|  |  |
| --- | --- |
|  |  |
| **Pl0编译器** |  |
|  |  |
|  | **2020.06.26**  **编译原理** |
|  |  |

目录

[1 成果概览 3](#_Toc44158556)

[1.1 理解编译器的工作机制，掌握编译器的工作原理 3](#_Toc44158557)

[1.2 具体编译器的实现细节 3](#_Toc44158558)

[1.3 编译系统的结构 4](#_Toc44158559)

[2 含扩展的PL0语言的描述 6](#_Toc44158560)

[2.1 PL0语言文法的EBNF表示 6](#_Toc44158561)

[2.2 PL0语言的语法图描述 8](#_Toc44158562)

[3 PL0编译程序的词法分析 10](#_Toc44158563)

[4 语法、语义分析部分 13](#_Toc44158564)

[4.1 核心语法介绍 14](#_Toc44158565)

[4.2 扩展后的类pcode代码 16](#_Toc44158566)

[5 运行结果展示 20](#_Toc44158567)

# 成果概览

## 理解编译器的工作机制，掌握编译器的工作原理

## 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++。

**Pl0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

## 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

# 含扩展的PL0语言的描述

## 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

## PL0语言的语法图描述



Figure 2：程序语法描述图

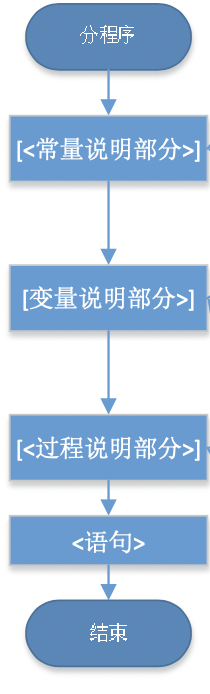


Figure 3：分程序描述图

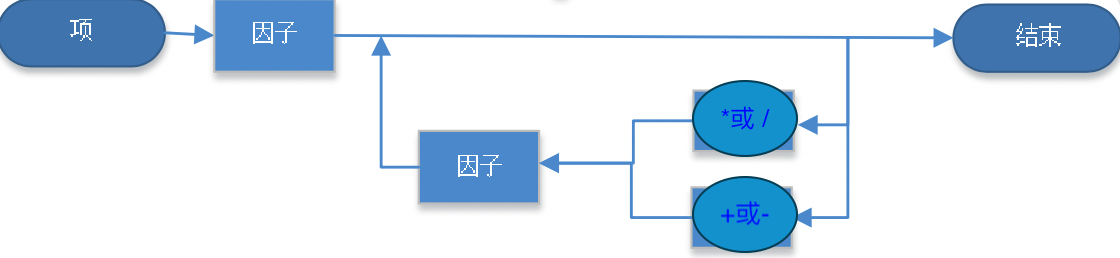


Figure 4：项语法描述图



Figure 5：因子语法描述图

# 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

1. {illegal} {
2. //column+=yyleng;
3. std::cout<<BOLDRED<<"Lexical error! Illegal input: row "<<row<<std::endl;
4. exit(0);
5. }

* 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

1. {Integer}|-{Integer} {

4. **if**(yyleng>14)
5. {
7. std::cout<<BOLDRED<<"Lexical error! Word length is overproof!\n";
8. exit(0);
9. }
10. column+=yyleng;
11. yylval.i\_val=atoi(yytext);
12. **return** INTEGER\_VAL;
13. }
14. {Float}|-{Float} {

17. **if**(yyleng>14)
18. {
20. std::cout<<BOLDRED<<"Lexical error! Word length is overproof!\n";
21. exit(0);
22. }
23. column+=yyleng;
24. yylval.f\_val=atof(yytext);
25. **return** FLOAT\_VAL;
26. }



Figure 6：词法分析程序的状态转换图

# 语法、语义分析部分



Figure 7：语法层次关系图

## 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。 |  |  |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  | |  | |  |
|  | 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。 | |  | |  |
|  | |  | |
| 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>)};) | |  | |

