计算单个NT的 first 集合的函数如下：

|  |
| --- |
| def calNTFirstSet(self, symbol):      eps = {'class': 'T', 'name': '', 'type': self.Epsilon}      hasEpsAllBefore = -1      prods = [prod for prod in self.prods if prod.left == symbol]      if len(prods) == 0:          return      is\_add = 1      while (is\_add):          is\_add = 0          for prod in prods:              hasEpsAllBefore = 0              for right in prod.right:                  if right['class'] == 'T' or (right['type'] == self.Epsilon and len(prod.right) == 1):                  if right['type'] not in self.firstSet[symbol]:                          self.firstSet[symbol].append(right['type'])                          is\_add = 1                      break                  if len(self.firstSet[right['type']]) == 0:                      if right['type'] != symbol:                          self.calNTFirstSet(right['type'])                  if self.Epsilon in self.firstSet[right['type']]:                      hasEpsAllBefore = 1                  for f in self.firstSet[right['type']]:                      if f != self.Epsilon and f not in self.firstSet[symbol]:                          self.firstSet[symbol].append(f)                          is\_add = 1              if hasEpsAllBefore == 1:                  if self.Epsilon not in self.firstSet[symbol]:                      self.firstSet[symbol].append(self.Epsilon)                      is\_add = 1      return |

在计算出单个NT的 first 集合之后，可以利用这些结果去计算所有T和NT的 first 集合。

|  |
| --- |
| def calFirstSet(self):      for symbol in self.TerminalSymbols:          self.firstSet[symbol] = [symbol]      for symbol in self.NonTerminalSymbols:          self.firstSet[symbol] = []      for symbol in self.NonTerminalSymbols:          self.calNTFirstSet(symbol)      return |

计算出所有T与NT的 first 集合后，可以利用这个结果去计算字符串的 first 集合，这是我们最终想要的。

|  |
| --- |
| def getStringFirstSet(self, symbols):      rst = []      hasEpsAllBefore = 0      for s in symbols:          tempSet = [i for i in self.firstSet[s]]          if self.Epsilon in tempSet:              if hasEpsAllBefore == 0:                  hasEpsAllBefore = 1              rst.extend([i for i in tempSet if i != self.Epsilon])          else:              hasEpsAllBefore = -1              rst.extend(tempSet)              break      if hasEpsAllBefore == 1:          rst.append(self.Epsilon)      return rst |

### **3.3.2 LR0项目**

在进行 DFA 的构建之前，首先需要对产生式加点，使之转化为 LR0 项目集。对于产生式加点，实际上就是用一个变量指出点在产生式右侧的位置，这一部分实现起来很容易。

|  |
| --- |
| def getDotItems(self):      for prod in self.prods:          if len(prod.right) == 1 and prod.right[0]['type']==self.Epsilon:              self.items.append(Item(prod.left, prod.right, 0, ['#']))              continue          for i in range(len(prod.right) + 1):              self.items.append(Item(prod.left, prod.right, i, ['#']))      return |

### **3.3.3 项目集闭包的构建**

项目集闭包的构建是生成DFA的关键，计算闭包的算法如下：

1）I 的任何项目都属于 CLOSURE(I)

2）如果A→α⋅Bβ属于 CLOSURE(I)，那么B→⋅γ也属于 CLOSURE(I)

重复上面步骤到 CLOSURE(I)不再增大为止。

具体的实现思路如下：

循环所有的产生式，取出产生式，对产生式进行判断，要是产生式的点在最后面，则直接查看下一个产生式。当点后面还有终结符或者非终结符的时候，用循环和栈来模拟递归，求解闭包，先把当前产生式点后面的字符入栈，循环直到栈为空，每次循环栈顶出栈，当栈顶符号是非终结符的时候，寻找所有产生式A →⋅ B形式的产生式，加入集合，并把新产生式的栈顶符号入栈，重复上述过程。

计算项目蔟闭包的函数如下：

|  |
| --- |
| def getLR1Closure(self, I):      rst = []      rst.extend(I)      rstStr = [item.toString() for item in rst]      while(True):          isAddItem = 0          for item in rst:              right = item.right              for i in range(len(right) + 1):                  if item.dotPos == len(right):                      continue                  if right[item.dotPos]['class'] == 'T':                      continue                  tempRst = self.extendItem(item)                  for i in tempRst:                      tempStr = i.toString()                      if tempStr not in rstStr:                          rstStr.append(tempStr)                          rst.append(i)                          isAddItem = 1          if isAddItem == 0:              break      return rst |

### **3.3.4 构建 DFA**

在计算项目出项目集闭包的过程中，会生成状态转移函数GO，该函数指出DFA的一个状态在遇到某个符号时该向何种状态跳转。

状态转移函数GO的实现如下：

|  |
| --- |
| def GO(self, I, X):      J = []      if len(I.items) == 0 or X == self.Epsilon:          return J      for item in I.items:          if item.dotPos == len(item.right):              continue          if len(item.right) == 1 and item.right[0] == self.Epsilon:              continue          if item.right[item.dotPos]['type'] == X:              temp = item.nextItem()              if temp != None:                  J.append(temp)      return self.getLR1Closure(J) |

在能够计算项目集闭包和实现了状态转移GO函数的情况下，从起始状态的闭包出发，不断调用GO函数生成新的项目族并计算闭包，重复这个过程直到不再有新的项目族生成，这样即完成了产生式所代表的 DFA 的构建，DFA 的构建函数如下：

|  |
| --- |
| def buildDFA(self):      iS = self.itemSets      startI = []      startI.append(self.itemPool[0])      iS.append(ItemSet('s0', self.getLR1Closure([startI[0]] + self.extendItem(startI[0]))))      setCnt = 1      setStrings = {}      setStrings['s0'] = iS[0].toString()      edgeStrings = []      while(True):          isBigger = 0          for I in iS:              for X in self.symbols:                  rstGO = self.GO(I, X)                  if len(rstGO) == 0:                      continue                  tempItemSet = ItemSet('s' + str(setCnt), rstGO)                  if tempItemSet.toString() in setStrings.values():                      tempItemSet.name = list(setStrings.keys())[list(setStrings.values()).index(tempItemSet.toString())]                  else:                      setStrings[tempItemSet.name] = tempItemSet.toString()                      iS.append(tempItemSet)                      isBigger = 1                      setCnt = setCnt + 1                  tempEdge = {'start' : I.name, 'symbol' : X, 'end' : tempItemSet.name}                  tempEdgeStr = tempEdge['start'] + '->' + tempEdge['symbol'] + '->' + tempEdge['end']                  if tempEdgeStr not in edgeStrings:                      self.edges.append(tempEdge)                      edgeStrings.append(tempEdgeStr)                      isBigger = 1          if isBigger == 0:              break      return |

### **3.3.5 LR1分析表**

LR1分析表的实质是根据生成的 DFA 建立归约所需要的 ACTION 表和 GOTO 表，它描述了在当前状态下，遇到下一个输入字符，将要进行的操作（移进、规约、结束、报错），以及下一步的状态。

由 DFA 出发，构造LR1分析表的算法如下：

1）若项目属于且,为终结符，则置为“把状态j和符号a移进栈”，即于

2）若项目属于,则置为“用产生式进行规约”，即

3）若项目属于, 则置为“接收”，即acc

4）若，则置

5）其他不能运用上面规则的地方均填入“出错”，即error。

想要运用以上算法实现LR1分析表的构建，首先需要知道某条产生式在产生式集合中的序号，这一功能的实现如下：

|  |
| --- |
| def item2prodIdx(self, item):      tempStr = item.left + '->@'      for right in item.right:          tempStr += (right['type'] + ' ')      tempStr += '# '      return self.prodStrs.index(tempStr) |

在此基础上，可以实现LR1分析表的构建，如下：

|  |
| --- |
| def getLR1Tables(self):      self.rst = []      for e in self.edges:          if e['symbol'] in self.TerminalSymbols:              self.M[e['start']][e['symbol']] = 'shift ' + e['end']          if e['symbol'] in self.NonTerminalSymbols:              self.M[e['start']][e['symbol']] = 'goto ' + e['end']      for I in self.itemSets:          for item in I.items:              if item.dotPos == len(item.right):                  if item.left == self.OriginStartSymbol and item.terms[0] == '#':                      if self.M[I.name][item.terms[0]] != ' ':                          print('LR(1)分析表有多重入口！')                      self.M[I.name][item.terms[0]] = 'acc'                  else:                      if self.M[I.name][item.terms[0]] != ' ':                          print('LR(1)分析表有多重入口！')                      self.M[I.name][item.terms[0]] = 'reduce ' + str(self.item2prodIdx(item))                  continue              if len(item.right) == 1 and item.right[0]['type'] == '$':                  if item.left == self.OriginStartSymbol and item.terms[0] == '#':                      if self.M[I.name][item.terms[0]] != ' ':                          print('LR(1)分析表有多重入口！')                      self.M[I.name][item.terms[0]] = 'acc'                  else:                      if self.M[I.name][item.terms[0]] != ' ':                          print('LR(1)分析表有多重入口！')                      self.M[I.name][item.terms[0]] = 'reduce ' + str(self.item2prodIdx(item))                  continue      return |

### **3.3.6 token流的规约**

在有了 goto 表和 action 表的基础上，对于一个状态和得到的词法分析结果，我们只需要查表，得到需要做的行为，然后做出对应的处理。

具体流程如下：

从词法分析的结果中读一个符号，查表，得到下一次操作的需要做的事情:

如果是需要移进，把当前符号和要转移到状态压栈;如果是接受，则结束规约过程;如果是规约，得到需要用哪条产生式规约，然后根据产生式右部符号的个数，依次出栈，然后针对当前符号和栈顶状态，查找 GOTO 表，进行转移。如果查表，发现当前状态，读入当前符号，表中没有对应行为，则报错。

