《编译技术》课程设计

文 档

学号：\_\_\_\_\_15231204\_\_\_\_\_\_

姓名：\_\_\_\_\_\_\_周扬\_\_\_\_\_\_\_\_

年 月 日

目录

[一．需求说明 3](#_Toc503207504)

[1．文法说明 3](#_Toc503207505)

[2．目标代码说明 6](#_Toc503207506)

[3. 优化方案\* 7](#_Toc503207507)

[二．详细设计 7](#_Toc503207508)

[1．程序结构 8](#_Toc503207509)

[2．类/方法/函数功能 9](#_Toc503207510)

[3．调用依赖关系 14](#_Toc503207511)

[4．符号表管理方案 15](#_Toc503207512)

[5．存储分配方案 16](#_Toc503207513)

[6. 四元式设计\* 17](#_Toc503207514)

[7. 目标代码生成方案\* 19](#_Toc503207515)

[8. 优化方案\* 21](#_Toc503207516)

[9. 出错处理 24](#_Toc503207517)

[三．操作说明 26](#_Toc503207518)

[1．运行环境 26](#_Toc503207519)

[2．操作步骤 26](#_Toc503207520)

[四．测试报告 28](#_Toc503207521)

[1．测试程序及测试结果 28](#_Toc503207522)

[2．测试结果分析 40](#_Toc503207523)

[五．总结感想 41](#_Toc503207524)

## 一．需求说明

### 1．文法说明

【说明获取的文法、对文法的改写和扩充】

**1.1 原始文法：**

＜加法运算符＞ ::= +｜-  
＜乘法运算符＞  ::= \*｜/  
＜关系运算符＞  ::=  <｜<=｜>｜>=｜!=｜==  
＜字母＞   ::= ＿｜a｜．．．｜z｜A｜．．．｜Z  
＜数字＞   ::= ０｜＜非零数字＞  
＜非零数字＞  ::= １｜．．．｜９  
＜字符＞    ::=  '＜加法运算符＞'｜'＜乘法运算符＞'｜'＜字母＞'｜'＜数字＞'  
＜字符串＞   ::=  "｛十进制编码为32,33,35-126的ASCII字符｝"                                
＜程序＞    ::= ［＜常量说明＞］［＜变量说明＞］{＜有返回值函数定义＞|＜无返回值函数定义＞}＜主函数＞  
＜常量说明＞ ::=  const＜常量定义＞;{ const＜常量定义＞;}  
＜常量定义＞   ::=   int＜标识符＞＝＜整数＞{,＜标识符＞＝＜整数＞}  
                            | char＜标识符＞＝＜字符＞{,＜标识符＞＝＜字符＞}  
＜无符号整数＞  ::= ＜非零数字＞｛＜数字＞｝  
＜整数＞        ::= ［＋｜－］＜无符号整数＞｜０  
＜标识符＞    ::=  ＜字母＞｛＜字母＞｜＜数字＞｝  
＜声明头部＞   ::=  int＜标识符＞|char＜标识符＞  
＜变量说明＞  ::= ＜变量定义＞;{＜变量定义＞;}  
＜变量定义＞  ::= ＜类型标识符＞(＜标识符＞|＜标识符＞‘[’＜无符号整数＞‘]’){,(＜标识符＞|＜标识符＞‘[’＜无符号整数＞‘]’) }  
＜类型标识符＞      ::=  int | char  
＜有返回值函数定义＞  ::=  ＜声明头部＞‘(’＜参数表＞‘)’ ‘{’＜复合语句＞‘}’  
＜无返回值函数定义＞  ::= void＜标识符＞‘(’＜参数表＞‘)’‘{’＜复合语句＞‘}’  
＜复合语句＞   ::=  ［＜常量说明＞］［＜变量说明＞］＜语句列＞  
＜参数表＞    ::=  ＜类型标识符＞＜标识符＞{,＜类型标识符＞＜标识符＞}| ＜空＞  
＜主函数＞    ::= void main‘(’‘)’ ‘{’＜复合语句＞‘}’  
＜表达式＞    ::= ［＋｜－］＜项＞{＜加法运算符＞＜项＞}  
＜项＞     ::= ＜因子＞{＜乘法运算符＞＜因子＞}  
＜因子＞    ::= ＜标识符＞｜＜标识符＞‘[’＜表达式＞‘]’｜＜整数＞|＜字符＞｜＜有返回值函数调用语句＞|‘(’＜表达式＞‘)’  
＜语句＞    ::= ＜条件语句＞｜＜循环语句＞｜<情况语句>|‘{’＜语句列＞‘}’｜＜有返回值函数调用语句＞;   
                      |＜无返回值函数调用语句＞;｜＜赋值语句＞;｜＜读语句＞;｜＜写语句＞;｜＜空＞;｜＜返回语句＞;  
＜赋值语句＞   ::=  ＜标识符＞＝＜表达式＞|＜标识符＞‘[’＜表达式＞‘]’=＜表达式＞  
＜条件语句＞  ::=  if ‘(’＜条件＞‘)’＜语句＞  
＜条件＞    ::=  ＜表达式＞＜关系运算符＞＜表达式＞｜＜表达式＞ //表达式为0条件为假，否则为真  
＜循环语句＞   ::=  do＜语句＞while ‘(’＜条件＞‘)’

＜常量＞   ::=  ＜整数＞|＜字符＞

＜情况语句＞  ::=  switch ‘(’＜表达式＞‘)’ ‘{’＜情况表＞ ‘}’  
＜情况表＞   ::=  ＜情况子语句＞{＜情况子语句＞}  
＜情况子语句＞  ::=  case＜常量＞：＜语句＞

＜有返回值函数调用语句＞ ::= ＜标识符＞‘(’＜值参数表＞‘)’  
＜无返回值函数调用语句＞ ::= ＜标识符＞‘(’＜值参数表＞‘)’  
＜值参数表＞   ::= ＜表达式＞{,＜表达式＞}｜＜空＞  
＜语句列＞   ::=｛＜语句＞｝  
＜读语句＞    ::=  scanf ‘(’＜标识符＞{,＜标识符＞}‘)’  
＜写语句＞    ::=  printf‘(’＜字符串＞,＜表达式＞‘)’|printf ‘(’＜字符串＞‘)’|printf ‘(’＜表达式＞‘)’  
＜返回语句＞   ::=  return[‘(’＜表达式＞‘)’]

文法的附加说明：

（1）char类型的表达式，用字符的ASCII码对应的整数参加运算，在写语句中输出字符

（2）标识符不区分大小写字母

（3）写语句中的字符串原样输出

（4）数组的下标从0开始

（5）情况语句中，switch后面的表达式和case后面的常量只允许出现int和char类型；每个情况子语句执行完毕后，不继续执行后面的情况子语句

**1.2 属性翻译文法**

<加法运算符>↑n::=+｜-

<乘法运算符>↑n::=\*｜/

<关系运算符>↑n::=<｜<=｜>｜>=｜!=｜==

<字母>↑n::=＿｜a｜．．．｜z｜A｜．．．｜Z

<数字>↑n↑c↑s::=0｜<非零数字>↑n↑c↑s

<非零数字>↑n↑c↑s::=1｜．．．｜9

<字符>↑c↑s::='<加法运算符>'｜'乘法运算符>'｜'<字母>'｜'<数字>'

<字符串>↑i::="｛十进制编码为32,33,35-126的ASCIICha｝"@insert

<程序>::=［<常量说明>］［<变量说明>］{<有返回值函数定义>|<无返回值函数定义>}<主函数>

<常量说明>::=const<常量定义>;{const<常量定义>;}

<常量定义>::=int↑t <标识符>↑n＝<整数>↑c↑s @insert↓t↓n↓c↓s {,<标识符>↑n＝<整数>↑c↑s@insert↓t↓n↓c↓s }

| char↑t <标识符>↑n＝<字符>↑c↑s @insert↓t↓n↓c↓s {,<标识符>↑n＝<字符>↑c↑s @insert↓t↓n↓c↓s }

<无符号整数>::=<非零数字>{<数字>}

<整数>↑c↑s::=[+|-]< 无符号整数>|0

<标识符>↑n::=< 字母>{<字母>|<数字>}

<声明头部>↑t↑n::=int↑t <标识符>↑n

｜char↑t <标识符>↑n

<变量说明>::=<变量定义>;{<变量定义>;}

＜变量定义＞::=＜类型标识符＞↑t＜标识符＞[‘[’＜无符号整数＞‘]’]{,＜标识符＞[‘[’＜无符号整数＞‘]’] }

<类型标识符>↑t::=int|char

<有返回值函数定义>::=<声明头部>↑t↑n @begin↓t↓n↑i ‘(’<参数表>↓n ‘)’ @stepin↓n@pop↓n ‘{’<复合语句>‘}’ @end @stepout

