《编译技术》课程设计文 档

学号： 15061124

姓名： 郭维泽

2017年 11 月 21日 初稿

2018年 1 月 13日 终稿

## 一．需求说明

### 1．文法说明

＜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｝"

＜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＞｝

＜常量＞ ::= ＜整数＞|＜字符＞

＜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＞}

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

＜类型标识符＞ ::= int | char

＜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＞‘]’ )}

＜变量说明＞ ::= ＜变量定义＞;{＜变量定义＞;}

＜声明头部＞ ::= int＜标识符＞ | char＜标识符＞

＜因子＞ ::= ＜标识符＞｜＜标识符＞‘[’＜表达式＞‘]’|‘(’＜表达式＞‘)’｜＜整数＞| ＜字符＞｜＜有返回值函数调用语句＞

＜项＞ ::= ＜因子＞{＜乘法运算符＞＜因子＞}

＜表达式＞ ::= ［＋｜－］＜项＞{＜加法运算符＞＜项＞}

＜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

＜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＞;|＜情况语句＞｜＜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＞

＜循环语句＞ ::= while ‘(’＜条件＞‘)’＜语句＞

＜情况子语句＞ ::= case＜常量＞：＜语句＞

＜情况表＞ ::= ＜情况子语句＞{＜情况子语句＞}

＜缺省＞ ::= default : ＜语句＞

＜情况语句＞ ::= switch ‘(’＜表达式＞‘)’ ‘{’＜情况表＞[＜缺省＞] ‘}’

＜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＞‘)’]

＜语句列＞ ::= ｛＜语句＞｝

＜参数＞ ::= ＜参数表＞

＜参数表＞ ::= ＜类型标识符＞＜标识符＞{,＜类型标识符＞＜标识符＞}

＜值参数表＞ ::= ＜表达式＞{,＜表达式＞}

＜有返回值函数定义＞ ::= ＜声明头部＞‘(’＜参数＞‘)’ ‘{’＜复合语句＞‘}’ | ＜声明头部＞‘{’＜复合语句＞‘}’ //第一种选择为有参数的情况，第二种选择为无参数的情况

＜无返回值函数定义＞ ::= void＜标识符＞(’＜参数＞‘)’‘{’＜复合语句＞‘}’ | void＜标识符＞{’＜复合语句＞‘}’//第一种选择为有参数的情况，第二种选择为无参数的情况

＜有返回值函数调用语句＞ ::= ＜标识符＞‘(’＜值参数表＞‘)’ | <标识符> //第一种选择为有参数的情况，第二种选择为无参数的情况

＜无返回值函数调用语句＞ ::= ＜标识符＞‘(’＜值参数表＞‘)’ | <标识符> //第一种选择为有参数的情况，第二种选择为无参数的情况

＜复合语句＞ ::= ［＜常量说明＞］［＜变量说明＞］＜语句列＞

＜主函数＞ ::= void main‘(’‘)’‘{’＜复合语句＞‘}’

＜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＞

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）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

