

华中科技大学

课程实验报告

题目: mini-c 编译器设计与实现

课程名称: 编译原理实验

专业班级:

学 号:

姓 名:

指导教师:

报告日期: 2020/07/30

计算机科学与技术学院

目录

1 概述	1
2 系统描述	2
2.1 自定义语言概述	2
2.2 单词文法与语言文法	2
2.3 符号表结构定义	4
2.4 错误类型码定义	5
2.5 中间代码结构定义	6
2.6 目标代码指令集选择	7
3 系统设计与实现	8
3.1 词法分析器	8
3.2 语法分析器	10
3.3 符号表管理	18
3.4 语义检查	19
3.5 报错功能	20
3.6 中间代码生成	22
3.7 代码优化	23
3.8 汇编代码生成	26
4 系统测试与评价	29
4.1 测试用例	29
4.2 正确性测试	30
4.3 报错功能测试	37
4.4 系统的优点	37
4.5 系统的缺点	37
5 实验小结或体会	38
参考文献	39

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。

可以根据自己对编程语言的喜好选择实现。建议大家选用 **decaf** 语言或 C 语言的简单集合 **SC** 语言。

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。

2 系统描述

2.1 自定义语言概述

实现目标语言为实验指导书中给出的简化的 C 语言-mini-c，在此基础上扩展了数组、结构、字符串的相关实现。

2.2 单词文法与语言文法

(1) 单词词法的定义：

表 2-1：单词的文法定义

单词符号	种类码	正则表达式
{char}	CHAR	(\.'\') (\'\.\')
{int}	INT	([1-9][0-9]*)(0)
{float}	FLOAT	([0-9]*\.[0-9+]) ([0-9]+\.)
{string}	STRING	\"[A-Za-z0-9]*"
“char”	TYPE	
“int”	TYPE	
“float”	TYPE	
"string"	TYPE	
"struct"	TYPE	
return	RETURN	
if	IF	
else	ELSE	
while	WHILE	
for	FOR	
id	ID	[A-Za-z][A-Za-z0-9]*
“.”	SEMI	
“,”	COMMA	
">" "<" ">=" "<=" "==" "!="	RELOP	
“=”	ASSIGNOP	
“+”	PLUS	
“-”	MINUS	
“*”	STAR	
“/”	DIV	
“&&”	AND	

" "	OR	
". "	DOT	
"!"	NOT	
"("	LP	
")"	RP	
"["	LB	
"]"	RB	
"{"	LC	
"}"	RC	
[\n]	yycolumn=1;	
[\r\t]	换行	
"/".*	单行注释	
"/*(([^*]*(\[^\])?)*)*"/	多行注释	

(2) 语言文法的定义

G[program]:

program \rightarrow ExtDefList

ExtDefList \rightarrow ExtDef ExtDefList $\mid \varepsilon$

ExtDef \rightarrow Specifier ExtDecList ; \mid Specifier FunDec CompSt

Specifier \rightarrow int \mid float

ExtDecList \rightarrow VarDec \mid VarDec , ExtDecList

VarDec \rightarrow ID

FucDec \rightarrow ID (VarList) \mid ID ()

VarList \rightarrow ParamDec , VarList \mid ParamDec

ParamDec \rightarrow Specifier VarDec

CompSt \rightarrow { DefList StmList }

StmList \rightarrow Stmt StmList $\mid \varepsilon$

Stmt \rightarrow Exp ; \mid CompSt \mid return Exp ;

\mid if (Exp) Stmt \mid if (Exp) Stmt else Stmt \mid while (Exp) Stmt

DefList \rightarrow Def DefList $\mid \varepsilon$

Def \rightarrow Specifier DecList ;

DecList \rightarrow Dec \mid Dec , DecList

Dec \rightarrow VarDec \mid 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 | ID | INT | FLOAT
    | ( Exp ) | - Exp | ! Exp | ID ( Args ) | ID ( )
Args → Exp , Args | Exp

```

在此文法的基础上扩展了数组和结构的实现。

2.3 符号表结构定义

符号表的结构体如下：

```

struct symbol
{
    char name[33]; //变量或函数名
    int level;     //层号
    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' 数组：'A'
    char 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，
    //或记录函数活动记录大小，目标代码生成时使用
    //函数入口等实验可能会用到的属性...
    struct symbol_array_info* array_info; //内情向量
};

```

其中 array_info 是在数组的实现中所需要的内情向量，symbol_array_info 的结构如下：

```

struct symbol_array_info
{
    //用来处理和数组相关的信息
    char id[20]; //数组名
    int dimension; //数组维度
    int type;     //数据类型
}

```

```

    int offset;    //偏移量
    int diminfo[MAXLENGTH]; //数组每维度的元素个数
    int tartget;   //访问目标
};

```

整个符号表的定义如下所示：

```

struct symboltable
{
    struct symbol symbols[MAXLENGTH];
    int index;
} symbolTable;

```

这里的符号表采用数组的形式来实现，此时的符号表 `symbolTable` 是一个顺序栈，栈顶指针 `index` 初始值为 0，每次填写符号时，将新的符号填写到栈顶位置，再栈顶指针加 1。

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` 自行处理。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) 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) 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; 有些需要根据定义的语言的语义自行进行界定, 比如: $32+'A'$, $10*12.3$, 如果使用强类型规则, 则需要报错, 如果按 C 语言的弱类型规则, 则是允许这类运算的, 但需要在后续阶段需要进行类型转换, 类型统一后再进行对应运算;
- (13) 函数返回值类型与函数定义的返回值类型不匹配;
- (14) 函数没有返回语句 (当函数返回值类型不是 `void` 时);
- (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 2-2 所示:

表 2-2: 中间代码定义

语法	描述	Op	Opn1	Opn2	Result
LABEL x	定义标号 x	LABEL			X
FUNCTION f:	定义函数 f	FUNCTION			F
x := y	赋值操作	ASSIGN	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
GOTO x	无条件转移	GOTO			X
IF x [relop] y GOTO z	条件转移	[relop]	X	Y	Z
RETURN x	返回语句	RETURN			X
ARG x	传实参 x	ARG			X
x:=CALL f	调用函数	CALL	F		X
PARAM x	函数形参	PARAM			X
READ x	读入	READ			X
WRITE x	打印	WRITE			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，选择了朴素的寄存器分配算法。中间代码与 MIPS 目标代码的对应关系如表 2-3 所示：

表 2-3 中间代码与 MIPS32 指令对应关系

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 + z	add reg(x), reg(y) , reg(z)
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)
GOTO x	j x
RETURN x	move \$v0, reg(x) jr \$ra
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	bge reg(x),reg(y),z
IF x<y GOTO z	blt reg(x),reg(y),z
IF x<=y GOTO z	ble reg(x),reg(y),z
X:=CALL f	jal f move reg(x),\$v0

3 系统设计与实现

3.1 词法分析器

mini-c 中需要识别的单词有五类：关键字（保留字）、运算符、界符、常量和标识符。依据词法分析器的构造技术线路，首选一个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。实验中，词法分析器可采用词法生成器自动化生成工具 GNU Flex，以正则表达式的形式给出词法规则，Flex 自动生成给定的词法规则的词法分析程序。

(1) 关键字：需要完全匹配的保留单元

