華中科技大學

课 程 实验 报 告

题目： Vector 语言 编译器

课程名称： 编译原理实验

专业班级： CS1703

学　　号： U201714670

姓　　名： 范 唯

指导教师： 徐丽萍

报告日期： 2020 年 7 月 25 日

计算机科学与技术学院

目录

1 概述

本次实验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。

本次将选择基于 miniC 与 python 的语言，将其命名为 Vector，实验的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，提高学生系统软件研发技术。以下是已实现的功能。

实验一 ：词法语法分析器的设计与实现

- 数据类型包括 char 类型、int 类型和 float 类型，字符串作为可选项；
- 基本运算包括算术运算、比较运算、自增自减运算和复合赋值运算；
- 控制语句包括 **if 语句、while 语句和 break、continue 语句**（不要求 goto 语句），另外 for、switch 语句作为可选项；
- **多维数组**。
- 语言支持行注释与块注释，不要求支持编译预处理命令和多文件程序.

实验二 ：符号表管理与语义检查

- (1)使用未定义的变量;
- (2)调用未定义或未声明的函数;
- (3)在同一作用域，名称的重复定义(如变量名、函数名、结构类型名以及结构体成员名等)。为更清楚说明语义错误，这里也可以拆分成几种类型的错误，如变量重复定义、函数重复定义、结构体成员名重复等;
- (4)对非函数名采用函数调用形式;
- (5)对函数名采用非函数调用形式访问;
- (6)函数调用时参数个数不匹配，如实参表达式个数太多、或实参表达式个数太少;
- (7)函数调用时实参和形参类型不匹配;
- (8)对非数组变量采用下标变量的形式访问;
- (9)数组变量的下标不是整型表达式;
- (10)对非结构变量采用成员选择运算符“.”;
- (11)结构成员不存在;
- (12)赋值号左边不是左值表达式;
- (13)对非左值表达式进行自增、自减运算;
- (14)对结构体变量进行自增、自减运算;
- (15)类型不匹配。如数组名与结构变量名间的运算，需要指出类型不匹配错误;
- (16)函数返回值类型与函数定义的返回值类型不匹配;
- (17)函数没有返回语句(当函数返回值类型不是 void 时);
- (18)break 语句不在循环语句或 switch 语句中;
- (19)continue 语句不在循环语句中;

实验三 ：中间代码生成与优化

- (1)中间代码结构；
- (2)基本快划分；
- (3)DAG 构造；
- (4)代码优化算法；

实验四 ：目标代码生成

- (1)指令系统选择；
- (2)寄存器分配算法；
- (3) AR 结构；
- (4)目标代码生成；
- (5)自己的 CPU 上运行增加项

2 系统描述

2.1 自定义语言概述

采用简化的 C 语言的文法，不妨将其称为 mini-c。其文法规则如下：

G[program]:

program → ExtDefList

ExtDefList→ExtDef ExtDefList | ε

ExtDef→Specifier ExtDecList ; |Specifier FunDec CompSt

Specifier→int | float |char

ExtDecList→VarDec | VarDec , ExtDecList

VarDec→ID | ID ArrayDec

ArrayDec→LB Exp RB | LB Exp RB ArrayDec

FucDec→ID (VarList) | ID ( )

VarList→ParamDec , VarList | ParamDec

ParamDec→Specifier VarDec

CompSt→{ DefList StmList }

StmList→Stmt StmList | ε

Stmt→Exp ; | CompSt | return Exp ; | if (Exp) Stmt | if (Exp) Stmt else Stmt | while (Exp) Stmt | FOR(Exp;Exp;Exp)Stmt | BREAK;|CONTIUNE;|;

DefList→Def DefList | ε

Def→Specifier DecList ;

DecList→Dec | Dec , DecList

Dec→VarDec | VarDec = Exp

Exp →Exp = Exp | Exp && Exp | Exp || Exp | Exp < Exp | Exp <= Exp
| Exp == Exp | Exp != Exp | Exp > Exp | Exp >= Exp| Exp + Exp
| Exp - Exp | Exp * Exp| Exp / Exp| Exp % Exp| Exp += Exp
| Exp -= Exp | Exp *= Exp | Exp /= Exp | Exp %= Exp| ID | INT
| FLOAT| (Exp) | - Exp | ! Exp | ID (Args) | ++Exp| Exp++
| --Exp | Exp-- | ID () | ID ArrayDec

Args→Exp , Args | Exp

2.2 单词文法与语言文法

2.2.1 单词文法描述

miniC 中的单词可以分为 6 类：标识符、关键字、运算符、界符、常量以及注释，具体每种单词的文法定义如表 2-1 所示。

表 2-1 单词的文法定义

单词符号说明	单词种类码	正则表达式

{ID}	ID	[A-Za-z][A-Za-z0-9]*
{INT}	INT	([0-9]+)|(0[xX][0-9a-fA-F]+)|(0[0-7]+)
{ FLOAT}	FLOAT	([0-9]*\.[0-9]+)|([0-9]+\.)
{CHAR}	CHAR	\'([^'\\]|\\['"?\\abfnrtv]|\\[0-7]{1,3}|\\[Xx][0-9A-Fa-f]+|{UCN})+\'
“int”	TYPE	
“float”	TYPE	
“char”	TYPE	
“return”	RETURN	
“if”	IF	
“else”	ELSE	
“while”	WHILE	
“for”	FOR	
“break”	BREAK	
“continue”	CONTINUE	
“;”	SEMI	
“,”	COMMA	
">"|"<"|">="|"<="|"=="|"!="	RELOP	
“=”	ASSIGNOP	
“+”	PLUS	
“-“	MINUS	
“*”	STAR	
“/”	DIV	
“%”	MOD	
“++”	AUTOADD	
“--”	AUTOSUB	
“+=”	COMADD	
“-=”	COMSUB	
“*=”	COMSTAR	
“/=”	COMDIV	
“%=”	COMMOD	
“&&”	AND	
“||”	OR	
“!”	NOT	
“(”	LP	

“)”	RP	
"["	LB	
"]"	RB	
“{”	LC	
“}”	RC	
[\n]	换行符无种类码	
[\r\t]	制表符无种类码	
"//"[^\n]*	注释，无种类码	
"/*" {BEGIN COMMENT;} <COMMENT>"*/" {BEGIN INITIAL;} <COMMENT>([^*]|\n)+|. <COMMENT><<EOF>>.	多行注释，无种类码	

2.2.2 语言文法描述

首先定义非终结符的类型，结合 bison 的语法规则，%type 定义非终结符的语义值类型，形式是%type <union 的成员名> 非终结符。

定义的非终结符如下：

%type <ptr> program ExtDefList ExtDef Specifier ExtDecList FuncDec ArrayDec CompSt VarList VarDec ParamDec Stmt StmList DefList Def DecList Dec Exp Args ArrayDec

其次，利用%token 定义终结符的语义值类型。

%token <type_int> INT,指定 INT 的语义值是 type_int，有词法分析得到的数值

%token <type_id> ID RELOP TYPE,指定 ID,RELOP 的语义值是 type_id,由词法分析得到的标识符字符串

%token <type_char> CHAR,指定 ID 的语义值是 type_char，由词法分析得到的标识符字符串

%token <type_float> FLOAT,指定 ID 的语义值是 type_id，由词法分析得到的标识符字符串

%token LP RP LC RC SEMI COMMA LB RB

%token COMADD COMSUB COMSTAR COMDIV COMMOD PLUS MINUS STAR DIV MOD ASSIGNOP AND OR NOT IF ELSE WHILE RETURN FOR AUTOADD AUTOSUB

%token BREAK CONTINUE

然后，定义运算符的优先级与结合性，如表 2-2 所示

表 2-2 算符优先级与结合性

优先级	结合性	符号
高	左	“+=”，”-=", “*=”，“/=”，“%=”
	左	“=”
	左	“||”
	左	“&&”
	左	">"|"<"|">="|"<="|"=="|"!="
	左	“+”, “-“
	左	“*”，“/”，“%”
	左	“++”，“--”
	右	“-”，“!”
	右	“[”
低	左	“]”

