

《编译技术》课程设计 文 档

学号：14061024

姓名：刘恒睿

2016 年 12 月 9 日

目录

一. 需求说明.....	3
1. 文法说明.....	3
2. 目标代码说明.....	5
3. 优化方案.....	6
二. 详细设计.....	7
1. 程序结构.....	7
2. 类/方法/函数功能.....	7
3. 调用依赖关系.....	15
4. 符号表管理方案.....	15
5. 存储分配方案.....	16
6. 解释执行程序.....	18
7. 四元式设计*.....	18
8. 目标代码生成方案.....	19
9. 优化方案.....	20
10. 出错处理.....	23
三. 操作说明.....	24
1. 运行环境.....	24
2. 操作步骤.....	24
四. 测试报告.....	25
1. 测试程序及测试结果.....	25
2. 测试结果分析.....	41
五. 总结感想.....	42

一. 需求说明

1. 文法说明

抽取的文法为**扩充C0文法**，总体如下：

<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> ']' }

<常量> ::= <整数> | <字符>

<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>; | <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>]
 <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>
 <情况语句> ::= switch ‘(’ <表达式> ‘)’ ‘{’ <情况表> [<缺省>] ‘}’
 <情况表> ::= <情况子语句>{<情况子语句>}
 <情况子语句> ::= case <常量>: <语句>
 <缺省> ::= default: <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> ‘)’]

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

2. 目标代码说明

本程序生成的为 MIPS 体系结构下的汇编代码, 具体用到的 MIPS 指令如下表¹:

Format	Purpose	Description
ADD rd, rs, rt	To add 32-bit integers. If an overflow occurs, then trap.	$GPR[rd] \leftarrow GPR[rs] + GPR[rt]$
ADDI rt, rs, immediate	To add a constant to a 32-bit integer. If overflow occurs, then trap.	$GPR[rt] = GPR[rs] + \text{immediate}$
BEQ rs, rt, offset	To compare GPRs then do a PC-relative conditional branch	if $GPR[rs] = GPR[rt]$ then branch
BGEZ rs, offset	To test a GPR then do a PC-relative conditional branch	if $GPR[rs] \geq 0$ then branch
BGTZ rs, offset	To test a GPR then do a PC-relative conditional branch	if $GPR[rs] > 0$ then branch
BLEZ rs, offset	To test a GPR then do a PC-relative conditional branch	if $GPR[rs] \leq 0$ then branch
BLTZ rs, offset	To test a GPR then do a PC-relative conditional branch	if $GPR[rs] < 0$ then branch
BNE rs, rt, offset	To compare GPRs then do a PC-relative conditional branch	if $GPR[rs] \neq GPR[rt]$ then branch
DIV rs, rt	To divide a 32-bit signed integers	$(HI, LO) \leftarrow GPR[rs] / GPR[rt]$
JAL target	To execute a procedure call within the current 256 MB-aligned region	$I: GPR[31] \leftarrow PC + 8$ $I+1: PC \leftarrow PCGPRLEN-1..28 \parallel \text{instr_index} \parallel 02$
JR rs	To execute a branch to an instruction address in a register	$PC \leftarrow GPR[rs]$
LW rt, offset(base)	To load a word from memory as a signed value	$GPR[rt] \leftarrow \text{memory}[GPR[base] + \text{offset}]$
MUL rd, rs, rt	To multiply two words and write the result to a GPR.	$GPR[rd] \leftarrow GPR[rs] \times GPR[rt]$

¹ 摘自 MIPS32® Architecture For Programmers Volume II: The MIPS32® Instruction Set

SLT rd, rs, rt	To record the result of a less-than comparison	$GPR[rd] \leftarrow (GPR[rs] < GPR[rt])$
SUB rd, rs, rt	To subtract 32-bit integers. If overflow occurs, then trap	$GPR[rd] \leftarrow GPR[rs] - GPR[rt]$
SW rt, offset(base)	To store a word to memory	$memory[GPR[base] + offset] \leftarrow GPR[rt]$
SYSCALL	To cause a System Call exception	A system call exception occurs, immediately and unconditionally transferring control to the exception handler. The code field is available for use as software parameters, but is retrieved by the exception handler only by loading.

目标代码采用 Mars 4.5 模拟器运行。

3. 优化方案

本程序采用了 3 种优化，分别是复制传播优化，图着色法分配全局寄存器，循环队列分配临时寄存器。

采用复制传播的优化基于以下的中间代码前提，即在我们的大部分语法成分中，<表达式>是用来计算的最终形式，但是在大部分的时候，表达式通常只会含有一个项，项中只含有一个因子，这样会导致大量的 ASSIGN 语句。例如一条简单的 $a=b$ ，会产生以下 4 条四元式：

```

ASSIGN b      .fac2
ASSIGN .fac2   .te2
ASSIGN .te2    .exp2
ASSIGN .exp2   a

```

而我们明显可以以上四条四元式合并成一条：

```

ASSIGN      b      a

```

复制传播的优化就是用来解决上述问题。

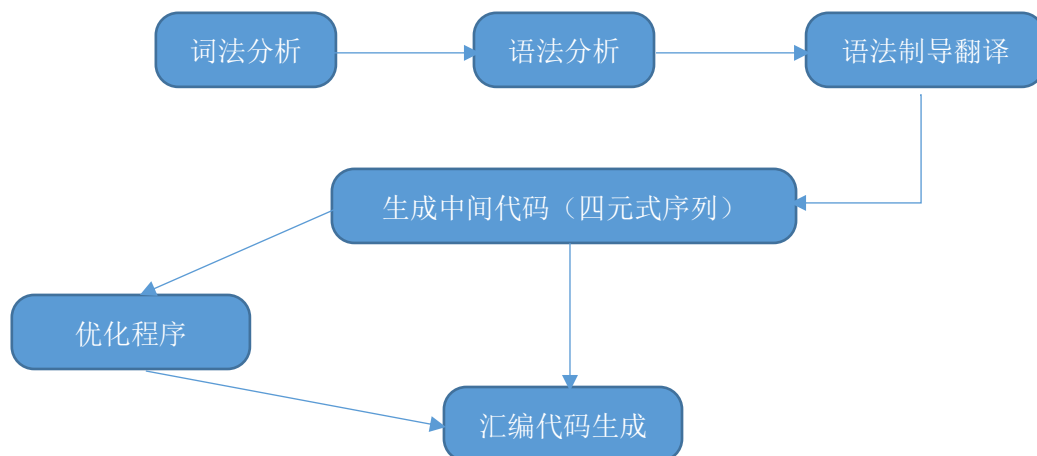
全局寄存器的分配采用图着色法，首先需要划分基本块进行活跃变量分析。计算出每一个出入口之间的冲突变量。然后构建冲突图。最后采用图着色法针对冲突图进行全局寄存器的分配。

