

华中科技大学

# 实践课程报告

题目： 面向对象语言 Decaf 编译器实验报告

课程名称： 编译原理实践

专业班级： ACM1401

学 号： U201414620

姓 名： 胡汉鹏

指导教师： 徐丽萍

报告日期： 2017 年 1 月 15 日

计算机科学与技术学院

## 目录

1. 选题背景 .....	3
1.1 任务 .....	3
1.2 目标 .....	3
1.3 源语言定义 .....	3
2. 实验一 词法分析和语法分析 .....	4
2.1 单词文法描述 .....	4
2.2 语言文法描述 .....	6
2.3 词法分析器的设计 .....	7
2.4 语法分析器设计 .....	8
2.5 语法分析器实现结果展示 .....	9
3. 语义分析 .....	11
3.1 语义表示方法描述 .....	11
3.2 符号表结构定义 .....	12
3.3 错误类型码定义 .....	13
3.4 语义分析实现技术 .....	14
3.5 语义分析结果展示 .....	17
4. 中间代码生成 .....	19
4.1 中间代码格式定义 .....	19
4.2 中间代码生成规则定义 .....	19
4.3 中间代码生成过程 .....	20
4.4 代码优化 .....	22
4.4 中间代码生成结果展示 .....	23
5. 目标代码生成 .....	25
5.1 指令集选择 .....	25
5.2 寄存器分配算法 .....	26
5.3 目标代码生成算法 .....	27
5.4 目标代码生成结果展示 .....	27
6. 结束语 .....	29
6.1 实践课程小结 .....	29
6.2 自己的亲身体会 .....	30
参考文献 .....	31

# 1. 选题背景

## 1.1 任务

主要是通过对简单编译器的完整实现，加深课程中关键算法的理解，提高学生对系统软件编写的能力。

## 1.2 目标

本次课程实践目标是构造一个高级语言的子集的编译器，目标代码是汇编语言。按照任务书，实现的方案可以有很多种选择。

## 1.3 源语言定义

可以根据自己对编程语言的喜好选择实现。建议大家选用 C 语言的简单集合 C--语言或教材中的 Decaf 语言。

## 2. 实验一 词法分析和语法分析

### 2.1 单词文法描述

Decaf 语言的词法规则如下，词法分析程序写在 lab2.1 文件中。

下面的是关键字，他们都是保留字：bool break class else extends for if int new null return string this void while static Print ReadInteger ReadLine instanceof

一个标识符是以字母开头的字母、数字和下划线的序列。Decaf/Mind 是大小写敏感的，例如 if 是一个关键字，但是 IF 却是一个标识符，binky 和 Binky 是两个不同的标识符。

空白字符（即空格、制表符和换行符）仅用于分隔单词。关键字和标识符必须被空白字符 或者一个既不是关键字也不是标识符的单词隔开。ifintthis 是单个标识符而不是三个关键字，但 if(23this 被识别成四个单词。

布尔常量是 true 或者 false，如同关键字一样，它们也是保留字。

一个整型常量既可以是十进制整数也可以是十六进制整数。一个十进制整数是一个十进制 数字（0-9）的序列；十六进制整数必须以 0X 或者 0x 开头（是零，而不是字母 O），后面跟着一个十六进制数字的序列。十六进制数字包括了十进制数字和从 a 到 f 的六个字母（大小写 均可）。合法的整数的例子有：8, 012, 0x0, 0X12aE。

一个字符串常量是被一对双引号包围的可打印 ASCII 字符序列。字符串常量中不可以包含 换行符，也不可以分成若干行，例如： "this is not a valid string constant" 字符串常量中支持以下几种转义序列：\"表示双引号，\\表示单个反斜杠，\t 表示制表 符，\n 表示换行符。其它情况下不认为反斜杠（\）是转义符。例如：\"t\"是一个长度为 1 的 字符串，其中的t 转义为制表符；而\"u\"是一个长度为 2 的字符串，它包含反斜杠和字母 u 两个字符。

这个语言中的操作符和分隔字符包括：

+ - \* / % < <= > >= == != && || ! ; , . [ ] ( ) { }

单行注释是以//开头直到该行的结尾。Decaf/Mind 中没有多行注释。如果单行注释出现 在程序末尾，那么单行注释的结尾需要换行。

对应单词元素和生成的 Token 关系见表格 2-1。

表格 2-1 词法及对应 token 关系表

正规式部分	生成的 token	正规式部分	生成的 token
int	INT	[a-zA-Z_][0-9A-Za-z_]*	ID
float	FLOAT	;	SEMI
bool	BOOL	,	COMMA
string	STRING	=	ASSIGNOP
void	VOID	<	LESS
class	CLASS	>	MORE
new	NEW	<=	LESSEQUAL
static	STATIC	>=	MOREEQUAL
extends	EXTENDS	==	EQUAL
for	FOR	!=	NOTEQUAL
return	RETURN	\+	PLUS
if	IF	-	MINUS
else	ELSE	\*	STAR
while	WHILE	\/	DIV
break	BREAK	&&	AND
print	PRINT	\	OR
this	THIS	\.	DOT
ReadInteger	READINTEGER	!	NOT
ReadLine	READLINE	%	MOD
instanceof	INSTANCEOF	\(	LP
\)	RP	\[	LB
\]	RB	\{	LC
\}	RC		

以上所有 token 生成时，为了构造语法树以及后续实验，除了返回 token 供语法分析使用，都会有以下几个动作。一是构造叶子节点，二是维持 yyrol 变量，token 所在的列号。

表格 2-1 反映了词法分析中分析到的其他元素，这些元素不返回 token，但也有语义动作，例如对 SPACE，yyrol 需要加 strlen(yytext)；对 TAB，yyrol 需要加 4；对 RET 和 EOF 都对 yyrol 置为 1。