```
{int}          {yyval.type_int=atoi(yytext); return INT;}
{float}        {yyval.type_float=atof(yytext); return FLOAT;}
{char}         {if(yytext[1]!='\\')
                {yyval.type_char[0]=yytext[1];return CHAR;}
                else
                {yyval.type_char[0] = '\\';
                yyval.type_char[0] = yytext[2];return CHAR;}}
```



```
{string}       {strcpy(yyval.type_string,yytext);return STRING;}
```



```
"int"          {strcpy(yyval.type_id,  yytext);return TYPE;}
"float"        {strcpy(yyval.type_id,  yytext);return TYPE;}
"char"         {strcpy(yyval.type_id,  yytext);return TYPE;}
"string"       {strcpy(yyval.type_id,  yytext);return TYPE;}
```



```
"return"       {return RETURN;}
"if"           {return IF;}
"else"         {return ELSE;}
"while"        {return WHILE;}
"for"          {return FOR; }
"struct"       {return STRUCT;}
"break"        {return BREAK;}
"continue"     {return CONTINUE;}
```

(2) 标识符：以字母开头的字母数字串

```
id    [A-Za-z][A-Za-z0-9]*  
{id}    {strcpy(yylval.type_id,yytext); return ID; }
```

(3) 常量：包括整形常量、浮点常量、字符常量以及字符串常量

```
id    [A-Za-z][A-Za-z0-9]*  
int    ([1-9][0-9]*)(0)  
float  ([0-9]*\.[0-9]+)|([0-9]+\.)  
char   ('\.'|'\'\\.\')  
string  (\".*\")
```

(4) 界符：

```
";"      {return SEMI;}  
","      {return COMMA;}  
"("      {return LP;}  
")"      {return RP;}  
"{"      {return LC;}  
"}"      {return RC;}  
"["      {return LB;}  
"]"      {return RB;}  
[\n]      {yycolumn=1;}  
[ \r\t]    {}
```

(5) 运算符：包括算数运算符以及逻辑运算符等

```
>"<">="<="=="!=" {strcpy(yylval.type_id, yytext);;return RELOP;}  
"="      {return ASSIGNOP;}  
"+"      {return PLUS;}  
"-"      {return MINUS;}  
"*"      {return STAR;}  
"/"      {return DIV;}  
"++"     {return DPLUS;}  
"--"     {return DMINUS;}  
"&&"     {return AND;}  
"||"     {return OR;}
```

```
"!" {return NOT;}
```

(6) 注释:

```
"//".* { } //单行注释
"/*"([^\*]*(*[^\*\/])?)*"*/" { } //多行注释
```

3.2 语法分析器

采用生成器自动化生成工具 GNU Bison 进行语法分析，该工具采用了 LALR (1) 的自底向上的分析技术，完成语法分析。在实验的语法分析阶段，当语法正确时，生成抽象语法树，作为后续语义分析的输入。

(1) 单词种类码

在 Flex 和 Bison 联合使用时，需要在 parser.y 中的 %token 后面罗列出所有终结符(单词)的种类码标识符，如：

```
%token <type_int> INT
%token <type_id> ID RELOP TYPE
%token <type_float> FLOAT
%token <type_char> CHAR
%token <type_string> STRING
%token LP RP LC RC SEMI COMMA
%token PLUS MINUS STAR DIV ASSIGNOP AND OR NOT IF ELSE WHILE
RETURN FOR SWITCH CASE COLON DEFAULT
//+ - * / = & | ! if else return for switch case colon default
%token LB RB DPLUS DMINUS EXP_DPLUS EXP_DMINUS DPLUS_EXP
DMINUS_EXP PLUS_ASSIGNOP MINUS_ASSIGNOP STAR_ASSIGNOP
DIV_ASSIGNOP ARRAY_DEF ARRAY_READ BREAK CONTINUE FOR_DEC
DOT STRUCT STRUCT_READ
// [ ] ++exp exp++ --exp exp-- += -= *= /=
//以下为接在上述 token 后依次编码的枚举常量，作为 AST 结点类型标记
%token EXT_DEF_LIST EXT_VAR_DEF FUNC_DEF FUNC_DEC
EXT_DEC_LIST PARAM_LIST PARAM_DEC VAR_DEF DEC_LIST DEF_LIST
COMP_STM STM_LIST EXP_STMT IF_THEN IF_THEN_ELSE STRUCTNAME
STRUCT_DEF STRUCT_DEC
%token FUNC_CALL ARGS FUNCTION PARAM ARG CALL LABEL GOTO
JLT JLE JGT JGE EQ NEQ
```

(2) 语义值的类型定义

mini-c 的文法中，有终结符，也有非终结符，如 `ExtDefList` 表示外部定义列表，`CompSt` 表示复合语句等。每个符号都会有一个属性值，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。语法分析时，需要建立抽象语法树，这时 `ExtDefList` 的属性值类型会是树结点（类型为 `struct ASTNode`）的指针。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，某个符号的属性值就是该联合中的一个成员的值：