临时寄存器分配采用循环队列的方法，因为临时寄存器的作用域较小，生命周期短，而

MIPS 中有 10 个临时寄存器，组成循环队列已经足够。所以采用该方法进行临时寄存器分配。

二. 详细设计

1. 程序结构



2. 类/方法/函数功能

1) class: lexical_analysis

功能：词法分析类，用于对源程序的词法进行分析。

包含方法：

a) private:

```
bool isLetter(char ch);    //判断是否为字母
bool isColon(char ch);    //判断是否为冒号
bool isBigger(char ch);   //判断是否为大于号
bool isLess(char ch);     //判断是否为小于号
bool isSurprise(char ch); //判断是否为感叹号
bool isEqu(char ch);      //判断是否为等号
bool isPlus(char ch);     //判断是否为加号
bool isMinus(char ch);    //判断是否为减号
bool isMul(char ch);      //判断是否为乘号
```

```

bool isLpar(char ch);    //判断是否为左括号
bool isRpar(char ch);    //判断是否为右括号
bool isLbraket(char ch); //判断是否为左方括号
bool isRbraket(char ch); //判断是否为右方括号
bool isLbrace(char ch);  //判断是否为左大括号
bool isRbrace(char ch);  //判断是否为右大括号
bool isComma(char ch);   //判断是否为逗号
bool isSemi(char ch);    //判断是否为分号
bool isDivi(char ch);    //判断是否为除号
bool isSquote(char ch);  //判断是否为单引号
bool isDquote(char ch);  //判断是否为双引号
int reserver(string word); //判断是否为保留字
int transNum(string num); //将字符串转换为数字
int error(int type);      //报错程序，将错误原因放入 error_type 变量中
                           供上层语法分析程序使用。

```

b) **public:**

```

int getsym(string words); //从一行字符串中分析一个单词出来，当前分析位
                           //置会被记录

void print_sym(ofstream &outfile); //打印分析的单词，调试用

int getnum();              //返回分析得到的数字
int getsymbol();          //返回分析得到的单词类型
int step_once();          //向后分析一个单词
unsigned int getline();    //返回当前分析到的行号
unsigned int getcol();     //返回当前分析到的列号
std::string gettoken();    //返回当前分析到的单词
inline int get_err() { return error_type; } //返回错误编号
lexical_analysis();        //类构造函数
~lexical_analysis();       //类析构函数

```

2) **class: parser**

功能：语法分析程序，对于词法分析出的单词进行语法成分分析，并调用符号

表类生成符号表。生成中间代码并调用 `midcode` 类保存中间代码。

包含方法:

a) `private`:

```
bool getsym();           //词法分析执行一次
void program();          //Analysis entry
void con_state();        //constant statement
void var_state();        //variable statement
void func_re();           //function with return value
void func_vo();          //function without return value
void func_main();        //main fuction
void con_def();          //constant definition
void var_def();          //variable definition
void def_head();         //head of a function definition
void func_para();        //parameters of a function
void compound_state();    //compound statement
void para_list();        //parameters list
void statement_col();    //statements column
void statement();
void _if();
void _while();
void assign();           //assign statement
void _read();
void _write();
void _switch();
void _return();
void condition();        //condition in if/while
void condition_list();
void _default();
void condition_sub();    //substatement of a condition list
void value_paras();      //parameters value list
```

```

void expression();    //表达式

void term();

void factor();

void _integer();

void funcr_call();    //有返回值函数调用

void funcv_call();    //无返回值函数调用

void error(int type); //出错报错

//以下为生成临时变量以及label

inline void gen_label()

{

    qlabel = ".label" + numT0strs(label_num);

    label_num++;

}

inline void gen_temp()

{

    qtemp = ".t" + numT0strs(t_num);

    t_num++;

}

inline void gen_exp()

{

    qexp = ".exp" + numT0strs(exp_num);

    exp_num++;

}

inline void gen_term()

{

    qterm = ".te" + numT0strs(term_num);

    term_num++;

}

inline std::string gen_fac()

{

```

```

    qfac = ".fac" + numT0strs(fac_num);

    fac_num++;

    return qfac;
}

inline std::string gen_str()
{
    strings_name = ".String" + numT0strs(str_num);

    str_num++;

    return strings_name;
}

inline void cln_temp() { t_num = 0; }

inline void cln_exp() { exp_num = 0; }

inline void cln_term() { term_num = 0; }

inline void cln_fac() { fac_num = 0; }

inline std::string get_label() { return qlabel; }

inline std::string get_temp() { return qtemp; }

inline std::string get_exp() { return qexp; }

inline std::string get_term() { return qterm; }

inline std::string get_fac() { return qfac; }

```

b) **public:**

```

void analysis(int need);    //语法分析

parser(std::string file_name); //构造函数

~parser();                //析构函数

```

3) **class: midcode**

功能：生成四元式的类，可以接受生成四元式的指令并将四元式保存在其中。

包含方法：

a) **private:**

```

void search_string(); //寻找程序中需要打印的字符串放在全局变量区

int get_temp(std::string var_name); //返回一个临时变量对应的寄存器

int get_save(std::string var_name); //返回一个全局变量对应的寄存器

```

```
int get_save_op(std::string var_name); //返回经过全局寄存器分配后一个全局变量对应的寄存器。
```

```
void s_wri_del(); //将全局寄存器内存写回内存并删除分配记录  
void t_wri_del(); //将临时寄存器内存写回内存并删除分配记录  
void s_wri(); //将全局寄存器内存写回内存  
void s_del(); //删除全局寄存器分配记录  
void pop_stack(); //调用函数后运行栈出栈  
void push_stack(); //调用函数前运行栈入栈  
void load_para(int paras); //调用函数前装载参数  
void clear_varTable(); //清理变量寄存器分配表  
Symbol_table *table; //符号表  
void error(int type, std::string var_name); //错误处理  
BB_Manager* BBm = nullptr; //基本块管理  
bool needop = false //是否需要优化
```

b) public:

```
void insert(string op, string a, string b, string res); //插入一条四元式
```

```
void gen_code(Symbol_table *table); //生成目标代码  
void gen_globaldata(); //生成程序中全局变量  
void printMidcode(); //打印中间代码  
void init_BB(); //初始化基本块  
void setOP(); //设置优化开关  
midcode(); //构造函数  
~midcode();
```