（5）数组的下标从0开始

**针对递归子程序法的分析：**

FIRST集：

<常量说明> const

<变量说明> int | char

<因子> Id | Integer | Character | ( | + | - | 0

<项> Id | Integer | Character | ( | + | - | 0

<表达式> Id | Integer | Character | ( | + | - | 0

<条件> Id | Integer | Character | ( | + | - | 0

<语句> Id | if | switch | while | scanf | printf | return | ; | {

赋值或调用|条件|情况|循环|读|写|返回|空|语句组

<赋值语句> Id

<条件语句> if

<循环语句> while

<情况表> case

<缺省> default

<情况语句> switch

<读语句> scanf

<写语句> printf

<返回语句> return

<参数><参数表> int | char

<值参数表> <表达式>

<有返回值函数定义> int | char

<无返回值函数定义> void

<有返回值函数调用语句> Id

<无返回值函数调用语句> Id

<主函数> void

<复合语句> <常量说明> | <变量说明> | <语句> | ε

<程序> <常量说明> | <变量说明> | <有返回值函数定义> | <主函数>

| <无返回值函数定义>

FOLLOW集：

<复合语句> }

结论：存在以下冲突：

（1）Id同时存在于<赋值语句>、<有返回值函数调用语句>和<无返回值函数调用语句>中

但是这个冲突可以依据Id的具体内容查符号表解决

（2）int | char 同时存在于<变量说明>和<有返回值函数定义>中

需要向前看两个单词，如果是 ( 或者 { 就是<有返回值函数定义>，如果是 [ 或者 , 或者 ; 则是<变量说明>

注意：虽然<参数>也是以int | char开头，但是它必须在函数定义中，所以并不冲突

（3）void 同时存在于<主函数>和<无返回值函数定义>中

需要向前看一个单词，如果是main则是<主函数>，如果不是main则是<无返回值函数定义>

（4）<因子> <项> <表达式> <条件>

虽然这四项FIRST集完全相同，但是属于嵌套关系，可以加以限制区分开。

### 2．目标代码说明

生成32位MIPS汇编，可在MARS软件中执行。

使用了少量的伪指令，包括：

(1)含有label的跳转指令

(2)b指令：bgt、bge、blt、ble

(3).asciiz字符串宏定义

(4)li、la和move

### 3. 优化方案\*

必须包含：

(1)基本块内部的公共子表达式删除（DAG图）；

(2)全局寄存器分配（引用计数或着色算法）；

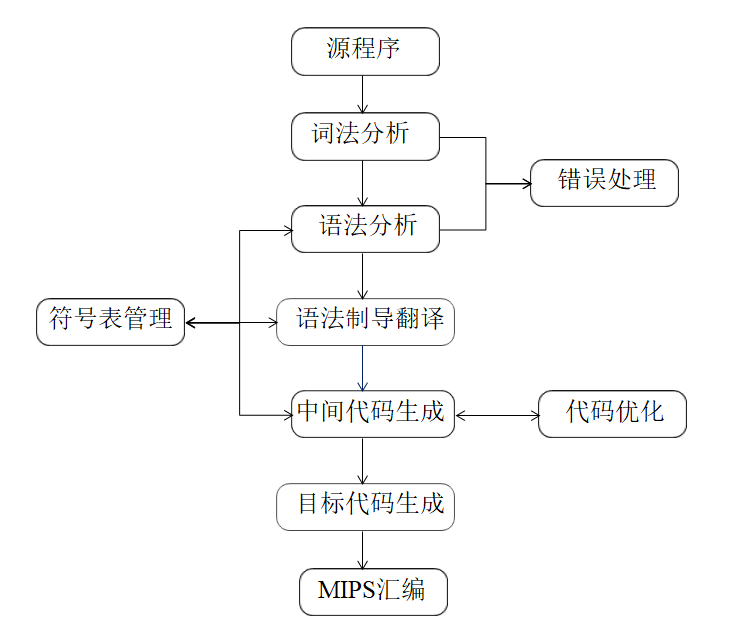
(3)数据流分析（通过活跃变量分析，或利用定义-使用链建网等方法建立冲突图）；

其它优化，例如复制传播、循环强度削弱等，可以自行选作。

要求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



词法分析和语法分析都和出错处理有交互；

符号表管理贯彻语法分析、语法制导翻译和中间代码生成；

### 2．类/方法/函数功能

程序采用C++语言编写，使用简单的面向对象方法封装了各个阶段：

·共计6个类：Lexer、Parser、Code、MIPS、Error、Table

·另有两个枚举类(Enum)：symbol和kind

·另有两个结构：token和record

·另有三个全局变量：fin、fcode、fout

各个部分的作用下面详细说明：

**全局变量**

fin 输入文件名

fcode 中间代码输出文件名

fout 最终MIPS代码输出文件名

**单词类型枚举 symbol**

包含共计40个单词类型：

NULLSYM, CONST, VOID, CHAR, INT, MAIN, IF, ELSE, SWITCH, CASE, DEFAULT, WHILE, RETURN, SCANF, PRINTF, ID, INTEGER, ZERO, CHARACTER, STRING, PLUS, MINUS,

STAR, DIV, EQ, BIG, LESS, BIGEQ, LESSEQ, NOTEQ, LSMALL, RSMALL, LMID, RMID,

LBIG, RBIG, COMMA, SEMI, COLON, ASSIGN

(1) 将0作为单独的类型处理：ZERO，其原因是在文法中0之前不能有正负号，数组声明时大小也不能为0。

(2) NULLSYM表示没有单词，初始化时会先使用NULLSYM。当然，不会在词法分析中返回。

**标识符类型枚举 kind**

包含共计14个标识符类型：

charconst, intconst :常量

charvar, intvar :变量

voidfunc, charfunc, intfunc :有参数函数

npvoidfunc, npcharfunc, npintfunc :无参数函数

charpara, intpara :函数参数

chararray, intarray :数组

**单词结构 token**

struct token {

symbol sym; //单词类型

std::string id; //名字

int num; //常量数值

};

**符号表项 record**

struct record {

std::string name;

kind kind; //类型

int value; //常量:值,数组:元素个数,函数:参数个数,参数:序号,变量:未定义

int level; //函数编号: 全局是0, 每进入一个新函数+1, 函数名是自身的编号

int size; //空间大小: 函数是返回值的空间大小; 数组是总大小

int addr; //内存地址: 常量变量是数据地址; 数组是起始地址; 函数是返回值的 地址; 参数是栈内偏移地址

};

**错误处理类 Error**

public:

int errnum; //错误计数

void err(string message); //报错

(1) errnum是内部错误计数器，每次报错都+1；

(2) 报错函数err向执行窗口输出错误原因

**词法分析类Lexer**

private:

Error\* error; //错误处理对象

char ch; //处理的字符

string id; //读到的词法单元名(不转换大小写)

token t; //读到的词法单元

int line; //行计数

int word; //每行代码字符计数

int is\_over; //判断是否结束

void getch(); //读取一个字符

public:

Lexer::Lexer(Error\* err\_p);

token getsym(); //词法分析主函数:读取单词

void all\_exit(); //退出程序

void has\_err(string message); //封装的报错函数(包含位置)

(1) 用getsym函数完成整个的词法分析：

·调用函数getch读取一个字符，保存在ch中，利用line和word记录读到的位置。

·仿照Pascal-S进行词法分析，返回一个单词，类型是token。

·如果读到文件结束，将is\_over置为1，驱动退出函数all\_exit退出程序。

·如果有词法分析错误，则直接调用has\_err报错。

(2) 调用Error的函数err，实现Lexer的报错函数has\_err，包含了错误的位置。

(3) 将词法分析中的各个变量都设为private，实现了封装，保证结构的可靠性。

(4) 在字符和字符串识别方面，如果读到一个左引号，则一直读字符，直到读到右引号、非法字符或者本行结尾，读到非法字符或者本行结尾时报错。读完字符后还需检查字符长度，应该为1，否则报错。

**符号表类 Table**

private:

static int num; //函数编号

vector<record> global; //全局变量记录

vector<record> t; //符号表记录

vector<record>::iterator iter; //迭代器

public:

Table();

int level; //运行时当前函数编号, 0是全局

bool is\_found (string name); //查全局或数据表, 标识符是否重复定义

bool is\_found\_global (string name); //查全局表, 标识符是否重复定义

record find (string name); //查全局和数据表, 返回记录

record find\_global (string name); //查全局表, 返回记录

record last\_global (); //全局表中最后一个函数记录

record find\_para (int func\_level, int para\_index); //查找参数

void add (record r); //在数据表添加

void add\_global(record r); //在全局表添加

void write\_value (int value); //回填value

void write\_addr (int addr); //回填addr

void new\_level (); //函数编号num+1, 并赋值给level

(1) 符号表用C++中的vector实现，完全封装在类中。

·符号表分为全局表和局部表，全局表保存所有的函数(因为函数都是全局的)、全局变量和全局常量，局部表保存函数中的局部变量和局部常量，包括参数。

·变量level表示当前所处的层次，全局是0，每个函数依次加一。

(2) 查找符号表有单独的函数：

·由于是小型软件，符号表的查找采用顺序查找法，通过迭代器iter实现查找。当level不相等时直接跳过。

·函数find先查找局部表，如果局部表不存在，在查找全局表。

·函数find\_para直接到所在的局部表里查找。

(3) 添加符号表有单独的函数：

·函数add在局部表中添加，但是当level是0的时候，意味着尚且处在全局阶段，此时在全局表中添加。

(4) 函数write\_value和write\_addr是回填符号表，用于函数声明后参数个数和地址的回填。

**语法语义分析类 Parser**

private:

