PIO 编译器

2020. 06. 26

编译原理

目录

1 成果概览

1.1 理解编译器的工作机制，掌握编译器的工作原理

1.2 具体编译器的实现细节

输入以及输出信息

- 输入：符合含扩展 PL/0 文法的源程序
- 输出：P-Code 文件，并且 interpret.cpp 程序可以执行。
- 错误信息：输出词法、语法、语义分析过程中遇到的错误。
- P-Code 指令集：在原本基础上做了一点扩展。
  - 增加了第四个参数，用来传递变量的数值以及类型。

编程实现

- 词法、语法分析部分要求统一使用 lex，yacc 源程序实现。
- 编程语言使用 C++。

PIO 语法的扩展部分

- 支持所有语法成分（含扩展成分）
  - 包含 repeat，for 语句
- 支持 int,float,char,string 类型的变量
  - Int，float 类型支持混合运算，编译器会进行类型转换
- 最多允许三层嵌套过程
- 支持并列过程
- 允许递归调用
- 支持数组
  - 数组支持 int,float 类型，并且支持多维数组

1.3 编译系统的结构

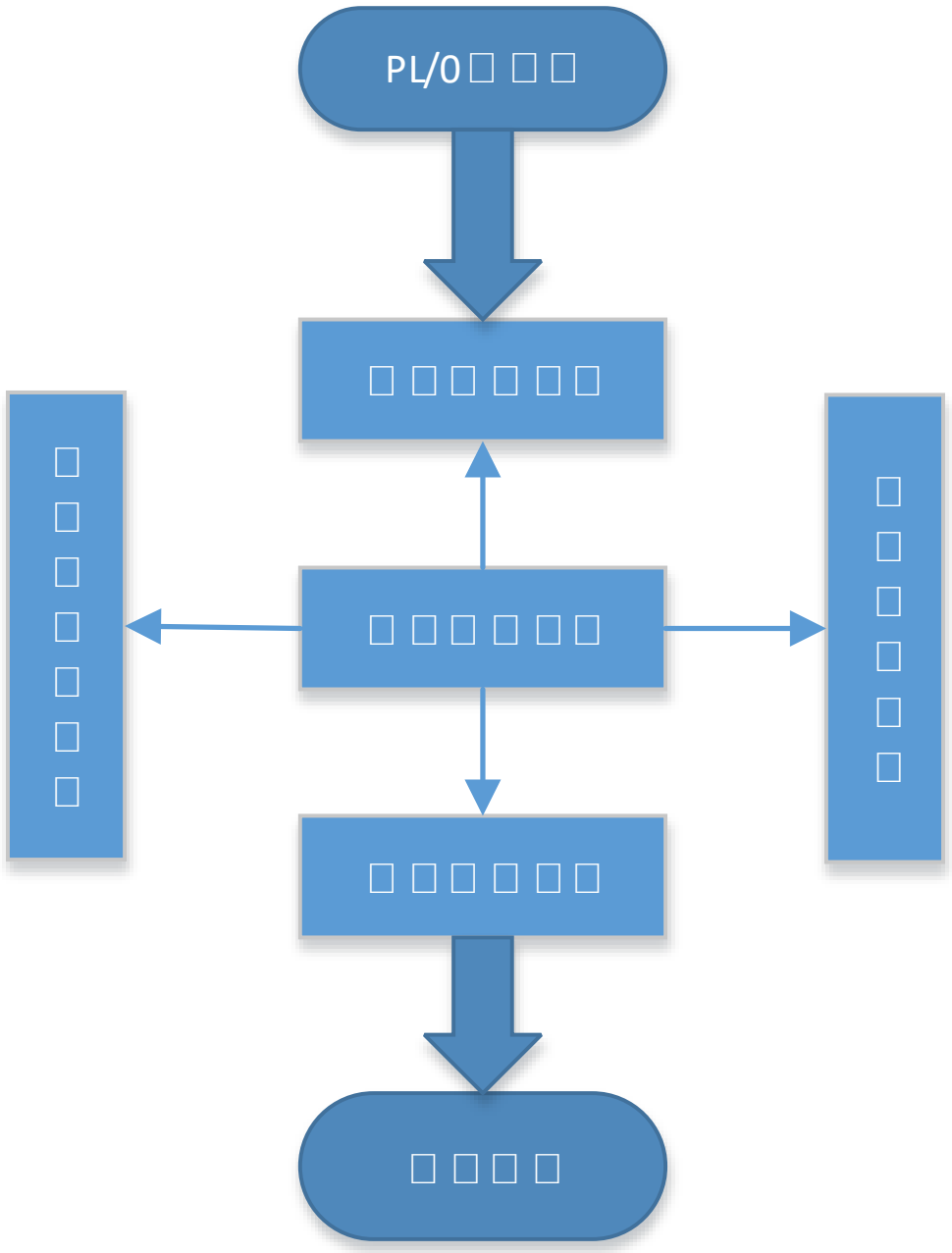

Figure 1: PL0 编译程序执行过程

具体程序的结构关系

```
                --> define.cpp          实现辅助数据结构与辅助函数
                --> define.h            声明辅助数据结构与辅助函数
main.cpp --------> LexicalPl0.l         词法分析器
                --> SyntaxPl0.y         语法、语义分析器
                --> Interpret.cpp       解释执行 pcode
```

