

**课 程 实 验 报 告**

**课程名称： 编 译 原 理 实 验**

**专业班级： CS1707**

**学 号： U201714785**

**姓 名： 聂鸿勇**

**指导教师： 吴海**

**报告日期： 2020年 6月 30 日**

**计算机科学与技术学院**

**目录**

[1 概述 3](#_Toc44344119)

[2 系统描述 4](#_Toc44344120)

[2.1自定义语言概述 4](#_Toc44344121)

[2.2单词文法与语言文法 4](#_Toc44344122)

[2.3 符号表结构定义 9](#_Toc44344123)

[2.4 错误类型码定义 10](#_Toc44344124)

[2.5中间代码结构定义 11](#_Toc44344125)

[2.6目标代码指令集选择 12](#_Toc44344126)

[3 系统设计与实现 13](#_Toc44344127)

[3.1词法分析器 13](#_Toc44344128)

[3.2语法分析器 14](#_Toc44344129)

[3.3符号表管理 14](#_Toc44344130)

[3.4语义检查 15](#_Toc44344131)

[3.5报错功能 17](#_Toc44344132)

[3.6中间代码生成 17](#_Toc44344133)

[3.7汇编代码生成 23](#_Toc44344134)

[4 系统测试与评价 26](#_Toc44344135)

[4.1测试用例 26](#_Toc44344136)

[4.2正确性测试 29](#_Toc44344137)

[4.3报错功能测试 33](#_Toc44344138)

[4.4系统的优点 33](#_Toc44344139)

[4.5系统的缺点 34](#_Toc44344140)

[5实验小结 35](#_Toc44344141)

[**参考文献** 36](#_Toc44344142)

[**附件：源代码** 37](#_Toc44344143)

# 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。

# 系统描述

## 2.1自定义语言概述

一个简化的C语言的文法，不妨将其称为mini-c，实现了基本类型int、float、char以及数组，能够识别五类单词：关键字（保留字）、运算符、界符、常量和标识符，实现部分静态语义分析，单词文法与语言文法见章节2.2。

## 2.2单词文法与语言文法

**单词文法**：

|  |  |  |
| --- | --- | --- |
| 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]\* |
| {char} | CHAR | ['][^\n]['] |
| {int} | INT | [0-9]+ |
| {float} | FLOAT | ([0-9]\*\.[0-9]+)|([0-9]+\.) |
| “char” | TYPE |  |
| “int” | TYPE |  |
| “float” | TYPE |  |
| return | RETURN |  |
| if | IF |  |
| else | ELSE |  |
| while | WHILE |  |
| for | FOR |  |
| “;” | SEMI |  |
| “,” | COMMA |  |
| ">"|"<"|">="|"<="|"=="|"!=" | RELOP |  |
| “=” | ASSIGNOP |  |
| “+” | PLUS |  |
| “-“ | MINUS |  |
| “\*” | STAR |  |
| “/” | DIV |  |
| “%” | MOD |  |
| ++ | AUTOPLUS |  |
| -- | AUTOMINUS |  |
| += | PLUSASSIGNOP |  |
| -= | MINUSASSIGNOP |  |
| \*= | STARASSIGNOP |  |
| /= | DIVASSIGNOP |  |
| &= | MODASSIGNOP |  |
| “&&” | AND |  |
| “||” | OR |  |
| “!” | NOT |  |
| “(” | LP |  |
| “)” | RP |  |
| "[" | LB |  |
| "]" | RB |  |
| “{” | LC |  |
| “}” | RC |  |
| "//"[^\n]\* | //单行注释 |  |
| "/\*" {BEGIN(COMMENT);}  <COMMENT>"\*/" {BEGIN(INITIAL);}  <COMMENT>([^\*]|\n)+|. | /\*\*/多行注释 |  |

**语言文法**：

|  |
| --- |
| %%  program: ExtDefList { display($1,0);} //显示语法树,语义分析  ;  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: 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  | error SEMI {$$=NULL;}  ;  Specifier: TYPE {$$=mknode(0,TYPE,yylineno);strcpy($$->type\_id,$1);$$->type=!strcmp($1,"int")?INT:FLOAT;}  ;  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: 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 Arraylist {$$=mknode(1,ARRAY\_LIST,yylineno,$2);strcpy($$->type\_id,$1);}  ;  //数组声明  Arraylist: LB INT RB {$$=mknode(0,ARRAY\_LIST,yylineno);$$->type\_int=$2;}  | LB INT RB Arraylist {$$=mknode(1,ARRAY\_LIST,yylineno,$4);$$->type\_int=$2;}  ;  //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$$->type\_id  ;  //参数列表  VarList: ParamDec {$$=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 {$$=mknode(2,COMP\_STM,yylineno,$2,$3);}  ;  //语句列表，由0个或多个语句stmt组成  StmList: {$$=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 {$$=mknode(2,WHILE,yylineno,$3,$5);}  | CONTINUE SEMI { $$ = mknode(0, CONTINUE, yylineno); strcpy($$->type\_id, "CONTINUE"); }  | BREAK SEMI { $$ = mknode(0, BREAK, yylineno); strcpy($$->type\_id, "BREAK"); }  | FOR LP ForDec RP Stmt {$$=mknode(2,FOR,yylineno,$3,$5);} //FOR循环识别 for(FORDEC):Stmt  ;  //增加各个情况for循环识别  ForDec: Exp SEMI Exp SEMI Exp {$$=mknode(3, FOR\_DEC,yylineno,$1,$3,$5);}  | Exp SEMI SEMI Exp {$$=mknode(3, FOR\_DEC,yylineno,$1,NULL,$4);}  | Exp SEMI Exp SEMI {$$=mknode(3, FOR\_DEC,yylineno,$1,$3,NULL);}  | Exp SEMI SEMI {$$=mknode(3, FOR\_DEC,yylineno,$1,NULL,NULL);}  | SEMI Exp SEMI Exp {$$=mknode(3, FOR\_DEC,yylineno,NULL,$2,$4);}  | SEMI Exp SEMI {$$=mknode(3, FOR\_DEC,yylineno,NULL,$2,NULL);}  | SEMI SEMI Exp {$$=mknode(3, FOR\_DEC,yylineno,NULL,NULL,$3);}  | SEMI SEMI {$$=mknode(3, FOR\_DEC, yylineno, NULL, NULL, NULL);}  ;  //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);}  ;  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 PLUSASSIGNOP Exp {$$=mknode(2,PLUSASSIGNOP,yylineno,$1,$3);strcpy($$->type\_id, "PLUSASSIGNOP");}  | Exp MINUSASSIGNOP Exp {$$=mknode(2,MINUSASSIGNOP,yylineno,$1,$3);strcpy($$->type\_id, "MINUSASSIGNOP");}  | Exp STARASSIGNOP Exp {$$=mknode(2, STARASSIGNOP,yylineno,$1,$3);strcpy($$->type\_id,"STARASSIGNOP");}  | Exp DIVASSIGNOP Exp {$$=mknode(2,DIVASSIGNOP,yylineno,$1,$3);strcpy($$->type\_id,"DIVASSIGNOP");}  | Exp MODASSIGNOP Exp {$$=mknode(2,MODASSIGNOP,yylineno,$1,$3);strcpy($$->type\_id, "MODASSIGNOP");}    | 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 AUTOPLUS {$$=mknode(1,AUTOPLUS,yylineno,$1);strcpy($$->type\_id, "RPAUTOPLUS");}  | Exp AUTOMINUS {$$=mknode(1,AUTOMINUS,yylineno,$1);strcpy($$->type\_id,"RPAUTOMINUS");}  | AUTOPLUS Exp {$$=mknode(1,AUTOPLUS,yylineno,$2);strcpy($$->type\_id, "LPAUTOPLUS");}  | AUTOMINUS Exp {$$=mknode(1,AUTOMINUS,yylineno,$2);strcpy($$->type\_id, "LPAUTOMINUS");}    | 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($$->type\_id,"NOT");}  | DPLUS Exp {$$=mknode(1,DPLUS,yylineno,$2);strcpy($$->type\_id,"DPLUS");}  | Exp DPLUS {$$=mknode(1,DPLUS,yylineno,$1);strcpy($$->type\_id,"DPLUS");}  | ID LP Args RP {$$=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);$$->type\_char=$1;$$->type=CHAR;}  | LB Args RB {$$=$2;}  ;  Args: Exp COMMA Args {$$=mknode(2,ARGS,yylineno,$1,$3);}  | Exp {$$=mknode(1,ARGS,yylineno,$1);}  ;    %% |

