

目 录

2. 程序的组织结构.....	1
2.1 词法分析器.....	1
2.2 语法分析器.....	3
2.3 语义分析与代码生成.....	5
2.4 虚拟机.....	5
2.5 功能结构图.....	5
3. 程序流程图.....	6
3.1 词法分析.....	6
3.2 语法、语义分析与代码生成.....	6
3.3 虚拟机.....	7
4. 系统实现与源代码.....	8
4.1 flex 部分设计	8
4.2 bison 部分设计	10
4.2.1 预处理部分.....	10
4.2.2 整体结构部分.....	12
4.2.3 声明部分.....	13
4.2.4 语句部分.....	15
4.2.5 表达式部分.....	17
4.2.6 函数部分.....	19
4.3 主程序头文件设计.....	20
4.4 主程序设计.....	22
5. 程序编译.....	27
6. 系统测试.....	29
6.1 基本测试.....	29
6.2 IF 测试.....	30
6.3 WHILE 测试.....	31
6.4 PROCEDURE 测试.....	32

6.5 ODD 测试.....	33
6.6 嵌套测试.....	34

github-codingchip

2. 程序的组织结构

基于 flex 和 bison 的 pl/0 编译器采用经典的编译器结构，主要由以下几个模块组成：

2.1 词法分析器

flex 是用于构建编译器前端的经典工具，是 unix 工具 lex 在 windows 上的开源实现，主要用于词法分析。依据 flex 的语法规则，在.l 文件中编辑对应关键词的正则式和声明等，就可以由 flex 自动进行分析与返回对应的动作。

flex 的语法较为简单，大体可分为三部分：定义(definitions)、规则(rules)、代码(codes)，不同部分之间用%%隔开。

而第一部分用%{与}%}框起来的部分是便我们可以使用 C 语言代码进行预处理，例如使用#include<stdio.h>或是定义常量等等，其余部分则是可以自行设置标识符绑定于正则式上用于下一部分进行配对，或是对词法分析进行设置等操作。

第二部分由多条规则(rule)组成，每条规则由 pattren 和 action 组成。其中 pattern 使用正则表达式表示，或是引用上一部分定义好的关键字，代表需要匹配的词的规则；而 action 则使用代码来表示，指匹配成功后执行的动作。

Pattern 参考如下：

x	匹配字符x
.	匹配除换行(newline)外的任意字符
[xyz]	x或y或z
[abj-oZ]	匹配a或者b或者j到o的某个字符或者Z
[^A-Z]	匹配除A到Z的字符
[^A-Z\n]	除A-Z和换行
r*	0个或多个r
r+	1个或多个r
r?	0个或1个r
r{2,5}	rr或者rrrr或者rrrrr (即2到5个r)
r{2,}	2个或者2个以上r
r{4}	rrrr
(name)	由name定义的扩展
"[xyz]"^foo"	[xyz]^foo
\x	若x是a、b、f、n、r、t、v则按照ANSI-C的规则转义, 否则是x本身
\0	NUL
\123	十进制值123
\x2a	16进制值2a
(r)	被括号改变优先级的r
rs	连接
r s	或
r/s	当且仅当r后面有s时才匹配
^r	只有当r在行首才匹配
r\$	只有当r在行末尾才匹配
<s>r	r, 但是只满足初始情形s (start condition)
<^>r	满足任何初始情形的r
<<EOF>>	文件结束符

https://blog.csdn.net/weixin_44097532

图 1 正则表达式参考

而第三部分则是用来写 c 语言的地方。如果仅仅只有 flex 文件，那么我们可以把主函数和调用函数写在这里。但我们接下来有与 bison 联动，并额外编写了一个 c 语言文件来负责运行，因此这部分可以空着。

```
//仅参考 flex 语法格式
%{
int num=0;
int chars=0;
%}
%%//左边为正则式，右边为匹配上了后的执行动作。本示例为计数功能。
[a-zA-Z]+ { chars++; }
[1-9]+[0-9]* {num++;}
\n
.
%%
main(int argc, char **argv)//以下部分可以写在其他文件里
{
yylex();
```

```

printf("%8d%8d\n", num, chars);
}
int yywrap()
{
    return 1;
}

```

本次设计的词法分析器便是由 Flex 实现,负责识别 PL/0 语言中的词法单元,并处理标识符、数字、关键字和运算符,返回对应的关键词与值用于接下来的语法分析。这里只是简单介绍一下 flex 工具。

2.2 语法分析器

bison 也是用于构建编译器前端的经典工具,是 unix 工具 yacc 在 windows 上的开源实现。主要是用于语法分析、语义分析和代码生成。bison 能够对.y 文件中定义的语法规则自动生成 LALR(1)分析器,同时还支持语义动作,可以在归约的时候执行对应的操作,并且也支持冲突处理与优先级处理,是一个非常高效的工具。

bison 的语法规则与 flex 类似,也是大体分为三个部分,之间用%%分隔。第一部分也用了%{与}%进行分割,以便使用 c 语言预处理。比较特别的是, bison 会在这一部分在%token 后边定义符号(关键字),它们会被翻译成 C 头文件,被 flex 引用与匹配对应正则式后执行的动作,通过 yylex()函数 return 回来。

而第二部分则是用于记录编译器的产生式,由冒号代替箭头,分号结束。从最顶层的 program:block PERIOD 部分到声明列表,到赋值语句等,以及最后的表达式。具体的产生式都在这里被详细记录着,在产生式后面有时也会跟着写上一些动作,这些动作会在执行对应产生式的归约时来执行,例如中间代码的生成就可以写在这个地方。

最后的第三部分与 flex 大致相同,这里不做赘述。

```

//仅作 bison 语法格式参考
/*test.y*/
%{
#include <stdio.h>
#include <string.h>
int yylex(void);
void yyerror(char *);
%}

```

```

%token NUM ADD SUB MUL DIV VAR CR//定义的关键字

%%//产生式
    line_list: line
               | line_list line
               ;

    line : expression CR {printf("YES\n");} //归约时执行动作

    expression: term
               | expression ADD term
               | expression SUB term
               ;

    term: single
         | term MUL single
         | term DIV single
         ;

    single: NUM
          | VAR
          ;

%%

void yyerror(char *str){
    fprintf(stderr,"error:%s\n",str);
}

int yywrap(){
    return 1;
}

int main()
{
    yyparse();
}

```

本次设计的语法分析部分就是使用 **bison** 实现，根据 **pl/0** 的语法定义了语法规则，能够对输入的代码进行语法分析。这部分利用 **bison** 对产生式自动进行语法树构建，同时生成 **LALR(1)**分析器，可以完整实现对输入的移进归约的合法操作，省去了手动归约操作的复杂。

2.3 语义分析与代码生成

这部分通过 C 语言和 `bison` 实现，主要是负责管理符号表的添加修改等，对语义进行检查是否有先调用后声明的情况出现等，同时也负责中间代码的生成，为代码移交给虚拟机编译运行做准备。

