

**课 程 实 验 报 告**

**课程名称： 编 译 技 术 实 验**

**专业班级： 软工2202班**

**学 号： U202217216**

**姓 名： 郑德凯**

**指导教师： 胡雯蔷**

**报告日期： 2024年 11月 20 日**

**软件学院**

目录

[学 号： U202217216 1](#_Toc20800)

[一、语言定义与特点 3](#_Toc12177)

[二、词法、语法分析 4](#_Toc9395)

[（一）、词法分析 4](#_Toc17911)

[1.概述 4](#_Toc3425)

[2.单词文法描述 4](#_Toc28152)

[（二）、语法分析 5](#_Toc15967)

[1.概述 5](#_Toc29329)

[2.语法分析描述 5](#_Toc12796)

[（三）、抽象语法树 6](#_Toc15144)

[三、语义分析 6](#_Toc30469)

[（一）、符号表 6](#_Toc29756)

[（二） 、支持的静态语义错误 7](#_Toc1377)

[13. 越界访问数组：访问数组时索引超出了数组的有效范围。 7](#_Toc9735)

[16. 函数重复定义：同一个函数被定义了多个版本（重名函数）。 7](#_Toc15494)

[18. 数组重复定义：同一个数组在同一作用域内被定义多次。 7](#_Toc29097)

[（三）语义分析（semantic\_Analysis） 7](#_Toc653)

[1.char类型的补充 7](#_Toc12392)

[2.局部变量数组类型的补充 8](#_Toc28915)

[3. 数组维度的分析 8](#_Toc19954)

[4.对BREAK和CONTINUE的支持 9](#_Toc9283)

[5.全局变量处理（ext\_var\_list） 10](#_Toc15827)

[6. 表达式的处理 ( Exp ) 10](#_Toc11845)

[四、中间代码生成 11](#_Toc11645)

[（一）数组处理 11](#_Toc26207)

[（二）break、continue处理 11](#_Toc24457)

[（三）for处理 11](#_Toc29262)

[（四）复合赋值运算 12](#_Toc16932)

[（五）自增自减运算 13](#_Toc29030)

[五、目标代码生成 13](#_Toc8643)

[（一）、运行环境 13](#_Toc18159)

[（二）、输出目标代码（objectCode） 14](#_Toc8850)

[1.添加对char类型的支持 14](#_Toc9313)

[2.添加对数组的支持 14](#_Toc15931)

[（三） 、代码 14](#_Toc14988)

[六、最终效果展示 19](#_Toc12528)

[（一）、代码目录 19](#_Toc28908)

[（二）、最终效果 20](#_Toc22509)

[1.编译运行 20](#_Toc15549)

[2.测试代码 20](#_Toc21698)

[3.运行效果 21](#_Toc15754)

[七、总结 24](#_Toc9247)

[八、参考文献及网址 25](#_Toc13126)

# 一、语言定义与特点

本次课程设计的任务是实现一个C语言子集的编译器。我将其命名为K语言。参考教程中给出的实例，同时我还进行了功能扩展，新增了以下内容：支持char类型，支持int、float、char三种类型的数组，增加了对注释的识别能力，加入了自增自减运算、复合赋值运算，同时还引入了continue语句和break语句的处理功能。 有K语言的文法如下：

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 [ INT ] | ID [ INT ] [ INT ] // 支持数组类型

FunDec → 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

| continue ; // 新增continue语句

| break ; // 新增break语句

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

| ID

| INT

| FLOAT

| CHAR // 支持char类型

| ( Exp )

| - Exp

| ! Exp

| Exp++ // 支持自增运算

| Exp-- // 支持自减运算

| ID ( Args )

| ID ( )

Args → Exp , Args | Exp

Comment → "//" .\* "\n" // 单行注释

| "/\*" .\* "\*/" // 多行注释

# 二、词法、语法分析

## （一）、词法分析

### 1.概述

首先在环境中安装flex文件，按照flex要求的格式，编写lex.l文件，通过**FLEX lex.l**指令，得到词法分析源程序，通过yylex()来进行词法分析，每识别出一个单词，将该单词的信息传递给语法分析程序。

对于lex.l文件的编写，主要在于通过使用正则表达式实现有穷状态机，每当读取一个单词时，在有穷状态机中识别出对应的类型，并执行对应的语句。

### 2.单词文法描述

在lex.l文件中，通过 **正则表达式 {动作}** 的形式，实现上述功能，除去示例给出的内容，添加的代码如下。

|  |  |  |
| --- | --- | --- |
| 单词符号说明 | 单词种类码 | 正则表达式 |
| {char} | CHAR | (\'.\') |
| "char" | TYPE |  |
| "break" | BREAK |  |
| "continue" | CONTINUE |  |
| "+=" | ADDASSIGNOP |  |
| "-=" | SUBASSIGNOP |  |
| "++" | DPLUS |  |
| "--" | DMINUS |  |
| //注释 |  | "//"(.\*) |
| /\*\*/注释 |  | "/\*"[^(\*/)]\*"\*/" |
| "[" | LS |  |
| "]" | RS |  |

## （二）、语法分析

### 1.概述

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。首先在环境中安装BISON，按照bison的要求，编写parser.y文件，通过**bison -d parser.y**指令，得到语法分析源程序Parser.tab.c。

二者联合在一起完成词法与语法分析时，要求统一单词的种类编码，这时可将各个单词在parser.y中逐个以标识符的形式，通过%token罗列出来。在Parser.y文件中，这些标识符在语法规则部分，作为语法规则的终结符；同时用Bison编译后，可生成一个文件Parser.tab.h，该文件中将这些标识符定义为枚举常量，每一个就对应一个（类）单词，这些枚举常量提供给Lex.l使用，每当识别出一类单词时，就可返回对应的种类码（枚举常量）。

### 2.语法分析描述

在bison文件 prologue的第二部分（Declarations 段）中，主要是对文法的终结符和非终结符做一些相关声明。这些声明主要有如下一些：%token，%left，%right，%nonassoc，%union，%type，%start。