## 2.3 符号表结构定义

定义一个符号表中一个符号的结构体如下：

|  |
| --- |
| 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，  //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...  int array[ARRAY\_LEN]; //记录数组维度  }; |

整个符号表的定义如下所示：

|  |
| --- |
| //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; |

## 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。

（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；

（11）结构成员不存在；

（10）对非结构变量采用成员选择运算符“.”；

（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；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；

（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；

错误定义（10）、（11）、（14）、（17）没有实现。

## 2.5中间代码结构定义

采用三地址代码TAC作为中间语言，中间语言代码的定义如表2-1所示。

**表2-1 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 |  | y |
| 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 | x |
| RETURN x | 返回语句 | RETURN |  |  | x |
| ARG x | 传实参x | ARG |  |  | x |
| x:=CALL f | 调用函数(有返回值) | CALL | f |  | x |
| CALL f | 调用函数(无返回值) | CALL | f |  |  |
| PARAM x | 函数形参 | PARAM |  |  | x |

三地址中间代码TAC是一个4元组，逻辑上包含（op、opn1、opn2、result），其中op表示操作类型说明，opn1和opn2表示2个操作数，result表示运算结果。后续还需要根据TAC序列生成目标代码，所以设计其存储结构时，每一部分要考虑目标代码生成时所需要的信息。

## 2.6目标代码指令集选择

目标语言可选定MIPS32指令序列， TAC指令和MIPS32指令的对应关系如表2-2所示。其中reg(x)表示变量x所分配的寄存器。

**表2-2 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 |

# 

# 系统设计与实现

## 3.1词法分析器

使用flex工具编写lex文件，对指定的高级语言程序进行词法分析，按lex文件格式编写定义部分、规则部分以及用户自定义代码部分。

(1)二元式展示:在lex.l文件中在每个识别字符后的相应动作添加输出语句例：printf("(%s, INT)\n", yytext)，在这里为了控制是否输出二元式可以添加if判断语句。

(2)单词分类：保留字、操作符、界符、标识符、常数、注释，对于不同的识别符需要给出其正则表达式，并定义识别后做出相应的动作，由parser.y中%token定义的标识生成的parser.tab.h用于返回识别类型，例如：return PLUS;。

(3)保留字识别；对于保留字需要识别准确，对于其识别时的规则只能够通过列举的方式来保证保留字的正确识别，例如if、else、for等，识别的定义直接为

"if" {if(V) printf("(%s, IF)\n", yytext);return IF;}

"else" {if(V) printf("(%s, ELSE)\n", yytext);return ELSE;}

"for" {if(V) printf("(%s, FOR)\n", yytext);return FOR;}

(4)操作符识别：操作符包含基本的算术运算符、逻辑运算符以及新添的+=、-=、\*=、/=、%=；部分识别定义如下：

"/=" {if(V) printf("(%s, DIVASSIGNOP)\n", yytext);return DIVASSIGNOP;}

"&&" {if(V) printf("(%s, AND)\n", yytext);return AND;}

(5)界符识别：界符包含: "{}"、"[]"、"()"、","、";"；其定义如下：

"[" {if(V) printf("(%s, LB)\n", yytext);return LB;}

"]" {if(V) printf("(%s, RB)\n", yytext);return RB;}

(6)标识符识别：标识符以字母开头，由字母和数字共同组成的字段，其正则表达式定义为：id [A-Za-z][A-Za-z0-9]\*；识别规则定义为：

{id} {if(V)printf("(%s,ID)\n", yytext);strcpy(yylval.type\_id, yytext);

(7)常数识别：给出类型int、float、char的识别正则表达式，然后给出相应的识别规则做出动作，char类型定义如下:

char ['][^\n][']

{char} { yylval.type\_float=atof(yytext); return CHAR;}

(8)注释的剔除：1、识别//开头的语句，识别语句后不做任何操作跳过该行；2、识别/\*---\*/类型注释，识别以/\*开头、循环直到以\*/结尾的语句，不做任何操作直接跳过。