Lexer\* lexer; //词法分析对象

Table\* table; //符号表

Code\* code; //中间代码生成器

token t; //处理的单词

record r; //处理的符号表项

int is\_forward; //预读个数

token forward1, forward2; //预读的单词

vector<symbol> all\_sym; //语句块和声明结束

void read(); //读入一个单词

token forwardread(); //预读一个单词

void skip(); //跳读

void skip(vector<symbol> fsys); //跳读

void newrecord(); //重置记录r

bool is\_same\_kind (kind const\_kind, kind other);//判断常量类型是否一致

bool is\_return (kind function\_kind, kind return\_kind); //判断返回类型 是否一致

bool is\_para (kind para\_kind, kind expr\_kind); //判断参数类型是否一致

public:

Parser(Lexer\* lexer, Table\* table\_p, Code\* code\_p);

//小的语法项目

int integer (); //整数(有符号和0)

record constant (); //字面常量

void arraydec (); //数组定义 ‘[’ Integer ‘]’

record arraycall (record array\_name); //数组引用 ‘[’ <表达式> ‘]’

int condition (); //条件

record factor (); //因子

record term (); //项

record expression (); //表达式

int formalpara (); //形参表

void actualpara (record function); //实参表

//语句项目

void constdec (); //常量声明

void constdef (); //常量定义

void vardec (); //变量声明

void vardef (); //变量定义

void funcdef (); //函数定义

void statement (); //单个语句

void stats (); //复合语句

record funcall (); //函数调用

void assignstat (); //赋值语句

void ifstat (); //if语句

void whilestat (); //while语句

void casestat (record switch\_on, int end\_label); //case语句

void defaultstat (); //default语句

void switchstat (); //switch语句

void returnstat (); //return语句

void scanfstat (); //输入Input

void printfstat (); //输出Output

//程序部分

void compoundstats (); //复合语句

void program (); //语法分析入口,开始符号

这是整个程序中最关键也是最复杂的类，是整个程序的唯一入口(主函数只调用一个program)。

(1)整个类采用了递归子程序法的结构，以非终结符<程序>开始，逐步展开，最终编译完整个程序。要注意一点的是，文法有几个地方并不满足递归子程序法的要求，能改写的已经尽可能的改写，其余的通过预读来解决，最多需要预读2个单词。

(2)每个非终结符都有一个函数与之对应，按照各自的语法进行操作。在需要读入新的单词时，调用read函数即可。有的需要符号表的支持，就调用符号表的相关函数。

(3)将Lexer的getsym函数进一步包装，加入了预读的功能，成为新的函数read。需要预读时，调用另一个函数forwardread。

(4)设置跳读函数skip，在报错之后调用skip进行跳读。

(5)设置几个类型判断函数，在不同的情况下进行类型的比较。

(6)语法语义分析的函数都只调用Code类，而不会调用MIPS类，实现了良好的封装性。

**中间代码类 Code**

private:

int lnum; //Label标号

int tnum; //临时变量标号

Table\* table; //符号表

MIPS\* mips; //MIPS生成器

public:

Code(Table\* table\_p, MIPS\* mips\_p);

//常用函数

int getlabel (); //返回新Label

int putlabel (); //放置新Label,并返回

void putlabel (int label); //放置已有的Label

record new\_temp (kind temp\_kind); //返回新的临时变量

//特殊过程

void alloc\_add (record r); //添加声明

void funcdec (record function); //函数声明

void para (record parameter); //参数声明

void force (); //强制返回

void ret (); //返回语句

void ret (record func, record name); //返回语句,有返回值

void scan (record name); //输入变量 scanf name

void print (record name); //输出变量 printf name

void prints (string text); //输出字符串 printf text

record call (record func); //调用语句 call func

void para\_trans (record para, record expr); //参数传递 push para

void assign (record left, record right); //赋值语句 left=right

void load (int reg, record rec); //取值

void store (int reg, record rec); //存入

//计算过程

record neg (record num); //取反

record plus (record op1, record op2); //加法

record minus (record op1, record op2); //减法

record mult (record op1, record op2); //乘法

record div (record op1, record op2); //除法

record array\_call (record array\_name, record index); //数组引用

//跳转过程, 不满足跳转到新Label

int condition (record single); //单一表达式条件

int equal (record left, record right); //条件 left == right

int notequal (record left, record right); //条件 left != right

int big (record left, record right); //条件 left > right int bigequal (record left, record right); //条件 left >= right

int less (record left, record right); //条件 left < right

int lessequal (record left, record right); //条件 left <= right

int jump (); //无条件跳转到新Label

void jump (int label); //无条件跳转到已有的Label

中间代码类只负责生成中间代码，在Parser类中被调用。

(1)每种中间代码都有一个函数与之对应。在要生成该中间代码时，被Parser调用。

(2)设置getlabel和putlabel两个函数。其中putlabel有两种情况，分别对应于放置已有的label和放置新label。

(3)Code调用MIPS来生成最终代码。

**MIPS指令生成类 MIPS**

private:

static string reg[32]; //寄存器名

static int saved\_reg[10]; //保存寄存器

static int string\_num; //字符串标号

static int alloc; //全局地址

static int offset; //栈内偏移地址

public:

MIPS();

void initial (); //初始化

//常用函数

void label (string label\_name); //放置函数标号

void label (int num); //放置标准标号

int alloc\_addr (int size); //申请全局地址

int alloc\_stack (int size); //申请栈内地址

void new\_offset (); //新的offset

void push (); //寄存器入栈

void pop (); //恢复寄存器

void scan\_int (); //读入整数

void scan\_char (); //读入字符

void print\_int (); //输出整数

void print\_char (); //输出字符

void print\_string (string text); //输出字符串

//常用伪指令

void li (int t1, int num);

void la (int t1, string label, int t2);

void move (int t1, int t2);