表格 2-2 不返回 token 的词法元素表

正规式	识别符号
[ ]*	SPACE
\t	TAB
\r	RET
\n	EOF

## 2.2 语言文法描述

Decaf 语法采用的是课程教材后面的语法，有部分小细节有所改动，但大体思想依旧相同。文法规则如下。

Program : ClassDef ProgramList

ProgramList : ClassDef |  $\xi$

VariableDef : Variable SEMI

Variable : Type ID

Type : INT | BOOL | FLOAT | STRING | VOID | CLASS ID | Type LB RB

Formals : FormalsList |  $\xi$

FormalsList : Variable COMMA FormalsList | Variable

FunctionDef : Type ID LP Formals RP StmtBlock

| STATIC Type ID LP Formals RP StmtBlock

ClassDef : CLASS ID LC Field RC

| CLASS ID EXTENDS ID LC Field RC

Field : VariableDef Field | FunctionDef Field |  $\xi$

StmtBlock : LC StmtList RC

StmtList :  $\xi$  | Stmt StmtList

Stmt : Variable SEMI | SimpleStmt SEMI | IfStmt | WhileStmt | ForStmt

| BreakStmt SEMI | ReturnStmt SEMI | PrintStmt SEMI | StmtBlock

SimpleStmt : LValue ASSIGNOP Expr | Call |  $\xi$

LValue : ID | Expr DOT ID | Expr LB Expr RB

Call : ID LP Actuals RP | Expr DOT ID LP Actuals RP

Actuals : ActualsList |  $\xi$

ActualsList : Expr COMMA ActualsList | Expr

ForStmt: FOR LP SimpleStmt SEMI BoolExpr SEMI SimpleStmt RP Stmt

WhileStmt : WHILE LP BoolExpr RP Stmt

```

IfStmt : IF LP BoolExpr RP Stmt %prec LOWER_THAN_ELSE
      | IF LP BoolExpr RP Stmt ELSE Stmt
ReturnStmt : RETURN | RETURN Expr
BreakStmt : BREAK
PrintStmt : PRINT LP ActualsList RP
BoolExpr : Expr
Expr : Constant | LValue | THIS | Call | LP Expr RP | Expr PLUS Expr
     | Expr MINUS Expr | Expr STAR Expr | Expr DIV Expr | Expr MOD Expr
     | MINUS Expr | Expr LESS Expr | Expr MORE Expr
     | Expr LESSEQUAL Expr | Expr MOREEQUAL Expr | Expr EQUAL Expr
     | Expr NOTEQUAL Expr | Expr AND Expr | Expr OR Expr | NOT Expr
     | READINTEGER LP RP | READLINE LP RP | NEW ID LP RP
     | NEW Type LB Expr RB | INSTANCEOF LP Expr COMMA ID RP
     | LP CLASS ID RP Expr
Constant : INTC | BOOLC | FLOATC | STRINGC

```

## 2.3 词法分析器的设计

词法分析器的主要任务是将输入文件中的字符流组织成词法单元流即 `token` 序列，在某些字符不符合程序设计语言词法规范时要有能力报告相应的错误。本实验中采用的方法是利用工具 `GNU Flex` 生成，根据工具要求的输入所编写的词法规范正则式，其理论基础是正则表达式和有限状态自动机。

所使用的 `decaf` 词法规范正则表达式见单词文法描述。在 `labtest.1` 文件中有实验 1 最初版本的文件，即没有附加语法分析等后续实验步骤的内容，而是在每次识别词法元素之后就打印对应 `token`，用于测试词法分析是否正确。编写完词法分析程序之后编译它，生成 `lex.yy.c` 文件，其中 `yylex()` 需要在后续任务中使用。

下图是对词法分析程序的测试，直接调用生成的各执行文件，输入字符，回车后会输出对应的 `token`。

```

u201414620@ubuntu:~/compile lab/lab1$ ./labtest
1
INT 1
huhanpeng
ID huhanpeng
bool
ID bool
while
ID while
if
IF if
;
SEMI ;
^Z
[1]+  Stopped                  ./labtest

```

图表 2-1 词法分析测试

最后版本的词法分析程序在文件 lab2.1 文件中。

## 2.4 语法分析器设计

语法分析程序的主要任务是读入词法单元流，判断输入程序是否匹配程序设计语言的语法规则，并在规范匹配的情况下构建起输入程序的静态结构。本实验中采用的方法是利用工具 **Bison**，其前身是 **yacc**，其生成的语法分析程序采用了自底向上的 LALR (1) 分析技术。

语法规则以及相应生成语法树的语义动作写在 `syntax.y` 文件中，主程序写在 `main.c` 主文件中。语法分析中实现生成语法树需要用到属性文法，下面是此次实验用到的关键的结构体的部分内容。

```

struct ast{
    int line;           //记录节点的行号
    int col;           //记录节点的列号
    char *name;        //节点名字
    int nt;            //是否为非终结符
    struct ast *l;      //指向左子树
    struct ast *r;      //指向右子树
    union{
        char *idtype; //当 name 为 ID 时，记录 ID 的具体名称
        int inter;    //当 name 为 INTC 时，记录实际整数值
        float flo;    //当 name 为 float 时，记录实际实数的值
    };
};

```

用到的主要函数为 `struct ast *newast(char *name, int num, ...)`，每次按照产生式规约，都要根据产生式右边的结点生成产生式左边的结点。其实现方法主要是

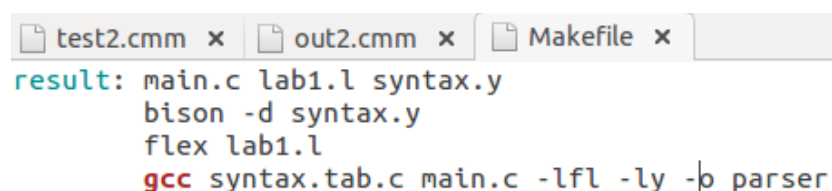


将产生式右边第一位设为左边节点的左孩子节点，若产生式右边还有多个节点，则每个节点作为前一节点的右孩子。

编写完语法分析程序之后编译它，会生成一个 C 语言文件，其中 `yyparse()` 和词法分析程序产生的 `yylex()` 需要联合使用，在 `main` 函数中编写显示调用，即可实现对文件的词法语法分析。

## 2.5 语法分析器实现结果展示

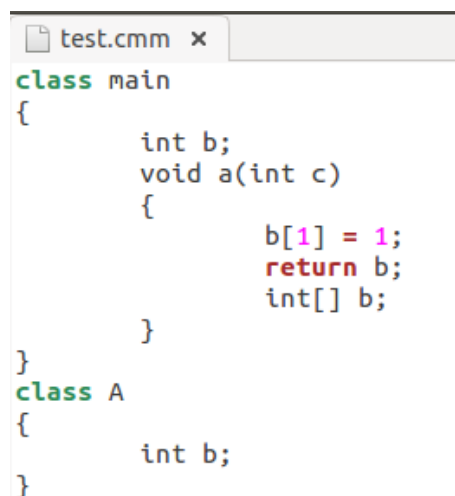
Makefile 文件中需要的命令见图表 2-2。



```
test2.cmm x out2.cmm x Makefile x
result: main.c lab1.l syntax.y
        bison -d syntax.y
        flex lab1.l
        gcc syntax.tab.c main.c -lfl -ly -o parser
```

图表 2-2 语法分析运行命令

生成可执行文件后用命令 `./parser test.cmm` 对文件 `test.cmm` 中的程序进行语法分析。文件 `test.cmm` 中内容如图表 2-3



```
test.cmm x
class main
{
    int b;
    void a(int c)
    {
        b[1] = 1;
        return b;
        int[] b;
    }
}
class A
{
    int b;
}
```