## 3.2语法分析器

同Flex源代码类似，Bison源代码也分为三个部分，其作用与Flex源代码大致相同，其分为定义部分、规则部分、用户函数部分。

(1)表达式求值：给出不同的表达式规则定义，根据每一条规则的定义进一步将识别出来的左右的表达式进行递归处理，构造相应节点。

(2)说明语句；全局变量、局部变量的声明，分别对全局变量规约规则ExtDef以及局部变量规约规则VAR\_DEF进行了不同的识别操作，并且在ast输出语法树时，对两种进行了区分。

(3) 赋值语句：还没有进入语义分析阶段，所以没有进行进一步的条件判断，在识别给定的符合条件的赋值语句后，构造相应的的ASSIGNOP节点，将两端的EXP标识节点作为该赋值节点的子节点。

(4)分支语句:if…else…then、if…then…,在parser.y中给出语句的识别规则，对于分支语句分别构造IF\_THEN、IF\_THEN\_ELSE的节点，根据识别语句，从中拿出分支条件节点以及复合语句节点作为该分支语句的子节点。

(5)循环语句：循环语句包括while、for；在这里举例介绍for循环语句的实现：首先声明需要使用到的非终结符以及节点类型枚举常量，因为for循环中含有比较特殊的条件语句识别，所以需要额外对该类型语句进行识别规则ForDec的定义，给出的定义包含了for循环条件有可能出现的所有情况，接下来就与要定义识别for循环语句，在识别for循环语句的规则中嵌套了ForDec，在构造相应FOR节点的时候需要把该条件语句节点以及循环体中的符合语句节点当作子节点。

## 3.3符号表管理

实验指导书说：语义分析这部分的一个非常重要的工作就是符号表的管理，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。所谓“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，所谓“特性信息”包括：上述名字的种类、具体类型、维数（如果语言支持数组）、函数参数个数、常量数值及目标地址（存储单元偏移地址）等。

顺序表。本实验指导采用这种方式管理符号表。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。

本实验中，为了方便测试程序、观察运行结果，事先默认了2个函数read和write，程序中可以直接使用，所以需要首先将其登记到符号表中。

符号表对之后的功能实现具有很重要的作用。

## 3.4语义检查

在编译过程中，使用数据结构为顺序表的符号表来记录源程序中各种名字的特性信息。在文件semantic.c中定义符号表操作以及静态语义检查，在遍历过程中，访问到了说明部分的结点时，在符号表中添加新的内容；访问到执行语句部分时，根据访问的变量（或函数）名称查询符号表，并分析其静态语义的正确性。

在文件semantic.c中当对抽象语法树的先根遍历，按display的控制结构修改完成符号表管理和语义检查和TAC生成（语句部分）过程中遇到的是不同列表即可调用对应函数判断即将可能出现的多种静态语义错误，给出函数定义ext\_var\_list处理变量列表，给出函数定义match\_param处理参数列表；给出函数定义boolExp处理布尔表达式；给出函数定义Exp处理基本表达式。

语义分析是建立在符号表之上的，每次语义错误的判断都需要对符号表进行插入或者查询操作，在遍历过程中，访问到了说明部分的结点时，在符号表中添加新的内容；访问到执行语句部分时，根据访问的变量（或函数）名称查询符号表，并分析其静态语义的正确性。

例ext\_var\_list函数定义如下：

|  |
| --- |
| //处理变量列表  void ext\_var\_list(struct ASTNode \*T)  {  int rtn, num = 1;  switch (T->kind)  {  case EXT\_DEC\_LIST:  T->ptr[0]->type = T->type; //将类型属性向下传递变量结点  T->ptr[0]->offset = T->offset; //外部变量的偏移量向下传递  T->ptr[1]->type = T->type; //将类型属性向下传递变量结点  //T->ptr[1]->offset = T->offset + T->width; //外部变量的偏移量向下传递  if (T->ptr[0]->kind = ARRAY\_DF)  ;  //T->ptr[1]->offset = T->offset + T->width \* calArrayNums(T->ptr[0]->ptr[0]);  else  T->ptr[1]->offset = T->offset + T->width;  T->ptr[1]->width = T->width;  ext\_var\_list(T->ptr[0]);  ext\_var\_list(T->ptr[1]);  T->num = T->ptr[1]->num + T->ptr[0]->num;  break;  case ID:  rtn = fillSymbolTable(T->type\_id, newAlias(), LEV, T->type, 'V', T->offset); //最后一个变量名  if (rtn == -1)  semantic\_error(T->pos, T->type\_id, "变量重复定义");  else  {  T->place = rtn;  T->num = 1;  }  break;  case ARRAY\_DF:  //数组定义  rtn = fillSymbolTable(T->type\_id, newAlias(), LEV, T->type, 'A', T->offset); //最后一个变量名  if (rtn == -1)  {  semantic\_error(T->pos, T->type\_id, "变量重复定义");  break;  }  struct ASTNode \*T0;  T0 = T->ptr[0];  while (T0)  {  if (T0->ptr[0] && T0->ptr[0]->type\_int <= 0 || T0->ptr[0]->type != INT)  {  semantic\_error(T->pos, T->type\_id, "数组下标不能为负值或0");  return;  }  T0 = T0->ptr[1];  }  T->place = rtn;  //T->num = calArrayNums(T->ptr[0]);  break;  }  } |

## 3.5报错功能

实验指导书说：

在使用FLEX进行词法分析、Bison进行语法分析的过程中，要求具有**报错和容错**功能。一旦有词法、语法错误，需要准确、及时地进行报错，给出错误位置以及错误性质。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。

在词法分析过程中，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”.”表示地正则式，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。

在静态语义分析过程中，通过给出的判断查找出不符合条件的语句，输出当前行号以及错误类型等错误信息。

## 3.6中间代码生成

**1．opn结构体定义**

|  |
| --- |
| 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  }; |

**2．codenode结构体定义**

|  |
| --- |
| typedef struct codenode { //三地址TAC代码结点,采用单链表存放中间语言代码  int op;  struct opn opn1, opn2, result;  struct codenode\* next, \* prior;  } codenode; |