```
%union {
    int    type_int;
    float  type_float;
    char   type_char[2];
    char   type_string[31];
    char   type_id[32];
    struct ASTNode *ptr;
};
```

(3) 非终结符的属性值的类型说明

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。例如 `parser.y` 中的：

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

(4) 优先级与结合性定义

对 `Bison` 文件进行翻译，得到语法分析程序的源程序时，通常会出现报错，大部分是移进和归约(`shift/reduce`)，归约和归约(`reduce/reduce`)的冲突类的错误。为了改正这些错误，需要了解到底什么地方发生错误。这时，使用 `bison -d -v parser.y`，会生成一个文件 `parser.output`。打开该文件，开始几行说明哪几个状态有多少个冲突项，再根据这个说明中的状态序号去查看对应的状态进行分析、解决错误，常见的错误一般都能通过单词优先级和结合性的设定解决，`mini-c` 中设定的优先级规则为：

```
%right ASSIGNOP PLUS_ASSIGNOP MINUS_ASSIGNOP STAR_ASSIGNOP
DIV_ASSIGNOP /= += -= *= /=
```

```

%left OR |||
%left AND // &&
%left RELOP // ">" "<" ">=" "<=" "==" "!="
%left PLUS MINUS //+ -
%left STAR DIV // * /
%right UMINUS NOT DPLUS DMINUS    //- ! ++ --
%right LB//(
%left  RB/)
%left DOT
%nonassoc LOWER_THEN_ELSE
%nonassoc ELSE

```

其中，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。另外对于推导式 $\text{Exp} \rightarrow -\text{Exp}$ ，单目-的运算优先级高于*与/，而词法分析时，无论是单目-还是双目-，识别出的种类码都是 MINUS，为此，需要在定义一个优先级高的单目-符号 UMINUS：

$\text{Exp} \rightarrow \text{MINUS Exp } \%prec \text{ UMINUS}$

表示这条规则的优先级等同于 UMINUS，高于乘除，这样对于句子 $-a*b$ 就会先完成 $-a$ 归约成 Exp，即先处理单目-，后处理*

(5) 文法规则和规约动作

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,例如对规则： $\text{Exp} \rightarrow \text{Exp} = \text{Exp}$ ，在parser.y中为：

```
Exp:  Exp ASSIGNOP Exp { $$=mknode(2,ASSIGNOP,yylineno,$1,$3); }
```

规则后面{}中的的是当完成归约时要执行的语义动作。规则左部的 Exp 的属性值用\$\$表示，右部有 2 个 Exp，位置序号分别是 1 和 3，其属性值分别用\$1和\$3 表示。在附录 4 中，定义了 mknode 函数，完成建立一个树结点，这里的语义动作是将建立的结点的指针返回赋值给规则左部 Exp 的属性值，表示完成此次归约后，生成了一棵子树，子树的根结点指针为\$\$，根结点类型是 ASSIGNOP，表示是一个赋值表达式。该子树有 2 棵子树，第一棵是\$1 表示的左值表达式的子树，第二棵对应是\$3 的表示的右值表达式的子树，yylineno 表示赋值语句的行号。

Mini-c 所使用的文法规则和规约动作为：

```

//program: 初始语法单元

program: ExtDefList    { display($1,0);

                        //semantic_Analysis0($1);

```

```

    } //显示语法树,语义分析

;

//ExtDefList: 零个或多个 ExtDef
ExtDefList:      {$$=NULL;}
    | ExtDef ExtDefList
{$$=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

;

//ExtDef: 一个全局变量、结构体或函数的定义
ExtDef:  Specifier ExtDecList SEMI
{$$=mknode(2,EXT_VAR_DEF,yylineno,$1,$2);} //该结点对应一个外部变量声明
    | Specifier SEMI {$$=$1;} //对应只有声明没有变量名的情况
    | Specifier FuncDec CompSt
{$$=mknode(3,FUNC_DEF,yylineno,$1,$2,$3);} //该结点对应一个函数定义
    | error SEMI {$$=NULL;}

;

//StructSpecifier: 结构体描述符
StructSpecifier: STRUCT StructName LC DefList RC
{$$=mknode(2,STRUCT_DEF,yylineno,$2,$4);} //定义结构体
    | STRUCT ID
{$$=mknode(1,STRUCT_DEC,yylineno,$2);strcpy($$->type_id,$2);} //定义结构体变
量

;

//StructName: 结构体名
StructName:  {$$=NULL;}
    | ID
{$$=mknode(0,STRUCTNAME,yylineno);strcpy($$->type_id,$1);}

;

//Specifier: 类型描述符
Specifier: TYPE  {$$=mknode(0,TYPE,yylineno);strcpy($$->type_id,$1);
    if(!strcmp($1,"int")) $$->type=INT;
    if(!strcmp($1,"float")) $$->type=FLOAT;
    if(!strcmp($1,"char")) $$->type=CHAR;
    if(!strcmp($1,"string")) $$->type=STRING;}

```

```

        |StructSpecifier {$=$1;}
    ;

    //ExtDecList: 零个或多个 VarDec
    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);}

    ;

    //VarDec: 一个变量的定义
    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

        | VarDec LB Exp RB {$$=mknode(2,ARRAY_DEF,yylineno,$1,$3);} //
任意维度的数组

    ;

    //FuncDec: 函数头
    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

    ;

    //VarList: 形参列表
    VarList: ParamDec {$$=mknode(1,PARAM_LIST,yylineno,$1);}

        | ParamDec COMMA VarList
    {$$=mknode(2,PARAM_LIST,yylineno,$1,$3);}

    ;

    //ParamDec: 一个形参的定义
    ParamDec: Specifier VarDec
    {$$=mknode(2,PARAM_DEC,yylineno,$1,$2);}

    ;

    //CompSt: 函数体、由花括号括起来的语句块, 即复合语句
    CompSt: LC DefList StmList RC
    {$$=mknode(2,COMP_STM,yylineno,$2,$3);}

```



```

;

//StmList: 语句列表
StmList:          {$$=NULL; }
    | Stmt StmList {$$=mknode(2,STM_LIST,yylineno,$1,$2);}
;