4) class: symbol_table

功能: 创建符号表并且管理符号表。

包含方法:

private:

```
std::vector<Item> global_var; //全局变量表
```

```
std::map< std::string, std::vector<Item> > func_table; //函数变量表
```

```

public:

void insert_global(std::string name, Type type, int size, int address,
bool isArray, bool isConst, int para_num, int position, int offset);

    //插入全局变量

void insert_func(std::string func_name, std::string name, Type type, int
size, int address, bool isArray, bool isConst, int para_num, int position,
int offset);

    //插入函数中的局部变量

Item* glo_find(std::string name); //全局查找变量

Item* func_find(std::string func_name, std::string var_name); //函数中查找
变量

int find_offset(std::string func_name); //求变量偏移量

void printTable(); //打印符号表

Symbol_table();

~Symbol_table();

```

5) class: BB_Manager

```

private:

vector<Basic_Block> BBs; //基本块

map<string, map<string, int>> func_reg_alloc; //函数寄存器分配表

int regs_num = 7; //全局寄存器数量

void count_inout(); //活跃变量分析

int _process(int i); //图着色法分配全局寄存器

public:

BB_Manager(vector<Quaternary> origin_mid); //通过四元式构造基本块

vector<Quaternary> get_op(); //返回优化完成的四元式

int find_reg(string funcname, string var); //寻找对应的寄存器

void Print_Block(); //打印基本块

~BB_Manager();

```

6) class: Basic_Block

```

private:

```

```

vector<Quaternary> origin_mid; //基本块内四元式

public:

vector<int> pre;      //前驱

vector<int> next;     //后继

vector<string> in;    //活跃变量

vector<string> out;

vector<string> use;

vector<string> def;

void count_usedef(); //计算use和def集合

void peephole();     //复制传播优化

vector<Quaternary> get_origin_mid(); //返回四元式

inline Quaternary get_first_oq()      //基本块的第一个四元式
{
    return origin_mid[0];
}

inline Quaternary get_last_oq()        //基本块的最后一个四元式
{
    return origin_mid.back();
}

inline Quaternary get_ith_oq(int i)    //第i个四元式
{
    if (i >= origin_mid.size())
    {
        return origin_mid[0];
    }

    return origin_mid[i];
}

~Basic_Block();

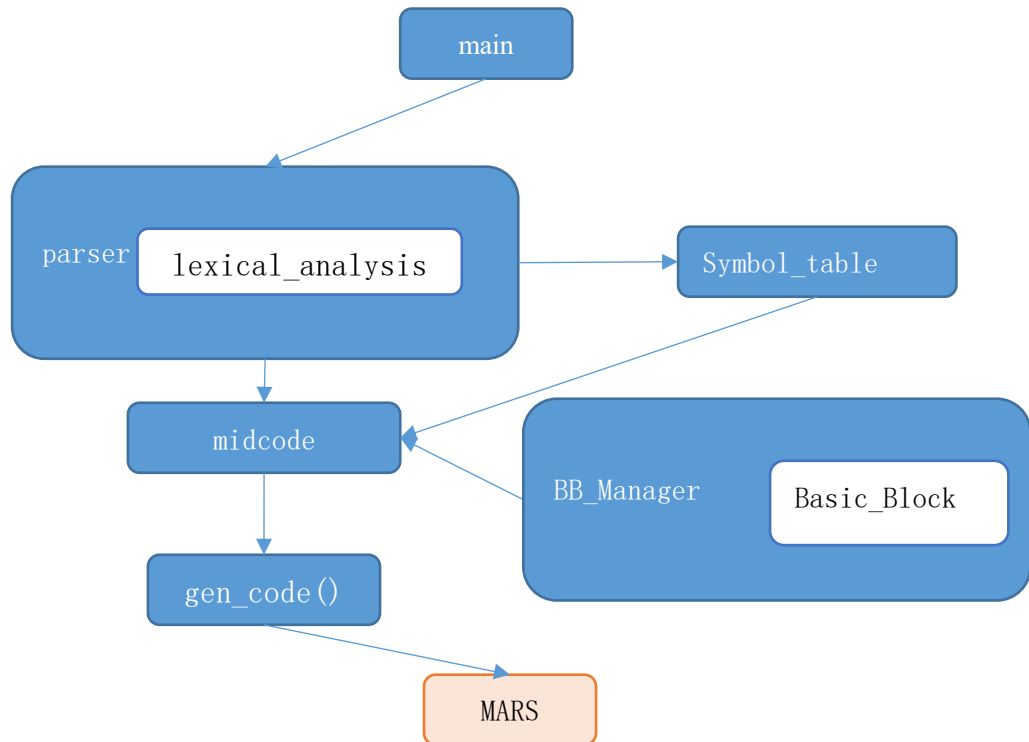
```

7) function: main

功能：统一调用各个函数以及类，完成完整的编译功能。

3. 调用依赖关系

类调用关系图：



4. 符号表管理方案

符号表分为两种表，一张表是全局变量表，包括定义的全局变量以及函数等等，另外一种表为函数符号表，为每一个函数单独建立一张符号表来保存其定义的变量。

表项定义如下：

```
enum Type
{
    VAR_int,           //variable
    VAR_char,
    FUNC_V,            //header of void function
    FUNC_R_int,        //header of function with return value
    FUNC_R_char,
    PAR_int,           //parameter of a function
    PAR_char,
    STR_p
};
```

```

struct Item
{
    std::string name;
    Type type;
    int size;           //Array size,if it's not array,set 1.
    int address;        //record the address of global var and const,
    bool isArray;
    bool isConst;
    int para_num;       //number of a parameter in a function
    int position;       //position in origin code
    int offset;
};

