《编译技术》课程设计文档

学号：14061213

姓名：杨森

2016年 12月 30日

## 一．需求说明

### 1．文法说明

#### 获取的文法

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

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

#### 文法改写

<程序> ::= ［<常量说明>］［<变量说明>］{(int|char) <标识符> <有返回值函数定义>| void <无返回值函数定义>} void <主函数>

<有返回值函数定义> ::= '(' <参数表>‘)’ ‘{’<复合语句>‘}’

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

<主函数> ::= main‘(’‘)’ ‘{’<复合语句>‘}’

<复合语句> ::=  ［<常量说明>］［<变量说明>］<语句列>

<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>}

<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>‘]’) }

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

<整数> ::= ［+｜-］<无符号整数>｜０

<语句> ::= <条件语句>｜<do循环语句> | <for循环语句>｜‘{’<语句列>‘}’｜<函数调用语句>;｜<赋值语句>;｜<读语句>;｜<写语句>;｜<空>;｜<返回语句>;

<条件> ::=  <表达式><关系运算符><表达式>

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

<do循环语句> ::=  do<语句>while ‘(’<条件>‘)’

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

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

<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>‘)’]

<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>‘)’

### 2．目标代码说明

.data

# 全局变量及全局数组声明

# 字符串常量声明

.text

move $fp, $sp

j end

divi0error:

li $v0 4

la $a0 divi0errorMessage

syscall

j endOfAll

arrayOutOfRange:

li $v0 4

la $a0 arrayOutOfRangeMeaage

syscall

j endOfAll

# 代码段

.

.

.

main:

.

.

.

end:

jal main

endOfAll:

具体的目标代码为 MIPS 指令集的一个子集:

la $r, a # 将数据区的数据的首地址加载到寄存器$r

lw $r, a # load a word at address a into register $r

sw $r, a # store a word in register $r at address a

sb $r, a # store a byte in register $r at address a

add $t1, $t2, $t3 # Addition with overflow: set $t1 to ($t2 p$t3)

sub $t1, $t2, $t3 # Subtraction with overflow: set $t1 to ($t2 plus + $t3)

mul $t1, $t2, $t3/im # Multiplication, set register HI to high 32 + bits, register LO to low 32 bits