//Stmt: 一条语句
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);}
    | BREAK SEMI
{$$=mknode(0,BREAK,yylineno);strcpy($$->type_id,"BREAK");}
    | CONTINUE SEMI
{$$=mknode(0,CONTINUE,yylineno);strcpy($$->type_id,"CONTINUE");}
    | FOR LP FORDEC RP Stmt {$$=mknode(2,FOR,yylineno,$3,$5);}
/* | FOR LP Exp SEMI Exp SEMI Exp RP
{$$=mknode(3,FOR,yylineno,$3,$5,$7);} */
    | SEMI {$$=NULL;} //对应于只有 Stmt 为空句子的情况
;

//For 语句体
FORDEC: Exp SEMI Exp SEMI Exp {$$=mknode(3, FOR_DEC, yylineno, $1, $3,
$5);}
;

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

//Def: 一条变量定义

```

```

Def:    Specifier DecList SEMI {$$=mknode(2,VAR_DEF,yylineno,$1,$2);}
        ;
//DecList:
DecList: Dec {$$=mknode(1,DEC_LIST,yylineno,$1);}
        | Dec COMMA DecList {$$=mknode(2,DEC_LIST,yylineno,$1,$3);}
        ;
//Dec:
Dec:    VarDec {$$=$1;}
        | VarDec ASSIGNOP Exp
{$$=mknode(2,ASSIGNOP,yylineno,$1,$3);strcpy($$->type_id,"ASSIGNOP");}
        ;
//Exp: 一个表达式
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($$->type_id,"DIV");}
        | Exp PLUS ASSIGNOP Exp
{$$=mknode(2,PLUS_ASSIGNOP,yylineno,$1,$4);strcpy($$->type_id,"PLUS_ASSIGN
OP");}

```

```

    | Exp MINUS ASSIGNOP Exp
    { $$=mknode(2,MINUS_ASSIGNOP,yylineno,$1,$4);strcpy($$->type_id,"MINUS_ASSI
GNOP");}

    | Exp STAR ASSIGNOP Exp
    { $$=mknode(2,STAR_ASSIGNOP,yylineno,$1,$4);strcpy($$->type_id,"STAR_ASSIGN
OP");}

    | Exp DIV ASSIGNOP Exp
    { $$=mknode(2,DIV_ASSIGNOP,yylineno,$1,$4);strcpy($$->type_id,"DIV_ASSIGNOP
");}

    | Exp DPLUS
    { $$=mknode(1,EXP_DPLUS,yylineno,$1);strcpy($$->type_id,"EXP_DPLUS");}

    | Exp DMINUS
    { $$=mknode(1,EXP_DMINUS,yylineno,$1);strcpy($$->type_id,"EXP_DMINUS");}

    | DPLUS Exp
    { $$=mknode(1,DPLUS_EXP,yylineno,$2);strcpy($$->type_id,"DPLUS_EXP");}

    | DMINUS Exp
    { $$=mknode(1,DMINUS_EXP,yylineno,$2);strcpy($$->type_id,"DMINUS_EXP");}

    | 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");}

    | ID LP Args RP
    { $$=mknode(1,FUNC_CALL,yylineno,$3);strcpy($$->type_id,$1);}

    | ID LP RP
    { $$=mknode(0,FUNC_CALL,yylineno);strcpy($$->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);}

    if($1[0]=='\\'){ $$->type_char[0]='\\';$$->type_char[1]=$1[1];$$->type=CHAR
; }

```

```

else{ $$->type_char[0]=$1[0]; $$->type=CHAR; }

| STRING
{ $$=mknode(0,STRING,yylineno); strcpy($$->type_string,$1); $$->type=STRING; }

| Exp LB Exp RB { $$=mknode(2,ARRAY_READ,yylineno,$1,$3); }

| Exp DOT ID
{ $$=mknode(2,STRUCT_READ,yylineno,$1,$3); strcpy($$->type_id,$3); }

;

//Args: 实参列表
Args:   Exp COMMA Args   { $$=mknode(2,ARGS,yylineno,$1,$3); }
      | Exp              { $$=mknode(1,ARGS,yylineno,$1); }

;

```

通过使用上述形式，给出所有规则的语义动作，当一个程序使用 LR 分析法完成语法分析后，如果正确则可生成一棵抽象语法树

3.3 符号表管理

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。

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

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。层号是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。

3.4 语义检查

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，名字包括：程序名、过程名、函数名、用户定义类型名、变量名、常量名、枚举值名、标号名等；特性信息包括：上述名字的种类、具体类型、数组维数、函数参数个数、常量数值及目标地址（存储单元偏移地址）等。

语义检查主要完成的是静态语义分析，包括四大类：

（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，那么久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。

（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。

实验中实现的语义分析及实现方式如下：

（1）使用未定义的变量；

在符号表中，从栈顶向栈底方向查层号为 LEV 的符号是否有当前变量，没有则报未定义错误。

（2）调用未定义或未声明的函数；

在符号表中，从栈顶向栈底方向查层号为 LEV 的符号是否有当前函数，没有则报未定义错误。

（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；

在定义时遍历符号表查看是否存在已定义的表项

（4）对非函数名采用函数调用形式；

在符号表中查找当前调用对象 id 并判断其属性是否为 V

（5）对函数名采用非函数调用形式访问；

在符号表中查找当前访问对象 id 并判断其属性是否为 F

(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；

在符号表中查找参数个数判断是否匹配

(7) 函数调用时实参和形参类型不匹配；

检查当前 `type` 和符号表中 `Id` 的 `type` 是否一致

(8) 对非数组变量采用下标变量的形式访问；

碰到数组访问时在符号表中查找当前 `ID` 的 `type` 是否为 “A” 即数组

(9) 数组变量的下标不是整型表达式；

检查当前访问下标或处理后的结果的 `type` 是否为 `INT`

(10) 赋值号左边不是左值表达式；

检查赋值号左边的对象在符号表中是否存在对应的表项

(11) 对非左值表达式进行自增、自减运算；

检查运算对象在符号表中是否存在对应的表项

(12) 类型不匹配；

通过查找符号表的方式比较操作数的 `type`

(13) 函数返回值类型与函数定义的返回值类型不匹配；

查找符号表获取函数的返回值类型后再与当前类型进行比价

(14) 函数没有返回语句（当函数返回值类型不是 `void` 时）；

查找符号表获取函数的返回类型是否为 `void`

(15) `break` 语句不在循环语句或 `switch` 语句中；

定义全局变量 `IN_LOOP`，在进入 `for` 和 `while` 语句中时自增，出循环是自减，当出现 `break` 时判断 `IN_LOOP` 的值是否为 0，为 0 说明不在循环体内部，报错

(16) `continue` 语句不在循环语句中；

定义全局变量 `IN_LOOP`，在进入 `for` 和 `while` 语句中时自增，出循环是自减，当出现 `continue` 时判断 `IN_LOOP` 的值是否为 0，为 0 说明不在循环体内部，报错

3.5 报错功能

分析过程中，为了方便标识错误位置，需要记录分析过程中的当前行号。在 FLEX 中定义了一个内部变量 `yylineno`，当在 FLEX 文件的定义部分加上 `%option yylineno` 后，就可以直接使用这个内部变量了，并且不需要去维护

yylineno的值，在词法分析过程中，每次遇到一个回车，yylineno会自动加一。同时在BISON文件的声明部分加上extern int yylineno，就可以共用yylineno的值。

词法分析过程中，一旦遇到识别不了的单词，需要进行报错。处理方法很简单，在FLEX文件的规则部分，前面是能识别出来的所有单词的正则式，最后一个条就是一个符号”.”表示的正则式，表示不能识别出的单词形式，这时结合变量yylineno给出错误信息即可。

语法报错由BISON文件中的yyerror函数负责完成，需要补充的就是错误定位，在源程序的哪一行有错。为了更准确的给出错误性质，可在BISON文件的辅助定义部分加上%error-verbose。

为了更准确的标识错误的位置，除了行号以外，还需要标识列号。这时可以利用一个表示位置的类型YYLTYPE，BISON中的每一个语法单元（终结符和非终结符）对应一个YYLTYPE类型的位置信息，YYLTYPE定义形式为：

