

**编译实践-PL\0编译系统实现**

|  |  |
| --- | --- |
| 姓名： | 杨承昊 |
| 专业： | 软件工程 |
| 学院： | 软件学院 |
| 提交时间： | 2018年1月1日 |

北京航空航天大学·软件学院

**编译实践-PL\0编译系统实现**

[一. 实验要求 2](#_Toc502580822)

[二. 实验原理 3](#_Toc502580823)

[2.1 PL/0语言描述 3](#_Toc502580824)

[2.2 PL/0语言文法的EBNF表示 3](#_Toc502580825)

[2.3 PL/0语言的语法图描述 4](#_Toc502580826)

[2.4 PL/0编译系统结构 5](#_Toc502580827)

[2.4.1 PL/0编译程序的词法分析（对应实现PL0LexParser类） 5](#_Toc502580828)

[2.4.2 PL/0编译程序的符号表管理（对应实现GrammarAnalyser类的symbol\_table\_stack函数族） 7](#_Toc502580829)

[2.4.3 PL/0编译程序的语法分析（对应实现GrammarAnalyser类） 7](#_Toc502580830)

[2.4.4 PL/0编译程序的目标代码结构和代码生成（嵌入GrammarAnalyser类的实现代码中） 10](#_Toc502580831)

[2.4.5 PL/0编译程序的语法错误处理 13](#_Toc502580832)

[2.4.6 编译程序的目标代码解释执行和存储分配（对应Interpreter类） 13](#_Toc502580833)

[三. 实验设计 18](#_Toc502580834)

[四. 实验总结 18](#_Toc502580835)

## 一. 实验要求

* 1. 以个人为单位进行开发，不得多人合作完成。
  2. 细节要求：
* 文件输入：符合PL/0文法的源程序（自己要有5个测试用例，包含出错的情况，还要用老师提供的测试用例进行测试）
* 输出：P-Code
* 错误信息：参见教材第316页表14.4（新书第411页表17.4）。
* 错误信息尽量详细（行号，错误类型）
* P-Code指令集：参见教材第351页表15.14。
* 语法分析部分要求统一使用递归下降子程序法实现。
* 编程语言自定，可使用C、C++、C#或Java等。（尽量选用有可执行文件的）
* 上交材料中不但要包括源代码（含注释）和可执行程序，还应有完整文档。

## 二. 实验原理

### 2.1 PL/0语言描述

PL/0语言是一种类PASCAL语言，是教学用程序设计语言，它比PASCAL语言简单，作了一些限制。PL/0的程序结构比较完全，赋值语句作为基本结构，构造概念有：

1）顺序执行、条件执行和重复执行，分别由begin/end,if then else和while do语句表示。

2）PL0还具有子程序概念，包括过程说明和过程调用语句。

3）在数据类型方面，PL0只包含唯一的整型，可以说明这种类型的常量和变量。

4）运算符有+，-，\*，/，=，<>，<，>，<=，>=，(，)。

5）说明部分包括常量说明、变量说明和过程说明。

### 2.2 PL/0语言文法的EBNF表示

<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> ::= <标识符>=<无符号整数>

<无符号整数> ::= <数字>{<数字>}

<标识符> ::= <字母>{<字母>|<数字>}

<变量说明部分>::= var<标识符>{,<标识符>};

<过程说明部分> ::= <过程首部><分程序>；

<过程首部> ::= procedure<标识符>;

<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>|<重复语句>|<空>

<赋值语句> ::= <标识符>:=<表达式>

<表达式> ::= [+|-]<项>{<加法运算符><项>}

<项> ::= <因子>{<乘法运算符><因子>}

<因子> ::= <标识符>|<无符号整数>|'('<表达式>')'

<加法运算符> ::= +|-

<乘法运算符> ::= \*|/

<条件> ::= <表达式><关系运算符><表达式>|odd<表达式>

<关系运算符> ::= =|<>|<|<=|>|>=

<条件语句> ::= if<条件>then<语句>[else<语句>]

