厦門大學

信息学院软件工程系

《编译技术》期末大作业报告

题　　目　PL/0 编译器的设计与实现

年　　级　软件工程 2021 级

姓　　名　顾畅

学　　号　22920212204375

2024 年 6 月 1 日

目录

1 实验要求

1.1 程序设计要求

设计并实现一个 PL/0 语言的编译器，能够将 PL/0 语言翻译成 P-code 语言。

1.2 PL/0 语言编译系统

PL/0 语言编译系统由编译程序和解释程序两部分组成，分别称为 PL/0 编译程序和类 P-code 解释程序。PL/0 语言程序被 PL/0 编译程序转换为等价的类 P-code 程序。当编译程序正常结束时，PL-/0 语言编译系统会调用解释程序(也称类 P-code 虚拟机)，解释执行所生成的目标程序，如图 1 所示。

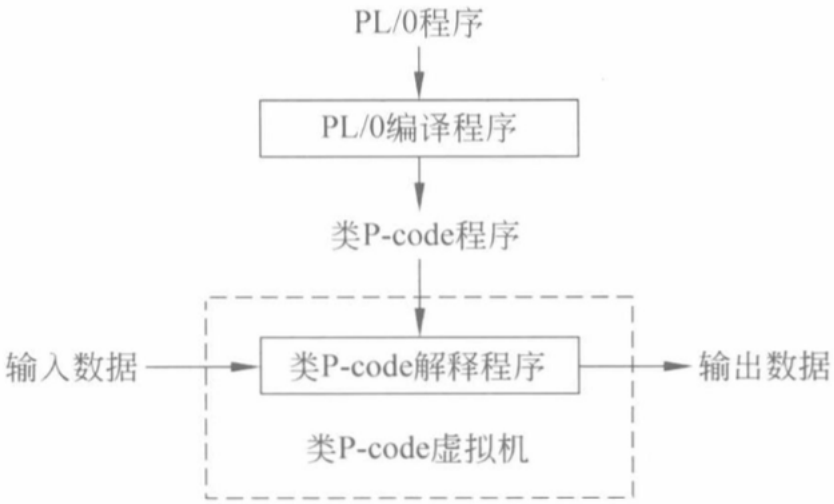

图 1 PL/0 语言编译系统

PL/0 编译程序将 PL/0 源程序翻译成类 P-code 目标程序，源语言为 PL/0，目标语言为类 Pcode，本实验的目的是实现一个 PL/0 编译程序，采用 C++实现，利于构建底层逻辑。

1.3 PL/0 编译程序

PL/0 编译程序采用单遍扫描方式的编译过程，由词法分析程序、语法语义分析程序以及代码生成程序 3 个独立的过程组成。PL/0 编译程序以语法语义分析程序为核心，当语法分析需要读单词时就调用词法分析程序，而当语法语义分析正确需生成相应语言成分的目标代码时，就调用代码生成程序

其编译过程采用一趟扫描方式，以语法分析程序为核心，词法分析和代码生成程序都作为一个独立的过程，当语法分析需要读单词时就调用词法分析程序，而当语法分析正确需要生成相应的目标代码时，则调用代码生成程序。PL/0 编译程序的组织结构如图 2 所示。

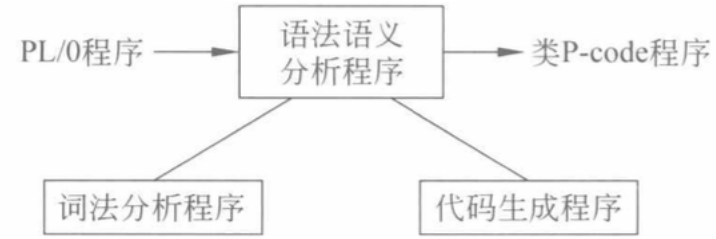

图 2 PL/0 编译程序组织结构

当源程序编译有错时，PL/0 编译程序用出错处理程序对词法和语法语义分析遇到的错误给出源程序出错的位置和错误性质。当源程序编译正确时，编译程序正常结束，可输出相应的类 P-code 目标程序。

