《编译技术》课程设计文 档

学号：12061228

姓名：晏旭瑞

2015年 1月 4日

## 需求说明

### 文法说明

【说明获取的文法、对文法的改写和扩充，对文法的解读】

#### 扩充C0文法-高-数组-无实型（无改动版）

＜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＞‘)’  ‘{’＜复合语句＞‘}’  
＜无返回值函数定义＞  ::= void＜标识符＞‘(’＜参数＞‘)’  ‘{’＜复合语句＞‘}’  
＜复合语句＞   ::=  ［＜常量说明＞］［＜变量说明＞］＜语句列＞  
＜参数＞    ::= ＜参数表＞  
＜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＞｜‘{’＜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＞［else＜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＞   ::=  while ‘(’＜条件＞‘)’＜语句＞| for‘(’＜标识符＞＝＜表达式＞;＜条件＞;＜标识符＞＝＜标识符＞(+|-)＜步长＞‘)’＜语句＞  
＜步长＞    ::=  ＜非零数字＞｛＜数字＞｝    
＜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＞‘)’]

#### 分析与改动

该文法无左递归，但是同一个非终结符的多个候选式存在相同的FIRST集，即存在回溯，为使语法结构更清晰，没有提取左公因子消除左递归。目前来看，是避免不了回溯的，即超前扫描，然后进入相应的子程序，所以需要增加单词的回退功能。

以上文法可以分为词法和语法两大部分，并作相应的改动

词法：

＜加法运算符＞ ::= +｜-

＜乘法运算符＞ ::= \*｜/

＜关系运算符＞ ::= <｜<=｜>｜>=｜!=｜==

＜字母＞ ::= ＿｜a｜．．．｜z｜A｜．．．｜Z

＜无符号整数＞ ::= ＜非零数字＞｛＜数字＞｝

＜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｝"

＜标识符＞ ::= ＜字母＞｛＜字母＞｜＜数字＞｝

从中可以提取出所有单词

标志符（包括保留字和自定义的标志符）

＜标识符＞ ::= ＜字母＞｛＜字母＞｜＜数字＞｝

保留字：

void int char const

main scanf printf

return

if else

while for

自定义的标志符分为：

变量名

常量名

函数名

无符号整数

＜无符号整数＞ ::= ＜非零数字＞｛＜数字＞｝

字符

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

字符串

＜字符串＞ ::= "｛十进制编码为32,33,35-126的ASCII字符｝"

分界符

＜加法运算符＞ ::= +｜-

＜乘法运算符＞ ::= \*｜/

＜关系运算符＞ ::= <｜<=｜>｜>=｜!=｜==

=

,:;

(){}[]

语法：

＜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＞}

＜整数＞ ::= ［＋｜－］＜无符号整数＞｜０

＜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＞‘)’ ‘{’＜复合语句＞‘}’

＜无返回值函数定义＞ ::= void＜标识符＞‘(’＜参数＞‘)’ ‘{’＜复合语句＞‘}’

＜主函数＞ ::= void main ‘(’ ‘)’ ‘{’＜复合语句＞‘}’

＜声明头部＞ ::= int＜标识符＞ | char＜标识符＞

~~＜参数＞ ::= ＜参数表＞~~

~~＜参数表＞ ::= ＜类型标识符＞＜标识符＞{,＜类型标识符＞＜标识符＞}| ＜空＞~~

＜参数＞ ::= ＜类型标识符＞＜标识符＞{,＜类型标识符＞＜标识符＞}| ＜空＞

//语句

＜复合语句＞ ::= ［＜常量说明＞］［＜变量说明＞］＜语句列＞

＜语句列＞ ::= ｛＜语句＞｝

＜语句＞ ::= ＜条件语句＞｜＜循环语句＞｜‘{’＜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＞;

＜条件语句＞ ::= if ‘(’＜条件＞‘)’＜语句＞［else＜语句＞］

＜条件＞ ::= ＜表达式＞＜关系运算符＞＜表达式＞｜＜表达式＞

＜循环语句＞ ::= while ‘(’＜条件＞‘)’＜语句＞| for‘(’＜标识符＞＝＜表达式＞;＜条件＞;＜标识符＞＝＜标识符＞(+|-)~~＜步长＞~~<无符号整数>‘)’＜语句＞

~~＜步长＞ ::= ＜非零数字＞｛＜数字＞｝~~

＜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＞

＜赋值语句＞ ::= ＜标识符＞＝＜表达式＞|＜标识符＞‘[’＜表达式＞‘]’=＜表达式＞

＜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＞‘)’]

//表达式

＜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＞‘)’

可以得出以下结论：

1. 命名冲突：同一层内不能重名，即函数名、全局变量、全局常量不重名。函数内部的参数名、局部变量、局部常量不重名。
2. 函数调用：前面的函数不能调用后面的函数。
3. Char型：char型和int的区别在于：char型的值只有8位，并且char在输出语句中按原样输出，因此完全可以将char当做int处理，在赋值或传值的时候不进行类型检查。
4. 参数更接近于局部的变量，可以被赋值，读入
5. printf语句何时输出字符？什么是char型表达式？如果printf语句中的表达式是char型变量、常量、数组元素、返回值类型为char型的函数调用语句，或字符字面值，则输出字符本身。简而言之，就是要判断表达式的类型，如果是char型则输出字符，判断一个表达是的类型是否是char型的关键就是看是否出现字符和字符是否参与运算，只要字符参与了运算就认为输出整数。如

printf(+'a');//输出整数

printf(('a'));//输出字符

1. 扩充C0文法的“扩充”大概是指增加了对字符和数组的支持。

### 目标代码说明

【说明生成的目标代码】

32位MIPS汇编。

包括：

addu, addiu,subu, mult, div, mflo, andi,

sw, lw, move, li,la,slt,

j, jal,jr,beq,bne, bltz, blez, bgtz, bgez,sll, syscall

说明：

1. 算术指令选择无符号型的，不检查溢出
2. 为方便起见，使用了move，la，li等伪指令

### 优化方案\*

【说明需要完成的优化方案及其要求】

基本块内部的公共子表达式删除（DAG图）；

全局寄存器分配（引用计数或着色算法）；

数据流分析（通过活跃变量分析，或利用定义-使用链建网等方法建立冲突图）；

其它优化自选；

代码生成时合理利用临时寄存器（临时寄存器池），并能生成较高质量的目标代码；

## 详细设计

【应包括但不限于以下内容】

### 程序结构

【从总体上描述程序的结构，文字或图示均可】

总体上分三个过程。

一、一遍完成词法分析、语法分析、语义分析的工作，词法分析程序作为独立的模块供语法分析程序调用，语法分析和语义分析同时进行，在进行语法分析的过程中同时处理错误。这个过程结束后生成四元式。