//加减指令

void add (int t1, int t2, int t3);

void addi (int t1, int t2, int num);

void sub (int t1, int t2, int t3);

void subi (int t1, int t2, int num);

//乘除指令

void mfhi (int t1);

void mflo (int t1);

void mult (int t1, int t2);

void div (int t1, int t2);

//跳转指令

void j (int label);

void jr (int t1);

void jal (string target);

//分支指令

void beq (int t1, int t2, int label);

void bne (int t1, int t2, int label);

void bge (int t1, int t2, int label);

void bgt (int t1, int t2, int label);

void ble (int t1, int t2, int label);

void blt (int t1, int t2, int label);

//存取指令(访存指令)

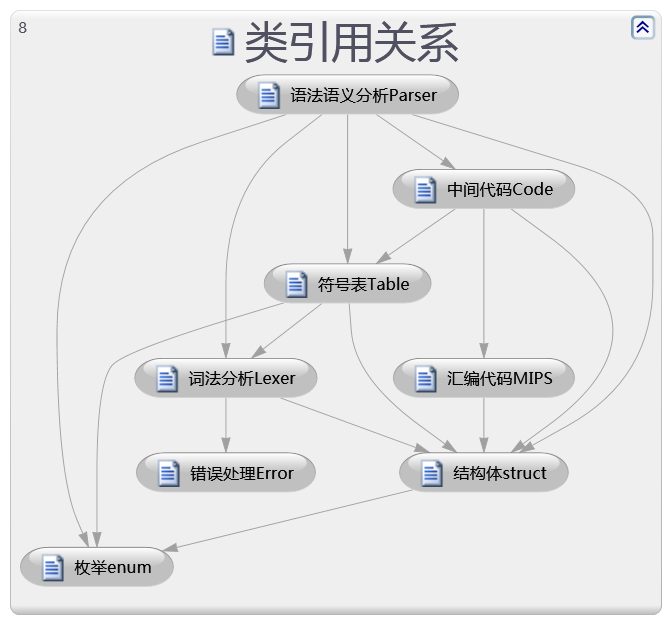
void lw (int t1, int num, int t2);

void sw (int t1, int num, int t2);

最终MIPS代码生成类，只能被Code调用。每种MIPS指令都对应一各函数。

### 3．调用依赖关系

以语法语义分析类Parser为中心，具体的引用关系如下图：



(1)Lexer类提供了两个关键函数：词法分析函数getsym和报错函数has\_err。getsym返回一个单词。has\_err包装了Error的err函数，添加了位置信息，其他类不能直接调用Error。符号表调用Lexer只是为了报错。

(2) 枚举类和结构体都是必要的基础类型，几乎所有的类都包含了这两个。

(3)MIPS类只负责生成最终代码，不需要依靠除中间代码外的任何信息，所以它只被Code类引用了。

(4)Code类负责生成中间代码，需要了解符号表和语法分析的信息，所以它引用了Table类和被Parser类引用。

(5)Parser类负责最关键的语法语义分析，而且它是最顶层的类，所以它引用了Lexer、Code、MIPS三个类，而且它里面有主函数调用的唯一函数program。

### 4．符号表管理方案

(1) 符号表用C++中的vector实现，完全封装在类Table中。

·符号表中存储的是标识符记录record

·符号表分为全局表和局部表，全局表保存所有的函数(因为函数都是全局的)、全局变量和全局常量，局部表保存函数中的局部变量和局部常量，包括参数。

·变量level表示当前所处的层次，全局是0，每个函数依次+1。

(2) 查找符号表有单独的函数：

·由于是小型软件，符号表的查找采用顺序查找法，通过迭代器iter实现查找。当level不相等时直接跳过。

·函数find先查找局部表，如果局部表不存在，在查找全局表。

·函数find\_para直接到所在的局部表里查找。

(3) 添加符号表有单独的函数：

·函数add在局部表中添加，但是当level是0的时候，意味着尚且处在全局阶段，此时在全局表中添加。

·添加局部表时，由于肯定是在表的最末端添加，直接用push\_back()即可。

### 5．存储分配方案

(1)地址分配方案

在MARS里的default分配方案中，可用的地址有0x10040000到0x7ffffffc的空间。其中，使用较高的地址空间作为栈空间，栈指针初始化为默认的值0x7ffffff8，栈从高地址向低地址生长。使用较低的地址空间作为全局空间，基地址为0x10040000。字符串采用宏定义存储，放在默认的全局指针0x10008000处。这样就保证了地址之间不会相互占用。

(2)地址寻址方式

采用了直接寻址和相对寻址两种方式：

·全局数据采用了直接寻址方式：符号表里存储绝对地址，用绝对地址的立即数(相对于$0偏移)进行寻址。

·局部数据采用了相对寻址方式：符号表里存储相对地址，用相对于当前栈帧指针$fp的偏移量进行寻址。由于每次调用时，偏移量不会改变，所以保证了不同变量的地址不会冲突。

(3)临时变量地址分配

临时变量不在符号表中，需要临时开辟地址空间。在运行栈中，将栈指针$sp减小4(一个字的空间)，为临时变量开辟地址。由于临时变量随取随用的特点，生成一个临时的记录record，记录临时变量的相关信息。利用这个记录来进行寻址操作，和变量一致。

(4)数组地址管理

数组由于其结构的特殊性，不能采用一般的寻址方法。在符号表中，存储了数组的基地址(数组的最低地址)和数组大小的信息。在寻址时，先按照通常的寻址方法找到数组的基地址，不管是直接寻址还是相对寻址。然后计算元素的相对偏移(元素下标\*4)，加上基地址，形成元素的地址，然后按照这个地址寻址即可。