再次，是语法定义的核心部分：语法规则部分。语法规则由 2.1 中所定义的语法规则编写，具体实现代码如下：

```
program: ExtDefList      { display($1,0); semantic_Analysis0($1);}      //显示语法树,语义分析
                ;
/*定义整个语法树*/
ExtDefList: {$$=NULL;}  //表示整个语法树为空
                | ExtDef ExtDefList {$$=mknode(2,EXT_DEF_LIST,yylineno,$1,$2);}    //每一个 EXTDEFLIST 的结点，其第 1 棵子树对应一个外部变量声明或函数
                ;
/*外部声明，外部变量或函数*/
ExtDef:     Specifier ExtDecList SEMI {$$=mknode(2,EXT_VAR_DEF,yylineno,$1,$2);}    //该结点对应一个外部变量声明
                |Specifier FuncDec CompSt {$$=mknode(3,FUNC_DEF,yylineno,$1,$2,$3);}          //该结点对应一个函数定义
        // | Specifier ID ArrayDec {$$=mknode(2,ARRAY_DF,yylineno,$1,$2);} //外部数组定义
                | error SEMI     {$$=NULL;printf("\t missing SEMI \t\n");}
                ;

/*表示一个类型,补充对 char 类型的定义*/
Specifier:   TYPE {$$=mknode(0,TYPE,yylineno);strcpy($$->type_id,$1);$$->type=!strcmp($1,"int")?INT:(!strcmp($1,"float")?FLOAT:CHAR);}
                ;

/*变量名称列表*/
```

```
    ExtDecList:    VarDec          {$$=$1;}            /*每一个 EXT_DECLIST 的结点，其第一棵子树对应一个变量名(ID 类型的结点),第二棵子树对应剩下的外部变量名*/
                    | VarDec COMMA ExtDecList
{$$=mknode(2,EXT_DEC_LIST,yylineno,$1,$3);}
                    ;

    /*变量名称，由一个 ID 组成*/
    VarDec:    ID                  {$$=mknode(0,ID,yylineno);strcpy($$->type_id,$1);}
//ID 结点，标识符符号串存放结点的 type_id
        | ID ArrayDec
{$$=mknode(1,ARRAY_DF,yylineno,$2);strcpy($$->type_id,$1);}      //数组
        ;

    //补充数组声明
    ArrayDec: LB Exp RB {$$ =
$2;/*$$=mknode(1,ARRAY_DEC,yylineno,$2);strcpy($$->type_id,"ARRAY_DEC")
;*/}
                | LB Exp RB ArrayDec
{$$=mknode(2,ARRAY_DEC,yylineno,$2,$4);strcpy($$->type_id,"ARRAY_DEC");
}
            | error RB {$$=NULL;printf("\t define array error \t\n");}
            ;

    /*函数定义*/
    FuncDec: ID LP VarList RP
{$$=mknode(1,FUNC_DEC,yylineno,$3);strcpy($$->type_id,$1);}//函数名存放在
$$->type_id
        |ID LP    RP
{$$=mknode(0,FUNC_DEC,yylineno);strcpy($$->type_id,$1);$$->ptr[0]=NULL;}//
函数名存放在$$->type_id
        //| error RP {$$=NULL; printf("\t define function error\n");}
        ;

    //这里是参数列表
    VarList: ParamDec    {$$=mknode(1,PARAM_LIST,yylineno,$1);}
              | ParamDec COMMA    VarList
{$$=mknode(2,PARAM_LIST,yylineno,$1,$3);}
              ;

    //参数定义
    ParamDec: Specifier VarDec
{$$=mknode(2,PARAM_DEC,yylineno,$1,$2);}
              ;
```

```
    //复合语句
    CompSt: LC DefList StmList RC
{$$=mknode(2,COMP_STM,yylineno,$2,$3);}
          //| error RC {$$=NULL; printf("\t compst error\n");}
              ;

    /*语句列表，由 0 个或多个语句 stmt 组成*/
    StmList: {$$=NULL; }
              | Stmt StmList    {$$=mknode(2,STM_LIST,yylineno,$1,$2);}
              ;

    //语句
    Stmt:     Exp SEMI       {$$=mknode(1,EXP_STMT,yylineno,$1);}
            | CompSt         {$$=$1;}          //复合语句结点直接最为语句结点，
不再生成新的结点
            | RETURN Exp SEMI     {$$=mknode(1,RETURN,yylineno,$2);}
            | IF LP Exp RP Stmt %prec LOWER_THEN_ELSE
{$$=mknode(2,IF_THEN,yylineno,$3,$5);}
            | IF LP Exp RP Stmt ELSE Stmt
{$$=mknode(3,IF_THEN_ELSE,yylineno,$3,$5,$7);}
            | WHILE LP Exp RP Stmt {$$=mknode(2,WHILE,yylineno,$3,$5);}
            | FOR LP Exp SEMI Exp SEMI Exp RP Stmt
{$$=mknode(4,FOR_NODE,yylineno,$3,$5,$7,$9);}
            | BREAK SEMI
{$$=mknode(0,BREAK_NODE,yylineno);strcpy($$->type_id,"BREAK");}
            | CONTINUE SEMI
{$$=mknode(0,CONTINUE_NODE,yylineno);strcpy($$->type_id,"CONTINUE");}
            | SEMI {$$ =
mknode(0,BLANK,yylineno);strcpy($$->type_id,"BLANK");}    //空语句
            ;

    /*定义列表*/
    DefList: {$$=NULL; }
              | Def DefList {$$=mknode(2,DEF_LIST,yylineno,$1,$2);}
              | error SEMI      {$$=NULL;}
              ;

    /*定义*/
    Def:      Specifier DecList SEMI {$$=mknode(2,VAR_DEF,yylineno,$1,$2);}
          //| Specifier ID ArrayDec SEMI
{$$=mknode(2,ARRAY_DF,yylineno,$1,$2);}
              ;
```

```
    DecList: Dec    {$$=mknode(1,DEC_LIST,yylineno,$1);}
              | Dec COMMA DecList    {$$=mknode(2,DEC_LIST,yylineno,$1,$3);}
          ;
    Dec:        VarDec    {$$=$1;}
              | VarDec ASSIGNOP Exp
{$$=mknode(2,ASSIGNOP,yylineno,$1,$3);strcpy($$->type_id,"ASSIGNOP");}
              ;

    //表达式
    Exp:      Exp ASSIGNOP Exp
{$$=mknode(2,ASSIGNOP,yylineno,$1,$3);strcpy($$->type_id,"ASSIGNOP");}//$$
结点 type_id 空置未用，正好存放运算符
              | Exp AND Exp
{$$=mknode(2,AND,yylineno,$1,$3);strcpy($$->type_id,"AND");}
              | Exp OR Exp
{$$=mknode(2,OR,yylineno,$1,$3);strcpy($$->type_id,"OR");}
              | Exp RELOP Exp
{$$=mknode(2,RELOP,yylineno,$1,$3);strcpy($$->type_id,$2);}   //词法分析关系
运算符号自身值保存在$2 中
              | Exp PLUS Exp
{$$=mknode(2,PLUS,yylineno,$1,$3);strcpy($$->type_id,"PLUS");}
              | Exp MINUS Exp
{$$=mknode(2,MINUS,yylineno,$1,$3);strcpy($$->type_id,"MINUS");}
              | Exp STAR Exp
{$$=mknode(2,STAR,yylineno,$1,$3);strcpy($$->type_id,"STAR");}
              | Exp DIV Exp
{$$=mknode(2,DIV,yylineno,$1,$3);strcpy($$->type_id,"DIV");}
          | Exp MOD Exp
{$$=mknode(2,MOD,yylineno,$1,$3);strcpy($$->type_id,"MOD");}
          | Exp COMADD Exp
{$$=mknode(2,COMADD,yylineno,$1,$3);strcpy($$->type_id,"COMADD");}
              | Exp COMSUB Exp
{$$=mknode(2,COMSUB,yylineno,$1,$3);strcpy($$->type_id,"COMSUB");}
              | Exp COMSTAR Exp
{$$=mknode(2,COMSTAR,yylineno,$1,$3);strcpy($$->type_id,"COMSTAR");}
              | Exp COMDIV Exp
{$$=mknode(2,COMDIV,yylineno,$1,$3);strcpy($$->type_id,"COMDIV");}
              | Exp COMMOD Exp
{$$=mknode(2,COMMOD,yylineno,$1,$3);strcpy($$->type_id,"COMMOD");}
              | LP Exp RP          {$$=$2;}
              | MINUS Exp %prec UMINUS
{$$=mknode(1,UMINUS,yylineno,$2);strcpy($$->type_id,"UMINUS");}
              | NOT Exp
{$$=mknode(1,NOT,yylineno,$2);strcpy($$->type_id,"NOT");}
```

```
            | AUTOADD    Exp
{$$=mknode(1,AUTOADD,yylineno,$2);strcpy($$->type_id,"AUTOADD");}
            | Exp AUTOADD
{$$=mknode(1,AUTOADD,yylineno,$1);strcpy($$->type_id,"AUTOADD");}
        | AUTOSUB    Exp
{$$=mknode(1,AUTOSUB,yylineno,$2);strcpy($$->type_id,"AUTOSUB");}
            | Exp AUTOSUB
{$$=mknode(1,AUTOSUB,yylineno,$1);strcpy($$->type_id,"AUTOSUB");}
      // | LB Args RB      {$$=$2;}
        //| ID ArrayDec    {$$=mknode(1,ID,yylineno,$3);strcpy($$->type_id,$1);}

            | ID LP Args RP
{$$=mknode(1,FUNC_CALL,yylineno,$3);strcpy($$->type_id,$1);}    //带括号应该
是函数括号内的参数内容
            | ID LP RP
{$$=mknode(0,FUNC_CALL,yylineno);strcpy($$->type_id,$1);}
        | ID ArrayDec
{$$=mknode(1,ARRAY_CALL,yylineno,$2);strcpy($$->type_id,$1);}
            | ID                    {$$=mknode(0,ID,yylineno);strcpy($$->type_id,$1);}
            | INT
{$$=mknode(0,INT,yylineno);$$->type_int=$1;$$->type=INT;}
            | FLOAT
{$$=mknode(0,FLOAT,yylineno);$$->type_float=$1;$$->type=FLOAT;}
        | CHAR              {$$=mknode(0,CHAR,yylineno);
strcpy($$->type_char,$1);$$->type=CHAR;}
              //     | SEMI {$$ =
mknode(0,BLANK,yylineno);strcpy($$->type_id,"BLANK");}    //空语句
            ;
    Args:       Exp COMMA Args       {$$=mknode(2,ARGS,yylineno,$1,$3);}
              | Exp                          {$$=mknode(1,ARGS,yylineno,$1);}
              ;
```

2.3 符号表结构定义

在编译过程中，编译器使用符号表来记录源程序中各种名字的特性信息。这本次实验中采用的是单表组织形式来实现符号表，用一个符号栈来表示当前在作用域内的符号，每当有一个新的符号出现，则将其属性压栈到符号栈中，当符号出作用域则立即将这个符号弹出符号栈。

符号表的属性列包括：变量名、别名、层号、类型、标记以及偏移量。别名在后续的实验步骤中将会用到。层号用来记录符号所在的作用域的，每当程序进入一个复合语句时，层号就加一，退出时层号就减一。类型记录变量的数据类型或者是函数的返回值类型。偏移量记录变量的偏移地址。

符号表采用顺序表进行管理，用单表实现，用一个符号栈老表示在当前作用域内的付哈，每当有一个新的符号出现，则将新的符号以及对应的属性压入符号栈中。当作用域结束之后就将退栈。

符号表定义如下：

```
//符号表，这里设置为顺序栈，index 初值为 0
struct symboltable {
   struct symbol symbols[MAXLENGTH];
   int index;
} symbolTable;
```

其中 symbol 的结构如下所示：

```
struct symbol {      //这里只列出了一个符号表项的部分属性，没考虑属性间的互斥
    char name[33];  //变量或函数名
    int level;       //层号，外部变量名或函数名为 0，形参为 1，入栈即复合语句+1，出-1
    int type;        //变量类型或函数返回值类型
    int  paramnum;  //对函数适用，记录形式参数个数
    char alias[10];  //别名，为解决嵌套层次使用
    char flag;      //符号标记，函数：'F'  变量：'V'   参数：'P'  临时变量：'T'
    int offset;     //外部变量和局部变量在其静态数据区或活动记录中的偏移量
    int array[ARRAY_LEN]; //记录数组维度
};
```

2.4 错误类型码定义

词法分析由工具 flex 实现，该阶段的错误由 flex 根据正则表达式语法进行处理。语法分析阶段，bison 对单词流进行文法规则匹配，如果遇到不能符合任何语法结构时会自动报错。

语义分析阶段负责检查各种语义错误，主要包括：

（1）使用未定义的变量；

（2）调用未定义或未声明的函数；

（3）在同一作用域，名称的重复定义。如变量重复定义、函数重复定义等；

（4）对非函数名采用函数调用形式；

（5）对函数名采用非函数调用形式访问；

（6）函数调用时参数个数不匹配，如实参表达式个数太多、或实参表达式个数太少；

（7）函数调用时实参和形参类型不匹配；

（8）对非数组变量采用下标变量的形式访问；

（9）数组变量的下标不是整型表达式；

（10）赋值号左边不是左值表达式；

（11）对非左值表达式进行自增、自减运算；

（12）对结构体变量进行自增、自减运算；

（13）类型不匹配。如整形与字符型参与运算，字符型与浮点型参与运算等类型不匹配情况；

（14）函数返回值类型与函数定义的返回值类型不匹配；

（15）break 语句不在循环语句或 switch 语句中；

（16）continue 语句不在循环语句中；

2.5 中间代码结构定义

选用四元式作为中间代码的形式，各种定义如表 2-3 所示：

表 2-3 中间代码定义

语法	描述	Op	Opn1	Opn2	Result
x[i] := y	数组赋值	ASSIGNARRAY	Y	I	X
x := y	赋值操作	ASSIGNOP	X		X
x := y + z	加法操作	PLUS	Y	Z	X
x := y - z	减法操作	MINUS	Y	Z	X
x := y * z	乘法操作	STAR	Y	Z	X
x := y / z	除法操作	DIV	Y	Z	X
x := x + 1	自增操作	AUTOADD	X	1	X
x := x - 1	自减操作	AUTOSUB	X	1	X
FUNCTION f:	定义函数 f	FUNCTION			F
PARAM x	函数形参	PARAM			X
LABEL x	定义标号 x	LABEL			X
GOTO x	无条件转移	GOTO			X
IF x [relop] y GOTO z	条件转移	[relop]	X	Y	Z
ARG x	传实参 x	ARG			X
CALL f	调用函数	CALL	F		
PARAM x	函数形参	PARAM			X

LABEL x	定义标号 x	LABEL			X
BLOCK x	定义基本块 x	BLOCK			X

2.6 目标代码指令集选择

选用 MIPS 作为对应的目标代码，在生成目标代码时，要完成寄存器的分配，为了降低实现的难度，选择朴素的寄存器分配算法。中间代码与目标代码的对应关系如表 2-4 所示：

表 2-4 中间代码与目标代码的对应关系

中间代码	MIPS32 指令

x[i] := y	lw $t1, y 的偏移量($sp)
	move $t3, $t1
	lw $t2, i 的偏移量($sp)　#取出 i 的值
	mul $t2, $t2, int/char 的具体位宽
	addi $t2, $t2, x 的偏移量
	add $t2, $t2, $sp
	sw $t3, ($t2)

x :=#k	li　$t3,k sw $t3, x 的偏移量($sp)

x := y	lw $t1, y的偏移量($sp)
	move　$t3,$t1
	sw $t3, x的偏移量($sp)

x := y + z	lw $t1, y的偏移量($sp)
	lw $t2, z的偏移量($sp)
	add　$t3,$t1,$t2
	sw $t3, x的偏移量($sp)

x := y - z	lw $t1, y的偏移量($sp)
	lw $t2, z的偏移量($sp)
	sub　$t3,$t1,$t2
	sw $t3, x的偏移量($sp)

x := y * z	lw $t1, y的偏移量($sp)
	lw $t2, z的偏移量($sp)
	mul　$t3,$t1,$t2
	sw $t3, x的偏移量($sp)

x := y / z	lw $t1, y的偏移量($sp)
	lw $t2, z的偏移量($sp)
	mul　$t3,$t1,$t2
	div　$t1,$t2
	mflo　$t3
	sw $t3, x 的偏移量($sp)

x := x + 1	lw $t1, x的偏移量($sp)

	addi $t2, $t1, 1 sw $t2, %d($sp)

x := x - 1	lw $t1, x的偏移量($sp)
	addi $t2, $t1, -1
	sw $t2, %d($sp)

FUNCTION	函数名: addi $sp, $sp, 函数的活动空间大小
LABEL x:	标签名:
GOTO x:	j 标签名
RETURN x	move $v0, x 的偏移量($sp) jr $ra

IF x==y GOTO z	lw $t1, x的偏移量($sp)
	lw $t2, y的偏移量($sp)
	beq $t1,$t2,z

IF x!=y GOTO z	lw $t1, x的偏移量($sp) lw $t2, y 的偏移量($sp) bne $t1,$t2,z
IF x>y GOTO z	lw $t1, x的偏移量($sp) lw $t2, y 的偏移量($sp) bgt $t1,$t2,z
IF x>=y GOTO z	lw $t1, x的偏移量($sp) lw $t2, y 的偏移量($sp) bge $t1,$t2,z
IF x<y GOTO z	lw $t1, x的偏移量($sp) lw $t2, y 的偏移量($sp) blt $t1,$t2,z
IF x<=y GOTO z	lw $t1, x的偏移量($sp) lw $t2, y 的偏移量($sp) blt $t1,$t2,z