除去示例给出的内容，添加的代码如下：

%type Arraylist

%token <type\_char> CHAR

%token DPLUS DMINUS LS RS

%token BREAK CONTINUE

%token ARR\_DEC ARR\_LIST ARR\_END ARR\_EXP

%left ASSIGNOP ADDASSIGNOP SUBASSIGNOP

%right DMINUS DPLUS

%right UMINUS NOT DPLUS\_PREFIX DMINUS\_PREFIX

%left DPLUS\_SUFFIX DMINUS\_SUFFIX

%left LP RP LC RC LS RS

在Bison语法中，语法规则可以有一个由C语句组成的动作。每次解析器识别出该规则的匹配项时，就会执行操作。

除去示例给出的内容，添加的代码如下：

Specifier: TYPE {$$=(ASTNode \*)malloc(sizeof(ASTNode)); $$->kind=TYPE;

$$->pos=yylineno; strcpy($$->type\_id,$1);

if(!strcmp($1,"int"))

$$->type=INT;

else if(!strcmp($1,"float"))

$$->type=FLOAT;

else if(!strcmp($1,"char"))

$$->type=CHAR;

}

VarDec: | ID Arraylist {$$=(ASTNode\*)malloc(sizeof(ASTNode));$$->kind=ARR\_DEC;strcpy($$->type\_id,$1);$$->Dec=$2;$$->pos=yylineno;}

Arraylist: LS Exp RS {$$=(ASTNode\*)malloc(sizeof(ASTNode));$$->kind=ARR\_END;$$->Exp1=$2;$$->pos=yylineno;}

| LS Exp RS Arraylist {$$=(ASTNode\*)malloc(sizeof(ASTNode));$$->kind=ARR\_LIST;$$->Exp1=$2;$$->pos=yylineno;$$->Exp2=$4;}

Stm ： BREAK SEMI {$$=(ASTNode \*)malloc(sizeof(ASTNode));$$->kind=BREAK;

$$->pos=yylineno; strcpy($$->type\_id,"BREAK");}

| CONTINUE SEMI {$$=(ASTNode\*)malloc(sizeof(ASTNode));$$->kind=CONTINUE;

$$->pos=yylineno; strcpy($$->type\_id,"CONTINUE");}

Exp ： ID Arraylist {$$=(ASTNode \*)malloc(sizeof(ASTNode)); $$->kind=ARR\_EXP; $$->pos=yylineno; $$->Exp1=$2;strcpy($$->type\_id,$1);}

/\*其余Exp表达式，例如自增自减，复合赋值等与其他二元或一元运算符大同小异，此处忽略不写\*/

## （三）、抽象语法树

主要添加了ARR\_DEC、BREAK、CONTINUE、ARR\_LIST、ARR\_END、ADDASSIGNOP、SUBASSIGNOP、DMINUS\_PREFIX、DMINUS\_SUFFIX、DPLUS\_PREFIX、DMINUS\_SUFFIX、ARR\_EXP的情况下的输出，以及其他所有情况下的对char类型和数组类型的支持。

# 三、语义分析

## （一）、符号表

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。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。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。符号表的作用在于对下述的静态语义错误分析进行支持，包括查找变量、定义变量、访问函数等等。

本次课设的符号表采用顺序表实现。

## 、支持的静态语义错误

1. **变量重复定义**：同名变量在同一作用域内被定义多次。
2. **数组定义维数需要用整型**：数组维度必须使用整型常量或变量。
3. **数组未定义**：尝试使用未定义或未声明的数组。
4. **赋值运算左右类型不匹配**：赋值操作符两侧的类型不一致，无法进行赋值。
5. **参数类型不匹配**：函数调用时传入的参数类型与函数定义时要求的类型不符。
6. **参数太多**：函数调用时提供的参数超过函数定义时要求的参数数量。
7. **参数太少**：函数调用时提供的参数少于函数定义时要求的参数数量。
8. **不能对函数名赋值**：函数名不能作为左值进行赋值操作。
9. **数组名，类型不匹配**：数组的类型和定义时声明的类型不匹配。
10. **赋值语句需要左值**：赋值操作的左侧必须是一个有效的可赋值目标（左值）。
11. **不可参与运算的左值**：某些左值（如常量或临时变量）不能参与算术运算。
12. **错误的自增自减变量**：自增（++）或自减（--）操作符应用于非法变量（如常量或表达式）。
13. **越界访问数组**：访问数组时索引超出了数组的有效范围。
14. **错误调用非函数的变量名**：尝试以函数形式调用一个非函数类型的变量。
15. **错误调用非数组的变量名**：尝试以数组形式访问一个非数组类型的变量。
16. **函数重复定义**：同一个函数被定义了多个版本（重名函数）。
17. **参数名重复定义**：函数内部或函数参数中定义了重复的参数名。
18. **数组重复定义**：同一个数组在同一作用域内被定义多次。
19. **break只能在循环中使用**：break语句只能用于退出循环结构。
20. **continue只能在循环中使用**：continue语句只能在循环中使用，跳过当前循环的剩余部分。
21. **返回值类型错误**：函数的返回值类型与函数声明时指定的返回类型不匹配。

## （三）语义分析（semantic\_Analysis）

### 1.char类型的补充

由于附录所给出的代码不包含char类型，因此附录中的代码多处使用三目运算符 : ? 来判断非INT即FLOAT的关系，对于这类代码，需要通过再次添加一次三目运算符( : ? ( : ? ))。包括计算字符宽度、判断类型等部分，都需要补充上char类型，由于涉及到的case太多，此处不详细列出。