(5)函数调用地址管理

每次函数调用都生成一个新的函数运行栈，结构为：保存寄存器、参数、局部变量，依次向低地址生长。

函数调用分为前后两个阶段：

·调用前，将要传递参数填入相应的地址中，由于知道当前栈的栈顶$sp，则从$sp-40开始向低地址依次填入参数值。然后保存寄存器，包括栈帧指针$fp。接着进行指针变换，将当前的栈帧指针$fp移动到当前的栈指针$sp处，然后向下移动$sp到$fp-40处，即指向参数的起始位置。由于被调用函数知道自己的参数个数，这时将$sp跳过参数部分，移动到新的栈顶，开始执行新的调用。

·调用后，将返回值先写入在全局数据区的预留空间中，然后进行指针变换，将当前的栈指针$sp恢复为当前的栈帧指针$fp，即恢复为调用前的位置。此时保存的寄存器值依然在栈指针$sp的下方。接着恢复寄存器的值，包括栈帧指针$fp的值。这样，两个指针就都恢复到调用之前的位置了。此时在开辟一个临时变量，存储函数的返回值，本次调用就完成了。

### 6. 四元式设计

采用给定的中间代码格式，稍加改动：

原则上按照中缀表达式格式输出中间代码，即，形如x = y op z，其中x为结果，y为左操作数，z为右操作数，op为操作符。以下根据基本语法现象举例说明。

1. 函数声明

源码形如：

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

中间代码：

int foo()

para int a

para char b

para int c

源码形如：

int foo

中间代码：

int foo()

2. 函数调用

源码形如：

i = tar(x,y)

中间代码：

push x

push y

call tar

i = RET

源码形如：

i = foo

中间代码：

call foo

3. 函数返回

源码形如：

return (x)

中间代码：

ret x

4. 变量声明和常数声明

不生成真正的中间代码

5. 表达式

源码形如：

x = a \* (b + c)

中间代码（中缀表达式，可优化）：

t1 = b + c

t2 = a \* t1

x = t2

6. 条件判断

源码形如：

x == y

中间代码：

x == y

7. 条件或无条件跳转

中间代码：

GOTO LABEL //无条件跳转到LABEL

BNZ LABEL //满足条件跳转到LABEL

BZ LABEL //不满足条件跳转到LABEL

函数的标签就是函数名，因为函数名不会重名；临时标签的形式是$L6，用$开头是为了不和函数名相同

8. 数组赋值或取值

源码形如：

a[i] = b \* c[j]

中间代码：

t1 = c[j]

t2 = b \* t1

a[i] = t2

### 7. 目标代码生成方案

选择了22条MIPS指令，具体如下：

选择的指令 指令介绍

li $t1,num $t1=num

la $t1,lable($t2) $t1={label}+$t2

move $t1,$t2 $t1=$t2

add $t1,$t2,$t3 $t1=$t2+$t3, signed, with overflow

addi $t1,$t2,num $t1=$t2+num, signed

sub $t1,$t2,$t3 $t1=$t2-$t3, signed, with overflow

subi $t1,$t2,num $t1=$t2-num, signed

mfhi $t1 $t1=HI

mflo $t1 $t1=LO

mult $t1,$t2 (HI,LO)=$t1\*$t2, signed

div $t1,$t2 LO=$t1/$t2, HI=$t1%$t2, signed, with overflow

j target PC=target

jal target $ra=PC, PC=target

jr $t1 PC=$t1

beq $t1,$t2,label if($t1=$t2) PC={label}

bne $t1,$t2,label if($t1!=$t2) PC={label}

bge $t1,$t2,label if($t1>=$t2) PC={label}

bgt $t1,$t2,label if($t1>$t2) PC={label}

ble $t1,$t2,label if($t1<=$t2) PC={label}

blt $t1,$t2,label if($t1<$t2) PC={label}

lw $t1,num($t2) $t1=m[$t2+num]

sw $t1,num($t2) m[$t2+num]=$t1

注：{label}:label的地址，PC为程序计数器，num最大为32无符号数或16位有符号数

### 8. 优化方案

虽然没有时间编写优化程序，但是我也查阅了相关的资料，进行了一些思考。

**一、修改Code类**

1. 运算操作符，运算数1，运算数2

最普通的三地址代码类型，这其中包括了所有的计算操作、参数传递和赋值操作。运算数1为数组名，运算数2为数组下标。参数传递的操作符为 & ，运算数1为函数名，运算数2为传入的变量或值。赋值操作的运算符为 = ，运算数1为左值，运算数2为右值。函数值返回值的存储也是一个赋值操作，左值为一个临时变量，右值为函数名。

2. ] 运算数1，运算数2 和 [ 运算数1，运算数2，运算数3

前者表示把数组作为右值赋值给其它变量，运算数1为数组名，运算数2为数组下标。后者表示对数组元素赋值，运算数1为数组名，运算数2为数组下标，运算数3为要赋的值。

3. #，foo

调用函数foo，一定在foo的参数都传完值后才能调用。

4. %f，运算数1，比较运算符，运算数2，6和 %t，运算数1，比较运算符，运算数2，6

有条件跳转，在分析时应当作为基本块的划分。单值条件被写成：%t 变量 != 0 6的形式。

5. %j，6

无条件跳转到标签6，在分析时应当作为基本块的划分。

6. %l，6

临时标签Label6，在分析时应当作为基本块的划分。

7. @f foo

函数foo的入口标签，在分析时应当作为整个流图的入口。

8. @r，运算数

函数返回，将运算数赋值给返回值。在分析时应当作为整个流图的出口。

**二、建立新的类**

1. Node类：实现了中间代码中的变量的数据结构。包括指向下层节点的指针数组Node\*[]，指向上层结点的指针数组Node\*[]，本节点的属性state和结点有效位valid。其中，state包括各种节点的类型。valid是一个boolean值，标识这个结点是否有效，初始值为true。

2. Block类：实现了基本块的数据结构。包含基本块的先前块数组和后继块数组vector<Block\*>，基本块的in集、out集、use集和def集vector<string>，以及是否更改标记valid。其中，valid初始值为1。

**三、进行具体优化**

1.基本块内部的公共子表达式删除(DAG图)