X:=CALL f	move $t0,$sp
	addi $sp, $sp, -函数活动空间大小
	sw $ra,0($sp)
	# 把形参压栈
	lw $t1, 函数的偏移量($t0)
	move $t3,$t1
	sw $t3, 形参的偏移量($sp)
	jal 函数名
	lw $ra,0($sp)
	addi $sp,$sp,函数的偏移量
	sw $v0,结果变量的偏移地址($sp)

3 系统设计与实现

3.1 词法分析器

词法分析器的构造技术线路，首选一个就是设计能准确表示各类单词的正则表达式。用正则表达式表示的词法规则等价转化为相应的有穷自动机 FA，确定化、最小化，最后依据这个 FA 编写对应的词法分析程序。

而在 miniC 中单词可分为 6 类：关键字、标识符、运算符、常量、界符、注释。因此可以考虑通过利用词法生成器自动化生成工具 Flex 编写 lex 文件，以正则表达式（正规式）的形式给出词法规则，遵循上述技术线路，Flex 自动生成给定的词法规则的词法分析程序，对指定的高级语言程序进行词法分析。

3.1.1 定义部分

定义部分其中可以有一个%{ 到%}的区间部分，主要包含 c 语言的一些宏定义，如文件包含、宏名定义，以及一些变量和类型的定义和声明，会直接被复制到词法分析器源程序 lex.yy.c 中。%{ 到%}之外的部分是一些正则式宏名的定义，例如：id [A-Za-z][A-Za-z0-9]*，定义了一个表示标识符的宏名 id，这些宏名在后面的规则部分会用到。

在 lex.l 文件中定义部分代码如下所示:

```
#include "parser.tab.h"
#include "string.h"
#include "def.h"
#include "stdio.h"
#define SHOW 0
int yycolumn=1;
#define YY_USER_ACTION    yylloc.first_line=yylloc.last_line=yylineno; \
yylloc.first_column=yycolumn;    yylloc.last_column=yycolumn+yyleng-1;
yycolumn+=yyleng;
typedef union {
int type_int;
float type_float;
char type_char[3];
char type_id[32];
struct node *ptr;
} YYLVAL;
#define YYSTYPE YYLVAL
```

正则表达式宏名定义部分代码如下：

```
UCN (\\u[0-9a-fA-F]{4}|\\U[0-9a-fA-F]{8})
```

```
ID      [A-Za-z][A-Za-z0-9]*
INT     ([0-9]+)|(0[xX][0-9a-fA-F]+)|(0[0-7]+)
FLOAT     ([0-9]*\.[0-9]+)|([0-9]+\.)
CHAR \'([^'\\]|\\["'?\\abfnrtv]|\\[0-7]{1,3}|\\[Xx][0-9A-Fa-f]+|{UCN})+\'
```

3.1.2 规则部分

规则部分的一条规则的组成为：

正则表达式　　动作

这部分以正则表达式的形式，罗列出所有种类的单词，表示词法分析器一旦识别出该正则表达式所对应的单词，就执行动作所对应的操作.

每当词法分析器识别出一个单词后，将该单词对应的字符串保存在 yytext 中，其长度为 yyleng，供后续使用。

规则部分代码如下：

```
{INT}             {if(SHOW) printf("(%s, INT)\n",
yytext);yylval.type_int=strtol(yytext,NULL,0);    return INT;}
    {FLOAT}       {if(SHOW) printf("(%s, FLOAT)\n",
yytext);yylval.type_float=atof(yytext); return FLOAT;}
    {CHAR}            {if(SHOW) printf("(%s, CHAR)\n",
yytext);yylval.type_char[0]=yytext[1];if(yytext[2] != '\'')
{yylval.type_char[1]=yytext[2];yylval.type_char[2]='\0';} else yylval.type_char[1] =
yylval.type_char[2] = '\0'; return CHAR;}

    "int"             {if(SHOW) printf("(%s, TYPR-INT)\n",
yytext);strcpy(yylval.type_id,    yytext);return TYPE;}
    "float"           {if(SHOW) printf("(%s, TYPE-FLOAT)\n",
yytext);strcpy(yylval.type_id,    yytext);return TYPE;}
    "char"            {if(SHOW) printf("(%s, TYPE-CHAR)\n",
yytext);strcpy(yylval.type_id,    yytext);return TYPE;}

    "return"          {if(SHOW) printf("(%s, RETURN)\n", yytext);return RETURN;}
    "if"              {if(SHOW) printf("(%s, RETURN)\n", yytext);return IF;}
    "else"            {if(SHOW) printf("(%s, ELSE)\n", yytext);return ELSE;}
    "while"           {if(SHOW) printf("(%s, WHILE)\n", yytext);return WHILE;}
    "for"             {if(SHOW) printf("(%s, FOR)\n", yytext);return FOR;}
    "break"           {if(SHOW) printf("(%s, BREAK)\n", yytext);return BREAK;}
    "continue"        {if(SHOW) printf("(%s, CONTINUE)\n", yytext);return
CONTINUE;}

    {ID}              {if(SHOW) printf("(%s, ID)\n", yytext);strcpy(yylval.type_id,
yytext); return ID;/*由于关键字的形式也符合标识符的规则，所以把关键字的处理全部放在标识符的前面，优先识别*/}
    ";"               {if(SHOW) printf("(%s, SEMI)\n", yytext);return SEMI;}
```

```
","             {if(SHOW) printf("(%s, COMMA)\n", yytext);return COMMA;}
">"|"<"|">="|"<="|"=="|"!=" {if(SHOW) printf("(%s, RELOP)\n",
yytext);strcpy(yylval.type_id, yytext);return RELOP;}
"="             {if(SHOW) printf("(%s, ASSIGNOP)\n", yytext);return
ASSIGNOP;}
"+"             {if(SHOW) printf("(%s, PLUS)\n", yytext);return PLUS;}
"-"             {if(SHOW) printf("(%s, MINUS)\n", yytext);return MINUS;}
"*"             {if(SHOW) printf("(%s, STAR)\n", yytext);return STAR;}
"/"             {if(SHOW) printf("(%s, DIV)\n", yytext);return DIV;}
"%"             {if(SHOW) printf("(%s, MOD)\n", yytext);return MOD;}

"++"            {if(SHOW) printf("(%s, AUTOADD)\n", yytext);return
AUTOADD;}
"--"            {if(SHOW) printf("(%s, AUTOSUB)\n", yytext);return
AUTOSUB;}
"+="            {if(SHOW) printf("(%s, COMADD)\n", yytext);return COMADD;}
"-="            {if(SHOW) printf("(%s, COMSUB)\n", yytext);return COMSUB;}
"*="            {if(SHOW) printf("(%s, COMSUB)\n", yytext);return COMSTAR;}
"/="            {if(SHOW) printf("(%s, COMDIV)\n", yytext);return COMDIV;}
"%="            {if(SHOW) printf("(%s, COMMOD)\n", yytext);return
COMMOD;}

"&&"              {if(SHOW) printf("(%s, AND)\n", yytext);return AND;}
"||"            {if(SHOW) printf("(%s, OR)\n", yytext);return OR;}
"!"             {if(SHOW) printf("(%s, NOT)\n", yytext);return NOT;}
"("             {if(SHOW) printf("(%s, LP)\n", yytext);return LP;}
")"             {if(SHOW) printf("(%s, RP)\n", yytext);return RP;}
"["         {if(SHOW) printf("(%s, LB)\n", yytext);return LB;}
"]"         {if(SHOW) printf("(%s, RB)\n", yytext);return RB;}
"{"             {if(SHOW) printf("(%s, LC)\n", yytext);return LC;}
"}"             {if(SHOW) printf("(%s, RC)\n", yytext);return RC;}

[\n]                {if(SHOW) printf("(\\n, line feed)\n");yycolumn=1;}
[ \r\t]             {}

"//"[^\n]*            {if(SHOW) printf("(%s, line comment)\n", yytext);}        /*跳
过单行注释*/
"/*" {BEGIN COMMENT;}
<COMMENT>"*/" {BEGIN INITIAL;}
<COMMENT>([^*]|\n)+|.       /*跳过多行注释*/
<COMMENT><<EOF>> {printf("%s：%d: Unterminated
comment\n",yytext,yylineno);return 0;}
.               {printf("Error type A :Mysterious character \"%s\"\n\t at
Line %d\n",yytext,yylineno);}
```

3.1.3 用户子程序部分

这部分代码会被原封不动的拷贝到 lex.yy.c 中，以方便用户自定义所需要执行的函数（包括之前的 main 函数）。如果用户想要对这部分用到的变量、函数或者头文件进行声明，可以前面的定义部分（即 Flex 源代码文件的第一部分）之前使用“%{“和”%}“符号将要声明的内容添加进去。被”%{“和”%}“所包围的内容也会被一并拷贝到 lex.yy.c 的最前面。

以下是用户子程序部分代码：

```
/* 和 bison 联用时，不需要这部分
void main()
{
yylex();
return 0;
}

*/
int yywrap()
{
return 1;
}
```

3.2 语法分析器

语法分析采用生成器自动化生成工具 GNU Bison（前身是 YACC），该工具采用了 LALR（1）的自底向上的分析技术，完成语法分析。在语法分析阶段，当语法正确时，生成抽象语法树，作为后续语义分析的输入。Bison 程序文件的扩展名为.y。Bison 源程序代码也分为三个部分，分别为声明部分、辅助定义部分、规则部分以及用户函数部分。

3.2.1 声明部分

%{到%}间的声明部分内容包含语法分析中需要的头文件包含，宏定义和全局变量的定义等，这部分会直接被复制到语法分析的 C 语言源程序中。

声明部分代码如下所示：

```
#include "stdio.h"
#include "math.h"
#include "string.h"
#include "def.h"
extern int yylineno;
extern char *yytext;
```

```
extern FILE *yyin;
int yylex();
void yyerror(const char* fmt, ...);
void display(struct ASTNode *,int);
```

3.2.2 辅助声明部分

在辅助声明部分，主要处理实验中要用到的几个主要内容：

（1） 终结符定义，在Flex和Bison联合使用时，parser.y 通过%token后面罗列出所有终结符(单词)的种类码标识符。

终结符定义代码如下：

```
%token LP RP LB RB LC RC SEMI COMMA          /*用bison对该文件编译时，带参数-d，生成的.tab.h中给这些单词进行编码，可在lex.l中包含parser.tab.h使用这些单词种类码*/
%token COMADD COMSUB COMSTAR COMDIV COMMOD PLUS MINUS STAR DIV MOD ASSIGNOP AND OR NOT IF ELSE WHILE RETURN FOR AUTOADD AUTOSUB //增加了+= -= *= /= %=
%token BREAK CONTINUE
```

（2） 语义值的类型定义，mini-c的文法中，每个符号（终结符和非终结符）都会有一个属性值，这个值的类型默认为整型。实际运用中，属性值的类型会有些差异，如ID的属性值类型是一个字符串，INT的属性值类型是整型。这样各种符号就会对应不同类型，这时可以用联合将这多种类型统一起来，联合定义代码如下：

```
%union {
    int       type_int;
    float     type_float;
    char      type_id[32];
    struct ASTNode *ptr;
      ......
    }
```

将所有符号属性值的类型用联合的方式统一起来后，某个符号的属性值就是该联合中的一个成员的值。

（3） 终结符属性值的类型说明，需要在%token定义终结符符号时指定其属性对应联合中的哪个成员。这样在parser.y文件中使用文法规则时，直接通过INT或ID，就可以使用整型常数自身值整数或标识符自身值字符串。

对应部分代码如下：

```
%token <type_int> INT                    /*指定INT的语义值是type_int，有词法分析得到的数值*/
%token <type_id> ID RELOP TYPE           /*指定ID,RELOP 的语义值
```

是type_id，有词法分析得到的标识符字符串*/

%token <type_char> CHAR

%token <type_float> FLOAT /*指定ID的语义值是type_id，有词法分析得到的标识符字符串*/

（4） 非终结符属性值的类型说明，对于非终结符，如果需要完成语义计算时，会涉及到非终结符的属性值类型，这个类型对应联合的某个成员，可使用格式：%type <union的成员名> 非终结符。

对应非终结符代码如下：

%type <ptr> program ExtDefList ExtDef Specifier ExtDecList FuncDec CompSt VarList VarDec ParamDec Stmt StmList DefList Def DecList Dec Exp Args ArrayDec

（5） 优先级与结合性定义。left表示左结合，right表示右结合，前面符号的优先级低，后面的优先级高。

相应的优先级与结合性定义代码如下所示：

```
%left COMADD COMSUB COMSTAR COMDIV COMMOD //+=这类复合运算优先级大于=
%left ASSIGNOP
%left OR
%left AND
%left RELOP
%left PLUS MINUS
%left STAR DIV MOD
%left AUTOADD AUTOSUB
%right UMINUS NOT
%right LB
%left RB
```