图表 2-3

图表 2-3 测试文件内容

运行结果即生成的语法树如图表 2-4，为每一个终结符打印对应的行号和列号。

```

u201414620@ubuntu:~/comptle Lab/lab2$ ./parser test.cmm
Program-ClassDef-CLASS(1:1)
  -ID:main(1:7)
  -LC(2:1)
  -Field-VariableDef-Variable-Type-INT(3:5)
    -ID:b(3:9)
    -SEMI(3:10)
  -Field-FunctionDef-Type-VOID(4:5)
    -ID:a(4:10)
    -LP(4:11)
    -Formals-FormalsList-Variable-Type-INT(4:12)
      -ID:c(4:16)
    -RP(4:17)
    -StmtBlock-LC(5:5)
      -StmtList-Stmt-SimpleStmt-LValue-Expr-LValue-ID:b(6:9)
        -LB(6:10)
        -Expr-Constant-INTC:1(6:11)
        -RB(6:12)
        -ASSIGNOP(6:14)
        -Expr-Constant-INTC:1(6:16)
      -SEMI(6:17)
      -StmtList-Stmt-ReturnStmt-RETURN(7:9)
        -Expr-LValue-ID:b(7:16)
      -SEMI(7:17)
      -StmtList-Stmt-Variable-Type-Type-INT(8:9)
        -LB(8:12)
        -RB(8:13)
        -ID:b(8:15)
        -SEMI(8:16)
      -RC(9:5)
    -RC(10:1)
  -ProgramList-ClassDef-CLASS(11:1)
    -ID:A(11:7)
    -LC(12:1)
    -Field-VariableDef-Variable-Type-INT(13:5)
      -ID:b(13:9)
      -SEMI(13:10)
    -RC(14:1)

```

图表 2-4 语法分析结果语法树图

## 3. 语义分析

### 3.1 语义表示方法描述

对实验 1 即语法分析中的数据结构 `struct ast` 进行扩充，增加以下几个域：

```
Var *var_h, *var_t;  
Fun *fun_h, *fun_t;  
Class *class_h, *class_t;
```

增加以上几个域的目的是方便根据语法树生成符号表，每次定义一个变量则为这个变量定义节点新建变量结构；对于函数定义节点会记录其包含的所有变量，并新建函数结构；对于类 `Class` 定义节点，该结构体记录 `Class` 内定义的非函数内变量结构以及函数结构，并新建 `Class` 型结构。当自底向上生成语法树的时候，根节点将所有信息连接到类型为 `Global *` 的 `glo` 指针，形成整个符号表结构。

其中记录变量信息的结构体信息如下：

```
typedef struct VAR  
{  
    char name[50];           //变量名称  
    struct typelist *type;    //变量类型  
    int line;                //变量所在行号  
    int rol;                 //变量所在列号  
    int place;               //待扩充至中间代码生成使用  
    struct VAR *next;  
}Var;
```

记录函数信息的结构体信息如下：

```
typedef struct FUNCTION  
{  
    char name[50];           //函数名称  
    struct typelist *returnType; //函数返回类型  
    int line;                //函数定义所在行号  
    int rol;                 //函数定义所在列号  
    Var *form_l;  
    //函数的参数列表
```

```

    Var *var_1;                                //该函数体内定义的局部变量序列
    struct FUNCTION *next;
}Fun;
记录结构体信息的结构体信息如下：
typedef struct CLASS
{
    char name[50];                            //结构体的名称
    int line;                                //结构体定义所在行号
    int rol;                                //结构体定义所在列号
    int size;                                //结构体占用的空间，用于中间代码生成
和目标代码生成
    Fun *fun_1;                                //结构体内定义的函数序列
    Var *var_1;                                //结构体内定义的域，不包括函数内定义
的变量
    struct CLASS *next;
}Class;
全局类型的结构体的定义如下：
typedef struct GLOBAL
{
    Var *var_1;                                //记录全局变量
    Fun *fun_1;                                //记录全局函数
    Class *class_1;                            //记录所有 class
}Global;

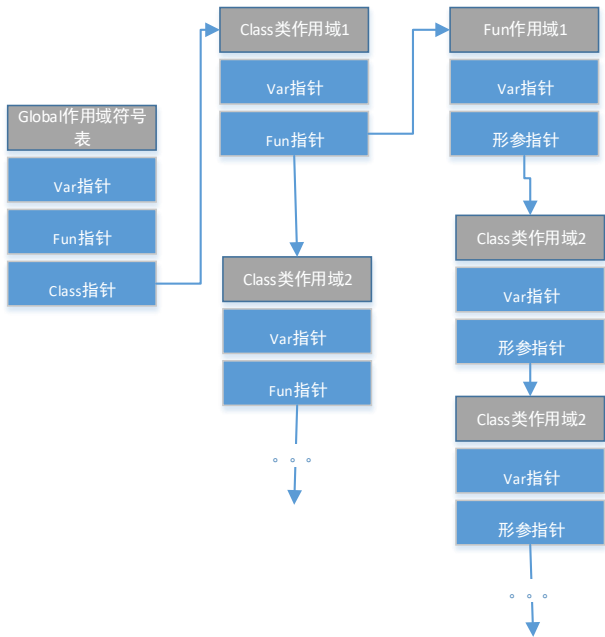
```

## 3.2 符号表结构定义

语义分析的第一个任务是建立符号表。本实验借鉴课本后面多个符号表的结构，采用了 Global、Class、Fun 三重符号表，分别代表全局作用域、class 类作用域、函数作用域，作用域大小依次减小。小作用域可以通过大的作用域访问到，如在 Class 的作用域中可以通过 fun\_1 指针访问在该类下所有函数的作用域内信息。

对于书上提到的形参作用域和 Local 作用域，此处都体现在函数的形参列表和局部变量列表中。用于两者其实都是变量列表，故所用的结构体都是 Var 类型的，统一认为是函数作用域里的两个部分。

本质上该符号表也是一个十字交叉链表，只是各个级别的结构类型不同。基于这个性质，可以在语义分析时从 `glo` 指针开始对链表检索，通过前序或后序等方法从根节点分析语法树可以同时实现自底向上和自顶向下方法。



图表 3-1 符号表组织示意图

### 3.3 错误类型码定义

语义分析的第二个任务是静态语义检查。对于违反语义规则的语句要实现报错，本实验用链表存储语义错误，因此要定义错误类型结构体。该结构体定义如下：

```
typedef struct ERROR
{
    int row;           //错误发生处的行号
    int col;          //错误发生的列号
    char *s;           //错误内容
    struct ERROR *next;
}Error;
```