其中语义分析逻辑直接嵌入在语法规则的语义动作当中，对应符号表的管理、类型检查与控制流检查都一起写在了 `.y` 文件当中，只有小部分辅助函数在 `.c` 文件中实现。

而代码生成部分则是先由 `bison` 在产生式归约时的动作里添加生成指令的语句，而指令具体的函数实现则是由 `.c` 文件中来进行执行。

2.4 虚拟机

虚拟机则负责解释前述部分所生成的代码，将其运行起来。这部分之中则要求虚拟机能够正确处理生成的代码，需要与之前分析和生成的部分做好关联对接。

2.5 功能结构图

本次编译器的功能结构图如下：

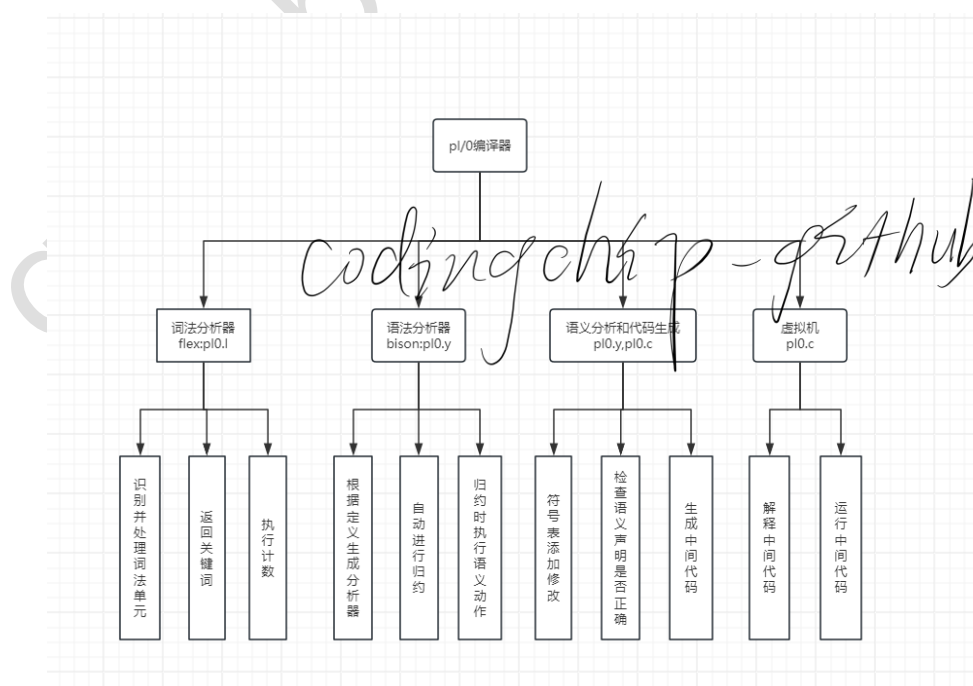


图 2 功能结构图

3. 程序流程图

3.1 词法分析

利用 flex 进行词法分析。在 flex 里用正则式规定好对应元素代表的符号范围后，就可以在 flex 对语句检测到该符号后返回对应的元素，同时还可以附带动作：例如计数等等。

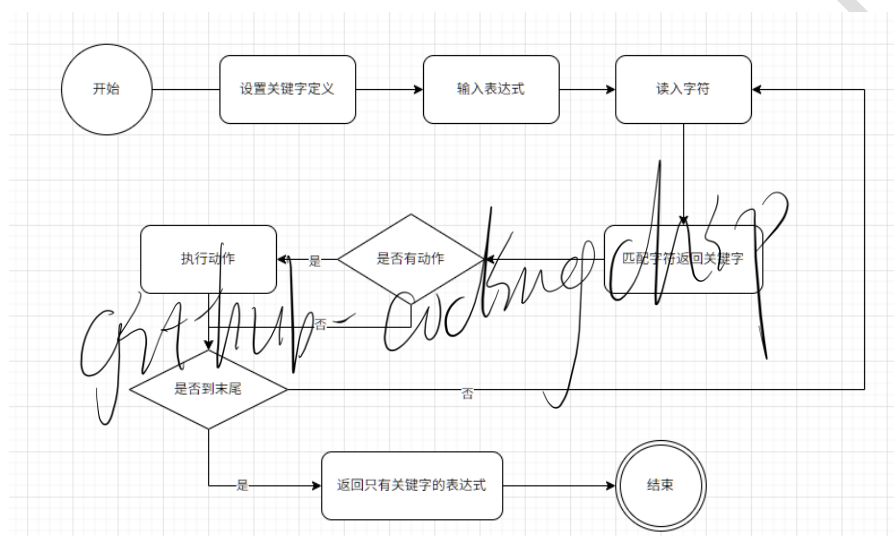


图 3 词法分析流程图

3.2 语法、语义分析与代码生成

本次设计中语法分析与语义分析和代码生成是同步进行的，一同写在.y 文件里利用 bison 进行执行。

语法分析器 bison 里能够自动根据定义的产生式生成 LALR(1)分析器进行语法分析，因此在 bison 里我们需要定义我们设计的 pl/0 编译器的产生式。同时产生式在归约时也可以执行输入的动作，例如产生中间代码等等，我们可以借助这个来实现在归约时生成中间代码。

在经过词法分析器将输入转换为 token 流后，分析器便启动了。分析器会从第一个产生式 program 开始推导，由 program 到 block(代码块)再到 declaration_list(声明)与最后的 statement(语句)，以此进行递归下降分析。同时递归下降分

析过程中，在声明部分分析器会检查并修改符号表，以确保声明的标识符存在且不冲突；在执行产生式归约时，分析器会调用对应动作进行代码生成。在生成时，若是对于代码中的需要位置未知，则会保留该位置，等到之后处理完后获得该位置了再进行回填，以确保代码生成的完整。最后等到所有语句处理完之后，进入表达式处理。由于表达式产生的是一个值，并不像语句会进行转跳等，是线性生成的，因此不需要进行地址回填，等到处理结束后就可以生成完整的中间代码了。

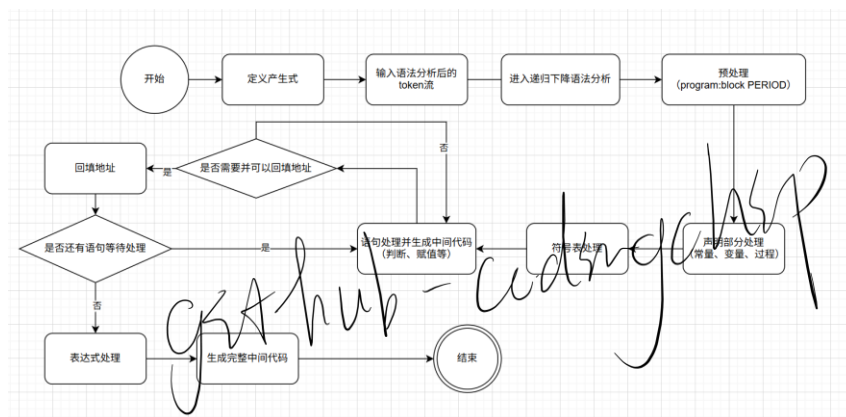


图 4 语法、语义分析与代码生成流程图

3.3 虚拟机

虚拟机的结构比较简洁。首先对虚拟机进行初始化，将指令指针、寄存器等都重置为初始状态。随后就根据先前生成的代码进行指令循环执行。在虚拟机中内置了不同操作码实现的不同功能指令，具体则由指令指针的指向来决定执行什么指令。