<无返回值函数定义>::=void<标识符>↑n @begin↓n↑i ‘(’<参数表>↓n ‘)’ @stepin↓n @ pop↓n ‘{’<复合语句>‘}’ @end @stepout

<复合语句>::=［<常量说明>］［<变量说明>］<语句列>

<参数表>↓n::=< 类型标识符>↑t <标识符>↑s@insert↓n↓t↓s {,<类型标识符>↑t <标识符>↑s @insert↓n↓t↓s }|<空>

<主函数>::=void main @begin ‘(’‘)’ @stepin ‘{’<复合语句>‘}’ @end @stepout

<表达式>↑i::=[+|-] ↑s <项>↑i1{<加法运算符>↑o <项>↑i @add↓i1↓i2↓o↑i3} sign↓in↑i

<项>↑i::=< 因子>↑i1 {<乘法运算符>↑o <因子>@mul↓i1↓i2↓o↑i3}

<因子>↑i::=< 标识符>↑n @find↓n↑i[‘[’<表达式>‘]’]

｜<整数>↑c↑s @ consttemp↓c↓s↑i

｜<字符>↑c↑s @ consttemp↓c↓s↑i

｜<有返回值函数调用语句>↑i

｜‘(’<表达式>↑i ‘)’

<语句>::=<条件语句>

｜<循环语句>

|<情况语句>

｜‘{’<语句列>‘}’

｜<有返回值函数调用语句>;

｜<无返回值函数调用语句>;

｜<赋值语句>;

｜<读语句>;

｜<写语句>;

｜<空>;

｜<返回语句>;

<赋值语句>::=<标识符>↑n @find↓n↑i1[‘[’<表达式>‘]’]＝<表达式>↑i2 @assign↓i1↓i2

<条件语句>::=

if‘(’<条件>↑i ‘)’ @ecbrf↓i↑a <语句>@labprod↓a

<条件>↑i::=< 表达式>↑i1<关系运算符>↑o <表达式>↑i2@rel↓i1↓i2↓o↑i3

｜<表达式>↑i //Exp为0Cond为假，否则为真

<循环语句>::=do@clabprod↑b <语句>@brf↓b @labprod↓a while‘(’<条件>↑i ‘)’ @ecbrf↓i↑a

<Step>↑i::=<UnInteger>

＜常量＞↑c↑s   ::=  ＜整数＞|＜字符＞

＜情况语句＞  ::=  switch ‘(’＜表达式＞↑i1 @push ↓i↓i1↓m ‘)’ ‘{’＜情况表＞↓i1 ‘}’  
＜情况表＞↓i1   ::=  ＜情况子语句＞↓i1{＜情况子语句＞↓i1}@labprod↓a  
＜情况子语句＞↓i1  ::=  case＜常量＞@rel↓i1↓i2↓o↑i3：@ecbrf↓i↑a＜语句＞@labprod↓a

<有返回值函数调用语句>↑j::=< 标识符>↑n @find↓n↑i ‘(’<值参数表>↓i ‘)’ @call↓i↑j

<无返回值函数调用语句>::=<标识符>↑n @find↓n↑i ‘(’<值参数表>↓i ‘)’ @call↓i

<值参数表>↓i::=< 表达式>↑i1 @push ↓i↓i1↓m {,<表达式>↑i2 @push ↓i↓i2↓m }｜<空>

<语句列>::=｛<语句>｝

<读语句>::=scanf‘(’<标识符>↑n @find↓n↑i @read↓i {,<标识符>↑n @find↓n↑i @read↓i }‘)’

＜写语句＞::=printf(‘(’＜字符串＞↑i1 [,＜表达式＞↑i2]‘)’ |‘(’＜表达式＞↑i2 ‘)’)

<返回语句>::=return[‘(’<表达式>↑i @ret↓i ‘)’] @ret

注：其中蓝色的语法除了插入予以翻译子程序，还在源语法的基础上做修改以适用于自顶向下的语法语义分析程序。

### 2．目标代码说明

【说明要生成的目标代码指令及含义】

Mips汇编代码，由于c0文法简单且实现的功能有限，所以一些基础指令足以支撑程序实现。以下是所需要生成的汇编代码及其语义举例：

|  |  |
| --- | --- |
| LW | 加载字 |
| LA | 取地址 |
| SW | 存储字节 |
| ADD | 符号加 |
| ADDI | 立即数加 |
| SUB | 符号减 |
| MULT | 符号乘 |
| DIV | 符号除 |
| BEQ | 等于转移 |
| BNE | 不等转移 |
| BLEZ | 小于等于零时转移 |
| BGTZ | 大于零转移 |
| BLTZ | 小于零转移 |
| BGEZ | 大于等于零转移 |
| J | 跳转 |
| JR | 跳转至寄存器中的存储地址 |
| JAR | 跳转并链接 |
| MFHI | 读HI寄存器 |
| MFLO | 读LO寄存器 |
| SYSCALL | 系统调用 |

### 3. 优化方案\*

【说明需要完成的优化方案及其要求】

**3.1基本块内部的公共子表达式删除（DAG图）；**

首先对中间代码进行基本块的划分，再对各个基本块中的表达式代码构造DAG图，最后利用构造好的DAG图，利用启发式算法消除基本块中的公共子表达式，生成优化后的表达式中间代码。

**3.2全局寄存器分配（着色算法）；**

通过活跃变量分析建立冲突图，在根据可以利用的全局寄存器数量，依照启发式冲突图着色算法分配全局寄存器

**3.3数据流分析（通过活跃变量分析建立冲突图）；**

对中间代码进行基本块的划分，对各分程序中的基本块构造def和use集，然后利用两者计算in和out集，得到变量的活跃信息，为全局寄存器分配做准备

## 二．详细设计

### 1．程序结构

【从总体上描述程序的结构，文字或图示均可】

**程序主文件main.cpp：**包含主函数main

**词法分析程序：**lexical.cpp：包含词法分析的主要函数getsym()，以及许多必须的符号处理函数，同时包含打开测试程序的函数和输出词法分析结果的函数（错误处理函数error放在了其中。）

**语法分析程序：**syntax.cpp：包含语法分析程序的顶部函数program()以及各类子语法程序的分析程序，自顶向下由<程序>语法结构开始分析，其中还包含了错误时的跳读程序next\_semi()等。（语义分析的一部分也放在了其中）

**符号表管理：**symboltable.cpp，包含符号表管理的函数，包括insertsymboltable，getsymindex等等。

**出错处理：**error()函数，放置在lexical.cpp中，未单独设置文件。

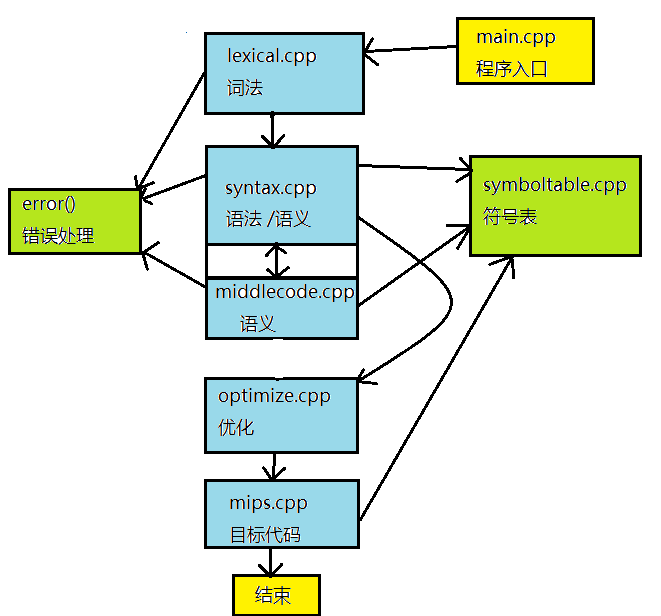
**语义分析与中间代码生成：**由属性翻译文法对语法分析程序中的程序进行改写与扩充，加入了语义分析的子程序，例如insertsymboltable等和生成中间代码的函数。而生成中间代码的函数都定义在middlecode.cpp中，包括ConstMidCode等。

**中间代码优化：**optimize.cpp:包括了中间代码优化和全局寄存器分配的函数，例如dataflowanalysis，dag\_opt等。

**目标代码生成：**mips.cpp：包含由中间代码生成mips目标代码的函数，例如主要函数mips\_generate()等。

**头文件：**global.h：对各文件中定义的全局变量、函数、结构体等进行声明，以让不同文件的函数能够相互调用。

程序结构的图示，其中黄色块为程序的入口与结束，蓝色块为编译的基本步骤的cpp文件与其所属步骤，绿色为错误处理和符号表管理的函数或所属文件。



### 2．类/方法/函数功能

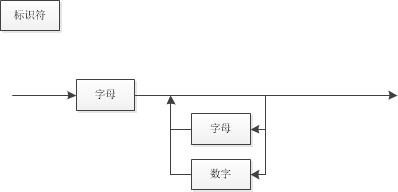
【描述各类/方法或函数的功能，以及关键算法】

**2.1 词法分析程序：**

主要函数：

|  |  |
| --- | --- |
| Getchar() | 获取一个字符 |
| getsym() | 获取一个单词 |

功能：从源程序的第一个字符开始顺序读字符，一次读一个，根据所读进的字符识别各类字符；对数字常数完成数字字符串到十进制数值的转换，并将其值保留；删去空格、换行、制表等字符；将识别出来的被空白字符分割的单词以二元式的形式输出。