```

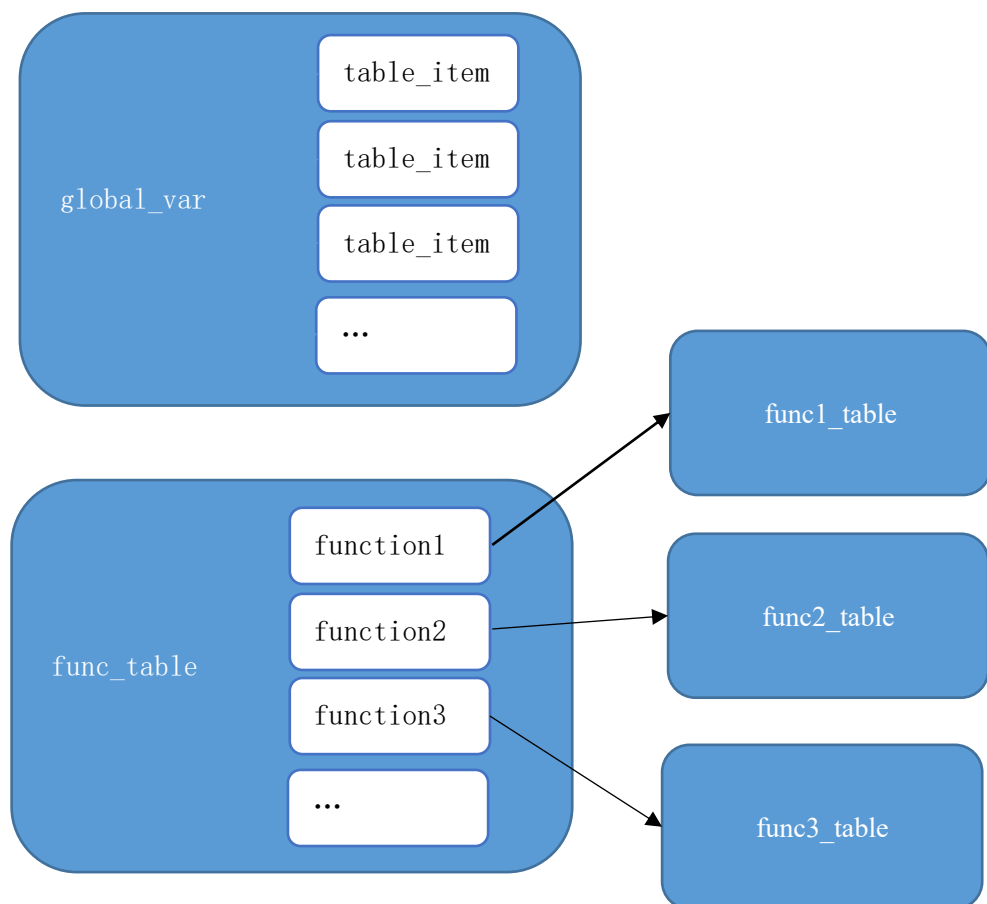
符号表的定义如下：

```
std::vector<Item> global_var;
```

```
std::map< std::string, std::vector<Item> > func_table;
```

map 的键为函数名，值为一张符号表。

一个程序的整体符号表结构如下图：



5. 存储分配方案

1) 存储组织

程序的存储分为以下几种。

a. 全局变量以及常量

采用 MIPS 汇编命令 `.data` 生成并存放在内存区域中。

b. 局部变量以及常量

在栈区为其分配地址存储。

c. 函数参数

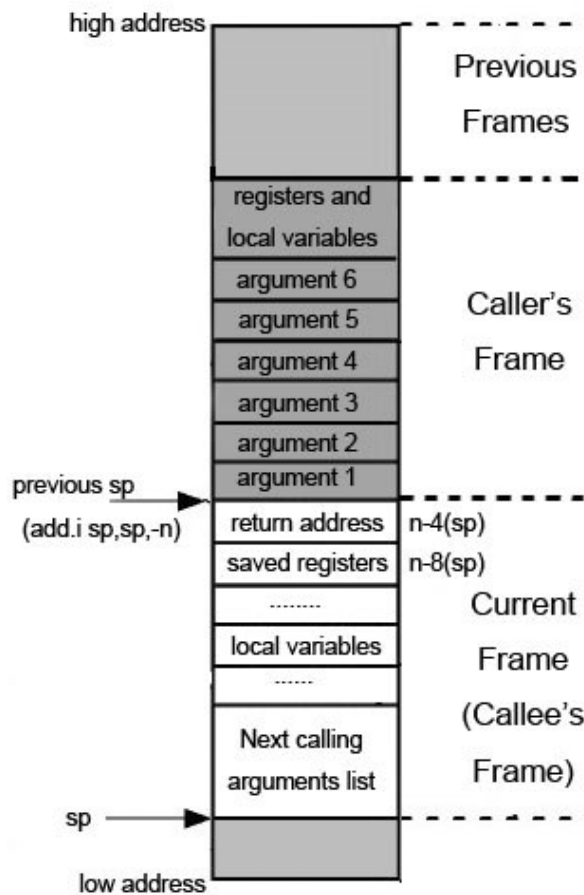
少于 4 个的参数存放在 MIPS 约定的参数寄存器中，多余的存入栈中。

d. 常数

不分配内存，直接加载进临时寄存器中进行运算。

2) 运行栈结构

在每次进行函数调用时，需要将当前寄存器中的值压入栈中，保存下来，并通过 `jal` 指令进行函数调用，`jr` 指令进行函数返回。



6. 解释执行程序

具体内容参看链接：

<http://courses.missouristate.edu/kenvollmar/mars/index.htm>

7. 四元式设计*

四元式的结构定义如下：

```
typedef struct {  
    string op;  
  
    string res;  
  
    string a;  
  
    string b;  
  
} Quaternary;
```

1) 四元式的详细定义如下表：

类别	四元式	含义
基本算术运算	ADD OP1 OP2 RST	$RST = OP1 + OP2$
	SUB OP1 OP2 RST	$RST = OP1 - OP2$
	MUL OP1 OP2 RST	$RST = OP1 * OP2$
	DIV OP1 OP2 RST	$RST = OP1 / OP2$
赋值运算	ASSIGN OP1 / RST	$RST = OP1$
	ASSFA OP1 OP2 RST	$RST = OP1[OP2]$
	ASSTA OP1 OP2 RST	$RST[OP2] = OP1$
逻辑判断	BEQ OP1 OP2 label	$OP1 = OP2$ 则跳转至 label
	BNE OP1 OP2 label	$OP1 \neq OP2$ 则跳转至 label
	BL OP1 OP2 label	$OP1 > OP2$ 则跳转至 label
	BS OP1 OP2 label	$OP1 < OP2$ 则跳转至 label
	BLE OP1 OP2 label	$OP1 \geq OP2$ 则跳转至 label
	BSE OP1 OP2 label	$OP1 \leq OP2$ 则跳转至 label
跳转	GO // label	无条件跳转至 label
	SET // label	设置标签
函数相关	BEGIN // func	函数 func 开始
	END // func	函数 func 结束
	CALL // func	调用无返回值函数 func
	CALL / OP2 func	调用含返回值 OP2 函数 func
	RET ///	无返回值函数返回
	RET // value	有返回值函数返回 value

	PUSH // RST	参数传递
定义相关	CONST type value name	定义常量
	type ² // name	定义变量
	type len / name	定义数组
	global ///	标志全局常/变量定义开始
	text ///	text 段开始
读写以及输出	RD // DST	读入数据
	PD str value /	打印数据
	PD // SRC	
	PD / c SRC	