逐条分析中间代码，目标是划分基本块并建立DAG图。每当进入一个新的基本块时，便为这个基本块生成新的DAG图，DAG图由Node类的对象连接而成。并建立一个Map记录变量和Node对应关系，用变量名作为键值，属性是指向对应的Node的指针。在这个基本块结束时，由当前的DAG图重新生成中间代码，进行下一步优化。

(1) 当读入一行普通运算的中间代码后，利用Map在DAG图中查找相关的操作结点。如果没有，则新创建一个Node对象，将对应关系加入Map中。如果找到了某个运算数的结点，那么就可以搜索其上层节点，是否有公共子表达式，如果有就利用，没有就新创建一个结点。并将新结点的信息加入Map。

(2) 当读入一个数组引用后，判断这个引用是左值还是右值。如果是右值，就先构建]结点，左值就构建[结点。有一点特殊的是，后者在创建时，要杀死所有当前创建的其值依赖于同一数组的结点。具体做法是:在创建[节点时，检查所有运算数1的结点的上级节点，如果依赖于这个结点，那么将其标记为无效，即：valid=false。当然，在构建新的节点之前也要检查是否有公共子表达式，这是不仅要看表达式含义是否相同，也要看是否有效。如果是一个无效的表达式，那么就不把它作为公共子表达式，要创建新的节点。

(3) 当读入一个赋值操作后，根据Map的信息找到右值的结点，然后在Map里加入左值的信息。

(4) 当读入一个参数传递后，意味着马上要进行函数的调用了，但是此时无需结束当前层分析，按照处理赋值语句时一样处理即可。

(5) 当读入一个函数调用后，虽然需要跳转，但是一定要返回的，所以不用作为基本块的划分。但是要新建立一个结点表示函数调用的结果，结点类型为#，子结点为之前函数参数的所有节点。

(6)当遇到一个普通标签时，结束当前基本块，向下预读一行代码，判断是不是标签，如果仍是标签，就再向下预读一行，直到读到不是标签的行。利用构建好的DAG图和书上的启发式算法，重新生成这一基本块的中间代码。然后将所有标签原样输出。

(7) 当读入一个跳转代码后，结束当前基本块。利用构建好的DAG图和书上的启发式算法，重新生成这一基本块的中间代码。然后将跳转代码原样输出。

(8) 当遇到一个函数入口标签时，意味着这是一个流图的入口，只需要原样输出。

(9) 当读入一个返回代码后，意味着这是一个流图的出口，只需要原样输出。

2.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）

采用活跃变量分析，确定每个基本块出口的活跃变量，对于所有的活跃变量，建立冲突图。

(1) 读取上一步删除公共子表达式之后的中间代码，确定每个流图的出入口。函数的入口作为整个流图的入口，函数的return作为出口。如果没有return，那么函数的结尾作为出口。

(2) 在当前流图中，为每个基本块建立一个Block对象，并分析def和use集。如果变量先被引用，那么算进use集，如果先被赋值，那么算进def集。用一个临时的vector存放被分好类的变量，确保不重复分类。

(3) 运用书上的活跃变量分析算法，从出口开始，对所有的基本块进行迭代。每更改一个基本块时，都要将其valid设置为0。设置一个计数器存放计算过的基本块个数，每一轮都进行到所有基本块都被计算过一遍，即计数器等于基本块个数。在每轮结束时，检查所有基本块的valid位是否为1。若都为1，算法结束。否则，将所有valid设置为1，清零计数器，重新开始一轮计算。

(4) 在计算好每个基本块的活跃变量集之后，对每个基本块结束的活跃变量建立冲突图。先判断所有变量是全局变量还是局部变量。对于两类变量，用两个邻接矩阵表示冲突图，所有在同一基本块中活跃的变量之间都有边连接。

3.全局寄存器分配（引用计数或着色算法）

依照书上的启发式算法，由冲突图得出冲突的变量，为变量分配寄存器，可用的寄存器为除了$0,$at,$v0,$a0,$k0,$k1,$gp,$sp,$fp,$ra之外的所有寄存器，共计22个。其中，$s0-$s7这8个是全局寄存器，其余的14个是临时寄存器，即$v1,$a1,$a2,$a3,$t0-$t9。

(1) 根据数据流分析得到的邻接矩阵，由书上的启发式图着色算法进行着色。有所不同的是，当某一情况下无法着色时，要选出当前图中边最多的点删除，依照邻接矩阵的顺序进行判断。建立一个临时的Map来记录分配的结果。

(2) 根据分配的结果Map，进行MIPS代码生成。每当使用到一个变量，通过Map判断它此时在不在寄存器中。如果在，就直接使用寄存器中的值。如果不在，看是否需要分配寄存器。如果需要分配，那么将它的值调入到寄存器中，并且更改Map。如果不需要，那么直接从内存中取出值。在基本块的结尾，将所有在出口处活跃的变量的值都存入内存。

### 9. 出错处理

跳读终止符包括：

; | { | } | [ | ] | ( | ) | = | , | : | Id | if | switch | while | scanf | printf | return | void | char | int | const

跳读时，先判断是什么类型的错误，针对不同的错误跳读的要求也不同，大多数错误都是跳读到一条语句结束，比如分号。

(1)严重错误：

找不到输入文件

代码不完整，缺少主函数main

程序不允许的字符

(2)词法错误：

字符不完整，缺少'结尾；字符中必须至少包含一个字符，不能为空；字符中字符多于1个

字符中包含非法字符，只有字母、数字、下划线和四则运算符合法；字符串中有非法字符

字符串不完整，缺少"结尾；不等号错误，应为!=

(3)语法错误：

缺少无符号整数；缺少整数；缺少常量；缺少表达式；缺少运算对象；缺少标识符；缺少参数；缺少字符；缺少类型，假定为int；缺少类型，假定为void；缺少语句；缺少关键字；缺少情况case；缺少输出的内容；缺少左大括号；缺少右大括号；缺少左中括号；缺少右中括号；缺少左小括号；缺少右小括号；缺少分号；缺少冒号；缺少赋值符号；声明类型错误，不可以为void；常量声明要以const开始；传入参数过多；传入参数过少；缺少常量定义；缺少变量定义；变量定义不完整