1.4 PL/0 语言语法的 EBNF 描述

PL/0 语言是 Pascal 的一个子集，表 1 是 PL/0 语言语法的一个 EBNF 描述。

PL/0 语法单位	EBNF 描述
<程序>	::=<分程序>.
<分程序>	::=[<常量说明部分>][<变量说明部分>][<过程说明部分>]<语句>
<常量说明部分>	::=**const**<常量定义>{,<常量定义>};
<常量定义>	::=<id>=<integer>
<变量说明部分>	::=**var**<id>{,<id>};
<过程说明部分>	::=<过程首部><分程序>{;<过程说明部分>};
<过程首部>	::=**procedure** <id>;
<语句>	::=<赋值语句>|<条件语句>|<当型循环语句>|<过程调用语句>|<读语句>|<写语句>|<复合语句>|<空语句>
<赋值语句>	::=<id>:=<表达式>
<复合语句>	::=**begin**<语句>{;<语句>} **end**
<空语句>	::=ε
<条件>	::=<表达式><关系运算符><表达式>|**odd** <表达式>
<表达式>	::=[+|-] <项>{<加减运算符><项>}
<项>	::=<因子>{<乘除运算符><因子>}
<因子>	::=<id>|<integer>|'('<表达式>')'
<加减运算符>	::=+|-
<乘除运算符>	::=*|/
<关系运算符>	::==|#|<|<=|>|>=
<条件语句>	::=**if**<条件>**then** <语句>
<过程调用语句>	::=**call**<id>
<当型循环语句>	::=**while** <条件>**do** <语句>
<读语句>	::=**read** '('<id>{,<id>} ')'
<写语句>	::=**write** '('<表达式>{,<表达式>}')'

表 1 PL/0 语言语法的 EBNF 描述

1.5 类 P-code 语言

类 P-code 语言是 PL/0 编译程序的目标语言，可以看作类 P-code 虚拟机的汇编语言。类 P-code 虚拟机的指令格式形如：

F L A

它由 3 个部分构成，其含义如下：

F：指令的操作码。

L：若起作用，则表示引用层与声明层之间的层次差；若不起作用，则为 0。

A：不同的指令含义不同。

类 P-code 虚拟机完整的指令集合件见表 2。

指令分类	指令格式	指令功能
过程调用相关指令	INT 0 A	在栈顶开辟 A 个存储单元
	OPR 0 0	结束被调用过程，返回调用点并退栈
	CAL L A	调用地址为 A 的过程，调用过程与被调用过程的层差为 L
存取指令	INT 0 A	立即数 A 存入 t 所指单元，t 加 1
	LOD L A	将层差为 L、偏移量为 A 的存储单元的值取到栈顶，t 加 1
	STO L A	将栈顶的值存入层差为 L、偏移量为 A 的单元，t 减 1
一元运算和比较指令	OPR 0 1	求栈顶元素的相反数，结果值留在栈顶
	OPR 0 6	栈顶内容若为奇数则变为 1，若为偶数则变为 0
二元运算指令	OPR 0 2	次栈顶与栈顶的值相加，结果存入次栈顶，t 减 1
	OPR 0 3	次栈顶的值减去栈顶的值，结果存放次栈顶，t 减 1
	OPR 0 4	次栈顶的值乘以栈顶的值，结果存放次栈顶，t 减 1
	OPR 0 5	次栈顶的值乘以栈顶的值，结果存放次栈顶，t 减 1
二元比较指令	OPR 0 8	次栈顶与栈顶内容若相等，则将 0 存于次栈顶，t 减 1
	OPR 0 9	次栈顶与栈顶内容若不相等，则将 0 存于次栈顶，t 减 1
	OPR 0 10	次栈顶内容若小于栈顶，则将 0 存于次栈顶，t 减 1
	OPR 0 11	次栈顶内容若大小于等于栈顶，则将 0 存于次栈顶，t 减 1
	OPR 0 12	次栈顶内容若大于栈顶，则将 0 存于次栈顶，t 减 1
	OPR 0 13	次栈顶内容若小于等于栈顶，则将 0 存于次栈顶，t 减 1
转移指令	JMP 0 A	无条件转移至地址 A
	JPC 0 A	若栈顶为 0，则转移至地址 A，t 减 1
输入输出指令	OPR 0 14	栈顶的值输出至控制台屏幕，t 减 1
	OPR 0 15	控制台屏幕输出一个换行
	OPR 0 16	从控制台读入一行输入，置入栈顶，t 加 1