div $t1, $t2, $t3/im # Division, set HI to reminder, LO to quotient

neg $t1, $t2 # set $t1 to the negation of $t2

jal target # set $ra to return address, jump to target

j target # jump to target

jr # jump $ra

beq $t1, $t2/im, target # 如果$t1等于$t2, 跳转到target

bne $t1, $t2/im, target # 如果$t1不等于$t2, 跳转到target

bge $t1, $t2/im, target # 如果$t1 >= $t2, 跳转到target

bgt $t1, $t2/im, target # 如果$t1 > $t2, 跳转到target

ble $t1, $t2/im, target # 如果$t1 <= $t2, 跳转到target

blt $t1, $t2/im, target # 如果$t1 < $t2, 跳转到target

bnez $t1 label # 如果$t1 != 0, 跳转到label

beqz $t1 label # 如果$t1 == 0, 跳转到label

move $t1, $t2 # set $t1 to $t2

li $t1, im # set $t1 to im

syscall # 调用系统服务

comment # 向目标代码中写入注释

### 3. 优化方案\*

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；

## 二．详细设计

### 1．程序结构

### 2．类/方法/函数功能

源程序

MIPS汇编

目标程序

词法分析

语法分析

语义分析

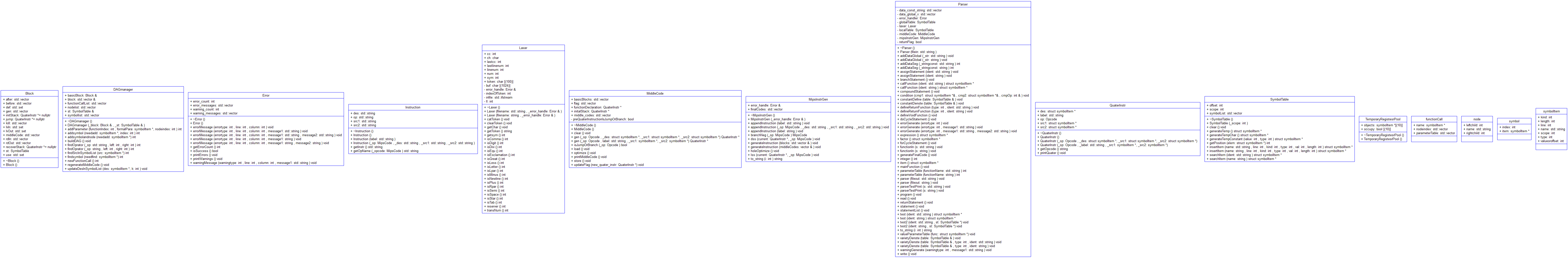
中间代码分析

目标代码生成

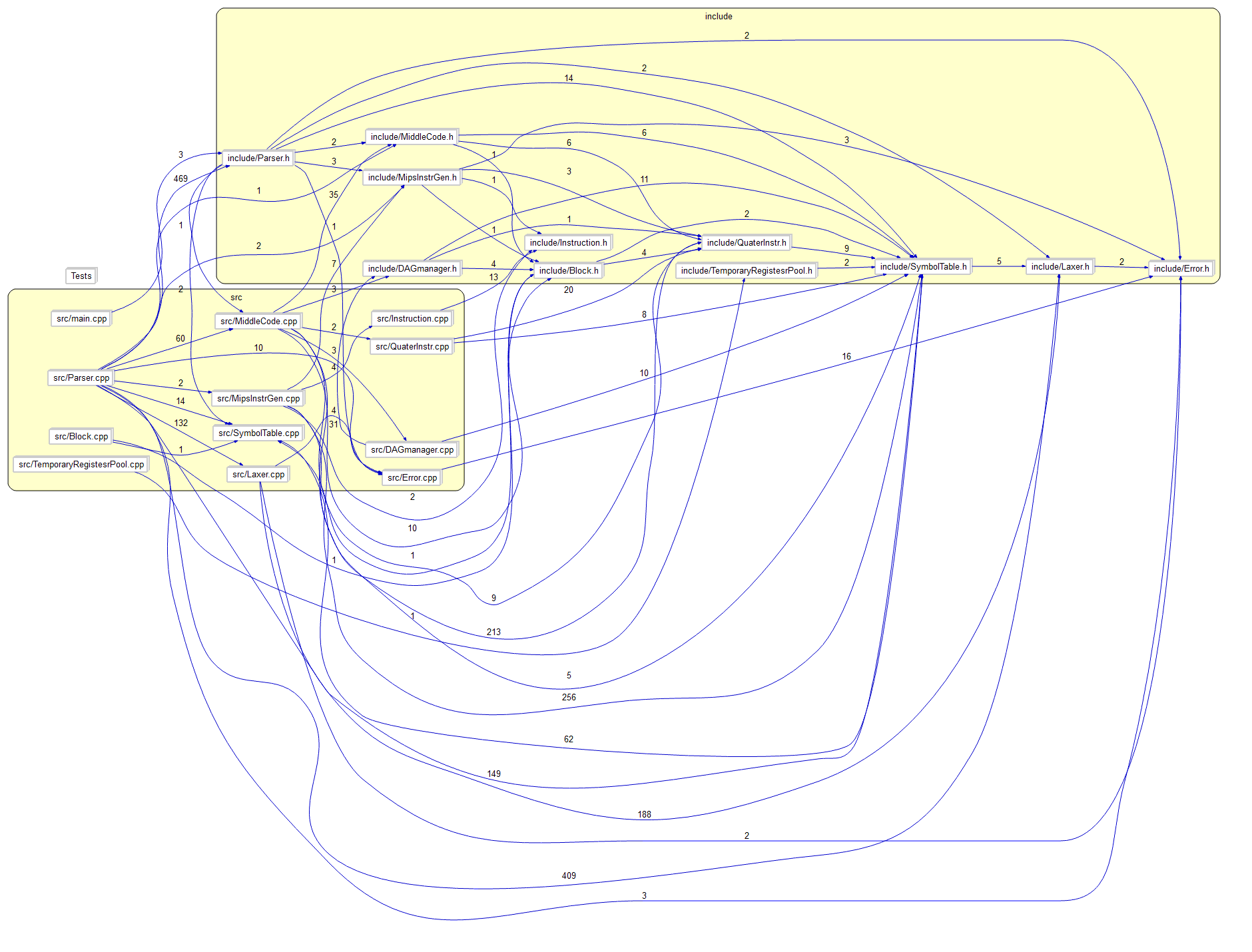
代码优化

分析

综合



### 3．调用依赖关系



### 4．符号表管理方案

#### 符号表项



#### 符号表存储管理类

class SymbolTable

{

public:

SymbolTable(int \_scope); // 构造函数

virtual ~SymbolTable(); // 析构函数

struct symbolItem\* searchItem(std::string ident);

struct symbolItem\* insertItem(std::string name, int line, int kind, int type, int val = 0, int length = 0);