同时虚拟机中也负责定义代码生成函数的具体内容，以及符号表的查找与登记等等功能。

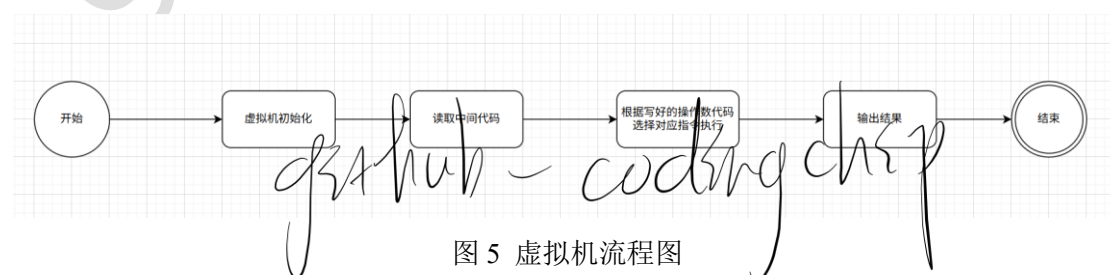


图 5 虚拟机流程图

4. 系统实现与源代码

4.1 flex 部分设计

Flex 的主要功能是进行词法分析，因此在.l 文件内我们导入对应库，并初始化了行号和列号以用于定位函数 `count()` 的计数，保证我们能够精确定位目前的编辑位置。随后我们根据 `pl/0` 编译器的功能和需要，设置了 `DIGIT`、`LETTER`、`NUMBER`、`ID`、`COMMENT` 五个标识符，并绑定了对应的正则表达式，用于第二部分的实现。

```
/* pl0.l - PL/0 词法分析器 第一部分*/

%{
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "pl0.h"
#include "pl0.tab.h"

int line_no = 1; //行列号初始化
int col_no = 1;

void count() { //定位函数
    int i;
    for (i = 0; yytext[i] != '\0'; i++) {
        if (yytext[i] == '\n') {
            col_no = 1;
            line_no++;
        } else if (yytext[i] == '\t') {
            col_no += 8 - (col_no % 8);
        } else {
            col_no++;
        }
    }
}

}%

%option noyywrap //对词法分析器进行设置，这里是不需要用 yywrap()函数
%option case-insensitive //以及不区分大小写
```

```

DIGIT    [0-9] //单个数字
LETTER   [a-zA-Z] //单个字母
ID        {LETTER}({LETTER}|{DIGIT})* //不为空的字母开头的字符串
NUMBER   {DIGIT}+ //不为空的数字
COMMENT  \{[^}]*\} //匹配在{}内除了}外的所有字符，表示注释

%%

```

第二部分则是根据 `pl/0` 的源代码中定义的关键字（例如 `CONST`、`VAR`）、支持的符号（例如：`=`、`+`）以及常量变量等设置了对应匹配的关键字（或正则表达式）以及在成功匹配时执行的动作：更新定位函数以及返回对应的关键字。特别注意，如果这里需要引用上一部分定义了正则式的标识符，需要用花括号框起来。这里的 `ID` 与 `NUMBER` 执行的动作不仅有定位更新，更有助于存储对应关键字代表的数值或字符的动作，以便后续代码执行时能够正确调用这些值。

```

/*第二部分*/
{COMMENT}      { count(); /* 忽略注释 */ }
[ \t\r]        { count(); /* 忽略空白字符 */ }
\n             { count(); } //换行

"CONST"        { count(); return CONST; } //返回关键字
"VAR"          { count(); return VAR; }
"PROCEDURE"    { count(); return PROCEDURE; }
"CALL"         { count(); return CALL; }
..... //省略一部分

";="           { count(); return ASSIGN; }
"="            { count(); return EQ; }
"#"            { count(); return NE; }
"<"           { count(); return LT; }
..... //省略一部分
{ID}           {
                count();
                yyval.ident = (char*)malloc(strlen(yytext) + 1); //分配空间
                strcpy(yyval.ident, yytext); //存储对应字符
                return IDENT;
            }

{NUMBER}       {
                count();
                yyval.number = atoi(yytext); //存储对应数值
                return NUMBER;
            }

```

```

        }

        {
            count();
            printf("Error: Unexpected character '%s' at line %d, column %d\n",
                yytext, line_no, col_no); //报错
            return yytext[0];
        }

%%

```

而第三部分就不需要写什么了，具体的内容放到其他文件里实现。

4.2 bison 部分设计

Bison 部分设计了语法分析、语义分析和代码生成，因此其代码量也比较多。

4.2.1 预处理部分

在第一部分里我们不仅由导入对应需要的库，也有对定义在其他文件里的全局函数或变量的声明，以及在本文件里定义的全局变量和函数的声明。

除此之外，在本代码中还编写了一段`%union`，用于自适应存储对应数值变量和字符变量的语义值。

随后则是对在`%token` 中根据 `pl/0` 编译器的关键字以及前文定义的词法分析匹配的定义词法单元，其中在关键字前方有类似`<number>`等的部分是用于表示该关键词有附带的语义值，而括号内的 `number` 则表示了对应语义值的类型。

而与之不同的还有`%left` 以及`%nonassoc` 这两个运算符声明，其中`%left` 代表了左结合，即从左到右依次计算（匹配），适用于我们正常的运算顺序。与之相对的还有`%right`，则是声明了从右到左依次运算的运算符，在本代码中并没有应用。而`%nonassoc` 则是声明了不可结合的运算符，代表该运算符不能连续出现，例如输入 `A<B<C` 则会报错。

```

/* pl0.y - PL/0 bison 第一部分*/

%{ //导入所需库
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

```

```

#include "pl0.h"
//声明全局函数以及变量
extern int yylex();
extern void yyerror(const char *s);
extern int line_no;
extern int col_no;
extern FILE *yyin;

/* 全局变量 */
int cx = 0; /* 代码索引 */
int level = 0; /* 当前层次 */
int tx = 0; /* 符号表索引 */
int dx = 0; /* 数据分配索引 */
int err_count = 0; /* 错误计数 */

struct instruction code[CXMAX]; /* 代码数组 */
struct symbol table[TXMAX]; /* 符号表 */

/* 函数声明 */
void enter(enum object kind);
int position(char *id);
void gen(enum fct f, int l, int a);
void error(int n);
void listcode(int from, int to);
void interpret();

%}

%union { //用于语义值的存储
    int number;
    char *ident;
}

%token <number> NUMBER //number 型语义值 (int)
%token <ident> IDENT //ident 型语义值 (char)
%token CONST VAR PROCEDURE CALL BEGIN_SYM END IF THEN WHILE DO ODD
READ WRITE
%token ASSIGN EQ NE LT LE GT GE PLUS MINUS TIMES SLASH
%token LPAREN RPAREN COMMA SEMICOLON PERIOD

%left PLUS MINUS //左结合
%left TIMES SLASH //右结合
%nonassoc UMINUS //不结合