简要介绍 main.cpp 的流程

- init()：初始化语法分析和语义分析的数据结构
- yyparse()：lex 和 yacc 源程序的接口，开始执行词法分析、语义和语法分析程序
- output_pcode()：将编译程序产生的 pcode 输出到指定的文件中
- interpret()：调用 interpret.cpp 程序解释并执行 pcode

2 含扩展的 PL0 语言的描述

2.1 PL0 语言文法的 EBNF 表示

EBNF 范式的符号说明

< >：表示语法构造成分，为非终结符

::= ：该符号的左部由右部定义，读作“定义为”

| ：表示或

{ }：表示括号内的语法成分可重复

[ ]：括号内成分为任选项

( )：圆括号内成分优先

与标准 PL0 比做出改变的标为红色

<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>=<整数>|<浮点数>

<整数> ::= <数字>{<数字>}

<变量说明部分> ::= VAR<标识符>[(:<类别> [ARRAY \[<无符号整数>..<无符号整数>{,<无符号整数>..<无符号整数>}\] OF <类别>]) | ((,<标识符>):<类别>)]{ (:<类别> [ARRAY \[<无符号整数>..<无符号整数>{,<无符号整数>..<无符号整数>}\] OF <类别>]) | ((,<标识符>):<类别>) };

<类别>::=INT, FLOAT

<标识符> ::= <字母>{<字母>|<数字>}

<过程说明部分> ::= <过程首部><分程序>{;<过程说明部分>};

<过程首部> ::= PROCEDURE<标识符>;

<语句> ::= <赋值语句>|<复合语句>|<条件语句>|<当型循环语句>|<过程调用语句>|<读语句>|<写语句>|<空>|<FOR 循环语句>|<REPEAT 循环语句>

<赋值语句> ::= <标识符>:=<表达式>

<复合语句> ::= BEGIN<语句>{;<语句>}END

<条件> ::= <表达式><关系运算符><表达式>|ODD<表达式>

<条件语句> ::= IF<条件>THEN<语句>

<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> ::= *|/

<关系运算符> ::= =|#|<|<=|>|>=

<当型循环语句> ::= WHILE<条件>DO<语句>

<过程调用语句> ::= CALL<标识符>

<读语句> ::= READ’(‘<标识符>{,<标识符>}’)’

<写语句> ::= WRITE’(‘<表达式>{,<表达式>}’)’

<字母> ::= a|b|...|X|Y|Z

<数字> ::= 0|1|...|8|9

<FOR 循环语句>::= FOR<>:=<表达式>TO<表达式>DO<语句>

<REPEAT 循环语句>::= REPEAT<语句>UNTIL<条件>

<浮点数>::=[-]{数字}(.{数字})[[Ee][-+]{数字}]

<字符>::= ‘.’ （.想表达 ‘’ 中任何字符都可以，借用正则表达式里的通配符来表示）

<字符串>::=”{<字符>}”

注意：

数据类型：无符号整数

标识符类型 ：简单变量(var)和常数(const)

变量类型：整数(int)和浮点数(float)