当不断进行规约最终得到文法的起始字符时，表明规约成功，语法分析成功。实现这一过程的函数如下：

|  |
| --- |
| def isRecognizable(self, originCode):      inputStr = []      # 输入串      inputStr += self.lex.getTokensOfOneLine(originCode)      sys.stdout.flush()      stateStack = []    # 栈内状态序列      shiftStr = []      # 移进规约串      self.parseRst = [] # 记录步骤      # 开始      wallSymbol = {'class': 'T', 'type': '#'}      shiftStr.append(wallSymbol)      stateStack.append('s0')      X = inputStr[0]      while (True):          if len(inputStr) <= 2:              tmpInputStr = self.lex.getTokensOfOneLine(originCode)              if len(tmpInputStr) == 0:                  inputStr.append(wallSymbol)              else:                  inputStr += tmpInputStr          self.parseRst.append({'stateStack': copy.deepcopy(stateStack),                                  'shiftStr': copy.deepcopy(shiftStr),                                  'inputStr': copy.deepcopy(inputStr)})          act = self.M[stateStack[-1]][X['type']].split(' ')[0]          target = self.M[stateStack[-1]][X['type']].split(' ')[1] if len(self.M[stateStack[-1]][X['type']].split(' ')) == 2 else None          # 移进操作          if act == 'shift':              stateStack.append(target)              inputStr.pop(0)              shiftStr.append(X)              X = inputStr[0]          # 转移操作          elif act == 'goto':              stateStack.append(target)              shiftStr.append(X)              X = inputStr[0]          # 规约操作          elif act == 'reduce':              prodIdx = int(target)              prod = self.prods[prodIdx]              self.semantic.semanticAnalyze(prod, shiftStr)              if False == self.semantic.semanticRst:                  return False              rightLen = len(prod.right)              stateLen = len(stateStack)              if rightLen == 1 and prod.right[0]['type'] == '$':# 空串                  dst = self.M[stateStack[-1]][prod.left].split(' ')[1]                  stateStack.append(dst)                  shiftStr.append({'class': 'NT', 'type': prod.left})                  X = inputStr[0]              else:# 不是空串                  stateStack = stateStack[0 : stateLen - rightLen]                  shiftStr = shiftStr[0 : stateLen - rightLen]                  X = {'class': 'NT', 'type': prod.left}          # 接受操作          elif act == 'acc':              self.semantic.semanticAnalyze(self.prods[1], shiftStr)              return True          # 出错          else:              self.syntacticRst = False              sys.stdout.flush()              self.syntacticErrMsg = "语法分析错误：" + str(X['row']) + "行" + str(X['colum']) + "列"          return False |

## 3.4 语义分析及中间代码生成

在实际应用中，比较流行的语义描述和语义处理方法主要是使用属性文法和进行语法制导翻译方法。该部分语义分析和中间代码生成的实现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的语法制导翻译的原理。

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。

实际操作中，可用一遍扫描实现属性文法的语义规则来进行计算。具体实现起来就是在语法分析的同时进行语义规则的计算，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。

语义分析得到的中间代码采用三地址语句(四元式)的方法来记录。三地址代码由X:=Y OP Z的一般形式的语句构成，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。

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。这四个域分别是 op, arg1, arg2 ,res。其中， 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 res 域，而 := 就是运算符。

### **3.4.1 符号表的查找与更新**

在语义分析的过程中，一旦遇到一个符号，就要在符号表中查找该符号，若该符号首次出现，则要将其登记在符号表中。

|  |
| --- |
| # 在符号表里查找符号  def findSymbol(self, name, function):      for s in self.symbolTable:          if s.name == name and s.function == function:              return s      return None  # 更新符号表  def updateSymbolTable(self, symbol):      for s in self.symbolTable:          if s.name == symbol.name and s.function == symbol.function:              self.symbolTable.remove(s)              break      self.symbolTable.append(symbol)      return |

### **3.4.2 函数表的查找与更新**

在语义分析的过程中，一旦遇到一个函数，就要在符号表中查找该函数，若该函数首次出现，则要将其登记在函数表中。

|  |
| --- |
| # 在函数表里查找函数  def findFuncSymbolByName(self, name):      for f in self.funcTable:          if f.name == name:              return f      return None  # 更新函数表  def updateFuncTable(self, functionSymbol):      for f in self.funcTable:          if f.name == functionSymbol.name:              self.funcTable.remove(f)              break      self.funcTable.append(functionSymbol)      return |

### **3.4.3 一些属性**

在进行语义分析之前，需要先实现几个语义分析的常用的工具函数，这些工具函数分别是生成新的标签和生成中间代码：

|  |
| --- |
| def getNewTemp(self):      self.curTempId += 1      return "t" + str(self.curTempId)  def getNewLabel(self):      self.curLabelId += 1      return 'l' + str(self.curLabelId)  def getNewFuncLabel(self):      self.curFuncId += 1      return 'f' + str(self.curFuncId)  def emit(self, n):      if len(n.stack) == 1:          n = copy.deepcopy(n.stack[0])          self.prtNodeCode(n)          n.stack = []          return True      n.code = []      self.prtNodeStack(n)      sys.stdout.flush()      nLeft = n.stack.pop(0)      while len(n.stack) > 0:          nOp = n.stack.pop(0)          nRight = n.stack.pop(0)          if nLeft.place == None:              arg1 = nLeft.data          else:              arg1 = nLeft.place          if nRight.place == None:              arg2 = nRight.data          else:              arg2 = nRight.place          if len(nLeft.code) > 0:              for code in nLeft.code:                  n.code.append(code)          if len(nRight.code) > 0:              for code in nRight.code:                  n.code.append(code)          nRst = Node()          nRst.name = None          nRst.place = self.getNewTemp()          nRst.type = nRight.type          code = (nOp.type, arg1, arg2, nRst.place)          n.code.append(code)          nLeft = nRst          n.type = nRight.type      n.place = n.code[-1][3]      return True |

在实现了这些语义分析常用的工具函数之后，对每一条产生式在规约的时候使用对应的语义规则进行规约，对符号的属性进行赋值或者使用emit生成中间代码，这就是语义分析。语义分析本质上就是语法分析的规约这一步，只不过它是在属性文法上的规约，即规约过程中需要处理文法的属性。

### **3.4.4 语义分析**

语义分析需要对每一条产生式进行特定的语义处理，这一部分较为冗杂，因此仅展示和数组有关的语义处理过程。

在数组的定义上，参照产生式进行语义处理，如下：

|  |
| --- |
| # arrayDeclaration -> [ num ] | [ num ] arrayDeclaration  if nt == 'arrayDeclaration':      if len(r) == 3:          n = Node()          n.name = nt          n.type = 'int array'          n.dims = [ int(shiftStr[-2]['data'])]          self.sStack.append(n)      elif len(r) == 4:          n = self.sStack.pop(-1)          n.name = nt          n.type = 'int array'          n.dims.insert(0, int(shiftStr[-3]['data']))          self.sStack.append(n)  # array -> id [ expression ] | array [ expression ]  if nt == 'array':      expression\_n = self.sStack.pop(-1)      self.calExpression(expression\_n)      array\_n = copy.deepcopy(self.sStack[-1])      # -> array [ expression ]      if shiftStr[-4]['type'] == 'array':          self.sStack.pop(-1)          expression\_n.name = 'array'          expression\_n.arrname = array\_n.arrname          expression\_n.position = copy.deepcopy(array\_n.position)          expression\_n.position.append(expression\_n.place if expression\_n.place else expression\_n.data)          expression\_n.place = array\_n.place          expression\_n.type = 'array'          self.sStack.append(expression\_n)      # -> id [ expression ]      else:          expression\_n.name = 'array'          expression\_n.arrname = shiftStr[-4]['data']          s = shiftStr[-4]['data']          nTmp = self.findSymbol(s, self.curFuncSymbol.label)          if nTmp == None:              print('使用未定义的数组变量!')              self.semanticRst = False              self.semanticErrMsg = "未定义的数组变量：" + shiftStr[-4]['data']              return          expression\_n.position.append(expression\_n.place if expression\_n.place else expression\_n.data)          expression\_n.place = s          expression\_n.type = 'array'          self.sStack.append(expression\_n) |

对于数组的调用，将其和一般的普通变量视为相同的，仅仅在符号表中检查其是否出现过，而数组值本身通过偏移地址去确认，并不对数组的调用模式和数组是否越界进行检查，如下：

|  |
| --- |
| # -> typeSpecifier id arrayDeclaration ;  if len(r) == 4:      array\_n = self.sStack.pop(-1)      n = self.sStack.pop(-1)      n.name = nt      defType = array\_n.type      defName = shiftStr[-3]['data']      s = self.findSymbol(defName, self.curFuncSymbol.label)      if s != None:          print("数组变量重定义！")          self.semanticRst = False          self.semanticErrMsg = "数组变量重定义。"          return      else:          s = Symbol()  ......  s = self.findSymbol(array\_name, self.curFuncSymbol.label)  if s == None:      print("使用未定义的数组变量！")      self.semanticRst = False      self.semanticErrMsg = "使用未定义的数组变量！" + array\_name      return  sys.stdout.flush()  ......  # 偏移地址临时变量  t\_offset = self.getNewTemp()  if len(array\_n.position) == 1: # 一维数组      n.code.append((':=', str(array\_n.position[0]), '-', t\_offset))  elif len(array\_n.position) == 2: # 二维数组      n.code.append(('\*', str(array\_n.position[0]), str(s.dims[1]), t\_offset))      n.code.append(('+', t\_offset, str(array\_n.position[1]), t\_offset))  ......  # -> array  else:      n = self.sStack.pop(-1)      nTmp = self.findSymbol(n.arrname, self.curFuncSymbol.label)      if nTmp == None:          print('使用未定义数组变量!')          self.semanticRst = False          self.semanticErrMsg = "未定义的数组变量！" + n.arrname          return |

## 3.5 目标代码生成

由于不需要进行对于中间代码的块内和块间优化，因此，目标代码的生成工作就是利用已经得到的中间代码，根据寄存器分配算法分配寄存器，逐步转换中间代码为目标代码的过程。

### **3.5.1 寄存器分配与释放**

在寄存器的分配过程中，有以下原则：

1）尽可能用B独占的寄存器

2）尽可能用空闲寄存器

3）抢占用非空闲寄存器

寄存器分配：GETREG(i: A:=B op C) 返回一个用来存放A的值的寄存器，它的分配算法如下：

1）如果B的现行值在某个寄存器中，RVALUE[]中只包含B，此外，或者B与A是同一个标识符，或者B的现行值在执行四元式A:=B op C之后不会再引用，则选取为所需要的寄存器R，并转4；

2）如果有尚未分配的寄存器，则从中选取一个为所需要的寄存器R，并转4；

3）从已分配的寄存器中选取一个为所需要的寄存器R。最好使得满足以下条件：占用的变量的值也同时存放在该变量的贮存单元中，或者在基本块中要在最远的将来才会引用到或不会引用到。