用 `Error *error_h`, `*error_t` 两个指针管理错误链表。本实验直接在语义分析的时候将具体错误写到结构体里，即用一个字符串 `s` 存储错误的具体类型，打印错误时直接讲 `s` 打印即可。

下表示本实验检查的错误以及产生的错误信息。由于本实验用的是 `decaf`，与实验书有出入，此处错误类型借鉴了实验书给的 23 个示例程序和 17 个错误类

型以及语法分析部分，设计错误类型。

表格 3-1 报错类型和报错信息

错误类型	错误信息
双操作数运算符两操作数类型不匹配	Expression type mismatch!
布尔运算操作数不是 bool 类型	Expression type error!Need bool!
双操作数两操作数类型不是可计算类型（如 string，void 等）	Expression type error!
函数定义时返回类型错误	Function definition error, return type does not match!
函数调用时参数错误	Function formals do not match!
布尔表达式的类型不是 bool 类型	Type error, type Bool is wanted!
左值引用位定义的数组	Undefined array!
数组下标类型不是一个整型	The type index of array is not int!
Class 重复定义	Class redefinition!
Extend 继承未定义的 class	Illegal inheritance!
函数重复定义	Function redefinition!
语句 break 不在一个循环内部	Break is not in a loop!
引用未定义 id	Undefined id!
引用未定义 class	Undefined class id!
引用类 class 里面未定义的变量 id	Undefined id in class!
引用类 class 里面未定义的函数	Undefined function id in class!
变量重复定义	Variable redefinon !
取负运算操作数错误	Invalid operands domain to MINUS operation!
取非运算操作数类型错误	Invalid operands domain to NOT operation!Need type Bool!

### 3.4 语义分析实现技术

本实验语义分析代码写在 syntax.c 文件，相关函数的声明在 syntax.h 文件中。每个节点存储的类型是一个 typelist 结构体指针，该结构体定义如下。

```
struct typelist
```

```

{
    int type;                //具体类型
    char *idtype;            //当类型为 class 或数组时，存储 class 名或
数组名
    struct typelist *next;    //某些非终结符类型可能为多个类型构成的
链表，用于传递类型信息
};

```

类型 int    type 对应的实际类型关系见下表。

表格 3-2 类型号和类型对应表

Type	0	1	2	3	4	5	6
类型	INT	BOOL	FLOAT	STING	VOID	CLASS	ARRAY

本实验语义分析总共用了 3 次遍历，具体每次遍历的任务和实现如下。

## 1) 第一次遍历

第一次遍历在语法分析生成语法树阶段，主要任务是初步建立符号表。采用属性文法的方法，其核心思想是为上下文无关文法中的每一个终结符或非终结符赋予一个或多个属性值。由于是自底向上的 LALR 分析，所以在语法生成规则后面增加语义动作来实现属性赋值操作。这里用到的主要函数有

```

void addGlo(Global *glo, struct ast* a, struct ast* b);
//建立全区符号表 glo

void addClass(struct ast* dest, struct ast* name, struct ast* child);
//生成一个 Class 域，并将该作用域给该 class 类型定义的语法节点

void passClass(struct ast* dest, int num, ...);
//将多个节点的 class 作用域相连接并传递给 dest 节点

void addFun(struct ast* dest, struct ast* name, struct ast* reType, struct ast*
formal, struct ast* stmt, int is_static, Global *glo);
//生成一个 Fun 域，并将该作用域赋给该函数定义语法节点

void passFun(struct ast* dest, int num, ...);
//将多个节点的 Fun 作用域相连接并传递给 dest 节点

void addVar(struct ast* dest, struct ast* name, struct ast* type);
//新生成一个 Var，并赋给该类型定义语法节点

```

```
void passVar(struct ast* dest, int num, ...);
```

//将多个 Var 链接并赋给 dest 节点

```
void addType(struct ast* a, int type, struct ast* b);
```

//将 1 个或多个节点的类型链接，并赋值给 a 作为 a 的类型

此次遍历为初步建立符号表，因为有些符号信息在本次遍历没有确定下来。如左值引用 Lvalue 的类型确定在自底向上过程中实现较为复杂，因此此处先为该语法节点建立一个空的 typelist，待第二次遍历时确定实际类型。

## 2) 第二次遍历

第二次遍历的主要任务为简单引用检查以及为确定所有左值的类型。检查的主要错误为引用未定义的 class、函数或变量，重复定义 class、函数或变量。此处用到三个查找函数

```
Class *findIdClass(char *classname, Global *glo, int line, int rol);
```

```
Var *findIdVar(char *varname, Global *glo, Class *pclass, Fun *pfun, int line, int rol);
```

```
Fun *findIdFun(char *funname, Global *glo, Class *pclass, int line, int rol);
```

分别能够查找引用之前是否有相应的 class、变量或函数定义。若查找失败返回 NULL。依此来确定是否未定义或者重复定义。

左值的类型确定也是通过查找函数实现，通过查找函数找到具体引用节点，并将类型拷贝给该左值节点。此处需要注意的是不能直接复值而需要拷贝，因为两个节点拥有两个独立的空间，左值节点该空间可能也被其他节点（某些非终结符节点）引用，如果直接用 typelist 结构指针赋值会出现作用域上的指向错误。

第二次遍历的主函数为：

```
void map1(struct ast *tree, Error **perror_h, Error **perror_t, Global *glo, Class **pclass, Fun **pfun);
```

## 3) 第三次遍历

第三次遍历检查所有的类型匹配错误，有与所有的类型都已经计算完成，通过遍历语法树并同时搜索相应的符号表，可以很方便实现所有的类型检查。其用到的主要函数为：

```
void checkType(Error **perror_h, Error **perror_t, int calType, int row, int rol, int num, ...);
```



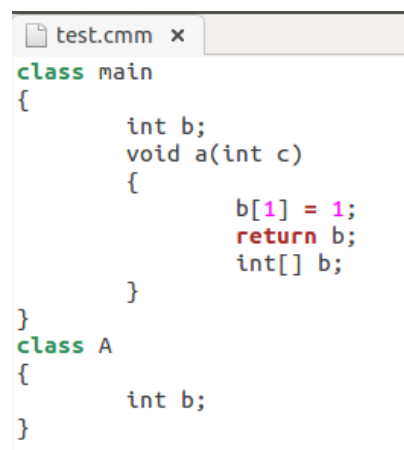
```

//检查表达式类型错误
int typeMatch(struct typelist *a, struct typelist *b);
//比较两个类型 typelist 结构是否相等
int formalTypeMatch(struct typelist *a, Var *form);
//函数调用时参数匹配错误
void map2(struct ast *tree, Error **perror_h, Error **perror_t, Global *glo, Class
**pclass, Fun **pfun);
//第三次遍历主函数
void checkReturnType(struct typelist *returnType, struct ast *a , Error
**perror_h, Error **perror_t);
//检查函数定义返回值是否匹配