表 2 类 P-code 虚拟机指令系统

1.6 类 P-code 虚拟机的假想结构

类 P-code 虚拟机是一种简单的纯栈式结构的机器，它有一个栈式存储器，4 个控制寄存器：指令寄存器 i，指令地址寄存器 p、栈顶寄存器 t 和基址寄存器 b。类 P-code 程序运行期间的数据存储及逻辑运算都在栈顶进行。

2 功能结构设计

2.1 概述

本 PL/0 编译器共包括词法分析、语法分析、语义分析（包括符号表管理和目标代码生成）、错误处理四大部分组成。

2.2 词法分析

根据所给的 PL/0 语言的 BNF 描述，该语言的组成单词包括以下元素：

关键字（程序保留字）：{program，const，var，procedure，begin，end，if，else，then，call，while，do，read，write}；

运算符：{+，-，*，/，odd}；

界符：{“，”，“;”，“（”，“）”}；

关系运算符：{=，<,<=,>,>=,<>}；

数字：只能为整型，且常量不可以是负数；

标识符：由用户定义，以字母开头，由数字和字母组成；

词法分析程序读取源文件，识别出上述关键字、界符、关系运算符、数字、标识符五种元素，输出到 lex.txt 文件中，供后续的语法分析程序使用。其中，除标识符外，剩余的每一种字符可以用数字表示；而标识符则统一用一个数字表示其为标识符，再将标识符本身存储起来；数字的存储和标识符存储类似。具体见图 3 所示。

```
enum Symbol {
    PROG = 1,   //program
    BEG,        //begin
    END,        //end
    IF,         //if
    THEN,       //then
    ELS,        //else
    CON,        //const
    PROC,       //procdure
    VAR,        //var
    DO,         //do
    WHI,        //while
    CAL,        //call
    REA,        //read
    WRI,        //write
    REP,        //repeate
    ODD,        //ODDl      和keyWord中每个字的序号是相等的
    EQU,        //"="
    LES,        //"<"
    LESE,       //"<="
    LARE,       //">="
    LAR,        //">"
    NEQE,       //"<>"
    ADD,        //"+"
    SUB,        //"-"
    MUL,        //"*"
    DIV,        //"/"
    SYM,        //标识符
    CONST,      //常量
    CEQU,       //":="
    COMMA,      //","
    SEMIC,      //","
    POI,        //"."
    LBR,        //"("
    RBR,        //")"
};
```

图 3 词法分析编号方法

2.3 语法分析

语法分析结合 BNF 产生式，利用递归下降的方法实现。具体实现方式是，为每一个非终结符编写一个子程序，以<prog> → program <id>；<block>此产生式为列，用伪码描述其子程序如下：

```
void prog()
    if 当前指向的字符==program
        指向下一个字符
        if 当前指向的字符==id
            指向下一个字符
            if 当前指向的字符==;
                block()
            else
                error()
    error()
```

2.4 语义分析

语义分析在语法分析的基础上完成，涉及到的操作有符号表的管理和目标代码的生成，分别对应说明语句和处理语句。

2.4.1 符号表的管理

符号表中存储以下数据：定义的变量、定义的常量、定义的过程；

定义的变量需要存储：变量的标识符，变量定义所在层次，相对于该层次基地址的偏移量（对于基地址将在后面活动记录中详细说明）

定义的常量需要存储：常量的标识符，常量的值，定义所在层次

定义的过程需要存储：过程名，过程处理语句的开始地址（处理语句不是说明语句，说明语句中涉及到符号表的操作，而处理语句中涉及到产生目标代码的操作），过程定义所在层次；