**3．ASTNode结构体定义**

|  |
| --- |
| 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  }; |

实验指导书说：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）。

按照给定的语句类结点翻译模式、基本表达式类结点翻译模式、控制语句布尔表达式语句结点翻译模式、其它类结点翻译模式如下：

**表3-1 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 |

**表3-2 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 | ti=newtemp, ti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： ti = 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 | ti=newtemp, ti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 || ti=T1.alias OP T2.alias |
| UMINUS  T1 操作数子树 | ti=newtemp, ti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 || ti=- T1.alias |
| RELOP 关系运算符  T1 第一操作数子树  T2 第二操作数子树 | ti=newtemp, ti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  || ti=0 || goto label2 || label1： || ti=1 || label2： |
| AND  T1 第一操作数子树  T2 第二操作数子树 | ti=newtemp, ti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 || ti=T1.alias \* T2.alias  || if ti==0 goto label1 || ti=1 || label1： |
| OR  T1 第一操作数子树  T2 第二操作数子树 | ti=newtemp, ti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 || ti=0  || if T1.alias ==0 goto label1 || ti=1 || goto label2 || label1：  || if T2.alias ==0 goto label2 || ti=1 || label2： |
| NOT  T1 操作数子树 | ti=newtemp, ti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  || ti=0 || goto label2 || label1：|| ti=1 || label2： |
| FUNC\_CALL  T1 实参列表子树 | ti=newtemp, ti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 || ti=CALL 函数名 |

**表3-3 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 | ti=newtemp, ti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 || ti=T1.alias OP T2.alias  if (T.true!=””)  T.code = T.code || if ti!=0 goto T.Etrue || goto T.Efalse |
| UMINUS  T1 操作数子树 | ti=newtemp, ti入口赋值给当前结点place  访问T的所有子树后：  T.code = T1.code || ti=- T1.alias  if (T.true!=””)  T.code=T.code || if ti!=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==””) ti=newtemp, ti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  || ti=0 || goto label2 || label1：|| ti=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  T2 第二操作数子树 | if (T.Etrue==””) 按基本表达式处理  else T1.Etrue=T2.Etrue= T.Etrue,T1.Efalse= newLabel,  T2..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 | ti=newtemp, ti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 ti!=0 goto T.Etrue || goto T.Efalse |

**表3-4 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 |

根据上述介绍的中间代码翻译模式，对程序进行处理，生成的中间代码序列。

## 3.7汇编代码生成

汇编代码的生成需要依靠的是实验三中生成中间代码时生成的双向链表，通过给定的朴素寄存器分配的翻译方式，遍历链表逐个节点翻译成为最终的目标代码并将目标代码写入文件object.s中。

实验指导书说：

当选择朴素的寄存器分配方案后，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，翻译的方法见2表3-2所示。

**表3-2 朴素寄存器分配的翻译**

|  |  |
| --- | --- |
| **中间代码** | **MIPS32指令** |
| Label x | x: |
| 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) |
| GOTO x | j x |
| 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 | jal f  move reg(x),$v0 |

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。活动记录的空间布局没有一个统一的标准，可根据自己的理解保存好数据，并能正确使用即可。

通常，使用4个寄存器完成参数的传递，多余4个的参数使用活动记录空间，这里做了简单处理，所有参数都使用活动记录空间。具体步骤：

1. 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。思考一下main函数的活动记录如何处理？
2. 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。

（3） 使用jal f 转到函数f处

（4） 释放活动记录空间和恢复返回地址。

（5） 使用sw $v0, X的偏移量($sp) 获取返回值送到X的存储单元中。

# 系统测试与评价

## 4.1测试用例

实验一

|  |
| --- |
| int a;//声明int  float b;//声明float  char c;//声明char  int d[2];  /\*  \*main函数定义  \*/  int main(){  int a = 2;  float b = 3.0;  int i;  int v[2][3]=[[1,2,3],[4,5,6]];  /\*  \*if-else语句测试  \*/  if(a>b){  a = b;  }else{  b = a;  }  /\*  \*while循环语句测试  \*/  while(a <= 5){  int k = 0;  }  /\*  \*for循环语句测试  \*/  for(i = 1; i < 10; i++){  if(i = 5)  break;  else  continue;  }  /\*  \*赋值语句测试  \*/  a = a + b;  a += 1;  a -= b;  a \*= 1;  a /= b;  a %= 1;  return 0;  } |

实验二

|  |
| --- |
| int a,b,c;  float m,n;  char o;  int d[5];  float f[2];  char k[2];  int fibo(int a)  {  int a;//测试重复定义  g = 0;//调用未声明变量  m();//测试对非函数名采用函数调用形式  if (a == 1 || a == 2) return 1;  return fibo(a-1)+fibo(a-2);  }  int main()  {  int m,n,i;  float g;  g[1];//测试非数组变量采用下标变量的形式访问  m = read();  i = 1;  i = 'a';//测试类型不匹配  0 = 0;//测试赋值号左边不是左值表达式  0++;//测试非左值表达式进行自增、自减运算  0--;//测试非左值表达式进行自增、自减运算  i = fibo(i);  i = fibo(g);//测试参数类型不匹配  i = fibo();//测试参数太少  i = fibo(m,n);//测试参数太多  break;//测试break语句不在循环语句或switch语句中  continue;//测试continue语句不在循环语句中  while(i<=m)  {  n = fibo(i);  write(n);  i=i+1;  break;  continue;  }  return g;//测试返回值类型不匹配  } |