数字位数：小于 14 位

标识符的有效长度：小于 10 位

过程嵌套：小于 3 层

2.2 PL0 语言的语法图描述

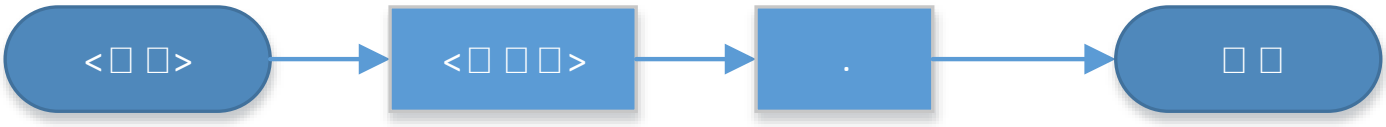

Figure 2: 程序语法描述图

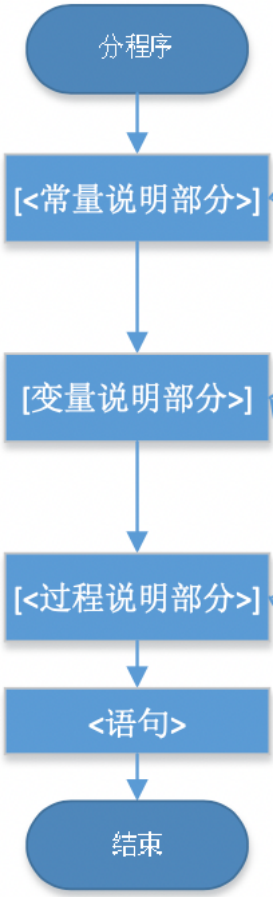

Figure 3: 分程序描述图

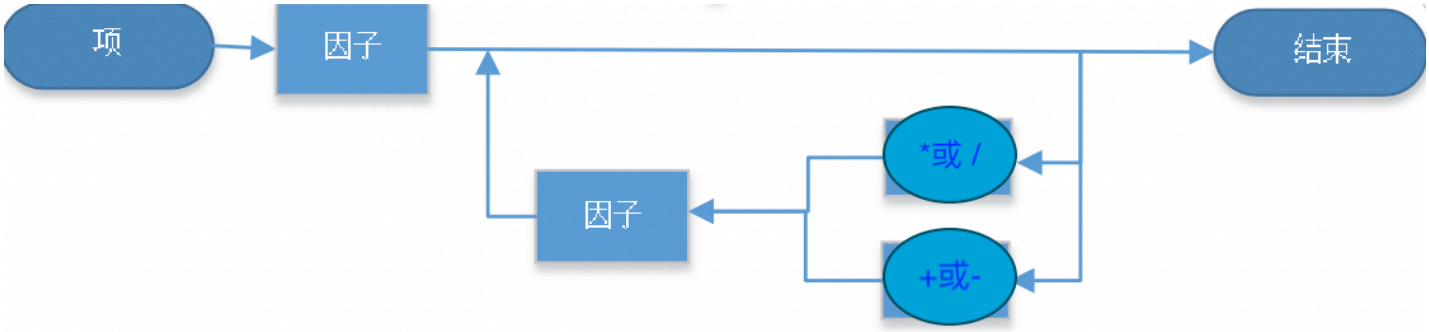

Figure 4: 项语法描述图

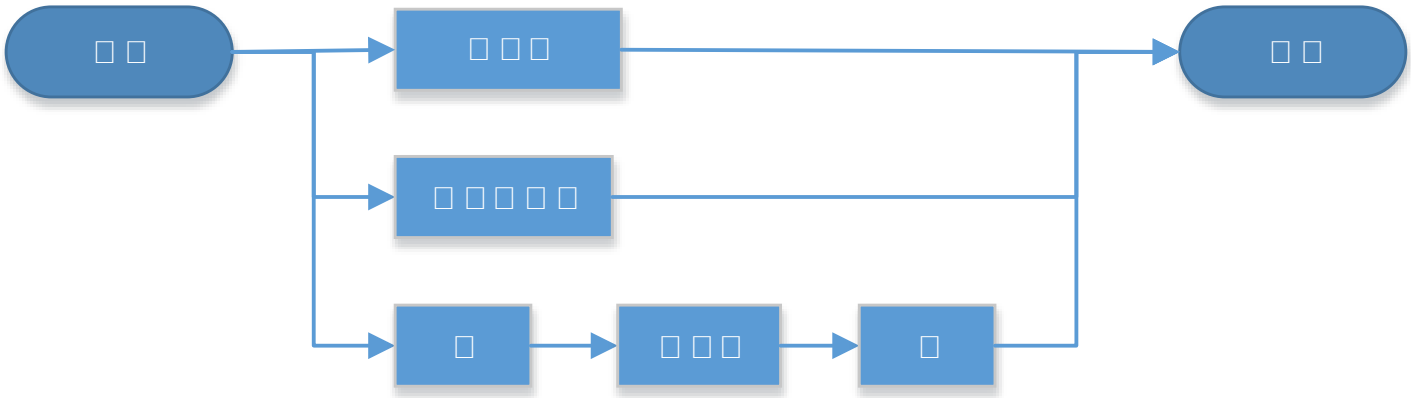

Figure 5: 因子语法描述图

3 PL0 编译程序的词法分析

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, int, float, char, string
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 1.0234, 1e4, 12.2e-2 等浮点数
界符	‘, ’, ‘.’, ‘; ’, ‘ (’, ‘) ’, ‘[’, ‘]’, ‘..’, ‘{’, ‘}’

PL/0 的词法分析程序 LexicalPl0.l 由 yyparse()接口调用

主要功能为：

- 跳过空格字符。
- 识别单词符号，返回单词类型
- 特别的，对于编译系统的保留字符（例如：const, if, then 等）返回 SyntaxPl0.y 中定义的 token
- 另外，如果读取的字符为数字，识别出数字的类型（int 或者 float），向 SyntaxPl0.y 中传递数字的值与类别.