### 2.局部变量数组类型的补充

附录所给代码中，VAR\_DEF语句仅支持ID和ASSIGNOP，即声明变量语句和声明时就初始化的语句，并不支持数组的声明，此处必须要填上对于数组类型的声明。

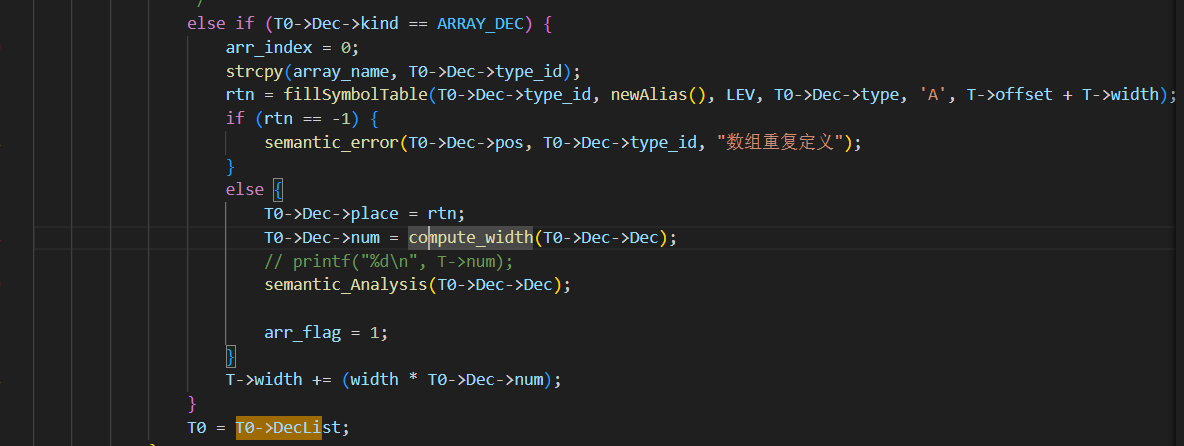


图 1 局部变量数组的补充

### 3. 数组维度的分析

我使用的数组分析的语法是，首先判断是否是数组，其次再处理数组的维度，在上述情况中只处理了是数组的情况，没有对数组的维度做进一步处理，为了判断数组维度是否合法，以及后续的偏移量计算，必须添加这一部分。



图 2 数组维度分析

### 4.对BREAK和CONTINUE的支持

通过在编译器中生成中间代码实现了 `break` 和 `continue` 的功能。`break` 使用 `genGoto(break\_label)` 生成跳转到 `break\_label` 的中间代码，`continue` 则生成跳转到 `continue\_label` 的代码，然后通过 `merge` 将这些跳转指令与当前节点已有的代码 `T->code` 合并，从而实现控制流的正确跳转。关键在于维护 `break\_label` 和 `continue\_label`，确保它们指向正确的位置。

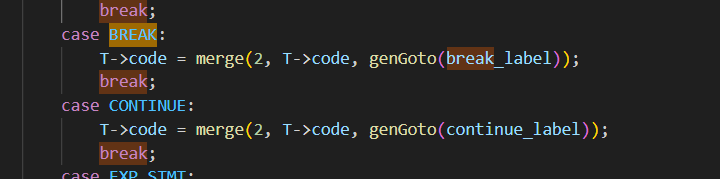


图 3BREAK和CONTINUE的处理

### 5.全局变量处理（ext\_var\_list）



图4 全局变量处理

添加了对数组的支持。

### 6. 表达式的处理 ( Exp )

同第1点，满足对CHAR类型的支持。

同时，需要单独列出一个case char: 具体代码类似于float，作为接受char的输入。

对于ASSIGNOP，需要处理左值或者右值是数组的情况，是数组时，在符号表中需要记下当前的偏移位置。

对于ADDASSIGNOP和SUBASSIGNOP，等同于上述情况，需要处理某一个表达式为数组时的情况，代码也是类似的。

对于DPLUS和DMINUS的情况，对于数组，只做到了语义分析，对于表达式，做到了目标代码生成。

对于PLUS、MINUS、STAR、DIV的情况，同样需要处理其中一个表达式为数组的情况。

对于FUNC\_CALL，需要添加传入参数是数组的情况。

最后，需要单独处理数组的表达式，即case ARR\_EXP。

# 四、中间代码生成

## （一）数组处理

对于涉及到数组的操作，把操作数的ID从数组的ID改为了数组ID + OFFSET + 偏移量，如果是变量作为下标，则偏移量使用变量的别名，如果是INT作为下标，则直接计算出偏移量。这样就可以将数组的处理归结为对表达式的处理。