```
typedef struct {  
    int first_line;  
    int first_column;  
    int last_line;  
    int last_column;  
} YYLTYPE;
```

其中first_line和first_column表示该语法单元第一个单词出现的行号和列号，last_line和last_column表示该语法单元最后一个单词出现的行号和列号。为了能正确引用位置信息，需要使用FLEX的内置变量yylloc，yylloc表示当前词法单元所在的位置信息。首先需要在FLEX文件的声明部分加上：

```
int yycolumn=1;  
#define YY_USER_ACTION    yylloc.first_line=yylloc.last_line=yylineno;\n    yylloc.first_column=yycolumn;    yylloc.last_column=yycolumn+yyleng-1;\n    yycolumn+=yyleng;
```

在FLEX文件规则部分的正则式'\n'后面将yycolumn赋值为1，表示一个新的行，列数从1开始。同时在BISON文件的辅助定义部分加上%locations，这样在

BISON中，就可以使用`yylloc.first_line`和`yylloc.first_column`表示当前单词位置的行号和列号，准确地标识错误的位置。同时每条规则后的动作部分通过`@$`、`@1`、`@2`等形式引用该规则各语法单位的位置信息。

编译过程中，待编译的mini-c源程序可能会有多个错误，这时，需要有容错的功能，跳过错误的代码段，继续向后进行语法分析，而不是遇到一个错误就停下来，这个步骤称为**同步**。一次尽可能多地报出源程序的语法错误，减少交互次数，提高编译效率。这时可通过跳过一段源程序代码段到指定的符号，再接着进行语法分析。例如：

Stm \rightarrow error SEMI

表示对语句进行语法分析时，一旦有错，跳过分号（SEMI），继续向后进行语法分析。可在`parser.y`中多处设置这种同步操作，比如对外部定义，可再加上：

ExtDef \rightarrow error SEMI

表示对外部定义进行语法分析时，一旦有错，跳过分号（SEMI），继续向后进行语法分析。多处设置这种同步操作可能会导致使用bison对`parser.y`进行翻译时出现移进/规约的冲突，但不会影响到对mini-c源程序的正常语法分析，所以对这类同步引起的移进/规约冲突可以不理睬。注意，通过同步可能会跳过大量的源程序代码，被跳过的代码中可能还含有其它的语法错误，为了避免大量错误的堆积，可以限制一下同步的次数，到达这个次数时就终止语法分析。

3.6 中间代码生成

中间代码生成的关键任务就是通过遍历AST，将各子树的中间代码链进行拼接。最终在AST的根节点得到能完整描述程序结构的code链表。基于该链表即可完成中间代码的输出以及目标代码的生成。

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，以及每次完成了计算后，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，可以使作用域嵌套，不同的作用域中的变量可以同名，这是在mini-c中，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，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。

.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，利用这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，方便计算变量、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在完成布尔表达式翻译时，表达式值为真假时要转移的程序位置（标号的字符串形式）。

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

3.7 代码优化

在生成 bool 表达式中 and 和 or 相关的中间代码时进行了短路处理，减少不必要的判断。当表达式作为控制语句的条件时，都会在其控制语句结点处给条件表达式子树的根结点属性.Etrue 和.Efalse 赋值，所以在处理表达式结点时，如果属性.Etrue 和.Efalse 的值为空就按基本表达式处理，否则按布尔表达式处理。

```
case AND:
    T->type = INT;
    T->ptr[0]->offset = T->offset;
    Exp(T->ptr[0]);
    T->place = fill_Temp(newTemp(), LEV, T->type, 'T', T->offset);
    T->code = T->ptr[0]->code;

    strcpy(label1, newLabel());
    strcpy(label2, newLabel());

    opn1.kind = ID;
    strcpy(opn1.id, symbolTable.symbols[T->ptr[0]->place].alias);
    opn1.offset = symbolTable.symbols[T->ptr[0]->place].offset;
```

```

    opn1.type = T->ptr[0]->type;

    opn2.kind = INT;
    opn2.const_int = 0;
    result.kind = ID;
    strcpy(result.id, label1);
    op = EQ;
    T->code = merge(2, T->code, genIR(op, opn1, opn2, result));