struct symbolItem\* generateTemp(); // 生成临时变量

struct symbolItem\* generateTempChar(); // 生成临时字符类型变量

struct symbolItem\* generateTempConstant(int value, int type); // 生成临时常量

int getPosition(struct symbolItem\* elem); // 获取元素的下标, 为找到函数的参数

void clear(); // 清空符号表

std::vector<struct symbolItem\*> symbolList;

int offset;

int scope;

protected:

private:

};

管理算法:

1. 在程序的常量声明部分, 将常量填入全局符号表; 在程序的变量声明部分, 将变量填入全局符号表. 在填之前先检查全局符号表中是否有同名符号, 如果有, 抛出重复声明错误; 如果没有, 才填.
2. 遇到函数定义, 将函数名填入全局符号表. 在填之前先检查全局符号表中是否有同名函数, 如果有, 抛出函数重复定义错误; 如果没有, 才填.
3. 进入一个函数, 构造一个局部变量表, 将所有参数填入局部符号表; 在函数的常量声明部分, 将常量填入局部符号表, 在变量声明部分, 将变量填入局部符号表. 在填之前先检查全局符号表中是否有同名符号, 如果有, 抛出重复声明错误; 如果没有, 才填. 在 语句列 部分, 如果遇到一个标识符, 先查找局部符号表, 如果查到, 则说明该标识符已声明; 如果没有查到, 在查找全局符号表, 如果查到, 则说明该标识符已声明, 否则抛出标识符未声明错误.
4. 当程序编译完成之后, 应将全部符号表都释放掉.

### 5．存储分配方案

静态存储区: 存储全局变量, 编译时分配地址和大小.

由于C0语言的简化, 动态内存管理只需管理运行栈, 不需要管理堆.

动态存储区(运行栈):

|  |
| --- |
| 局部数据区 |
| 参数区 |
| 保存现场 |
| prevfp(指向前一个调用栈的基地址) |
| 返回地址 |

运行时动态管理, 当进入新函数后, 创建新的调用栈, 返回地址$ra进栈, $fp进栈, $fp指向当前栈顶$sp+8(当前栈帧基地址), 将$s0~$s7寄存器依次入栈, 根据符号表, 将参数一次压栈, 再将变量依次压入栈中, 根据函数内容, 若临时变量过多, 寄存器不够用, 此时也需要使用栈, 根据具体情况入栈出栈. 在函数返回时, 恢复现场, $ra从栈帧中得到返回地址, $fp从栈帧中得到上一个栈帧的基地址, 销毁当前栈帧(事实上只是将$sp指向$fp).