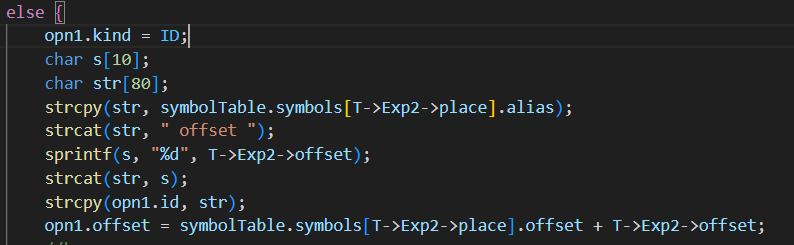


图 5 数组处理

## （二）break、continue处理

对于循环中的break和continue，break语句跳转到while和for条件为FALSE的下一条语句，而continue则跳转到循环体的下一个语句。

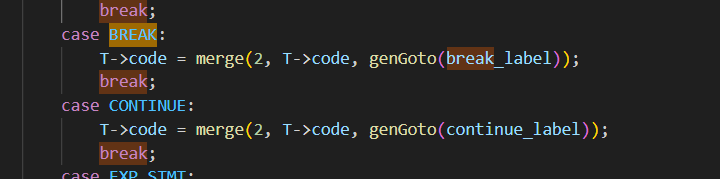


图 6 BREAK和CONTINUE的支持

## （三）for处理

参照案例中给出的while循环，写出对应的for循环。

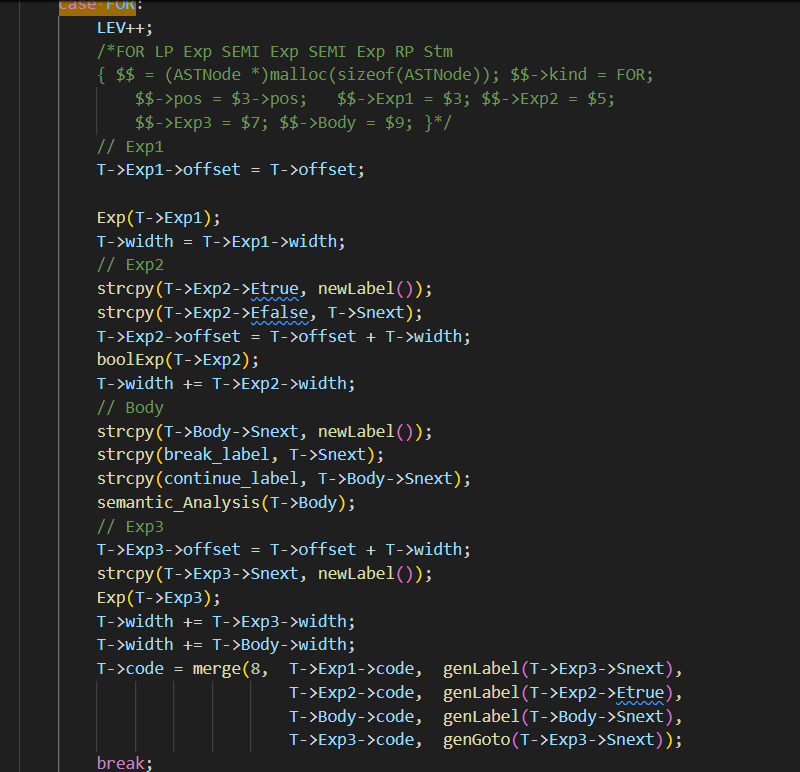


图 7 FOR的处理

## （四）复合赋值运算

复合赋值运算的本质是其中一个操作数和结果操作数是同一个，即opn1和result指向的是同一个变量。在处理时可以参照case PLUS。

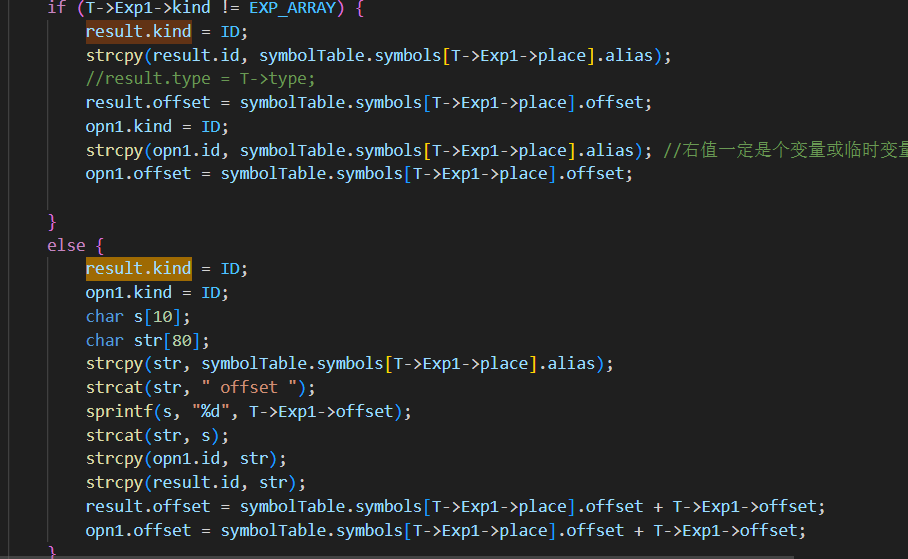


图 8 复合赋值运算

## （五）自增自减运算

自增自减运算的计算方法设计的较为复杂。首先是T1=OP1+OP2,OP1=1,OP2=0，T1是临时变量，OP1和OP2是操作数，然后是T2=X1+T1,T1=1，T2是临时变量，X1是要自增的变量，最后是X1=T2+OP2,OP2=0，从而实现X1=X1+1的操作。

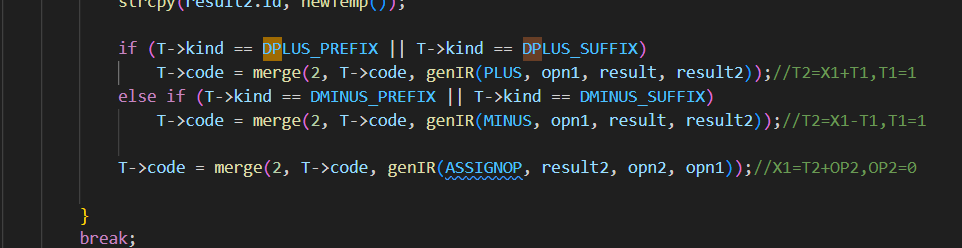


图 9自增自减运算

# 五、目标代码生成

## （一）、运行环境

我选用了MARS（MIPS Assembler and Runtime Simulator），MARS是密苏里州立大学开发的一个轻量级的交互式开发环境（IDE），它集成了汇编器和运行时模拟器，允许用户编写、汇编并运行 MIPS 汇编代码。

## （二）、输出目标代码（objectCode）

### 1.添加对char类型的支持

仿照int和float类型的代码即可。

### 2.添加对数组的支持

由于时间关系，只完成了对于下标是INT型的数组的支持，对于下标的是变量的数组，没有实现对应的代码。

通过前面的中间代码生成，将数组的名字取出，找到在符号表中的位置，根据全局变量和局部变量的不同，输出不同的代码。

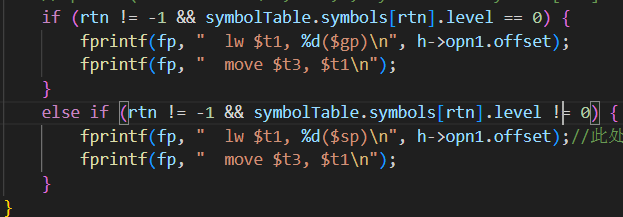


图 10目标代码生成 数组支持

## 、代码

#include "def.h"

int searchSymbolTableByAlias(char \*alias)