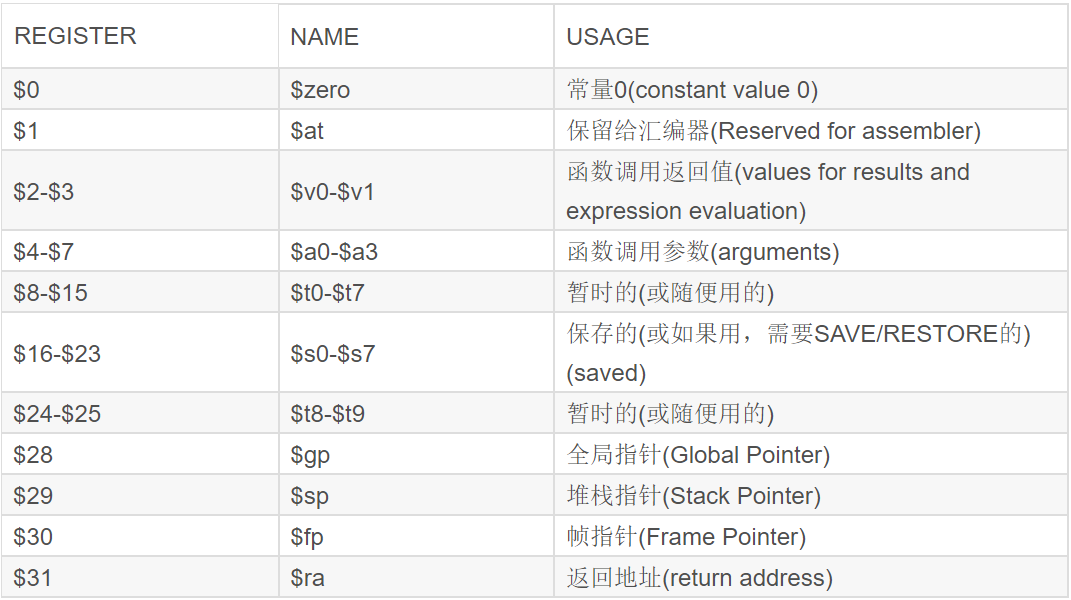
4）为中的变量生成存数指令：如果V的地址描述数组AVALUE[V]说V还保存在R之外的其他地方，则不需要生成存数指令；如果V是A，且不是B或C，则不需要生成存数指令；如果V不会在此之后被使用，则不需要生成存数指令；否则，生成目标代码

根据以上算法，寄存器的分配与释放函数如下：

|  |
| --- |
| def getRegister(self, identifier, codes):      if identifier[0] != 't':          return identifier      if identifier in self.varStatus and self.varStatus[identifier] == 'reg':          for key in self.regTable:              if self.regTable[key] == identifier:                  return key      while True:          for key in self.regTable:              if self.regTable[key] == '':                  self.regTable[key] = identifier                  self.varStatus[identifier] = 'reg'                  return key          self.freeRegister(codes)  def freeRegister(self, codes):      # 提取出使用了 reg 的变量      varRegUsed = list(filter(lambda x: x != '', self.regTable.values()))      # 统计这些变量后续的使用情况      varUsageCnts = {}      for code in codes:          # print(code)          for item in code:              # print(item)              tmp = str(item)              if tmp[0] == 't':  # 是个变量                  if tmp in varRegUsed:                      if tmp in varUsageCnts:                          varUsageCnts[tmp] += 1                      else:                          varUsageCnts[tmp] = 1      sys.stdout.flush()      flag = False      # 找出之后不会使用的变量所在的寄存器      for var in varRegUsed:          if var not in varUsageCnts:              for reg in self.regTable:                  if self.regTable[reg] == var:                      self.regTable[reg] = ''                      self.varStatus[var] = 'memory'                      flag = True      if flag:          return      # 释放最少使用的寄存器，      sorted(varUsageCnts.items(), key=lambda x: x[1])      varFreed = list(varUsageCnts.keys())[0]      for reg in self.regTable:          if self.regTable[reg] == varFreed:              for item in self.symbolTable:                  if item.place == varFreed:  # t1, t2, ...                      self.mipsCode.append('addi $at, $zero, 0x{}'.format(self.DATA\_SEGMENT))                      self.mipsCode.append('sw {}, {}($at)'.format(reg, item.offset))                      self.regTable[reg] = ''                      self.varStatus[varFreed] = 'memory'                      return      return |

### **3.5.2 mips指令集特点**

mips指令集有32个寄存器，但各个寄存器并不是平等通用的，一般来说有如下使用规范：



因此寄存器的分配最好是能够遵守mips的规范的，尤其是其中的$0寄存器不应该被改动，使用数值0时可以引用$0；$1，$28，$29，$30，$31号寄存器尽量不要修改；临时变量最好放在$8-$15里面；过程调用的参数传递最好放在$4-$7里面等。

### **3.5.3 生成目标代码**

目标代码的生成算法如下：

对每个四元式:，依次执行：

1）以四元式: 为参数，调用函数过程，返回一个寄存器R，用作存放A的寄存器。

2）利用AVALUE[B]和AVALUE[C]，确定B和C现行值的存放位置和。如果其现行值在寄存器中，则把寄存器取作和

3）如果，则生成目标代码：

否则生成目标代码

如果为R，则删除AVALUE[B]或AVALUE[C]中的R。

4）令AVALUE[A]={R}, RVALUE[R]={A}。

5）若B或C的现行值在基本块中不再被引用，也不是基本块出口之后的活跃变量，且其现行值在某寄存器中，则删除RVALUE[]中的B或C以及AVALUE[B]或AVALUE[C] 中的 ，使得该寄存器不再为B或C占用。

根据以上算法，实现生成目标代码的函数如下：

|  |
| --- |
| def genMips(self):      mc = self.mipsCode      dc = self.middleCode      dc.insert(0, ('call', '\_', '\_', 'programEnd'))      dc.insert(0, ('call', '\_', '\_', 'main'))      mc.append('.data')      for s in self.symbolTable:          if s.type == 'int array':              size = 4              for dim in s.dims:                  size \*= int(dim)              mc.append('    ' + s.place + ': .space ' + str(size))      mc.append('')      mc.append('.text') # 代码段      mc.append('addiu $sp, $zero, 0x{}'.format(self.DATA\_SEGMENT + self.STACK\_OFFSET))      mc.append('or $fp, $sp, $zero')      while dc:          code = dc.pop(0)          tmp = []          for item in code:              if item == 'v0':                  tmp.append('$v0')              else:                  tmp.append(item)          code = tmp          if code[0] == ':=':              src = self.getRegister(code[1], dc)              dst = self.getRegister(code[3], dc)              mc.append('    add {},$zero,{}'.format(dst, src))          elif code[0] == '[]=': # []=, t21, \_, t17[t22]              src = self.getRegister(code[1], dc)              base = code[3][ : code[3].index('[')]              offset = code[3][code[3].index('[') + 1 : -1]              dst\_offset = self.getRegister(offset, dc)              mc.append('la $v1,{}'.format(base))              mc.append('mul {},{},4'.format(dst\_offset,dst\_offset))              mc.append('addu {},{},$v1'.format(dst\_offset, dst\_offset))              mc.append('sw {},'.format(src)+'0({})'.format(dst\_offset))          elif code[0] == '=[]': # =[], t17[t23], -, t24              dst = self.getRegister(code[3], dc)              base = code[1][ : code[1].index('[')]              offset = code[1][code[1].index('[') + 1 : -1]              src\_offset = self.getRegister(offset, dc)              mc.append('la $v1,{}'.format(base))              mc.append('mul {},{},4'.format(src\_offset,src\_offset))              mc.append('addu {},{},$v1'.format(src\_offset, src\_offset))              mc.append('lw {},'.format(dst)+'0({})'.format(src\_offset))          # function or label          elif code[1] == ':':              if code[0] in self.funcNameTable or code[0][0] == 'f':                  mc.append('')  # empty line              mc.append('{}:'.format(code[0]))          # 跳转到函数的label处          elif code[0] == 'call':              mc.append('    jal  {}'.format(code[3]))          # actual arg of a function call          elif code[0] == 'push':              if code[3] == 'ra':  # return addr                  mc.append('sw $ra, {}($fp)'.format(code[2]))              else:                  register = self.getRegister(code[3], dc)                  if str(register)[0] != '$':                      mc.append("add $a0, $zero, {}".format(register))                      register = '$a0'                  mc.append('sw {}, {}($fp)'.format(register, code[2]))          # get args inside the function          elif code[0] == 'pop':              if code[3] == 'ra':                  mc.append('lw $ra, {}($fp)'.format(code[2]))              else:                  register = self.getRegister(code[3], dc)                  mc.append('lw {}, {}($fp)'.format(register, code[2]))          # store var from reg to memory          elif code[0] == 'store':              if code[3] == 'ra':                  mc.append('sw $ra, {}($sp)'.format(code[2]))              else:                  register = self.getRegister(code[3], dc)                  if str(register)[0] != '$':                      mc.append("    add $a0,$zero,{}".format(register))                      register = '$a0'                  mc.append('sw {}, {}($sp)'.format(register, code[2]))          # load var from memory to reg          elif code[0] == 'load':              if code[3] == 'ra':                  mc.append('lw $ra, {}($sp)'.format(code[2]))              else:                  register = self.getRegister(code[3], dc)                  mc.append('lw {}, {}($sp)'.format(register, code[2]))          # jump instruction          elif code[0] == 'j':              mc.append('    j {}'.format(code[3]))          elif code[0] == 'j>':              arg1 = self.getRegister(code[1], dc)              mc.append('    bgt {},$zero,{}'.format(arg1, code[3]))          elif code[0] == 'return':              mc.append('    jr $ra')          # algorithm operations, has 3 oprand          else:              if code[0] == '+':                  if code[1] == 'fp':                      mc.append("    add $fp,$fp,{}".format(code[2]))                  elif code[1] == 'sp':                      mc.append("    add $sp,$sp,{}".format(code[2]))                  else:                      arg1 = self.getRegister(code[1], dc)                      arg2 = self.getRegister(code[2], dc)                      arg3 = self.getRegister(code[3], dc)                      if str(arg1)[0] != '$':                          mc.append("    add $a1,$zero,{}".format(arg1))                          arg1 = '$a1'                      mc.append("add {},{},{}".format(arg3, arg1, arg2))              elif code[0] == '-':                  if code[1] == 'fp':                      mc.append("    sub $fp,$fp,{}".format(code[2]))                  elif code[1] == 'sp':                      mc.append("    sub $sp,$sp,{}".format(code[2]))                  else:                      arg1 = self.getRegister(code[1], dc)                      arg2 = self.getRegister(code[2], dc)                      arg3 = self.getRegister(code[3], dc)                      if str(arg1)[0] != '$':                          mc.append("    add $a1,$zero,{}".format(arg1))                          arg1 = '$a1'                      if str(arg2)[0] != '$':                          mc.append("    add $a2,$zero,{}".format(arg2))                          arg2 = '$a2'                      mc.append("sub {},{},{}".format(arg3, arg1, arg2))              elif code[0] == '\*':                  arg1 = self.getRegister(code[1], dc)                  arg2 = self.getRegister(code[2], dc)                  arg3 = self.getRegister(code[3], dc)                  if str(arg1)[0] != '$':                      mc.append("    add $a1,$zero,{}".format(arg1))                      arg1 = '$a1'                  if str(arg2)[0] != '$':                      mc.append("    add $a2,$zero,{}".format(arg2))                      arg2 = '$a2'                  mc.append("    mul {},{},{}".format(arg3, arg1, arg2))              elif code[0] == '/':                  arg1 = self.getRegister(code[1], dc)                  arg2 = self.getRegister(code[2], dc)                  arg3 = self.getRegister(code[3], dc)                  if str(arg1)[0] != '$':                      mc.append("    add $a1,$zero,{}".format(arg1))                      arg1 = '$a1'                  if str(arg2)[0] != '$':                      mc.append("    add $a2,$zero,{}".format(arg2))                      arg2 = '$a2'                  mc.append("    div {},{},{}".format(arg3, arg1, arg2))              elif code[0] == '%':                  arg1 = self.getRegister(code[1], dc)                  arg2 = self.getRegister(code[2], dc)                  arg3 = self.getRegister(code[3], dc)                  if str(arg1)[0] != '$':                      mc.append("    add $a1,$zero,{}".format(arg1))                      arg1 = '$a1'                  if str(arg2)[0] != '$':                      mc.append("    add $a2,$zero,{}".format(arg2))                      arg2 = '$a2'                  mc.append("    div {},{},{}".format(arg3, arg1, arg2))                  mc.append("    mfhi {}".format(arg3))              elif code[0] == '<':                  arg1 = self.getRegister(code[1], dc)                  arg2 = self.getRegister(code[2], dc)                  arg3 = self.getRegister(code[3], dc)                  if str(arg1)[0] != '$':                      mc.append("    add $a1,$zero,{}".format(arg1))                      arg1 = '$a1'                  if str(arg2)[0] != '$':                      mc.append("    add $a2,$zero,{}".format(arg2))                      arg2 = '$a2'                  mc.append("    slt {},{},{}".format(arg3, arg1, arg2))              elif code[0] == '>':                  arg1 = self.getRegister(code[1], dc)                  arg2 = self.getRegister(code[2], dc)                  arg3 = self.getRegister(code[3], dc)                  if str(arg1)[0] != '$':                      mc.append("    add $a1,$zero,{}".format(arg1))                      arg1 = '$a1'                  if str(arg2)[0] != '$':                      mc.append("    add $a2,$zero,{}".format(arg2))                      arg2 = '$a2'                  mc.append("    sgt {},{},{}".format(arg3, arg1, arg2))              elif code[0] == '!=':                  arg1 = self.getRegister(code[1], dc)                  arg2 = self.getRegister(code[2], dc)                  arg3 = self.getRegister(code[3], dc)                  if str(arg1)[0] != '$':                      mc.append("    add $a1,$zero,{}".format(arg1))                      arg1 = '$a1'                  if str(arg2)[0] != '$':                      mc.append("    add $a2,$zero,{}".format(arg2))                      arg2 = '$a2'                  mc.append("    sne {},{},{}".format(arg3, arg1, arg2))              elif code[0] == '==':                  arg1 = self.getRegister(code[1], dc)                  arg2 = self.getRegister(code[2], dc)                  arg3 = self.getRegister(code[3], dc)                  if str(arg1)[0] != '$':                      mc.append("    add $a1,$zero,{}".format(arg1))                      arg1 = '$a1'                  if str(arg2)[0] != '$':                      mc.append("    add $a2,$zero,{}".format(arg2))                      arg2 = '$a2'                  mc.append("    seq {},{},{}".format(arg3, arg1, arg2))              elif code[0] == '<=':                  arg1 = self.getRegister(code[1], dc)                  arg2 = self.getRegister(code[2], dc)                  arg3 = self.getRegister(code[3], dc)                  if str(arg1)[0] != '$':                      mc.append("    add $a1,$zero,{}".format(arg1))                      arg1 = '$a1'                  if str(arg2)[0] != '$':                      mc.append("    add $a2,$zero,{}".format(arg2))                      arg2 = '$a2'                  mc.append("    sgt {},{},{}".format(arg3, arg1, arg2))                  mc.append("    xori {},{},1".format(arg3, arg3))              elif code[0] == '>=':                  arg1 = self.getRegister(code[1], dc)                  arg2 = self.getRegister(code[2], dc)                  arg3 = self.getRegister(code[3], dc)                  if str(arg1)[0] != '$':                      mc.append("    add $a1,$zero,{}".format(arg1))                      arg1 = '$a1'                  if str(arg2)[0] != '$':                      mc.append("    add $a2,$zero,{}".format(arg2))                      arg2 = '$a2'                  mc.append("    slt {},{},{}".format(arg3, arg1, arg2))                  mc.append("    xori {},{},1".format(arg3, arg3))      mc.append('')      mc.append('programEnd:')      mc.append('    nop')      self.prtMips()      sys.stdout.flush()      return |