报错部分：

- 所有非保留字、算数运算符、比较运算符、赋值符、标示符、常数、界符的符号均为 illegal
- Illegal 正则表达式定义为

  {illegal} (\'.{2,}\')
- 使用深红色将 illegal 内容的位置打印出来

```
{illegal} {
    //column+=yyleng;
    std::cout<<BOLDRED<<"Lexical error! Illegal input: row "<<row<<std::endl;
    exit(0);
}
```

- 对于 int 或者 float 类型的变量的数值部分长度过长也进行报错处理

```
{Integer}|-{Integer} {


    if(yyleng>14)
    {

        std::cout<<BOLDRED<<"Lexical error! Word length is overproof!\n";
        exit(0);
    }
         column+=yyleng;
         yylval.i_val=atoi(yytext);
         return INTEGER_VAL;
     }
     {Float}|-{Float} {


         if(yyleng>14)
         {

             std::cout<<BOLDRED<<"Lexical error! Word length is overproof!\n";
             exit(0);
         }
         column+=yyleng;
         yylval.f_val=atof(yytext);
         return FLOAT_VAL;
     }
```

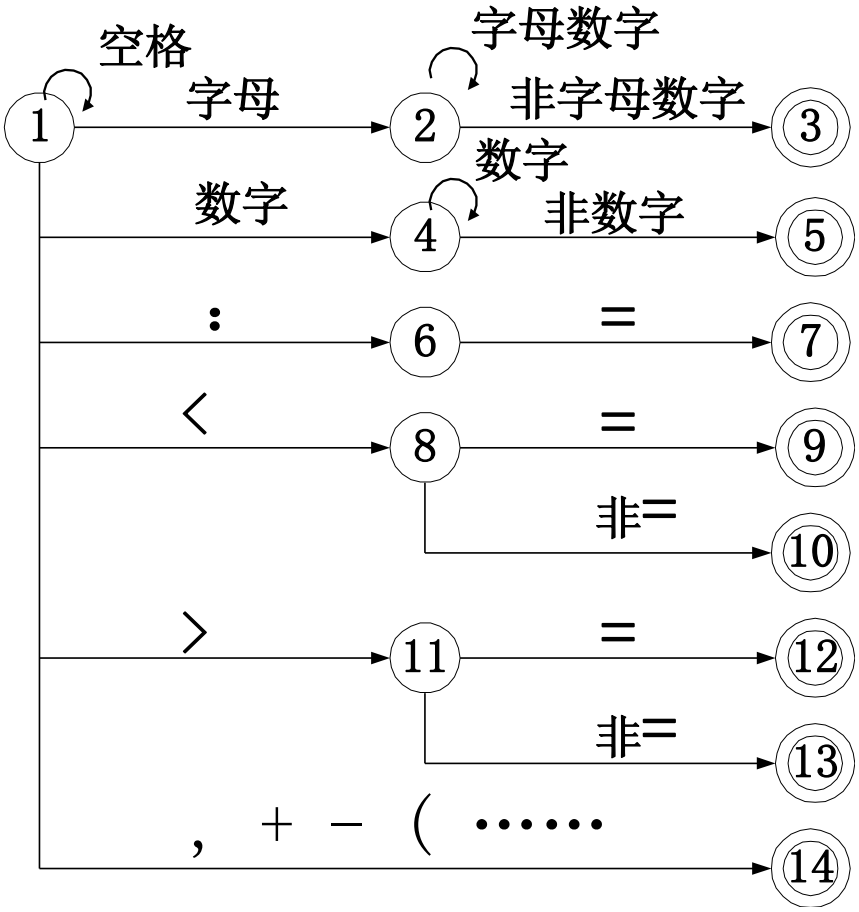

Figure 6: 词法分析程序的状态转换图

4 语法、语义分析部分

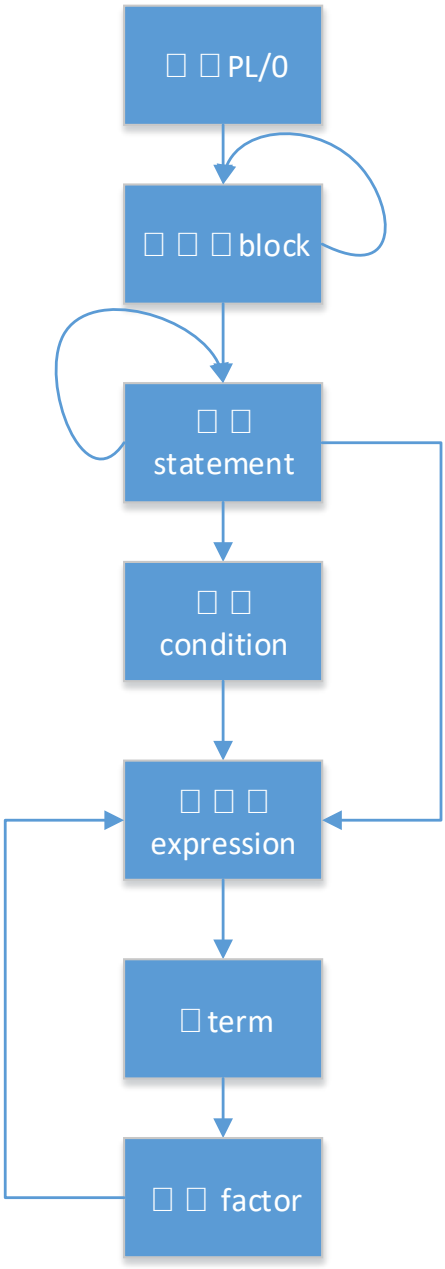

Figure 7: 语法层次关系图

4.1 核心语法介绍

左为 EBNF 范式
右为对应的语法规则

<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>

语法规则就是对 EBNF 范式进行了翻译，并且进行了语义动作的处理。

其中 BeforeDec 和 BeforeSta 部分对符号表和 display 表进行了处理。

```
Program     : SubPro DOT {
                all_type_val_value p;
                init_for_gen_pcode(p);
                gen(instruction::OPR,0,0,p);
              }
            ;
SubPro      : BeforeDec DeclarePart BeforeSta Statement {
              }
            | BeforeDec BeforeSta Statement {
              }
            ;
BeforeDec   : {
                if(symtable_size()!=0)
                {
                  display_top++;
                  display_stack[display_top]=symtable_size()+1;
                }
                backfill[++backfill_top]=code_line;

                all_type_val_value p;
                init_for_gen_pcode(p);
                gen(instruction::JMP,0,0,p);
              }
            ;
BeforeSta   : {
                if(backfill_top>-1)
                  backpatch(backfill[backfill_top--],code_line);