2) 四元式生成设计

在进行语义分析的时候生成四元式。

3) 四元式转换为 MIPS 汇编

具体转换形式参看 midcode.cpp 的 1647-2442 行。

```
for (std::vector<Quaternary>::iterator it = code_mid.begin(); it != code_mid.end(); it++)
{
    if (it->op == "global" || it->op == "text") { ... }
    else if (it->op == "ASSIGN") { ... }
    else if (it->op == "ASSFA") { ... }
    else if (it->op == "ASSTA") { ... }
    else if (it->op == "ADD") { ... }
    else if (it->op == "SUB") { ... }
    else if (it->op == "MUL") { ... }
    else if (it->op == "DIV") { ... }
    else if (it->op == "BEQ") { ... }
    else if (it->op == "BNE") { ... }
    else if (it->op == "BL") { ... }
    else if (it->op == "BS") { ... }
    else if (it->op == "BLE") { ... }
    else if (it->op == "BSE") { ... }
    else if (it->op == "GO") { ... }
    else if (it->op == "SET") { ... }
    else if (it->op == "BEGIN") { ... }
    else if (it->op == "RET") { ... }
    else if (it->op == "END") { ... }
    else if (it->op == "CALL") { ... }
    else if (it->op == "PUSH") { ... }
    else if (it->op == "PD") { ... }
    else if (it->op == "RD") { ... }
    else if (it->op == "const") { ... }
}
```

8. 目标代码生成方案

直接根据四元式生成 MIPS 汇编代码。

首先扫描所有四元式将可能要打印的字符串放到.global 区域, 以及为全局变量分配空间。局部变量不会显示的分配空间, 通过编译器维护运行栈, 在使用时查找符号表在运行栈上分配和使用对应的空间。

² type 为 int 或 char, 下同。

函数调用时首先先 `pushstack` 操作保存当前运行栈，然后通过 `load_para` 传递参数，最后通过 `jal` 指令跳转至函数，返回后通过 `popstack` 恢复现场。

9. 优化方案

1) 基本块设计

基本块的核心由 6 个 `vector` 维护的数据结构构成：

```
vector<Quaternary> origin_mid;

vector<int> pre;

vector<int> next;

//live-variable analysis

vector<string> in;

vector<string> out;

vector<string> use;

vector<string> def;
```

`origin_mid` 中为基本块所包含的四元式，`pre` 和 `next` 为基本块的前驱和后继。

`in`, `out`, `use`, `def` 为活跃变量分析所用。

2) 基本块管理

基本块由 `BB_manager` 类划分和管理。同时该类还负责通过活跃变量分析生成冲突图。

划分规则参照书上规则，并针对我自己的四元式做了一些修改，具体规则如下：

1. `BEGIN SET` 为基本块的开头。
2. `BEQ BNE BL BS BLE BSE GO END RET` 为基本块的结尾。
3. `CALL PUSH RD PD` 作为单独基本块。

划分出的基本块类似下图：

```

Block 33
pre: 32
next: 31
in: i tnum mm
out: i tnum mm a
use: i tnum
def: a .te5
ASSIGN i .exp3
ASSIGN tnum .exp4
ASSTA .exp4 .exp3 a
ASSIGN i .te5
ASSIGN 1 .te6
ADD .te5 .te6 .te5
ASSIGN .te5 i
GO .label16

```

3) 求 In 和 Out 集合的算法

此算法与编译原理书上的算法并无二致，具体划分可参见编译原理教科书，这里只展示一下划分结果（见上图）。

因为临时寄存器不参与分配，所以这里计算的时候忽略了临时寄存器（以“.”开头）。

4) 全局寄存器分配

采用图着色这种启发式算法进行全局寄存器的分配。分配后存入 `map<string, map<string, int>> func_reg_alloc` 中。记录如下：

```

quicksort:
a: 6
i: 2
j: 3
max: 5
min: 4
ttt: 0
val: 1
main:
a: 1
i: 0
mm: 2
tnum: 3

```

5) 复制传播算法

该优化着重于尽量用一个变量表达相同含义的变量，以删除不必要的四元式，由于在较小的程序中这样的情况所占比重很大，优化效果显著，可减少约 1/3 的代码量。

优化前后效果见下图：

优化前：

```
BEGIN                quicksort
ASSIGN min           .fac1
ASSIGN .fac1         .te1
ASSIGN .te1          .exp1
ASSFA a .exp1 .fac0
ASSIGN .fac0         .te0
ASSIGN .te0          .exp0
ASSIGN .exp0         val
ASSIGN min           .fac2
ASSIGN .fac2         .te2
ASSIGN .te2          .exp2
ASSIGN .exp2         i
ASSIGN max           .fac3
ASSIGN .fac3         .te3
ASSIGN 1             .fac4
ASSIGN .fac4         .te4
SUB .te3 .te4 .te3
ASSIGN .te3          .exp3
ASSIGN .exp3         j
ASSIGN max           .fac5
ASSIGN .fac5         .te5
ASSIGN min           .fac6
ASSIGN .fac6         .te6
SUB .te5 .te6 .te5
ASSIGN .te5          .exp4
ASSIGN 1             .fac7
ASSIGN .fac7         .te7
ASSIGN .te7          .exp5
BSE .exp4 .exp5 .label1
ASSIGN i             .fac8
ASSIGN .fac8         .te8
ASSIGN .te8          .exp6
ASSIGN .exp6         ttt
SET                 .label2
```

优化后：

```
BEGIN                quicksort
ASSIGN min           .exp1
ASSFA a .exp1 .fac0
ASSIGN .fac0         val
ASSIGN min           i
ASSIGN max           .te3
ASSIGN 1             .te4
SUB .te3 .te4 .te3
ASSIGN .te3          j
ASSIGN max           .te5
ASSIGN min           .te6
SUB .te5 .te6 .te5
ASSIGN .te5          .exp4
ASSIGN 1             .exp5
BSE .exp4 .exp5 .label1
ASSIGN i             ttt
SET                 .label2
```

10. 出错处理

1) 出错处理方案

当编译程序发现错误以后，尽可能的设法把错误限制在一个局部的范围内，避免错误扩散和影响程序的其他部分的分析和检查。

当检测到错误后，编译程序就暂停对后面的符号进行分析，跳过错误所在的语法单位。比如在词法分析时跳过出错的单词；在语法分析时，跳过错误所在的短语或者语句，找到下一个新的短语或者语句进行分析。

2) 错误信息以及含义

编号	错误信息	错误原因
1	Number begin with 0	数字以 0 开头
2	Unknown Symbol	无法识别的符号
3	Missing '	此处缺少单引号
4	Missing "	此处缺少双引号
5	Unknown char	无法识别的 <code>char</code> 类型声明
6	Unknown string	无法识别的字符串
7	Missing =	此处缺少=
11	Missing ;	此处缺少分号
12	Should be char or int	应该是 <code>char</code> 或者 <code>int</code>
13	Should be 'const'	应该是 <code>const</code>
14	Missing }	
15	Missing {	
16	Missing (
17	Missing)	
18	Should be identifier	
19	Missing void	
20	Missing main	
21	Should be a assign op	
22	This type should have a char	缺少字符