二、代码优化得到优化后的四元式。

三、目标代码生成以四元式和符号表作为输入，生成mips汇编代码。

### 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】

关键算法用黄底突出显示。

**Lexer：词法分析类**

public:

//构造函数，参数为源文件的路径

Lexer(string file\_name);

//获取下一个单词

void getsym();

//回退x个单词，x最大值为5，支持连续回退，并且连续回退的数量也不超过5

void rollback(int n);Parser：语法分析类

private:

//返回下一个字符

void getch();

//回退一个字符，只能回退一个

void retract();

Rollback：内部类，维护一个循环队列实现单词的回退，保存最近max个单词的信息，这里max取5

void push(CLASS, string, int);

void get(int i, CLASS&, string&, int&);

**Parser：语法分析类**

递归下降子程序法

public:

//构造函数，传入文件路径，符号表Tabel的引用，中间代码Middle的引用，并创建一个词法分析类lexer对象

Parser(string file,Tabel &t,Middle &m):lexer(file),tabel(t),middle(m){};

//语法分析

int analyze();

private:

//＜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＞

int program();

//＜常量说明＞ :: = const＜常量定义＞; { const＜常量定义＞; }

//进入前向前多读一个

//退出后向前多读一个单词

int const\_desc();

//＜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＞ }

int const\_def();

//＜整数＞::= ［＋｜－］＜无符号整数＞｜０

int integer(int &val);

//＜变量说明＞::= ＜变量定义＞;{＜变量定义＞;} //带有试探性，可能会遇到有返回值的函数定义

//进入前向前多读一个

//退出后向前多读一个单词

int variable\_desc();

//＜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＞‘]’ )}

//进入前多读1个单词

//退出后也向前多读1个单词

int variable\_def();

//＜有返回值函数定义＞::=＜声明头部＞‘(’＜参数＞‘)’ ‘{’＜复合语句＞‘}’

//进入前先读1个单词

int function();

//＜无返回值函数定义＞ ::= void＜标识符＞‘(’＜参数＞‘)’ ‘{’＜复合语句＞‘}’

//进入前先读2个单词

int void\_function();

//＜主函数＞::= void main ‘(’ ‘)’ ‘{’＜复合语句＞‘}’

//进入前先读2个

int main\_function();

//＜参数表＞::=＜类型标识符＞＜标识符＞{,＜类型标识符＞＜标识符＞}| ＜空＞

int parameter\_def();

//＜复合语句＞::= ［＜常量说明＞］［＜变量说明＞］＜语句列＞

int compound\_statements();

//＜语句列＞:: = ｛＜语句＞｝

int statements();

//＜语句＞ ::= ＜条件语句＞｜＜循环语句＞｜‘{’＜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＞;

int statement();

//＜条件语句＞::= if ‘(’＜条件＞‘)’＜语句＞［else＜语句＞］

//进入前先读1个单词

int condition\_statement();

//＜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

int condition();

//＜循环语句＞ ::= while ‘(’＜条件＞‘)’＜语句＞

| for‘(’＜标识符＞＝＜表达式＞;＜条件＞;＜标识符＞＝＜标识符＞(+|-)＜步长＞‘)’＜语句＞

int loop();

//＜有返回值函数调用语句＞ :: = ＜标识符＞‘(’＜值参数表＞‘)’

int call(string&);

//＜无返回值函数调用语句＞ ::= ＜标识符＞‘(’＜值参数表＞‘)’

//进入前先读1个单词

int call\_void();

//＜值参数表＞::= ＜表达式＞{,＜表达式＞}｜＜空＞

//只检查参数个数，不进行类型一致性检查

int parameter\_pass(const string&);

//＜赋值语句＞ ::= ＜标识符＞＝＜表达式＞|＜标识符＞‘[’＜表达式＞‘]’=＜表达式＞

//进入前多读一个单词

int assign();

//＜读语句＞::= scanf ‘(’＜标识符＞{,＜标识符＞}‘)’

int read();

//＜写语句＞::= printf ‘(’＜字符串＞,＜表达式＞‘)’| printf ‘(’＜字符串＞‘)’| printf ‘(’＜表达式＞‘)’

//进入前先读1个单词

//字符型表达式输出字符

int write();

//＜返回语句＞ ::= return[‘(’＜表达式＞‘)’]

//进入前先读1个单词

int ret();

//＜表达式＞:: = ［＋｜－］＜项＞{ ＜加法运算符＞＜项＞ }

Type expression(string&);

//＜项＞::= ＜因子＞{ ＜乘法运算符＞＜因子＞ }

Type term(string&);

//＜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＞‘)’

Type factor(string&);

**Tabel：管理符号表的类**

public:

Tabel();

//填表，成功返回1，名字冲突返回0

//如果插入的是函数名，则自动新建一个符号表

int insert(string name, Kind k, Type t = NULL\_TYPE, int val = 0);

//查找指定种类的名字是否存在，如果指定的种类为UNKNOW，则同时返回种类

//首先在当前符号表中查找，然后在全局符号表中查找，找到返回1，否则返回0

int lookup(string name,Kind &k);

//获取常量的数值

int getValue(string);

//获取函数的参数个数

vector<Type> getParas(const string&);

//查找标识符的类型，可能为INT,CHAR或NULL\_TYPE

Type getType(const string&);

//对局部变量（包括数组）填地址

//参数为寄存器或在栈中相对活动记录的基的偏移量

void setAddr(string&,string);

//查询地址，参数可能是局部变量（包括数组、参数），不存在返回“-2”

string getAddr(string&);

//设置符号表指针cur的值

void setCursor(int);

vector<vector<string>> ident\_lists;

private:

//创建一个新的符号表

void create\_tabel();

//在当前符号表中查找名字

int lookup\_local(string name);

//在全局符号表中查找名字

int lookup\_global(string name);

**Middle：生成中间代码的类**

public:

void new\_function(string name);

//四元式的数量，可以表示下一个四元式的序号

int size() { return MidCodes.size(); };

//生成一般的中间代码

//包括add,sub,mul,dive,minus(MIN),assign(ASS),parameter,call,call\_void,return,scanf,printf,printf\_s,subscript,ass\_arr

void emit(\_operator op, string op1, string op2, string res);

//生成跳转指令的中间代码

//包括J, JE, JNE, JLT, JLE, JGT, JGE

void emit(\_operator op, string op1, string op2);