## 3.6 图形化界面

图形化界面采用扁平化的设计，使用pyqt5里的栈式布局做出了可以在多个界面之间相互切换的效果。图形化界面中只有两个函数和大量QT组件的定义。其中之一的函数是完成栈式布局不同界面之间的切换的，另一个函数是完成编译的。完成编译的函数调用了前述的所有类完成编译的操作，得到编译的各种结果（token流、语法规约过程、中间代码、符号表、函数表、目标代码），并且将这些结果放在界面的组件上面进行展示，同时，还将最后生成 mips 目标代码保存到文件中。

# **4 调试分析**

调试分析这一部分主要包含两个内容：

一是对于正确的源代码程序，测试编译器是否能够得到正确的汇编代码；

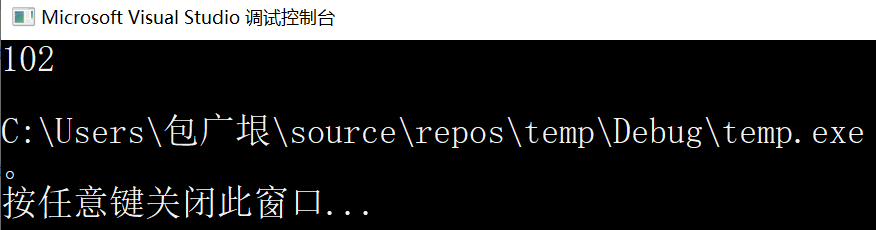
二是对于存在简单语法或者静态语义错误的源代码，测试编译器是否能够发现这些错误。

## 4.1 正确性测试

将带有数组的测试用例稍加修改，得到可以在一般C++编译器下编译的代码如下：

|  |
| --- |
| #include<iostream>  using namespace std;  int program(int a, int b, int c)  {  int i = 0, j;  if (a > (b + c))  j = a + (b\*c + 1);  else  j = a;  while (i <= 100) {  i = j \* 2;  j = j + 1;  }  return i;  }  int demo(int a)  {  a = a + 2;  return a \* 2;  }  int main()  {  int a[2][2];  a[0][0] = 3;  a[0][1] = a[0][0] + 1;  a[1][0] = a[0][0] + a[0][1];  a[1][1] = program(a[0][0], a[0][1], demo(a[1][0]));  cout << a[1][1] << endl;  return 0;  } |