具体如图 4 所示。

```
/**
 * TableRow是符号表中的每一行
 */
struct TableRow {
    int type;           //表示常量、变量或过程
    int value;          //表示常量或变量的值
    int level;          //嵌套层次，最大应为3，不在这里检查其是否小于等于，在SymbolTable中检查
    int address;      //相对于所在嵌套过程基地址的地址
    int size;         //表示常量，变量，过程所占的大小，此变量和具体硬件有关，实际上在本编译器中为了方便，统一设为4了,设置过程在SymTable中的三个enter函数中
    std::string name;   //变量、常量或过程名
    TableRow(int type = 0, int value = 0, int level = 0, int address = 0, int size = 0, std::string name = "");
};

/**
 * SymTable符号表，每一列的具体含义见TableRow
 */
struct SymbolTable {
    const int rowMax = 10000;           //最大表长
    const int valueMax = 100000;        //最大常量或变量值
    const int levelMax = 3;                 //最深嵌套层次
    const int addressMax = 10000;       //最大地址数


    const int myconst = 1;                  //常量类型用1表示
    const int var = 2;                      //变量类型用2表示
    const int proc = 3;             //过程类型用3表示

    //TableRow是符号表中的每一行
    //tablePtr指向符号表中已经填入值最后一项的下一项
    //length表示符号表中填入；了多少行数据，实际上可以用tablePtr来表示
    TableRow *const table;

    int tablePtr = 0;
    int length = 0;
```

图 4 符号表的定义

（1） 常量说明的翻译

```
void condecl()
    if 当前指向的字符==const
        指向下一个字符
        const()
        while 当前指向的字符==,
            指向下一个字符
            const()
void const()
    if 当前指向的字符==id
        指向下一个字符
        用 name 记录下字符
        if 当前指向的字符== :=
            指向下一个字符
            if 当前指向的字符==数字
                用 value 记录下值
                将其记录到符号表
```

（2） 变量说明的翻译

```
void vardecl()
    if 当前指向的字符==id
        用 name 记录下字符
        将其记录到符号表
        指向下一个字符
    While  当前指向的字符==,
           指向下一个字符
           if 当前指向的字符==id
               用 name 记录下字符
               将其记录到符号表
               指向下一个字符
```

（3） 过程说明的翻译

proc 的登录符号表的操作与上述方法类似。

2.4.2 目标代码的生成

由于说明语句是不产生目标代码的，结合 PL/0 的 BNF 描述，会产生目标代码的只有<block>、<body>、<statement>、<lexp>、<exp>、<term>、<factor>、<lop>、<mop>,下面说明其翻译模式。

（1）<block>

分析可以发现，主程序除去 program<id>，过程除 procedure<id>([id{,id}])，剩下的部分都是 block。则进入 block 有两种情况：从主程序进入，此时符号表中没有任何元素，也没有产生任何目标代码；或从过程进入，此时符号表中有了该子过程所有外层过程定义的变量，常量和过程，并且还产生了一部分目标代码；对于后者，将主程序处理语句开始的地址叫做程序的入口，由于嵌套过程的存在，导致寻找程序入口有些麻烦，在这里用到了回填技术:

在进入 block 的第一步就先产生一个无条件跳转指令，由于中间子过程嵌套还会产生目标代码 ，导致不知道跳转到哪里才是主程序的入口，这里先记下这个无条件跳转指令在目标代码中的位置，当程序分析完变量说明、常量说明和过程说明即将进入语句处理的分析时，这时目标代码的地址就是主程序的入口，根据刚才记下的无条件跳转指令的位置将这个地址回填到跳转指令的目标地址上，从而目标代码第一句就是跳转到主程序入口处的指令，对于程序中嵌套程序，用这个方法依然是可行的；

除此之外，在进行其他任何分析之前，还要记录下主程序或者当前过程的数据量，即符号表中的变量数目，也即 address，以便后面活动记录的划分可以在结束过程后返回到原来的数据栈的位置；

另外，如果是从过程进入到 block，在产生过程调用指令需要用到这个入口地址，所以这个过程入口地址也要记录下来，用来填写过程调用指令的目标位置。则在进行其他任何分析之前，此时符号表中最新的一项必定是该过程的相关信息，可以将本过程的 value 值设为其入口地址，因为 value 值对于符号表 procedure 来说是无用的，即将开始地址登入到符号表，从而在做任何分析前也要记录下符号表中的最新项（记录其序号），在经过变量说明分析、常量说明

分析、过程分析后即将进入语句处理分析时，便得到了该过程的入口地址，将其登录到刚刚记录的那个符号表最新项的 value 域；可以通过判断符号表中的最新项的序号是否为 0 来判断是从过程进入<block>还是从过程进入<block>