关键算法：将各个种类的字符编码，根据文法中离终结符号较近的底层文法画出单词的状态图，再由这些状态图和词法分析程序算法构建词法分析程序。（算法示例《编译技术》P71）例如标识符的语法图：

**2.2 语法分析程序：**

主要函数：

|  |  |
| --- | --- |
| nextsym() | 读取下一个单词 |
| program() | <程序>的分析程序 |
| constdefine() | <常量声明>的分析程序 |
| vardefine() | <变量声明>的分析程序 |
| functiondefine() | 函数定义的分析程序 |
| mainfunction() | <主函数>的分析程序 |
| compoundstatement() | <复合语句>的分析程序 |
| conditionalstatement() | <条件语句>的分析程序 |
| loopstatement() | <循环语句>的分析程序 |
| casestatement() | <情况语句>的分析程序 |
| statementlist() | <语句列>的分析程序 |
| assignmentstatement() | <赋值语句>的分析程序 |
| functioncall(bool return\_not) | 函数调用的分析程序 |
| readstatement() | <读语句>的分析程序 |
| writestatement() | <写语句>的分析程序 |
| returnstatement() | <读语句>的分析程序 |
| expression() | <表达式>的分析程序 |
| term() | <项>的分析程序 |
| factor() | <因子>的分析程序 |

功能：接受从词法分析程序中得来的单词，按照文法识别出各语法成分，同时进行语法检查，为语义分析和代码生成做准备。

关键算法：利用递归下降分析法，自顶向下地分析语法结构。约定：当调用某个分析子程序时，它所要分析的第一个符号已经读入了symbol。注意，在实现语法分析程序之前，要保证文法不带有左递归与回溯。与词法分析程序一样依照语法图构建程序，例如表达式的语法图：



**2.3 符号表管理**

主要函数：

|  |  |
| --- | --- |
| checkname(string name,int level) | 检查是否在同层次有重名变量 |
| insertsymboltable() | 符号表插入（参数略） |
| getsymindex(string name,string kind) | 取得标识符在符号表中的索引 |
| deletefunction() | 函数结束后删除局部变量信息 |
| insertmipssymboltable() | 生成mips代码时的符号表插入 |
| getMIPSindex(string name) | 取得mips变量在mips符号表中的索引 |
| getaddress(string name) | 得到变量在栈中与fp的相对地址 |

功能：收集、记录和使用源程序中的一些语法符号（标识符）的相关信息，已便检查语义的正确性并辅助生成正确的代码。

关键算法： c0语言是分程序结构语言，因此除了普通的插入删除等符号表操作，还需要具有定位与重定位的能力。考虑用结构指针实现，这样方便函数管理字表。具体设计在符号表管理方案中。

**2.4 出错处理**

主要函数：

|  |  |
| --- | --- |
| error(int e) | 根据参数e进行相应的错误处理 |

功能：检查出各类错误，定位出错位置，返回错误信息，并跳过错误部分继续编译，尽可能找出所有错误。具体设计在出错处理中。

**2.5 语义分析与中间代码生成**

主要函数：

|  |  |
| --- | --- |
| ConstMidCode(string op, string type, int num, string name) | 生成常量定义中间代码 |
| VarMidCode(string type, int length, string name) | 生成变量定义中间代码 |
| FunctionMidCode(string name) | 生成函数定义中间代码 |
| ParaMidCode(string type, string name) | 生成形参中间代码 |
| SetParaMidCode(string t\_para) | 生成实参中间代码 |
| CallReturnFunctionMidCode(string name, string V) | 生成有返回值函数调用中间代码 |
| CallFunctionMidcode(string name) | 生成无返回值函数调用中间代码 |
| ExpressionMidCode(string op, string ob1, string ob2, string V) | 生成表达式中间代码 |
| AssignmentMidCode(string op, string ob1, string ob2, string ob3) | 生成赋值中间代码 |
| JumpMidCode(string op,string ob1,string ob2,string label) | 生成跳转中间代码 |
| LabelMidCode(string label) | 生成标签中间代码 |
| ScanfMidCode(string name) | 生成读语句中间代码 |
| PrintfMidCode(string str, string name) | 生成写语句中间代码 |
| ReturnMidCode(string name) | 生成返回语句中间代码 |
| FstartMidCode() | 函数开始标志中间代码 |
| FendMidCode() | 函数结束标志中间代码 |

功能：将与上下文有关的信息记录在符号表或其他数据区中，对表达式和赋值语句中的操作数进行一致性检查，类型不一致时要进行类型转换，分析由语法分析所识别的语句的意义，并作出相应的语义处理。

关键算法：在原语法的基础上进行改写，改写为属性翻译文法，将与语义有关的函数插入原语法分析程序中，再利用自顶向下语法指导翻译，对不同的语句进行不同的语义处理，包括：声明语句、表达式语句、赋值语句、控制语句、输入/输出语句、返回语句等。

**2.6 中间代码优化：**

主要函数：

|  |  |
| --- | --- |
| dataflowanalysis(int start) | 从索引为start的中间代码开始数据流分析 |
| search\_dag() | 在dag图中寻找变量（参数略） |
| search\_op\_dag() | 在dag图中寻找操作符（参数略） |
| dag\_opt() | dag图消除基本块内多余代码 |

功能：为目标代码生成提供优化后的中间代码，提高目标程序的运行效率，获得更为紧凑简洁的目标代码。

关键算法：基本块划分算法、基本块的DAG图表示、通过DAG图消除局部公共子表达式、基本块的活跃变量数据流分析、全局寄存器的图着色算法。

**2.7 目标代码生成**

主要函数：

|  |  |
| --- | --- |
| defineGlobalMIPS() | 全局变量或常量MIPS代码生成（参数略） |
| defineMIPS(string name, string value) | 局部变量或常量MIPS代码生成 |
| returnMIPS() | 函数返回MIPS代码生成 |
| savestackMIPS() | 保存现场MIPS代码生成 |
| calculateMIPS(string op,string ob1,string ob2,string ob3) | 表达式MIPS代码生成 |
| assignarrMIPS(string ob1, string ob2, string ob3) | 数组赋值MIPS代码生成 |
| getarrMIPS(string ob1, string ob2, string ob3) | 取数组元素MIPS代码生成 |
| assignMIPS(string ob1, string ob3) | 普通赋值MIPS代码生成 |
| scanfMIPS(string ob3) | 读MIPS代码生成 |
| printfMIPS(string ob1, string ob2) | 写MIPS代码生成 |
| jumpMIPS(string op, string ob1, string ob2, string ob3) | 跳转MIPS代码生成 |
| mips\_generate() | MIPS代码生成的总函数 |