(4)语义错误：

数组访问越界；数组引用下标非法,必须为整型值；符号未定义；不是可以引用的值；参数类型不符；标识符重复定义；不是可以被赋值的左值；不能将整型赋值给字符型；条件数据类型不符；返回值类型不符；缺少return；不是可以读取输入的值

## 三．操作说明

### 1．运行环境

Visual Studio 2010版本

Mars任意版本

### 2．操作步骤

编译程序运行在Visual Studio 2010上，需设置项目-属性-配置属性-链接器-清单文件-生成清单为：否。

点击调试运行，在出现Please input the input-file name:提示后，输入源文件的路径，回车开始执行，错误信息将出现在执行窗口上。如果没有报错，关闭执行窗口，打开项目文件out.txt，文件中是编译生成的所有MIPS代码。

将代码复制到MARS软件的新文件中，选择Settings——Memory Configuration，选择default选项，点击Apply and Close。然后汇编、执行。

## 四．测试报告

### 1．测试程序及测试结果

1.定义声明检查

const char const\_c0='\_';

const char const\_c1='\*', const\_c2='2';

const int const\_i0=0;

const int const\_i1=1, const\_i2=2;

char char\_0; char a\_c1[1];

char char\_1, a\_c2[2], a\_c3[3], char\_2, char\_3;

int int\_0; int a\_i1[1];

int int\_1, a\_i2[2], a\_i3[3], int\_2, int\_3;

void void\_np {

printf("void\_np");

return;

}

void void\_p (char p1, int p2) {

printf("void\_p");

return;

}

char char\_np {

printf("char\_np");

return (const\_c0);

}

char char\_1p (char p) {

char r;

printf("char\_1p");

r = p;

return (r);

}

char char\_p (char p1, int p2, int p3, char p4, char p5) {

printf("char\_p");

return (p1);

}

int int\_np {

printf("int\_np");

return (const\_i0);

}

int int\_1p (int p) {

int r;

printf("int\_1p");

r= p;

return (p);

}

int int\_p (int p1, char p2, char p3, int p4, int p5) {

printf("int\_p");

return (p4);

}

void main () {

char \_cnp;

char \_c1p, \_cp;

int \_inp;

int \_i1p, \_ip;

printf("Test: defination, IO, paramenter and return.");

char\_0=const\_c0; char\_1=const\_c1; char\_2=const\_c2;

int\_0=const\_i0; int\_1=const\_i1; int\_2=const\_i2;

a\_c1[0]=char\_0; a\_c2[1]=char\_1; a\_c3[0]=a\_c1[0]; a\_c3[1]=a\_c2[1];

a\_i1[0]=int\_0; a\_i2[1]=int\_1; a\_i3[0]=a\_i1[0]; a\_i3[1]=a\_i2[1];

scanf(char\_3); scanf(int\_3);

printf(const\_c0); printf(const\_i0);

printf(char\_0); printf(char\_1); printf(char\_2);

printf(int\_0); printf(int\_1); printf(int\_2);

void\_np;

void\_p(char\_1, int\_1);

\_cnp = char\_np;

\_c1p = char\_1p(char\_1);

\_cp = char\_p(char\_2, int\_0, int\_1, char\_0, char\_1);

\_inp = int\_np;

\_i1p = int\_1p(int\_1);

\_ip = int\_p(int\_0, char\_0, char\_1, int\_2, int\_1);

printf(\_cnp);printf(\_c1p);printf(\_cp);

printf(\_inp);printf(\_i1p);printf(\_ip);

}

输入：\*1 注：输入一个字符和一个数字

结果：

Test: defination, IO, paramenter and return.

\*1 一个字符和一个数字

\_

0

\_

\*

2

0

1

2

void\_np

void\_p

char\_np

char\_1p

char\_p

int\_np

int\_1p

int\_p

\_

\*

2

0

1

2

2.函数检查

const int a=1, b=2;

int max (int p1, int p2) {

if (p1>p2){

return (a);

}

else return(0);

}

int loop (int a, int b) {

if (a\*b<4) return (a+b);

else return (loop(a-1, b-1));

}

int loop\_l (int a, int b) {

if (a\*b<4) return (a+b);

else return (loop(a-1, b));

}

int loop\_r (int a, int b) {

if (a\*b<4) return (a+b);

else return (loop(a, b-1));

}

void main () {

const int p1=1, p2=2;

int a,b;

int array[3];

array[0]=1; array[1]=2; array[2]=3;

scanf(a); scanf(b);

printf("loop:", loop(a,b));

printf("loop\_l:", loop\_l(a,b));

printf("loop\_r:", loop\_r(a,b));

printf(max(p2, 5));

printf(max(array[2], p1));

printf(max(3, a));

printf(max(b, array[1]));

}

输入：5 6 两个数字

结果：

5

6

loop:

3

loop\_l:

4

loop\_r:

2

0

1

0

1

3.表达式检查

int \_int (int a) {

return (+a+1);

}

char \_char {

return ('K');

}

void main () {

const char c0='a', c1='A', c2='\_';

char c\_array[4];

int zero, a;

c\_array[0]='+';

c\_array[1]='-';

c\_array[2]='\*';

c\_array[3]='/';

zero = 0;

scanf(a);

printf(\_char\*a/5 + c\_array[1]/c1\*\_int(a));

printf(- (c0+a-\_char) \* (c\_array[1]-\_int(a)+6) );

printf(+-4-0/+1+0);

printf(--66/(-(+2++5))--8);

printf(-\_int(a));

printf(+\_char);

}

输入：6 一个数字

结果：

6

90

-1232

-4

-17

-7

K

4.结构检查

int a[3];

void \_while {

int i, j;

i=0; j=0;

printf("while statement");

while (i<=2) {

while (j<3) {

j = j+1;

}

if (i<3) {

j = 0;

printf(j);

}

else ;

switch (i){

case 2: i=666;

default: i = i+1;

}

printf(i);

}

}

void \_if (int i) {

printf("if-else statement");

if (i>0) {

i=3;

while (i>0) {

i = i-1;

printf(i);

}

switch (i){

case 0: printf("if-switch OK");

default: printf("if-switch error");

}

if (i!=0) {}

else printf("if-if OK");