{

    int i;

    for (i = symbolTable.index - 1; i >= 0; i--)

        if (!strcmp(symbolTable.symbols[i].alias, alias))

            return i;

    return -1;

}

//输出目标代码

void objectCode(struct codenode \*head)

{

    char opnstr1[32],opnstr2[32],resultstr[32];

    struct codenode \*h=head,\*p;

    int i;

    FILE \*fp;

    fp=fopen("object.s","w");

    fprintf(fp,".data\n");

    fprintf(fp,"\_Prompt: .asciiz \"Enter an integer:  \"\n");

    fprintf(fp,"\_ret: .asciiz \"\\n\"\n");

    fprintf(fp,".globl main\n");

    fprintf(fp,".text\n");

    fprintf(fp, "j main\n");//跳转到main函数

    fprintf(fp,"read:\n");

    fprintf(fp,"  li $v0,4\n");

    fprintf(fp,"  la $a0,\_Prompt\n");

    fprintf(fp,"  syscall\n");

    fprintf(fp,"  li $v0,5\n");

    fprintf(fp,"  syscall\n");

    fprintf(fp,"  jr $ra\n");

    fprintf(fp,"write:\n");

    fprintf(fp,"  li $v0,1\n");

    fprintf(fp,"  syscall\n");

    fprintf(fp,"  li $v0,4\n");

    fprintf(fp,"  la $a0,\_ret\n");

    fprintf(fp,"  syscall\n");

    fprintf(fp,"  move $v0,$0\n");

    fprintf(fp,"  jr $ra\n");

    do { //采用朴素寄存器分配

        switch (h->op) {

            case ASSIGNOP:

                        /\*if (h->opn1.kind==INT)

                            fprintf(fp, "  li $t3, %d\n", h->opn1.const\_int);

                        else {

                            fprintf(fp, "  lw $t1, %d($sp)\n", h->opn1.offset);

                            fprintf(fp, "  move $t3, $t1\n");

                            }

                        fprintf(fp, "  sw $t3, %d($sp)\n", h->result.offset);

                        break;\*/

                if (h->opn1.kind == INT) {

                    fprintf(fp, "  li $t3, %d\n", h->opn1.const\_int);

                }

                else if (h->opn1.kind == FLOAT) {

                    fprintf(fp, "  li $t3, %f\n", h->opn1.const\_float);

                }

                else if (h->opn1.kind == CHAR) {

                    fprintf(fp, "  li $t3, %c\n", h->opn1.const\_char);

                }

                else {

                    int rtn;

                    char arrName[10] = { '\0' };

                    if (strstr(h->opn1.id, " offset ") != NULL) {

                        for (int i = 0; i < strlen(h->result.id); i++) {

                            if (h->result.id[i] != ' ') {

                                arrName[i] = h->result.id[i];

                            }

                        }

                        rtn = searchSymbolTableByAlias(arrName);

                    }

                    else

                        rtn = searchSymbolTableByAlias(h->opn1.id);

                    // printf("%d %s %d %d\n", rtn, , symbolTable.symbols[rtn].level, LEV);

                    if (rtn != -1 && symbolTable.symbols[rtn].level == 0) {

                        fprintf(fp, "  lw $t1, %d($gp)\n", h->opn1.offset);

                        fprintf(fp, "  move $t3, $t1\n");

                    }

                    else if (rtn != -1 && symbolTable.symbols[rtn].level != 0) {

                        fprintf(fp, "  lw $t1, %d($sp)\n", h->opn1.offset);//此处应该计算offset，并应该由MIPS去计算，而不是由C

                        fprintf(fp, "  move $t3, $t1\n");

                    }

                }

                int rtn;

                char arrName[10] = { '\0' };

                if (strstr(h->result.id, " offset ") != NULL) {

                    for (int i = 0; i < strlen(h->result.id); i++) {

                        if (h->result.id[i] != ' ') {

                            arrName[i] = h->result.id[i];

                        }

                    }

                    rtn = searchSymbolTableByAlias(arrName);

                }

                else

                    rtn = searchSymbolTableByAlias(h->result.id);

                // fprintf(fp, "  sw $t3, %d($sp)\n", h->result.offset);

                // printf("%s %d %d\n", h->result.id, symbolTable.symbols[rtn].level, LEV);

                if (rtn != -1 && symbolTable.symbols[rtn].level == 0) {

                    fprintf(fp, "  sw $t3, %d($gp)\n", h->result.offset);//此处应该计算offset，并应该由MIPS去计算，而不是由C

                }

                else {

                    fprintf(fp, "  sw $t3, %d($sp)\n", h->result.offset);

                }

                break;

            case PLUS:

            case MINUS:

            case STAR:

            case DIV:

                       fprintf(fp, "  lw $t1, %d($sp)\n", h->opn1.offset);

                       fprintf(fp, "  lw $t2, %d($sp)\n", h->opn2.offset);

                       if (h->op==PLUS)       fprintf(fp, "  add $t3,$t1,$t2\n");

                       else if (h->op==MINUS) fprintf(fp, "  sub $t3,$t1,$t2\n");

                            else if (h->op==STAR)  fprintf(fp, "  mul $t3,$t1,$t2\n");

                                 else  {fprintf(fp, "  div $t1, $t2\n");

                                        fprintf(fp, "  mflo $t3\n");

                                        }

                        fprintf(fp, "  sw $t3, %d($sp)\n", h->result.offset);

                        break;

            case FUNCTION:

                        fprintf(fp, "\n%s:\n", h->result.id);