在上述代码中，将最后结果a[1][1]输出，运行结果如下：

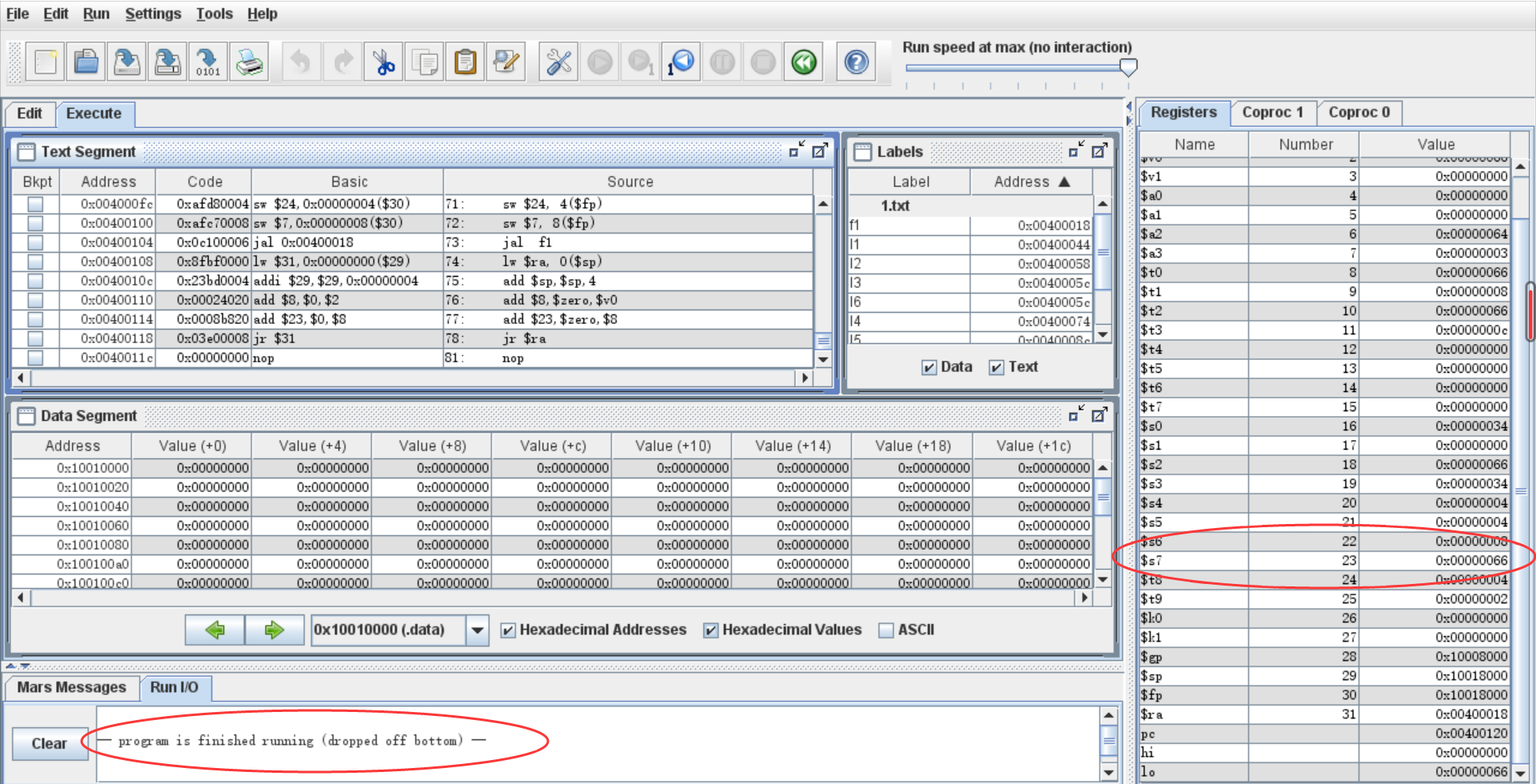


可以看到，正确执行类 C 源代码，则 a[1][1] 的值应该为 102 。

用自制编译器编译所得的汇编代码如下：

|  |  |
| --- | --- |
| .data  .text      addiu $sp, $zero, 0x10018000      or $fp, $sp, $zero      jal  main      jal  programEnd  f1:      lw $7, 0($fp)      lw $8, 4($fp)      lw $9, 8($fp)      sub $fp,$fp,12      add $10,$zero,0      add $11,$8,$9      sgt $12,$7,$11      bgt $12,$zero,l1      j l2  l1:      mul $13,$8,$9      add $14,$13,1      add $15,$7,$14      add $16,$zero,$15      j l3  l2:      add $16,$zero,$7  l3:  l6:      add $a2,$zero,100      sgt $17,$10,$a2      xori $17,$17,1      bgt $17,$zero,l4      j l5  l4:      add $a2,$zero,2      mul $18,$16,$a2      add $10,$zero,$18      add $19,$16,1      add $16,$zero,$19      j l6 | l5:      add $v0,$zero,$10      jr $ra  f2:      lw $20, 0($fp)      sub $fp,$fp,4      add $21,$20,2      add $20,$zero,$21      add $a2,$zero,2      mul $22,$20,$a2      add $v0,$zero,$22      jr $ra  main:      add $23,$zero,3      add $24,$zero,4      add $25,$zero,2      sub $sp,$sp,4      sw $ra, 0($sp)      sub $sp,$sp,4      sw $ra, 0($sp)      add $fp,$fp,4      sw $25, 0($fp)      jal  f2      lw $ra, 0($sp)      add $sp,$sp,4      add $7,$zero,$v0      add $fp,$fp,12      sw $23, 0($fp)      sw $24, 4($fp)      sw $7, 8($fp)      jal  f1      lw $ra, 0($sp)      add $sp,$sp,4      add $8,$zero,$v0      add $23,$zero,$8      jr $ra  programEnd:      nop |

从黄色高亮部分可以看到，汇编代码将数组的值 a[1][1] 最后放入第 23 号寄存器，现将汇编代码放入Mars中执行，结果如下：



可以看到，mips汇编代码可以正确执行，并且 $23 存放了预期结果(十六进制的66即十进制的102)。



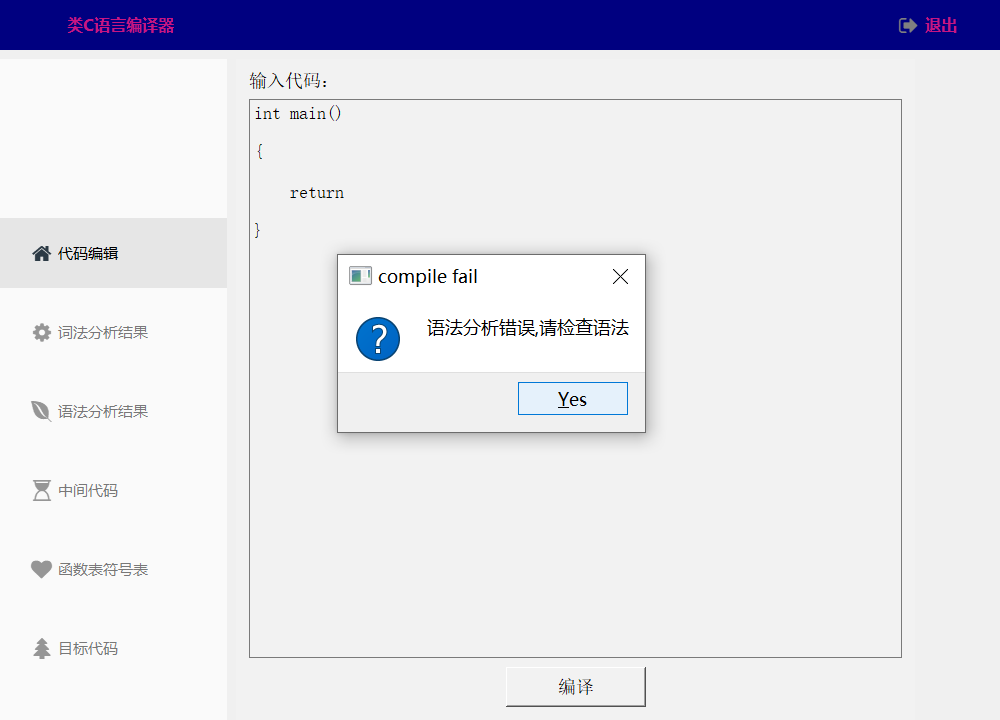


## 4.2 语法错误测试

（1）测试用例1

|  |
| --- |
| int main()  {  return  } |

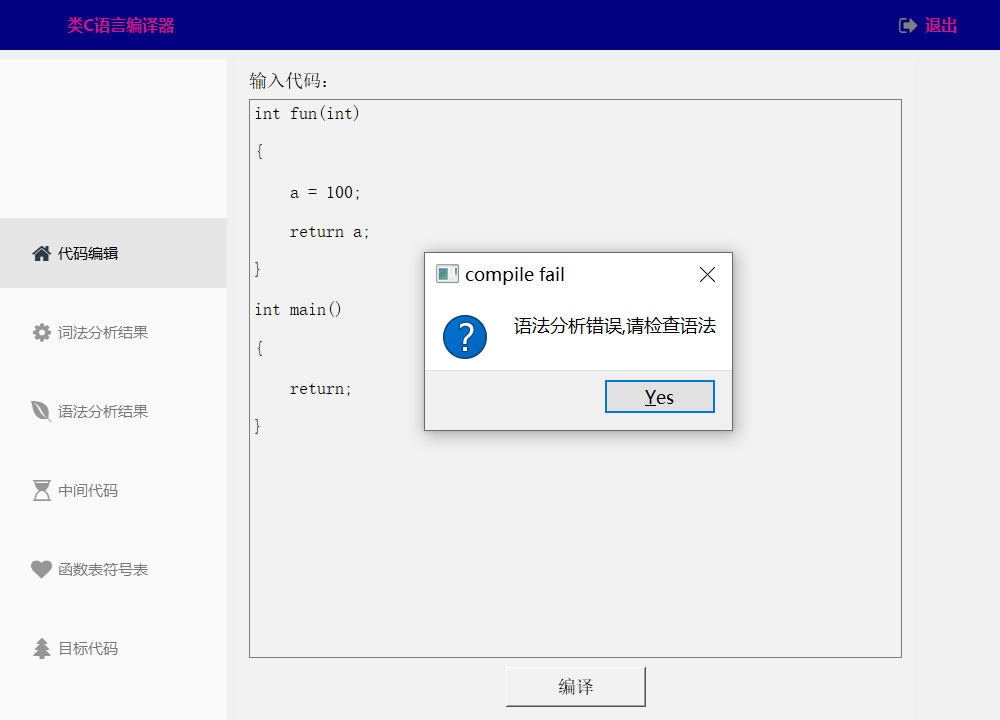
该测试用例存在语法错误，return后缺少分号，程序运行结果如下：



（2）测试用例2

|  |
| --- |
| int fun(int) {  a = 100;  return a;  }  int main() {  return;  } |