```

### 3.5 语义分析结果展示

测试文件如图所示，此处有两个错误，函数 main.a 返回类型错误和引用未定义的数组错误。



```

test.cmm x
class main
{
    int b;
    void a(int c)
    {
        b[1] = 1;
        return b;
        int[] b;
    }
}
class A
{
    int b;
}

```

图表 3-2 语义分析测试文件

Makefile 文件命令如下

```
result: main.c lab2.l syntax.y
```

```
bison -d syntax.y
```

```
flex lab2.l
```

```
gcc syntax.tab.c main.c -lfl -ly -o parser
```

生成可执行文件后运行 `./parser test.cmm`，运行结果如图表 3-3。符号表按作用域打印，最后打印出错信息。

```
-----  
-- Symbol table --  
-----  
Global  
Class:main line:1 rol:7  
Class:A line:11 rol:7  
-----  
Class:main  
Function:a line:4 rol:5  
Variable:b line:3 rol:5 type:INT  
  
Class:A  
Variable:b line:13 rol:5 type:INT  
-----  
Class:main Function:a ReturnType:VOID  
Formal:c line:4 rol:12  
Variable:b line:8 rol:9 type:ARRAY  
-----  
print Symbol table ok!  
-----  
Error:7:9: Function definition error, return type does not match!  
Error:6:9: Undefined array!
```

图表 3-3 语义分析运行结果

## 4. 中间代码生成

### 4.1 中间代码格式定义

中间代码是编译器里最核心的数据结构之一，中间代码是编译器根据输入程序所构造出来的绝大多数数据结构，比如词法流、语法树、带属性的语法树等，其目的是为了更方便编写编译器程序的各种操作。中间代码根据表示方式可大致分为线性、图形和混合型，本次实验中采用的是线性中层次中间代码——三地址码即四元式。其格式形式为

Op      place[0]      place[1]      place[2]

其结构体定义为 CodeLine，结构体定义如下：

```
typedef struct CodeLine {  
    char op[10];  
    char place[3][20];  
    int kind; //扩充功能，用于识别四元式的类型  
    struct CodeLine *next;  
    struct CodeLine *pre;  
} CodeLine;
```

### 4.2 中间代码生成规则定义

每条语句生成的关系表如,空代表为 null。表中每一个 X1 可以为临时标号  $T_i$ ，也可以为常数（如#0），也可以是取地址表达式  $*T_i$ 。

表格 4-1 中间代码对应关系表

op	place[0]	place[1]	place[2]	对应语义
Lable	x			标号 x
FUNCTION	f	:		定义函数 f
ARG	T			传实参
CALL	函数名			函数调用
+	X1	X2	X3	$X3 = X1 + X2$
-	X1	X2	X3	$X3 = X1 - X2$
*	X1	X2	X3	$X3 = X1 * X2$
/	X1	X2	X3	$X3 = X1 / X2$

%	X1	X2	X3	X3 = X1 % X2
<	X1	X2	X3	X3 = X1 < X2
>	X1	X2	X3	X3 = X1 > X2
<=	X1	X2	X3	X3 = X1 <= X2
>=	X1	X2	X3	X3 = X1 >= X2
==	X1	X2	X3	X3 = X1 == X2
&&	X1	X2	X3	X3 = X1 && X2
	X1	X2	X3	X3 = X1    X2
!=	X1	X2	X3	X3 = X1 != X2
~	X1		X3	X3 = !X1
-	#0	X2	X3	X3 = -X2
=	X1		X3	X3 = X1
=	VTBL	<classname>	X3	X3= VTBL<classname>
return	X1			return X1
return				return
PARAM	X1			函数形参 X1
if	X1	>/</>= /<= / == /!= /&& /	X2	if X1 >/</>= /<= / == /!= /&& /   X2
Goto	L1			跳转到标号 L1

### 4.3 中间代码生成过程

中间代码的相关过程在文件 `genCode.c` 和 `genCode.h` 两个文件中。程序中运用结构 `ThreeAddCode` 封装四元式结构体，该结构体有两个 `CodeLine` 指针，记录该段代码包括的四元式序列，若没有包含四元式序列，初始化为 `NULL`。

```
typedef struct ThreeAddCode {
    struct CodeLine *head, *tail;
} ThreeAddCode;
```

中间代码生成过程用到的基本函数为，本实验对标号采用的基本策略是区分变量标号、中间变量标号和跳转标号。且每次可能使用标号前都会申请一个新的标号，不管后面会不会使用这个标号，且数量不限制上限。

```
ThreeAddCode *CreateTAC(char *newOp, char *place1, char *place2, char
*place3, int kind)
```

```

//用于生成三地址码，返回的是已经封装的 ThreeAddCode 指针
ThreeAddCode *LinkTAC(ThreeAddCode *code1, ThreeAddCode *code2)
//连接两段三地址码，返回最后连接好的三地址码序列
void CreateVarP(char *place);
//申请新的变量标号
void CreatePlace(char *place);
//申请新的临时标号
void CreateLable(char *l);
//申请新的跳转标号

```

利用上述几个函数，根据中间代码的生成规则，针对每一条会产生中间代码的语句进行单独分析输出。定义 `ThreeAddCode *code` 总的三地址指针，最后整个程序的三地址码由 `code` 指针管理。相关函数定义如下

```

void gen(struct ast *tree);           //中间代码生成的主调函数
ThreeAddCode *translate_IfStmt(struct ast *a, char *l_out);
ThreeAddCode *translate_WhileStmt(struct ast *a, char *l_out);
ThreeAddCode *translate_ReturnStmt(struct ast *a);
ThreeAddCode *translate_PrintStmt(struct ast *a);
ThreeAddCode *translate StmtBlock(struct ast *a, char *l_out);
ThreeAddCode *translate Stmt(struct ast *a, char *l_out);
ThreeAddCode *translate_SimpleStmt(struct ast* a);
ThreeAddCode *translate_ForStmt(struct ast* a, char *l_out);
ThreeAddCode *translate_Exp(struct ast *a, char *t_in);
ThreeAddCode *translate_Lvalue(struct ast* a, char *t_in);
ThreeAddCode *translate_Parm(struct ast* a, ParmList **pph);
ThreeAddCode *translate_Call(struct ast* a, char *t_in);
ThreeAddCode *translate_ReadInteger(struct ast *a, char *t_in);