    T->ptr[1]->offset = T->offset + T->ptr[0]->width;
    Exp(T->ptr[1]);
    T->code = merge(2, T->code, T->ptr[1]->code);

    opn1.kind = ID;
    strcpy(opn1.id, symbolTable.symbols[T->ptr[1]->place].alias);
    opn1.offset = symbolTable.symbols[T->ptr[1]->place].offset;
    opn1.type = T->ptr[1]->type;

    T->code = merge(2, T->code, genIR(op, opn1, opn2, result));

    opn1.kind = INT;
    opn1.const_int = 1;

    result.kind = ID;
    strcpy(result.id, symbolTable.symbols[T->place].alias);
    result.type = T->type;
    result.offset = symbolTable.symbols[T->place].offset;

    T->code = merge(4, T->code, genIR(ASSIGNOP, opn1, opn2, result),
    genGoto(label2), genLabel(label1));

    opn1.kind = INT;
    opn1.const_int = 0;
    opn1.type = INT;

    T->code = merge(3, T->code, genIR(ASSIGNOP, opn1, opn2, result),
    genLabel(label2));
    break;

```

case 0R:

```

    T->type = INT;
    T->ptr[0]->offset = T->offset;
    Exp(T->ptr[0]);
    T->place = fill_Temp(newTemp(), LEV, T->type, 'T', T->offset);

```

```

T->code = T->ptr[0]->code;
strcpy(label1, newLabel());
strcpy(label2, newLabel());

opn1.kind = ID;
strcpy(opn1.id, symbolTable.symbols[T->ptr[0]->place].alias);
opn1.offset = symbolTable.symbols[T->ptr[0]->place].offset;
opn1.type = T->ptr[0]->type;

opn2.kind = INT;
opn2.const_int = 0;
result.kind = ID;
strcpy(result.id, label1);
op = NEQ;
T->code = merge(2, T->code, genIR(op, opn1, opn2, result));

T->ptr[1]->offset = T->offset + T->ptr[0]->width;
Exp(T->ptr[1]);
T->code = merge(2, T->code, T->ptr[1]->code);
opn1.kind = ID;
strcpy(opn1.id, symbolTable.symbols[T->ptr[1]->place].alias);
opn1.offset = symbolTable.symbols[T->ptr[1]->place].offset;
opn1.type = T->ptr[1]->type;

T->code = merge(2, T->code, genIR(op, opn1, opn2, result));
opn1.kind = INT;
opn1.const_int = 0;

result.kind = ID;
strcpy(result.id, symbolTable.symbols[T->place].alias);
result.type = T->type;
result.offset = symbolTable.symbols[T->place].offset;

T->code = merge(4, T->code, genIR(ASSIGNOP, opn1, opn2, result),
genGoto(label2), genLabel(label1));

opn1.kind = INT;
opn1.const_int = 1;
opn1.type = INT;

T->code = merge(3, T->code, genIR(ASSIGNOP, opn1, opn2, result),
genLabel(label2));
break;

```

3.8 汇编代码生成

这部分要完成将 TAC 指令序列转换成目标代码，目标代码选择为 MIPS 汇编指令。最终生成的 MIPS 汇编指令可以在 MARS 上运行。中间代码与 MIPS32 指令对应关系参考 2.6。

实验中选择朴素的寄存器分配方案，目标代码生成时，每当运算操作时，都需要将操作数读入到寄存器中，运算结束后将结果写到对应的单元。由于选择朴素的寄存器分配，只会用到几个寄存器，这里约定操作数使用 \$t1 和 \$t2，运算结果使用 \$t3，翻译的方法如表 3-1 所示。

表 3-1 朴素寄存器分配的翻译

中间代码	MIPS32 指令
$x := \#k$	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)
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	

对于函数调用 X:=CALL f，需要完成开辟活动记录的空间、参数的传递和保存返回地址等，函数调用返回后，需要恢复返回地址，读取函数返回值以及释放活动记录空间。

首先根据符号表中函数定义项得到该函数活动记录的大小，开辟活动记录空间和保存返回地址。根据函数定义中的参数个数 paramnum，即在 X:=CALL f 之前有 paramnum 个 ARG 形式的中间代码，可获得各个实参值所存放的单元，取出后送到形式参数的单元中。之后使用 jal f 转到函数 f 处。

函数调用 CALL f:

```
for (p = h, i = 0; i < symbolTable.symbols[h->opn1.offset].paramnum; i++)
    p = p->prior;

fprintf(fp, "  move $t0,$sp\n");
fprintf(fp, "  addi $sp, $sp, -%d\n",
symbolTable.symbols[h->opn1.offset].offset);
fprintf(fp, "  sw $ra,0($sp)\n");
i = h->opn1.offset + 1;

//参数入栈
while (symbolTable.symbols[i].flag == 'P')
{
    fprintf(fp, "  lw $t1, %d($t0)\n", p->result.offset);
    fprintf(fp, "  move $t3,$t1\n");
    fprintf(fp, "  sw $t3,%d($sp)\n", symbolTable.symbols[i].offset);
    p = p->next;
    i++;
}
```

```
}

//跳转到目标函数处
fprintf(fp, " jal %s\n", h->opn1.id);
fprintf(fp, " lw $ra,0($sp)\n");
fprintf(fp, " addi $sp,$sp,%d\n",
symbolTable.symbols[h->opn1.offset].offset);
fprintf(fp, " sw $v0,%d($sp)\n", h->result.offset);
```

函数执行完后，需要释放活动记录空间和恢复返回地址

```
case RETURN:
    fprintf(fp, " lw $v0,%d($sp)\n", h->result.offset);
    fprintf(fp, " jr $ra\n");
    break;
```

4 系统测试与评价

4.1 测试用例

实验一测试代码如下所示：

```
struct A{
    int x,y;
    float a;
};
int main()
{
    struct A a;
    int i,j,k;
    float x,y;
    char c1[2][3][4];
    i+=-i*++j;
    c[1][2]=10;
    a.b=10.56;
    for(i=1;i<10;i++)
    {
        if (i!=j) break;
        else continue;
    }
    return !(1+a);
}
```

实验二测试代码