<当型循环语句> ::= while<条件>do<语句>

<过程调用语句> ::= call<标识符>

<复合语句> ::= begin<语句>{;<语句>}end

<重复语句> ::= repeat<语句>{;<语句>}until<条件>

<读语句> ::= read'('<标识符>{,<标识符>}')'

<写语句> ::= write'('<标识符>{,<标识符>}')'

<字母> ::= a|b|...|X|Y|Z

<数字> ::= 0|1|2|...|8|9

### 2.3 PL/0语言的语法图描述



图1-1 程序语法描述图



图1-2 分程序语法描述图



图1-6 项语法描述图



图1-7 因子语法描述图

### 2.4 PL/0编译系统结构



图 1-8 PL/0编译程序和解释执行过程

这里我将PL/0的词法分析单独设计为一趟(pass)（对应实现为PL0LexParser类）,语法分析、符号表管理、出错处理和代码生成单独设计为一趟(pass)（对应实现为GrammarAnalyser类）。为了运行目标代码，我还设计解释程序（对应实现Interpreter类）

#### 2.4.1 PL/0编译程序的词法分析（对应实现PL0LexParser类）

PL/0编译系统中所有的字符，字符串的类型为，如下表格：

|  |  |
| --- | --- |
| 保留字 | begin, end, if,then, else, const, procedure,  var,do,while, call,read, write, repeat, until |
| 算数运算符 | + ,—，\*，/ |
| 比较运算符 | <> , < ,<= , >, >= ,= |
| 赋值符 | := , = |
| 标识符 | 变量名，过程名，常数名 |
| 常数 | 10,25等整数 |
| 界符 | ‘，’，‘.’，‘；’，‘（’，‘）’ |

PL/0的词法分析程序PL0LexParser类先分析整个源代码，将源代码转换为词法单元描述序列，主要功能为：

* 跳过空格字符。
* 识别单词符号，返回单词类型（按照在PL0LexParser. BOUNDSYM,PL0LexParser. OPERATORS,PL0LexParser. KEYWORDS定义的编译系统的字符判定单词类型）
* 特别的，对于编译系统的保留字符（例如：const, if, then等）需要查找系统的保留字符表PL0LexParser. KEYWORDS，为了加快查找速度，采用C#内置的HashSet进行检索（底层实现是Hash算法，效率非常之高，远比排序要强要稳定）
* 另外，如果读取的字符为数字，需要将该字符转换成整数值（调用公式num = 10 \* num + (ch - '0');），再存入符号表的Value区域.
* 采用“单符号先行”技术，在识别完每个符号的类型后，必须再度入下一个字符，以保证下一次再调用getsym()时，PL0LexParser.index保存的是指向该符号的首字符的指针



图1-9 词法分析程序的状态转换图

#### 2.4.2 PL/0编译程序的符号表管理（对应实现GrammarAnalyser类的symbol\_table\_stack函数族）

1. **符号表结构**：为了统一各种数据类型的符号表条目，方便管理，我将所有的符号条目定位如下所示的三元组：

（NAME,VARIABLE\_TYPE,VALUE）

至于变量名和过程名定位所需要的二元地址（BL,ON），在语法分析扫描到变量名和过程名的时候根据栈式符号表动态计算，这样就可以节省表的内容，也大大简化了维护的过程。

2. **符号表管理函数**：由于符号表管理与语法分析和中间代码生成密切相关，所以将符号表管理函数做成语法分析与中间代码生成对应类GrammarAnalyser的成员函数：

string build\_entry(string name,string variable\_type,string value)//构造符号表条目

bool add\_to\_symbol\_table\_stack(string entry)//将符号表条目加入栈式符号表

string query\_symbol\_table\_stack(string name,string type)//查询栈式符号表，并返回相应的信息，相应的信息封装在返回的字符串中，以\r分割