进行完上述三步操作后即可进入语句的处理部分；

在语句的处理部分完成后，生成目标代码的退出过程或主程序的语句，然后恢复 address 和符号表的序号。

（2）<body>

此部分不直接产生目标代码

（3）<statement>

<statement>中有赋值语句(<id>:=<exp>)、if 语句、while 语句、call 语句、read 语句、write 语句，下面分别说明其翻译模式：

赋值语句：

对于每一个过程或主程序，都要开辟 3+变量个数的存储空间，其中 3 个是用来存放 SL，DL，RA 的，遇到一个变量就生成 STO L A 目标代码，其中 L 和 A 据可由符号表中存储的相关信息获得，意思是将此时数据栈栈顶的元素，通过层差 L（调用位置和定义位置之差）和偏移地址 A 找到这个变量在其定义的过程开辟的空间中为该变量开辟的存储空间，然后将栈顶元素存到这个位置。

if 语句：

if <lexp>
    产生条件转移指令“JPC，0，0”；
    用记录 cx1 上面那条条件转移指令的位置；
then
    <statement>
    产生无条件转移指令“JMP，0，0”；
    用 cx2 记录上面那条无条件转移指令的位置；
    cx3 为即将产生的目标代码的位置；
    用 cx3 回填 cx1 和 cx2 记录位置的指令
else
    <statement>
    cx4 为即将产生的目标代码的位置；
    用 cx4 回填 cx2 记录位置的指令

while 语句：

while<lexp>do
    产生条件转移指令“JPC，0，0”
    用 cx1 记录下上面那条条件转移指令的位置
<statement>
    产生无条件转移指令“JMP,0,0”
    cx2 为即将产生的目标代码的位置
    用 cx2 回填 cx1 记录位置的指令

call 语句：

产生“CAL，L，A”，意思是通过层差 L 和过程入口地址 A，找到过程的入口地址，这里 A 可以有该过程在符号表中的 value 域的值获得（原因见<block>的翻译）

read 语句：

首先从命令行获取一个值放在栈顶，然后将栈顶值赋值给相应变量(赋值过程见前面赋值语句的翻译)

write 语句：

直接输出栈顶值

3 程序结构说明

3.1 代码结构说明

LexAnalysis+ RValue+ chTable 完成词法分析。
AllPcode+PerPcode 记录生成的目标代码。
SymTable+TableRow 进行符号表管理。
MpgAnalysis 的一个函数 showError 完成错误处理程序。
MpgAnalysis 综合上述类完成编译功能。
具体结构如图 4、5 所示