23	Missing]	
24	Missing [
25	Should be integer	
26	Unmatch op	
27	Unknown symbol	
28	Write format wrong	
29	Missing :	
30	Unknown factor.	
31	duplicate definition	变量已经被定义
32	duplicate parameter	
33	should not assign to a const	
34	unmatch number of parameters	
35	void function should not return a value	
36	Should be a function with return value.	
37	Should be a variable	

三. 操作说明

1. 运行环境

本程序采用 C++ 编写，在 VS2016 环境下以及 Code_Blocks 环境下均编译测试通过。

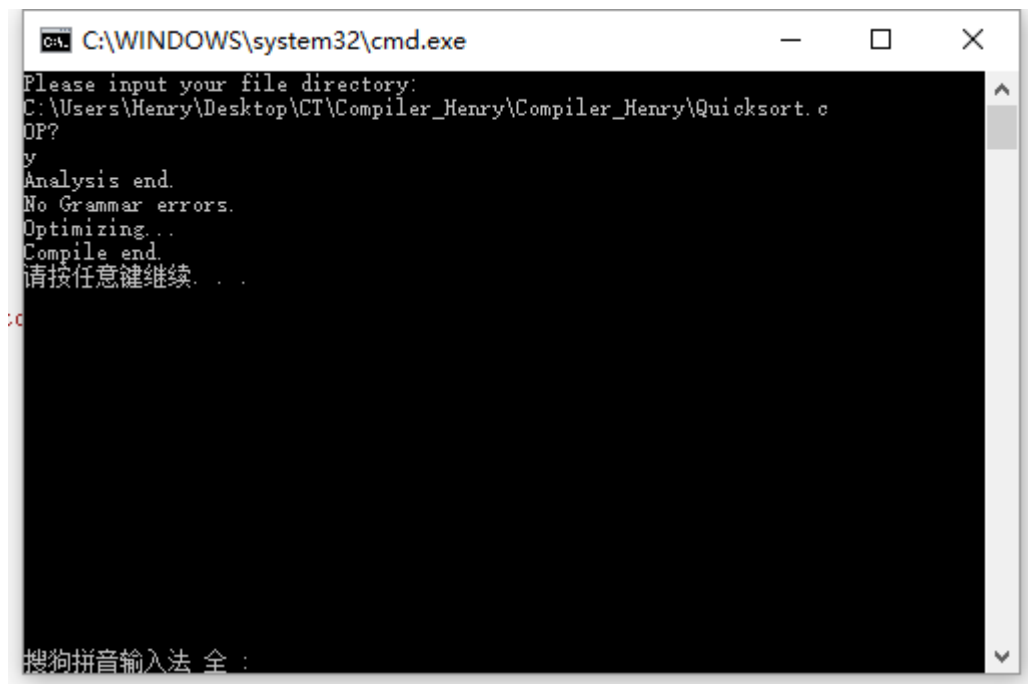
生成的目标代码在工程目录中的 aimcode.txt 中，可直接复制到 Mars4.5 模拟器中解释执行。

2. 操作步骤

- 1) 编译并运行（以 VS2016 为例）：



- 2) 输入路径以及是否进行优化，y 表示优化，其他表示不优化（注意，如果选择不优化也会默认进行临时以及全局寄存器分配，只不过用循环队列方法分配）：



- 3) 程序结束

四. 测试报告

1. 测试程序及测试结果

- 1) Quicksort.c

A. 程序代码以及说明

该程序接受 10 个数字输入并排好序输出。

```
int a[10];

void QuickSort(int min,int max)
{
    int i,j;
    int val,ttt;
    val = a[min];
    i=min;
    j=max-1;
    if(max-min>1)
    {
        ttt = i;
        while(i<j)
        {
            while(j > i)
            {
                if(a[j]<val)
                {
                    a[i]=a[j];
                    i = i +1;
                    ttt = j;
                    j = j - 20;
                }
                j = j -1;
            }
            if(j >= 0)
            {
                ttt = j;
            }
        }
        j = ttt;
```

```

        while(i < j)
        {
            if(a[i]>val)
            {
                a[j]=a[i];
                ttt = i;
                i = i +20;
            }
            i = i+1;
            if(i < 10)
                ttt = i;
        }
        i = ttt;
    }
    a[i]=val;
    QuickSort(min,i);
    QuickSort(i+1,max);
}

return;
}

void main()
{
    int i,mm, tnum;

    i=0;
    while(i<10)
    {
        scanf(tnum);

        a[i] = tnum;

        i = i+1;
    }
}

```

```

    }

    mm=0;
    QuickSort(mm,10);
    i=0;

    while(i<10)
    {
        printf(a[i]);
        printf(" ");
        i = i+1;
    }
    return;
}

```

B. 测试结果

输入 1: 1 2 3 4 5 6 7 8 9 0

输出: 0 1 2 3 4 5 6 7 8 9

输入 2: 4 3 2 1 5 6 7 9 0

输出: 0 1 2 3 4 5 6 7 8 9

结果符合预期。

2) test_ct.c

A. 程序代码以及说明

该程序力求覆盖所有语法成分。

```

/*测试常量声明*/

const int _const_1 = +108;

const int _const_2 = -101, _const_4 = 5;

const char _const_c_1 = '+', _const_c_2 = '_', _const_c_3 = 'a';

/*测试变量声明*/

int _var_1;

int _var_2, _int_array[10];

```

```

char _var_4,_char_array[10];

/*测试函数定义*/

int _return_0(){return (0);}

/*return a*b*/

int _return_ab(int a, int b)
{
    int temp;
    if (a == 0)
        return (b);
    else
    {
        a = a - 1;
        temp = _return_ab(a,b);
    }
    return (b + temp);
}

void _print_na(char a, int b)
{
    while(b>0)
    {
        b = b - 1;
        printf(a);
    }
}

/*echo a*/

char _echo_4(char a)
{
    return (a);
}

void main()

```

```

{
    const int _const_1 = 101;

    int _varm_1, _;

    /*test if*/

    if(_return_0() != 0)

        _print_na(_const_c_2,_const_4);

    else

        _varm_1 = _return_0();

    /*test while*/

    while(_varm_1 < 10)

    {

        _varm_1 = _varm_1 + 1;

        _int_array[_varm_1-1] = _varm_1 - 1;

        _varm_1 = _int_array[_varm_1-1] + 1;

        _print_na(_const_c_3,_const_4);

    }

    printf("\n");

    printf("_varm_1:",_varm_1);

    printf("\n");

    /*test sentence lines*/

    {

        _varm_1  =  ++1-+2*_const_1-( _echo_4(_const_c_1)+1)/4*2+'a'-

's'*_int_array[_varm_1-2]; /*1046*/

    }

    printf("_varm_1:",_varm_1);

    printf("\n");

    printf("1:",_int_array[1]); /*=1*/

    printf("\n");

    _var_4 = _echo_4(_const_c_1);

    printf("+:",_var_4);