### 6. 四元式设计\* (28个)

+ 加法: ADD des, src1, src2

+ 减法: SUB des, src1, src2

+ 乘法: MUL des, src1, src2

+ 除法: DIV des, src1, src2

+ 取数组中的值 LAV des, array\_name, index

+ 向数组中存值 SAV src, array\_name, index

+ 取反: NEG des, src

+ 赋值: ASS des, src

+ 大于: BGT label, src1, src2

+ 大于等于: BGE label, src1, src2

+ 小于: BLT label, src1, src2,

+ 小于等于: BLE label, src1, src2

+ 等于: BEQ label, src1, src2

+ 不等于: BNE label, src1, src2

+ 不等于0: BNZ label, src

+ 等于0: BEZ label, src

+ 无条件跳转: JUMP label

+ 设置标签: SET label

+ 函数调用: CALL function\_name

+ 进入函数定义: DEC func

+ 函数返回操作-无返回值: RET 函数返回操作-有返回值: RET src

+ 下降栈顶 DSP label(a number) 进入函数 和 进入基本块使用

+ 上升栈顶 USP label(a numner) 基本块结束使用

+ 从标准输入读: READ des

+ 向标准输出写: WRITE src

+ 往栈里压参数 PUSH src, index

+ 取$v0的值放到一个临时变量里 PUT des

### 7. 目标代码生成方案\*

#### 数据结构

#ifndef MIPSINSTRGEN\_H

#define MIPSINSTRGEN\_H

#include <string>

#include <vector>

#include "QuaterInstr.h"

#include "Block.h"

#include "Error.h"

#include "Instruction.h"

enum Register

{

$zero,

$at,

$v0,

$v1,

$a0,

$a1,

$a2,

$a3,

$t0,

$t1,

$t2,

$t3,

$t4,

$t5,

$t6,

$t7,

$s0,

$s1,

$s2,

$s3,

$s4,

$s5,

$s6,

$s7,

$t8,

$t9,

$k0,

$k1,

$gp,

$sp,

$fp,

$ra,

};

class MipsInstrGen

{

public:

MipsInstrGen(Error &\_error\_handle) : error\_handle(\_error\_handle) {};

virtual ~MipsInstrGen();

MipsCode branchNeg(MipsCode \_op); // 分支指令取反

void appendInstruction(std::string label); // 生成final code

void appendInstruction(MipsCode \_op, std::string \_des = "", std::string \_src1 = "", std::string \_src2 = "");

void generateInstruction(std::vector<Block \*> &blocks); // 以基本块为单位生成目标代码

void generateInstruction(std::vector<QuaterInstr\*>& middleCodes); // 根据中间代码生成目标代码

void dss(QuaterInstr\* current, MipsCode \_op); // dss 型四元式 des src src

void lss(QuaterInstr\* current, MipsCode \_op); // lss 型四元式 label src src

std::string to\_string(int i); // 修复code blocks g++ (tdm-1) 4.7.1 bug

void holeOptimize(); // 窥孔优化

std::vector<Instruction> finalCodes; // 目标代码缓冲区

Error& error\_handle; // 错误处理程序

protected:

private:

};

#endif // MIPSINSTRGEN\_H

#### 算法

##### 基本目标代码生成

由于中间代码已经比较接近目标代码, 只需要逐条对照符号表进行翻译就可以了. 当然, 生成目标代码的前提是前面的词法分析, 语法分析和语义分析没有出现编译错误. 一旦发生编译错误, 目标代码停止生成, 但之后的语法分析和语义分析仍然进行, 以尽量多地报错.

##### 窥孔优化

循环遍历生成的目标代码, 如果出现相邻的的两条指令出现类似先load后store或先store后load的情况, 省掉后一条指令, 并不会对目标代码运行结果产生影响, 这时完全可以省掉后一条指令.

### 8. 优化方案\*

在目标代码生成阶段进行了常量合并优化和窥孔优化.