3.2.3 规则部分

Bison 采用的是 LR 分析法，需要在每条规则后给出相应的语义动作。具体来讲是在书写产生式。第一个产生式左边非终结符默认为初始符号。产生式里的箭头在这里用冒号”:”表示，一组产生式与另一组之间以分号“;”隔开。产生式里无论是终结符还是非终结符都各自对应一个属性值，产生式右边的符号值按从左到右的顺序依次对应$1、$2、$3。

规则部分代码如下所示：

```
program: ExtDefList { display($1,0); semantic_Analysis0($1);} //显示语法树,语义分析
        ;
/*定义整个语法树*/
ExtDefList: {$$=NULL;}  //表示整个语法树为空
```

```
        | ExtDef ExtDefList {$$=mknode(2,EXT_DEF_LIST,yylineno,$1,$2);}    //每一个 EXTDEFLIST 的结点，其第 1 棵子树对应一个外部变量声明或函数
        ;
    /*外部声明，外部变量或函数*/
    ExtDef: Specifier ExtDecList SEMI {$$=mknode(2,EXT_VAR_DEF,yylineno,$1,$2);}    //该结点对应一个外部变量声明
        |Specifier FuncDec CompSt {$$=mknode(3,FUNC_DEF,yylineno,$1,$2,$3);}          //该结点对应一个函数定义
        // | Specifier ID ArrayDec {$$=mknode(2,ARRAY_DF,yylineno,$1,$2);}
    //外部数组定义
        | error SEMI    {$$=NULL;printf("\t missing SEMI \t\n");}
        ;

    /*表示一个类型,补充对 char 类型的定义*/
    Specifier: TYPE {$$=mknode(0,TYPE,yylineno);strcpy($$->type_id,$1);$$->type=!strcmp($1,"int")?INT:(!strcmp($1,"float")?FLOAT:CHAR);}
        ;

    /*变量名称列表*/
    ExtDecList:  VarDec      {$$=$1;}        /*每一个 EXT_DECLIST 的结点，其第一棵子树对应一个变量名(ID 类型的结点),第二棵子树对应剩下的外部变量名*/
        | VarDec COMMA ExtDecList {$$=mknode(2,EXT_DEC_LIST,yylineno,$1,$3);}
        ;

    /*变量名称，由一个 ID 组成*/
    VarDec:  ID      {$$=mknode(0,ID,yylineno);strcpy($$->type_id,$1);} //ID 结点，标识符符号串存放结点的 type_id
        | ID ArrayDec {$$=mknode(1,ARRAY_DF,yylineno,$2);strcpy($$->type_id,$1);}    //数组
        ;

    //补充数组声明
    ArrayDec: LB Exp RB {$$ = $2;/*$$=mknode(1,ARRAY_DEC,yylineno,$2);strcpy($$->type_id,"ARRAY_DEC");*/}
        | LB Exp RB ArrayDec {$$=mknode(2,ARRAY_DEC,yylineno,$2,$4);strcpy($$->type_id,"ARRAY_DEC");}
        | error RB {$$=NULL;printf("\t define array error \t\n");}
```

```
            ;

    /*函数定义*/
    FuncDec: ID LP VarList RP {$$=mknode(1,FUNC_DEC,yylineno,$3);strcpy($$->type_id,$1);}// 函 数 名 存 放 在 $$->type_id
            |ID LP RP {$$=mknode(0,FUNC_DEC,yylineno);strcpy($$->type_id,$1);$$->ptr[0]=NULL;}// 函数名存放在$$->type_id
            //| error RP {$$=NULL; printf("\t define function error\n");}
            ;

    //这里是参数列表
    VarList: ParamDec  {$$=mknode(1,PARAM_LIST,yylineno,$1);}
            | ParamDec COMMA VarList {$$=mknode(2,PARAM_LIST,yylineno,$1,$3);}
            ;

    //参数定义
    ParamDec: Specifier VarDec {$$=mknode(2,PARAM_DEC,yylineno,$1,$2);}
            ;

    //复合语句
    CompSt: LC DefList StmList RC {$$=mknode(2,COMP_STM,yylineno,$2,$3);}
            //| error RC {$$=NULL; printf("\t compst error\n");}
            ;

    /*语句列表，由 0 个或多个语句 stmt 组成*/
    StmList: {$$=NULL; }
            | Stmt StmList  {$$=mknode(2,STM_LIST,yylineno,$1,$2);}
            ;

    //语句
    Stmt:  Exp SEMI  {$$=mknode(1,EXP_STMT,yylineno,$1);}
          | CompSt  {$$=$1;}  //复合语句结点直接最为语句结点，不再生成新的结点
          | RETURN Exp SEMI  {$$=mknode(1,RETURN,yylineno,$2);}
          | IF LP Exp RP Stmt %prec LOWER_THEN_ELSE {$$=mknode(2,IF_THEN,yylineno,$3,$5);}
          | IF LP Exp RP Stmt ELSE Stmt {$$=mknode(3,IF_THEN_ELSE,yylineno,$3,$5,$7);}
          | WHILE LP Exp RP Stmt {$$=mknode(2,WHILE,yylineno,$3,$5);}
```

```
        | FOR LP Exp SEMI Exp SEMI Exp RP Stmt {$$=mknode(4,FOR_NODE,yylineno,$3,$5,$7,$9);}
        | BREAK SEMI {$$=mknode(0,BREAK_NODE,yylineno);strcpy($$->type_id,"BREAK");}
        | CONTINUE SEMI {$$=mknode(0,CONTINUE_NODE,yylineno);strcpy($$->type_id,"CONTINUE");}
        | SEMI {$$ = mknode(0,BLANK,yylineno);strcpy($$->type_id,"BLANK");}  //空语句
        ;

    /*定义列表*/
    DefList: {$$=NULL; }
             | Def DefList {$$=mknode(2,DEF_LIST,yylineno,$1,$2);}
             | error SEMI    {$$=NULL;}
             ;

    /*定义*/
    Def:    Specifier DecList SEMI {$$=mknode(2,VAR_DEF,yylineno,$1,$2);}
            //| Specifier ID ArrayDec SEMI {$$=mknode(2,ARRAY_DF,yylineno,$1,$2);}
             ;

    DecList: Dec  {$$=mknode(1,DEC_LIST,yylineno,$1);}
             | Dec COMMA DecList  {$$=mknode(2,DEC_LIST,yylineno,$1,$3);}
            ;
    Dec:     VarDec  {$$=$1;}
             | VarDec ASSIGNOP Exp {$$=mknode(2,ASSIGNOP,yylineno,$1,$3);strcpy($$->type_id,"ASSIGNOP");}
             ;

    //表达式
    Exp: Exp ASSIGNOP Exp {$$=mknode(2,ASSIGNOP,yylineno,$1,$3);strcpy($$->type_id,"ASSIGNOP");}//$$结点 type_id 空置未用，正好存放运算符
        | Exp AND Exp {$$=mknode(2,AND,yylineno,$1,$3);strcpy($$->type_id,"AND");}
        | Exp OR Exp {$$=mknode(2,OR,yylineno,$1,$3);strcpy($$->type_id,"OR");}
        | Exp RELOP Exp {$$=mknode(2,RELOP,yylineno,$1,$3);strcpy($$->type_id,$2);}  //词法分析关系运算符号自身值保存在$2 中
        | Exp PLUS Exp {$$=mknode(2,PLUS,yylineno,$1,$3);strcpy($$->type_id,"PLUS");}
        | Exp MINUS Exp
```

```
{$$=mknode(2,MINUS,yylineno,$1,$3);strcpy($$->type_id,"MINUS");}
          | Exp STAR Exp
{$$=mknode(2,STAR,yylineno,$1,$3);strcpy($$->type_id,"STAR");}
          | Exp DIV Exp
{$$=mknode(2,DIV,yylineno,$1,$3);strcpy($$->type_id,"DIV");}
          | Exp MOD Exp
{$$=mknode(2,MOD,yylineno,$1,$3);strcpy($$->type_id,"MOD");}
          | Exp COMADD Exp
{$$=mknode(2,COMADD,yylineno,$1,$3);strcpy($$->type_id,"COMADD");}
          | Exp COMSUB Exp
{$$=mknode(2,COMSUB,yylineno,$1,$3);strcpy($$->type_id,"COMSUB");}
          | Exp COMSTAR Exp
{$$=mknode(2,COMSTAR,yylineno,$1,$3);strcpy($$->type_id,"COMSTAR");}
          | Exp COMDIV Exp
{$$=mknode(2,COMDIV,yylineno,$1,$3);strcpy($$->type_id,"COMDIV");}
          | Exp COMMOD Exp
{$$=mknode(2,COMMOD,yylineno,$1,$3);strcpy($$->type_id,"COMMOD");}
          | LP Exp RP      {$$=$2;}
          | MINUS Exp %prec UMINUS
{$$=mknode(1,UMINUS,yylineno,$2);strcpy($$->type_id,"UMINUS");}
          | NOT Exp
{$$=mknode(1,NOT,yylineno,$2);strcpy($$->type_id,"NOT");}
          | AUTOADD Exp
{$$=mknode(1,AUTOADD,yylineno,$2);strcpy($$->type_id,"AUTOADD");}
          | Exp AUTOADD
{$$=mknode(1,AUTOADD,yylineno,$1);strcpy($$->type_id,"AUTOADD");}
          | AUTOSUB Exp
{$$=mknode(1,AUTOSUB,yylineno,$2);strcpy($$->type_id,"AUTOSUB");}
          | Exp AUTOSUB
{$$=mknode(1,AUTOSUB,yylineno,$1);strcpy($$->type_id,"AUTOSUB");}
        // | LB Args RB     {$$=$2;}
        //| ID ArrayDec    {$$=mknode(1,ID,yylineno,$3);strcpy($$->type_id,$1);}

          | ID LP Args RP
{$$=mknode(1,FUNC_CALL,yylineno,$3);strcpy($$->type_id,$1);}   //带括号应该
是函数括号内的参数内容
          | ID LP RP
{$$=mknode(0,FUNC_CALL,yylineno);strcpy($$->type_id,$1);}
          | ID ArrayDec
{$$=mknode(1,ARRAY_CALL,yylineno,$2);strcpy($$->type_id,$1);}
          | ID      {$$=mknode(0,ID,yylineno);strcpy($$->type_id,$1);}
          | INT
{$$=mknode(0,INT,yylineno);$$->type_int=$1;$$->type=INT;}
          | FLOAT
```

```
{$$=mknode(0,FLOAT,yylineno);$$->type_float=$1;$$->type=FLOAT;}
          | CHAR          {$$=mknode(0,CHAR,yylineno);
strcpy($$->type_char,$1);$$->type=CHAR;}
          //                | SEMI          {$$ =
mknode(0,BLANK,yylineno);strcpy($$->type_id,"BLANK");}   //空语句
          ;
    Args:     Exp COMMA Args      {$$=mknode(2,ARGS,yylineno,$1,$3);}
          | Exp                   {$$=mknode(1,ARGS,yylineno,$1);}
          ;
```

3.2.4 用户函数部分

这部分的代码会被原封不动的拷贝到 parser.tab.c 中，以方便用户自定义所需要的函数。在这次实验中，提供了指错函数 yyerror, 一旦有词法、语法错误，即可准确、及时地进行报错，并给出错误位置以及错误性质。

用户函数部分代码如下所示：

```
int main(int argc, char *argv[]){
    yyin=fopen(argv[1],"r");
    if (!yyin) return -1;
    yylineno=1;
    yyparse();
    return 0;
    }

#include<stdarg.h>
void yyerror(const char* fmt, ...)
{
    va_list ap;
    va_start(ap, fmt);
    fprintf(stderr, "Grammar Error at Line %d Column %d: ",
yylloc.first_line,yylloc.first_column);
    vfprintf(stderr, fmt, ap);
    fprintf(stderr, ".\n");
}
```

3.2.5 建立抽象语法树

在语法分析阶段，一个很重要任务就是生成待编译程序的抽象语法树AST，AST不同于推导树，它去掉了一些修饰性的单词部分，简明扼要地把程序的语法结构表示出来，后续的语义分析、中间代码生成都可以通过遍历抽象语法树来完成。

AST抽象语法树节点的定义如下：

```
struct ASTNode {
    //以下对结点属性定义没有考虑存储效率，只是简单地列出要用到的一些属性
    //int kind;
    enum node_kind kind;   //节点类型
    union {
        char type_id[33];                  //由标识符生成的叶结点
        int type_int;                      //由整常数生成的叶结点
        float type_float;                  //由浮点常数生成的叶结点
        char type_char[3];   //由字符类型生成的叶节点
    };
    struct ASTNode* ptr[4];              //由kind确定有多少棵子树
    int place;                           //存放（临时）变量在符号表的位置序号
    char Etrue[15], Efalse[15];          //对布尔表达式的翻译时，真假转移目标的标号
    char Snext[15];                      //结点对应语句S执行后的下一条语句位置标号
    struct codenode* code;               //该结点中间代码链表头指针
    int type;                            //用以标识表达式结点的类型
    int pos;                             //语法单位所在位置行号
    int offset;                          //偏移量
    int width;                           //占数据字节数
    int num;                             //计数器，可以用来统计形参个数
};
```