                all_type_val_value p;
                init_for_gen_pcode(p);
                if(display_stack[display_top]==1)
                  gen(instruction::INI,0,sym_top+3,p);
                else
                  gen(instruction::INI,0,sym_top-display_stack[display_top]+1+3,p);
              }
            ;
```

For 和 Repeat

<FOR 循环语句>::= FOR<变量>:=<表达式>TO<表达式>DO<语句>

<REPEAT 循环语句>::= REPEAT<语句>UNTIL<条件>

For 和 Repeat 的语义动作均使用了回填来实现。

```
755 While_Loop      : WHILE BeforeCond Condition DO BeforeCondDo Statement {
756                   all_type_val_value p;
757                   init_for_gen_pcode(p);
758                   gen(instruction::JMP,0,whilepos[whiletop--],p);
759                   backpatch(backPatch_table[backPatch_table_top--],code_line);
760                 }
761               ;
762 For_Loop        : FOR AssignS TO BeforeCond Condition DO BeforeCondDo Statement {
763                   all_type_val_value p;
764                   init_for_gen_pcode(p);
765                   gen(instruction::JMP,0,whilepos[whiletop--],p);
766                   backpatch(backPatch_table[backPatch_table_top--],code_line);
767                 }
768               ;
769 Repeat_Loop     : REPEAT BeforeRepeat Statement SEMI UNTIL Condition {
770                   all_type_val_value p;
771                   init_for_gen_pcode(p);
772                   gen(instruction::JPC,0,whilepos[whiletop--],p);
773                 }
774               ;
775 BeforeCond  : {
776                   whilepos[++whiletop]=code_line;
777                 }
778               ;
779 BeforeCondDo: {
780                   backPatch_table[++backPatch_table_top]=code_line;
781                   all_type_val_value p;
782                   init_for_gen_pcode(p);
783                   gen(instruction::JPC,0,0,p);
784                 }
785               ;
786
787 BeforeRepeat: {
788                   whilepos[++whiletop]=code_line;
789                 }
790               ;
```