```

从语法树根节点开始按顺序依次翻译每一条语句，对某些语句会有特殊操作。如针对 `if` 语句（`while` 语句和 `for` 循环语句同样转化成 `if` 语句来做），其后紧跟一条跳转语句，输出时对其做特殊处理，即在同一行输出。

针对循环过程和分支语句跳转标号的确定，由于检索语法树可以方便实现自顶向下和自底向上，当翻译到相关非终结符（如 `IFStmt`）的时候就可以直接指出跳转的标号，只需要调整生成标号的语句和递归调用翻译函数语句的相对顺序即可。

针对 `break` 语句，翻译函数调用的时候用一个 `char *l_out` 记录上一层的跳转

标号，若 `l_out` 为 `null` 说明 `break` 不在循环中，报错。

针对数组和类 `class` 定义语句，实验课本用的是 `DEC` 语句实现空间分配，此处用 `CALL Alloc` 代替，相关 `Alloc` 函数默认为能直接调用的库函数在目标代码中给出。

针对输入输出函数，此处翻译结果为 `Call ReadInteger` 和 `Call Print`，同样默认为可以直接调用函数，在目标代码中给出。

## 4.4 代码优化

在中间代码生成过程中以及生成后对中间代码做了部分简单的优化

### 1) 常数置换

对于左值引用时常数的情况，在 `translateLvalue` 中生成 `t_in = #x` 格式的中间代码，但 `t_in` 是个临时标号，马上会赋值给其他标号，故直接改写 `t_in` 成 `#x`，而不生成多余中间代码。

### 2) 跳转语句优化

由于 `if` 分支语句和循环语句都会转化为 `if` 语句处理，程序中会有非常多的 `if` 跳转，故这部分优化特别重要。起初的实现方法是 `if` 语句后紧跟条件为真时的跳转语句，下一行为条件为假时的跳转语句。可以简化为讲条件置为和原来相反的条件，后紧跟此刻为真的跳转语句，该语句相当于未改前的条件为假时的跳转语句，下一句直接跟条件为真时的运行程序。这样优化后的代码和改动之前是等价的，可以少一条 `Goto` 语句。

### 3) 连续赋值优化

由于程序实现的特性，代码中会有类似下述形式的代码

$$T_x = \langle \text{表达式} \rangle$$
$$X = T_x$$

即一个临时标号被赋值后未经任何改动就被赋值给其他标号，那这个标号可以省略，这里考虑到一个临时标号不会被重复使用，故可以直接归并，归并后的形式为

$$X = \langle \text{表达式} \rangle$$

每次归并可以减少一条语句。

## 4.4 中间代码生成结果展示

Makefile 文件如图所示

```
Makefile x
result: main.c lab2.l syntax.y
      bison -d syntax.y
      flex lab2.l
      gcc syntax.tab.c syntax.c genCode.c main.c -lfl -ly -o parser
```

图表 4-1 实验 3 Makefile 文件

生成可执行文件后，直接调用 `./parser test.cmm` 执行程序，翻译 `test.cmm` 中的程序，其中间代码打印结果如下图所示。

```
class main
{
    int b;
    int a(int c, int d, int e)
    {
        int b;
        for(b = 0; b < 1; b = b+1)
        {
            d = d + e;
        }
        return 0;
    }
    int yin()
    {
        int xx;
        xx = 1;
        xx = a(xx, xx, xx);
        while(2 > 1)
        {
            xx = xx - 1;
        }
        return 1;
    }
}
class A
{
    int fun()
    {
        class main cm;
        cm = new main();
        cm.b = 0;
        int a;
        a = fun();
        if(a > 0)
        {
            a = 1;
        }
        return 0;
    }
}

FUNCTION _main_New :
_main_New :
_T2 = #5
ARG _T2
_T3 = CALL Alloc
_T4 = _T3 + #4
*_T4 = #0
_T5 = VTBL <main>
*_T3 = _T5
return _T3

FUNCTION _main_a :
PARAM _T7 : #4
PARAM _T8 : #8
PARAM _T9 : #12
PARAM _T10 : #16
_T13 = #0
_T11 = _T13
Lable _L1 :
_T15 = #1
if _T11 < _T15 Goto _L2
Goto _L0
Lable _L2 :
_T18 = _T9 + _T10
_T9 = _T18
_T26 = #1
_T24 = _T11 + _T26
_T11 = _T24
Goto _L1
Lable _L0 :
_T28 = #0
return _T28

FUNCTION _main_yin :
PARAM _T29 : #4
_T32 = #1
_T30 = _T32
ARG _T30
ARG _T30
ARG _T30
_T35 = CALL _main_a
_T34 = _T35
_T30 = _T34
Lable _L4 :
_T42 = #2
_T43 = #1
if _T42 > _T43 Goto _L5
Goto _L3
Lable _L5 :
_T47 = #1
_T45 = _T30 - _T47
_T30 = _T45
Goto _L4
Lable _L3 :
_T49 = #1
return _T49

FUNCTION _A_New :
_A_New :
_T52 = #4
ARG _T52
_T53 = CALL Alloc
_T54 = VTBL <A>
*_T53 = _T54
return _T53
```

图表 4-2 实验 3 测试文件和生成的中间代码

```
FUNCTION _A_fun :
PARAM _T55 : #4
CALL _main_New
_T56 = _T58
_T59 = _T56 + #4
_T62 = #0
_T59 = _T62
_T66 = CALL _A_fun
_T65 = _T66
_T63 = _T65
_T68 = #0
if _T63 > _T68 Goto _L6
Goto _L7
Lable _L6 :
_T71 = #1
_T63 = _T71
Lable _L7 :
_T72 = #0
return _T72
```

图表 4-3 生成的中间代码





## 5. 目标代码生成

### 5.1 指令集选择

指令集选择可以看成是一个模式匹配问题，本次实验采用的是 MIPS 指令集作为目标体系结构，原因是它属于 RISC 范畴，与 x86 等体系结构相比形式简单，便于处理。MIPS 体系结构共有 32 个寄存器，在汇编代码中可以使用 \$0-\$31 来表示。

