PL/O 简易编译器的实现

该实验PL/O编译器结构满足PL/0 语言的文法规范， 目的为应课题要求，完成课业任务，结合上海大学《编译原理课程-实验指导书》，实验项目如下：

目录

> 项目框架

本项目采用MAVEN搭建，环境为JDK14，IntelliJ IDEA 2019.3.4 (Ultimate Edition)，测试环境JUNIT5

@Author : 白皓天

@ID ： 17121444

@Institution ： 上海大学计算机工程与科学学院

@Major ： 计算机科学与技术

标识符的识别

实验内容

输入 PL/0 语言源程序，输出源程序中所有标识符的出现次数。

实验要求

- 识别程序读入 PL/0 语言源程序（文本文件），识别结果也以文本文件保存。
- 按标识符出现的顺序输出结果， 每个标识符一行， 采用二元式序列，即： (标识符值, 标识符出现次数)
- 源程序中字符不区分大小写，即：“a1” 和“A1” 是同一个标识符。
- 准备至少 5 组测试用例，每组测试用例包括：输入源程序文件和输出结果。
- 测试用例应该考虑各种合法标识符的组合情况。

输入输出样例

输入：（文本文件）

```
Const num=100;
Var a1,b2;
Begin
Read(A1);
b2:=a1+num;
write(A1,B2);
End.
```

输出：（文本文件）

```
（num: 2）
（a1: 4）
（b2: 3）
```

实验分析

根据标识符定义以及EBNF元符号的说明可知：

标识符可以描述为：以字母开头的字母与数字的组合。

而根据实验要求：应当读入，输出文件，

因此，本次实验共有两个分步骤：

- 创建文件读入和输出I/O类，输入输出txt文件
- 编写Identifiers.java 抽象类，其实现的主要功能有：
  - 区分关键字，运算符与合法的标识符
  - 以二元组的形式输出合法标识符的数目统计结果

文法类?

即对源程序给出精确无二义的语法描述。（严谨、简洁、易读），根据文法定义四元组：

文法G定义为四元组（ VN， VT， P， S）
– VN ：非终结符集
– VT ：终结符集
– P：产生式（规则）集合
– S：开始符号， S∈ VN， S必须要在一条规则的左部出现。
VN∩ VT= φ
V=VN∪ VT，称为文法G的文法符号集合

标识符的文法定义如下：

文法G=（ VN， VT， P， S）
VN ={标识符，字母，数字}
VT ={a,b,c,…x,y,z,0,1,…,9}
P={<标识符>→<字母>
<标识符>→<标识符><字母>
<标识符>→<标识符><数字>

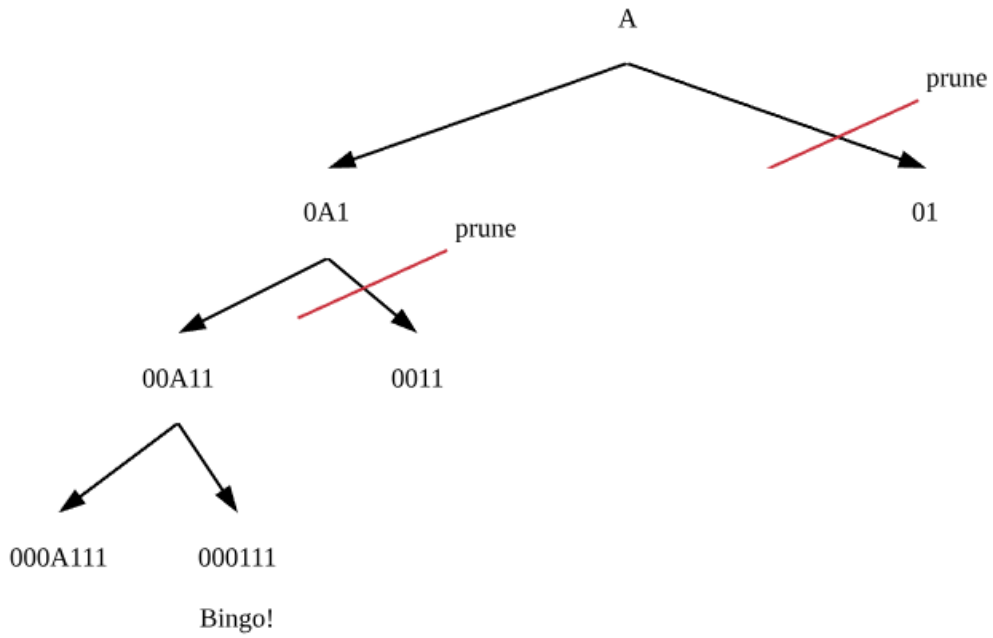
<字母>→a,…, <字母>→z
<数字>→0,…, <数字>→9 }
S=<标识符>