建立抽象语法树节点的函数mknode代码如下所示：

```
struct ASTNode* mknode(int num, int kind, int pos, ...) {
    struct ASTNode* T = (struct ASTNode*)calloc(sizeof(struct ASTNode), 1);
    int i = 0;
    T->kind = kind;
    T->pos = pos;
    va_list pArgs;
    va_start(pArgs, pos);
    for (i = 0; i < num; i++)
        T->ptr[i] = va_arg(pArgs, struct ASTNode*);
    while (i < 4) T->ptr[i++] = NULL;
    va_end(pArgs);
    return T;
}
```

建立完抽象语法树AST后，可以通过先根遍历，逐一输出抽象语法树AST里的节点内容。遍历函数名display，代码设计如下：

```
void display(struct ASTNode* T, int indent)
{//对抽象语法树的先根遍历
```

```
#if SHOW

    int i = 1;
    struct ASTNode* T0;
    if (T)
    {
        switch (T->kind) {
        case EXT_DEF_LIST:
            display(T->ptr[0], indent);       //显示该外部定义（外部变量和函
数）列表中的第一个
            display(T->ptr[1], indent);       //显示该外部定义列表中的其它外
部定义
            break;
        case EXT_VAR_DEF:
            printf("%*c外部变量定义：(%d)\n", indent, ' ', T->pos);
            display(T->ptr[0], indent + 3);             //显示外部变量类型
            printf("%*c变量名：\n", indent + 3, ' ');
            display(T->ptr[1], indent + 6);             //显示变量列表
            break;
        case TYPE:
            printf("%*c类型：  %s\n", indent, ' ', T->type_id);
            break;
        case EXT_DEC_LIST:
            display(T->ptr[0], indent);       //依次显示外部变量名，
            display(T->ptr[1], indent);       //后续还有相同的，仅显示语法
树此处理代码可以和类似代码合并
            break;
        case FUNC_DEF:
            printf("%*c函数定义：(%d)\n", indent, ' ', T->pos);
            display(T->ptr[0], indent + 3);         //显示函数返回类型
            display(T->ptr[1], indent + 3);         //显示函数名和参数
            display(T->ptr[2], indent + 3);         //显示函数体
            break;
        case FUNC_DEC:
            printf("%*c函数名：%s\n", indent, ' ', T->type_id);
            if (T->ptr[0]) {
                printf("%*c函数形参：\n", indent, ' ');
                display(T->ptr[0], indent + 3);   //显示函数参数列表
            }
            else printf("%*c无参函数\n", indent + 3, ' ');
            break;
        case ARRAY_DF:
            printf("%*c  数组  %s", indent, ' ', T->type_id);
            display(T->ptr[0], indent + 3);
```

```
            printf("\n");
            break;
        case ARRAY_DEC:
            if (T->ptr[0])
                printf("[%d]", T->ptr[0]->type_int);
            display(T->ptr[1], 0);
            break;
        case PARAM_LIST:
            display(T->ptr[0], indent);        //依次显示全部参数类型和名
称，
            display(T->ptr[1], indent);
            break;
        case PARAM_DEC:
            printf("%*c类型：%s, 参数名：%s\n", indent, ' ',
T->ptr[0]->type_id, T->ptr[1]->type_id);
            break;
        case EXP_STMT:
            printf("%*c表达式语句：(%d)\n", indent, ' ', T->pos);
            display(T->ptr[0], indent + 3);
            break;
        case RETURN:
            printf("%*c返回语句：(%d)\n", indent, ' ', T->pos);
            display(T->ptr[0], indent + 3);
            break;
        case COMP_STM:
            printf("%*c复合语句：(%d)\n", indent, ' ', T->pos);
            printf("%*c复合语句的变量定义部分：\n", indent + 3, ' ');
            display(T->ptr[0], indent + 6);          //显示定义部分
            printf("%*c复合语句的语句部分：\n", indent + 3, ' ');
            display(T->ptr[1], indent + 6);          //显示语句部分
            break;
        case STM_LIST:
            display(T->ptr[0], indent);          //显示第一条语句
            display(T->ptr[1], indent);             //显示剩下语句
            break;
        case FOR_NODE:
            printf("%*cFOR循环语句：\n", indent, ' ');
            display(T->ptr[0], indent + 3); //显示循环条件
            printf("%*c循环体：\n", indent + 3, ' ');
            display(T->ptr[1], indent + 6); //显示循环体
            break;
        case FOR_DEC:
            //printf("%*c循环初始定义：\n", indent + 3, ' ');
            display(T->ptr[0], indent + 6);
```

```
            //printf("%*c循环条件：\n", indent + 3, ' ');
            display(T->ptr[1], indent + 6);
            //printf("%*c循环变换表达式：\n", indent + 3, ' ');
            display(T->ptr[2], indent + 6);
            break;
        case FOR_EXP1:
            printf("%*cFOR循环初始语句\n", indent, ' ');
            display(T->ptr[0], indent + 3);
            display(T->ptr[1], indent + 3);
            break;
        case FOR_EXP2:
            printf("%*cFOR循环条件语句\n", indent, ' ');
            T0 = T;
            while (T0)
            {
                display(T0->ptr[0], indent + 3);
                T0 = T0->ptr[1];
            }
            display(T->ptr[0], indent + 3);
            break;
        case FOR_EXP3:
            printf("%*cFOR循环结束条件\n", indent, ' ');
            T0 = T;
            while (T0)
            {
                display(T0->ptr[0], indent + 3);
                T0 = T0->ptr[1];
            }
            break;
        case BREAK_NODE:
            printf("%*cBreak语句\n", indent, ' ');
            break;
        case BLANK:
            //printf("%*c空语句\n", indent, ' ');
            break;
        case CONTINUE_NODE:
            printf("%*cContinue语句\n", indent, ' ');
            break;
        case WHILE:             printf("%*c循环语句：(%d)\n", indent, ' ',
T->pos);
            printf("%*c循环条件：\n", indent + 3, ' ');
            display(T->ptr[0], indent + 6);          //显示循环条件
            printf("%*c循环体：(%d)\n", indent + 3, ' ', T->pos);
            display(T->ptr[1], indent + 6);          //显示循环体
```

```
                break;
            case IF_THEN:           printf("%*c条件语句(IF_THEN)：(%d)\n",
indent, ' ', T->pos);
                printf("%*c条件：\n", indent + 3, ' ');
                display(T->ptr[0], indent + 6);          //显示条件
                printf("%*cIF子句：(%d)\n", indent + 3, ' ', T->pos);
                display(T->ptr[1], indent + 6);          //显示if子句
                break;
            case IF_THEN_ELSE:   printf("%*c条件语句(IF_THEN_ELSE)：
(%d)\n", indent, ' ', T->pos);
                printf("%*c条件：\n", indent + 3, ' ');
                display(T->ptr[0], indent + 6);          //显示条件
                printf("%*cIF子句：(%d)\n", indent + 3, ' ', T->pos);
                display(T->ptr[1], indent + 6);          //显示if子句
                printf("%*cELSE子句：(%d)\n", indent + 3, ' ', T->pos);
                display(T->ptr[2], indent + 6);          //显示else子句
                break;
            case DEF_LIST:
                display(T->ptr[0], indent);      //显示该局部变量定义列表中的第
一个
                display(T->ptr[1], indent);      //显示其它局部变量定义
                break;
            case VAR_DEF:
                printf("%*c局部变量定义：(%d)\n", indent, ' ', T->pos);
                display(T->ptr[0], indent + 3);    //显示变量类型
                display(T->ptr[1], indent + 3);    //显示该定义的全部变量名
                break;
            case DEC_LIST:
                printf("%*c变量名：\n", indent, ' ');
                T0 = T;
                while (T0) {
                    if (T0->ptr[0]->kind == ID)
                        printf("%*c %s\n", indent + 6, ' ', T0->ptr[0]->type_id);
                    else if (T0->ptr[0]->kind == ASSIGNOP)
                    {
                        if (T0->ptr[0]->ptr[0]->kind == ARRAY_DF)
                            display(T0->ptr[0]->ptr[0], indent);
                        //var_dec assignop exp的情况
                        printf("%*c %s ASSIGNOP\n ", indent + 6, ' ',
T0->ptr[0]->ptr[0]->type_id);
                        display(T0->ptr[0]->ptr[1], indent +
strlen(T0->ptr[0]->ptr[0]->type_id) + 7);            //显示初始化表达式
                    }
                    else if (T0->ptr[0]->kind == ARRAY_DF) {
```

```
                        printf("%*c  数组  %s\n", indent + 6, ' ',
T0->ptr[0]->type_id);
                        display(T0->ptr[0], indent);
                    }
                    T0 = T0->ptr[1];
                }
                break;
            case ARRAY_CALL:
                display(T->ptr[0], indent);
                printf("%*c数组下标：\n", indent, ' ');
                display(T->ptr[1], indent);
                break;
            case ID:
                printf("%*cID：  %s\n", indent, ' ', T->type_id);
                break;
            case INT:
                printf("%*cINT：%d\n", indent, ' ', T->type_int);
                break;
            case FLOAT:
                printf("%*cFLAOT：%f\n", indent, ' ', T->type_float);
                break;
            case CHAR:
                printf("%*cCHAR: %s\n", indent, ' ', T->type_char);
                break;
            case ASSIGNOP:
                T0 = T;
                if (T0->ptr[0]->kind == ARRAY_DF) {
                    printf("%*c  数组  %s", indent, ' ', T->type_id);
                    display(T0->ptr[0]->ptr[0], indent + 3);
                    printf("\n");
                }
            case AND:
            case OR:
            case RELOP:
            case PLUS:
            case MINUS:
            case STAR:
            case DIV:
            case COMADD:
            case COMSUB:
            case COMSTAR:
            case COMDIV:
            case COMMOD:
                printf("%*c%s\n", indent, ' ', T->type_id);
```

```
                display(T->ptr[0], indent + 3);
                display(T->ptr[1], indent + 3);
                break;
            case AUTOADD:
            case AUTOSUB:
                printf("%*c%s\n", indent, ' ', T->type_id);
                display(T->ptr[0], indent + 3);
                display(T->ptr[1], indent + 3);
                break;
            case NOT:
            case UMINUS:
                printf("%*c%s\n", indent, ' ', T->type_id);
                display(T->ptr[0], indent + 3);
                break;
            case FUNC_CALL:
                printf("%*c函数调用：(%d)\n", indent, ' ', T->pos);
                printf("%*c函数名：%s\n", indent + 3, ' ', T->type_id);
                display(T->ptr[0], indent + 3);
                break;
            case ARGS:          i = 1;
                while (T) {   //ARGS表示实际参数表达式序列结点，其第一棵
子树为其一个实际参数表达式，第二棵子树为剩下的
                    struct ASTNode* T0 = T->ptr[0];
                    printf("%*c 第 %d 个实际参数表达式：\n", indent, ' ', i++);
                    display(T0, indent + 3);
                    T = T->ptr[1];
                }
                //                          printf("%*c第%d个实际参数表达式：
\n",indent,' ',i);
                  //                        display(T,indent+3);
                printf("\n");
                break;
            }
        }

    #endif // SHOW
    }
```

3.3 符号表管理

符号表结构定义如 2.3 节所述，在本节不在赘述。

在语义分析过程中，各个变量名有其对应的作用域，一个作用域内不允许名字重复，为此，通过一个全局变量 LEV 来管理，LEV 的初始值为 0。这样在处

理外部变量名，以及函数名时，对应符号的层号值都是 0；处理函数形式参数时，固定形参名在填写符号表时，层号为 1。由于 mini_C 中允许有复合语句，复合语句中可定义局部变量，函数体本身也是一个复合语句，这样在 AST 的遍历中，通过 LEV 的修改来管理不同的作用域。

（1） 每次遇到一个复合语句的结点 COM_STM，首先对 LEV 加 1，表示准备进入一个新的作用域，为了管理这个作用域中的变量，使用栈 symbol_scope_TX，记录该作用域变量在符号表中的起点位置，即将符号表 symbolTable 的栈顶位置 symbolTable.index 保存在栈 symbol_scope_TX 中。

（2） 每次要登记一个新的符号到符号表中时，首先在 symbolTable 中，从栈顶向栈底方向查层号为 LEV 的符号，是否有和当前待登记的符号重名，是则报重复定义错误，否则使用 LEV 作为层号将新的符号登记到符号表中。

（3） 每次遍历完一个复合语句结点 COM_STM 的所有子树，准备回到其父结点时，这时该复合语句语义分析完成，需要从符号表中删除该复合语句的变量，方法是首先 symbol_scope_TX 退栈，取出该复合语句作用域的起点，再根据这个值修改 symbolTable.index，同时 LEV 减一，很简单地完成了符号表的符号删除操作。