```
int f3()
{
    int a,b=0;
    if (a-12.3) {a=b+1.0;continue;}
    else break;
    while ( a || f3()) {while (1) break;continue;}
    for(a=1;a>0 && f3(>0;a++)
    if (a+b==0.0) break;
    break;
    return 1+10.0;
}
```

实验三四测试代码

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

int main()
{
    int m,n,i;
    m = read();
    i=1;
    while(i<=m)
    {
        n = fibo(i);
        write(n);
        i=i+1;
    }
    return 1;
}
```

4.2 正确性测试

实验一的测试结果如图 4-1 所示：


```

结构体定义：(4)
  结构名：A
  局部变量定义：(2)
    类型：int
    变量名：
      ID: x
      变量名：
        ID: y
  局部变量定义：(3)
    类型：float
    变量名：
      ID: a
函数定义：(21)
  类型：int
  函数名：main
  无参函数
复合语句：(21)
  复合语句的变量定义部分：
    局部变量定义：(7)
      定义struct变量：
        结构名：A
        变量名：
          ID: a
      局部变量定义：(8)
        类型：int
        变量名：
          ID: i
          变量名：
            ID: j
            变量名：
              ID: k
      局部变量定义：(9)
        类型：float
        变量名：
          ID: x
          变量名：
            ID: y
      局部变量定义：(10)
        类型：char
        变量名：
          数组：
            第1级数组大小：
              INT: 4
            第2级数组大小：
              INT: 3
            第3级数组大小：
              INT: 2
            数组名：
              ID: c1
复合语句的语句部分：
  表达式语句：(11)
    PLUS_ASSIGNOP
      ID: i
      STAR
      UMINUS
      ID: i
      DPLUS_EXP
      ID: j
  表达式语句：(12)
    ASSIGNOP
      数组访问表达式：
        第1级数组下标：
          INT: 2
        第2级数组下标：
          INT: 1
        数组名：
          ID: c
          INT: 10
  表达式语句：(13)
    ASSIGNOP
      结构体访问表达式：
        ID: a
        结构成员：b
        FLOAT: 10.560000
  循环语句：(14)
    循环条件：
      ASSIGNOP
        ID: i
        INT: 1
    循环体：
      <
        ID: i
        INT: 10
      循环后执行操作：
        EXP_DPLUS
        ID: i
  复合语句：(18)
    复合语句的语句部分：
      条件语句(IF_THEN_ELSE): (17)
        条件：
          !=
          ID: i
          ID: j
        IF子句：(17)
          break语句(16)
        ELSE子句：(17)
          continue语句(17)
  返回语句：(19)
    NOT
    PLUS
      INT: 1
      ID: a

```

图 4-1 实验一结果

实验二的测试结果如图 4-2 所示：

```
./parser test.c
在4行,x 访问数组方式错误
在4行,a 非数组变量, 不能用下标访问
在4行,f5 函数名不能用下标访问
在4行, 运算符左右变量类型不匹配
在5行, 数组维数错误
在5行, 数组下标表达式非法
在6行, 函数返回值类型与函数定义的返回值类型不匹配
```

图 4-2 实验二结果

实验三的测试结果如图 4-3 所示:

FUNCTION fibo :	FUNCTION main :
PARAM v7	temp11 := CALL read
temp1 := #1	v9 := temp11
IF v7 == temp1 GOTO label3	temp12 := #1
GOTO label4	v11 := temp12
LABEL label4 :	LABEL label10 :
temp2 := #2	IF v11 <= v9 GOTO label9
IF v7 == temp2 GOTO label3	GOTO label8
GOTO label2	LABEL label9 :
LABEL label3 :	ARG v11
temp3 := #1	temp13 := CALL fibo
RETURN temp3	v10 := temp13
LABEL label2 :	ARG v10
temp4 := #1	CALL write
temp5 := v7 - temp4	temp15 := #1
ARG temp5	temp16 := v11 + temp15
temp6 := CALL fibo	v11 := temp16
temp7 := #2	GOTO label10
temp8 := v7 - temp7	LABEL label8 :
ARG temp8	temp17 := #1
temp9 := CALL fibo	RETURN temp17
temp10 := temp6 + temp9	LABEL label5 :
RETURN temp10	
LABEL label1 :	

图 4-3 实验三结果

实验四的目标 mips 代码:

```
.data
_Prompt: .asciiz "Enter an integer:  "
_ret: .asciiz "\n"
.globl main
.text
main0:
```

```

jal main
li $v0,10
syscall
read:
    li $v0,4
    la $a0,_Prompt
    syscall
    li $v0,5
    syscall
    jr $ra
write:
    li $v0,1
    syscall
    li $v0,4
    la $a0,_ret
    syscall
    move $v0,$0
    jr $ra

fibo:
    li $t3, 1
    sw $t3, 16($sp)
    lw $t1, 12($sp)
    lw $t2, 16($sp)
    beq $t1,$t2,label3
    j label4
label4:
    li $t3, 2
    sw $t3, 16($sp)
    lw $t1, 12($sp)
    lw $t2, 16($sp)
    beq $t1,$t2,label3
    j label2
label3:
    li $t3, 1

```

```

    sw $t3, 16($sp)
    lw $v0, 16($sp)
    jr $ra
label2:
    li $t3, 1
    sw $t3, 16($sp)
    lw $t1, 12($sp)
    lw $t2, 16($sp)
    sub $t3, $t1, $t2
    sw $t3, 20($sp)
    move $t0, $sp
    addi $sp, $sp, -44
    sw $ra, 0($sp)
    lw $t1, 20($t0)
    move $t3, $t1
    sw $t3, 12($sp)
    jal fibo
    lw $ra, 0($sp)
    addi $sp, $sp, 44
    sw $v0, 24($sp)
    li $t3, 2
    sw $t3, 28($sp)
    lw $t1, 12($sp)
    lw $t2, 28($sp)
    sub $t3, $t1, $t2
    sw $t3, 32($sp)
    move $t0, $sp
    addi $sp, $sp, -44
    sw $ra, 0($sp)
    lw $t1, 32($t0)
    move $t3, $t1
    sw $t3, 12($sp)
    jal fibo
    lw $ra, 0($sp)
    addi $sp, $sp, 44