如何利用文法判定标识符是否合法呢？有两种方案，自顶向下与自顶向下，通过剪枝文法树的方式，判断合法性。

举个例子：设产生式集合：

P={ A→0A1, A→01 } 判断000111是否合法

A为文法G的开始符号

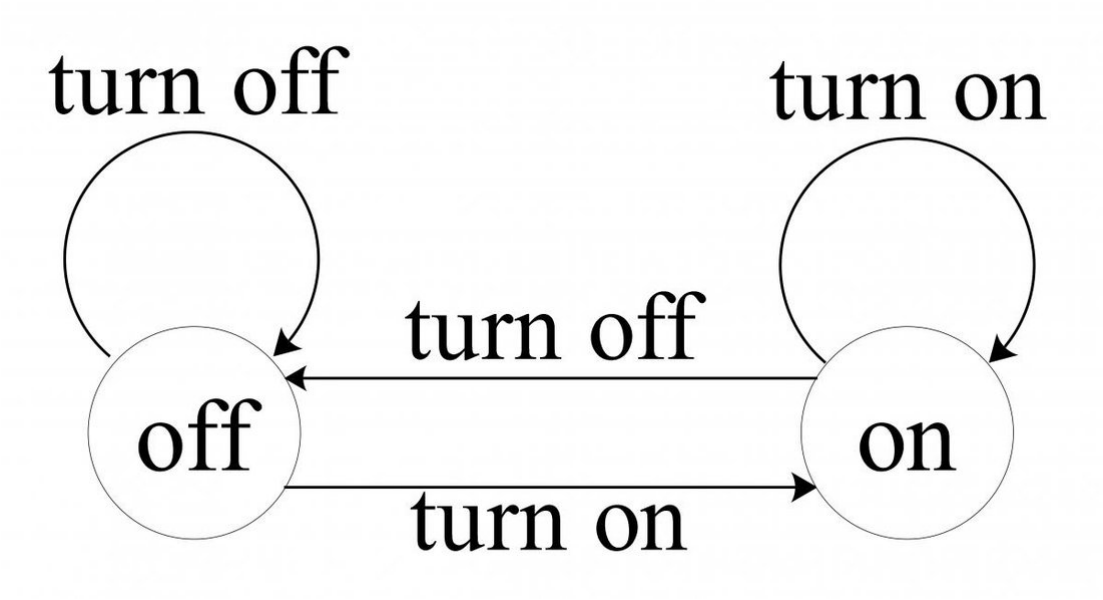


> 注：前面所讲为查阅资料之前的想法

在阅读了几篇文献后，发现这样做（文法类的建立）虽然可行，但具有以下不足：

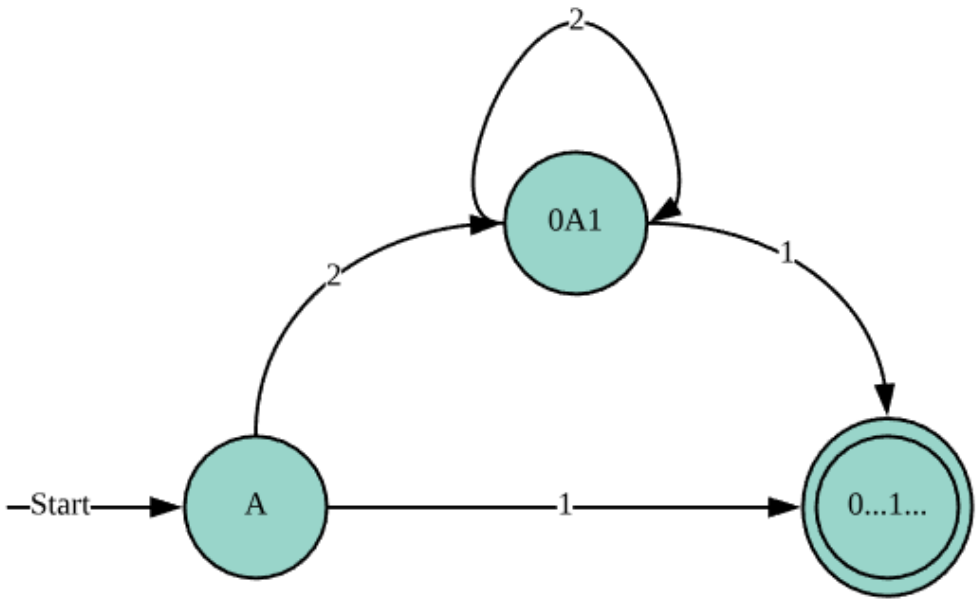
- 定义剪枝条件：如上图，运用A→01推导，因为此时没有A，无法再“延长”字符串以达到要求，但这是文法类做的事情么。如果产生式改变，这个推到条件又得变化了，如0A1→A，要求“缩短”，故无法达到整合处理的目的。
- 其实熟悉**状态机**后会发现，prune的子树回到了开始状态，如果转换成图，节省了判断剪枝条件，并且数据结构也变得简单。【这也是主流编译器通用的词法分析手段】

状态机



引用一位博主在介绍状态机时举的例子【1】，我们日常生活中，灯关闭时，打开，为开启状态；再关闭，回到关闭状态；开启时同上。其实很多程序内部我们都用到了状态机的方法，只不过较为隐晦。

在状态有限的条件下，也就是常说的FSM（finite state machine)，对该方法最为直接的运用便是正则表达式。