（4） 符号表的查找操作，在 AST 的遍历过程中，分析各种表达式，遇到变量的访问时，在 symbolTable 中，从栈顶向栈底方向查询是否有相同的符号定义，如果全部查询完后没有找到，就是该符号没有定义；如果相同符号在符号表中有多处定义，按查找的方向可知，符合就近优先的原则。如果查找到符号后，就进一步进行语义分析，如：（1）函数调用时，根据函数名在符号表找到的是一个变量，不是函数，需要报错；（2）函数调用时，根据函数名找到这个函数，需要判断参数个数、类型是否匹配；（3）根据变量名查找的是一个函数。等等，需要做出各种检查。

3.4 语义检查

语义检查这部分完成的是静态语义分析，具体实现的检错功能如 2.4 节所述，在本节不在赘述。语义检查部分主要包括：

（1） 控制流检查。控制流语句必须使得程序跳转到合法的地方。例如一个跳转语句会使控制转移到一个由标号指明的后续语句。如果标号没有对应到语句，那么就出现一个语义错误。再者，break、continue 语句必须出现在循环语句当中。

在 mini-c 中没有定义各种转移语句，实验时可以考虑增加上去；教材中有 break 语句的介绍，可供参考。

（2） 唯一性检查。对于某些不能重复定义的对象或者元素，如同一作用域的标识符不能同名，需要在语义分析阶段检测出来。

（3） 名字的上下文相关性检查。名字的出现在遵循作用域与可见性的前提下应该满足一定的上下文的相关性。如变量在使用前必须经过声明，如果是面向对象的语言，在外部不能访问私有变量等等。

（4） 类型检查包括检查函数参数传递过程中形参与实参类型是否匹配、是否进行自动类型转换等等。

3.5 中间代码生成

3.5.1 AST 节点属性定义

为了完成中间代码的生成，对于 AST 中的结点，需要考虑设置以下属性，在遍历过程中，根据翻译模式给出的计算方法完成属性的计算。

.place 记录该结点操作数在符号表中的位置序号，这里包括变量在符号表中的位置，或每次完成了计算后，中间结果需要用一个临时变量保存，临时变量也需要登记到符号表中。另外由于使用复合语句，作用域可以嵌套，不同作用域中的变量可以同名，mini-c 语言和 C 语言一样采用就近优先的原则，但在中间语言中，没有复合语句区分层次，直接根据变量名对变量进行操作，无法区分不同作用域的同名变量，所以每次登记一个变量到符号表中时，会多增加一个**别名（alias）**的表项，通过别名实现数据的唯一性。翻译时，对变量的操作替换成对别名的操作，别名命名形式为 **v+序号**。生成临时变量时，命名形式为 **temp+序号，**在填符号表时，可以在符号名称这栏填写一个空串，临时变量名直接填写到别名这栏。

.type 一个结点表示数据时，记录该数据的类型，用于表达式的计算中。该属性也可用于语句，表示语句语义分析的正确性（OK 或 ERROR）。

.offset 记录外部变量在静态数据区中的偏移量以及局部变量和临时变量在活动记录中的偏移量。另外对函数，利用该数据项保存活动记录的大小。

.width 记录一个结点表示的语法单位中，定义的变量和临时单元所需要占用的字节数，借此能方便地计算变量、临时变量在活动记录中偏移量，以及最后计算函数活动记录的大小。

.code 记录中间代码序列的起始位置，如采用链表表示中间代码序列，该属性就是一个链表的头指针。

.Etrue 和.Efalse 该结点布尔表达式值为真、假时要转移的程序位置（标号字符串形式）。此属性仅对控制语句中的布尔表达式结点有效，其它情况属性值都是空串。

.Snext 该结点的语句序列执行完后，要转移到的程序位置（标号字符串形式）。

3.5.2 中间代码物理结构定义

opn 结构定义如下所示：

```
struct opn {
    int kind;       //标识联合成员的属性
    int type;       //标识操作数的数据类型
    union {
        int         const_int;          //整常数值，立即数
        float       const_float;        //浮点常数值，立即数
        char        const_char[3];      //字符常数值，立即数
        char        id[33];             //变量或临时变量的别名或标号字符串
    };
    int level;                          //变量的层号，0 表示是全局变量，数据保存在静态数据区
    int offset;                         //偏移量，目标代码生成时用
};
```

codenode 结构定义如下所示：

```
typedef struct codenode {     //三地址 TAC 代码结点
int    op;
struct opn opn1, opn2, result;
struct codenode* next, * prior;
} codenode;
```

其采用双向循环链表的方式存储中间代码。

3.5.3 翻译模式

定义完上述属性和结构体后，就需要根据翻译模式表示的计算次序，计算规则右部各个符号对应结点的代码段，再按语句的语义，将这些代码段拼接在一起，组成规则左部非终结符对应结点的代码段。

这种翻译模式表示的翻译方法，在实验时的具体体现是对抽象语法树进行遍历，在遍历过程中，完成各种属性的计算，并根据各语法成分的语义，完成中间代码的翻译。

当从函数定义结点开始，准备遍历函数体时，首先给函数体子树的根结点生成一个.Snext 属性，标识函数体语句执行完成后到达的位置，接着以此为起点，遍历函数体子树，计算函数体子树中所有结点的.Snext 属性。这样处理到每个语句结点时，都会有一个.Snext 属性值，方便语句的翻译。

部分执行语句类结点的中间代码翻译方法如表 4-1 所示：

表4-1 语句类结点的中间代码生成

当 前 结 点 类 型	翻 译 动 作
COMP_STM T1　说明部分子树 T2　语句部分子树	访问到 T：T2.Snext=T.Snext 访问 T 的所有子树后： T.code=T1.code || T2.code
IF_THEN T1　条件子树 T2　if 子句子树	访问到 T：T1.Etrue=newLabel, T1.Efalse= T2.Snext=T.Snext 访问 T 的所有子树后： T.code=T1.code || T1.Etrue ||T2.code
IF_THEN_ELSE T1　条件子树 T2　if 子句子树 T3　else 子句子树	访问到 T：　T1.Etrue=newLabel,T1.Efalse=T.Snext T1.Snext= T2.Snext=T.Snext 访问 T 的所有子树后： T.code=T1.code || T1.Etrue || T2.code || goto T.Enext || T1.Efalse || T3.code
WHILE T1　条件子树 T2　if 子句子树	访问到 T：T1.Etrue=newLabel, T1.Efalse=T.Snext, T2.Snext= newLabel; 访问 T 的所有子树后： T.code=T2.Snext || T1.code || T1.Etrue || T2.code || goto T2.Snext

STM_LIST
T1　语句 1 子树
T2　语句 2 子树（可空）
	访问到 T：if(T2 非空)　{T1.Snext=newLabel , T2.Snext=T.Snext;}
	else T1.Snext=S.next;
	访问 T 的所有子树后：
	if (T2 为空)　T.code=T1.code
	else　T.code=T1.Snext || T1.Snext || T2.code

EXP_STM T1　表达式子树	访问到 T：T1.Snext=T.Snext; 访问 T 的所有子树后：T.code=T1.code
RETURN T1　表达式子树（可空）	访问到 T：T1.Snext=T.Snext; 访问 T 的所有子树后： if (T1 非空)　T.code=T1.code|| return T1.alias else　T.code=T1.code|| return

部分基本表达式类结点的中间代码生成如表4-2所示：

表4-2 基本表达式类结点的中间代码生成

当 前 结 点 类 型	翻 译 动 作
INT 其它如 FLOAT 类的结点 按类似方法处理	t_i=newtemp, t_i 在符号表的入口赋值给 T.place。后续可通过 T.alias 读取该值。 T.code 为： t_i = INT 的值
ID	ID 在符号表中的入口赋值给 T.place，代码为空
ASSIGNOP T1 左值表达式子树 T2 左值表达式子树	访问到 T： T. palce=T1.place 访问 T 的所有子树后：T.code=T1.code || T2.code || T1.alias= T2.alias
OP 算术运算符。 T1 第一操作数子树 T2 第二操作数子树	t_i=newtemp, t_i 在符号表的入口赋值给 T.place 访问 T 的所有子树后： T.code=T1.code || T2.code || t_i=T1.alias OP T2.alias
UMINUS T1 操作数子树	t_i=newtemp, t_i 在符号表的入口赋值给 T.place 访问 T 的所有子树后：T.code=T1.code || t_i=- T1.alias
RELOP 关系运算符 T1 第一操作数子树 T2 第二操作数子树	t_i=newtemp, t_i 在符号表的入口赋值给 T.place， Label1=newLabel，Label2=newLabel。 访问 T 的所有子树后： T.code=T1.code || T2.code || if T1.alias RELOP T2.alias goto label1 || t_i=0 || goto label2 || label1： || t_i=1 || label2：
AND T1 第一操作数子树 T2 第二操作数子树	t_i=newtemp, t_i 在符号表的入口赋值给 T.place， Label1=newLabel。 访问 T 的所有子树后： T.code=T1.code || T2.code || t_i=T1.alias * T2.alias || if t_i==0 goto label1 || t_i=1 || label1：
OR T1 第一操作数子树 T2 第二操作数子树	t_i=newtemp, t_i 在符号表的入口赋值给 T.place， Label1=newLabel，Label2=newLabel。 访问 T 的所有子树后： T.code=T1.code || T2.code || t_i=0 || if T1.alias ==0 goto label1 || t_i=1 || goto label2 || label1： || if T2.alias ==0 goto label2 || t_i=1 || label2：
NOT T1 操作数子树	t_i=newtemp, t_i 在符号表的入口赋值给 T.place， Label1=newLabel，Label2=newLabel。 访问 T 的子树后： T.code= if T1.alias ==0 goto label1 || t_i=0 || goto label2 || label1： || t_i=1 || label2：
FUNC_CALL T1 实参列表子树	t_i=newtemp, t_i 在符号表的入口赋值给 T.place T.code=T1.code; 访问 T 的子树，从上至下依次对每个 ARGS 实参结点 T0，完成实参处理。 T.code= T. code || ARG T01.alias

	这里 T01 表示 T0 的第一个孩子，访问 T 的子树后： T.code= T.code || t_i=CALL 函数名

控制语句布尔表达式语句结点的中间代码生成如表 4-3 所示：

表 4-3 控制语句布尔表达式语句结点的中间代码生成

当 前 结 点 类 型	翻 译 动 作
INT 其它如 FLOAT 类的结点 按类似方法处理	if (T.Etrue==””) 按基本表达式处理 else if (INT 的值) T.code= goto T.Etrue else T.code= goto T.Efalse
ID	ID 在符号表中的入口赋值给 T.place if (T.Etrue==””) 按基本表达式处理 else T.code= if T.alias!=0 goto T.Etrue || goto T.Efalse
ASSIGNOP T1 左值表达式子树 T2 左值表达式子树	T.palce=T1.place 访问 T 的所有子树后： T.code=T1.code || T2.code || T1.alias= T2.alias if (T.true!=””) T.code=T.code || if T1.alias!=0 goto T.Etrue || goto T.Efalse
OP 算术运算符。 T1 第一操作数子树 T2 第二操作数子树	t_i=newtemp, t_i 入口赋值给当前结点 T.place 访问 T 的所有子树后： T.code =T1.code || T2.code || t_i=T1.alias OP T2.alias if (T.true!=””) T.code = T.code || if t_i!=0 goto T.Etrue || goto T.Efalse
UMINUS T1 操作数子树	t_i=newtemp, t_i 入口赋值给当前结点 place 访问 T 的所有子树后： T.code = T1.code || t_i=- T1.alias if (T.true!=””) T.code=T.code || if t_i!=0 goto T.Etrue || goto T.Efalse

RELOP 关系运算符
T1 第一操作数子树
T2 第二操作数子树
	if (T.true==””) t_i=newtemp, t_i 入口赋值给 T.place，
	Label1=newLabel，Label2=newLabel。
	访问 T 的所有子树后：
	T.code = T1.code || T2.code
	if (T.true!=””)
	T.code=T.code || if T1.alias RELOP T2.alias goto T.Etrue
	|| goto T.Efalse
	else T.code=T.code || if T1.alias RELOP T2.alias goto label1
	|| t_i=0 || goto label2 || label1： || t_i=1 || label2：

AND T1 第一操作数子树 T2 第二操作数子树	if (T.Etrue==””) 按基本表达式处理 else T1.Etrue= newLabel, T2.Etrue= T.Etrue, T1.Efalse= T2.Efalse =T.Efalse; T.code=T1.code || T1.Etrue || T2.code
OR T1 第一操作数子树	if (T.Etrue==””) 按基本表达式处理 else T1.Etrue=T2.Etrue= T.Etrue,T1.Efalse= newLabel,