```

4.2.2 整体结构部分

而对于 bison 部分的第二部分则是本部分的重要内容，它定义了从程序整体结构到表达式细节的所有语法规则，并与语义动作（例如生成中间代码、符号表管理）等紧密结合。

首先是整体结构 **program**，该部分定义了程序完整的结构，由 **block** 中蕴含各个代码块，且最后必须以 **PERIOD**（句点）结束。

```
program:
    block PERIOD
    {
        gen(OPR, 0, 0); /* 返回指令 */
        if (err_count == 0) {
            printf("\nCompilation successful!\n");
            listcode(0, cx); //输出中间代码
            printf("\nStart PL/0\n");
            interpret(); //解释执行程序
        } else {
            printf("\n%d errors in PL/0 program\n", err_count);
        }
    }
    ;
```

其次就是 **block** 结构，**block** 描述了整个程序的初始化部分。首先他会进行程序的预处理，当预处理完成后则会进行声明部分分析 **declaration_list**，处理完此部分后会紧接着进行中间部分处理，例如回填转跳地址，记录过程地址和大小等等。当这一部分也处理完后，就到语句的分析部分 **statement**，正式进入语句的处理部分。

其中的 **\$<number>\$** 是临时的 **number** 型存储栈，另外 **\$1** 则表示引用产生式右边第一位关键字的语义值，**\$2** 则表示第二位，以此类推。

```
block:
    {
        dx = 3;          /* 为链接数据预留空间 */
        int jmpaddr = cx;
        gen(JMP, 0, 0); /* 产生跳转指令，跳转地址未知 */
    }
```

```

        if (level > LEVMAX) {
            error(32);    /* 嵌套层次过深 */
        }
        $<number>$ = jmpaddr;
    }
    declaration_list
    {
        int jmpaddr = $<number>1;
        code[jmpaddr].a = cx;    /* 回填跳转地址 */
        /* 如果是过程，记录其入口地址 */
        int i;
        for (i = tx; i > 0; i--) {
            if (table[i].kind == PROCEDURE_SYM && table[i].adr == 0) {
                table[i].adr = cx;
                table[i].size = dx;
                break;
            }
        }
        gen(INT, 0, dx);    /* 分配空间 */
    }
    statement
    ;

```

4.2.3 声明部分

接下来的声明处理部分则是根据 pl/0 编译器的声明语法进行产生式的设计。其中包含了三个部分：常量声明 (const_declaration)、变量声明 (var_declaration) 与过程声明 (proc_declaration)。

对于常量声明，在 pl/0 编译器中是由常量关键字，标识符，等号以及值来组成的。因此我们在 bison 中编写时，也是遵从这个流程：先匹配 CONST 关键字，然后解析 const_list，list 中包含了关键字后面的内容，因此可以产生定义式 const_def (例如 a=2) 或者另一个 const_list 与逗号和定义式 (例如 a=2,b=3) 依次来保证多个定义能够共同匹配。而在 def 中，则需要匹配标识符 IDENT，等号 EQ 以及数值 NUMBER，在执行完这部分归约后会执行语义动作，目的是将标识符添加到符号表中，并存储该标识符附带的语义值。

```

declaration_list: //声明部分
    /* empty */

```

```

| declaration_list declaration
;

declaration: //三个部分
    const_declaration
    | var_declaration
    | proc_declaration
;

const_declaration: //常量声明
    CONST const_list SEMICOLON
;

const_list: //可以处理一个或多个
    const_def
    | const_list COMMA const_def
;

const_def: //具体定义
    IDENT EQ NUMBER
    {
        strcpy(id, $1);
        num = $3;
        enter(CONSTANT); //添加到符号表中，类型为常量
        free($1);
    }
;

```

对于变量声明和过程声明在大体上也与常量声明一致。变量声明不同的地方是不需要定义其语义值，只需要关键字与标识符即可。而过程声明则是还需要过程代码的输入，用 **BEGIN...END;** 来输入（例如：**PROCEDURE p; BEGIN ... END;**）同时进入过程时，程序会统计嵌套层次，以确保层次不会超过预定值。

```

//变量声明
var_declaration:
    VAR var_list SEMICOLON
;

var_list:
    IDENT //标识符
    {
        strcpy(id, $1);
        enter(VARIABLE); //添加到符号表中，类型为变量
    }
;

```



```

        free($1);
    }
| var_list COMMA IDENT //多个定义
{
    strcpy(id, $3);
    enter(VARIABLE);
    free($3);
}
;
//过程声明
proc_declaration:
    PROCEDURE IDENT //标识符
    {
        strcpy(id, $2);
        enter(PROCEDURE_SYM);
        free($2);
        level++; //层数计数
        dx = 3; /* 重置数据分配索引 */
    }
    SEMICOLON block SEMICOLON
    {
        gen(OPR, 0, 0); /* 过程返回 */
        level--; //退出减去层数
    }
;

```

4.2.4 语句部分

接下来则是语句部分，这一部分主要是根据 p1/0 中实现的语句功能进行修改实现，这里简单举几个例子介绍：

对于赋值语句 `assignment_statement`，会需要匹配标识符 `IDENT`，赋值符号 `ASSIGN` 以及表达式 `expression`。其中包括了查表失败的报错，标识符类型不对的报错，以及匹配成功归约后进行的代码生成语句。

```

statement: //各个语句
    /* empty */
| assignment_statement
| call_statement
| compound_statement
| if_statement
| while_statement

```

```

| read_statement
| write_statement
;

```

assignment_statement:

```

IDENT ASSIGN expression //表达式定义在后边
{
    int i = position($1);
    if (i == 0) {
        error(11); /* 标识符未声明 */
    } else if (table[i].kind != VARIABLE) {
        error(12); /* 不能给常量或过程赋值 */
    } else {
        gen(STO, level - table[i].level, table[i].adr);
    }
    free($1);
}
;

```

IF 语句部分则是需要匹配 IF 关键字，判断条件 condition，THEN 关键字，以及最后执行的语句 statement。

而在初次分析处理的时候，在 condition 与 statement 产生的中间代码中，并不知道具体的跳转地址。因此在 IF 语句归约时，编写了回填转跳地址的语义动作，在执行完其中 condition 以及 statement 的分析之后再执行该回填地址动作，保证指令的完整性。

if_statement:

```

IF condition THEN statement
{
    code[$<number>2].a = cx; /* 回填跳转地址 */
}
;