> 1 : A→01 ； 2 : A→0A1 ；0...1...表示相同个数的0和1

对于上方的状态转换可以该图描述。

正则表达式

正则表达式，与用EBNF描述语法的形式颇有类似，及无二义性地对概念进行定义。

不过当前，很多主流语言如java，python，c++都对正则表达式有了充分支持，正则表达式只是描述，但最为核心的一点是，**任何的正则表达式都可以用FSM表示**，所以我们使用正则表达式进行字符串匹配的时候：

可以知道其实经过了以下步骤：

1. 将正则表达式转换成FSM
2. 生成FSM代码

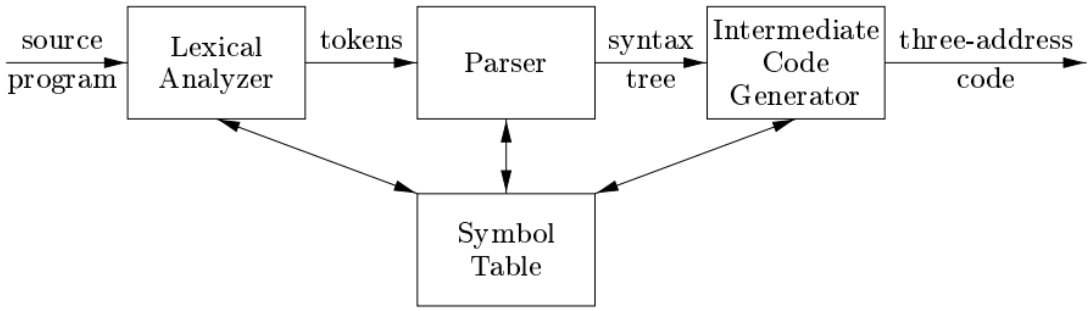
FSM代码的思路通常为：

```
state = start_state
input = get_next_input
whilenotend of input do
  state = move(state, input)
  input = get_next_input
endif state is a final state
  return"ACCEPT"else return"REJECT"
```

所以第二步也是十分清晰的。

实现

摸着石头过河必定会花费大量时间，但有时候也是值得的，特别是想要思考来龙去脉的时候。但要进行实现的时候，参考前人的经验，既可以激发思绪，也可以节省大量尝试时间。想要搭建较为满意的编译器，采用现有的结构框架是一个很好的做法。