实验三

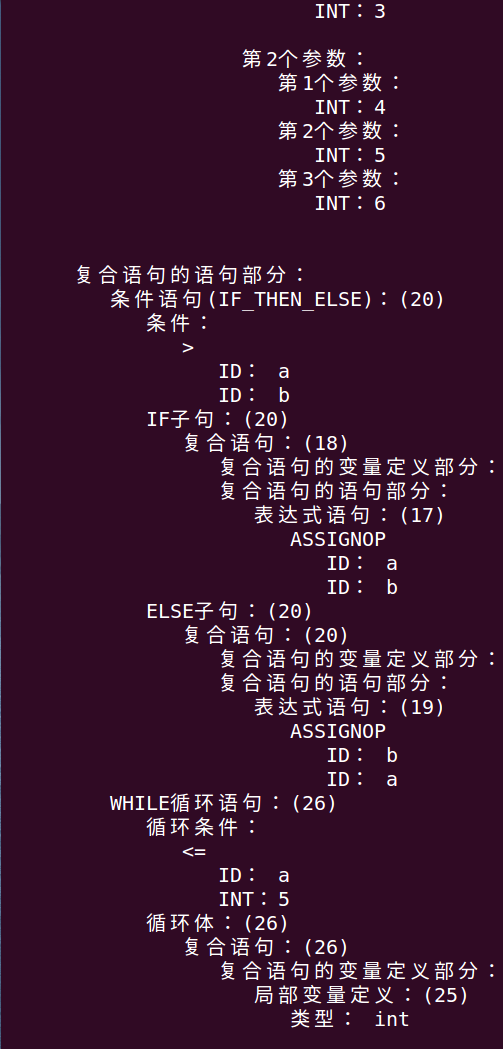
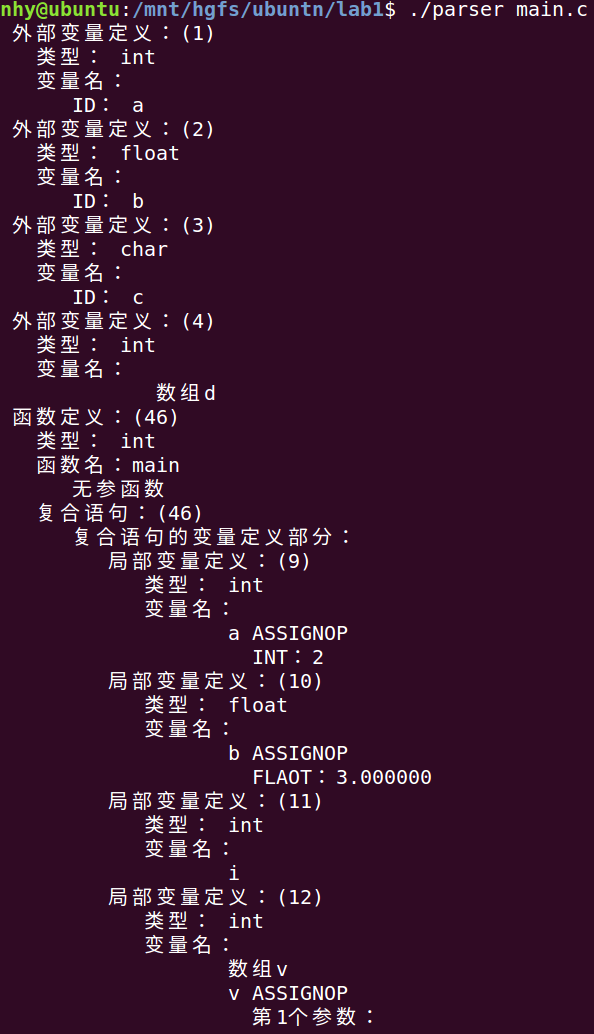
|  |
| --- |
| int a,b,c;  float m,n;  int fibo(int a)  {  if (a == 1 || a == 2) return 1;  return fibo(a-1)+fibo(a-2);  }  int main()  {  int m,n,i;  m = read();  i = 1;  while(i<=m)  {  n = fibo(i);  write(n);  i=i+1;  }  return 1;  } |

实验四

|  |
| --- |
| 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

**实验一测试结果，如**图 4—1**：**



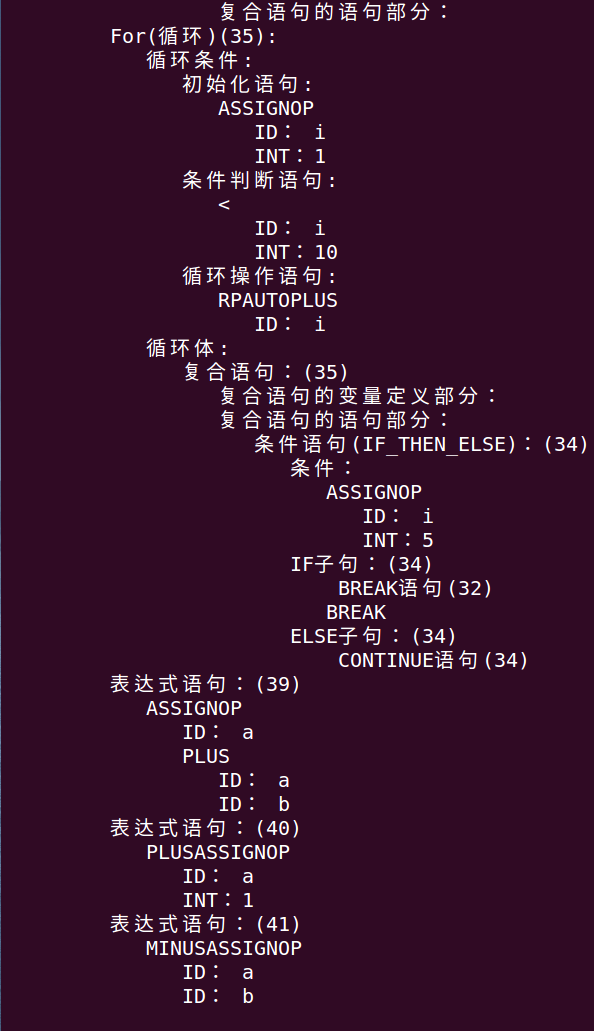


图 4—1 语法树输出结果

总结：实验一中增添了基本类型char、数组以及for、while、break、continue的实现，所以测试代码较长，如上图所示，经过对照测试代码可知输出的语法树是正确的，实验一功能实现正常。