}

else if (i==0) {

printf("else-if OK");

}

else {

i=3;

while (i>=1) {

i = i-1;

printf(i);

}

switch (i){

case 0: printf("else-switch OK");

default: printf("else-switch error");

}

if (i==0) {

printf("else-if OK");

} else;

}

}

void \_switch (int a) {

printf("switch-case statement");

switch (a){

case 1:

{

while (a<3) {

a = a+1;

printf(a);

}

}

case 0:

{

switch (a) {

case 0: printf("switch-switch OK");

default: printf("switch-switch error");

}

}

case -1:

{

if (a<0) {

printf("switch-if OK");

} else;

}

default: {;;{;;;;{}{;{};}};}

}

}

int \_array {

printf("array index");

a[0]=1; a[1]=2; a[2]=3;

printf("3?=",a[a[1]]);

}

void main () {

int a;

scanf(a);

\_while;

\_if(a);

\_switch(a);

\_array;

}

输入：6 一个数字

结果：

6

while statement

0

1

0

2

0

666

if-else statement

2

1

0

if-switch OK

if-if OK

switch-case statement

array index

3?=

3

5.特殊情况检查

void L1 {printf("L1");}

void Label1 {printf("Label1");}

void L\_1 {printf("L\_1");}

void Label\_1 {printf("Label\_1");}

vOid mAin () {

const int t1=1, \_t1=1, t\_1=1, \_t\_1=1;

const int temp1=1, \_temp1=1, temp\_1=1, \_temp\_1=1;

InT AbcD;

aBcd = 666;

AbcD = 1+t1+\_t1+t\_1+\_t\_1+temp1+\_temp1+temp\_1+\_temp\_1;

pRIntf(abCD);

prInTf("ABCDEFGHIJKLMNOPQRSTUVWXYZ abcdefghijklmnopqrstuvwxyz");

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

}

输入：无

结果：

9

ABCDEFGHIJKLMNOPQRSTUVWXYZ abcdefghijklmnopqrstuvwxyz

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

6.错误程序：词法错误

int main () {

const char a='', b='ab', c='%', d=';

if (a<>'5') {

printf("\"");

}

else printf(");

}

7.错误程序：语法错误-缺失

const char b=;

const int c=;

const e = 3;

const int f 4;

char d[3;

int a[];

int f1 (int ) {

int temp

while (f<7 {}

temp = 3+;

}

void f2 {

printf();

d 4] = 1;

while (e<3) }

}

f3 (int a) {}

void f4 {

int main ) {

switch () {

default ;

}

() {}

else

}

8. 错误程序：语法错误-其他

int foo (int a) {

return a;

}

int main () {

const int c=1,d=5;

const;

void a;

int b;

int ;

foo(a,b);

}

9.错误程序：语义错误

char b, e;

int a[3], c;

int e;

int foo (int a) {

return b;

}

void main () {

a[b] = a[3];

c=d;

foo(b);

foo(c) = b;

c=a;

scanf(a);

b=c;

switch (b) {

case 3:;

}

}

10.错误程序：其他错误

int main {

int a = 0;

int b = &a;

return (0);

}

### 2．测试结果分析

正确的程序对基本的文法符号全覆盖，对一些常见的、易错的组合也覆盖到了，还对一些可能的错误编译方案进行了检查。

错误的测试程序覆盖了全部的错误类型，包括一些可能的语义疏忽。

## 五．总结感想

早就听说过编译这门课的难度，要求自己独立实现一个完整的编译器，难免有些畏惧情绪。经历过计组的艰辛，我当时认为编译的课程设计是一个巨大的挑战。直到现在完成了才发现：原来并没有想象中那么的艰难，只要你坚持去做。

刚开始时，我也很迷茫，编译器这么多的部分，每一部分应该如何实现？各个部分之间怎么联系起来？幸运的是，在理论课的学习中，老师会告诉我们一些具体实现的方法，这让我逐步对各个部分有了一定的认识，给了我最初的信心。

在课设开始阶段，我花了大量的时间阅读分析Pascal-S源代码，力争把源代码的每一部分都理解透彻。通过阅读，我一方面赞叹老一辈教授的水平，能用这么简明的程序完成一个编译器，另一方面也发现原来编译器并没有想象中那么的复杂，也学到了不少有用的东西，比如符号表的结构、错误处理、语法制导分析等，这为我之后的实现有很大的帮助。

接下来就开始真正进行编译器设计了。在刚刚拿到文法时，我既激动又紧张。激动可能给更多的是因为新奇，紧张的是要对文法进行仔细的分析。这个文法有哪些易混淆的地方？有没有二义性？有没有不符合递归子程序法的地方？怎么解决这些问题？一系列的问题需要我去解决。为此，我还和很多相同文法的同学进行了交流，一起研究这个文法。最终得出结论：没有二义性，但是有些地方需要向前看多个字符才能解决。很巧的是，第一次课上抽查就抽到了我，还是杨老师亲自检查。杨老师人很好，我也借着这次机会向杨老师请教了一些问题。

接下来的几周，我按部就班的开始编程。在这之中，我也遇到了很多问题，比如代码结构、错误处理、符号表构建、运行时的内存管理等等。我通过查找Pascal-S代码和与同学交流，逐步摸索清楚了脉络，接下来的编程就容易很多了。在这之中，我又翻阅了C++的相关书籍，学习了一些C++的用法，优化了自己的代码，方便了编程。在接下来的测试代码评审时，我也积极参与，提出了很多问题，助教也都一一解答了，让我收获很大。

接下来就是紧张的测试环节了，很幸运的是，我的代码没有太大的问题。之后我又针对出现的问题进行了细微的调整，成为了现在这个最终版本。

回顾整个历程，我想说我很感谢这门课程，我不仅学到了编译器的相关知识，同时还锻炼了我开发一个上千行系统的能力，以及使用C++编程的能力。我在这门课程中逐渐建立了信心，最终提高了水平。

最后，我想提一个建议：如果可以的话，把课程设计的时间再靠前一些，语义分析和代码生成只有一周还是挺紧张的，如果多给一周的时间可能完成的更好。