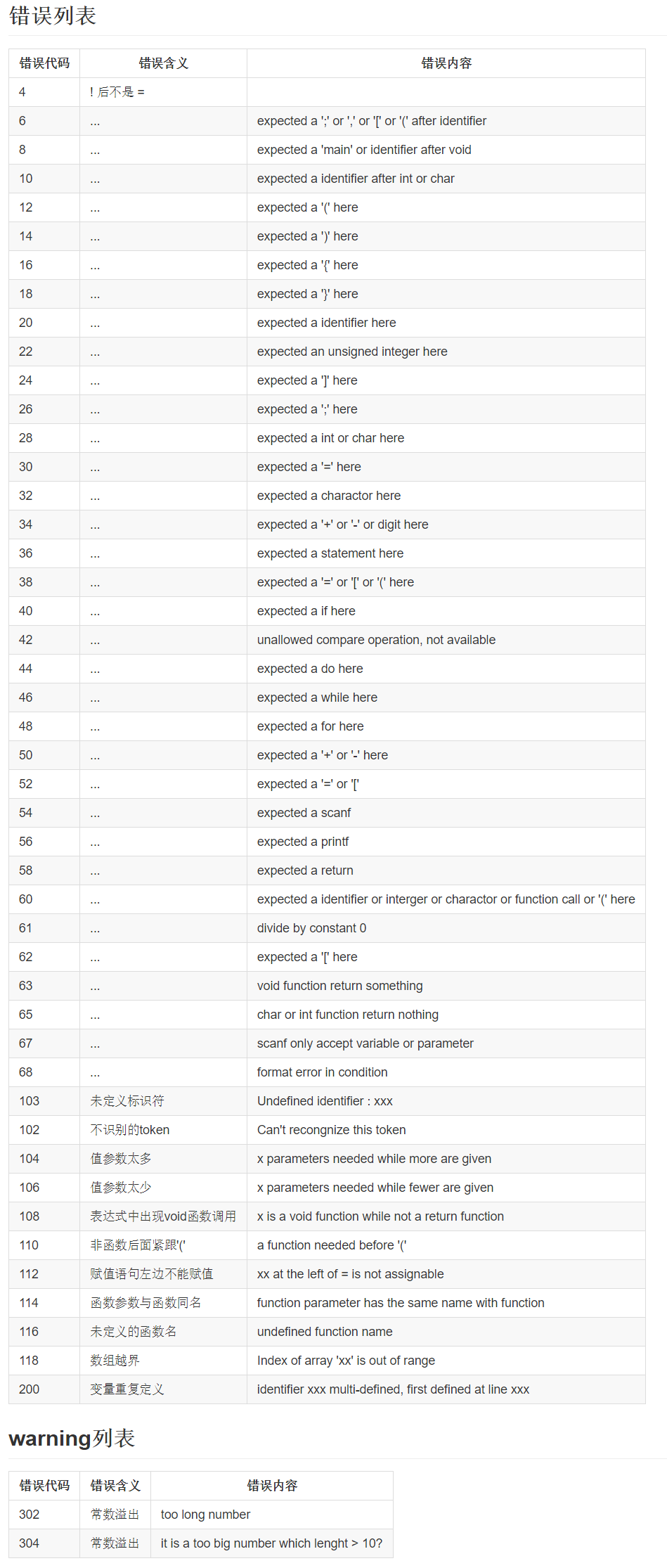
### 9. 出错处理

错误可分为 编译时错误, 运行时错误.

编译时错误由错误处理程序进行处理. 编译时错误又可分为词法分析错误, 语法分析错误等. 发生这类错误后, 需要实时输出错误位置和信息, 而且不能以发现有错误就停止编译, 而是要对错误进行适当的处理, 试图揣摩错误的原因, 并进行改正, 或者尽可能使错误波及的范围尽可能小, 也就是错误的局部化处理, 从而使编译工作可以继续进行下去. 错误信息一般有: “无法识别的标识符”, “[小|中|大]括号不匹配”, “未声明的函数”, “未声明的变量”, “变量重复声明”, “函数重复声明”, “不符合文法规范”, “找不到main函数”, “函数需要\*个参数, 而你给了\*\*个”, “引用和声明的种类/类型不同”, “表达式中的参数类型不一致”, “数组下标越界”, “出现除0操作”, “常量溢出”.

运行时错误由编译程序在编译时向目标程序插入相关代码进行处理, 比如引用数组寻址前检查是否越界, 是否出现除零的情况, 栈溢出等. 错误信息会在运行时输出, 避免程序的崩溃.