该测试用例存在语法错误，未指明函数的形参是什么，程序运行结果如下：



可以用来进行测试的语法错误很多，这里只列举极少数部分。

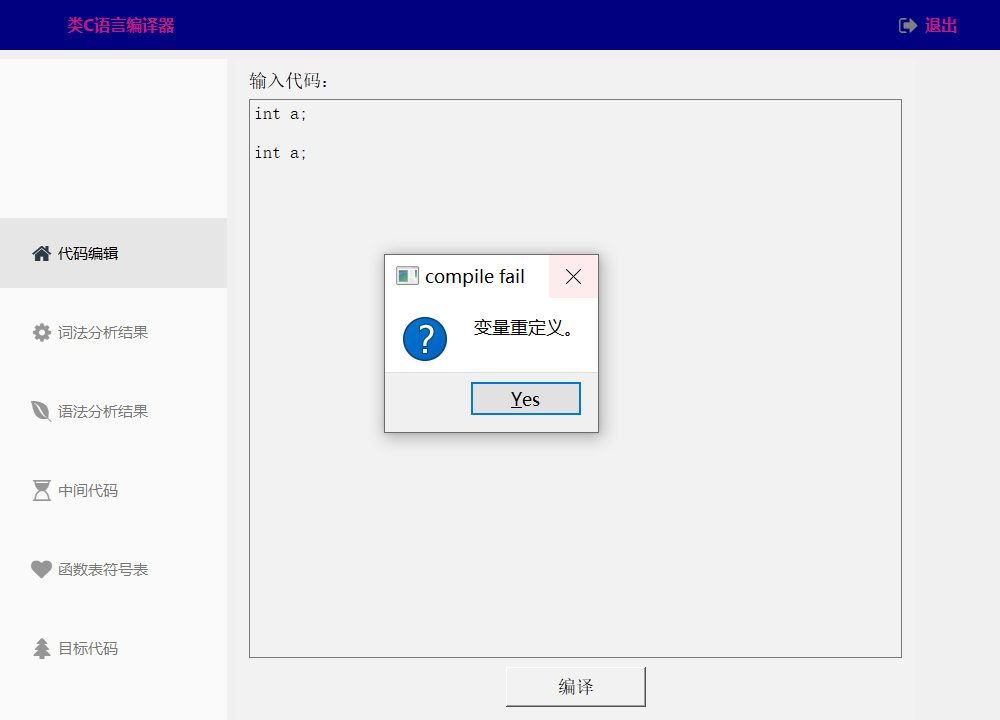
## 4.3 静态语义错误测试

### **4.3.1 变量重定义**

测试用例如下：

|  |
| --- |
| int a; int a; |

测试结果如下：

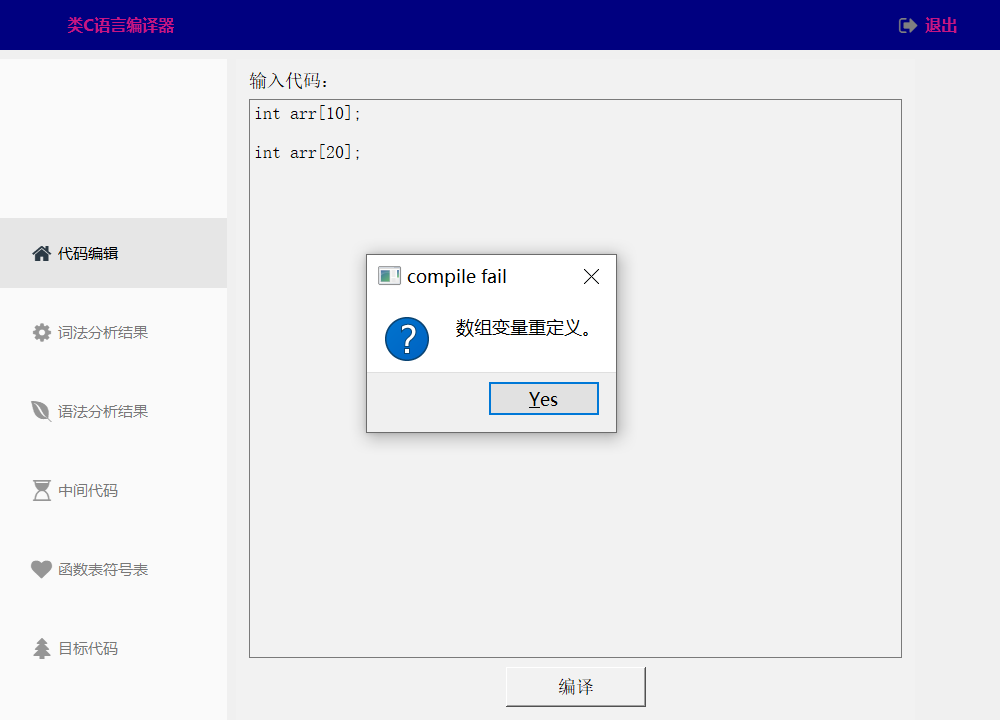


### **4.3.2 数组重定义**

测试用例如下：

|  |
| --- |
| int arr[10]; int arr[20]; |

测试结果如下：

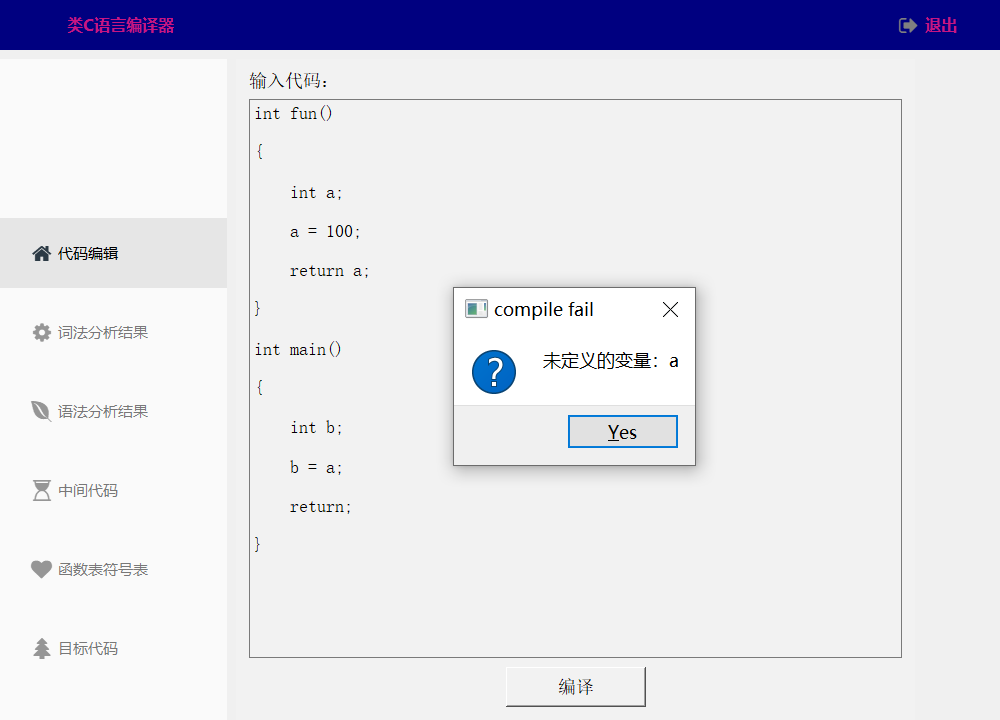


### **4.3.3 使用未声明的变量**

测试用例如下：

|  |
| --- |
| int fun()  {  int a;  a = 100;  return a;  }  int main()  {  int b;  b = a;  return;  } |

测试结果如下：

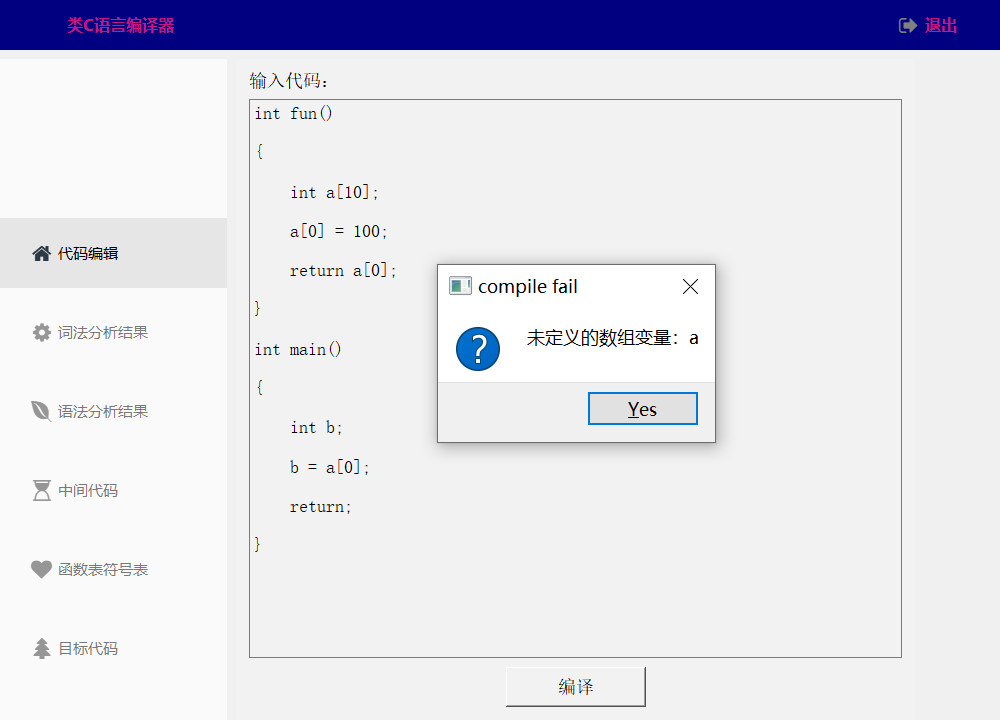


### **4.3.4 使用未声明的数组**

测试用例如下：

|  |
| --- |
| int fun()  {  int a[10];  a[0] = 100;  return a[0];  }  int main()  {  int b;  b = a[0];  return;  } |

测试结果如下：

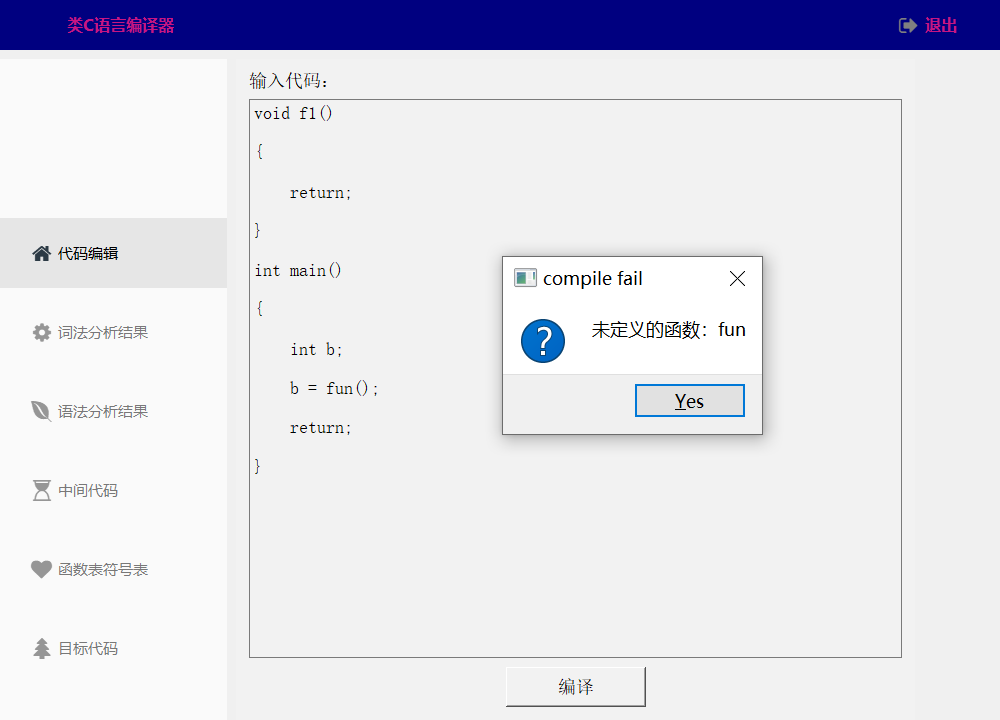


### **4.3.5 使用未定义的函数**

测试用例如下：

|  |
| --- |
| void f1()  {  return;  }  int main()  {  int b;  b = fun();  return;  } |

测试结果如下：

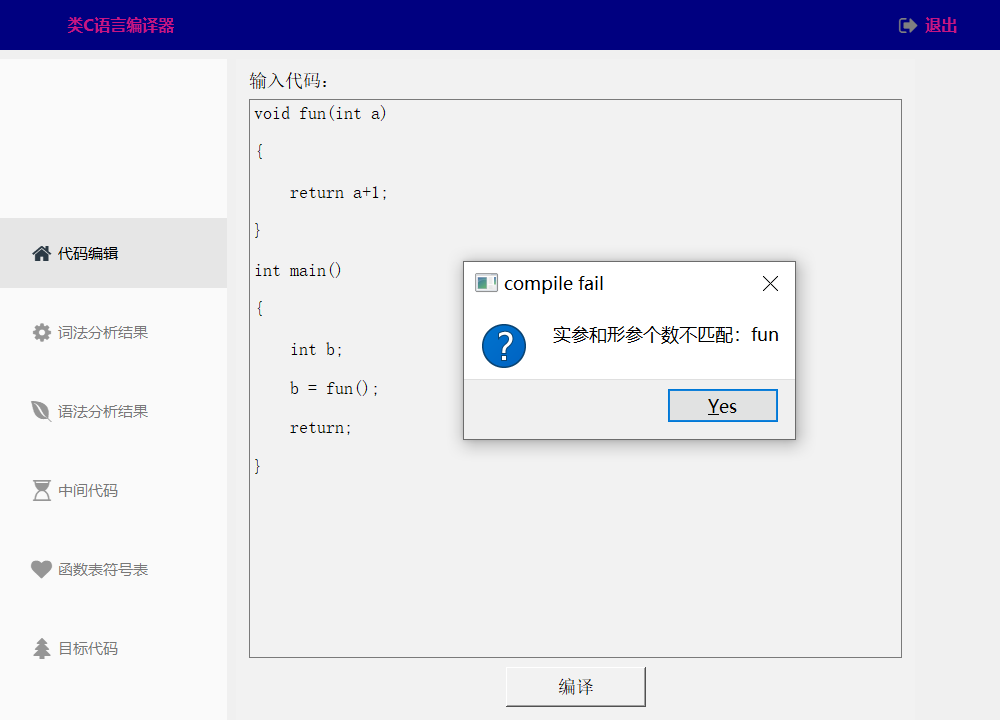


### **4.3.6 函数形参和实参不匹配**

测试用例如下：

|  |
| --- |
| void fun(int a) {  return a+1;  }  int main() {  int b;  b = fun();  return;  } |

