

# EX 2

20308003 曾伟超

## 词汇表

这里使用了一些扩展的正则表达，例如 `a-z` 表示所有的小写字母，`a-zA-Z` 表示所有字母

类型	正则表达
十进制数字	<code>(1-9)(0-9)*</code>
八进制数字	<code>0(0-7)*</code>
标识符	<code>(a-zA-Z)(a-zA-Z 0-9)*</code>
关键字(忽略大小写)	<code>integer   boolean   write   writeln   read</code>
保留字(忽略大小写)	<code>if   elsif   then   else   while   begin   do   end   of   var   const   type   procedure   record   array   module</code>
标点	<code>,   ;</code>
算数运算符	<code>+   -   \*   div   mod</code>
关系运算符	<code>&gt;   &gt;=   &lt;   &lt;=   =   #</code>
逻辑运算符	<code>&amp;   or   ~</code>
赋值运算符	<code>:=</code>
选择运算符	<code>\.   [   ]</code>
括号	<code>(   )</code>
类型声明	<code>:</code>
注释	<code>\( \* . * ? \* \)</code>

## 分类依据以及一些说明

关键字和保留字按照了 `ex1` 中的处理办法，将在语法中存在的作为保留字，其它的放入关键字

还有一些是我个人处理中存在些许疑惑的，例如 `div` 是放在算术运算符还是保留字中，`mod` 也是，但是最终考虑到一致性，还是将其放入了算数运算符中，这点对于后面的 `or` 也是相同的道理

而在注释，由于 `*` 和 `()` 本身在正则中有其含义，从而将 `*` 使用 `\` 做转义，`*` 的类似，而为了防止嵌套，需要采取非贪心的策略，即左注释遇到第一个右注释即停止，这里参考了一些扩展正则的标记，使用 `. * ?` 来做非贪心的匹配，这样遇到第一个 `(*)` 的时候即结束

## 词法规则

这里为了简化，将所有的运算符都统一使用 `Operator` 代替

$$\begin{aligned}
Number &\rightarrow DEC \mid OCT \\
DEC &\rightarrow 0(0-9)^* \\
OCT &\rightarrow (1-9)(0-9)^* \\
Identifier &\rightarrow (a-zA-Z)(a-zA-Z|0-9)^* \\
Marks &\rightarrow , \mid ; \\
Operator &\rightarrow + \mid - \mid * \mid div \mid mod \mid > \mid >= \mid < \mid <= \mid = \mid \# \mid \& \mid or \mid \sim \mid := \mid \backslash \cdot \mid [ \mid ] \mid \backslash ( \mid \backslash ) \mid : \\
Comment &\rightarrow \backslash ( \backslash * . * ? \backslash * \backslash ) \\
Keywords &\rightarrow integer \mid boolean \mid write \mid writeln \mid read \\
ReservedWords &\rightarrow if \mid elsif \mid then \mid else \mid while \mid begin \mid do \mid end \mid of \mid var \mid const \mid type \mid procedure \mid record \mid array \mid module
\end{aligned}$$

## 和高级语言的对比

1. 高级语言如 C/C++, Java 等, 其标识符的正则会更加的负责, 可以包含如下划线(`_`)存在, 而不是仅仅有字母和数字
2. 一些运算符的不同, 例如逻辑与在 C/C++ 中是 `&&`, `&` 在 C/C++ 中是按位与, 还有 `or` 在 C/C++ 中是 `||`, 不等号和相等也都有区别
3. 赋值的区别, 赋值在 C/C++ 等高级语言中是 `=`, 而在这里是 `:=`, 因为这里的 `=` 沿用了数学的相等
4. 注释的不同, 这里没有严格的区分单行和多行注释, 这里都是用统一的注释, 任意被括住的都是被注释部分
5. 类型符号, C/C++ 中没有这个类型符号, 靠的是类型前缀声明

## JFlex

由于之前曾经使用过 `GNU Flex`, 而 `JFlex` 在使用上有很多相似的地方, 一些区别可以通过阅读文档很容易的区分, `JFlex` 文件分为三部分, 每两个部分之间使用 `%%` 进行分割, 从上到下依次为用户代码, 一些可选的选项设置, 以及最后的词法规则

用户代码部分, 主要包含了 `exceptions` 错误类, 用于抛出异常

这里由于大小写不敏感, 所以需要开启 `ignorecase`, 同时手动指定 `class` 为 `OberonScanner`

而对于各个操作符, 则是进行了区分, 实际上, 每一个无论是符号还是保留字还是关键字, 都会有独一无二的一个标识符用来区分, 具体可以参考 `TokenType.java`, 编写完成后进行测试, 以 `standalone` 模式进行测试, 由于后续的 `JavaCUP` 只是使用返回值, 从而可以忽略输出值

```
./gen.sh src/oberon.flex && ./build.sh && ./run.sh ../ex1/testcases/fib.obr
```