一旦发生编译错误, 目标代码停止生成, 但之后的语法分析和语义分析仍然进行, 以尽量多地报错. 语义错误可以同时精准的报多个, 语法错误尽可能的准确和多报, 但只能保证第一个的准确, 因为第一个语法错误会影响递归向下分析程序的走向, 可能会引起混乱和误报.



## 三．操作说明

### 1．运行环境

编程环境:code blocks 13.12

编译器: TDM-GCC, v4.7.1, 32 bit

目标代码测试环境: Mars 4.5

code blocks 13.12(自带TDM-GCC编译器)windows下载地址:

<https://sourceforge.net/projects/codeblocks/files/Binaries/13.12/Windows/>

我选择的目标代码是MIPS, 所以需要一个MIPS Assembler and Runtime Simulator, 也就是MARS, 其最新版本是4.5, 可以在<http://courses.missouristate.edu/kenvollmar/mars/下载并安装>. Mars需要提前配置好java环境, <https://www.java.com/en/download/>可以在这里下载并安装Java环境.

### 2．操作步骤

1. codeblocks13.12打开codeblocks工程文件C0Compiler.cbp;
2. 对C0Copiler工程执行build操作; 生成的可执行文件为bin\Debug\C0Compiler.exe或\bin\Release\C0Compiler.exe(取决于build时所处的模式);
3. 在CMD执行命令/path/to/ C0Compiler.exe /path/to/source.txt /path/to/out.asm即可对编译源文件.
4. 如果编译成功, 目标代码会写入/path/to/out.asm; 如果编译不成功, 给出错误信息, 并不改动/path/to/out.asm;
5. 运行目标代码 java -jar /path/to/Mars4.5.jar /path/to/out/asm

## 四．测试报告

### 1．测试程序及测试结果

#### 测试1 corrcet1.c（正确）

Input a number, get its fibonacci: 28

Its fibonacci: 514229

Input a number, get its fibonacci: 25

Its fibonacci: 121393

Input a number, get its fibonacci: 20

Its fibonacci: 10946

Input a number, get its fibonacci: 10

Its fibonacci: 89

Input a number, get its fibonacci: 5

Its fibonacci: 8

Input a number, get its fibonacci: 3

Its fibonacci: 3

Input a number, get its fibonacci: 2

Its fibonacci: 2

Input a number, get its fibonacci: 1

Its fibonacci: 1

Input a number, get its fibonacci: 0

Its fibonacci: 1

Input a number, get its fibonacci: -1

Its fibonacci: 0

#### 测试2 corrcet2.c（正确）

13

7

6

13

#### 测试3 corrcet3.c（正确）

before : 123 -23 4 23 15 -4 6 -3 23 123 -23 4 23 5 -4 6 -3 23 123 -23 4

after : -23 -23 -23 -4 -4 -3 -3 4 4 4 5 6 6 15 23 23 23 23 123 123 123

#### 测试4 corrcet4.c（正确）

Input 2 Integer:2

2

4

Input 2 Integer:3

4

81

Input 2 Integer:4

5

1024

Input 2 Integer:-2

0

1

Input 2 Integer:2

-2

1

Input 2 Integer:2

4

16

#### 测试5 corrcet5.c（正确）

\_37

#### 测试6 error1.c（错误）

error occurs at (line: 10, column: 0) : expected a ';' here

error occurs at (line: 16, column: 0) : expected a ';' here

error occurs at (line: 22, column: 11) : Undefined identifier : x1 !

error occurs at (line: 25, column: 17) : expected a identifier or interger or charactor or function call or '(' here

Compiled error! exit code -1. error count: 4

Press any key to continue . . .

#### 测试7 error2.c（错误）

error occurs at (line: 7, column: 12) : expected a identifier or interger or charactor or function call or '(' here

error occurs at (line: 11, column: 16) : expected an unsigned integer here

error occurs at (line: 25, column: 19) : expected a identifier or interger or charactor or function call or '(' here

Compiled error! exit code -1. error count: 3

Press any key to continue . . .