测试结果如下：

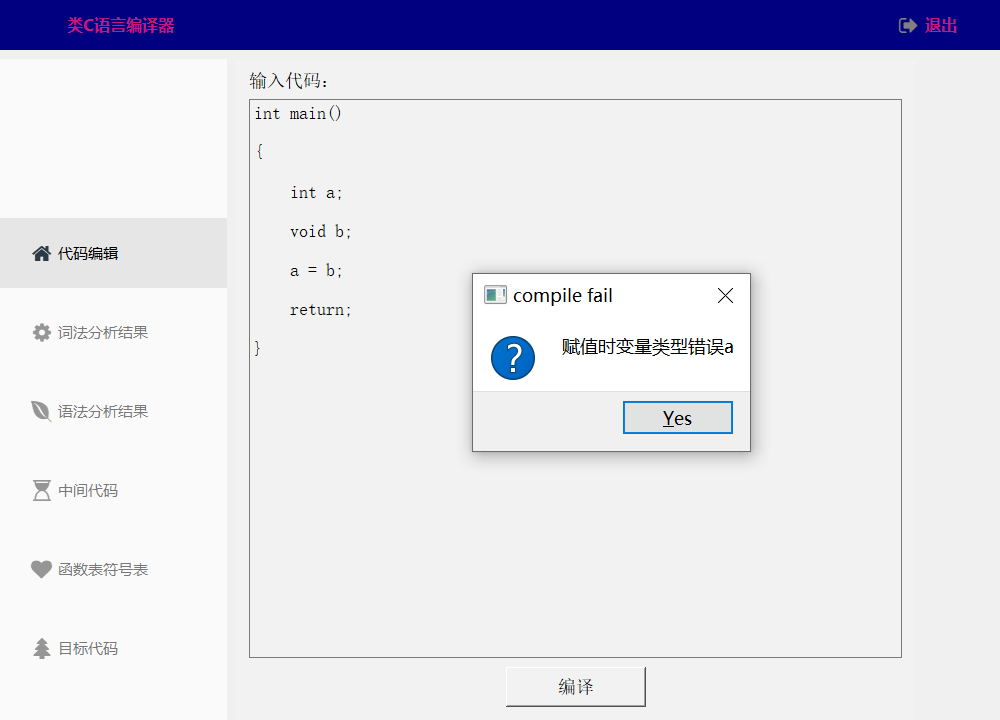


### **4.3.7 变量赋值时类型错误**

测试用例如下：

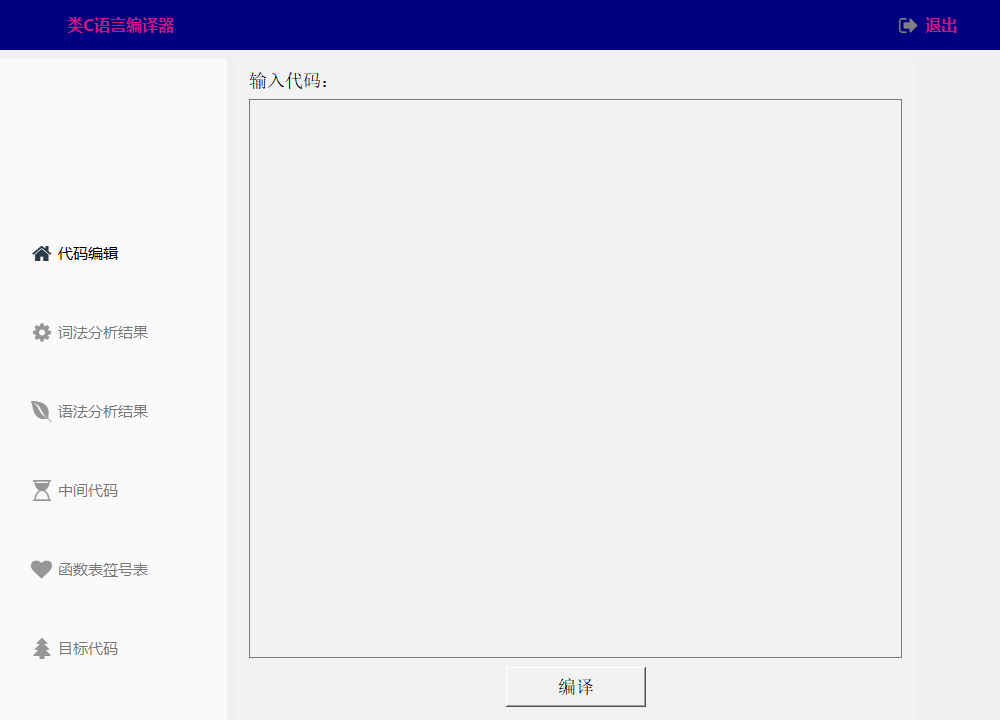
|  |
| --- |
| void fun(int a)  {  return a+1;  }  int main()  {  int b;  b = fun();  return;  } |

测试结果如下：



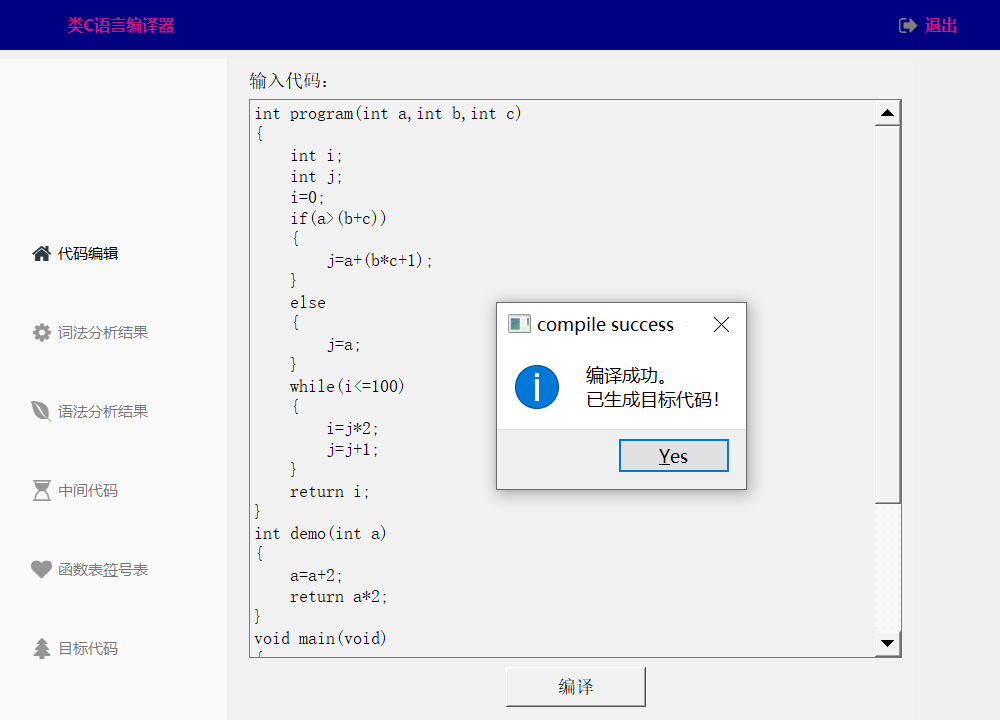
# **5 用户使用说明**

运行程序，即可看到以下的界面：

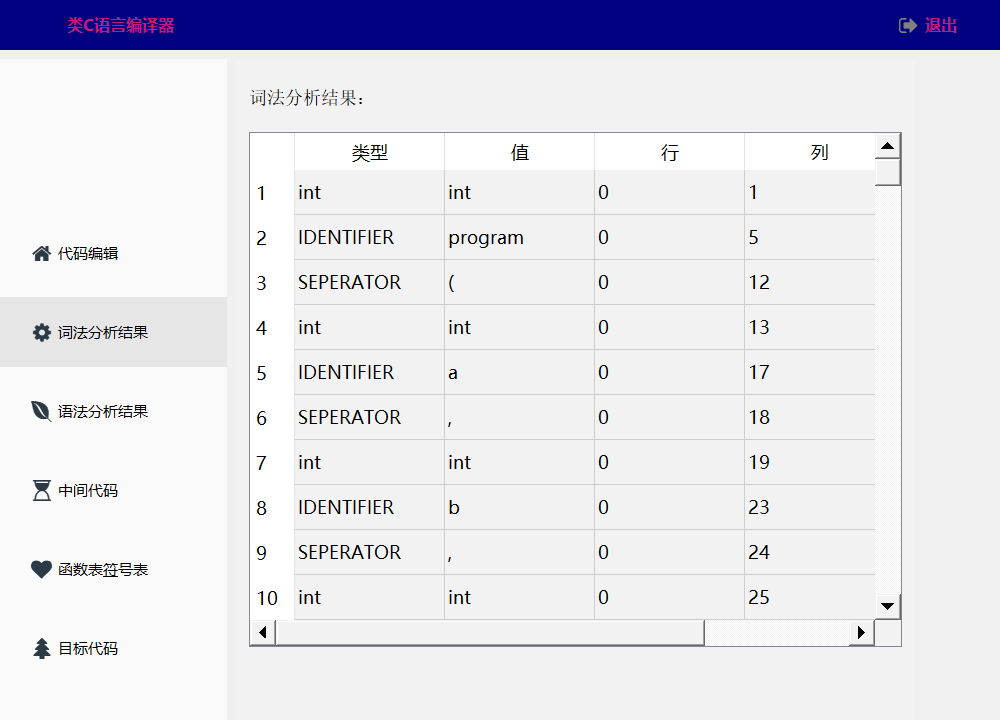
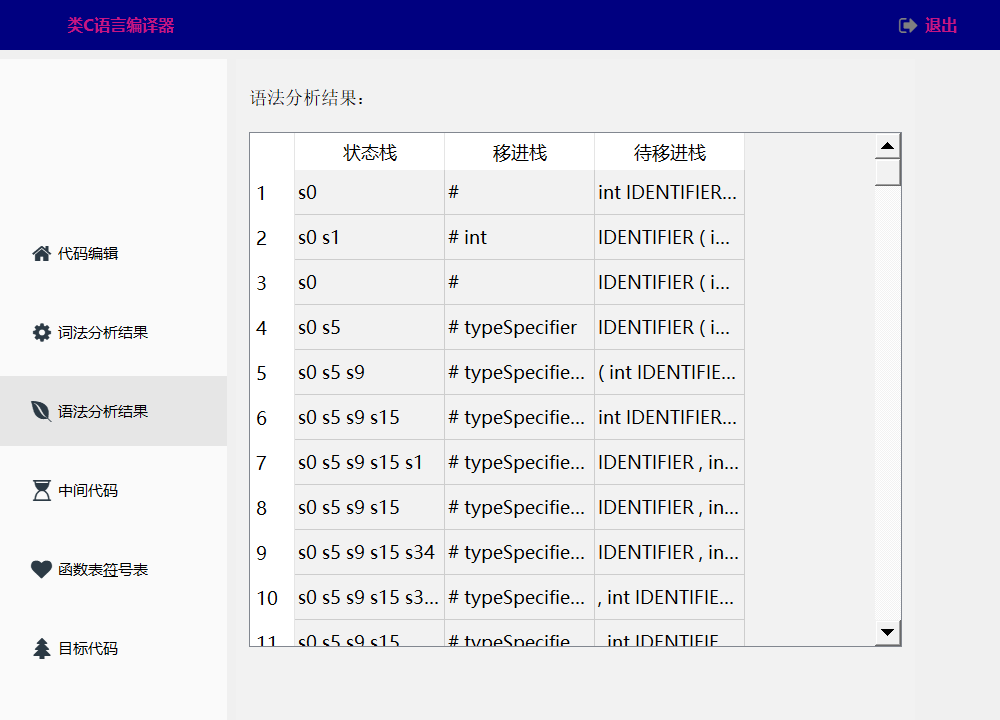