这是著名的Dragon Book2 中的编译器的前端模型，后面内容会继续引用该图：

目前，要做的为Lexical Analyzer，

主要功能用一句话概括为：读入输入流转化为tokens对象组。

1.定义Tag类

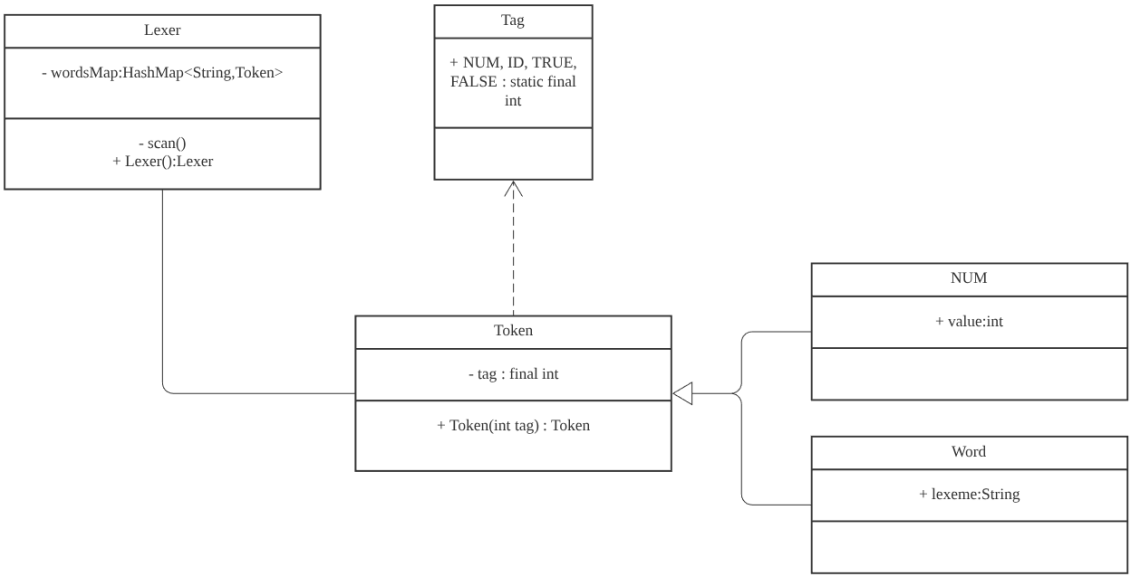
```
public enum Tag {
    ID("[a-zA-Z][[0-9]|[a-zA-Z]]*"),
    NUM("[0-9]+"),
    OPE("\\+|-|\\*|/|=|#|<|<=|>|>=|:="),
    DELIMITER("\\(|\\)|（|）|,|;|\\."),
    BASIC("begin|call|const|do|end|if|odd|procedure|read|then|var|while|write");

    public final String pattern;

    Tag(String pattern) {
        this.pattern = pattern;
    }
}
```

以描述tokens的类别

测试结果如下：（通过测试）

```
// NUM
        assertTrue(Pattern.matches(Tag.NUM.pattern, "555666600"));
        assertFalse(Pattern.matches(Tag.NUM.pattern, "555xe2"));
        // ID
        assertTrue(Pattern.matches(Tag.ID.pattern, "u123x"));
        assertFalse(Pattern.matches(Tag.ID.pattern, "12Uddd"));
        // OPE
        assertTrue(Pattern.matches(Tag.OPE.pattern, "+"));
        assertTrue(Pattern.matches(Tag.OPE.pattern, "-"));
        assertTrue(Pattern.matches(Tag.OPE.pattern, "*"));
        assertTrue(Pattern.matches(Tag.OPE.pattern, "/"));
        assertTrue(Pattern.matches(Tag.OPE.pattern, "="));
        assertTrue(Pattern.matches(Tag.OPE.pattern, ":="));
        assertTrue(Pattern.matches(Tag.OPE.pattern, "#"));
        assertTrue(Pattern.matches(Tag.OPE.pattern, "<="));
        assertTrue(Pattern.matches(Tag.OPE.pattern, ">="));
        // BASIC
        assertTrue(Pattern.matches(Tag.BASIC.pattern, "begin"));
        // DELIMITER
        assertTrue(Pattern.matches(Tag.DELIMITER.pattern, "）"));
        assertTrue(Pattern.matches(Tag.DELIMITER.pattern, "("));
        assertTrue(Pattern.matches(Tag.DELIMITER.pattern, ")"));
        assertTrue(Pattern.matches(Tag.DELIMITER.pattern, "（"));
        assertTrue(Pattern.matches(Tag.DELIMITER.pattern, ","));
        assertTrue(Pattern.matches(Tag.DELIMITER.pattern, ";"));
        assertTrue(Pattern.matches(Tag.DELIMITER.pattern, "."));
```