#### 2.4.3 PL/0编译程序的语法分析（对应实现GrammarAnalyser类）



图1-10 语法调用关系图

采用不带回溯的递归子程序法，对于语言的文法要求：

1) 该文法必须是非左递归。

2) 文法的非终结符，其规则右部所生成的first集合两两不相交

3) 若文法具有形如，则

递归子程序设计实例:

|  |
| --- |
| * <expression>::=[+|-]<term>{(+|-)<项>}   bool expr()  {  bool first = false;  bool neg = false;  if (sym\_list[index][0]=="+" || sym\_list[index][0]=="-")  {  if (sym\_list[index][0]=="-")  {  neg = true;  //gen("OPR", 0, 1);  }  //gen("OPR",0,1);  index += 1;  first = true;  }  int backup = index;  if (term())  {  if (neg)  gen("OPR", 0, 1);  first = true;  while (index < sym\_list.Count() && sym\_list[index][0] == "+" || sym\_list[index][0] == "-")  {  int cx0 = index;    index += 1;  bool flag = term();  if (!flag)  {  error\_message += "(Error Code 34)项定义错误(line:" + sym\_list[backup][3] + ")\r\n";  return false;  }  if (sym\_list[cx0][0] == "+")  gen("OPR", 0, 2);  else if (sym\_list[cx0][0] == "-")  gen("OPR", 0, 3);  //index += 1;  }  //have increased index by 1  return true;  }  else  {  if (!first)  {  error\_message += "(Error Code 24)表达式不能以此开头(line: " + sym\_list[backup][3] + ")\r\n";  }  else  {  error\_message += "(Error Code 33)表达式首项定义错误(line: " + sym\_list[backup][3] + ")\r\n";  }  }  return false;  } |

#### 2.4.4 PL/0编译程序的目标代码结构和代码生成（嵌入GrammarAnalyser类的实现代码中）

* + **代码结构**

P-code 语言：一种栈式机的语言。此类栈式机没有累加器和通用寄存器，有一个栈式存储器，有四个控制寄存器（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），算术逻辑运算都在栈顶进行。

|  |  |  |
| --- | --- | --- |
| F | L | A |

指令格式

F ：操作码

L ：层次差 （标识符引用层减去定义层）

A ：不同的指令含义不同

表5 P-code 指令的含义

|  |  |
| --- | --- |
| 指令 | 具体含义 |
| LIT 0, a | 取常量a放到数据栈栈顶 |
| OPR 0, a | 执行运算，a表示执行何种运算(+ - \* /) |
| LOD l, a | 取变量放到数据栈栈顶(相对地址为a,层次差为l) |
| STO l, a | 将数据栈栈顶内容存入变量(相对地址为a,层次差为l) |
| CAL l, a | 调用过程(入口指令地址为a,层次差为l) |
| INT 0, a | 数据栈栈顶指针增加a |
| JMP 0, a | 无条件转移到指令地址a |
| JPC 0, a | 条件转移到指令地址a |

|  |
| --- |
| PCODE类的结构：以字符串形式存储，存储如下三元组，以’\r’连接：  （COMMAND,FIRST\_PARAM,SECOND\_PARAM）  存放虚拟机代码的列表：Grammar.codes  //生成虚拟机代码  void gen(string command,int l,int a)  {  if (!procedure\_pcode\_list.ContainsKey(now\_ptr))  {  procedure\_pcode\_list.Add(now\_ptr, new List<string>());  }  procedure\_pcode\_list[now\_ptr].Add(command + " "+l.ToString()+" " + a.ToString() + "\r\n");  codes.Add(command + " " + l.ToString() + " " + a.ToString() + "\r\n");  } |

* + 代码生成与地址返填

对于if then [else],while do和repeat until语句，要生成跳转指令，故采用地址返填技术。

* + - if-then-else语句的目标代码生成模式：

|  |  |
| --- | --- |
| if <condition> then <statement>[else] | |
|  | <condition> |
|  | JPC addr1 |
|  | <statement> |
| addr1: | [else] |

* + - while-do语句的目标代码生成模式：

|  |  |
| --- | --- |
| while <condition> do <statement> | |
| addr2: | <condition> |
|  | JPC addr3 |
|  | <statement> |
|  | JPC addr2 |
| addr3: |  |

* + - repeat-until语句的目标代码生成模式：

|  |  |
| --- | --- |
| repeat <statement> until <condition> | |
| addr4: | <statement> |
|  | <condition> |
|  | JPC addr4 |