#### 测试8 error3.c（错误）

error occurs at (line: 25, column: 0) : expected a '(' here

error occurs at (line: 71, column: 11) : 0 parameters needed while more are given !

error occurs at (line: 72, column: 11) : 2 parameters needed while fewer are given !

Compiled error! exit code -1. error count: 3

Press any key to continue . . .

#### 测试9 error4.c（错误）

error occurs at (line: 6, column: 16) : char or int function return nothing

error occurs at (line: 6, column: 16) : expected a ';' here

error occurs at (line: 6, column: 16) : expected a '}' here

error occurs at (line: 6, column: 16) : expected a '}' here

Compiled error! exit code -1. error count: 4

Press any key to continue . . .

#### 测试10 error5.c（错误）

error occurs at (line: 2, column: 10) : identifier MAX multi-defined, first defined at line 1.

error occurs at (line: 6, column: 12) : Undefined identifier : ch !

error occurs at (line: 6, column: 14) : expected a '[' here

error occurs at (line: 7, column: 11) : void function return something

error occurs at (line: 7, column: 15) : expected a '[' here

error occurs at (line: 12, column: 8) : identifier ch multi-defined, first defined at line 11.

error occurs at (line: 12, column: 12) : identifier j multi-defined, first defined at line 11.

error occurs at (line: 12, column: 15) : identifier k multi-defined, first defined at line 11.

error occurs at (line: 12, column: 18) : identifier i multi-defined, first defined at line 12.

error occurs at (line: 16, column: 16) : 0 parameters needed while more are given !

error occurs at (line: 27, column: 54) : Divide by constant 0

Compiled error! exit code -1. error count: 11

Press any key to continue . . .

### 2．测试结果分析

#### 测试1（正确）

本程序重点测试了函数的递归调用，同时还有输入输出语句，循环语句等等。

程序的功能是等待用户输入一个数，输出其斐波那契数，然后等待用户输入下一个数,结直到用户输入的数为负数, 结果正确。

#### 测试2（正确）

本程序重点测试数组的取值和赋值以及更复杂的数组嵌套, 最后输出一些的关键数组值, 以判断程序的正确运行. 正确结果为分行输出13 7 6 13这4个数. 结果正确。

#### 测试3（正确）

本程序是受测试一的启发而写的快排, 涉及嵌套循环, 常量计算, 加减符号连接, 数组的取值和赋值, 分支语句, 递归调用. 最后输出排序前后的结果. 结果正确。

#### 测试4（正确）

本程序受测试二启发, 模仿测试二, 从控制台读入两个数, 计算它们的, 并输出结果. 结果正确。

#### 测试5（正确）

本程序主要测试分支语句, 实现了多层if语句的嵌套测试, 条件几乎覆盖所有可能. 最终执行最深层的输出语句, 输出” \_37”, 同时测试了空语句. 结果正确.

#### 测试6（错误）

该程序测试了语法和语义错误, 共4处. (line: 10, column: 0), (line: 16, column: 0)两个语句缺少分号; (line: 22, column: 11), 未定义变量; (line: 25, column: 17), 缺少语法成分. 结果正确.

#### 测试7（错误）

该程序测试了多个语法错误的正常报错, 共3处. (line: 7, column: 12), printf缺少参数; (line: 11, column: 16), 声明数组时没有声明长度; (line: 25, column: 19), 取数组值时缺少下标. 结果正确.

#### 测试8（错误）

该程序测试了1个语义错误和2个语法错误. (line: 25, column: 0), if语句缺少条件; (line: 71, column: 11), 函数调用时向不需要参数的函数传了一个参数; (line: 72, column: 11), 函数调用时向需要2个参数的函数只传了一个参数. 结果正确.

#### 测试9（错误）

测试多处语法错误和语义错误. (line: 6, column: 16), return一个返回值缺少小括号, 之后的语法错误和语义错误会被越过甚至还会乱报一些其他的语法错误, 这是由于递归向下子程序. 结果正确.