2. UML类图



整个项目结构清晰，Lexer将输入的字符处理，主要用scan（）调用Tag中的NUM，与ID的识别方法，从而构建了相应的Token，并放入HashMap以存储字。

为了实验需要，有设置了一个专门统计Identifier的HashMap。

如此得到结果。

3.一个关键问题

老师的测试数据任然不算严格，例如正常编译器中，有3+2>4，而非3 + 2 > 4, 这样规范的写法，如何识别连续输入的字符串呢？

这里Tag中的正则表达式发挥了作用：

1. 匹配所有满足表达式的结果并储存。
2. 字符串缩进，提高效率。

关键还是正则表达式，可见上方Tag类定义。

测试用例结果

举一个例子，其余的请见测试resources文件夹：

case02.txt

```
CONST A=10;
VAR   B,C;
PROCEDURE  P;
      VAR D;
      PROCEDURE  Q;
           VAR X;
           BEGIN
            READ(X);
                D:=X;
                WHILE X#0
                    DO CALL P;
          END;
       BEGIN
```

```
        WRITE(D);
       CALL Q;
  END
BEGIN
 CALL P;
END.
```

result/case02Result.txt

```
(X: 4)
(D: 3)
(C: 1)
(B: 1)
(Q: 2)
(A: 1)
(P: 3)
```

可见结果正确。

我加入了注释功能，如java的// 与 /* */

测试用例：homework_1.txt

```
// 编写一个PL/0程序，输入3个正整数a、b、c，按从小到大的顺序输出这三个数
var a,b,c,t;
begin
    read(a);
    read(b);
    read(c);
    if a > b then
        begin
            t := b;
            b := a;
            a := t;
        end
    if b > c then
        begin
            t := b;
            b := c;
            c := t;
        end
    if a > c then
        begin
            t := a;
            a := c;
            c := t;
        end
    begin
        write(a);
        write(b);
        write(c);
    end
end
```

结果：

```
(t: 7)
(a: 9)
(c: 9)
(b: 9)
```

后续工作

接下来就是变量表，文法，报错等内容，都留下了一些接口。总的来说，第一次工作量挺大，查资料，写代码，测试，整个周末都搭进去了，但收获明显：正则表达式，FSM，编译器基本框架。

词法分析

一、实验目的：

1. 根据 PL/0 语言的 文法规范，编写 PL/0 语言的词法分析程序。
2. 通过设计调试词法分析程序，实现从源程序中分出各种单词的方法；加深对课堂教学的理解；提高词法分析方法的实践能力 。
3. 掌握从源程序文件中读取有效字符的方法和产生源程序的内部表示文件的法。
4. 掌握词法分析的实现方法。
5. 上机调试编出的词法分析程序。

二、实验内容：

输入PL/0语言程序，输出程序中各个单词符号（关键字、专用符号以及其它标记）。

三、实验要求：

1. 确定编译中单词种类、使用的表格、标识符与关键字的区分方法等。
2. 词法分析器读入PL/0语言源程序（文本文件），识别结果也以文本文件保存。
3. 词法分析器的输出形式采用二元式序列，即：

(单词种类,单词的值)

4. 源程序中字符不区分大小写，即：“a1”和“A1”是同一个标识符。
5. 准备至少5组测试用例，每组测试用例包括：输入源程序文件和输出结果。
6. 测试用例必须包含所有的基本字、运算符、界符、以及各种标识符和常数。对不合法单词进行分类考虑测试用例，特别是对一些运算。