//生成常量赋值语句,op为ASS\_I

Void emit(\_operator op, string res, int val);

//分配临时变量

string alloc();

//释放临时变量

临时变量的分配算法：

设置一个计数变量，初值是0，每次申请返回计数变量的值，计数变量+1，每当申请者用完后随即释放，释放后计数变量-1。

void release(int);

//设置下标为index的中间代码（跳转指令）的label

//这里暂时用行号代表标号，最后替换成Label+编号

void setLabel(unsigned index, int label);

//指示分析工作结束，完成标号的插入

void end();

**Generator：生成目标代码的类**

Private：

//生成一条指令

void code(Op,string op1="",string op2="",string op3="");

//生成一条数据定义指令

void new\_data(string,string,string value="");

//进入每个函数后先初始化

void initial();

//存储分配

根据参数执行以下操作：

m=-1表示一开始在数据段定义全局变量；

否则，为函数内部的临时变量分配全局寄存器；

在符号表中填写第4个以后的参数的地址；

为局部变量分配全局寄存器；

数组和未分配到全局寄存器的局部变量保存在栈中，同时填符号表；

void alloc\_sto(int);

//加载变量，根据给定的标识符名字返回它的位置（寄存器）

//输入的参数可能是临时变量、局部变量（包括参数，数组）、全局变量

如果是临时变量，可能保存在s寄存器，也可能保存在t寄存器，否则，分配新的临时寄存器；返回该寄存器。

其他，查符号表返回地址

如果地址可能是s寄存器或a寄存器，否则，分配新的临时寄存器；返回该寄存器。

string load(string&);

//给变量分配临时寄存器

首先在临时寄存器分配列表中寻找该名字是否已经分配过临时寄存器，若是，则返回以前的临时寄存器；否则分配一个新的临时寄存器，并将它的值放到该寄存器中。

string alloc\_tr(string&, string addr);

//保存临时寄存器$t1~$t9的值

n=-1：保存所有临时寄存器（默认），在函数调用前，caller需要保存临时寄存器；

n=0,：只保存全局变量，在函数退出前，只需要将保存在临时寄存器中的全局变量的值写回数据段；

n=1：溢出最久未使用的临时寄存器，当没有空闲的临时寄存器，需要溢出。

void save\_tr(int n=-1);

//恢复参数寄存器的值

void restore\_a(int);

//在栈上开辟n个字的空间

void new\_space(int n);

//在栈上释放n个字的空间

void free\_space(int n);

//检查一个地址是否是栈上相对$fp的偏移地址，地址必须是4的倍数

void check\_addr(string);

//获取数组的起始地址

//1.全局数组，在数据段定义；2.局部数据，保存在运行栈

string get\_arr\_addr(string &);

public:

//构造函数，传入Tabel和Middle的引用

//利用四元式和符号表中的信息生成Mips

Generator(Tabel &t, Middle &m) :tabel(t), middle(m) {};

void generate();

//将目标代码输出到文件中

void output();

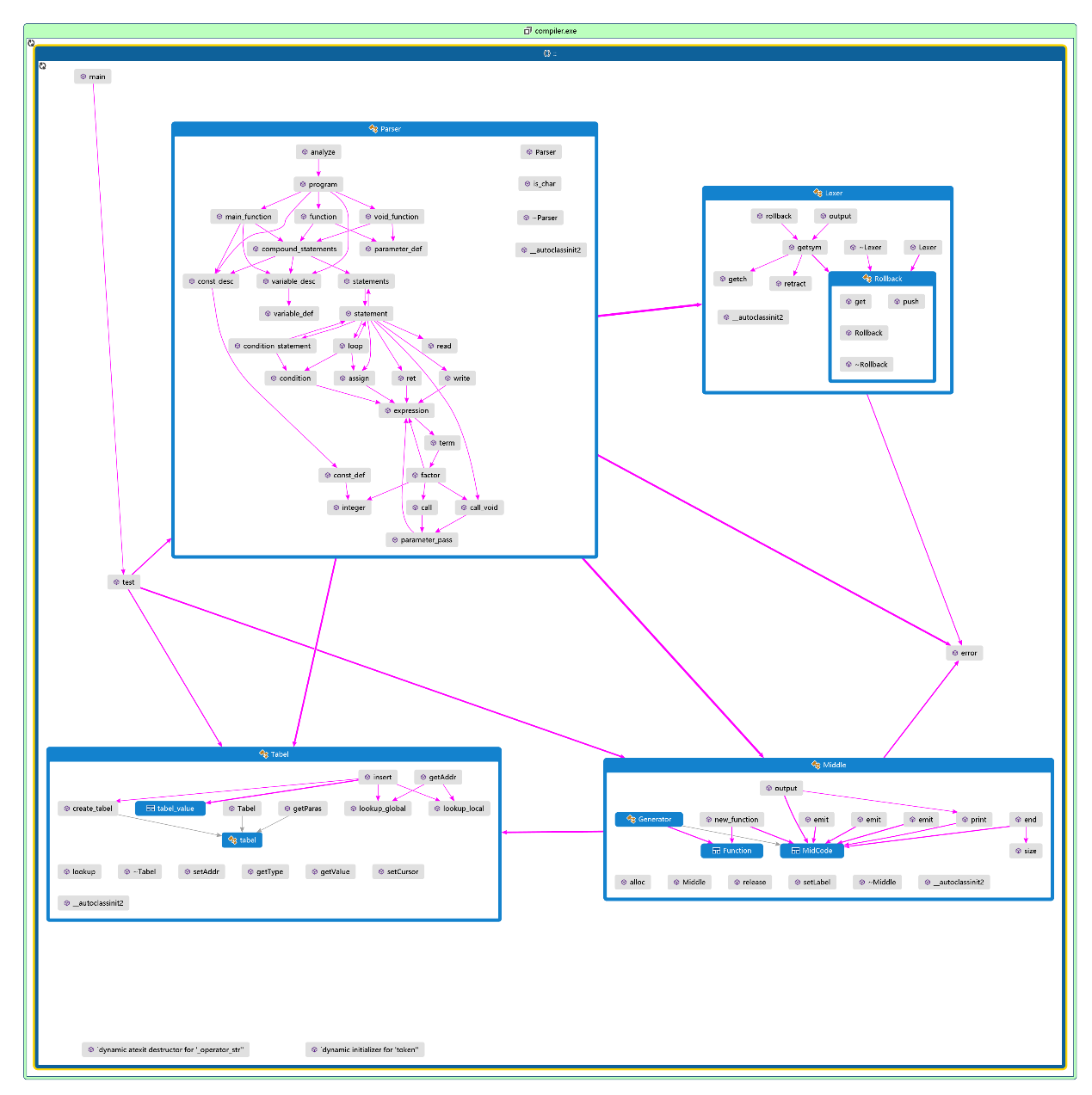
//输出一条目标代码

void print(Instruction,ostream&);