**实验二：**

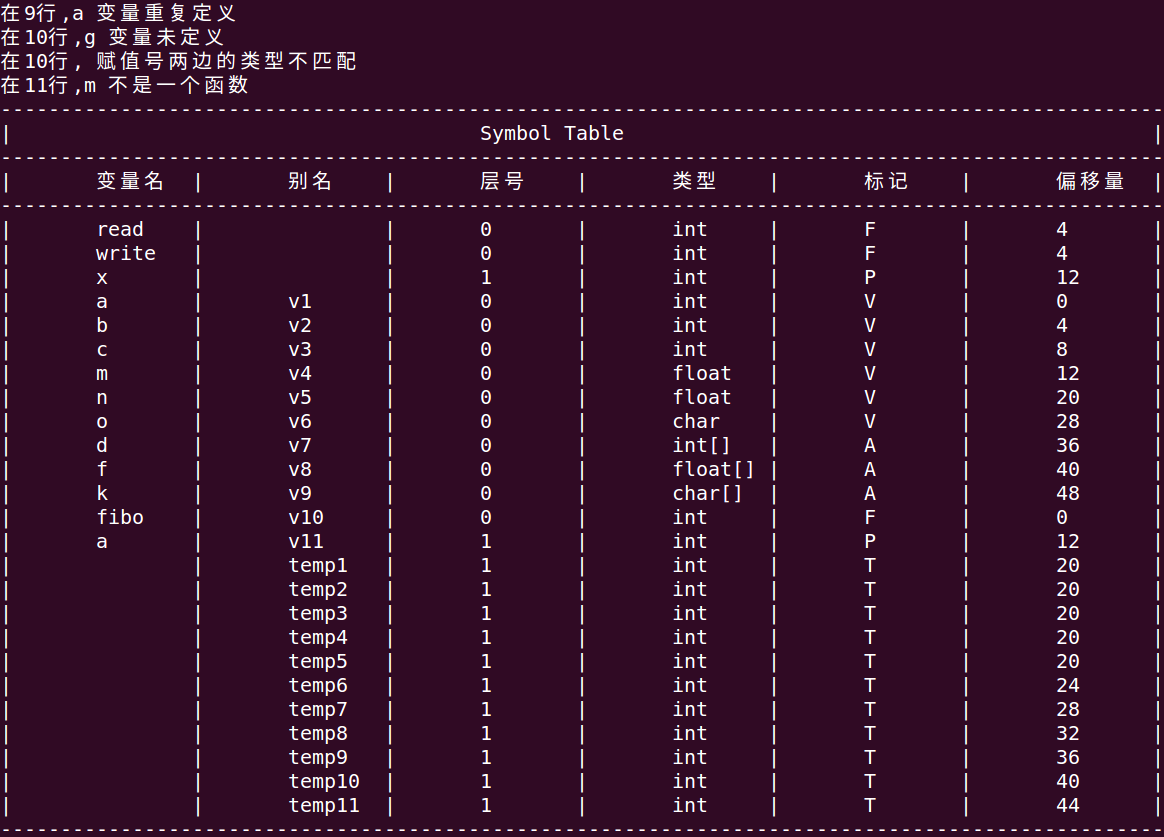


图 4—2 符号表输出1

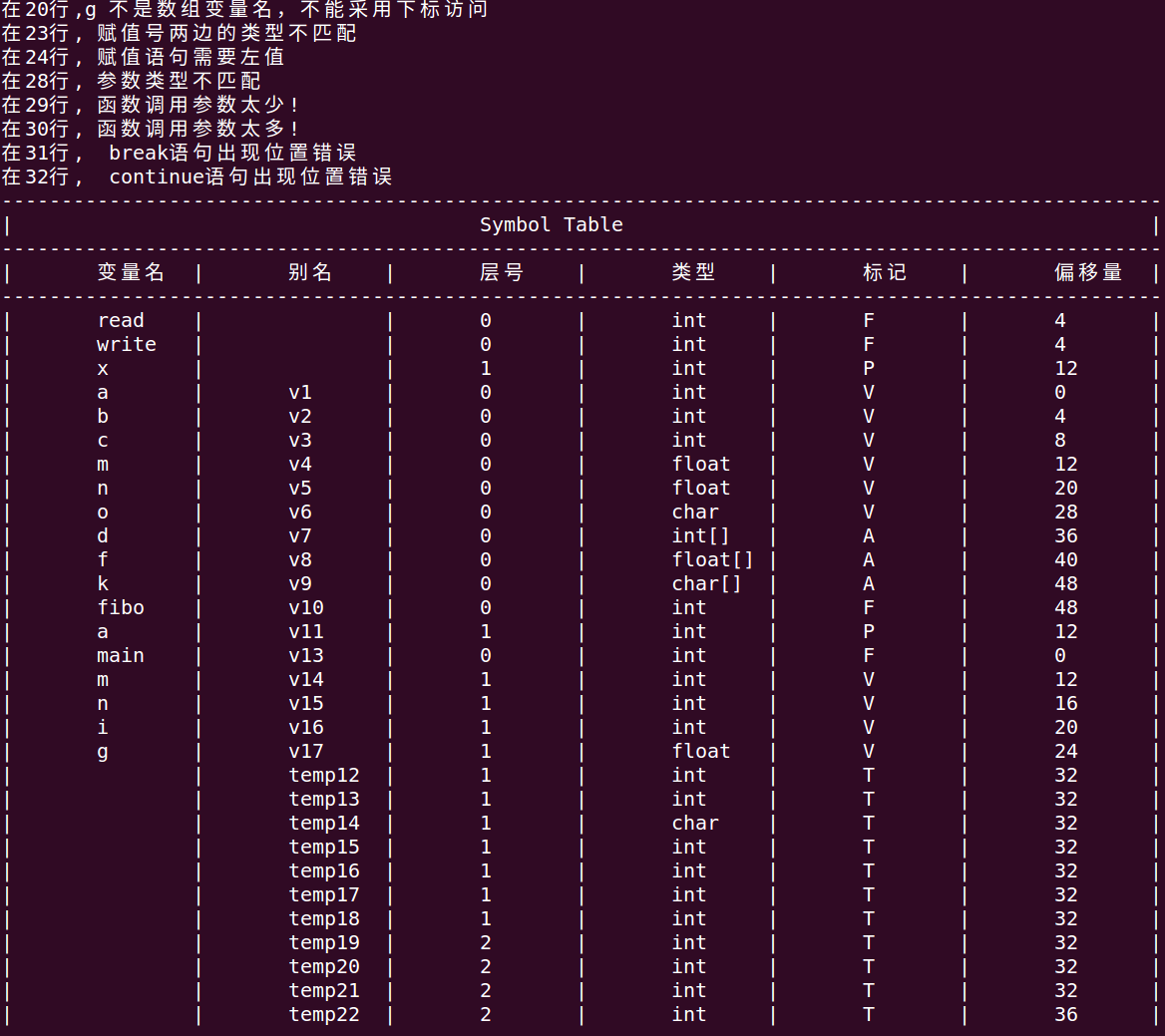


图 4—3符号表输出2

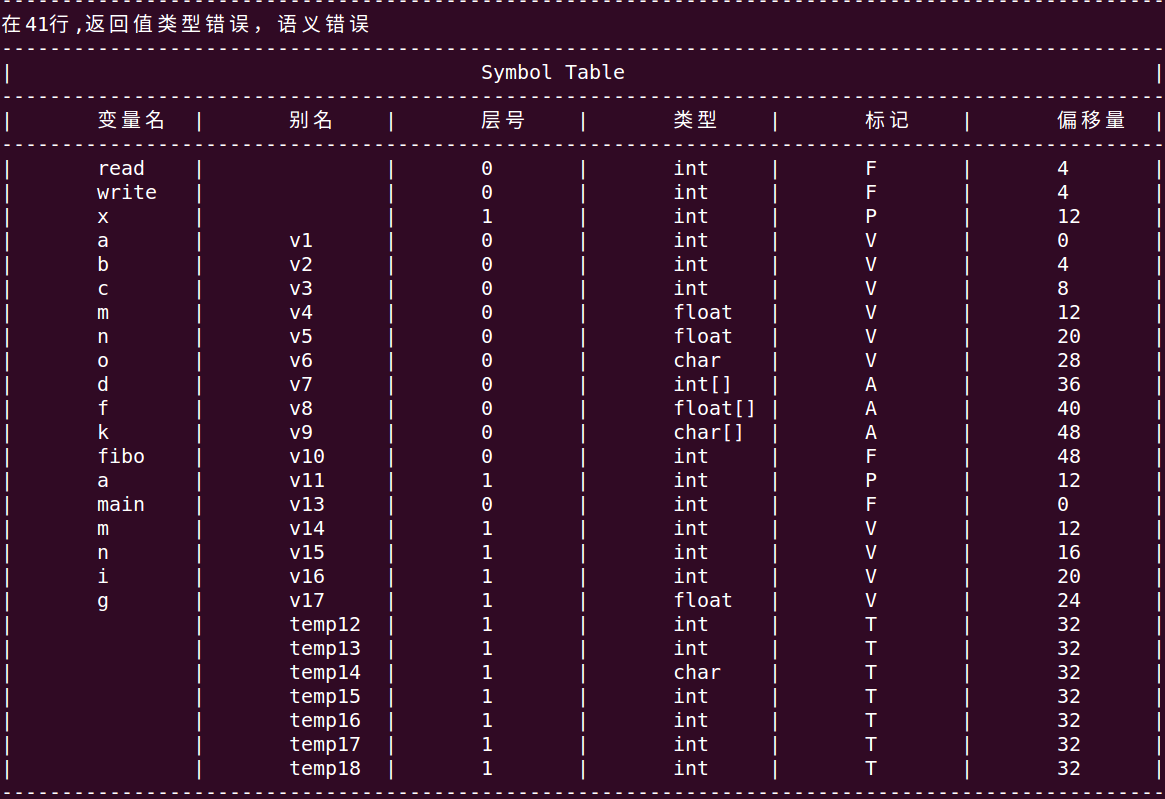


图 4—4符号表输出3

实验二的测试代码中在给定的错误语义出给出了注释，对照如下图生成符号表以及错误信息显示，可知，成功建立了正确的符号表以及对语义分析进行了正确的识别。