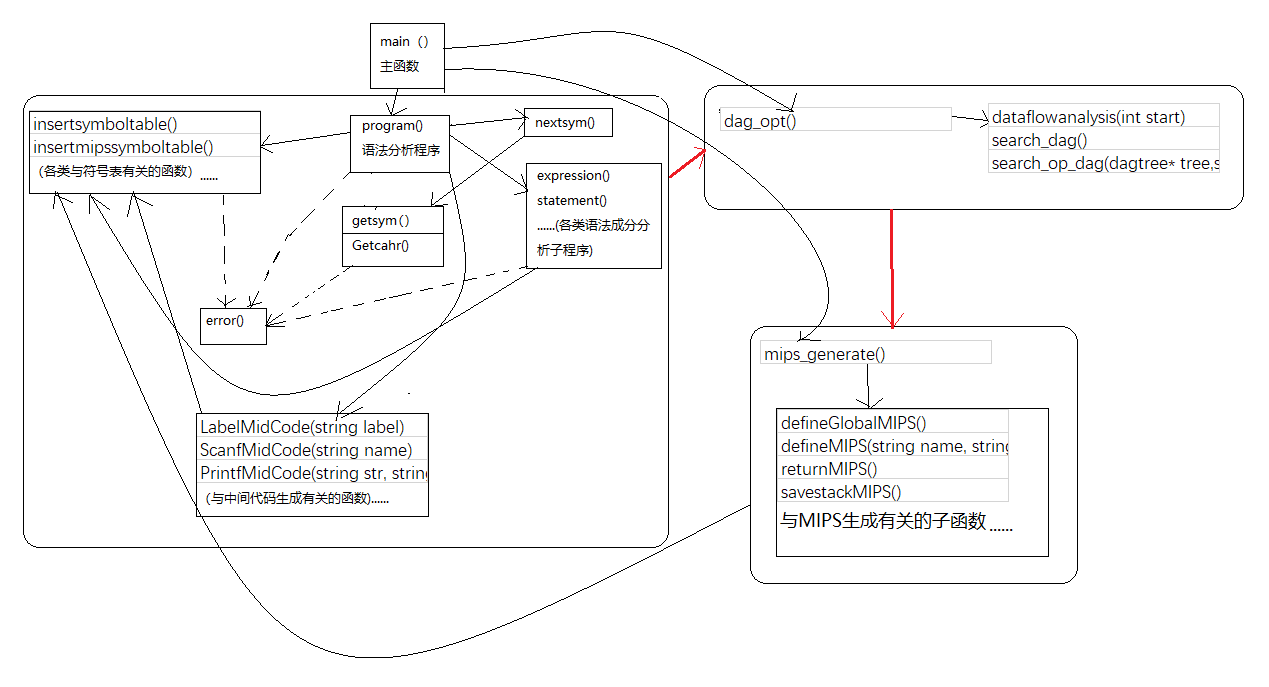
功能：生成目标代码。

关键算法： 利用中间代码与目标代码的对应关系生成MIPS代码。例如return中间代码对应了sp、fp复位，寄存器恢复以及跳转$ra。具体在目标代码生成部分详述。

### 3．调用依赖关系

【说明各类之间的关系，方法/函数之间的调用关系】

下图为函数间的调用关系：



其中黑色实箭头代表调用关系，红色实箭头代表调用顺序，黑色虚箭头为错误处理程序的调用，矩形代表一个或一类函数，三个圆角矩形分别代表：生成中间代码及其前、中间代码优化、目标代码生成。

### 4．符号表管理方案

【说明符号表的数据结构、管理算法】

typedef struct symnode

{

string name; //名

string kind; //const，var....

string type; //int char string...

int value; //值

bool arr; //数组标志

int length; //长度

int level; //层次

int depth; //函数运行栈深度

}symele;

typedef struct mipsnode

{

string name; //名

string type; //类型

string length; //长度

string level; //层次

}mipsele;

extern vector<symele> symboltable;

extern vector<mipsele> mipssymboltable;

管理算法：

C0结构简单，故在初步的符号表设计中省略了pascal-S符号表中的一些内容。

符号表以vector存在。由于在我的语法语义设计中，在一个函数编译时，中间代码随之生成，而在该函数编译结束后可计算出该函数所需要的运行栈深度，然后将其局部变量删除，因此将符号表分为目标代码生成前的符号表symboltable以及目标代码生成时的符号表mipssymboltable，symboltable为第一遍生成的符号表，在整个程序结束时不包括局部变量，mipssymboltable为由中间代码生成的符号表，用于目标代码的生成。

在遇到新的标识符声明时判断若不是关键字，则调用checkname()函数判断该标识符有无同名同层次已定义标识符，有则报错，否则则调用insertsymboltable()插入符号表。当程序需要判断一个标识符是否已经被定义且可被使用或者需要查找表中的某些信息，则调用getsymindex()，若未定义则报错，否则返回标识符在表中的位置。在一个函数编译结束后可计算出该函数所需要的运行栈深度，然后调用deletefunction()将其局部变量删除，能够这样操作的原因是所有变量的初值都已经记录在了中间代码中。

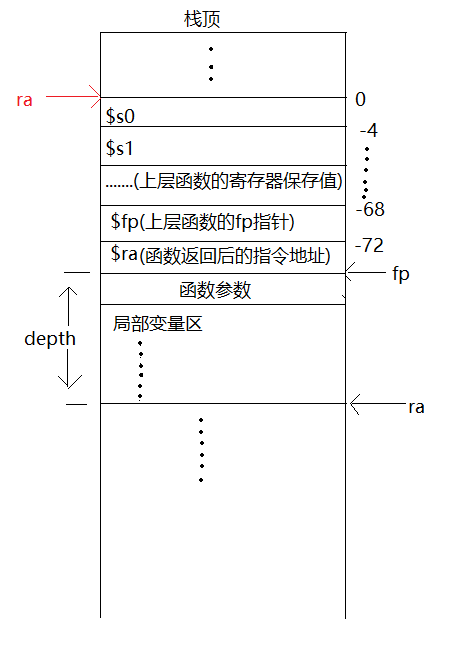
而在生成MIPS代码时，就能够利用symboltable中的全局变量信息、函数信息，以及记录在中间代码中的局部变量和临时变量重构函数的符号表，新符号表为mipssymboltable。在生成目标代码时，需要的符号表操作都是建立在mipssymboltable之上的。

### 5．存储分配方案

【说明运行时的存储组织及管理方案，运行栈结构】

由于c0语言中存在对函数的递归调用，因此需要使用动态存储分配进行管理，在进入一个程序模块（函数）时，在运行栈栈顶创建其活动记录。

运行栈结构如下图，其中红色的ra为上一层函数的ra，黑色的ra为运行时刻函数的ra：



### 6. 四元式设计\*

【对采用的四元式进行详细说明】（待修改）

四元式结构体：

typedef struct middle

{

string op;

string ob1;

string ob2;

string ob3;

}midcode;

四元式设计表：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 编号 | 操作符 | 操作数一 | 操作数二 | 操作数三 | 语义 |
| 0 | CONST | type | num | name | 常量定义，num为常量的值 |
| 1 | INT |  | length | name | 变量定义 |
| 2 | CHAR |  | length | name | 变量定义 |
| 3 | LABEL |  |  | name/labelname | 函数定义/标签生成 |
| 4 | PARA | type |  | name | 形参定义 |
| 5 | PUSH |  |  | t\_para | 实参入栈（在符号表生成t\_para,保存实参的值） |
| 6 | RCALL | name |  | V | 调用有返回值的函数，返回值为符号表中的V |
| 7 | CALL | name |  |  | 调用无返回值的函数 |
| 8 | ADD | op1 | op2 | V | + |
| 9 | SUB | op1 | op2 | V | - |
| 10 | MUL | op1 | op2 | V | \* |
| 11 | DIV | op1 | op2 | V | / |
| 12 | GETAR | name | op | V | V=name[op] |
| 13 | ASN | op |  | name | name=op |
| 14 | ASNAR | op | index | name | name[index]=op |
| 15 | JB | op1 | op2 | label | 大于跳转 |
| 16 | JBE | op1 | op2 | label | 大于等于跳转 |
| 17 | JS | op1 | op2 | label | 小于跳转 |
| 18 | JSE | op1 | op2 | label | 小于等于跳转 |
| 19 | JNE | op1 | op2 | label | 不等于跳转 |
| 20 | JE | op1 | op2 | label | 等于跳转 |
| 21 | JUMP |  |  | label | 无条件跳转 |
| 22 | JNEI | op1 | case中的整数值 | label | case整数不等跳转 |
| 23 | JNEC | op1 | case1中的字符值 | label | case字符不等跳转 |
| 24 | SCANF |  |  | name | 读值入name（scanf(name)） |
| 25 | PRINTF | string(可为空) | name |  | printf(string,name） |
| 26 | RETURN |  |  | name | return(name) |
| 27 | FSTART |  |  |  | 标志函数开始，便于生成代码 |
| 28 | FEND |  |  |  | 标志函数结束，便与生成代码 |

### 7. 目标代码生成方案\*

【说明代码生成有关的数据结构、关键算法】

**有关数据结构：**

在语义阶段（优化）后生成的中间代码vector序列。

**关键算法：**

在目标代码生成前设置mainflag和textflag标号，分别标记是否进入main函数以及是否进入函数段。

从中间代码的开头开始，生成".data"，在未遇到第一个FSTART时，都是全局常量或全局变量的声明或定义或声明，并在MIPS数据段进行相应的有变量名标号的全局数据定义。

当遇到第一个LABEL，说明程序进入代码段，textflag置ture，生成".text"，"addi $fp,$sp,0"，"j main"，设置好栈指针和帧指针的初值，并生成跳转至主函数的跳转指令，保证程序从主函数开始。

每遇到一个LABEL生成MIPS的label。若LABEL中间代码的ob3为函数，则检查是否为main，则说明直接进入主函数，mainflag置ture。所有是函数的LABEL，都要同时由查符号表取出该函数的运行栈深度depth为其分配运行空间。

当中间代码为CALL或者RCALL，生成现场保存的指令与跳转到所调用函数的指令，例如：

"sw $s0,0($sp)"

"sw $fp,-68($sp)"