### 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】

Vs2015导出的codemap如下所示，[点击这里查看原图](类图.png)



### 符号表管理方案

【说明符号表的数据结构、管理算法】

#### 相应的数据结构：

//符号表表项的种类，分别代码变量、常量、数组、参数、有返回值函数、无返回值函数、为止

enum Kind

{

VAR, CON, ARR, PARA,FUN, V\_FUN,UNKNOW

};

//符号表表项的类型

enum Type

{

INT, CHAR, NULL\_TYPE

};

struct tabel\_value

{

Kind kind;

Type type;

int value;//可以代表1.常量的值2.函数的私有符号表的索引3.数组的维数

string addr;//寄存器名或相对地址

1.栈中相对帧指针的偏移（如果是前4个以外的参数，则为0,4,8……，否则为负并且是4的倍数，-12，-16，……）

2.保存寄存器或参数寄存器的名称（如a0，s0）

3.-1，表示已经在数据段中定义，名字就是标识符

};

//使用map作为符号表的数据结构，实现快速查找

typedef unordered\_map<string, tabel\_value> tabel;

//符号表

//每个函数一个符号表，下标为0的符号表是全局符号表

vector<tabel> tabels;

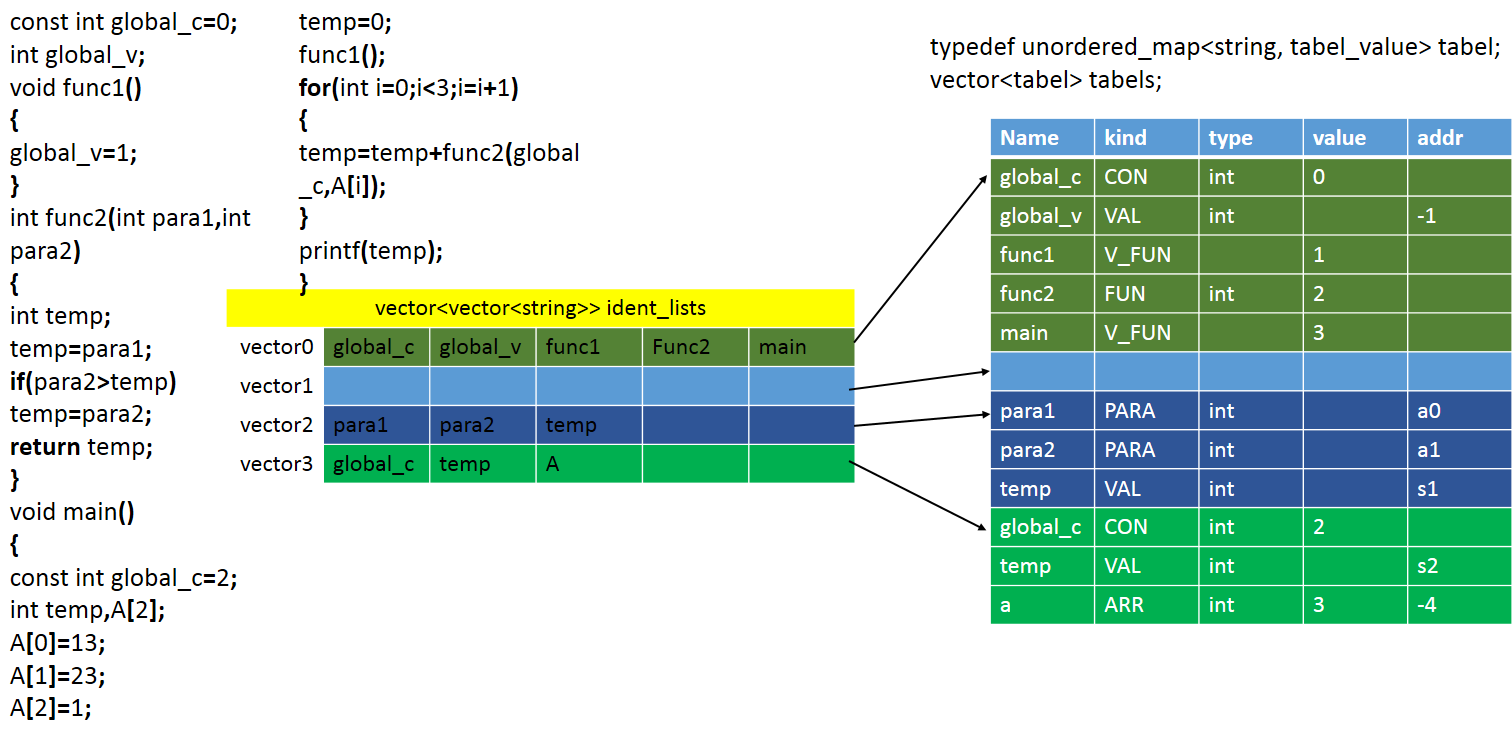
//名字集合

vector<vector<string>> ident\_lists;

#### 管理算法：

1. 使用统一符号表，结构简单，操作方便。有序符号表，按标识符的字典序排序。
2. 由于扩充C0文法的函数不能嵌套，所以由它所定义的语言是非分程序结构语言。因此，不存在定位和重定位操作，但标识符具有全局和局部两种作用域，处理方法如下：
3. 将公共区变量和函数名填入全局符号表。
4. 在函数的声明部分构造局部符号表；在读到一个声明的标识符时，首先要检查该符号表所在的函数的局部符号表，如果表中已存在同名的标识符说明重复声明了标识符，转出错处理；否则，将新的标识符插入表中；
5. 在函数的可执行语句部分读到标识符时，需要查表以判断该标识符是否已声明。首先查标识符所在的函数的局部符号表，如果表中已存在同名的标识符说明该标识符已经声明；否则再查全局符号表，如果仍未找到，说明使用一个未申请的标识符。

#### 示例，[点击这里查看原图](符号表组织方式示例.png)



### 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\*】

这是目标代码生成的难点。

#### 程序运行栈的设计

abp $fp(dynamic link)

|  |
| --- |
| 除前4个以外的参数 |
| 返回地址$ra |
| 全局寄存器 |
| 数组和未分配到全局寄存器的局部变量 |
| 临时变量保存区 |