```

```

printf("\n");

_int_array[0] = _return_ab(_var_4, _int_array[3]); /*=30*/

printf("132:", _int_array[0]);

printf("\n");

/*test switch*/

switch (_varm_1)

{

    case -1046:

        {

            printf("Right!", _int_array[2]);

        }

    default:

        {

            printf("Wrong!");

        }

}

/*test read*/

printf("\n");

scanf(_var_1, _var_2, _);

printf(_var_1);

}

```

B. 测试结果

输入： 3 5 6

输出：

```

aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
_varm_1:10
_varm_1:-1046
1:1
+:+

```

132:132

Right!2

3

结果符合预期。

3) test_ct1.c

A. 程序代码以及说明

该程序重点测试了输出。

```
/*测试常量声明*/

const int _const_1 = +101;

const int _const_2 = -101, _const_3 = 0, _const_4 = 5;

const char _const_c_1 = '+', _const_c_2 = '_', _const_c_3 = 'a';

/*测试变量声明*/

int _var_1;

int _var_2, _int_array[10];

char _var_4, _char_array[10];

/*测试函数定义*/

int _return_0(){return (0);}

void main()

{

    const int _const_1 = 101;

    _var_4 = 67;

    printf("should not happend!!");

    printf(_const_1);

    printf("43:", _var_4);

    _var_2 = _return_0();

    printf("0:", _var_2);

    printf("0:", _return_0());

    return;

}
```


B. 测试结果

should not happend!!10143:C0:00:0

结果符合预期。

4) test_switch.c

A. 程序代码以及说明

本程序重点测试 switch

```
void main()
{
    int num,temp1,temp2;
    scanf(num);
    switch(num){
        case 0:printf("result:0\n");
        case 1:{printf("result:1\n");}
        case 2:{scanf(temp1,temp2);printf("result:\n",temp1+temp2);}
        default:
            {
                temp2=0;
                printf("result:\n",temp2);
            }
    }
}
```

B. 测试结果

输入 1: 0

输出: result:0

输入 2: 1

输出: result:1

输入 3: 2 3 4

输出:

result:

输入 4:

输出:

result:

0

5) test.c

A. 程序代码及说明

本程序主要测试寄存器分配。

```
int a;  
int b;  
int c;  
int d;  
int e;  
int f;  
int g;  
int h;  
int i;  
int j;  
int k;  
int l;  
int m;  
int n;  
int o;  
int p;  
int q;
```

```
void main()  
{  
    a = 1;  
    b = 1;  
    c = 1;
```

$d = 1;$

$e = 1;$

$f = 1;$

$g = 1;$

$h = 1;$

$i = 1;$

$j = 1;$

$k = 1;$

$l = 1;$

$m = 1;$

$n = 1;$

$o = 1;$

$p = 1;$

$q = 1;$

$a = b + c;$

$a = a + d;$

$a = a + e;$

$a = a + f;$

$a = a + g;$

$a = a + h;$

$a = a + i;$

$a = a + j;$

$a = a + k;$

$a = a + l;$

$a = a + n;$

$a = a + m;$

$a = a + q;$

$a = a + o;$

$a = a + p;$

$a = a + a;$

```

        printf(a);
    }

```

B. 测试结果

输出: 32

6) error1.c

A. 程序代码及说明

程序缺少分号

```

/*测试常量声明*/

const int _const_1 = +101;

const int _const_2 = -101, _const_3 = 0, _const_4 = 5;

const char _const_c_1 = '+', _const_c_2 = '_', _const_c_3 = 'a'

/*测试变量声明*/

int _var_1;

int _var_2, _int_array[10];

char _var_4, _char_array[10];

/*测试函数定义*/

int _return_0(){return (0);}

void main()

{

    const int _const_1 = 101;

    _var_4 = 67;

    printf("should not happend!!");

    printf(_const_1);

    printf("43:", _var_4)

    _var_2 = _return_0();

    printf("0:", _var_2);

    printf("0:", _return_0());

    return;

```

```
}
```

B. 测试结果

```
Find error! E:
11:Missing ;
Information:
symbol: int in line: 5 col: 3
Symbol: 6

请按任意键继续. . .

Find error! E:
11:Missing ;
Information:
symbol: _var_2 in line: 18 col: 10
Symbol: 1

请按任意键继续. . .
Analysis end.
Find 2 error(s).
Compile end.
```

7) error2.c

A. 程序代码及说明

printf 格式错误。

```
/*测试常量声明*/

const int _const_1 = +101;

const int _const_2 = -101, _const_3 = 0, _const_4 = 5;

const char _const_c_1 = '+', _const_c_2 = '_', _const_c_3 = 'a';

/*测试变量声明*/

int _var_1;

int _var_2, _int_array[10];

char _var_4, _char_array[10];

/*测试函数定义*/

int _return_0(){return (0);}

void main()

{

    const int _const_1 = 101;

    _var_4 = 67;

    printf("should not happend!!");

    printf();
```

```

printf("43:",_var_4);

_var_2 = _return_0();

printf("0:",_var_2);

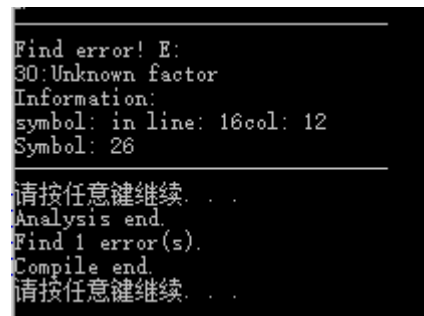
printf("0:",_return_0());

return;

}

```

B. 测试结果



```

Find error! E:
30:Unknown factor
Information:
symbol: in line: 16 col: 12
Symbol: 26

请按任意键继续...
Analysis end.
Find 1 error(s).
Compile end.
请按任意键继续...