                        if (!strcmp(h->result.id,"main")) //特殊处理main

                            fprintf(fp, "  addi $sp, $sp, -%d\n",symbolTable.symbols[h->result.offset].offset);

                        break;

            case PARAM: //直接跳到后面一条

                        break;

            case LABEL: fprintf(fp, "%s:\n", h->result.id);

                        break;

            case GOTO:  fprintf(fp, "  j %s\n", h->result.id);

                        break;

            case JLE:

            case JLT:

            case JGE:

            case JGT:

            case EQ:

            case NEQ:

                        fprintf(fp, "  lw $t1, %d($sp)\n", h->opn1.offset);

                        fprintf(fp, "  lw $t2, %d($sp)\n", h->opn2.offset);

                        if (h->op==JLE) fprintf(fp, "  ble $t1,$t2,%s\n", h->result.id);

                        else if (h->op==JLT) fprintf(fp, "  blt $t1,$t2,%s\n", h->result.id);

                        else if (h->op==JGE) fprintf(fp, "  bge $t1,$t2,%s\n", h->result.id);

                        else if (h->op==JGT) fprintf(fp, "  bgt $t1,$t2,%s\n", h->result.id);

                        else if (h->op==EQ)  fprintf(fp, "  beq $t1,$t2,%s\n", h->result.id);

                        else                 fprintf(fp, "  bne $t1,$t2,%s\n", h->result.id);

                        break;

            case ARG:   //直接跳到后面一条,直到函数调用，回头反查参数。

                        break;

            case CALL:  if (!strcmp(h->opn1.id,"read")){ //特殊处理read

                            fprintf(fp, "  addi $sp, $sp, -4\n");

                            fprintf(fp, "  sw $ra,0($sp)\n"); //保留返回地址

                            fprintf(fp, "  jal read\n"); //保留返回地址

                            fprintf(fp, "  lw $ra,0($sp)\n"); //恢复返回地址

                            fprintf(fp, "  addi $sp, $sp, 4\n");

                            fprintf(fp, "  sw $v0, %d($sp)\n",h->result.offset);

                            break;

                            }

                        if (!strcmp(h->opn1.id,"write")){ //特殊处理write

                            fprintf(fp, "  lw $a0, %d($sp)\n",h->prior->result.offset);

                            fprintf(fp, "  addi $sp, $sp, -4\n");

                            fprintf(fp, "  sw $ra,0($sp)\n");

                            fprintf(fp, "  jal write\n");

                            fprintf(fp, "  lw $ra,0($sp)\n");

                            fprintf(fp, "  addi $sp, $sp, 4\n");

                            break;

                            }

                        for(p=h,i=0;i<symbolTable.symbols[h->opn1.offset].paramnum;i++)  //定位到第一个实参的结点

                                p=p->prior;

                        //开活动记录空间

                        fprintf(fp, "  move $t0,$sp\n"); //保存当前函数的sp到$t0中，为了取实参表达式的值

                        fprintf(fp, "  addi $sp, $sp, -%d\n", symbolTable.symbols[h->opn1.offset].offset);

                        fprintf(fp, "  sw $ra,0($sp)\n"); //保留返回地址

                        i=h->opn1.offset+1;  //第一个形参变量在符号表的位置序号

                        while (symbolTable.symbols[i].flag=='P')

                            {

                            fprintf(fp, "  lw $t1, %d($t0)\n", p->result.offset); //取实参值

                            fprintf(fp, "  move $t3,$t1\n");

                            fprintf(fp, "  sw $t3,%d($sp)\n",  symbolTable.symbols[i].offset); //送到被调用函数的形参单元

                            p=p->next; i++;

                            }

                        fprintf(fp, "  jal %s\n",h->opn1.id); //恢复返回地址

                        fprintf(fp, "  lw $ra,0($sp)\n"); //恢复返回地址

                        fprintf(fp, "  addi $sp,$sp,%d\n",symbolTable.symbols[h->opn1.offset].offset); //释放活动记录空间

                        fprintf(fp, "  sw $v0,%d($sp)\n", h->result.offset); //取返回值

                        break;

            case RETURN:fprintf(fp, "  lw $v0,%d($sp)\n",h->result.offset); //返回值送到$v0

                        fprintf(fp, "  jr $ra\n");

                        break;

        }

    h=h->next;

    } while (h!=head);

fclose(fp);

}

# 六、最终效果展示

## （一）、代码目录

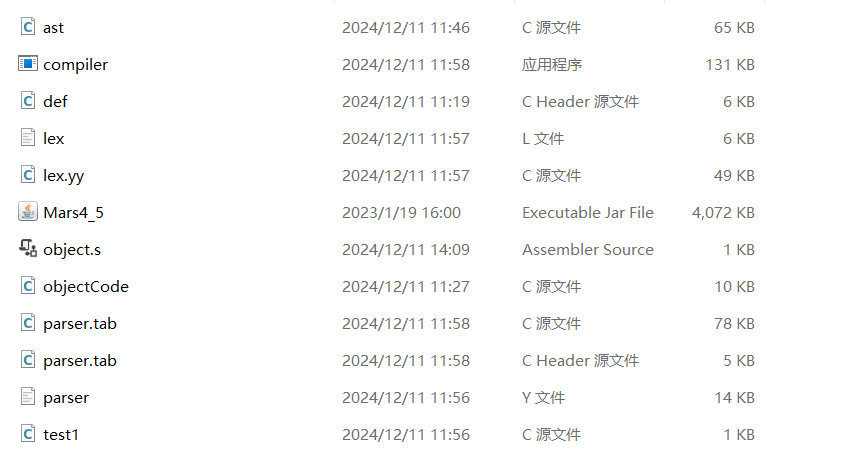


图 11最终代码目录

## （二）、最终效果

### 1.编译运行

（1）生成词法分析器

运行以下命令：

**flex lex.l**

这会生成 lex.yy.c 文件。

（2）生成语法分析器

运行以下命令：

**bison -d parser.y**

这会生成 parser.tab.c 和 parser.tab.h 文件。

（3）编译所有 C 源文件

**gcc -o compiler lex.yy.c parser.tab.c ast.c objectCode.c**

（4）运行编译器

运行编译器，输入测试用例文件（test1）：

**./compiler test1.c**

这会生成目标代码文件（ object.s）。

（5）运行目标代码

**java -jar Mars4\_5.jar object.s**

### 2.测试代码

int main(int a, int b, int c) {

    int sum ;

    int arr[5];

    sum = read();

    arr[2] = read();

    arr[1] = read();

    sum += arr[1];

    sum += arr[2];

    write(sum);