#### 存储分配过程

1. 程序运行前，先将全局变量保存在数据段中
2. 进入函数时，前4个参数保存在参数寄存器中，全局寄存器依次分配给前4个临时变量（最活跃）和局部变量，数组和未分配到全局寄存器的局部变量、第4个以后的参数保存在栈中，并且在符号表中填地址——寄存器名或相对于$fp的偏移量
3. 运行过程中，临时寄存器在需要时申请：

不管是临时变量、局部变量、参数、全局变量，只要需要，就可以申请使用临时寄存器，因为mips的操作数只能是寄存器或立即数。

动态存储分配的地址保存在哪？

保存在符号表中，动态存储分配的地址是不固定的，因为基地址$fp在变，但是分配地址的代码是固定的，也就是相对$fp的偏移量是固定的，并不会因为递归就把前面的地址覆盖了。

#### Char型的特殊处理

为了处理方便，无论是int型还是char型变量在栈或数据段中都占用一个字的空间。因此，需要考虑char型变量的截断的时机：

* 对char型变量、数组赋值；
* 被调用函数获取char型的参数；
* 调用函数获取char型的返回值；

#### 临时变量

临时变量的信息不保存在符号表中，既参与临时寄存器的分配，也参与全局寄存器的分配。当临时寄存器中的临时变量被溢出或函数调用发生时保存临时寄存器的值，临时变量才会出现在栈中。

#### 全局变量

不能被保存在全局寄存器中，需要在数据段分配永久空间。在运行过程中动态加载到临时寄存器中，函数退出前，保存在临时寄存器中的全局变量的值需要写回数据段中。

#### 字符串的处理

按照文法，字符串只会出现在输出语句中，用MIPS指令如何输出字符串？系统调用。Mips提供了打印字符串常量的系统调用，需要提供起始字符的地址和长度。

语发分析的时候字符串又保存在什么地方？全局数据段。

#### 数组的处理

数组保存在哪？文法中的数组仅仅是一维数组，没必要建模板。分全局和局部两类，局部数组直接保存在运行栈中。全局数组保存到数据段。

### 解释执行程序\*

【说明解释执行程序的数据结构，关键算法，及解释执行过程】

不要求解释执行。

### 四元式设计\*

【对采用的四元式进行详细说明】

生成四元式便于做优化处理。中间代码不必考虑机器的特性，但是四元式设计越接近目标指令，则后续的工作越容易。

关于四元式的一个显然的结论：

声明部分是不用生成四元式的，只需填符号表，包括全局声明和函数的参数声明与局部变量声明。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 操作符 | 操作数1 | 操作数2 | 结果 | 说明 |
| 算术运算 | | | | |
| ADD | op1 | op2 | res | res=op1+op2 |
| SUB | op1 | op2 | res | res=op1-op2 |
| MUL | op1 | op2 | res | res=op1\*op2 |
| DIVE | op1 | op2 | res | res=op1/op2 |
| MIN | op1 |  | res | res=-op1 |
| 跳转 |  |  |  |  |
| J |  |  | label | 无条件跳转到label处 |
| JE | op1 | op2 | label | if(op1==op2) goto label |
| JNE | op1 | op2 | label | if(op1!=op2) goto label |
| JLT | op1 | op2 | label | if(op1<op2) goto label |
| JLE | op1 | op2 | label | if(op1<=op2) goto label |
| JGT | op1 | op2 | label | if(op1>op2) goto label |
| JGE | op1 | op2 | label | if(op1>-op2) goto label |
| 赋值 |  |  |  |  |
| ASS | op1 |  | res | res=op1 |
| ASS\_I | val |  | res | 将整数val赋给res |
| ASS\_ARR | op1(arr) | op2(index) | res(value) | 对数组arr赋值，下标为index，值为value |
| SUBSCRIPT | op1(arr) | op2(index) | res | 对数组arr做下标运算，下标为index，结果送res |
| 函数调用 |  |  |  |  |
| PARAMETER | op1 |  |  | 将op1作为参数传递 |
| CALL\_VOID | op1 |  |  | 调用无返回值函数op1 |
| Call | op1 |  | res | 调用有返回值函数，结果送res |
| RETURN | op1 |  |  | 将op1作为返回值，op1可能为空 |
| SCANF | op1 |  |  | 从控制台读入数到op1中 |
| PRINTF\_S | op1 |  |  | op1为字符串 |
| PRINTF\_C | op1 |  |  | op1为字符型表达式 |
| PRINTF | op1 |  |  | 将op1（表达式）输出到控;制台 |
| LABEL | op1 |  |  | 插入一个label |

### 优化方案\*

【说明代码优化有关的数据结构、关键算法】

**全局寄存器的优化**

根据我的临时变量分配策略，很显然，函数内部最活跃的是t0，其次是t1（如果用到）。因此我的分配算法比较特别：

1. 前4个临时变量（如果有）优先分配；
2. 顺序分配局部变量；
3. 如果还有未使用的全局寄存器并且还有未分配的临时变量，则将全局寄存器分配给这些临时变量。

**临时寄存器的优化——临时寄存器池**

临时寄存器资源：

**$8-$15 $t0-$t7**

**$24-$25 $t8-$t9**

相关数据结构：

//临时寄存器分配列表，保存名字和它对应的临时寄存器名

deque<pair<string, string>> tr\_list;

//溢出的临时变量列表，保存临时变量名和它在栈上的地址（偏移量）

map<string, int> t\_overflow;

//执行寄存器分配的方法

string Generator::alloc\_tr(string &id, string addr);

分配算法：

1. 首先在临时寄存器分配列表中寻找该名字是否已经分配过临时寄存器，若是，则将该名字移到队头，并返回以前的临时寄存器；
2. 否则分配一个新的临时寄存器

如果没有可用的临时寄存器，则先溢出最久未使用的临时寄存器（队尾的寄存器）。新的变量获得一个临时寄存器，根据该变量的种类（addr）执行以下操作：

* 1. 全局变量：将数据段中的值取到寄存器中；
  2. 临时变量：该临时变量可能被溢出到栈上了，有关信息保存在t\_overflow中，重新加载到寄存器；如果在t\_overflow中没有，说明是一个新的临时变量。
  3. 局部变量：将栈上的值取到寄存器中。

将该名字与所在的寄存器信息插入队头中。

### 出错处理

【说明出错处理方案、错误信息及含义】

对于错误处理，是编译器设计中的一大难题，也是衡量编译器质量好坏的一个重要指标，即使是vs也很难做到令人满意，因此我这里只保证准确的报告错误的位置和信息，继续编译或者终止视情况而定。在实践中发现，如果加入大量处理错误的功能，代码将变得混乱不堪，难于调试。