#### 测试10（错误）

本测试程序测试多种语义错误, 包括变量和常量的重复定义, 变量的未定义, void函数返回值, 函数参数数目不符, 除数为参数0. 结果正确.

## 五．总结感想

经过近8周的设计, 编写, 测试, 调试, 我终于完成了一个比较完整的扩展C0编译器. 经历过计组和OO的洗礼, 我们的能力有了很大的提高, 编译相对地变得可行了. 在这之前, 很难想象自己可以独立完成一个4000+行代码的工程, 而且还是用C++. 当工程变得很大, 开发周期变得很长时, 很多之前注意不到的问题就会凸显出来. 比如, 如何在几周后仍然看懂自己的代码细节; 如何充分地对程序进行测试; 当修复bug或改进程序后, 如何进行回归测试, 保证自己对程序的改动没有引入新的bug. 这些都是我们在写编译时特别注意的.

接下来我总结几点, 完成编译课设需要的注意事项:

1. 勤写注释, 勤写文档. 要保证自己可以在一个月后, 迅速看懂自己写的代码, 没有注释是很难做到的. 我之前的习惯是每个函数写一个注释, 注明这个函数的作用, 参数和返回值的意义. 这让我在做编译时受益颇丰. 但我发现这是不够的. 在做编译的时候, 很多情况下是, 刚开始设计的时候, 你想了很多, 什么兼容性, 可扩展性. 第二次看的时候, 有时很难会想到最初设计时的想法, 有时会把之前设计好的某些巧妙的设计否定到. 到最后需要这个巧妙的设计时, 才发现自己多么地蠢. 为了避免这个问题, 我养成设计时就撰写文档, 编写程序时随时回顾文档, 进行修改或增删. 而且写文档的时候也是思考的过程, 发现错误的过程. 这间接证明了我们学习软件工程的必要性.

2. 设计时要充分考虑最后的生成代码和优化. 否则代码生成和优化的时候会很痛苦的, 改又不舍得改, 在原先的基础上实现起来又和吃屎一样. 最后还是改了好多设计, 为了方便实现代码生成和优化.

3. 保重好自己的身体. 这句话我不知道该不该讲, 最后还是决定说出来. 克制地熬夜, 按时吃饭, 规律作息, 坚持锻炼. 这些都是拥有好身体的前提. 可惜的是, 这几点我一点都没有做到. 有一周生成目标代码时, 我理解成了要生成可执行的正确代码, 熬夜到早上6点调出来, 休息了3个小时又起来接着写文档, 当时就感到心率上升, 胸闷, 恶心, 肚子难受, 上午的篮球课都没敢去上, 就是怕猝死在篮球场上. 到了最后几周的时候, 各种结课, 大作业和考试, 熬夜成了常态, 有时甚至要刷夜. 坚持到了现在, 只剩下两门课设了, 但明显地感觉到自己的身体不如从前了. 以前认为自己年轻, 可以熬, 现在我真的不这么认为了. 生命诚可贵, 下学期我一定不会这样拼了, 保重好自己的身体才是第一位的.

比较遗憾的是, 由于之前被一些bug拖累和最后几周课设考试频繁, 时间紧张, 最后我没有完成所有计划的编译优化工作.

我的建议是, 理论课可以和实验课更相关, 多讲些我们实际实现编译器时可以用到的技巧和会遇到的坑. 这可能和老师相关? 听说后大班的史晓华老师讲实验相关讲的多很多. 在实现dag优化局部公共子表达式时, 这种感觉尤为明显. 原理课中学习的dag图很简单, 按照算法来就可以完成了, 而我们实际需要的完全不是这样, 什么数组, 读写语句, 函数调用, 都需要我们自己考虑和实现. 而之前的重课, 像计组, 操作系统, 理论和实验联系都是极其紧密的, 操作系统甚至理论和实验都是一个老师带. 相比之下, 编译的理论课和实践课的脱节是很明显的. 虽然张莉老师为了实验而改变教学顺序, 强行赶进度, 但对实验的细节涉及比较少, 实验的坑都需要我们自己踩很多. 不过单作为原理课本身是很优秀的, 我也从中学到了很多.