"addi $sp,$sp,-76"。

当中间代码为PUSH，生成计算参数在运行栈中未知的指令，将push的值存入参数地址。

当中间代码为PARA，生成计算参数在运行栈中位置的指令，将其值取出备用。

当中间代码为RETURN或FEND，生成现场恢复的指令。若mainflag为true，则生成跳转至结束的指令，例如：

"addi $sp,$fp,76"

"lw $fp,-68($sp)"

"lw $ra,-72($sp)"

"lw $s0,0($sp)"。

对于非全局的变量和常量的定义或声明，根据符号表计算其在运行栈中的地址，生成"addi $t0,$zero,(值)"，"sw $t0,-(地址)($fp)"。

当中间代码为表达式的计算，先为运算的变量查找是否非配寄存器，若是，则直接用寄存器生成指令，若无，则查符号表计算其在运行栈中的位置，将其值取出至临时寄存器，若计算之后变量值改变，则将其存回栈中的相应位置。类似的赋值ASN,GETAR,ASNAR，都是先对变量进行是否有分配寄存器的检查，然后再根据检查结果生成相应的指令。

当中间代码为SCANF，先计算所要读的变量在运行栈中的位置或是所分配的寄存器，然后根据所要读的变量的类型，若是整型，生成"addi $v0,$zero,5"，若是字符型，生成"addi $v0,$zero,12"，然后生成"syscall"进行系统调用读入变量。

当中间代码为PRINTF，若ob1的str不为空，则生成：

".data"

stringLabel << ": .asciiz \"" << ob1 << "\""

".text"

"la $a0," << stringLabel

"addi $v0,$zero,4"

"syscall"

而至于写的变量则先计算所要读的变量在运行栈中的位置或是所分配的寄存器，然后根据所要读的变量的类型，若是整型，生成"addi $v0,$zero,1"，若是字符型，生成"addi $v0,$zero,11"，然后"syscall"。

当中间代码为跳转时，根据相应的跳转类型生成相应的跳转MIPS代码。

在中间代码的结尾，也就是最后一个FEND中间代码，生成"EXIT :"，标志着程序的结束。

### 8. 优化方案\*

【说明代码优化有关的数据结构、关键算法】

有关数据结构：

在语义阶段后生成的中间代码vector序列。

DAG图节点结构：

typedef struct dagnode

{

vector<string> var; //该节点的变量

vector<dagnode\*> father; //指向该节点的父节点

string nodevalue; //该节点的值，可为常量或表达式操作符

struct dagnode\* leftChild = NULL; //该节点的左孩子，初始化为NULL

struct dagnode\* rightChild = NULL; //该节点的右孩子，初始化为NULL

}dagele;

DAG图：

typedef struct DAG

{

vector<dagele\*> element; //在同一棵DAG树上的DAG节点的指针vector

}dagtree; //在DAG生成的过程中可能会有多个dagtree，最后都会合并为一个DAG图

数据流活跃变量分析基本块结构：

typedef struct blockinf

{

vector<string> pre; //该基本块的前驱集合

vector<string> suc; //该基本块的猴急集合

int prenum; //该基本块的前驱数目

int sucnum; //该基本块的后继数目

string label; //该基本块的用于标记入口语句的字符串，同时也作为前驱和后继的名

vector<string> def; //该基本块的def集

int defnum; //该基本块的def数量

vector<string> use; //该基本块的use集

int usenum; //该基本块的use数量

vector<string> in; //该基本块的in集

int innum; //该基本块的in数量

vector<string> out; //该基本块的out集

int outnum; //该基本块的out数量

int inchanged; //该基本块在流图分析过程中是否发生改变的标志

}block;

关键算法：

1. **划分基本块：**

（1）首先确定入口语句（每个基本块的第一条语句）的集合

• 规则1：整个语句序列的第一条语句属于入口语句

• 规则2：任何能由条件/无条件跳转语句转移到的第一条

语句属于入口语句

• 规则3：紧跟在跳转语句之后的第一条语句属于入口语句

（2）每个入口语句直到下一个入口语句，或者程序结束，它们之间的所有语句都属于同一个基本块

1. **构建DAG图：**

（1）. 首先建立节点表，初始状态为空。

（2）. 形如z = x op y的中间代码，首先在节点表中寻找x，记录下x当前所对应的节点号i；如果未找到，在DAG图中新建一个叶节点，设其节点号仍为i，标记为x（如x为变量名，该标记更改为x0）；在节点表中增加新的一项(x, i)，表明二者之间的对应关系。右操作数y与x同理，设其对应节点号为j。

（3）. 在DAG图中寻找中间节点，其标记为op，且其左操作数节点号为i，右操作数节点号为j。记录下其节点号k；如果未找到，在DAG图中新建一个中间节点，设其节点号仍为k，并将节点i和j分别与k相连，作为其左子节点和右子节点；

（4）. 在节点表中寻找z，如果找到，将z所对应的节点号更改为k；如果未找到，在节点表中新建一项(z, k)，表明二者之间的对应关系。

（5）. 对输入的中间代码序列依次重复上述步骤3～5。

1. **从DAG导出中间代码的启发式算法**

（1）. 初始化一个放置DAG图中间结点的队列。

（2）. 如果DAG图中还有中间节点未进入队列，则执行步骤3，否则执行步骤5

（3）. 选取一个尚未进入队列，但其所有父节点均已进入队列的中间节点n，将其加入队列；或选取没有父节点的中间节点，将其加入队列

（4）. 如果n的最左子节点符合步骤3的条件，将其加入队列；并沿着当前节点的最左边， 循环访问其最左子节点，最左子节点的最左子节点等，将符合步骤3条件的中间节点依次加入队列；如果出现不符合步骤3条件的最左子节点，执行步骤2

（5）. 将中间节点队列逆序输出，便得到中间节点的计算顺序，将其整理成中间代码序列

1. **数据流的活跃变量分析**

（1）. 计算函数中的基本块的def和out集

（2）. 将包括代表流图出口基本块Bexit在内的所有基本块的in集合，初始化为空集。

（3）. 根据方程out[B] = B的后继基本块P in[P]，in[B] = use[B] (out[B] – def[B]) ，为每个基本块B依次计算集合out[B]和in[B]。如果计算得到某个基本块的in[B]与此前计算得出的该基本块in[B]不同，则循环执行步骤2，直到所有基本块的in[B]集合不再产生变化为止。

1. **冲突图着色分配全局寄存器**

（1）. 找到第一个连接边数目小于K的结点，将它从图G中移走，形成图G’。

（2）. 重复步骤1，直到无法再从G’中移走结点。

（3）. 在图中选取适当的结点，将它记录为“不分配全局寄存器”的结点，并从图中移走。

（4）. 重复步骤1~步骤3，直到图中仅剩余1个结点

（5）. 给剩余的最后一个结点选取一种颜色，然后按照结点被移走的顺序，反向将结点和边添加进去，并依次给新加入的结点选取颜色。（保证有链接边的结点着不同的颜色）

### 9. 出错处理

【说明出错处理方案、错误信息及含义】

**出错处理函数：**

void error(int e)

**出错类型：**

|  |  |
| --- | --- |
| 编码 | 错误描述 |
| 1 | 单字符赋值中缺少‘ |
| 2 | 不合法的单字符 |
| 3 | 字符串缺少“或出现了不合法字符 |
| 4 | 不等于缺少= |
| 5 | 缺少函数名或main |
| 6 | 缺少类型标识符或者常变量定义错位 |
| 7 | 标识符后缺少，或；或[ |
| 8 | 缺少标识符 |
| 9 | 不合法整数 |
| 10 | 缺少等号 |
| 11 | 缺少单字符 |
| 12 | 缺少； |
| 13 | 缺少] |
| 14 | 缺少{ |
| 15 | 缺少） |
| 16 | 缺少( |
| 17 | 缺少} |
| 18 | 函数无返回值 |
| 19 | 主函数返回值不为void |
| 20 | 缺少表达式 |
| 21 | 不合法因子 |
| 22 | 不合法赋值语句 |
| 23 | 不合法语句 |
| 24 | 不合法比较字符 |
| 25 | 缺少while关键字 |
| 26 | 缺少： |
| 27 | 不合法常量 |
| 28 | 缺少= |
| 29 | 不合法操作符 |
| 30 | 重定义标识符 |
| 31 | 程序表溢出 |
| 32 | 标识符在符号表中不存在 |
| 33 | 调用函数时参数个数不匹配 |
| 34 | 函数本无返回值 |
| 35 | 常量不可被再次赋值 |

**出错处理方案：**

1. 在子程序中发现错误，调用error()，传入错误的编号。
2. error()根据错误编号产生相应输出并打印出错行位置和内容。
3. 返回调用error()的子函数，按照出错位置和出错类型采取下面处理方式的其中一种。随后继续编译直到被编译文件结束，不生成MIPS代码。

|  |  |
| --- | --- |
| 函数 | 处理方式 |
| next\_semi() | 跳读至下一个分号 |
| next\_RBRACE() | 跳读至下一个右大括号 |
| next\_LBRACE() | 跳读至下一个左大括号 |
| next\_declare\_head() | 跳读至下一个声明语句的头部 |
| next\_RPAR() | 跳读至下一个右括号 |
| next\_statement() | 跳读至下一个语句头部 |