遇到常见的语法错误则改正，采用的方法是自动补全，使用这种处理方式的有以下几种情况：

1. 缺少右括号则继续编译，包括(,[,{，缺少左括号停止编译
2. 缺少,;'"继续编译
3. 具体情况具体分析，通过预读作出判断

1) 函数调用的参数匹配

2) 使用未定义的标识符

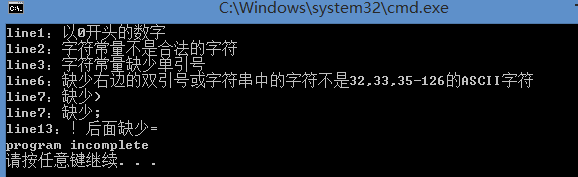
3) 参数说明缺少类型说明符

4) 函数中间某处出现常量或变量定义

5) 变量定义之间的;误写成,，变量说明之间的;误写成,或缺少;

示例：

词法分析阶段识别的错误



语法分析阶段识别的错误



其他情况并不跳过错误，而是直接终止编译。因为即使跳过错误，出错的情况太多，完全无法预料，以致于会产生一连串莫名其妙的错误。

另外，数组越界放在了运行时动态检查，在生成目标代码时生成有关指令，检查下标是否越界。

我没有采用错误代码标志不同的错误信息。

### 函数调用规则

根据寄存器规范，函数调用的流程如下：

**调用方caller**

调用前

1. $t0-$t9:在函数调用发生时写回相应的内存空间。任何函数在任何时候都可以更新这些寄存器中的值，Caller 如果在调用完Callee后还需要用到这些寄存器中的值，就需要在调用Callee前将这些值保存在自己的栈空间内。
2. $a0-$a3:这几个都是易变的参数寄存器，只要调用了其他有参函数或需要参数的系统调用后还需要用到这些寄存器中的值，就需要在调用Callee前将这些值保存在自己的栈空间内
3. 传参：前4个依次放入$a0~$a3，超过4个的压入栈中

jal...

调用后

1. 如果参数超过4个，先释放栈中的参数值
2. $a0-$a3:恢复
3. $t0-$t9:恢复他们的值
4. $v0:如果是有返回值函数调用，则在jal指令后保存最新的返回值。

**被调用方callee**

进入后

1. 保存$fp，$fp即将指向新的活动记录的基

sw $fp,-4($sp) # push old frame pointer (dynamic link)

1. 设置$fp的值，同时保存$sp

move $fp, $sp # frame pointer now points to the top of stack

1. 保存$ra，只要调用了其他函数，就会改变，jal 会自动更改这个寄存器的值。

subu $sp, $sp, 500 # allocate say 500 bytes in the stack

# (for frame size = 500)

sw $ra, -4($fp) # save return address in frame

1. $s0-$s7: 进入函数时统统保存，退出前再恢复其值。
2. 分配寄存器和栈空间

退出前

1. $s0-$s7: callee 由于各种原因很有可能改变了这些值, 所以必须在返回之前将这些寄存器的原始值恢复
2. 恢复$ra、$sp、$fp的值

lw $ra, -4($fp) # restore return address

move $sp, $fp # pop frame

lw $fp, ($fp) # restore old frame pointer (follow dynamic link)

1. 返回

jr $ra # return

### 改进

由于各种原因还有很多地方需要改进，一方面是检查语义约束，另一方面是进行优化。

**语义检查：**

1. 不能对常量赋值；
2. 变量先赋值后使用
3. 有返回值函数必须有返回值，如果程序有分支结构，那么判断是否有return语句将变得很复杂

**优化：**

1. 变量在初次被加载到临时寄存器时无需对寄存器赋值，因为此时他还没有值
2. 计算临时变量数的方案不妥，临时变量数会比实际使用的少，以致于浪费全局寄存器。
3. $a0~$a3

main函数在使用参数寄存器不需要保存与恢复

连续使用参数寄存器只需要一次保存和一次恢复

避免下面的冗余工作

sw $a0,($sp)

move $a0,$a0

1. 被调用函数只需要保存它将要使用的s寄存器，没必全部要都保存
2. 运行时的溢出检查
3. 相同的字符串只需在数据段中定义一次

## 操作说明

### 运行环境

【说明搭建运行环境的步骤】

操作系统：windows 7及以上

开发工具：Microsoft Visual Studio 2010 professional

### 操作步骤

【详细说明操作步骤】

在VS2010里面直接运行，在控制台中输入源文件的路径，回车，在当前目录中生成文件名为mips.S的汇编代码。在mars中打开即可直接运行。

## 测试报告

### 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，无需截屏】

1. **（正确）**

void main()

{

int a;

char c;

a = 97;

c = 'a';

printf("a=");

printf(a);

printf("\nc=", c);

}

本程序重点测试了输出语句：

＜写语句＞::= printf ‘(’＜字符串＞,＜表达式＞‘)’| printf ‘(’＜字符串＞‘)’| printf ‘(’＜表达式＞‘)

后面所有的测试都是通过输出语句检验结果的正确性，所以首先测试printf。

程序的功能是输出字符串、整数和字符，结果正确。

1. **（正确）**

void main()

{

int i1, i2, i3;

char c1, c2;

scanf(i1, c1, i2, c2, i3);

printf("\n", i1);

printf(c1);

printf(i2);

printf(c2);

printf(i3);

}

本程序重点测试了输入语句，结果正确。

1. **（正确）**

void main()

{

int a;

a = 0;

if (1 == 1)

a = a + 1;

if (1 == 2)

a = a - 1;

if (1 == 1)

a = a + 1;

else a = a - 1;

if (1 != 1)

a = a - 1;

else a = a + 1;

if (a == 3)

printf("success");

else

printf("fail");

}

本程序重点测试了条件语句：

＜条件语句＞ ::= if ‘(’＜条件＞‘)’＜语句＞［else＜语句＞］

程序的功能是输入一个数，输出其斐波那契数，结果正确。

1. **（正确）**

void main()

{

int i, sum;

i = 0;

sum = 0;

while (i <= 100)

{

sum = sum + i;

i = i + 1;

}

printf(sum);

sum = 0;

for (i = 0; i <= 100; i = i + 1)

sum = sum + i;

printf(sum);

}

本程序重点测试了循环语句：

＜循环语句＞ ::= while ‘(’＜条件＞‘)’＜语句＞| for‘(’＜标识符＞＝＜表达式＞;＜条件＞;＜标识符＞＝＜标识符＞(+|-)＜步长＞‘)’＜语句＞