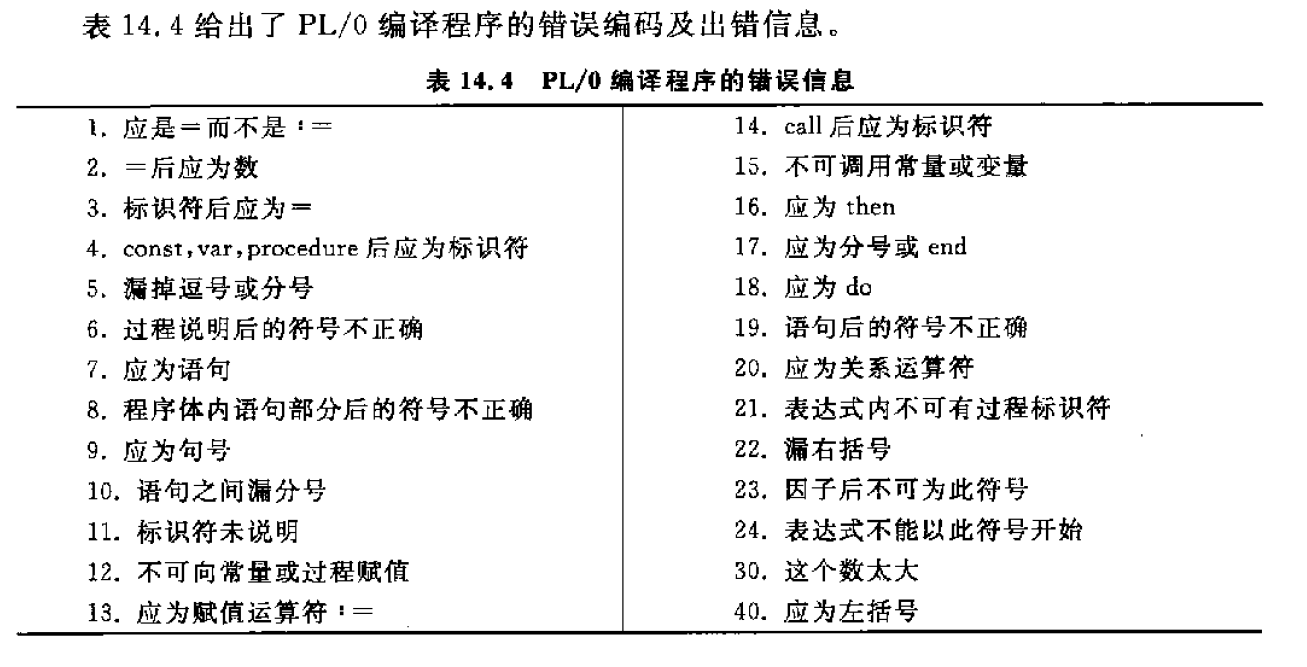
|  |
| --- |
|  |

#### 2.4.5 PL/0编译程序的语法错误处理

错误处理的原则

* + 尽可能准确指出错误位置和错误属性
  + 咨询了邵兵老师，本次实践无需做错误处理的校正，遇到错误直接汇报即可，不必往下继续分析。

PL/0编译系统中，所定义的错误类型如下列举：



#### 2.4.6 编译程序的目标代码解释执行和存储分配（对应Interpreter类）

* 类pcode解释器的结构

1. . 目标代码存放在数组GrammarAnalyser.Codes中
2. . 定义一维整型数组Interpreter. datastack作为运行栈
3. .栈顶寄存器（指针）t;
4. .基址寄存器（指针）b;
5. .程序地址寄存器 p;
6. .指令寄存器 index.