    //return 0;

}

### 3.运行效果

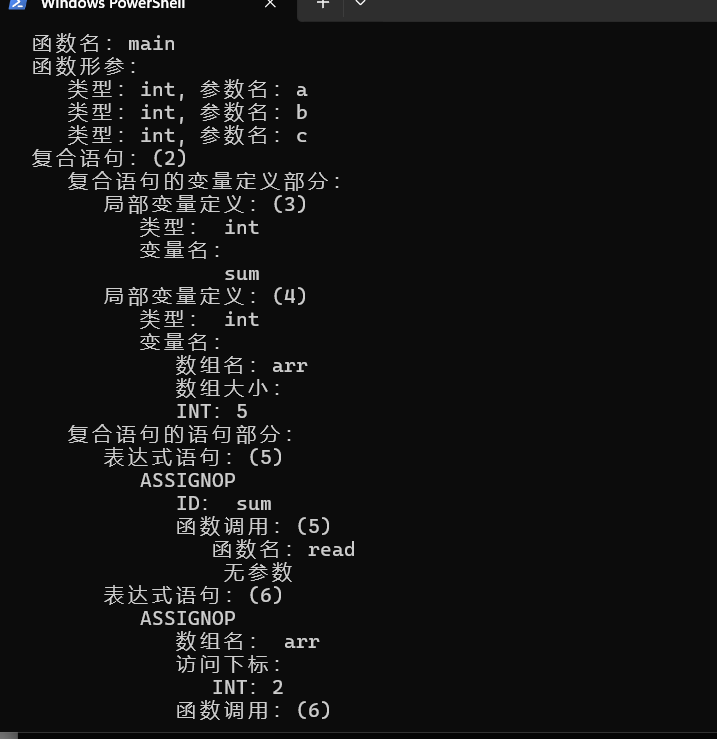


图 12词法分析

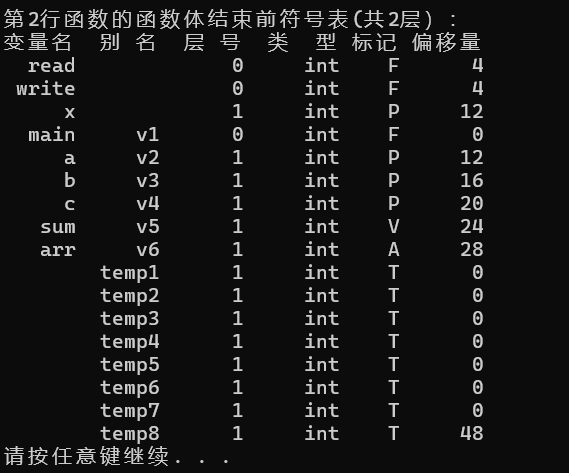


图 13符号表生成

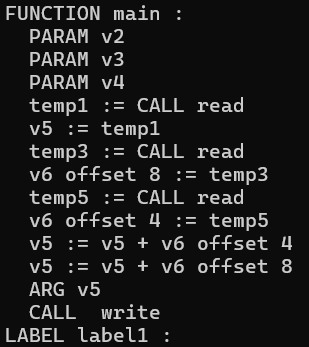


图 14中间代码生成

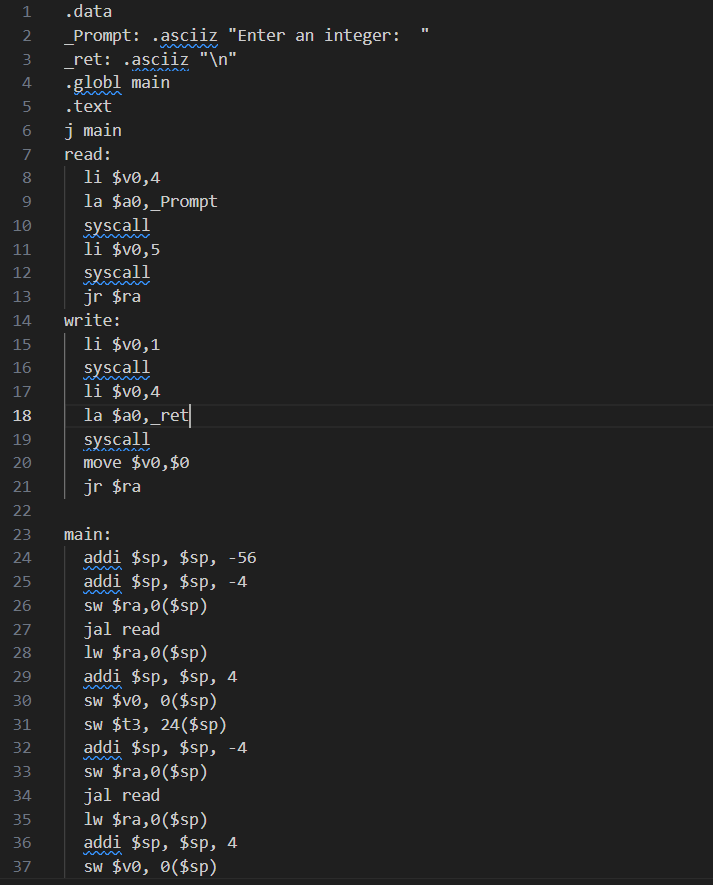


图 15目标代码生成

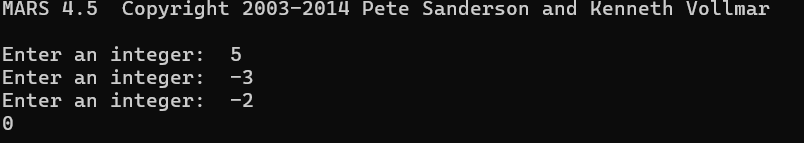


图 16最终代码运行效果

# 七、总结

在整个编译原理实验的过程中，从词法语法分析开始，到最后的目标代码生成，在此期间我遇到了非常多的bug，可以说做出一个K语言编译器的过程，也即是分析bug解决bug的过程。