**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。

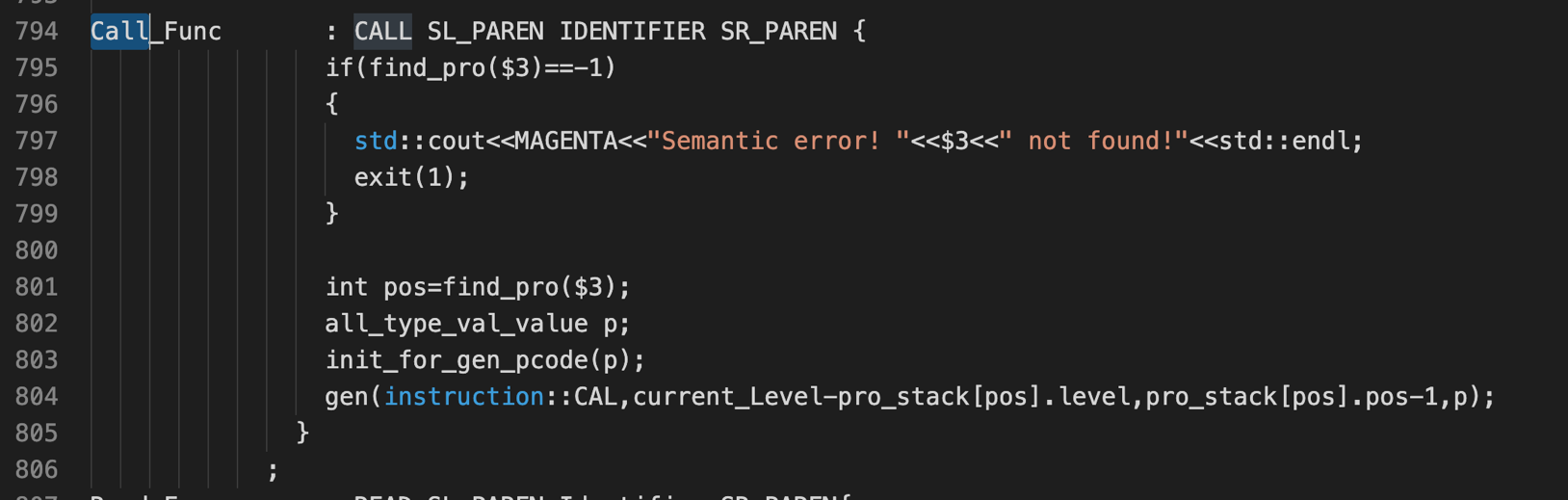


Figure 8：语法中的报错处理

## 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. };

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

# 运行结果展示



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;
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**;
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;
23. n := m;
24. **end**;
25. n := tmp;
26. **end**;
28. **procedure** sumdigit;
29. **begin**
30. m := digit[1] \* digit[1] \* digit[1] +
31. digit[2] \* digit[2] \* digit[2] +
32. digit[3] \* digit[3] \* digit[3];
33. **write**(m);
34. **end**;
36. **begin**
37. n := 1;
38. **while** n < 10 **do**
39. **begin**
40. call (init);
41. call (parse);
42. call (sumdigit);
43. **if** m == n **then**
44. **begin**
45. **write**(n);
46. **end**;
47. n := n+10;
48. **end**;
49. **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。

1. JMP 0 82
2. JMP 0 2
3. INT 0 3
4. LIT 0 1
5. STO 1 5
6. LOD 1 5
7. LIT 0 4
8. OPR 0 10
9. JPC 0 17
10. LOD 1 5
11. LIT 0 0
12. STO 1 7
13. LOD 1 5
14. LIT 0 1
15. OPR 0 2
16. STO 1 5
17. JMP 0 5
18. OPR 0 0
19. JMP 0 19
20. INT 0 4
21. LOD 1 3
22. STO 0 3
23. LIT 0 0
24. STO 1 5
25. LOD 1 3
26. LIT 0 0
27. OPR 0 9
28. JPC 0 46
29. LOD 1 3
30. LIT 0 10
31. 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。

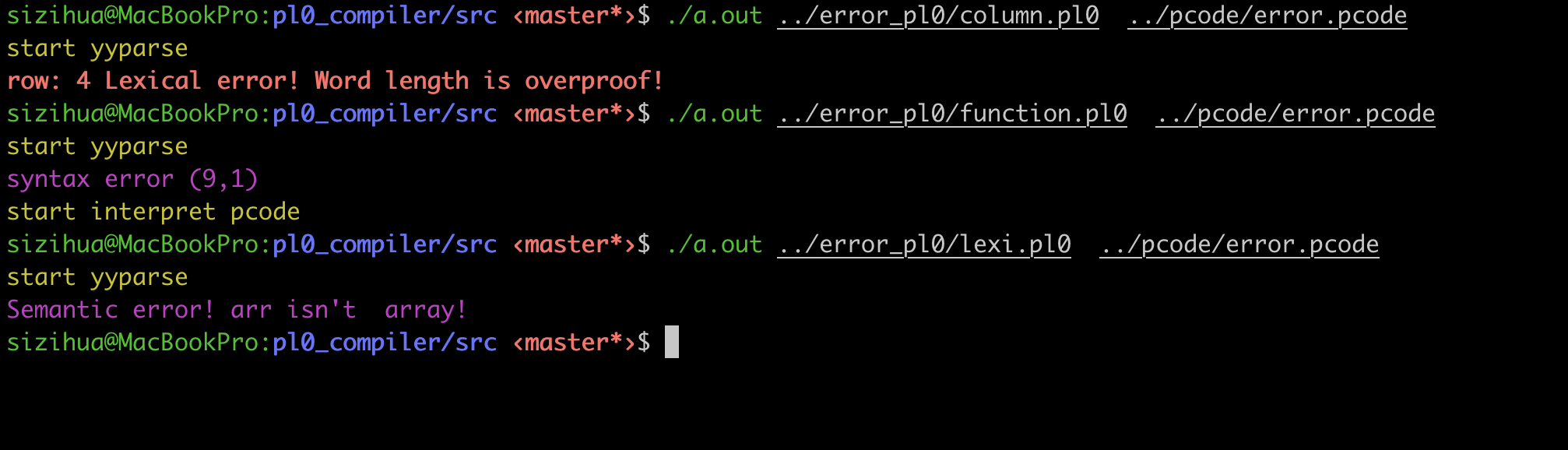


Figure 14:报错处理

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。