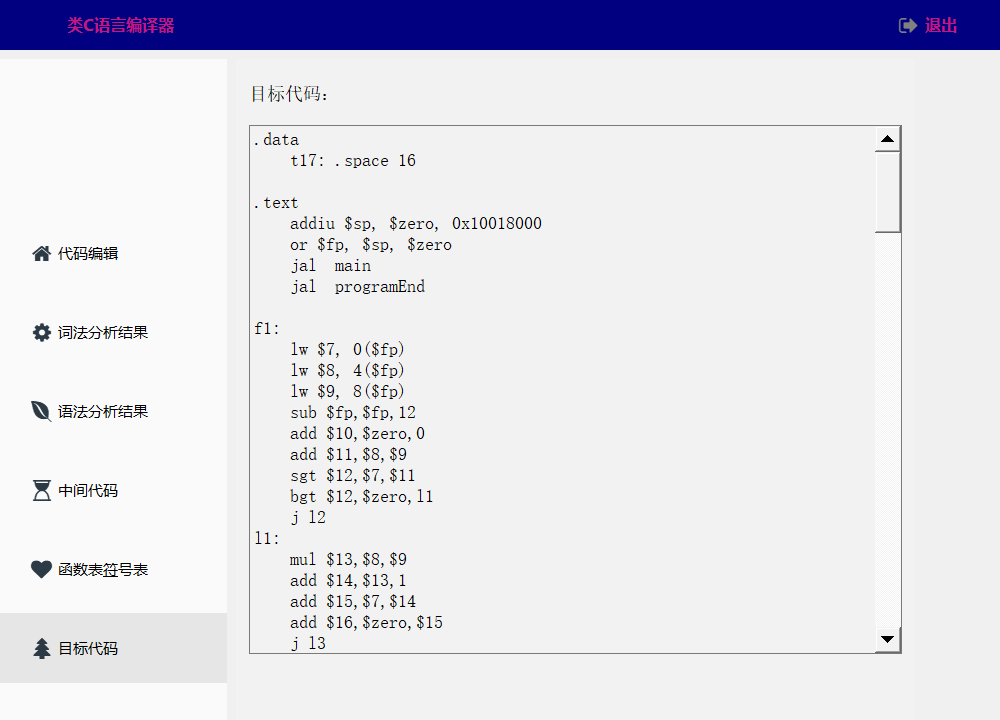
在代码输入框中，可以编辑源代码。编辑完成后，可以点击“编译”按钮进行源代码的编译。

若编译成功，将会进行提示，如下图：

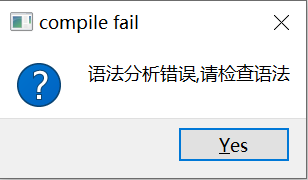
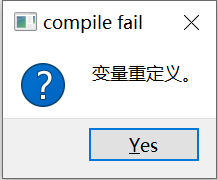
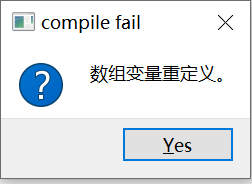


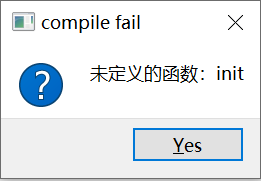
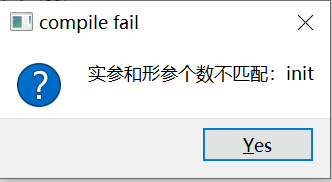
点击确认按钮，即可发现左侧栏目均变为有效，此时可以查看编译的中间结果和最终结果，如下图：



当然，若一开始的编译失败，则会进行错误提示，并且不能查看编译结果。以下是部分可能出现的错误提示：

# **6 课程设计总结**

## 6.1 总结

在本次课设中，我完成了类 C 语言编译器的设计。

基于上学期设计的、从词法分析到中间代码的类 C 语言编译器，我将这部分设计由 C++ 语言转换为更加灵活方便的 python 语言，并在此基础上完成了从中间代码到生成目标代码的过程，并加上了对于数组的处理。

在生成目标代码部分，我严格参照 mips 指令集的特点，使编译器能够编译出完整的、能够在Mars上模拟运行的汇编代码。

此外，我还开发了一个简洁、直观的用户图形界面，使得用户能够方便得使用我的类 C 语言编译器。

## 6.2 可改进的地方

由于时间仓促，此类C 语言编译器仅对正确的测试用例有很好的表现，对于很多可能出现在源代码中的错误，缺少处理。因此，还有许多能够改进也应该改进的地方，这一部分主要集中在静态语义处理部分和错误提示部分。

（1）对于数组的处理

由于测试用例中，仅仅是涉及数组的声明以及直接使用，为了能够让这样的测试用例能够编译，我仅仅是在语法规则中加入相应的产生式，并且用偏移量和对待一般变量的处理方法处理数组的使用。可以说，这样使用的数组“几乎就是一个普通变量”，因为在语义检查中我只检验了数组名是否出现在符号表中。但对于数组的处理不应该仅限于此：

由于数组在定义时指定了数组的大小，因此对于数组越界的问题理论上是可以给出错误提示的（多维数组可能稍微麻烦）；

在数组的使用上，我仅仅检查了数组名是否合理，并没有检查数组的使用是否合理。例如，定义二位数组“int arr[2][2]”，但在使用时却是“arr[0]=2”，这样的错误并不会检测出来。

（2）对于变量作用域的处理

我在处理变量的作用域时，是记录每个变量所在的函数（全局变量相当于在global这个函数中），每个变量只能在自己的作用域中被使用，否则就会报变量未定义。这样的规则并不符合 C 语言规则，它处理不了不同层次的变量的作用域的问题。一种可行的方法是通过同时记录该变量的直接外层函数名和该变量外套有的大括号层数来共同决定变量的作用域，这会更加合理但也更加复杂。

对函数的作用域也存在这样的问题，只不过测试用例完全无法体现。

（3）错误提示的显示

我设计的编译器的错误提示较为粗略，只能够提出是在语法分析还是语义分析阶段出错，对于具体出错的位置并不能提示。一种可行的方案是记住每个 token 在源代码中的位置，当分析到某个 token 出错时，就将其位置作为错误位置进行错误提示。

（4）语法规则的扩充

由于使用的语法规则较为基础，对于一些符合 C 语言的使用习惯的使用方法，在该类 C 语言的编译器中，存在着无法编译的语法分析上的问题。例如定义函数时不指明变量名“int fun(int, int){}”，这种定义方式在 C 中是允许的，可是由于语法规则的限制，这种定义方法在我的类 C 语言编译器中会提示语法分析出错。这样的问题只能够通过丰富语法规则来改进。

（5）寄存器的分配

在寄存器的分配上，参照了一定的 mips 的32个寄存器的使用规范，但仅仅参考了极少部分，很多 mips 寄存器使用规范我缺少深刻理解，因此并没有完全参照。

（6）其他

本次课设还有不少可以改进的地方，包括代码结构的划分和冗余程度以及注释问题，包括图形界面的优化和人性化设计，包括程序的稳定性和鲁棒性等等。

## 6.3 感悟

通过大三一整年对于编译原理的学习，我从一开始对于“编译”的一无所知到现在的能够设计属于自己的类 C 语言编译器、能够将一段 C 语言代码翻译成能够运行的汇编代码，我很有成就感。

在上学期的课程中，我们从编译的定义出发，学习的自动机、词法分析、语法分析、属性文法与语义分析、中间代码、优化、目标代码生成，而在这学期的课设中，我们也基本上是按照这一流程来进行编译器的设计的。首先借助自动机的原理设计词法分析器，然后对于实际的语法产生式构建first集合、构建LR0项目集、构建项目蔟闭包、构建DFA，从而得到LR1分析表，在LR1分析表的基础上进行静态语义处理与中间代码的翻译工作，最后使用寄存器分配算法来得到目标代码。可以说，所学与所用紧密相结合。

在编写编译器的过程中也遇到的了不少问题，有理论上的困难也有编程上的：处理产生式、语义分析难以下手、寄存器分配过程也比较繁杂……在遇到这些困难后，查阅资料和寻求同学的帮助起到了很大的作用，也因为此，我才能顺利完成此次课设任务。

经过这次编译原理课程实验，我们对编译原理系统知识有了更深一步的掌握。经过真正动手实践之后，才能对编译原理整个知识体系有更加深刻的理解，进而回归到知识本身，产生不一样的看法与见解。编译原理是计算机中比较难的领域，要掌握好并不简单，但是我一定会继续努力学习，扩充自己的知识，掌握更多的编程设计方法。