* 运行栈的存储分配

1. . SL:静态链，指向定义该过程的直接外过程(或主程序)运行时最新数据段的基地址。
2. .DL:动态链，指向调用该过程前正在运行过程的数据段基地址。
3. .RA:返回地址，记录调用该过程时目标程序的断点，即调用过程指令的下一条指令的地址

例如，假定有过程 A，B，C，其中过程 C 的说明局部于过程 B，而过程 B 的 说明局部于过程 A，程序运行时，过程 A 调用过程 B，过程 B 则调用过程 C，过 程 C 又调用过程 B，如下图所示：

图9-1过程说明嵌套图 过程调用图 表示 A 调用 B

从静态链的角度我们可以说A是在第一层说明,B是在第二层说明,C则是在第三层说明。

若在B中存取A中说明的变量a,由于编译程序只知道A,B间的静态层差为1,如果这时沿着动态链下降一步,将导致对C的局部变量的操作。

为防止这种情况发生，设置第二条链，将各个数据区连接起来。我们称之为动态链（dynamic link）DL。

这样，编译程序所生成的代码地址，指示着静态层差和数据区的相对修正量。下面是过程 A、B 和 C 运行时刻的数据区图示：



附录：

样例测试

|  |  |
| --- | --- |
| //test.pl0 | //generated p-code |
| const z=0;  var head,foot,cock,rabbit,n;  begin  n := z;  cock := 1;  head:= 3;  foot:=8;  while cock <= head do  begin  rabbit :=head-cock;  if cock\*2+rabbit\*4=foot then  begin  write(cock,rabbit);  n:=n+1  end;  cock:=cock+1  end;  if n=0 then write(0,0)  end. | JMP 0 1  INT 0 8  LIT 0 0  STO 0 4  LIT 0 1  STO 0 2  LIT 0 3  STO 0 0  LIT 0 8  STO 0 1  LOD 0 2  LOD 0 0  OPR 0 12  JPC 0 42  LOD 0 0  LOD 0 2  OPR 0 3  STO 0 3  LOD 0 2  LIT 0 2  OPR 0 4  LOD 0 3  LIT 0 4  OPR 0 4  OPR 0 2  LOD 0 1  OPR 0 7  JPC 0 37  LOD 0 2  WRT 0 0  LOD 0 3  WRT 0 0  LOD 0 4  LIT 0 1  OPR 0 2  STO 0 4  JMP 0 37  LOD 0 2  LIT 0 1  OPR 0 2  STO 0 2  JMP 0 10  LOD 0 4  LIT 0 0  OPR 0 7  JPC 0 51  LIT 0 0  WRT 0 0  LIT 0 0  WRT 0 0  JMP 0 51  OPR 0 0 |

## 三. 实验设计

为了提高程序的可维护性和可扩展性，我将词法分析单独做成一趟，语法分析+符号表管理+错误处理+中间代码生成单独做成一趟，解析器单独做成一个模块，GUI单独做成一个模块，这样的设计使得分模块测试和修复变得容易，而且系统运行稳定性也提高了。

我的设计基于这样的理念：尽可能多的**模块化**。这主要是和递归下降子程序算法的特点相符，而且这也能带来**解耦**的好处。举例而言，我将符号表管理拆成建条目、加条目、查条目，并且符号表中不保存层次，偏差等信息，仅在查表的时候计算，这样使得符号表维护更为方便而灵活。

另外，我在程序中大量使用字符串来表示表格条目信息，用’\r’隔开，这样使得表格列项可以是不固定的，不对齐的，可以根据不同的类型做调整，而且也避免了定义复杂的数据类型，降低程序的运行效率和可读性。

使用C#的WPF编程是因为C#的WPF提供了丰富的图形界面接口，而且在windows平台有良好的兼容性。

关于错误处理，我是这么考虑的：其实多数情况下，修改编译错误的时候，程序员一次只关注一个错误，而且编译错误的程序不应该给予运行的机会，所以我的错误处理是：尽可能的把相关的错误信息都报出来，不往下继续编译，也不允许执行，

关于交互式程序，我觉得频繁弹窗不是一个良好的用户体验，应该设计成批处理的模式，让用户一次性输入全部的输入，这样可以在不打扰用户的情况下完成所有的计算工作，更充分的利用计算资源。

## 四. 实验总结

通过本次实验，我对词法分析，语法分析，符号表管理，出错处理，中间代码生成和解释器的运行原理有了更为深刻的理解，加强了自己的动手能力，体会到了要将理论付诸实践需要做出的巨大努力。