四、实验过程：

1. 设计思路

   在上一个步骤中，已经基本实现了词法分析，即将一段源代码作为输入，能够正确区分词的类型。这一步是对上面的补充。

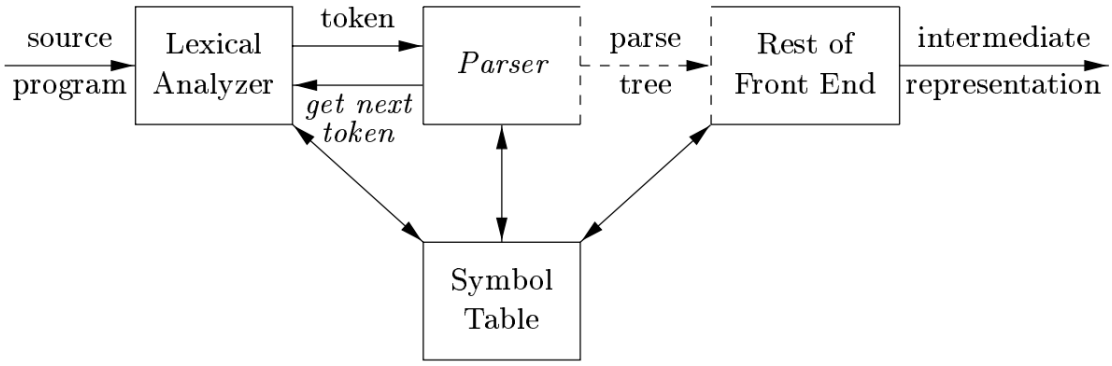
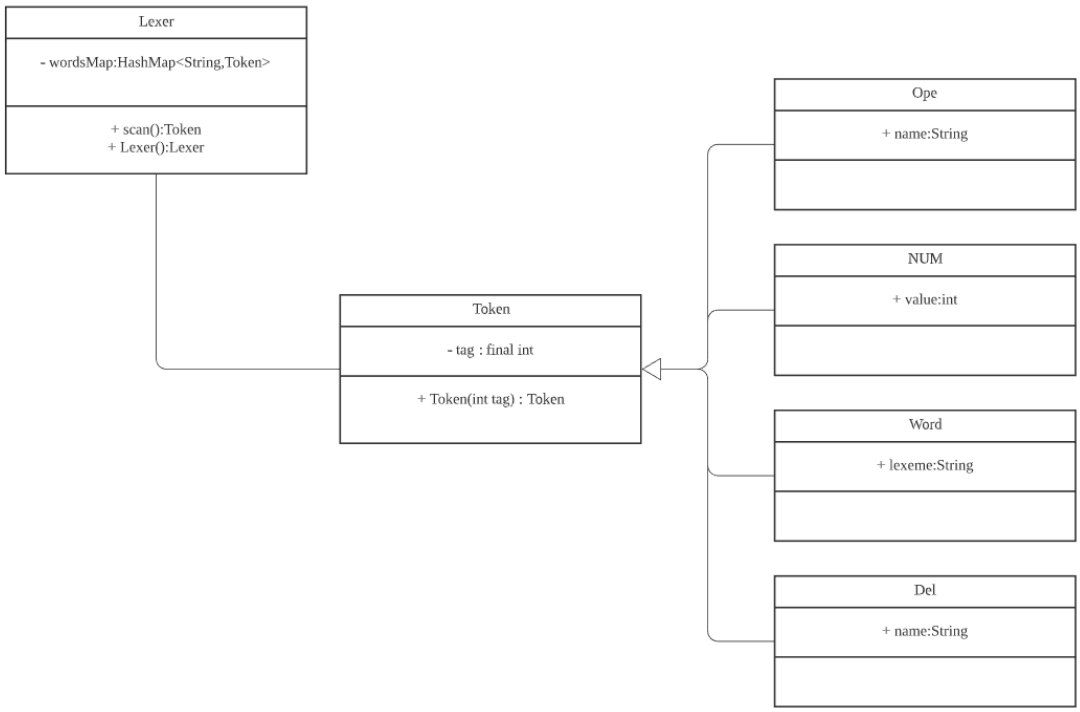


Figure 4.1: Position of parser in compiler model

由于上次对Lexical Analyzer的功能有少许误解，这次修改scan（）功能，以只得到一个套接字。以方便后续Parser的编写。

修改后的设计UML类图如下：



4. 实验结果

举一个例子：case01.txt

```
var

    x,y,z;

begin

    x := 10;

    y := -5;

    read(z);

    if z > 3 then

        write(x)
```

```
    else
        write(y);
end.
```