```

WHILE 语句也类似，先匹配 WHILE 关键字，之后保存循环开始的地址，以确保之后能够正确跳转到对应位置；随后匹配 condition 语句与 DO 关键词，并保存 JPC 目标地址，之后在匹配后续语句。当分析完毕后，再在归约时生成转跳指令并且回填转跳地址。

while_statement:

```

WHILE
{

```

```

    $<number>$ = cx; /* 保存循环开始地址 */
}
condition DO
{
    $<number>$ = $<number>3; /* 保存 JPC 地址 */
}
statement
{
    gen(JMP, 0, $<number>2); /* 跳回循环开始 */
    code[$<number>5].a = cx; /* 回填条件跳转地址 */
}
;

```

4.2.5 表达式部分

接下来则是有关表达式处理部分，其中包括了 **condition** 判断语句，**rel_op** 比较符，表达式 **expression**，项 **term** 以及因子 **factor**。这一部分都是根据 pl/0 源代码中的相关函数进行改写，其中 **condition** 主要负责奇偶判断以及表达式关系判断，会在归约时生成对应的条件跳转代码。

```

condition:
    ODD expression
    {
        gen(OPR, 0, 6); /* ODD 操作 */
        $<number>$ = cx;
        gen(JPC, 0, 0); /* 条件跳转 */
    }
    | expression rel_op expression
    {
        gen(OPR, 0, $<number>2); /* 关系运算 */
        $<number>$ = cx;
        gen(JPC, 0, 0); /* 条件跳转 */
    }
;

```

其中的 **rel_op** 就是关系运算符，存有对应的 OPR 指令码。

```

rel_op:
    EQ    { $<number>$ = 8; }
    | NE  { $<number>$ = 9; }
    | LT  { $<number>$ = 10; }

```

```
| GE  { $<number>$ = 11; }  
| GT  { $<number>$ = 12; }  
| LE  { $<number>$ = 13; }  
;  
;
```

表达式 **expression** 部分则负责加减法（以及取负），同时它也可以继续产生项 **term**。在这里处理加减法是为了能够优先处理 **term** 中处理的乘除，保证乘除法优先于加减法，确保运算顺序的正确。

在匹配完对应关键字执行归约时，该部分代码也会生成中间代码。

```
expression:  
  term  
  | PLUS term  
  | MINUS term %prec UMINUS  
  {  
    gen(OPR, 0, 1); /* 取负 */  
  }  
  | expression PLUS term  
  {  
    gen(OPR, 0, 2); /* 加法 */  
  }  
  | expression MINUS term  
  {  
    gen(OPR, 0, 3); /* 减法 */  
  }  
  ;  
  
term:  
  factor  
  | term TIMES factor  
  {  
    gen(OPR, 0, 4); /* 乘法 */  
  }  
  | term SLASH factor  
  {  
    gen(OPR, 0, 5); /* 除法 */  
  }  
  ;
```

最后到因子 **factor** 部分的处理。这一部分主要是用于查找对应标识符所拥有的语义值，进行变量或者常量的引用。首先会到符号表里查询是否有此标识符，再进入到对应标识符位置上寻找对应的语义值，如果出现过程标识符、数字越界、

标识符未声明等未进行报错。

```
factor:
  IDENT
  {
    int i = position($1);
    if (i == 0) {
      error(11); /* 标识符未声明 */
    } else {
      switch (table[i].kind) {
        case CONSTANT: //常量标识符
          gen(LIT, 0, table[i].val);
          break;
        case VARIABLE: //变量标识符
          gen(LOD, level - table[i].level, table[i].adr);
          break;
        case PROCEDURE_SYM:
          error(21); /* 表达式中不能有过程标识符 */
          break;
      }
    }
    free($1);
  }
  | NUMBER
  {
    if ($1 > AMAX) {
      error(30); /* 数值越界 */
      $1 = 0;
    }
    gen(LIT, 0, $1);
  }
  | LPAREN expression RPAREN
  ;

%%
```

4.2.6 函数部分

接下来就是 bison 文件里的最后一个部分。这一部分比较简单，只定义了报错函数，用于打印报错的行数列数，以及报错数量的统计，便于我们定位 bug 进行修正。

```
void yyerror(const char *s) {
    fprintf(stderr, "Error: %s at line %d, column %d\n", s, line_no, col_no);
    err_count++;
}
```

而其他的函数，我写在了另一个主程序 c 语言文件里，方便我们一起编译调用。

4.3 主程序头文件设计

头文件主要包含了主程序要使用到的数据结构以及基础的常量定义。例如标识符类型的符号枚举定义，虚拟机指令的枚举定义，符号表结构的实现，指令结构的实现，全局变量的声明，以及一些方法的声明：例如错误处理，符号表的登入与查找，代码生成与打印，虚拟机的解释执行。具体代码的对应作用和意义都标注在对应代码的注释上了。

```
/* pl0.h - PL/0 编译器头文件 */

#ifndef PL0_H
#define PL0_H

#include <stdio.h>

/* 常量定义 */
#define TXMAX 100 /* 符号表最大容量 */
#define CXMAX 500 /* 最多的虚拟机代码数 */
#define STACKSIZE 500 /* 运行时数据栈大小 */
#define LEVMAX 3 /* 最大嵌套层数 */
#define AMAX 2047 /* 地址上界 */
#define NMAX 14 /* 数字的最大位数 */
#define AL 10 /* 标识符的最大长度 */

/* 符号类型 */
enum object {
    CONSTANT, //常量
    VARIABLE, //变量
    PROCEDURE_SYM //过程
};

/* 虚拟机指令 */
```

```

enum fct {
    LIT,    /* 0: 将常数放到栈顶 */
    OPR,    /* 1: 执行运算 */
    LOD,    /* 2: 将变量放到栈顶 */
    STO,    /* 3: 将栈顶内容存到变量 */
    CAL,    /* 4: 调用过程 */
    INT,    /* 5: 分配空间 */
    JMP,    /* 6: 无条件跳转 */
    JPC     /* 7: 条件跳转 */
};

/* 符号表结构 */
struct symbol {
    char name[AL + 1];
    enum object kind;
    int val;    /* CONSTANT 使用 */
    int level; /* 所在层次 */
    int adr;    /* 地址 */
    int size;   /* PROCEDURE 使用 */
};

/* 指令结构 */
struct instruction {
    enum fct f; /* 功能码 */
    int l;      /* 层差 */
    int a;      /* 位移或运算号 */
};

/* 全局变量声明 */
extern char id[AL + 1]; /* 当前标识符 */
extern int num;         /* 当前数字 */
extern int cx;          /* 代码分配索引 */
extern int level;       /* 当前层次 */
extern int tx;          /* 符号表当前尾指针 */
extern int dx;          /* 数据分配索引 */
extern int err_count;   /* 错误计数 */

extern struct instruction code[CXMAX]; /* 存放虚拟机代码 */
extern struct symbol table[TXMAX];     /* 符号表 */

/* 错误处理 */
void error(int n);

/* 符号表管理 */