## 三．操作说明

### 1．运行环境

【说明搭建运行环境的步骤】

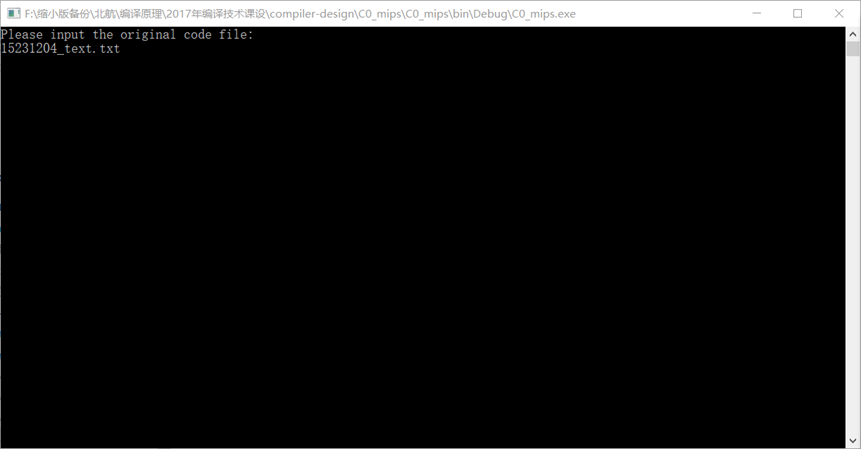
Codeblocks13.12进行运行。

Mars4.5模拟执行生成的MIPS汇编代码。

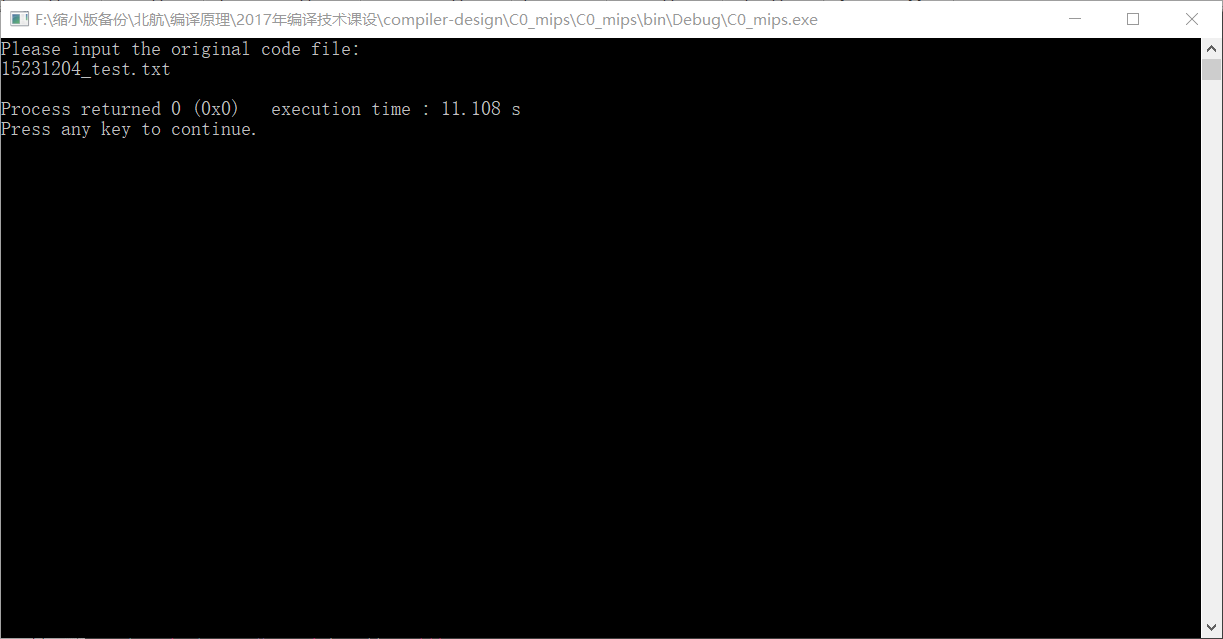
### 2．操作步骤

【详细说明操作步骤】

1. 打开工程文件，运行，输入需要编译的源程序代码文件路径：



1. 正确输入后回车，程序开始编译，若源程序没有错误，则运行结束。



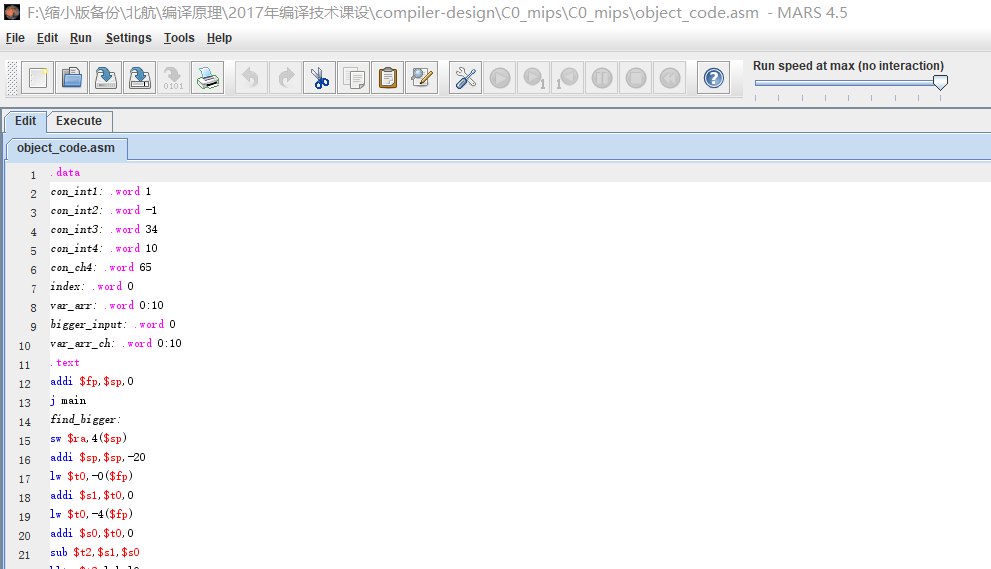
1. 在工程文件夹目录下生成中间编译信息文件debug\_information，其中是中间代码和符号表信息，需要的时候可以查看:



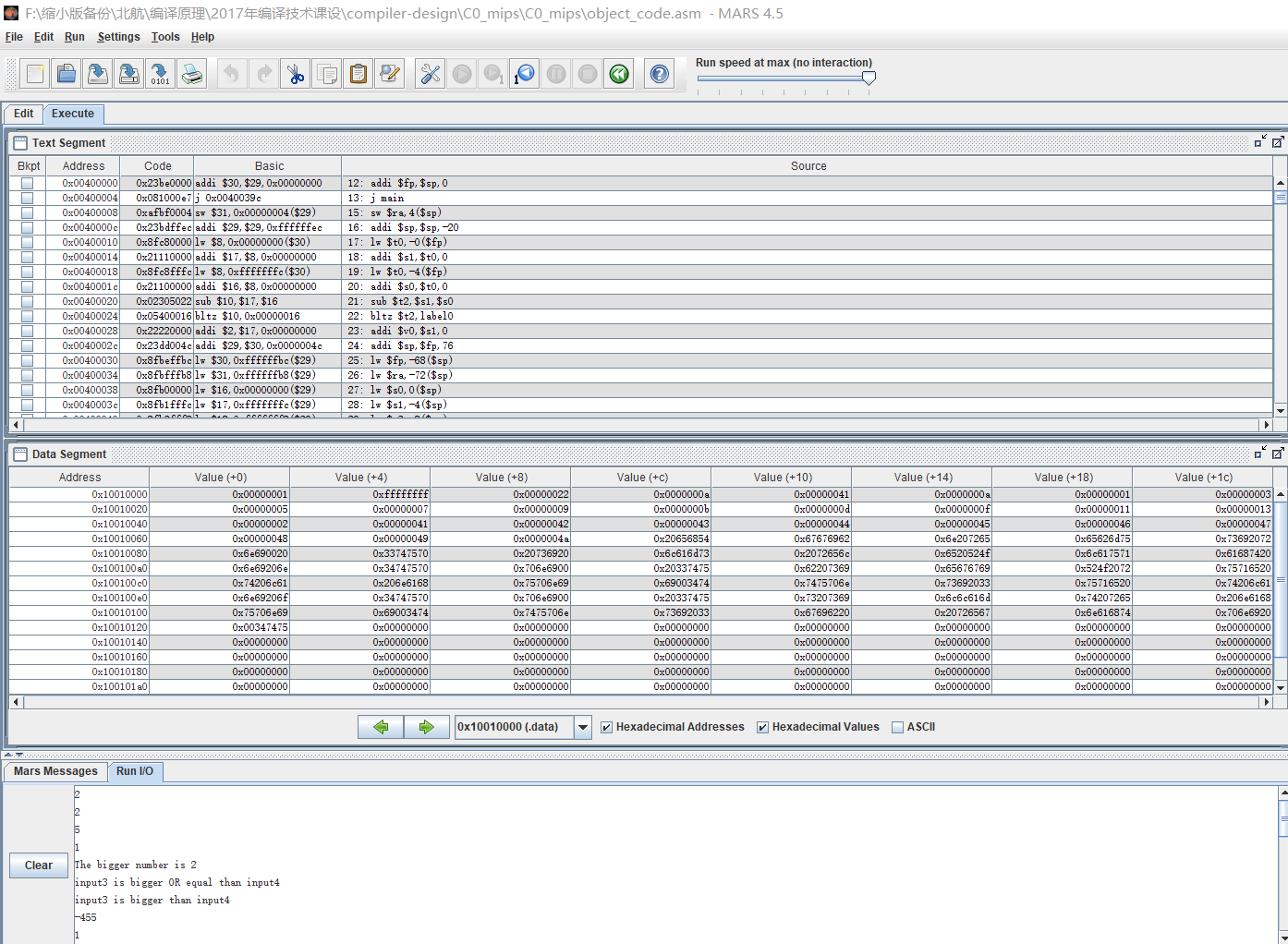
1. 同时还生成目标代码文件object\_code.asm：



1. 用mars4.5打开object\_code.asm：



1. 汇编后可运行，在I/O台显示期望结果：



## 四．测试报告

### 1．测试程序及测试结果

【给出提供的测试程序以及每个程序的测试结果，至少5个正确程序，5个错误程序，无需截屏】

1. **正确程序：**
2. **const** **int** con\_int1 = +1, con\_int2 = -1, con\_int3 = 34, con\_int4 = 10;
3. **const** **char** con\_ch4 = 'A';
5. **int** index, var\_arr[10], bigger\_input;
6. **char** var\_arr\_ch[10];
8. **int** find\_bigger(**int** a, **int** b)
9. {
10. **if** (a >= b)
11. **return**(a);
12. **return**(find\_bigger(a + 1, b));
13. }
15. **void** print\_bigger()
16. {
17. printf("The bigger number is ", bigger\_input);
18. }
20. **void** relation(**int** a, **int** b)
21. {
22. **const** **int** sml\_eql = 0, bgr\_eql = 1;
24. **int** re;
26. **if** (a <= b)
27. {
28. re = sml\_eql;
29. printf("input3 is smaller OR equal than input4");
30. }
31. **if** (a >= b)
32. {
33. re = bgr\_eql;
34. printf("input3 is bigger OR equal than input4");
35. }
36. **if** (a != b)
37. {
38. ;
39. }
40. **if** (a == b)
41. {
42. printf("input3 is equal to input4");
43. }
44. **switch** (re)
45. {
46. **case** 0:
47. {
48. **if** (a < b)
49. {
50. printf("input3 is smaller than input4");
51. }
52. }
53. **case** 1:
54. {
55. **if** (a > b)
56. {
57. printf("input3 is bigger than input4");
58. }
59. }
60. }
61. }
63. **void** main()
64. {
65. **int** input1, input2;
66. **int** input3, input4;
68. scanf(input1, input2);
69. scanf(input3, input4);
71. index = 0;
72. bigger\_input = find\_bigger(input1, input2);
73. print\_bigger();
75. relation(input3, input4);
77. printf(-input3 + input4 \* (input3 / -1) \* 'a' + find\_bigger(input3, input4) \* 7);
79. **do**
80. {
81. var\_arr[index] = index \* 2 + 1;
82. var\_arr\_ch[index] = con\_ch4 + index;
83. printf(var\_arr[index]);
84. printf(var\_arr\_ch[index]);
85. index = index + 1;
87. } **while** (index < con\_int4)
88. }

输入：2 2 5 1

输出：The bigger number is 2

input3 is bigger OR equal than input4

input3 is bigger than input4

-455

1

a

3

b

5

c

7

d

9

e

11

f

13

g

15

h

17

i

19

j

1. **正确程序：**

斐波那契数，输入一个斐波那契的索引，输出对应的斐波那契数。

1. **int** fibo(**int** a)
2. {
3. **if**(a == 1)
4. **return**(1);
5. **if**(a == 2)
6. **return**(1);
7. **return**(fibo(a-2) + fibo(a-1));
8. }
10. **void** main()
11. {
12. **int** input,result;
13. printf("Please input a fibo index:");
14. scanf(input);
15. result = fibo(input);
16. printf("The fibo number is ",result);
17. }

输入：6

输出：

Please input a fibo index:

>>6

The fibo number is 8

1. **正确程序：**

着重测试了写语句与读语句，以及表达式的计算。

1. **void** main()
2. {
3. **const** **char** ch = 'A';
4. **const** **int** in = 2;
5. **int** i;
6. **int** c;
7. printf("ABCDEFGHIJKLMNOPQRSTUVWXYZ abcdefghijklmnopqrstuvwxyz");
8. printf("!#$%&'()\*+,-./:;<=>?@[]^\_`{|}~");
9. printf("const ch and const in respectively are ");
10. printf(ch);
11. printf(in);
12. printf("Please input a number i ");
13. scanf(i);
14. c = -i\*i\*i\*i/-1\*i + i - i;
15. printf("i = ",i);
16. printf("c = -i\*i\*i\*i/-1\*i + i - i, after calculate , c = ");
17. printf(c);
18. }

输入：1

输出：

ABCDEFGHIJKLMNOPQRSTUVWXYZ abcdefghijklmnopqrstuvwxyz

!#$%&'()\*+,-./:;<=>?@[]^\_`{|}~

const ch and const in respectively are

A

2

Please input a number i

>>1

i = 1

c = -i\*i\*i\*i/-1\*i + i - i, after calculate , c =

1

1. **正确程序：**

着重测试了条件语句（if）和情况语句（switch）。

1. **int** a, n;
2. **void** switch\_t(**int** t)
3. {
4. **switch**(t)
5. {
6. **case** 1: printf("switch number is 1");
7. **case** 2: printf("switch number is 2");
8. **case** 3: printf("switch number is 3");
9. **case** 4: printf("switch number is 4");
10. **case** 5: printf("switch number is 5");
11. **case** 6: printf("switch number is 6");
12. }
13. **switch**('A'+ t - 1)
14. {
15. **case** 'A': printf("switch char is A");
16. **case** 'B': printf("switch char is B");
17. **case** 'C': printf("switch char is C");
18. **case** 'D': printf("switch char is D");
19. **case** 'E': printf("switch char is E");
20. **case** 'F': printf("switch char is F");
21. }
22. }
23. **void** main()
24. {
25. a = 1;
26. n = 0;
27. **if** (a == 1) n = n + 1;
28. switch\_t(n);
29. a = 2;
30. **if** (a > 1) n = n + 1;
31. switch\_t(n);
32. a = 0;
33. **if** (a < 1) n = n + 1;
34. switch\_t(n);
35. **if** (a != 1) n = n + 1;
36. switch\_t(n);
37. **if** (a >= 0) n = n + 1;
38. switch\_t(n);
39. **if** (a <= 0) {n = n + 1;}
40. switch\_t(n);
41. **if** (n !=  6)
42. {
43. printf("if statement error!");
44. **return**;
45. }
46. printf("if statement succeed!");
47. }

输出：

switch number is 1

switch char is A

switch number is 2

switch char is B

switch number is 3

switch char is C

switch number is 4

switch char is D

switch number is 5

switch char is E

switch number is 6

switch char is F

if statement succeed!