本次实验中间代码中采用了线性结构的三地址码，那么其指令选择方式是逐条翻译中间代码成为目标代码，目标代码的中间代码和 MIPS 指令集的对应关系见下表。

表格 5-1 中间代码和 MIPS 指令对应关系表

中间代码	Mips32 指令
LABEL x:	x:
x:= #k	li reg(x),k
x:= y	move reg(x),reg(y)
x:= y+#k	addi reg(x),reg(y),k
x:= y+z	add reg(x),reg(y),reg(z)
x:= y-#k	addi reg(x),reg(y),-k
x:= y-z	sub reg(x),reg(y),reg(z)
x:= y*z	mul reg(x),reg(y),reg(z)
x:= y/z	div reg(y),reg(z) mflo reg(x)
x:= *y	lw reg(x),0(reg(y))
*x = y	sw reg(y),0(reg(x))
GOTO x	j x
x:= CALL f	jal f move reg(x),\$v0
RETURN x	move \$v0,reg(x)
IF x == y GOTO z	beq reg(x),reg(y),z
IF x != y GOTO z	bne reg(x),reg(y),z
IF x > y GOTO z	bgt reg(x),reg(y),z
IF x < y GOTO z	blt reg(x),reg(y),z

IF $x \geq y$ GOTO $z$	bge reg(x),reg(y),z
IF $x \leq y$ GOTO $z$	ble reg(x),reg(y),z

## 5.2 寄存器分配算法

由于 RISC 机器的特点是除了 load/store 型指令，其他所有操作数必须来自于寄存器而不是内存，除了结构体和数组必须存放在内存中，其余任何参与运算的变量都要放入寄存器才行。而寄存器数量却不是无限制的，所以需要好的寄存器分配算法。

本实验寄存器的管理基本思路是使用最久未使用算法淘汰暂时不使用的标号。本实验使用结构体 RegDes 管理 32 个寄存器，使用结构体 VarDes 管理变量标号（包括临时标号），具体定义如下。

```
typedef struct VarDes_t{
    char op[5];                //标号名称
    int regNo;                 //对应寄存器号，若未分配初始为-1
    int offset;                //相对于标号所在函数体的栈底的偏移
    struct VarDes_t* next;
} VarDes;

typedef struct RegDes_t{
    char name[3];              //寄存器名称
    VarDes* var;               //存储对应的标号结构体
    int old;                    //用于寄存器记录未使用时间，便于制定
```

淘汰算法

```
} RegDes;
```

分配寄存器的主要函数为：

```
int allocateRegForOp(char* opin, FILE* fp)
```

```
int getReg(FILE* fp)
```

allocateRegForOP 函数返回标号 opin 对应的寄存器，若标号已经在标号列表 varlist 中则查看其对应的寄存器号，若有对应的寄存器号，则直接返回该寄存器号，否则说明虽然该标号已经出现过但之前占用的寄存器已经被其他标号占有，而标号已经写入栈中，此时用 getReg 函数获取一个寄存器分配给该标号，根据标号的相对偏移重新读出该标号写入寄存器中。

若标号未出现过，则将该标号加入 varlist，并用 getReg 函数获取一个寄存器

分配给该标号。

分配寄存器 `getReg` 函数首先查找有无空闲的寄存器，若有则直接返回，否则说明所有寄存器都已经被占用，此时选择寄存器中 `old` 最大的寄存器淘汰，即最久未使用算法。未使用时间 `old` 需要在运行时维护，每次寄存器分配该寄存器 `old` 置为 0，每次 `getReg` 查询寄存器组是没有被释放的寄存器的 `old` 都会加 1。淘汰寄存器中根据标号的偏移写入栈中相应的地址。

## 5.3 目标代码生成算法

针对实验 3 生成的线性结构的中间代码，按照上述中间代码和 MIPS 指令对应关系表，逐条翻译生成目标代码，翻译成的目标代码直接写入目标生成文件中。

实验过程中用以 `offset` 全局变量维护当前相对栈底的偏移，以记录每个变量标号各自相对各自栈的栈底的偏移。每次进入新函数，更改栈帧寄存器 `$fp` 和 `$sp` 并将返回地址 `$ra` 和 `$fp` 存入内存，将当前所有标号写入内存，重置所有的变量寄存器为初始状态，`offset` 重置为 `$fp-$sp`。以上步骤思想类似于教材中的保存活动记录。函数传参时小于 4 个参数直接使用寄存器传承，多余 4 个的参数直接压栈传参。

针对不同标号，分别做如下的处理。

### 1) 常数

针对中间代码中形如 `#x` 的常数，若 `x` 为 0，则翻译成 `$0` 寄存器；若为非 0 常数，则直接翻译成 `x`。

### 2) 跳转标号

针对跳转标号 `_Lx`，由于跳转标号的翻译和寄存器分配无关，直接翻译为“`labelx:`”即可。

### 3) 地址访问标号

对如下形式的标号 `*_Tx`，申请寄存器号时当做 `_Tx` 申请，实际写成目标代码时应输出 `0 (reg(_Tx))` 的形式，表示寄存器间接寻址。

## 5.4 目标代码生成结果展示

实验 4 Makefile 文件内容如下

result: main.c lab2.1 syntax.y

bison -d syntax.y

flex lab2.1

gcc syntax.tab.c syntax.c genCode.c main.c -lfl -ly -o parser

运行后生成可执行文件，输入命令./parser test2.cmm out2.,生成目标代码如图所示。

```
1 .data
2 _prompt: .asciiz "Enter an integer:"
3 _ret: .asciiz "\n"
4 .globl main
5 .text
6
7 read:
8 subu $sp, $sp, 8
9 sw $ra, 4($sp)
10 sw $fp, 0($sp)
11 addi $fp, $sp, 8
12 li $v0, 4
13 la $a0, _prompt
14 syscall
15 li $v0, 5
16 syscall
17 subu $sp, $fp, 8
18 lw $ra, 4($sp)
19 lw $fp, 0($sp)
20 jr $ra
21
22 write:
23 subu $sp, $sp, 8
24 sw $ra, 4($sp)
25 sw $fp, 0($sp)
26 addi $fp, $sp, 8
27 li $v0, 1
28 syscall
29 li $v0, 4
30 la $a0, _ret
31 syscall
32 subu $sp, $fp, 8
33 lw $ra, 4($sp)
34 lw $fp, 0($sp)
35 move $v0, $0
36 jr $ra
37
38 .main New:
39 subu $sp, $sp, 8
40 sw $ra, 4($sp)
41 sw $fp, 0($sp)
42 addi $fp, $sp, 8
43 subu $t0, $fp, 20
44 move $t2, $t1
45 sw $t2, 0($t0)
46 subu $sp, $fp, 8
47 lw $ra, 4($sp)
48 lw $fp, 0($sp)
49 move $v0, $t0
50 jr $ra
51
52 .main fact:
53 subu $sp, $sp, 8
54 sw $ra, 4($sp)
55 sw $fp, 0($sp)
56 addi $fp, $sp, 8
57 move $a0, $t0
58 li $t1, 1
59 bne $t0, $t1, label1
60 label0:
61 subu $sp, $fp, 8
62 lw $ra, 4($sp)
63 lw $fp, 0($sp)
64 move $v0, $t0
65 jr $ra
66 j label2
67 label1:
68 addi $t2, $t0, -1
69 subu $v1, $fp, 16
70 sw $t2, 0($v1)
71 subu $v1, $fp, 12
72 sw $t0, 0($v1)
73 subu $sp, $fp, 16
74 subu $v1, $fp, 16
75 lw $t0, 0($v1)
76 move $a0, $t0
77 jal main fact
78 move $t2, $v0
79 subu $v1, $fp, 12
80 lw $t3, 0($v1)
81 lw $t3, 0($v1)
82 mul $t4, $t3, $t2
83 subu $sp, $fp, 8
84 lw $ra, 4($sp)
85 lw $fp, 0($sp)
86 move $v0, $t4
87 jr $ra
88 label2:
89 main:
90 subu $sp, $sp, 8
91 sw $ra, 4($sp)
92 sw $fp, 0($sp)
93 addi $fp, $sp, 8
94 subu $sp, $fp, 8
95 jal read
96 move $t0, $v0
97 move $t1, $t0
98 li $t2, 1
99 blt $t1, $t2, label4
100 label3:
101 subu $v1, $fp, 16
102 sw $t1, 0($v1)
103 subu $v1, $fp, 12
104 sw $t0, 0($v1)
105 subu $sp, $fp, 16
106 subu $v1, $fp, 16
107 lw $t0, 0($v1)
108 move $a0, $t0
109 jal main fact
110 move $t1, $v0
111 j label5
112 label4:
113 li $t3, 1
114 move $t1, $t3
115 label5:
116 subu $sp, $fp, 20
117 move $a0, $t1
118 jal write
119 subu $sp, $fp, 8
120 lw $ra, 4($sp)
121 lw $fp, 0($sp)
122 move $v0, $0
123 jr $ra
```