程序的功能是分别用while和for循环计算1+2+…+100的值，结果正确。

1. **（正确）**

void main()

{

int a;

a = 0;

if (1 == 2)

a = a + 1;

if (1 != 1)

a = a + 1;

if (1<1)

a = a + 1;

if (1 <= 0)

a = a + 1;

if (1>1)

a = a + 1;

if (1 >= 2)

a = a + 1;

if (a == 0)

printf("success");

}

本程序重点测试了条件判断，结果正确。

1. **（正确）**

int G\_A[7];

void main()

{

int A[10];

G\_A[2] = 2;

A[1] = 1;

G\_A[1] = A[1];

A[2] = g\_a[2];

printf(g\_A[1]);

printf(g\_A[2]);

printf(A[1]);

printf(A[2]);

}

本程序重点测试了数组，包括对数组元素的赋值，使用数组元素的值，结果正确。

1. **（正确）**

int M, N;

char C;

void fn()

{

N = 99;

printf("fn:");

printf(M);

printf(C);

M = 100;

C = 'a';

}

void main()

{

M = -100;

C = 'z';

fn();

printf("\nmain:");

printf(M);

printf(N);

printf(C);

}

本程序重点测试了全局变量，同时还有函数调用等。

程序的功能是在函数间共享全局变量，函数在结束前除需要恢复s寄存器中的值以外还需要将临时寄存器中的全局变量写回数据段，结果正确。

1. **（正确）**

int fn1(int a)

{

printf("I am fn1\n");

return (a + 1);

}

int fn2(int a)

{

int b;

b = fn1(a);

printf("I am fn2\n");

return (b + 1);

}

void main()

{

int b;

b = fn2(0);

if (b == 2)

printf("success\n");

}

本程序重点测试了函数调用与返回。

结果正确。

1. **（正确）**

void main()

{

int a, b;

a = 0; b = 1;

printf(a);

printf(b);

printf(1 + (2 + (3 + (4 + (5 + (6 + (7)))))));

}

本程序重点测试了多个临时变量的情况。

共产生7个临时变量，前6个分配到的全局寄存器依次是s0，s1，s2，s3，s6，s7，第7个使用临时寄存器。结果正确。

1. **（正确）**

void fn(int i)

{

printf(i);

}

void fn2(int a, int b, int c)

{

printf(a + b + c);

}

int fn3(int a, int b, int c, int d, int e, int f)

{

printf(a);

printf(b);

printf(c);

printf(d);

printf(e);

printf(f);

}

void main()

{

fn3(1, 2, 3, 4, 5, 6);

}

本程序重点测试了参数传递，超过4个参数的传递，压入栈。

共有6个参数，前次依次放入参数寄存器a0,~a3，后两个压入栈，结果正确。

1. **（正确）**

void main()

{

int a, b, c;

char c1, c2, c3;

int d, e, f, g;

a = 1; b = 2; c = 3;

c1 = 52; c2 = 53; c3 = 54;

d = 7; e = 8; f = 19; g = 10;

f = 9;

printf(a); printf(b); printf(c);

printf(c1); printf(c2); printf(c3);

printf(d); printf(e); printf(f); printf(g);

}

本程序重点测试了多个局部变量的情况，未分配到寄存器的局部变量保存在栈上。

共有10个局部变量，一个临时变量，前7个局部变量依次分配到的全局寄存器是s1~s7，后面3个局部变量保存到栈上，结果正确。

1. **（正确）**

char fn(int a, char b, char c)

{

printf(a);

printf(b);

printf(c);

return (a\*b\*c);

}

void main()

{

char c;

char A[5];

c = 1684234849;

A[1] = 1684234849;

printf(c);

printf(A[1]);

printf(fn(10000, 1684234849, 1701077858));

}

本程序重点测试了字符的截断。

Int型向Char型的转换发生在赋值，传参与返回值。

1. **（正确）**

int m, n, o, p;

void fn()

{

int a, b, c, d, e;

e = 0; m = 1; n = 2; o = 3; p = 4;

printf(1 + (2 + (3 + (4 + (5 + (6 + (7 + (8 + (9 + (10))))))))));

printf(e);

}

void main()

{

fn();

printf(m);

printf(n);

printf(o);

printf(p);

}

本程序重点测试临时寄存器的分配和溢出。

当第5行结束时，临时寄存器中依次保存emnop，第6行执行到9+(10)之前时临时寄存器已满，此时进行溢出，淘汰最久未使用的e，写会栈，当执行到第6行的(10）时，溢出m，写会数据段中，结果正确。

1. **（正确）**

int M;

int fn()

{

return (8);

}

void main()

{

int a, b, c, d, n;

n = 7;

M = 5;

printf(1 + (2 + (3 + (4 + (M + (6 + (n + (fn()))))))));

}

本程序重点测试调用函数前保存所有临时寄存器。

程序的功能是计算1+2+……+8，结果正确。

1. **（正确）**

void main()

{

int a, b, c, d, e;

int arr[5];

int i, j, temp, k;

printf("please enter 5 integer");

scanf(a, b, c, d, e);

arr[0] = a; arr[1] = b; arr[2] = c; arr[3] = d; arr[4] = e;

for (i = 0; i<5; i = i + 1)

{

temp = arr[i]; k = i;

for (j = i; j<5; j = j + 1)

if (arr[j]<temp)

{

k = j;

temp = arr[j];

}

arr[k] = arr[i];

arr[i] = temp;

}

for (i = 0; i<5; i = i + 1)

printf(arr[i]);

}

本程序是综合测试。

程序的功能是输入5个整数，求输出从小到大排序的结果。

1. **（正确）**

int min(int var1, int var2, int var3)

{

int min;

min = var1;

if (var2<min)

min = var2;

if (var3<min)

{

min = var3;

}

return (min);

}

void main()

{

int flag;

char t;

int a, b, c, i;

a = 0;

flag = 1;

while (flag == 1)

{

scanf(t);

if (t >= '0')

if (t <= '9')

a = a \* 10 + t - '0';

else flag = 0;

else flag = 0;

}

scanf(b);

b = -(-b\*b / b) - 0;

c = 0;

for (i = 0; i != 10; i = i + 1)

c = c + i;

printf("result is ");

printf(min(a, b, c));

printf("result is ", min(a, b, c));

}

本程序是综合测试，包括输入，输出，循环等等。

程序的功能是，从控制台输入两个整数，与1+2+……+9的结果三个数求最小值，结果正确。

1. **（正确）**

int max(int a, int b, int c, int d, int e, int f, int g)