1. **正确程序：**

输入一组数，进行冒泡排序，输出从小到大的数列。着重测试了循环语句（do-while）。

1. **int** a[100];
3. **void** main()
4. {
5. **int** i,j,n,temp;
6. printf("Please input the number of interger:");
7. scanf(n);
8. **if**(n>=100)
9. {
10. printf("The number is too big");
11. **return**;
12. }
13. **if**(n<=0)
14. {
15. printf("The number is illegal");
16. **return**;
17. }
18. i=0;
19. printf("Please input the interger: ");
20. **do**
21. {
22. scanf(temp);
23. a[i] = temp;
24. i = i+1;
25. }**while**(i<n)
26. i = 0;
27. **do**
28. {
29. j=i+1;
30. **if**(j<n)
31. {
32. **do**
33. {
34. **if**(a[i]>a[j])
35. {
36. temp=a[i];
37. a[i]=a[j];
38. a[j]=temp;
39. }
40. j=j+1;
41. }**while**(j<n)
42. }
43. i=i+1;
44. }**while**( i < n -1)
45. printf("After sorting:");
46. i=0;
47. **do**
48. {
49. temp = a[i];
50. printf(temp);
51. i=i+1;
52. }**while**(i<n)
53. **return**;
54. }

输入：

5 22 44 653 32 5

输出：

Please input the number of interger:

>>5

Please input the interger:

>>22

>>44

>>653

>>32

>>5

After sorting:

5

22

32

44

653

1. **错误程序：**
2. **int** a;
3. **const** **int** b = 1;
4. **void** main()
5. {
6. **return**(1);
7. }

错误类型：

* 1. 常变量定义位置错位
  2. Void函数具有返回值
  3. 主函数返回值不为空

测试结果：报错：

error: in line:2 Missing type identification: int|char|void or the const definition is in wrong position

const int b = 1;

error: in line:2 Missing ',' or ';' or [

const int b = 1;

error: in line:5 This called function has no return!

return(1);

error: in line:6 The main function return is not void

}

Total error num : 4

1. **错误程序：**
2. **void** main()
3. {
4. **const** **char** ch = 'A';
5. **const** **int** in = 2;
6. **int** i
7. **int** c;
8. printf("ABCDEFGHIJKLMNOPQRSTUVWXYZ abcdefghijklmnopqrstuvwxyz");
9. printf("!#$%&'()\*+,-./:;<=>?@[]^\_`{|}~");
10. printf("const ch and const in respectively are ");
11. printf(ch);
12. printf(in);
13. printf("Please input a number i ");
14. scanf(i);
15. c = -i\*i\*i\*i/-1\*i + i - i;
16. printf("i = ",i);
17. printf("c = -i\*i\*i\*i/-1\*i + i - i, after calculate , c = ");
18. printf(c);
19. **if**(in ! 2) **return**;
20. }

错误类型：

* 1. 第5行，缺少分号。
  2. 第13、14、15行，因为第5行的错误导致变量i未录入符号表，未定义。
  3. 第18行，不等于缺少等号，即不合法比较符。

测试结果：

error: in line:6 Missing semi: ; in the former line

int c;

error: in line:6 Ileagal assignment statement

printf("ABCDEFGHIJKLMNOPQRSTUVWXYZ abcdefghijklmnopqrstuvwxyz");

error: in line:14 The identification does not exist in the symbol table!

c = -i\*i\*i\*i/-1\*i + i - i;

error: in line:17 The identification does not exist in the symbol table!

printf(c);

error: in line:18 Missing '=' in NEQUAL

if(in ! 2) return;

error: in line:18 Ileagal comparational symbol

if(in ! 2) return;

Total error num : 6

1. **错误程序：**
2. **void** test(**int** a,**int** b)
3. {
4. **return** (a+b);
5. }
6. **void** main()
7. {
8. **int** c;
9. c = test(3);
10. }

错误类型：

* 1. 第3行，无返回值函数test中返回了值
  2. 第8行，函数参数个数不匹配

测试结果：

error: in line:3 This called function has no return!

return (a+b);

error: in line:8 The number of parameters is not matched!

c = test(3);

error: in line:8 This called function has no return!

c = test(3);

Total error num : 3

1. **错误程序：**
2. **const** **int** a = 1;
3. **int** b;
4. **void** main()
5. {
6. b = 0;
7. a = 3;
8. **do**
9. {
10. b = b+1;
11. }(b<7)
12. ;;;;;;
13. }

错误类型：

* 1. 第6行，常量被再次赋值
  2. 第10行，缺少while关键字

测试结果：

error: in line:6 The const can not be assigned again

a = 3;

error: in line:10 Missing while

}(b<7)

error: in line:10 Ileagal assignment statement

}(b<7)

Total error num : 3

1. **错误程序：**
2. **void** main()
3. {
4. **int** a ,b;
5. a = 1;
6. b = a + ();
7. **switch**(a)
8. {
9. **case** 1: b = 1;
10. **case** 2: b = 2;
11. **default**: b = 0 ;
12. }
13. }

错误类型：

* 1. 第5行，括号中缺少表达式
  2. 第10行，出现了非case关键字且非右大括号

测试结果：

error: in line:5 Missing expression

b = a + ();

error: in line:10 Missing right brace:}

default: b = 0 ;

error: in line:10 Missing right brace:}

default: b = 0 ;

Total error num : 3

### 2．测试结果分析

【说明上述测试程序对语法成分的覆盖情况】

正确程序能够覆盖所有的语法成分。

**第一个正确测试程序**覆盖了所有的语法成分，包括常量、变量、数组、函数定义、语句、表达式、函数调用等，同时对情况语句（switch）以及情况语句（if）的每一个分支都进行了覆盖，有基础的函数递归调用，系统读写等，能较好地完成测试要求。

而第2到第5个正确程序，则是有针对性的对特别的语法成分或者功能进行了较强的测试。

**第二个正确测试程序**为斐波那契数程序，着重测试了函数递归调用功能。

**第三个正确测试程序**着重测试了读语句和写语句。

**第四个正确测试程序**着重测试了情况语句（switch）以及情况语句（if）。

**第五个正确测试程序**为冒泡排序，着重测试了循环语句的功能。

经过测试，五个正确程序均能正确编译得到预期结果。

而由于错误类型较多，不能完全展示，所以便选取了其中具有代表性的语法或语义错误进行展示。

**第一个错误程序**常量定义和变量定义错位，编译程序能够识别错误，但表述上稍微不同，因为在编程时我是按照变量定义后一定是变量定义或者函数声明，因此报错类型为“缺少int关键字”；同时main函数的return语句跟了具体值，也需要报错。

**第二个错误程序**缺少分号，以及又不合法的比较符。

**第三个错误程序**除了无返回值函数test中返回了值，还有函数调用的参数个数不匹配。

**第四个错误程序**中，已经定义的常量被再次赋值，且循环语句缺少while关键字。

**最后一个错误测试程序**中，表达式里出现了空括号的情况，在我的语法中是不被允许的，而且在情况语句中出现了非case的表述，也需要报错。

这五个错误测试程序说明系统能够较好的报出错误以及错误信息，同时能实现一定程度上的跳读继续编译。

## 五．总结感想

【说明在完成课程设计中的收获、认识和感想】

编译课设，是一门超越自我的课。9个文件，超过4000行的代码，是我计算机学习生涯中所独自编写的最大的系统。

在编译的课设中，我对编译的基本原理有了更加深入而实际的理解，编程能力和debug能力得到了很大的提高，对MIPS汇编语言有了更好的理解，对代码优化有了自己的看法，系统设计能力得到了提高。最主要的是，“毅力”和“体力”的锻炼。多少次在bug中熬夜不知所措，多少次在放弃的边缘中挣扎，多少次对自己能力的怀疑，坚持，让我最后发现，原来自己能做到。

一开始的词法分析程序和语法分析程序还算简单，因为课本上的算法和方法非常的详尽，而且也几乎不存在系统各部分的连接问题，因此在并没有完全设计好自己的编译器的时候，我还是能将这两个部分较快较好地完成。而到了中间代码和语义分析阶段，之前设计上的疏忽终于爆发了：符号表的信息保存不够、中间代码的设计不完善……只能是不断地在bug中修改设计，最后又在修改设计中发现bug，着实熬了不少的夜，脱了不少的发。也是从这个阶段开始，我才意识到设计对于编译课设整个系统是有多么的重要，这也让我在后面的任务中转变了，注重了系统联系的设计。在中间代码生成的时候，就开始考虑其域目标代码生成甚至优化之间的关系，不断地完善中间代码的设计，做足了准备工作。我也是从编译课设开始，才真正的认识到了设计的必要性与重要性，也让我能够从多家度多方面去思考问题。

而在课设后期的代码优化，虽然算法在理论课上已经讲得比较详尽，但在编程实现的时候，也还是遇到了很多的困难。可以说，编译课设的整个过程，就是将理论转化为实际的过程。可以说，我自己的实践能力得到了很大的提高，而且和同学们的交流中，也学习到了很多。

在编译课设还未开始的时候，就从各种渠道听说了其困难程度，心里总有一丝害怕，在一开始选择难度的时候也打算选择第二难度，但在周遭同学的“怂恿”下，最后抱着“大不了做不来再换”的态度选择了难度三，直到做起来我才发现，难确实是难，但是只要坚持，多与其他人沟通交流，在bug面前勇于面对，或许编译课设也并没有那么难。最后，我坚持了下来，尽管自己的系统仍有很大的改进空间，比如说设计在一开始还能做得更好，代码优化中的寄存器分配和函数栈的使用还有改进的空间，但总归是完成了整个系统的设计与实现。编译课设，锻炼了我的能力，也让我对今后的的学习有了更大的信心。