**实验三：**

实验三中给出的是标准的测试样例，能够直接将生成的结果与标准输出进行对比，生成结果如下所示，通过linux的diff函数对比之后可以发现生成的结果与标准输出相同，可知，成功生成了样例的中间代码，生成中间代码功能实现正常。

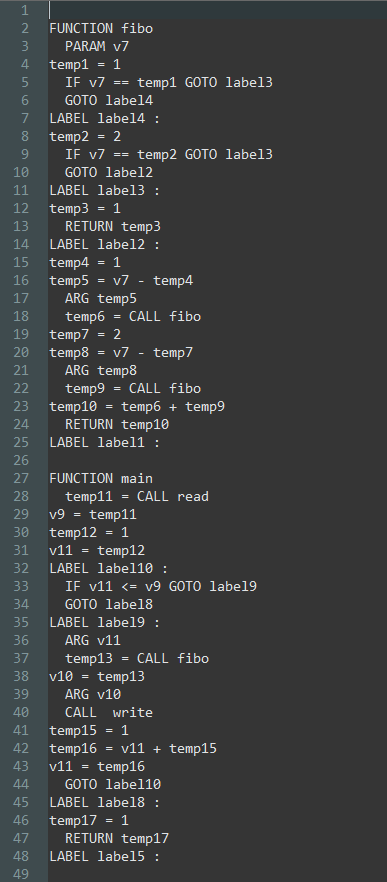


图 4—5 中间代码

**实验四：**

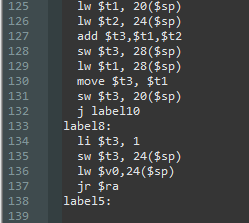
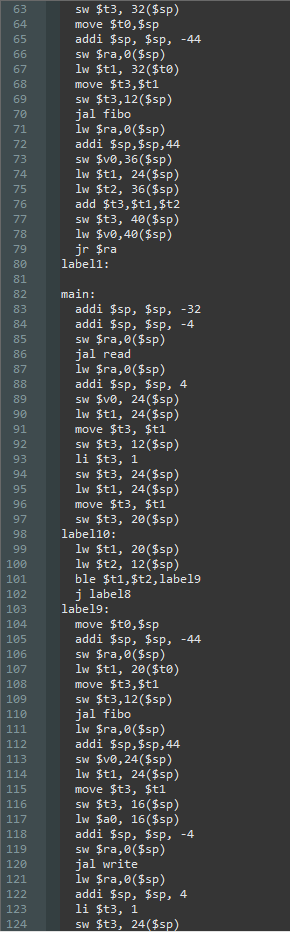
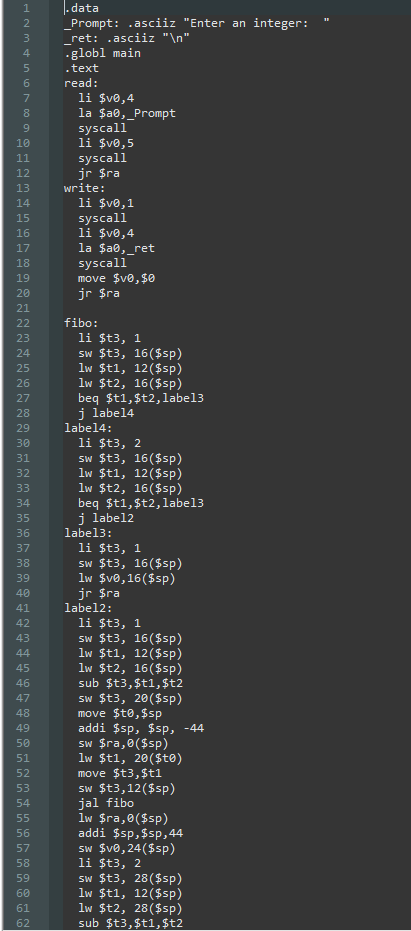