```

```

void enter(enum object kind); //符号表登记
int position(char *id); //符号表查找

/* 代码生成 */
void gen(enum fct f, int l, int a); //生成代码
void listcode(int from, int to); //打印代码

/* 虚拟机 */
void interpret(); //解释执行
int base(int l, int b, int s[]);

#endif /* PL0_H */

```

4.4 主程序设计

主程序主要是根据 pl/0 编译器的实现功能，对头文件中的函数进行具体实现。在这一部分中，主程序主要实现了：

错误处理函数，用于错误计数以及返回对应的报错信息；

```

/* 错误处理函数 */
void error(int n) {
    printf("Error %d: %s\n", n, err_msg[n]);
    err_count++;
}

```

符号表登记函数，用于将声明的常量、变量等登入进符号表；

```

/* 在符号表中登记符号 */
void enter(enum object kind) {
    tx++;
    if (tx > TXMAX) {
        printf("Program too long\n");
        exit(1);
    }

    strcpy(table[tx].name, id);
    table[tx].kind = kind;

    switch (kind) {
        case CONSTANT:
            if (num > AMAX) {
                error(30);
                num = 0;
            }
    }
}

```



```

        table[tx].val = num;
        break;

    case VARIABLE:
        table[tx].level = level;
        table[tx].adr = dx++;
        break;

    case PROCEDURE_SYM:
        table[tx].level = level;
        break;
}
}

```

符号表查找函数，用于在符号表中查找标识符的位置，以确认是否有对应的标识符已被声明；

```

/* 查找标识符在符号表中的位置 */
int position(char *id) {
    int i;
    strcpy(table[0].name, id); /* 哨兵 */
    i = tx;

    /* 从当前位置向前搜索，支持嵌套作用域 */
    while (strcmp(table[i].name, id) != 0) {
        i--;
    }

    /* 如果找到的是哨兵 (i==0)，说明未找到 */
    if (i == 0) {
        return 0;
    }

    /* 检查找到的标识符是否在当前可访问的作用域内 */
    /* PL/0 允许内层访问外层的标识符 */
    if (table[i].level <= level) {
        return i;
    }

    /* 如果找到的标识符在更深的层次，继续向前搜索 */
    int saved_i = i;
    i--;
    while (i > 0 && strcmp(table[i].name, id) != 0) {
        i--;
    }
}

```

```

    if (i > 0 && table[i].level <= level) {
        return i;
    }

    /* 如果没有找到合适的，返回第一个找到的（可能报错） */
    return saved_i;
}

```

生成虚拟机指令函数，用于 bison 文件中对应产生式归约时执行的生成中间代码的语义动作；

```

/* 生成虚拟机指令 */
void gen(enum fct f, int l, int a) {
    if (cx >= CXMAX) {
        printf("Program too long\n");
        exit(1);
    }
    code[cx].f = f;
    code[cx].l = l;
    code[cx].a = a;
    cx++;
}

```

代码列表输出函数，用于打印生成的指令；

```

/* 输出代码列表 */
void listcode(int from, int to) {
    int i;
    printf("\n=== OBJECT CODE ===\n");
    for (i = from; i < to; i++) {
        printf("%03d: %s %d %d\n", i, mnemonic[code[i].f], code[i].l, code[i].a);
    }
}

```

虚拟机解释器，用于解释并执行生成的中间代码。其中 base 函数是用来求目标地址的。

```

/* 通过静态链求上1层的基地址 */
int base(int l, int b, int s[]) {
    int b1 = b;
    while (l > 0) {
        b1 = s[b1];
        l--;
    }
    return b1;
}

```

```

/* 虚拟机解释执行 */
void interpret() {
    int p = 0;      /* 程序计数器 */
    int b = 1;      /* 基地址寄存器 */
    int t = 0;      /* 栈顶寄存器 */
    int s[STACKSIZE] = {0}; /* 数据栈 */
    struct instruction i; /* 当前指令 */

    printf("\n=== RUNNING PL/0 ===\n");
    s[1] = s[2] = s[3] = 0;

    do {
        i = code[p++];

        switch (i.f) {
            case LIT:
                s[++t] = i.a;
                break;

            case OPR:
                switch (i.a) {
                    case 0: /* 返回 */
                        t = b - 1;
                        p = s[t + 3];
                        b = s[t + 2];
                        break;
                    case 1: /* 取负 */
                        s[t] = -s[t];
                        break;
                    .....// 省略一部分
                }
                break;

            case LOD:
                t++;
                s[t] = s[base(i.l, b, s) + i.a];
                break;

            case STO:
                s[base(i.l, b, s) + i.a] = s[t];
                t--;
                break;
        }
    } while (i.f != 0);
}

```

```

        .....//省略一部分
    }
} while (p != 0);

printf("\n=== END PL/0 ===\n");
}

```

以上这些函数，大部分都是从 p1/0 编译器里复制过来的。有部分地方根据 flex 和 bison 里设置的变量和标识符进行对应的更改，但整体逻辑和流程基本上还是一致的。

由于主程序是写在了一个另外的 c 语言文件里，因此在主函数中需要做的部分有四个：预处理，根据输入的文件名打开对应文件，利用 yyparse() 函数启动语法分析，执行完成后清理资源并退出。

```

/* 主函数 */
int main(int argc, char *argv[]) {
    char filename[256];

    printf("PL/0 Compiler (Flex & Bison version)\n");

    if (argc > 1) { //命令行参数的输入文件模式
        strcpy(filename, argv[1]);
    } else { //交互式输入文件模式
        printf("Input PL/0 source file: ");
        scanf("%s", filename);
    }

    yyin = fopen(filename, "r"); //只读模式打开文件
    if (!yyin) {
        printf("Cannot open file %s\n", filename);
        return 1;
    }

    /* 初始化全局变量 */
    cx = 0;
    level = 0;
    tx = 0;
    dx = 0;
    err_count = 0;

    printf("Compiling %s...\n", filename);
    yyparse(); //启动语法分析
}

```

```
fclose(yyin); //关闭文件
return 0;
}
```

至此需要编写的代码就已经结束了，有关的 flex 文件和 bison 文件以及主程序文件都已经准备完全，从词法分析到语法分析、语义分析、代码生成以及虚拟机解释器都已经准备完全。接下来我们需要做的便是编译和运行对应的文件。