```

8) error3.c

A. 程序代码及说明

void 函数返回值。

```

/*测试常量声明*/

const int _const_1 = +101;

const int _const_2 = -101, _const_3 = 0, _const_4 = 5;

const char _const_c_1 = '+', _const_c_2 = '_', _const_c_3 = 'a';

/*测试变量声明*/

int _var_1;

int _var_2, _int_array[10];

char _var_4, _char_array[10];

/*测试函数定义*/

void _return_0(){return 0;}

void main()

{

    const int _const_1 = 101;

```

```

    _var_4 = 67;

    printf("should not happend!!");

    printf(_const_1);

    printf("43:",_var_4);

    _var_2 = _return_0();

    printf("0:",_var_2);

    printf("0:",_return_0());

    return;

}

```

B. 测试结果

```

Find error! E:
35:void function should not return a value
Information:
symbol: in line: 9col: 25
Symbol: 25

请按任意键继续 . . .

Find error! E:
36:Should be a function with return value.
Information:
symbol: in line: 18col: 25
Symbol: 24

请按任意键继续 . . .

Find error! E:
36:Should be a function with return value.
Information:
symbol: in line: 20col: 28
Symbol: 26

请按任意键继续 . . .
Analysis end.
Find 3 error(s).
Compile end.
请按任意键继续 . . .

```

9) error4.c

A. 程序代码及说明

常量未赋值。

```

/*测试常量声明*/

const int _const_1 = +101;

const int _const_2 = -101, _const_3 = 0, _const_4 = 5;

const char _const_c_1 = '+', _const_c_2 = '_', _const_c_3 = 'a';

/*测试变量声明*/

```

```

int _var_1;

int _var_2,_int_array[10];

char _var_4,_char_array[10];

/*测试函数定义*/

int _return_0(){return (0);}


void main()
{
    const int _const_1;

    _var_4 = 67;

    printf("should not happend!!");

    printf(_const_1);

    printf("43:",_var_4);

    _var_2 = _return_0();

    printf("0:",_var_2);

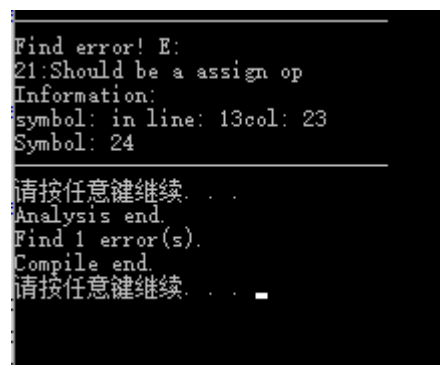
    printf("0:",_return_0());

    return;

}

```

B. 测试结果



```

Find error! E:
21:Should be a assign op
Information:
symbol: in line: 13col: 23
Symbol: 24

请按任意键继续. . .
Analysis end.
Find 1 error(s).
Compile end.
请按任意键继续. . .

```

10) error5.c

A. 程序代码及说明

printf 格式错误

```

const char _const_c_1 = '+', _const_c_2 = '_', _const_c_3 = 'a';

/*测试变量声明*/

```



```

int _var_1;

int _var_2,_int_array[10];

char _var_4,_char_array[10];

/*测试函数定义*/

int _return_0(){return (0);}


void main()

{

    const int _const_1 = 101;

    _var_4 = 67;

    printf("should not happend!!");

    printf(_const_1);

    printf("43:",_var_4);

    _var_2 = _return_0();

    printf("0:",_var_2,_var_2);

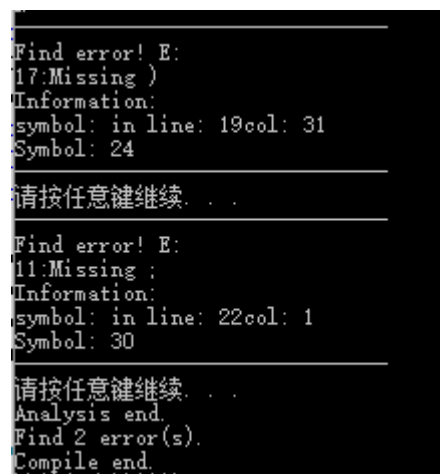
    printf("0:",_return_0());

    return

}

```

B. 测试结果



```

Find error! E:
17:Missing )
Information:
symbol: in line: 19col: 31
Symbol: 24

请按任意键继续. . .

Find error! E:
11:Missing ;
Information:
symbol: in line: 22col: 1
Symbol: 30

请按任意键继续. . .
Analysis end.
Find 2 error(s).
Compile end.

```

2. 测试结果分析

上述测试程序完整覆盖了所有的语法成分，具体来说，包括以下内容：

- 1) 函数传值与递归调用
- 2) 函数与赋值语句
- 3) 读写语句
- 4) 数组元素赋值与修改
- 5) 整形常量的定义与使用
- 6) while 循环语句
- 7) 数组元素取值
- 8) 引用全局常量、变量
- 9) 嵌套 while 循环
- 10) 函数内部调用其它函数
- 11) 字符常量的定义与使用

覆盖度很高。

五. 总结感想

在文档写到这里的时候，北航计算机学院传说中的 COCO 四大课程终于结束了。回首这两年，收获还是很大的。相比计算机组成，操作系统，面向对象其他三门课程，编译的难度不大不小，但是这学期还有数据库，机器学习，移动计算，大数据等其他课程，所有的大作业基本上都在 12 月要完成，工作量就十分巨大了。

做编译时最大的困难在于，一切都只能自己开发，从零开始。但这也可以算作一个优点，毕竟自己写的代码是最了解的。

做编译最大的收获在于，对于一个大型工程，一定要静下心来写，不能认为他太简单就掉以轻心。工程量本身也是一种难度。在语法和词法分析阶段，由于一些粗心大意，给自己后面埋下了很多坑。

其次做完编辑器的设计后，我对 C++ 有了更为深入的了解，类的使用以及 STL 库的使用都更加的熟练了。

另外还有的收获在于，我发现了理论和实践的差距是巨大的。书上的很多算法看起来都十分简单，但都仅仅是理想情况下的算法，在我们真正手动去编写程序的时候，要考虑的东西实在是太多。作为一名工程师，很多东西还是需要实践才能感受到其中的问题都在哪些地

方。理论再好，没有实践依然是纸上谈兵。

最后便是测试的重要性，很多时候程序的 bug 需要靠大量的测试才能发现。在我完成编译器编写的初期，我仅仅是针对编译的文法进行了一两个测试程序的测试。但后来我发现这是远远不够的。比如我在初期写的时候，函数调用保存寄存器时，参数寄存器\$a0 和临时寄存器\$t9 保存在了内存中的同一地方，导致\$t0 被覆盖，但是由于调用返回之后没有用过\$t9 寄存器，这个错误一直就被隐藏了，这个错误一直到我写完优化再做测试的时候才被偶然触发得以发现。这让我深深体会到了测试的重要性。

编译课设到这里就要告一段落了。从此计院再无战事，我的能力也在硝烟中得到了很大的打磨与锻炼。感谢 COCO 四大课程，感谢编译课设的存在！