T2　第二操作数子树	T2..Efalse =T.Efalse; T.code=T1.code || T1.Efalse || T2.code
NOT T1　操作数子树	f (T.Etrue==””) 按基本表达式处理 else　T1.Etrue=T.Efalse, T1.Efalse =T.Etrue; T.code=T1.code：
FUNC_CALL T1　实参列表子树	t_i=newtemp, t_i 入口赋值给 T.place T.code=T1.code; 访问 T 的子树，从上至下依次对每个 ARGS 结点 T0，完成实参处理。 T.code= T. code || ARG T01.alias 这里 T01 表示 T0 的第一个孩子，访问 T 的子树后： T.code= T.code || ti=CALL 函数名 f (T.Etrue!=””) T.code=T.code || if t_i!=0 goto T.Etrue || goto T.Efalse

其它类结点的翻译如表 4-4 所示。

表4-4 其它类结点的中间代码生成

当 前 结 点 类 型	翻 译 动 作
FUNC_DEF T1　返回值类型 T2　函数名与参数 T3　函数体	访问 T 时：T3.Snext=newLabel 访问 T 的所有子树后: T.code=T2.code || T3.code || T3.Snext
FUNC_DEC T1　参数列表（可空）	访问 T 的所有子树后: T.code=FUNCTION 函数名 if　(T1 非空) T.code=T.code || T1.code
PARAM_LIST T1 形参说明子树 T2 形参列表子树（可空）	访问 T 的所有子树后: T.code=T1.code if　(T2 非空)　T.code=T.code || T2.code
PARAM_DEC T1 形参类型 T2 形参名	访问 T 的所有子树后: T.code=PARAM T2.alias
ARGS T1 实参子树 T2 实参列表子树（可空）	访问 T 的所有子树后： if（T2==NULL）T.code= T1.code else　T.code= T1.code || T2.code

3.6 基本块划分

基本块划分相对较为简单，在这次实验中核心思想就是通过 Label、RETURN 以及跳转语句进行划分基本块，首先找出所有 GOTO 语句涉及到的 Label，接着在第一条语句之前生成第一个基本块 block，merge 进入 code 链表之后开始遍历，如果当前节点是 Label 并且前驱节点不是 Block，那么可以认定

为是其后代码块一个基本块，如果当前语句是跳转或者返回语句，并且后继语句不是 Block，那么也可以认为其后代码块也是一个基本块，之所以要判断后继语句不是 Block 节点，是避免最后一句 return 语句的后继节点为第一句语句的第一块基本块情况。因而基本块划分的代码如下：

```
codenode *splitBlock(codenode *head)
{
    codenode *p = head;
    int num;
    //找出所有 GOTO 涉及的 Label
    int *labelsTable = findAllGOTOLabel(head);
    //在开始语句之前生成一句 block
    p = merge(2, genBlock(newBlock()), p);
    head = p;
    do
    {
        //当前节点是 LABEL，并且前驱节点不是 BLOCK
        if (p->op == LABEL)
        {
            sscanf(&(p->result.id[5]), "%d", &num);
            if (labelsTable[num] == 1 && p->prior->op != BLOCK)
            {
                p = merge(2, genBlock(newBlock()), p);
            }
        }
        //当前节点是跳转，且下条语句不是 BLOCK
        if ((p->op == GOTO || p->op == JGE || p->op == JGT || p->op == JLE
|| p->op == JLT ||
             p->op == EQ || p->op == NEQ || p->op == RETURN) &&
            p->next->op != BLOCK)
            p->next = merge(2, genBlock(newBlock()), p->next);
        p = p->next;
    } while (head != p);
    return head;
}
```

3.7 代码优化

3.7.1 删除死代码

死代码在这次实验中定义为即使缺失也不会对程序运行产生任何影响的代码，在本次实验中，删除死代码这项工作主要应用于删除无用/多余的赋值语句

或者删除不会使用的变量对应的代码。

首先定义存储结构体 dead 如下所示：

```
struct dead
{
    char label[20]; //存储变量名
    int flag;
    int nlines;      //记录出现次数
    int line[20]; //记录出现位置
} var[255];    //变量结构
```

dead 结构数组 var 用来存储全局里出现的所有变量，首先逐行扫描中间代码，接着提取中间代码里出现的变量，如果目的变量是新变量就写入 var 变量数组，如果不是新变量，则判断有无上下文引用，没有上下文引用即可记录其出现位置，以便最后删除；对于运算变量则根据其出现状态做相应处理，如果处在非运算/赋值语句，则意味着存在上下文引用，flag 置 1，表示最后遍历时不能删除此处代码；如果处在运算/赋值语句则根据其出现次数保留最近更新的值即可。

删除死代码函数 deadcode 对应的代码设计如下所示：

```
void deadCode(int l)
{
    lineNo++; //行号递增
    int n, m, j;
    int state = isSPStatement(tokens[0]);
    if (state == 1)
    {
        j = find(tokens[1]);
        m = atoi(tokens[3]);
        n = find(tokens[5]);
    }
    else if (state == 2)
    {
        j = find(tokens[1]);
        m = -1;
        n = -1;
    }
    else
    {
        n = atoi(tokens[4]); //将操作数整形化
        m = atoi(tokens[2]);
        j = find(tokens[0]); //寻找目标变量结果的位置
    }
    if(containsLBRB(tokens[0]))
```

```
            return;
        if(l == 5 && isBeginWith("BLOCK",tokens[0]) == 0 &&
isBeginWith("LABEL",tokens[0]) == 0){
            //特殊情况  temp = call func \n
            if(j == -1){
                strcpy(var[varlen].label, tokens[0]);
                var[varlen].flag = 1;
                var[varlen].line[0] = lineNo;
                var[varlen].nlines = 1;
                varlen++;
            }else{
                var[j].flag = 1;
            }
            return;
        }
        if (varlen == 0 || j == -1) //写入新变量
        {
            if (state == 2 || state == 1){
                strcpy(var[varlen].label, tokens[1]);
                var[varlen].flag = 1;
            }
            else{
                strcpy(var[varlen].label, tokens[0]);
                var[varlen].flag = 0;
            }
            var[varlen].line[0] = lineNo;
            var[varlen].nlines = 1;
            varlen++;
        }
        else if (j > -1) //tokens[0]  也是变量
        {
            if (state == 2 || state == 1){
                var[j].flag = 1;
            }
            if (var[j].flag == 0)
            {
                var[j].line[var[j].nlines] = lineNo;
                var[j].nlines++;
            }
        }
        if (m == 0) //说明是变量
        {
            if (state == 1)
                j = find(tokens[3]);
```

```
            else
                j = find(tokens[2]);
            if (j > -1)
            {
                if (state == 2 || state == 1 || var[j].nlines == 1) //出现了，将 flag 置 1
                    var[j].flag = 1;
                    else{
                        var[j].nlines--;
                        var[j].flag = 0;
                    }
            }
        }
        if (n == 0 && strcmp(tokens[4], "0") && l > 3) //说明最后一个操作数是变
量
        {
            if (state == 1)
                j = find(tokens[5]);
            else
                j = find(tokens[4]);
            if (j > -1)
            {
                if (state == 2 || state == 1 || var[j].nlines == 1) //出现了，将 flag 置 1
                    var[j].flag = 1;
                    else{
                        var[j].nlines--;
                        var[j].flag = 0;
                    }
            }
        }
    }

    /给出剔除掉无用代码的结果 getLiveCode 函数设计如下所示：
    void getLiveCode(FILE *f, FILE *g)
    {
        int i, j, k = 0, n;
        int rmlines[1024];
        char tac[50];
        for (i = 0; i < varlen; i++)
        {
            if (var[i].flag == 0)
                for (j = 0; j < var[i].nlines; j++)
                    rmlines[k++] = var[i].line[j]; //记录变量行
        }
        n = k;
```

```
        k = 1;
        while (fgets(tac, 50, f))
        {
            for (i = 0; i < n; i++)
            {
                if (rmlines[i] == k)
                    break;
            }
            if (i == n)
            {
                //printf("%s", tac);
                fputs(tac, g);
            }
            k++;
        }
    }
```

3.7.2 变量替换

对于编译时产生的临时变量，如果其值固定为整数，那么就形成了一个 char* 到 int 型的替换关系，将产生的中间代码里的临时变量直接替换为给定的值即可。

由于实验测试用例不多加之代码能力较低，因而采用了单链表存储临时变量的变量名和变量值以及下一个临时变量节点的指针。对中间代码逐行遍历，当确定中间代码为形如“tempx = 1”类格式时，即可写入链表；如果当前行的中间代码有临时变量时，即可查询单链表，将临时变量替换为给定值。对应的查询函数代码如下所示：

```
int getValue(hash * head,char * key){
    hash * p = head;
    while (p->val != INTEGER_MINVALUE &&    p != NULL)
    {
        if(!strcmp(p->label,key))
            return p->val;
        else
            p = p->next;
    }
    return INTEGER_MINVALUE;
}
```

3.7.3 算数优化

算数优化主要工作在于对于能形如“x := 3*6”这类型可以直接算出运算结果的提前算出运算结果，同时对于乘除法能直接转化为移位运算的表达式也直接转化为移位运算，相应的代码如下所示：

```
void optimize(int l,hash * head,FILE * f)
{
    int n, m, i, j;
    float flo;
    if (l < 5)
    {
        for (i = 0; i < l; i++){
            j = strlen(tokens[i]);
            if(tokens[i][j-1] == '\n')
                tokens[i][j-1] = '\0';
            n = getValue(head,tokens[i]);
            if(n == INTEGER_MINVALUE){
                //printf("%s ", tokens[i]);
                fprintf(f,"%s ", tokens[i]);
            }
            else
            {
                //printf("%d ",n);
                fprintf(f,"%d ",n);
            }
        }
        return;
    }
    int state = isSPStatement(tokens[0]);
    char * tmp_1;
    char * tmp_2;
    if(state == 0){
        tmp_1 = tokens[2];
        tmp_2 = tokens[4];
    }else if(state == 1){
        tmp_1 = tokens[1];
        tmp_2 = tokens[3];
    }else {
        tmp_1 = tokens[1];
        tmp_2 = NULL;
    }

    m = atoi(tmp_1);
```

```
            n = atoi(tmp_2);
            if(n == 0 && tmp_2 != NULL && strcmp(tmp_2, "0")){
                j = getValue(head,tmp_2);
                if(j != INTEGER_MINVALUE)
                    n = j;
            }
            if(m == 0 && tmp_1 != NULL && strcmp(tmp_1, "0")){
                j = getValue(head,tmp_1);
                if(j != INTEGER_MINVALUE)
                    m = j;
            }
            if (m != 0 && n != 0)
            {
                if (strcmp(tokens[3], "+") == 0)
                    fprintf(f,"%s = %d", tokens[0], m + n);
                if (strcmp(tokens[3], "*") == 0)
                    fprintf(f,"%s = %d", tokens[0], m * n);
                if (strcmp(tokens[3], "/") == 0)
                    fprintf(f,"%s = %d", tokens[0], m / n);
                if (strcmp(tokens[3], "-") == 0)
                    fprintf(f,"%s = %d", tokens[0], m - n);
            }else if (strcmp(tokens[3], "*") == 0 && strcmp(tokens[4], "1\n") == 0)
                fprintf(f,"%s = %s", tokens[0], tokens[2]);
            else if (strcmp(tokens[3], "+") == 0 && strcmp(tokens[4], "0\n") == 0)
                fprintf(f,"%s = %s", tokens[0], tokens[2]);
           else if (strcmp(tokens[3], "*") == 0 && tmp_2[0] <= '9' && ceilf(flo = log2(n))
== flo)
                fprintf(f,"%s = %s << %d", tokens[0], tokens[2], (int)flo);
            else if (strcmp(tokens[3], "/") == 0 && tmp_2[0] <= '9' && ceilf(flo = log2(n))
== flo)
                fprintf(f,"%s = %s >> %d", tokens[0], tokens[2], (int)flo);
            else{
                for (i = 0; i < l; i++){
                    j = strlen(tokens[i]);
                    if(tokens[i][j-1] == '\n')
                        tokens[i][j-1] = '\0';
                    n = getValue(head,tokens[i]);
                    if(n == INTEGER_MINVALUE)
                        fprintf(f,"%s ", tokens[i]);
                    else
                    {
                        fprintf(f,"%d ",n);
                    }
                }
```

```
        }
    }
```