数组

<变量说明部分> ::= (VAR<标识符>[(ARRAY \[<无符号整数>..<无符号整数>|<表达式>{,<无符号整数>..<无符号整数>|<表达式>}\]OF<类别 1>) | (:<类别 2>)]{,<标识符>[(ARRAY \[<无符号整数>..<无符号整数>|<表达式>{,<无符号整数>..<无符号整数>|<表达式>}\]OF<类别 1>) | (:<类别 2>)};)

```
IDefine     : IDENTIFIER OF VarType {
                if(if_declared($1)==0)
                {
                    struct all_type_val_value temp;
                    init_for_gen_pcode(temp);
                    temp.type_=all_type_val_value::define($3);
                    symtable_push($1,Symbol::var_,temp);
                }
              }
              | IDENTIFIER ARRAY ML_PAREN Dimension MR_PAREN OF ArrayType {
                if(if_declared($1)==0)
                {
                    array_stack_top_index++;
                    ary_stack[array_stack_top_index].id=$1;
                    ary_stack[array_stack_top_index].dimension=$4.dimension;
                    ary_stack[array_stack_top_index].type_=$7;
                    for(int index=0;index<$4.dimension;index++)
                    {
                      ary_stack[array_stack_top_index].low[index]=$4.low[index];
                      ary_stack[array_stack_top_index].high[index]=$4.high[index];
                    }
                    int s=$4.high[0]-$4.low[0]+1;
                    if($4.dimension>1)
                      for(int i=1;i<$4.dimension;i++)
                        s=s*($4.high[i]-$4.low[i]+1);
                    ary_stack[array_stack_top_index].size=s;

                    all_type_val_value temp;
                    init_for_gen_pcode(temp);
                    temp.type_=all_type_val_value::define($7);
                    for(int i=0;i<s;i++)
                      symtable_push($1,Symbol::var_,temp);
                }
              }

              ;
Dimension   : Dimension COMMA INTEGER_VAL DOTDOT INTEGER_VAL {
                $$.dimension=$1.dimension+1;
                $$.low[$1.dimension]=$3;
                $$.high[$1.dimension]=$5;
              }
            | INTEGER_VAL DOTDOT INTEGER_VAL {
                $$.dimension=1;
                $$.low[0]=$1;
                $$.high[0]=$3;
              }
            ;
```

报错处理：

在语义动作中，有检查语义是否正确的地方，如果出现错误，会输出错误类型。

例如图中(Figure 8)所示，在调用 call 函数时，如果 call 的 procedure 不存在，会报告错误。除此之外，在多处语义动作，例如：赋值、数组操作等，均设置了语义正确性检查和报错处理。语法处理中的报错均使用 Magenta 颜色，与词法中的 Bold Red 区别。

```
794  Call_Func       : CALL SL_PAREN IDENTIFIER SR_PAREN {
795                   if(find_pro($3)==-1)
796                   {
797                     std::cout<<MAGENTA<<"Semantic error! "<<$3<<" not found!"<<std::endl;
798                     exit(1);
799                   }
800
801                   int pos=find_pro($3);
802                   all_type_val_value p;
803                   init_for_gen_pcode(p);
804                   gen(instruction::CAL,current_Level-pro_stack[pos].level,pro_stack[pos].pos-1,p);
805                 }
806               ;
```

Figure 8: 语法中的报错处理

4.2 扩展后的类 pcode 代码

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	E

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

E ： 传递栈中元素(element)，有一些指令中没有用到

由于需要现实 int, float, string, char 这些类型的变量，所以元素的类型有四类，分别用三种对应的方式存储。

```
1.  struct stack_table_element
2.  {
3.      int var_int;
4.      float var_float;
5.      char* var_string;
6.      enum ele_type{int_=0,float_=1,char_=2,string_=3,undefine=-1};
7.      ele_type type_;
8.  };
9.
```

根据新增的第四个参数，对类 pcode 指令进行扩展

扩展后的 P-code 指令集

表 5 P-code 指令的含义

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

类 pcode 解释器的结构

- 一维整型数组 S 作为运行数据区
- 指令寄存器 I：存放当前正在解释的目标指令
- 栈顶寄存器 t：每个过程执行时，给它分配的数据区最新分配的单元地址
- 基址寄存器 b:每个过程被调用时，在数据区给它分配的数据段起始地址
- 指令地址寄存器 p：指向下一条要执行的目标程序的地址

类 pcode 运行时的存储空间分配

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：

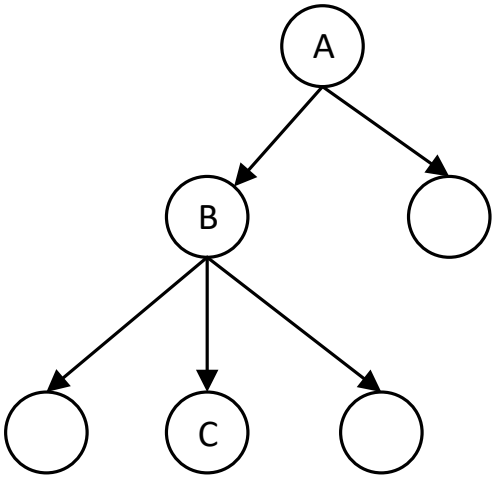

Figure 9: 过程说明嵌套图

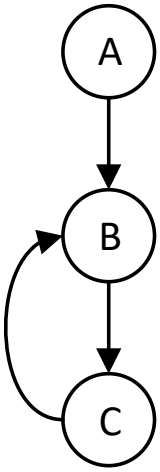

Figure 10: 过程调用图

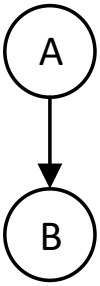

Figure 11: 表示A调用B

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:

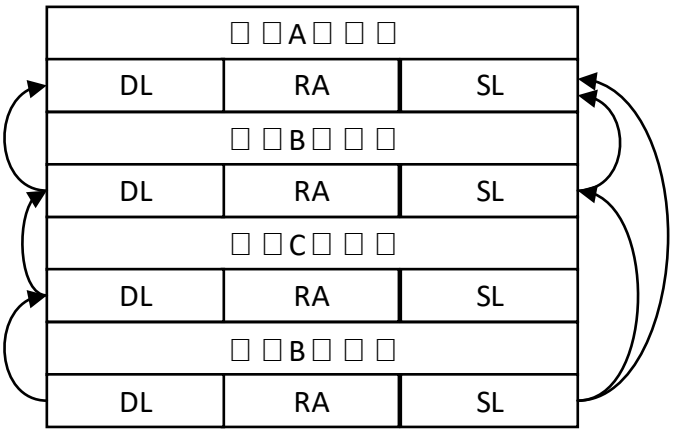

Figure 12

5 运行结果展示

```
sizihua@MacBookPro:pl0_compiler/src <master*>$ g++ main.cpp interpret.cpp lex.yy.c SyntaxPl0.tab.c define.cpp
clang: warning: treating 'c' input as 'c++' when in C++ mode, this behavior is deprecated [-Wdeprecated]
clang: warning: treating 'c' input as 'c++' when in C++ mode, this behavior is deprecated [-Wdeprecated]
sizihua@MacBookPro:pl0_compiler/src <master*>$ ./a.out  ../../succ_pl0/Sn.pl0 ../../pcode/Sn.pcode
start yyparse
start interpret pcode
2
2
24
sizihua@MacBookPro:pl0_compiler/src <master*>$ ./a.out  ../../succ_pl0/column.pl0 ../../pcode/column.pcode
start yyparse
start interpret pcode
23
32
138
1587
7590
50784
sizihua@MacBookPro:pl0_compiler/src <master*>$ ./a.out  ../../succ_pl0/daffodilnum.pl0 ../../pcode/daffodilnum.pcode
start yyparse
start interpret pcode
6
sizihua@MacBookPro:pl0_compiler/src <master*>$ ./a.out  ../../succ_pl0/function.pl0 ../../pcode/function.pcode
start yyparse
syntax error (9,1)
start interpret pcode
sizihua@MacBookPro:pl0_compiler/src <master*>$ ./a.out  ../../succ_pl0/gcd.pl0 ../../pcode/gcd.pcode
start yyparse
start interpret pcode
3
2
1
```

Figure 13: 部分测试集运行展示

下为测试样例 daffoilnum.pl0 的代码

```
1.  var n of int, m of int, i of int, j of int, digit array[1..3] of int;
2.
3.  procedure init;
4.  begin
5.      i := 1;
6.      while i < 4 do
7.          begin
8.          digit [i] := 0;
9.              i := i + 1;
10.         end;
11. end;
12.
13. procedure parse;
14. var tmp of int;
15. begin
16.   tmp := n;
17.     i := 0;
18.     while n != 0 do
19.         begin
20.             m := n / 10;
21.           i := i + 1;
22.           digit[i] := n - m * 10;
```

```
            n := m;
        end;
   n := tmp;
end;

procedure sumdigit;
begin
    m := digit[1] * digit[1] * digit[1] +
        digit[2] * digit[2] * digit[2] +
        digit[3] * digit[3] * digit[3];
    write(m);
end;

begin
    n := 1;
    while n < 10 do
        begin
            call (init);
            call (parse);
            call (sumdigit);
            if m == n then
                begin
                write(n);
                end;
            n := n+10;
        end;
end.
```