结果：

```
(VAR, var)
(ID, X)
(Del, comma)
(ID, Y)
(Del, comma)
(ID, Z)
(Del, semicolon)
(BEGIN, begin)
(ID, X)
(OPE, becomes)
(NUM, 10)
(Del, semicolon)
(ID, Y)
(OPE, becomes)
(OPE, minus)
(NUM, 5)
(Del, semicolon)
(READ, read)
(Del, lParen)
(ID, Z)
(Del, rParen)
(Del, semicolon)
(IF, if)
(ID, Z)
(OPE, greaterThan)
(NUM, 3)
(THEN, then)
(WRITE, write)
(Del, lParen)
(ID, X)
(Del, rParen)
(ELSE, else)
(WRITE, write)
(Del, lParen)
(ID, Y)
(Del, rParen)
(Del, semicolon)
(END, end)
(Del, period)
```

四、结论及体会：

此次实验较为简单，但然我弄明白了基本的词法分析的功能，每次一个套接字，比起多个一起识别，还稍微复杂一点，因为没有了Pattern 这样方便的道具。

引用文章

[1] : by Eli Bendersky, 2003. http://archive.gamedev.net/archive/reference/programming/features/af1/index.html Copyright

[2] : Dragon Book, Aho, Sethi, Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986. ISBN [0-201-10088-6],2ed.

[3] :Engineering with math and computation, http://giocc.com/writing-a-lexer-in-java-1-7-using-regex-named-capturing-groups.html.

附录

EBNF 的元符号

‘< >’ 是用左右尖括号括起来的中文字表示语法构造成分，或称语法单位，为非终结符。

‘::=’ 该符号的左部由右部定义，可读作‘定义为’

‘|’ 表示‘或’，即左部可由多个右部定义

‘{ }’ 表示花括号内的语法成分可以重复；在不加上下界时可重复0到任意次数，有上下界时为可重复次数的限制

‘[ ]’ 表示方括号内的成分为任选项

‘( )’ 表示圆括号内的成分优先

PL/0 语言文法的 EBNF

<程序>::=<分程序>.
<分程序> ::=【<常量说明>】[<过程说明>]<语句>
<常量说明> ::=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>}
<变量说明> ::=VAR <标识符>{, <标识符>};
<标识符> ::=<字母>{<字母>|<数字>}
<过程说明> ::=<过程首部><分程序>{; <过程说明> };
<过程首部> ::=PROCEDURE <标识符>;
<语句> ::=<赋值语句>|<条件语句>|<当循环语句>|<过程调用语句>
|<复合语句>|<读语句><写语句>|<空>
<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> ::= <表达式> <关系运算符> <表达式> |ODD<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> ::= *|/
<关系运算符> ::= =|#|<|<=|>|>=
<条件语句> ::= IF <条件表达式> THEN <语句>
<过程调用语句> ::= CALL 标识符
<当循环语句> ::= WHILE <条件表达式> DO <语句>
<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

类型、上下文约束与作用域规则

数据类型只有整数类型

数据结构只支持简单变量和常数

所支持的数字为最长 9 位的十进制数

标识符的有效长度为10

标识符引用前先要声明

过程无参数

过程可嵌套，最多嵌套 3 层

过程可递归调用

内层过程可以引用包围它的外层过程的标识符

PL/O语言的词汇表

序号	类别	单词	编码
1	基本字	begin、 call、 const、 do、 end、 if、 odd、 procedure、 read、 then、 var、 while、 write	beginsym, callsym, constsym dosym, endsym, ifsym, oddsym proceduresym, readsym, thensym varsym, whilesym, writesym
2	标识符	ident	
3	常数	number	
4	运算符	+、 -、 *、 /、 =、 #、 <、 <=、 >、 >=、 :=	plus, minus, times, slash, eql, neq, lss, leq, gtr, geq, becomes