遇到的第一个困难就是lex.l文件如何编写，以及如何编译。我参考了任务指导书中间的正则表达式一部分的内容，又参考了给出的几个示例，并去网上搜索了相关资料，最后能够独立的用正则表达式写出char类型和注释的表达方式。并能够正确的传达单词的含义。

其次是如何处理parser.y文件中的数组，在解决多次未果后，我决定采用笨办法，就是把数组一点一点的处理，先处理数组名，再处理数组的每一维，对数组的最后一维做个标记。

在编译器的语义分析阶段，处理变量类型错误（如类型不匹配、未声明或重复声明）相对直接，可以利用现有代码逻辑迅速判断和处理。然而，对于`break`和`continue`这类控制流语句，则需要额外的设计来确保它们仅能在循环结构（如`for`或`while`）中被正确使用。为了解决这一问题，我选择在抽象语法树（AST）节点中引入了一个名为`inWhileFor`的标志位，用于标识当前是否位于`for`或`while`循环体内。考虑到循环体内可能嵌套多层语句，为了维持`inWhileFor`标志的有效性，在语句传递过程中必须同步更新和传递此标志。

数组的处理是语义分析中的一个复杂点。为此，我将数组操作区分为声明和调用两种情况，其中只有调用涉及中间代码和目标代码的生成。进一步地，根据下标类型的不同，将数组调用细分为整数索引（INT型）和标识符索引（ID型）。对于ID型索引，每次调用时需重新计算偏移地址；而对于INT型索引，则可以在程序初始化时一次性计算。此外，我还设计了两种计算偏移地址的方法：一种用于确定数组宽度，另一种用于多维数组元素相对于首地址的偏移量。

当遇到赋值操作（ASSIGNOP）时，原有的代码直接使用了传入数据的偏移地址，这在处理数组时会导致只能获取到数组的首地址，而非特定元素的地址。因此，对数组类型的传入数据进行了特殊处理，确保能正确计算出对应元素的偏移地址。类似地，对于自增（DPLUS）、自减（DMINUS）、复合赋值（如ADDASSIGNOP、SUBASSIGNOP）及算术运算（+-\*/），也采取了相应的调整措施以应对表达式中包含数组的情况

在中间代码生成阶段，虽然大部分情况下可以直接沿用现有逻辑，但对于`for`语句，由于其相较于`while`语句多了两个额外的表达式，因此需要特别处理。此外，基于之前对数组调用的分类，中间代码生成也需要针对不同的情况进行适配，特别是当下标为ID时，应传递下标的临时变量（temp value）作为操作数的偏移地址；而当下标为INT时，则可直接计算偏移地址。

最终，在目标代码生成阶段，为实现可运行代码的生成，我简化了部分功能，例如只支持整数索引的数组操作，放弃了对标识符索引的支持。同时，基于给定的代码示例，实现了对自定义函数返回值的处理，并区分了main函数和其他自定义函数的返回机制。

上述工作展示了我在编译器开发过程中面对挑战时逐步优化解决方案的过程，尽管某些高级特性未能完全实现，但成功解决了数组的基本赋值与运算、`break`和`continue`语句的跳转等功能，这些都是编译器项目中的难点与亮点。

# **八、**参考文献及网址

1. **Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2009). 编译原理 (陈火旺, 顾小丰, 蔡莲红, & 蒋维杜, Trans.). 机械工业出版社. (Original work published 2006).**
2. **Kernighan, B. W., & Ritchie, D. M. (2006). C程序设计语言 (徐宝文 & 郑宗敏, Trans.). 机械工业出版社. (Original work published 1988).**
3. **Shao, B. (2006). 编译原理与实践. 清华大学出版社.**
4. **LLVM Project. (n.d.). Getting started with the LLVM system. [https://llvm.org/docs/GettingStarted.html.zhcn](https://llvm.org/docs/GettingStarted.html.zhcn" \t "_blank)**
5. **GCC Team. (n.d.). GCC internals. GNU Project. [https://gcc.gnu.org/onlinedocs/gccint/index.html](https://gcc.gnu.org/onlinedocs/gccint/index.html" \t "_blank)**
6. **Flex Tutorial. (n.d.). Yuque. [https://www.yuque.com/books/share/6B6A6F5E-7D5C-4D64-B832-286E83E7F462](https://www.yuque.com/books/share/6B6A6F5E-7D5C-4D64-B832-286E83E7F462" \t "_blank)**
7. **Bison Tutorial. (n.d.). Yuque. [https://www.yuque.com/books/share/35E5C4A3-AE64-4CA3-9A5C-4EBD5E1F3E9C](https://www.yuque.com/books/share/35E5C4A3-AE64-4CA3-9A5C-4EBD5E1F3E9C" \t "_blank)**
8. **ISO/IEC. (2011). ISO/IEC 9899:2011 - Programming languages -- C. [http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf](http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf" \t "_blank)**
9. [lingbai-kong/C-like-compiler: 同济大学CS《编译原理》课程设计: 类C语言编译器TongJi University CS compiler theory assignment](https://github.com/lingbai-kong/C-like-compiler)
10. [《编译原理》课程设计: 类C语言编译器+源代码+文档说明\_编译原理课程设计c语言编译器-CSDN博客](https://blog.csdn.net/m0_73728511/article/details/139532934)