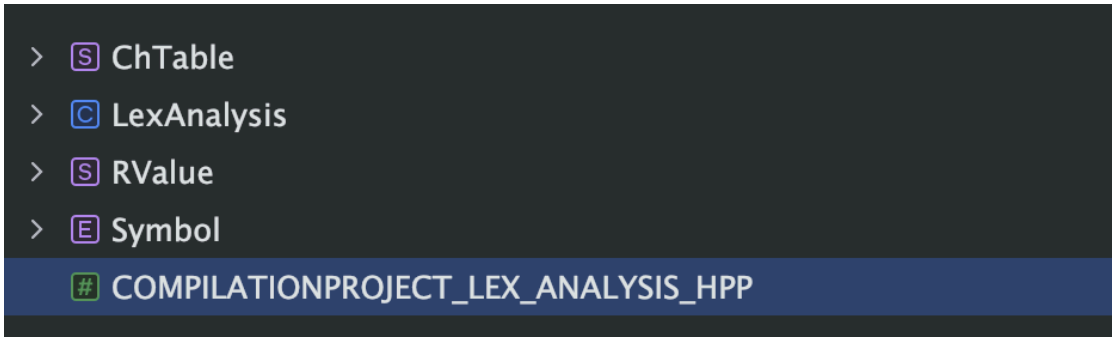

图 5 lex_analysis 代码结构图

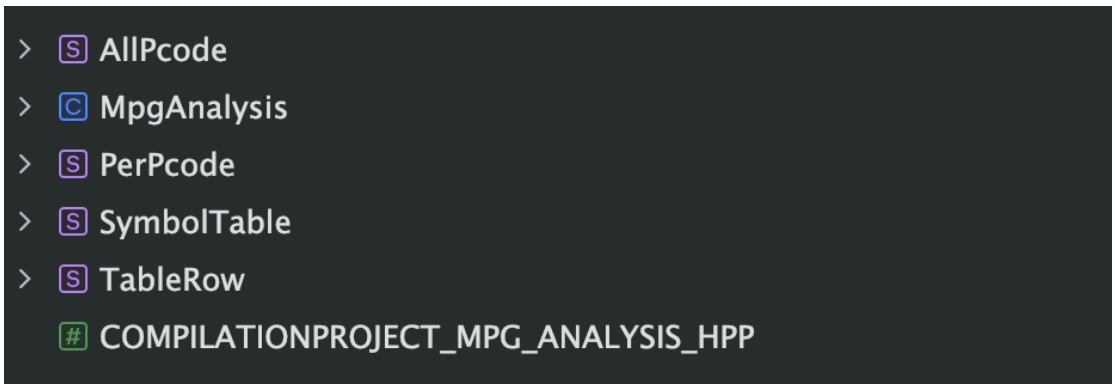

图 6 mpg_analysis 代码结构图

3.2 类接口说明

LexAnalysis 类对外提供 bAnalysis（）接口用于循环识别出所有字符并输出到文件 lex.txt 中

MpgAnalysis 类对外提供 mgpAnalysis()用于完成源代码编译，showPcodeInStack()用于向控制台输出类 P-code 目标代码。

```
class LexAnalysis {
public:
    LexAnalysis(std::string _filename);

    /**
     * 循环识别出所有字符并输出到文件lex.txt中
     */
    void bAnalysis();

private:
    std::string filename;
    const ChTable ct;
    RValue rv;
    const std::vector<std::string> keyWord = ct.keyWord;
    std::vector<std::string> symTable = ct.symTable;
    const int symLength = symTable.size();
    std::vector<std::string> constTable = ct.constTable;
    const int conLength = constTable.size();
    char ch = ' ';
    std::string strToken;
    std::string buffer;
    int searchPtr = 0;
    int line = 1;
    bool errorHappen = false;
```

图 7 LexAnalysis 类定义

```
class MpgAnalysis {
public:

    explicit MpgAnalysis(std::string filename);

    bool mgpAnalysis();

    void showPcodeInStack();

private:
    const int RVLENGTH;
    RValue *const rv;
    LexAnalysis lex;
    int terPtr = 0;
    bool errorHapphen = false;
    int address = 0;
    int level = 0;
    AllPcode Pcode;
    SymbolTable STable;
```

图 8 MpgAnalysis 类定义

3.3 测试类说明

测试类使用命令行参数，接受两个参数，例如：compiler <filename>，解析源代码文件<filename>，调用 MpgAnalysis 类提供的 public 方法，生成 P-code 目标代码并输出。具体如图 9 所示

```cpp
int main(int argc, char *argv[]) {
    if (argc < 2) {
        std::cerr << "Usage: " << argv[0] << " <filename>" << std::endl;
        return 1;
    } else {
        std::string filename = argv[1];
        MpgAnalysis mp(filename);
        if (!mp.mgpAnalysis()) {
            std::cout << ">compile succeed!" << "\n";
        }
        mp.showPcodeInStack();
    }
}
```

图 9 测试代码

4 测试结果

测试文件在 testPL0 中给出，具体如图 10 所示。

图 10 测试文件

使用命令行参数运行测试，其中 testPL1.txt 测试结果如图 11 所示。

```
/Users/changgu/CLionProjects/CompilationProject/cmake-build-debug/CompilationProject testPL1.txt
>compile succeed!
JMP  0 1
INT  0 5
OPR  0 16
STO  0 3
OPR  0 16
STO  0 4
LOD  0 3
LIT  0 1
OPR  0 2
STO  0 3
LOD  0 4
LIT  0 2
OPR  0 2
STO  0 4
LOD  0 3
OPR  0 14
LOD  0 4
OPR  0 14
OPR  0 15
OPR  0 0
```

图 11 测试结果截图

代码已上传到 github：
https://github.com/ChangGu328/CompilationProject.git