5. 程序编译

在完成程序的代码编写之后，就要进行对应程序的编译。

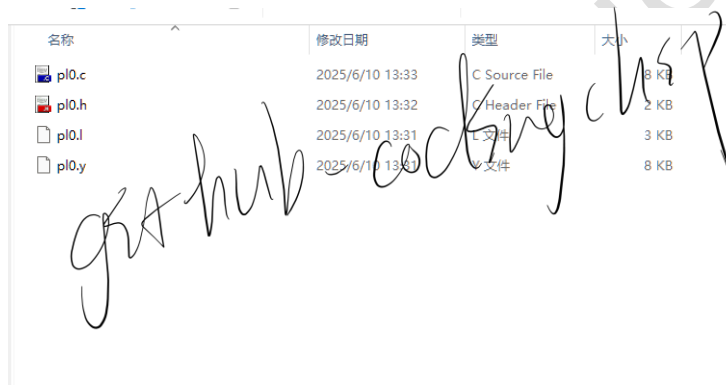


图 6 当前已有文件

```
>bison -d p10.y
>flex p10.l
```

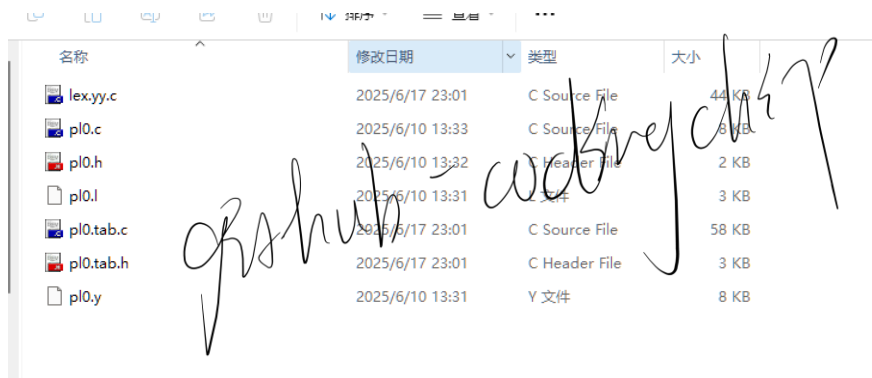
编译 bison 的.y 文件与 flex 的.l 文件。注意，由于 flex 文件中引用了 bison 文件编译后生成的 p10.tab.h 文件，因此需要优先编译 bison 的.y 文件，再编译 flex 的.l 文件。



图 7 cmd 窗口

编译完这两个文件后，将会生成三个文件，分别是 lex.yy.c, p10.tab.c 以及

pl0.tab.h。



名称	修改日期	类型	大小
lex.yy.c	2025/6/17 23:01	C Source File	44 KB
pl0.c	2025/6/10 13:33	C Source File	8 KB
pl0.h	2025/6/10 13:32	C Header File	2 KB
pl0.l	2025/6/10 13:31	L 文件	3 KB
pl0.tab.c	2025/6/17 23:01	C Source File	58 KB
pl0.tab.h	2025/6/17 23:01	C Header File	3 KB
pl0.y	2025/6/10 13:31	Y 文件	8 KB

图 8 编译后文件

lex.yy.c 是词法分析源代码, pl0.tab.c 以及 pl0.tab.h 是语法分析源代码和头文件。具体工作流程是由 lex.yy.c 读取源代码, 生成 token 流以及附加信息传递给 pl0.tab.c, pl0.tab.c 根据 pl0.tab.h 定义的 token 和语义值的类型进行归约并执行语义动作 (如生成代码)。

在之后我们就可以用 gcc 对这两者以及主程序 pl0.c 进行编译。

```
>gcc -c pl0.tab.c
>gcc -c lex.yy.c
>gcc -c pl0.c

>bison -d pl0.y
>flex pl0.l
>gcc -c pl0.tab.c
>gcc -c lex.yy.c
>gcc -c pl0.c
>
```

图 9 cmd 窗口

编译后生成了 pl0.tab.o 以及 lex.yy.o, pl0.o 文件。



名称	修改日期	类型	大小
pl0.h	2025/6/10 13:32	C Header File	2 KB
pl0.tab.h	2025/6/17 23:01	C Header File	3 KB
lex.yy.c	2025/6/17 23:01	C Source File	44 KB
pl0.c	2025/6/10 13:33	C Source File	8 KB
pl0.tab.c	2025/6/17 23:01	C Source File	58 KB
pl0.l	2025/6/10 13:31	L 文件	3 KB
lex.yy.o	2025/6/17 23:12	O 文件	14 KB
pl0.o	2025/6/17 23:15	O 文件	7 KB
pl0.tab.o	2025/6/17 23:12	O 文件	10 KB
pl0.y	2025/6/10 13:31	Y 文件	8 KB

图 10 编译后文件

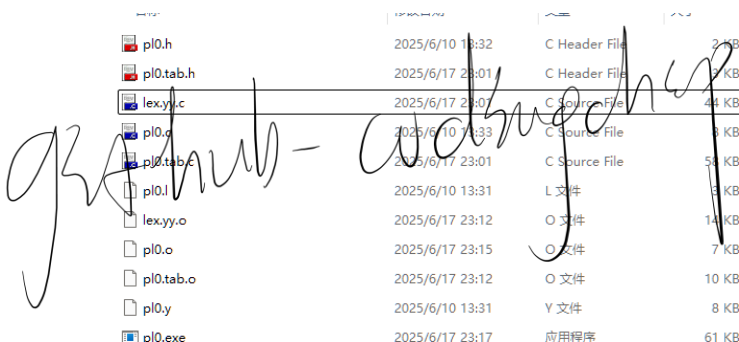
最后我们就可以将以上已经编译完成的目标.o 文件链接成一个可执行程序 pl0.exe。

```
>gcc -o pl0.exe pl0.tab.o lex.yy.o pl0.o
```

```
编译原理\课程设计\kcsj
>bison -d pl0.y
>flex pl0.l
>gcc -c pl0.tab.c
>gcc -c lex.yy.c
>gcc -c pl0.c
>gcc -o pl0.exe pl0.tab.o lex.yy.o pl0.o
>
```

图 11 链接完成后 cmd 窗口

可执行文件如图：



pl0.h	2025/6/10 13:32	C Header File	2 KB
pl0.tab.h	2025/6/17 23:01	C Header File	2 KB
lex.yy.c	2025/6/17 23:01	C Source File	44 KB
pl0.c	2025/6/10 13:33	C Source File	3 KB
pl0.tab.c	2025/6/17 23:01	C Source File	5 KB
pl0.l	2025/6/10 13:31	L 文件	3 KB
lex.yy.o	2025/6/17 23:12	O 文件	14 KB
pl0.o	2025/6/17 23:15	O 文件	7 KB
pl0.tab.o	2025/6/17 23:12	O 文件	10 KB
pl0.y	2025/6/10 13:31	Y 文件	8 KB
pl0.exe	2025/6/17 23:17	应用程序	61 KB

图 12 链接完成后文件总览

到目前为止，我们的编译器基本上已经完成了。接下来我们会编写几个 pl/0 编译器的简单程序进行测试。

6. 系统测试

本次测试准备了六个简单的程序，主要是测试编译器的各个关键字功能是否得到了正确的实现。

六个程序都已经编写好并已经封装为后缀为.pl0 的文件。只需要在该目录下的命令行窗口里执行编译器运行指令即可开始运行。

```
>pl0.exe test.pl0
```

6.1 基本测试

本阶段测试基本变量声明，常量声明和简单运算。

程序代码如下：

```

CONST
    a = 10,
    b = 20;
VAR
    x, y, z;
BEGIN
    x := a;
    y := b;
    z := x + y;
    WRITE(x, y, z)
END.