图 4—6 目标代码

将上面生成的object.c文件用qtspim打开进行测试，可得结果如下所示：

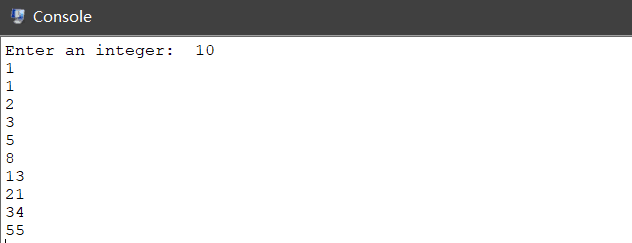


图 4—7 QTSpim测试结果

对比标准测试样例可知该程序的标准输出应该是斐波那契数列，可知目标代码生成成功，生成目标代码功能实现正常。

## 4.3报错功能测试

实验三中给出静态语义分析时报错信息，如图 4—8 报错功能测试：

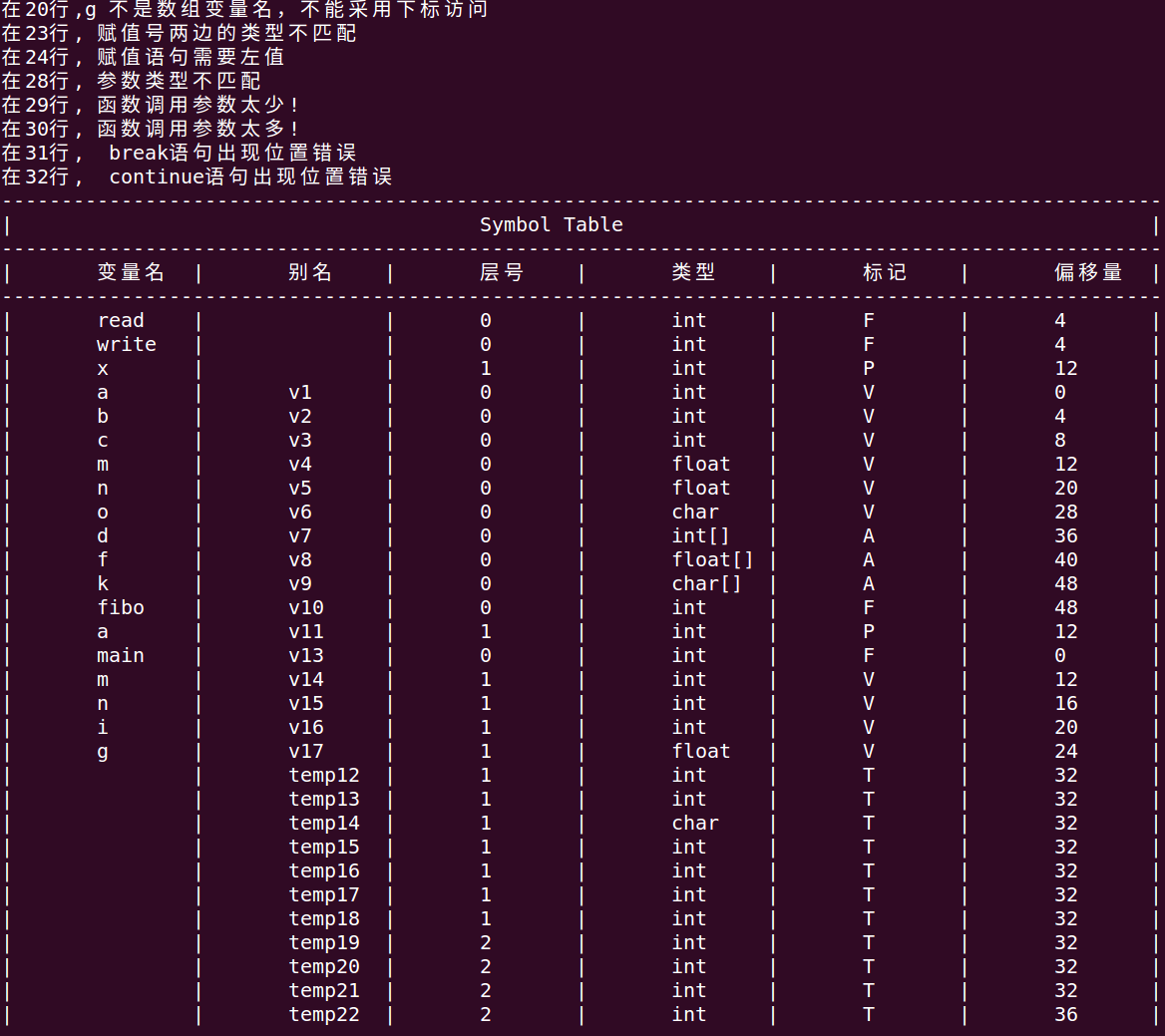


图 4—8 报错功能测试

## 4.4系统的优点

该编译器在对代码进行编译的过程中对各部分输出信息实现了可控化，能够随意的控制其打印出某一部分的编译过程的信息，实现了mini-C基本的词法、语法的定义以及大部分静态语义分析的处理，对出现的错误能够给出明确的报错信息。

## 4.5系统的缺点

目前只实现了最基本的简单操作，在前两次实验中还实现了数组等结构的定义以及识别，但是在后面两次实验中由于实现这些定义需要大量比较繁琐的规则定义以及处理，所以并没有实现数组、结构体、联合等高级的数据结构和操作。

# 5实验小结

通过这四次实验，简单实现了一个mini-C编译器，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，实验主要任务有：

（1）、利用lex工具定义词法分析的正则表达式，实现词法分析功能

（2）、利用bison编写语法分析部分，通过定义声明部分、辅助定义部分、规则部分实现对需要实现的语法规则的定义，能够完成语法分析功能并在错误发生时，准确定位错误的位置。

（3）、实现打印语法树，在语法分析的过程中建立语法树，通过先序遍历的方式打印语法树信息。

（4）、遍历生成符号表，利用符号表对语句进行语义分析，对相应的错误信息进行定位报错，能够打印出错误信息以及每个阶段的符号表信息。

（5）、在进行语义分析的函数当中穿插生成双向链表记录语句信息，在语义分析的过程中进行中间代码的生成。

（6）、目标代码的生成。

该编译原理实验的难度较大，但是给出了比较详细的指导书，在实验当中对我们的帮助比较大。通过实验加深了我们对理论课上所讲的概念的理解，同时提高了逻辑思维以及编程能力。

**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

**附件：源代码**