图表 5-1 目标代码生成

用 SPIM Simulator 对生成的目标代码进行测试，输入命令 spim -file out2.cmm，输入整数 7，输出对应的斐波那契数，结果如图。

```
u201414620@ubuntu:~/compile lab/lab4$ spim -file out2.cmm
SPIM Version 8.0 of January 8, 2010
Copyright 1990-2010, James R. Larus.
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /usr/lib/spim/exceptions.s
Enter an integer:7
5040
```

图表 5-2 生成的目标代码测试结果

## 6. 结束语

### 6.1 实践课程小结

此次实验的目标是借用 `flex` 和 `bison` 等工具，写一个简单面向对象语言 `decaf` 的编译器，分为四个任务，分别为词法语法分析、语义分析、中间代码生成、目标代码生成。

词法语法分析的主要是利用 `flex` 和 `bison` 工具来实现，需要用到语义分析里的部分关于属性文法知识，其他相关知识是理论课程所学，借助课本和实验书指导，实验容易理解，所花时间也不是很多。

语义分析的主要任务是建立符号表和静态语义分析。这个阶段是整个编译实验所花时间最多的地方。因为实验采用了多级符号表组织，在做实验时对符号表具体如何实现思考了很久。另外对于自底向上分析时左值引用的类型无法确定的问题我也花了很多时间思考，最后放弃了一遍遍历的方法，后面又增加了两次遍历，对符号表也做了改进，如节点中存储的类型转化为一个结构指针，使类型表达可以更加自由，自底向上传递的类型一般情况下其实公用一个类型结构空间，但对于左值引用，需要新建一个类型节点，重新分配空间，具体类型再第二次遍历的时候确定。

中间代码生成比较简单，主要是自顶向下遍历语法树，灵活使用前序和后续遍历等方法。但是中间还是出现了许多细节性问题，比如讨论不同情况如何产生中间代码的时候，对于不需要特定处理的部分忘记加语句 `return NULL` 而造成编译时不会报错，但当语法树中出现这种情况时会报错 `segmentation fault`。由于是通过语法树实现中间代码生成就不需要使用书上所说的拉链回填的技术。代码优化方面只做了一部分简单的优化，书上所说的系统代码优化没有实现。

生成目标代码的难点是寄存器的分配方法以及堆栈管理。思考后结合所学的知识采用了最久未使用 `LRU` 的寄存器淘汰算法，每个寄存器有一个 `old` 域记录未使用时间。另一个关键点是在调用函数的时候活动记录的处理。对返回地址和调用者栈帧需要进行压栈保存，对应调用者的变量标号要全部写内存。

四个实验完成后，也算是自己设计了一个较为建议的编译器，能够跑面向对象的 `dacaf` 代码。但仍然还有些功能还未实现，如多维数组等功能，但是如果时间充足相信这些功能都是能实现的。

## 6.2 自己的亲身体会

本次实验相比来说难度比较大，且任务量也比较重，时间跨度长，尤其是最后进入了考试复习阶段，又有其他几个大实验一起，确实有段时间会感受到痛苦。而且一开始做实验无法与理论知识联系起来，自己摸索着慢慢做一个个实验，每个实验花费的时间相比其他实验也比较长，可能做一个实验每天需要花七八个小时在电脑前。

虽然过程很辛苦，但在完成之后的确对编译原理的课程理论知识的了解更加深刻，比如 `syntax.output` 文件里有语法分析后产生的状态信息，又比如属性文法的实现过程等，寄存器分配中还用到了操作系统等其他课程所学的 LRU 最久未使用算法等等。这些理论知识在课程学过后如果不做实验相信过不久我就会忘记，但是做了此次实验加深了理解，有过这样一个经历也基本知道编译的基本原理和简单编译器的构造方法，也许一辈子也不会忘记这段时光。

除了收获课程知识，这次实验也让自己对 C 语言的实际应用有了一定了解。大一做过 C 语言的课设，但是 C 课设的内容还是比较简单，所用的知识无非就是十字链表。就算如此，在做 C 课设的时候我还是感觉到了吃力，通过这次编译的实验，自己代码能力也有所提高。

编译器完成了从高级语言向低级语言的一步步转变，而且最后经过修改代码可以在组原课设的 CPU 上运行，让自己觉得逐渐向一名兼具硬件知识和软件能力的工程师接近。总之，编译实验是非常有必要的，如果没大学没做这个实验，会是一个巨大损失。

## 参考文献

- [1] 吕映芝等. 编译原理(第二版). 北京: 清华大学出版社, 2005
- [2] 胡伦俊等. 编译原理(第二版). 北京: 电子工业出版社, 2005
- [3] 王元珍等. 80X86 汇编语言程序设计. 武汉: 华中科技大学出版社, 2005
- [4] 王雷等. 编译原理课程设计. 北京: 机械工业出版社, 2005
- [5] 曹计昌等. C 语言程序设计. 北京: 科学出版社, 2008