下为对应的产生的 pcode

由于只有 LIT 中第三个参数没有用处，第四个参数有用，并且其他的指令中第四个指令大部分没有用，或者是仅仅用来传递元素的类别，此处生成的 pcode 中，LIT 对应第三个参数改为第四个参数，剩下指令的第四个参数省略不输出。

```
JMP 0 82
JMP 0 2
INT 0 3
LIT 0 1
STO 1 5
LOD 1 5
LIT 0 4
OPR 0 10
JPC 0 17
LOD 1 5
LIT 0 0
STO 1 7
LOD 1 5
LIT 0 1
OPR 0 2
STO 1 5
JMP 0 5
OPR 0 0
JMP 0 19
INT 0 4
LOD 1 3
STO 0 3
LIT 0 0
STO 1 5
LOD 1 3
LIT 0 0
OPR 0 9
JPC 0 46
LOD 1 3
LIT 0 10
OPR 0 5
```

```
32. STO 1 4
33. LOD 1 5
34. LIT 0 1
35. OPR 0 2
36. STO 1 5
37. LOD 1 5
38. LOD 1 3
39. LOD 1 4
40. LIT 0 10
41. OPR 0 4
42. OPR 0 3
43. STO 1 7
44. LOD 1 4
45. STO 1 3
46. JMP 0 24
47. LOD 0 3
48. STO 1 3
49. OPR 0 0
50. JMP 0 50
51. INT 0 3
52. LIT 0 1
53. LOD 1 7
54. LIT 0 1
55. LOD 1 7
56. OPR 0 4
57. LIT 0 1
58. LOD 1 7
59. OPR 0 4
60. LIT 0 2
61. LOD 1 8
62. LIT 0 2
63. LOD 1 8
64. OPR 0 4
65. LIT 0 2
66. LOD 1 8
67. OPR 0 4
68. OPR 0 2
69. LIT 0 3
70. LOD 1 9
71. LIT 0 3
72. LOD 1 9
73. OPR 0 4
74. LIT 0 3
75. LOD 1 9
76. OPR 0 4
77. OPR 0 2
78. STO 1 4
79. LOD 1 4
80. OPR 0 14
81. OPR 0 15
82. OPR 0 0
83. INT 0 13
84. LIT 0 1
85. STO 0 3
86. LOD 0 3
87. LIT 0 10
88. OPR 0 10
89. JPC 0 104
90. CAL 0 1
91. CAL 0 18
92. CAL 0 49
93. LOD 0 4
94. LOD 0 3
95. OPR 0 8
96. JPC 0 99
97. LOD 0 3
98. OPR 0 14
99. OPR 0 15
100.       LOD 0 3
101.       LIT 0 10
102.       OPR 0 2
```

```
103.        STO 0 3
104.        JMP 0 85
105.        OPR 0 0
```

由于篇幅有限，其他的测试样例的结果在上机检查时展示，此处省略。

接下来展示一下报错处理的运行结果。

报错处理分为三类：

1. 词法分析时出错，比如：标识符不符合规范，含有非法符号等。
2. 语法分析时出错，比如：赋值语句缺乏“；”作为结尾，if-else 语句只含有 if，不含 else 等。
3. 语义分析错误，比如：数组变量访问越界，修改 const 常量等。

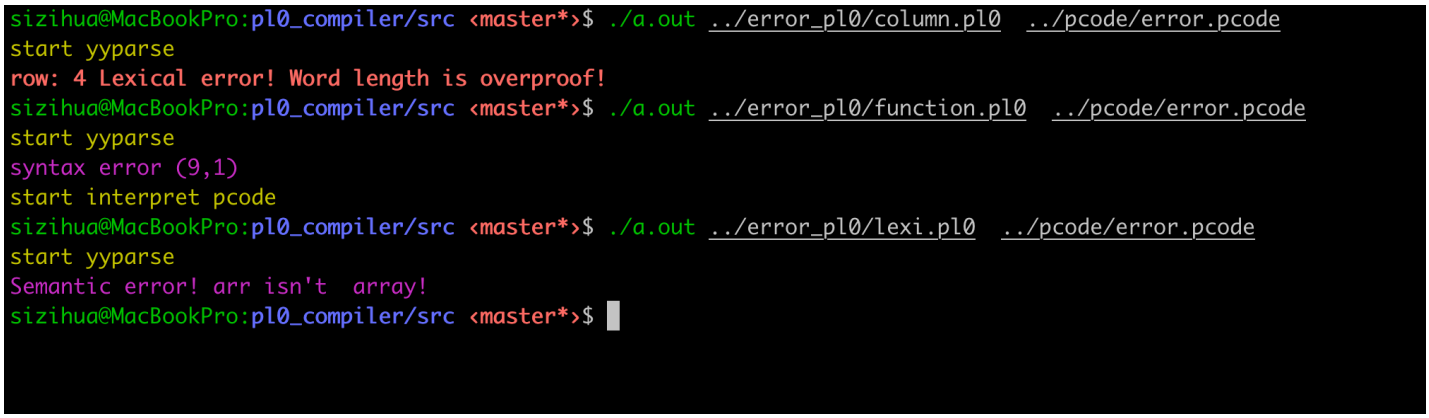

Figure 14:报错处理

Figure 14 中从上到下三个运行结果分别对应词法、语法、语义中错误的情况。源代码中各个部分的错误处理的情形比展示的这三个丰富的多，但是篇幅限制，在此不截图展示了。