```

```

    sw $v0,36($sp)
    lw $t1, 24($sp)
    lw $t2, 36($sp)
    add $t3,$t1,$t2
    sw $t3, 40($sp)
    lw $v0,40($sp)
    jr $ra
label1:

main:
    addi $sp, $sp, -32
    addi $sp, $sp, -4
    sw $ra,0($sp)
    jal read
    lw $ra,0($sp)
    addi $sp, $sp, 4
    sw $v0, 24($sp)
    lw $t1, 24($sp)
    move $t3, $t1
    sw $t3, 12($sp)
    li $t3, 1
    sw $t3, 24($sp)
    lw $t1, 24($sp)
    move $t3, $t1
    sw $t3, 20($sp)
label10:
    lw $t1, 20($sp)
    lw $t2, 12($sp)
    ble $t1,$t2,label9
    j label8
label9:
    move $t0,$sp
    addi $sp, $sp, -44
    sw $ra,0($sp)
    lw $t1, 20($t0)

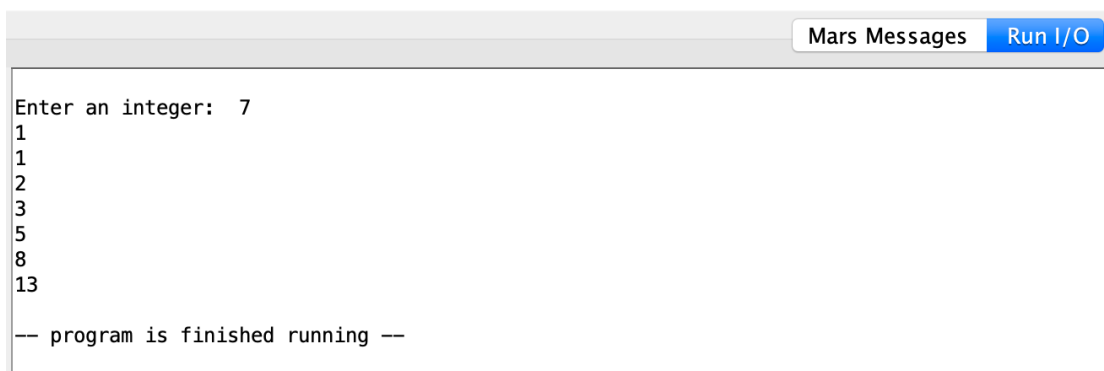
```

```

    move $t3,$t1
    sw $t3,12($sp)
    jal fibo
    lw $ra,0($sp)
    addi $sp,$sp,44
    sw $v0,24($sp)
    lw $t1, 24($sp)
    move $t3, $t1
    sw $t3, 16($sp)
    lw $a0, 16($sp)
    addi $sp, $sp, -4
    sw $ra,0($sp)
    jal write
    lw $ra,0($sp)
    addi $sp, $sp, 4
    li $t3, 1
    sw $t3, 24($sp)
    lw $t1, 20($sp)
    lw $t2, 24($sp)
    add $t3,$t1,$t2
    sw $t3, 28($sp)
    lw $t1, 28($sp)
    move $t3, $t1
    sw $t3, 20($sp)
    j label10
label8:
    li $t3, 1
    sw $t3, 24($sp)
    lw $v0,24($sp)
    jr $ra
label5:

```

MARS 运行结果如图 4-4 所示:



```
Enter an integer: 7
1
1
2
3
5
8
13
-- program is finished running --
```

图 4-4 实验四测试结果

4.3 报错功能测试

实验二代码存在几处错误，相关错误报错结果如图 4-5 所示：

- 在 4 行, x 访问数组方式错误
- 在 4 行, a 非数组变量，不能用下标访问
- 在 4 行, f5 函数名不能用下标访问
- 在 4 行, 运算符左右变量类型不匹配
- 在 5 行, 数组维数错误
- 在 5 行, 数组下标表达式非法
- 在 6 行, 函数返回值类型与函数定义的返回值类型不匹配

图 4-5 实验二报错

4.4 系统的优点

系统在 miniC 的基础上，扩展了单行注释、多行注意、char 类型、string 类型，数组，结构体等多种实现。能够检测多种错误并给出正确的提示信息。能够清晰并且详细的展示编译过程中每一个步骤的结果。

4.5 系统的缺点

结构体仅实现到实验二语义分析阶段，数组实现到实验三中间代码生成阶段，在计算偏移值的时候和预期不同，未能解决，最终数组未实现生成可执行的 mips 代码。

5 实验小结或体会

就我个人而言，整个编译原理的实验过程比较艰辛，花费了大量的时间和精力才顺利完成，收获颇多。

从实验一开始，由于对于整个实验的框架的理解很局限，刚开始做实验一头雾水，反复观看实验指导，消耗了很多时间在理解相关的数据结构和 `lex` 和 `bison` 的写法上，虽然最终仿照实验一固有的写法完成了基本要求，但对于实验本身的理解还停留在一个较浅的水平上；直到完成了实验二以后自己的认知才有了进一步的提升，能够慢慢写出一些个性化的东西来了，后面就实现了对数组和结构的解析。实验三在实验二的基础上进行中间代码的生成，在完成基本目标后，补充了之前做的不好的 `for` 循环实现，实验三是难度最大的一次实验，中间涉及到对于各个符号的偏移量的计算，很容易出错，调试不容易，最终在数组的偏移值计算上尚存一些问题没有解决。实验四由于实验三对于数组偏移量计算的差错，实现了数组的赋值后，在数组的读取上无法定位到正确的内存位置，花费了两天时间没能最后解决掉这个问题，有些许遗憾，最终实现了中间代码转换成 MIP32 的汇编程序，跑出斐波那契程序的正确结果时，还是特别开心的。

本次实验，是对课程内容的很好实践，我了解了编译程序整个的流程，详细的体会了词法分析、语法分析、语义分析、中间代码生成和目标代码生成的整个流程。由此加深对理论课上所讲的概念的理解，同时提高了自己的动手能力，实打实的提高了编程调试能力，收货很多。

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