序号	类别	单词	编码
5	界符	（、）、，、、；、．	Lparen, rparen, comma, semicolon period

ASCII

ASCII TABLE

Decimal	Hexadecimal	Binary	Octal	Char	Decimal	Hexadecimal	Binary	Octal	Char	Decimal	Hexadecimal	Binary	Octal	Char
0	0	0	0	[NULL]	48	30	110000	60	0	96	60	1100000	140	`
1	1	1	1	[START OF HEADING]	49	31	110001	61	1	97	61	1100001	141	a
2	2	10	2	[START OF TEXT]	50	32	110010	62	2	98	62	1100010	142	b
3	3	11	3	[END OF TEXT]	51	33	110011	63	3	99	63	1100011	143	c
4	4	100	4	[END OF TRANSMISSION]	52	34	110100	64	4	100	64	1100100	144	d
5	5	101	5	[ENQUIRY]	53	35	110101	65	5	101	65	1100101	145	e
6	6	110	6	[ACKNOWLEDGE]	54	36	110110	66	6	102	66	1100110	146	f
7	7	111	7	[BELL]	55	37	110111	67	7	103	67	1100111	147	g
8	8	1000	10	[BACKSPACE]	56	38	111000	70	8	104	68	1101000	150	h
9	9	1001	11	[HORIZONTAL TAB]	57	39	111001	71	9	105	69	1101001	151	i
10	A	1010	12	[LINE FEED]	58	3A	111010	72	:	106	6A	1101010	152	j
11	B	1011	13	[VERTICAL TAB]	59	3B	111011	73	;	107	6B	1101011	153	k
12	C	1100	14	[FORM FEED]	60	3C	111100	74	<	108	6C	1101100	154	l
13	D	1101	15	[CARRIAGE RETURN]	61	3D	111101	75	=	109	6D	1101101	155	m
14	E	1110	16	[SHIFT OUT]	62	3E	111110	76	>	110	6E	1101110	156	n
15	F	1111	17	[SHIFT IN]	63	3F	111111	77	?	111	6F	1101111	157	o
16	10	10000	20	[DATA LINK ESCAPE]	64	40	1000000	100	@	112	70	1110000	160	p
17	11	10001	21	[DEVICE CONTROL 1]	65	41	1000001	101	A	113	71	1110001	161	q
18	12	10010	22	[DEVICE CONTROL 2]	66	42	1000010	102	B	114	72	1110010	162	r
19	13	10011	23	[DEVICE CONTROL 3]	67	43	1000011	103	C	115	73	1110011	163	s
20	14	10100	24	[DEVICE CONTROL 4]	68	44	1000100	104	D	116	74	1110100	164	t
21	15	10101	25	[NEGATIVE ACKNOWLEDGE]	69	45	1000101	105	E	117	75	1110101	165	u
22	16	10110	26	[SYNCHRONOUS IDLE]	70	46	1000110	106	F	118	76	1110110	166	v
23	17	10111	27	[ENG OF TRANS. BLOCK]	71	47	1000111	107	G	119	77	1110111	167	w
24	18	11000	30	[CANCEL]	72	48	1001000	110	H	120	78	1111000	170	x
25	19	11001	31	[END OF MEDIUM]	73	49	1001001	111	I	121	79	1111001	171	y
26	1A	11010	32	[SUBSTITUTE]	74	4A	1001010	112	J	122	7A	1111010	172	z
27	1B	11011	33	[ESCAPE]	75	4B	1001011	113	K	123	7B	1111011	173	{
28	1C	11100	34	[FILE SEPARATOR]	76	4C	1001100	114	L	124	7C	1111100	174	|
29	1D	11101	35	[GROUP SEPARATOR]	77	4D	1001101	115	M	125	7D	1111101	175	}
30	1E	11110	36	[RECORD SEPARATOR]	78	4E	1001110	116	N	126	7E	1111110	176	~
31	1F	11111	37	[UNIT SEPARATOR]	79	4F	1001111	117	O	127	7F	1111111	177	[DEL]
32	20	100000	40	[SPACE]	80	50	1010000	120	P					
33	21	100001	41	!	81	51	1010001	121	Q					
34	22	100010	42	"	82	52	1010010	122	R					
35	23	100011	43	#	83	53	1010011	123	S					
36	24	100100	44	$	84	54	1010100	124	T					
37	25	100101	45	%	85	55	1010101	125	U					
38	26	100110	46	&	86	56	1010110	126	V					
39	27	100111	47	'	87	57	1010111	127	W					
40	28	101000	50	(	88	58	1011000	130	X					
41	29	101001	51	)	89	59	1011001	131	Y					
42	2A	101010	52	*	90	5A	1011010	132	Z					
43	2B	101011	53	+	91	5B	1011011	133	[					
44	2C	101100	54	,	92	5C	1011100	134	\					
45	2D	101101	55	-	93	5D	1011101	135	]					
46	2E	101110	56	.	94	5E	1011110	136	^					
47	2F	101111	57	/	95	5F	1011111	137	_					

JAVA SE 14

Java® Platform, Standard Edition & Java Development Kit
Version 14 API Specification