{

int max;

max = a;

if (b>max)

max = b;

if (c>max)

max = c;

if (d>max)

max = d;

if (e>max)

max = e;

if (f>max)

max = f;

if (g>max)

max = g;

return (max);

}

void main()

{

int a, b, c, d, e, f, g;

a = 1;

b = a + 1;

c = a + b;

d = b\*b;

e = d - 1;

f = d / b;

g = -f;

printf(max(a, b, c, d, e, f, g));

}

本程序是综合测试，包括表达式，条件判断等。

程序的功能是，求7个数的最小值，结果正确。

1. **（错误）**

void main()

{

const int const1 = 01;

const char const2 = ''';

const char const3 = '(';

const char const4 = '0;

printf("");

printf("hello);

if (1!2)

printf("right");

。

printf("end");

}

本程序重点测试了词法分析阶段能跳过的错误。

1. **（错误）**

const int M1 = 12

int v1

int v2, v3

int A[12;

void fn1(

{}

int fn2()

{

int fn3(a)

{}

void main()

{

v1 = 1

for (v1 = 2 v1<v2 v1 = v1 + 1;

fn100();

fn3(v1, v2);

fn3();

id = 100;

unknow[10] = 100;

A[0 = 0;

scanf(M1);

scanf(M2);

scanf(v1;

printf("hell";

return (v1;

printf("end");

}

本程序重点测试语法分析阶段能跳过的错误。

1. **（错误）**

void main()

{

char A[10];

int b;

A[10] = 97;

b = A[10];

printf(b);

}

本程序重点测试了数组越界检查。

程序在运行时检测是否越界。

### 测试结果分析

【说明测试程序对语法成分的覆盖情况】

事实上上面的测试程序仅仅是针对最终的目标代码生成模块的测试，每当完成一个阶段的任务时我都进行局部测试，比如专门针对词法分程序、语法分析程序、错误报告、符号表的测试。

1. 语法分析程序部分

对于语法成分，那是100%的覆盖，例如对表达式的测试如下，通过查看符号表或观察生成的四元式判断结果是否正确。

＜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＞‘)’

因子

int a,b,A[10];

int fn()

{

return (97);

)

void main()

{

b=97;

a=b;

A[2]=97;

a=A[2];

a='a';

a=fn();

a=(97);

}

项

void main()

{

int a,b,t;

t=a;

t=a\*b;

t=a/b;

t=a\*b/10;

t=3\*b\*b/60/12;

}

表达式

void main()

{

int a,b,e;

e=a;

e=+a;

e=-a;

e=a+b;

e=a-b;

e=a+b-10;

e=a-10+b+10;

}

1. 错误报告

错误报告的测试几乎无死角。

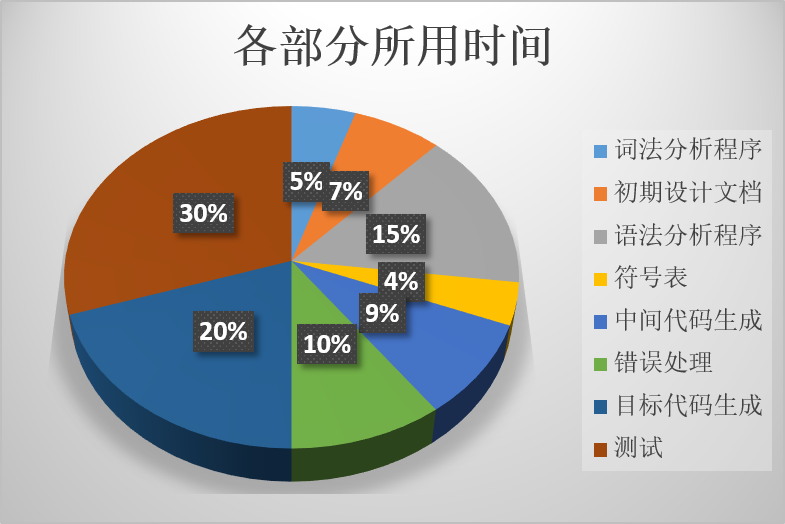
1. 目标代码

最终的mips指令的正确性则由上面给出的测试用例保证。

## 总结感想

【说明在完成课程设计中的收获、认识和感想】

我毫不犹豫的选择了高组的题目，但遗憾的是因为时间原因没能完成要求的代码优化。虽然是一个再简陋不过的编译器，但却是到目前为止从头到尾由我自主开发的最为复杂、庞大的程序了。想想都有点小激动。我用了一周的时间集中编程，不算太长，没有想象中的那么煎熬，简单说也就是编写一个稍微复杂些的程序。完成各部分任务所用时间分布大概如下所示，可以看到其中大部分的时间是测试。



这个程序完成得还算顺利主要是得益于计组上学过Mips，这也是为什么我选高组的一个重要原因，如果要求生成X86的汇编，那就不知道能不能做得完了。

对编译实验的感悟：

1. 在设计编译器的过程中最大的收获是加深了对编译原理课上所学知识的理解。从一开始的完全摸不着头脑，到反复研读课本后豁然开朗，才发现其实课堂上讲就是马上要用的东西，虽然一开始听不太懂。这也印证了计算机这门学科边学边用的特点。
2. 这个实验让我深感面向对象编程的方便，一开始我并不打算用类，但是随着程序的增长，越来越无法控制，我就本着最基本的想法，把与相应数据相关的操作封装在一起形成一个个类，使得整个程序的逻辑结构清晰，各部分的分工明确。
3. 我也从单元测试中受益匪浅，我并不是严格的用vs对每个函数自动生成单元测试，使用断言来判断执行的正确性，而是将测试局部功能的测试用例作为整个程序的输入，看最终的运行情况，但效果与单元测试类似。这让我更明确的测试各个不同的地方，以快速发现错误。
4. 我也体会到了设计文档的重要性，以前一直觉得设计文档是个累赘，总是在开发完成后再来撰写，现在看来，设计文档非常有用，尤其是能提高编程的效率。我首先在文档中理清设计方案，然后参照文档实现，如有不合适的地方，再逐步修改文档。

编译原理的重要性可能要等到以后真正用到了这些技术才能慢慢体会得到，据了解大部分系统软件和应用软件的开发，通常要用到编译的原理和技术。设计词法分析器的串匹配技术已用于正文编辑器、信息检索系统和模式识别程序；上下文无关文法和语法制导定义已用于创建诸如排版、绘图系统和语言结构化编辑器中，代码优化技术已用于程序验证器和从非结构化的程序产生结构化程序的编程之中。

注：【】内的文字为文档模板说明，完成的作业中需去掉。

标\*的章节需根据题目的难度进行取舍。