3.8 汇编代码生成

目标语言选定 MIPS32 指令序列，可以在 MARS 上运行，寄存器分配采用的是朴素寄存器分配算法。朴素寄存器分配算法的思想最简单，但也最低效，将所有的变量以及临时变量都放入内存中。如此一来，每翻译一条中间代码都需要把要用到的变量先加载到寄存器中，得到该代码的计算结果之后又需要将结果写回内存。这种方法的确能够将中间代码翻译成可正常运行的目标代码，而且实现和调试都特别容易，不过它最大的问题是寄存器的利用率实在太低。它不仅闲置了 MIPS 提供的大部分通用寄存器，那些未被闲置的寄存器也没有对减少目标代码的访存次数做出任何贡献。中间代码与汇编代码的对应关系在 2.6 节已经说明，本节就不在赘述了。直接注意的是在做数组的赋值时要取出数组首元素的偏移地址，并且取出变量 i 的偏移地址后读入 i 的值到寄存器，将 i 的值根据是 int 型数组还是 float 型数组或者 char 型数组乘上对应的位宽后加上数组首元素的偏移地址才能得到写入位置的具体偏移地址。

4 系统测试与评价

5.1 测试用例

实验一测试用例如下所示：

```
char c1[2][3][4];
int main(){
	int i,j,k;
	float x,y;
	i+=-i*++j;
	c[1][2]=10.12;
	for(int i = 0;i < 10; i++)
	{
	    if(i != j) break;
	    else continue;
	};
	return 'a';
}


int i,j;
float m,n;
float fun(int x,char y)
{
	int a[10],b[10][20];
	//i+j=b[3,5];
	j=0789;
	m=0xAB;
	return 1;

}

float test1(int x,char y)
{	int a,b;	//注释 1
 /*	注释 2
	  a=0；//
*/	a='a';
	    b='\n';
	 a*/*b+x*/b;
	;; return 1;
}
```

实验二测试代码如下所示：

```
int a, b, c;
float d, e, f;
char g;
int arr[5];
float b;   //变量名重复定义
char arr[10];   //变量名重复定义
int array[-5][0];   //数组定义错误
int funcA(int i, int j) {
    float x = 1.0;
    return x;   //函数返回值类型与函数定义的返回值类型不匹配
}

int funcA(int i, int j) {   //函数名重复定义

}

int funcB(){
    ;   //函数没有返回语句
}
int main() {
    k++;   //使用未定义的变量
    z = y * 10;   //使用未定义的变量
    callFunc(a);   //调用未定义或未声明的函数
    funcA--;   //对函数名采用非函数调用形式访问；
    a(i, j);   //对非函数名采用函数调用形式
    funcA(b);   //参表达式个数太少
    funcA(a, b, c); //  实参表达式个数太多
    funcA(d, e);   //函数调用时实参和形参类型不匹配
    a[2] = 5;   //对非数组变量采用下标变量的形式访问
    arr[2.5];   //数组变量的下标不是整型表达式
    arr['B'];   //数组变量的下标不是整型表达式
    b = 1 + 'A';   //类型不匹配
    d = 10 * 12.3;   //类型不匹配
    break;   //break 语句不在循环语句或 switch 语句中；
    continue;   //continue 语句不在循环语句中；
    10=a;   //赋值号左边不是左值表达式
    funcA(i,j) = i;
    funcA(i, j)++;//对非左值表达式进行自增、自减运算
    10++;
    ++10;
    'A'--;
    --'A';
    'A' = g;   //赋值号左边不是左值表达式
    g = 'A';   //合法
```

```
    g--;
    --g;
    ++g;
    g++;
    'A' += 1;   //复合运算类型匹配
    1.02 -= 'A';
    a *= 'A';
    20.96 /= 3;
    a[10] %= 10.0;
    a += 10.0;
    array[1][-5];   //数组访问出错
    array[1]['A'];
    return 'A';   //函数返回值类型与函数定义的返回值类型不匹配；
}
```

实验三测试用例代码如下所示：

```
int max(int a, int b)
{
    if (a > b)
        return a;
    else
        return b;
}
int main()
{
    int a = 1, b = 2, c = 3, d = 4, m, n;
    int k[10], i;
    int x, y, z;
    x = read();
    y = read();
    z = max(x, y);
    write(z);

    m = 1;
    m = a + b;
    n = a + b - m * (c + d) * (c + d) + 2 * 3;
    for (i = 0; i < 10; i++)
        k[i] = read();
    while (a < b)
    {
        if (a * 2 < b)
            continue;
        for (c = 1; c < 10; c++)
            if (c < 5)
                continue;
```

```
                else
                    break;
        }

        return 1;
}
```

实验四测试用例代码如下所示：

```
int a,b,c;
int fibo(int a)
{
        if (a == 0 || a == 1) return 1;
        return fibo(a-1)+fibo(a-2);
}

int main()
{
        int m,n,i;
        int x[5];
        m = read();
        i = 0;
        while(i <= m)
        {
                n = fibo(i);
                if(i <= 5)
                        x[i] = read();
                write(n);
                i++;
        }
        return 1;
}
```

5.2 正确性测试

实验一的测试结果如图 5-1 所示：

图 5-1 实验一测试结果图

实验二测试结果如图 5-2 所示：

Symbol Table

变量名	别名	层号	类型	标记	偏移量
read		0	int	F	4
write		0	int	F	4
a	v1	0	int	V	0
b	v2	0	int	V	4
c	v3	0	int	V	8
d	v4	0	float	V	12
e	v5	0	float	V	20
f	v6	0	float	V	28
g	v7	0	char	V	36
arr	v8	0	int[]	A	37
array	v11	0	int[]	A	41
funcA	v12	0	int	F	0
i	v13	1	int	P	12
j	v14	1	int	P	16
x	v15	1	float	V	20
	temp1	1	float	T	28

图 5-2 实验二测试结果

实验三初始中间代码如图 5-3 所示：

```
BLOCK block1 :
FUNCTION max
PARAM v2
PARAM v3
IF v2 > v3 GOTO label2
BLOCK block2 :
GOTO label3
BLOCK block3 :
LABEL label2 :
RETURN v2
BLOCK block4 :
GOTO label1
BLOCK block5 :
LABEL label3 :
RETURN v3
BLOCK block6 :
LABEL label1 :
FUNCTION main
temp1 = 1
v5 = temp1
temp2 = 2
v6 = temp2
temp3 = 3
```

图 5-3 初始中间代码

优化后中间代码如图 5-4 所示：

```
BLOCK block1 :
FUNCTION max
PARAM v2
PARAM v3
IF v2 > v3 GOTO label2
BLOCK block2 :
GOTO label3
BLOCK block3 :
LABEL label2 :
RETURN v2
BLOCK block4 :
GOTO label1
BLOCK block5 :
LABEL label3 :
RETURN v3
BLOCK block6 :
LABEL label1 :
FUNCTION main
v5 = 1
v6 = 2
v7 = 3
v8 = 4
temp5 = CALL read
```

图 5-4 优化后中间代码图

实验 4 生成的目标代码如图 5-5 所示：

```
27  fibo:
28    li $t3, 0
29    sw $t3, 16($sp)
30    lw $t1, 12($sp)
31    lw $t2, 16($sp)
32    beq $t1,$t2,label3
33    j label4
34  label4:
35    li $t3, 1
36    sw $t3, 16($sp)
37    lw $t1, 12($sp)
38    lw $t2, 16($sp)
39    beq $t1,$t2,label3
40    j label2
41  label3:
42    li $t3, 1
43    sw $t3, 16($sp)
44    lw $v0,16($sp)
45    jr $ra
46  label2:
47    li $t3, 1
48    sw $t3, 16($sp)
49    lw $t1, 12($sp)
50    lw $t2, 16($sp)
51    sub $t3,$t1,$t2
52    sw $t3, 20($sp)
53    move $t0,$sp
54    addi $sp, $sp, -44
55    sw $ra,0($sp)
56    lw $t1, 20($t0)
57    move $t3,$t1
58    sw $t3,12($sp)
59    jal fibo
60    lw $ra,0($sp)
61    addi $sp,$sp,44
62    sw $v0,24($sp)
63    li $t3, 2
64    sw $t3, 28($sp)
65    lw $t1, 12($sp)
66    lw $t2, 28($sp)
```

图 5-5 生成的目标代码图

在 mars 环境中输入 10 求出斐波那契数列前 10 项，并且依次读入 10-15 到数组，运行结果如图 5-6 所示：

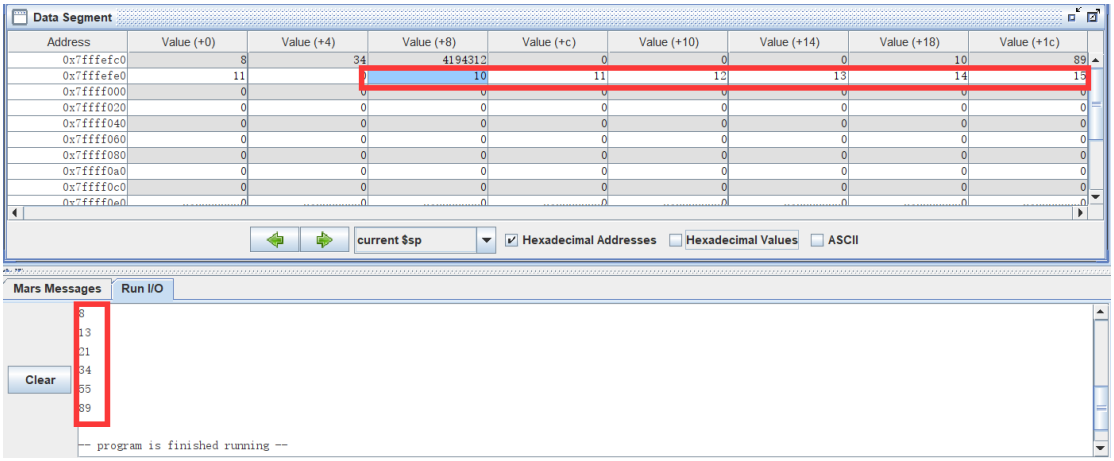

图 5-6 运行目标代码测试结果

5.3 报错功能测试

在实验二测试样例代码中已经通过注释的形式给出了 16 种错误代码的结果，关闭符号后的输出如图 5-7 所示：

图 5-7 报错功能测试结果图

5.4 系统的优点

系统在 miniC 的基础上，实现了单行注释、多行注释、char 类型、数组支持、continue、break 语句等多种功能，并且也实现了空语句、16 进制、8 进制赋值等功能，能够检测 16 种静态语义错误并给出详细的出错信息，能够详细详细地展示编译过程中每一步的结果。

5.5 系统的缺点

没有实现结构体、联合等高级数据结构，在对于中间代码的优化以及目标代码生成方面仍有很大的提升空间，包括但不限于优化不够、没有充分利用寄存器等情况。

6 实验小结或体会

编译原理实验可以说是大学 3 年以来所有实验里最难的一门实验，我在这次实验上耗费数不清的日日夜夜，整个实验的用时多达 100 多小时，最终才磕磕绊绊地完成了编译原理实验。

通过这四次编译原理实验任务我最终完成了一个 Vector 编译器，虽然这个编译器还有很多功能不支持，还有着这样那样的缺点与不足，但是在这次实验的过程中加深了我对编译器工作流程的理解，加深了我对编译器词法分析、语法分析、语义分析、中间代码生成以及最后目标代码生成的理解与掌握。

在这 4 次编译原理实验中，我所完成的内容如下：

1. 运用 flex 工具编写词法分析器，通过编写单词对应的正则表达式规则，实现了程序的词法分析。
2. 运用 bison 编写语法分析器，并在老师给定的框架上加入了新的语法规则，如 for 循环、break、continue 语句、数组识别等，实现了源代码的语法分析。
3. 完善实验指导教程里的 display 函数，实现了抽象语法树的遍历。
4. 按照实验指导教程设计符号表，并完成符号表里符号的管理。
5. 在前面完成内容的基础上，实现了静态语义检查，能够检查出代码里诸如类型不匹配，break、continue 语句出现位置错误，表达式需要左值等合计 16 种静态语义错误。
6. 实现了中间代码的生成，并根据生成的中间代码实现了基本块的划分以及进行了简单的代码优化。
7. 将生成的中间代码翻译为最终的目标代码，并且成功在 mars 在运行起来。

通过这次实验，我感受到编译原理实验是一个非常具有挑战性的实验，编译原理实验不仅巩固了课上所讲述的课程内容，也让我进一步了解到真正设计编程语言时应该注意到的问题以及诸如寄存器分配算法等许多课外知识。让我在这次实验中不仅巩固了课程知识，也学到了许多课外知识。此外，这次的编译原理实验进一步锻炼了我的编码能力，让我认识到我的编码效率之低下，逻辑之不严谨，考虑问题不够全面，暴露了我许多的问题。

编译原理是所有计算机课程里最硬的一门课，同时也是最重要的一门课，通过这次实验，我认识到我们学习的仅仅是冰山一角，要完成一个真正的编译器，需要付出的知识、人力时间和精力都不是这次实验所能比的。

最后，感谢《编译原理》课程组，感谢指导老师徐丽萍老师，通过这次实验让我看到自己的缺陷与不足，今后我也会更加地脚踏实地，刻苦学习，提高自己的知识技术水平。

参考文献

[1] 王生元 等. 编译原理(第三版). 北京：清华大学出版社，20016
[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