```

运行结果如下：

```

C:\Users\... \study\编译原理\课程设计\kcsj>p10.exe test1.p10
PL/0 Compiler (Flex & Bison version)
Compiling test1.p10...

Compilation successful!

=== OBJECT CODE ===
 0: JMP 0 1
 1: INT 0 6
 2: LIT 0 10
 3: STO 0 3
 4: LIT 0 20
 5: STO 0 4
 6: LOD 0 3
 7: LOD 0 4
 8: OPR 0 2
 9: STO 0 1
10: LOD 0 3
11: OPR 0 14
12: LOD 0 4
13: OPR 0 14
14: LOD 0 1
15: OPR 0 14
16: OPR 0 15
17: OPR 0 0

Start PL/0

=== RUNNING PL/0 ===
10 20 30

=== END PL/0 ===

```

图 13 test1 运行结果

6.2 IF 测试

本阶段测试 READ 输入功能，IF-THEN 语句功能，比较运算和算术乘法运算。

程序代码如下：

```

VAR
    x, y;
BEGIN

```



```
READ(x);
IF x > 0 THEN
    y := x * 2;
WRITE(y)
END.
```

运行结果如下：

```
PL/0 Compiler (Flex & Bison version)
Compiling test2.pl0...

Compilation successful!

=== OBJECT CODE ===
0: JMP 0 1
1: INT 0 5
2: OPR 0 16
3: STO 0 3
4: LOD 0 3
5: LIT 0 0
6: OPR 0 12
7: JPC 0 12
8: LOD 0 3
9: LIT 0 2
10: OPR 0 4
11: STO 0 1
12: LOD 0 1
13: OPR 0 14
14: OPR 0 15
15: OPR 0 0

Start PL/0

=== RUNNING PL/0 ===
? 3
6

=== END PL/0 ===
```

图 14 test2 运行结果

6.3 WHILE 测试

本阶段是循环结构和复合语句测试。

运行代码如下：

```
VAR
    i, sum;
BEGIN
    i := 1;
    sum := 0;
    WHILE i <= 10 DO
    BEGIN
        sum := sum + i;
        i := i + 1
    END;
    WRITE(sum)
END.
```

测试结果如下：

```

PL/0 Compiler (Flex & Bison version)
Compiling test3.pl0...
Compilation successful!

=== OBJECT CODE ===
0: JMP 0 1
1: INT 0 5
2: LIT 0 1
3: STO 0 3
4: LIT 0 0
5: STO 0 1
6: LOD 0 3
7: LIT 0 10
8: OPR 0 13
9: JPC 0 19
10: LOD 0 1
11: LOD 0 3
12: OPR 0 2
13: STO 0 1
14: LOD 0 3
15: LIT 0 1
16: OPR 0 2
17: STO 0 3
18: JMP 0 6
19: LOD 0 1
20: OPR 0 14
21: OPR 0 15
22: OPR 0 0

Start PL/0

=== RUNNING PL/0 ===
65
=== END PL/0 ===

```

图 15 while 测试结果

6.4 PROCEDURE 测试

本阶段主要测试过程和递归调用。

程序代码如下：

```

CONST
    max = 100;
VAR
    n;

PROCEDURE factorial;
VAR
    f, i;
BEGIN
    READ(n);
    f := 1;
    i := 1;
    WHILE i <= n DO
    BEGIN
        f := f * i;
        i := i + 1
    END;
    WRITE(f)
END;

BEGIN
    CALL factorial
END.

```

运行结果如下：

```
PL/O Compiler (Flex & Bison version)
Compiling test4.pl0...

Compilation successful!

=== OBJECT CODE ===
0: JMP 0 26
1: JMP 0 2
2: INT 0 5
3: OPR 0 16
4: STO 1 3
5: LIT 0 1
6: STO 0 3
7: LIT 0 1
8: STO 0 4
9: LOD 0 4
10: LOD 1 3
11: OPR 0 13
12: JPC 0 22
13: LOD 0 3
14: LOD 0 4
15: OPR 0 4
16: STO 0 3
17: LOD 0 4
18: LIT 0 1
19: OPR 0 2
20: STO 0 4
21: JMP 0 9
22: LOD 0 3
23: OPR 0 14
24: OPR 0 15
25: OPR 0 0
26: INT 0 5
27: CAL 0 2
28: OPR 0 0

Start PL/O

=== RUNNING PL/O ===
? 5
120

=== END PL/O ===
```

图 16 procedure 测试结果

6.5 ODD 测试

本阶段主要是测试奇偶判断。

输入代码如下：

```
VAR
    x;
BEGIN
    READ(x);
    IF ODD x THEN
        WRITE(1)
END.
```

运行结果如下：

```
PL/0 Compiler (Flex & Bison version)
Compiling test5.pl0...

Compilation successful!

=== OBJECT CODE ===
0: JMP 0 1
1: INT 0 4
2: OPR 0 16
3: STO 0 3
4: LOD 0 3
5: OPR 0 6
6: JPC 0 10
7: LIT 0 1
8: OPR 0 14
9: OPR 0 15
10: OPR 0 0

Start PL/0

=== RUNNING PL/0 ===
? 3
1

=== END PL/0 ===
```

图 17 ODD 测试结果

6.6 嵌套测试

本阶段主要是测试嵌套过程是否正确执行，以及变量是否正确在作用域内起到作用。

输入代码如下：

```
VAR
    a, b;

PROCEDURE p1;
VAR
    c;

    PROCEDURE p2;
    VAR
        d;
    BEGIN
        d := 4;
        c := c + d;
        a := a + c
    END;

BEGIN
    c := 3;
    CALL p2;
    b := a + c
END;
```

```
BEGIN
    a := 1;
    b := 2;
    CALL p1;
    WRITE(a, b)
END.
```

运行结果如下：

```
PL/0 Compiler (Flex & Bison version)
Compiling test6.pl0...
Compilation successful!

=== OBJECT CODE ===
0: JMP 0 24
1: JMP 0 15
2: JMP 0 3
3: INT 0 4
4: LIT 0 4
5: STO 0 3
6: LOD 1 3
7: LOD 0 3
8: OPR 0 2
9: STO 1 3
10: LOD 2 3
11: LOD 1 3
12: OPR 0 2
13: STO 2 3
14: OPR 0 0
15: INT 0 4
16: LIT 0 3
17: STO 0 3
18: CAL 0 3
19: LOD 1 3
20: LOD 0 3
21: OPR 0 2
22: STO 1 4
23: OPR 0 0
24: INT 0 4
25: LIT 0 1
26: STO 0 3
27: LIT 0 2
28: STO 0 4
29: CAL 0 15
30: LOD 0 3
31: OPR 0 14
32: LOD 0 4
33: OPR 0 14
34: OPR 0 15
35: OPR 0 0

Start PL/0

=== RUNNING PL/0 ===
8 8

=== END PL/0 ===
```

图 17 嵌套测试结果

至此，编译器的基本功能都已经测试完毕且正确执行。可见本编译器已经达成了应有的功能。