得到的结果如下图所示

```
..sysu/exp3/ex2 x + v
writeln: 'writeln' Loc=<60:12>
elseif: 'ELSIF' Loc=<61:8>
identifier: 'n' Loc=<61:14>
greater: '>' Loc=<61:16>
Octal: '0' Loc=<61:18>
then: 'THEN' Loc=<61:20>
write: 'write' Loc=<62:12>
lparen: '(' Loc=<62:17>
identifier: 'n' Loc=<62:18>
rparen: ')' Loc=<62:19>
semicolon: ';' Loc=<62:20>
writeln: 'writeln' Loc=<63:12>
else: 'ELSE' Loc=<64:8>
write: 'write' Loc=<65:12>
lparen: '(' Loc=<65:17>
identifier: 'n' Loc=<65:18>
rparen: ')' Loc=<65:19>
semicolon: ';' Loc=<65:20>
writeln: 'writeln' Loc=<66:12>
end: 'END' Loc=<67:8>
end: 'END' Loc=<68:4>
identifier: 'Main' Loc=<68:8>
semicolon: ';' Loc=<68:12>
end: 'END' Loc=<70:0>
identifier: 'Fib' Loc=<70:4>
dot: '.' Loc=<70:7>

~/minijava_sysu/exp3/ex2 on main !5 > py base at 01:39:00
```

可以看到，能够被成功的进行解析，之后，参考之前的变异，其中设计词法错误的为 001, 002, 012, 013, 014, 015，依次进行测试，如下图

001

```
..sysu/exp3/ex2 x + v
end: 'END' Loc=<67:8>
end: 'END' Loc=<68:4>
identifier: 'Main' Loc=<68:8>
semicolon: ';' Loc=<68:12>
end: 'END' Loc=<70:0>
identifier: 'Fib' Loc=<70:4>
dot: '.' Loc=<70:7>

~/minijava_sysu/exp3/ex2 on main !5 > ./run.sh testcases/fib.001 py base at 01:39:00
comment: '(* Illegal Identifier(n@) *)' Loc=<0:0>
module: 'MODULE' Loc=<2:0>
identifier: 'Fib' Loc=<2:7>
semicolon: ';' Loc=<2:10>
var: 'VAR' Loc=<3:4>
identifier: 'n' Loc=<4:8>
colon: ':' Loc=<4:9>
integer: 'INTEGER' Loc=<4:11>
semicolon: ';' Loc=<4:18>
procedure: 'PROCEDURE' Loc=<5:4>
identifier: 'Fib1' Loc=<5:14>
lparen: '(' Loc=<5:18>
identifier: 'n' Loc=<5:19>
Unexpected exception:
exceptions.IllegalSymbolException: Unknown Character Detected
at OberonScanner.yylex(OberonScanner.java:871)
at OberonScanner.main(OberonScanner.java:1236)

~/minijava_sysu/exp3/ex2 on main !6 > py base at 01:39:20
```

符合报错预期

002

```
..sysu/exp3/ex2 x + v
semicolon: ';' Loc=<14:24>
identifier: 'result' Loc=<15:12>
assign: ':= ' Loc=<15:19>
identifier: 'i' Loc=<15:22>
plus: '+' Loc=<15:24>
identifier: 'j' Loc=<15:26>
end: 'END' Loc=<16:8>
end: 'END' Loc=<17:4>
identifier: 'Fib1' Loc=<17:8>
semicolon: ';' Loc=<17:12>
procedure: 'PROCEDURE' Loc=<19:4>
identifier: 'Fib2' Loc=<19:14>
lparen: '(' Loc=<19:18>
var: 'VAR' Loc=<19:19>
identifier: 'n' Loc=<19:23>
colon: ':' Loc=<19:24>
integer: 'INTEGER' Loc=<19:26>
rparen: ')' Loc=<19:33>
semicolon: ';' Loc=<19:34>
const: 'CONST' Loc=<20:4>
identifier: 'c' Loc=<21:8>
equal: '=' Loc=<21:10>
Unexpected exception:
exceptions.IllegalOctalException: Illegal Octal Detected
    at OberonScanner.yylex(OberonScanner.java:1017)
    at OberonScanner.main(OberonScanner.java:1236)

~/minijava_sysu/exp3/ex2 on main !7 > py base at 01:39:34
```

符合报错预期

012

```
..sysu/exp3/ex2 x + v
end: 'END' Loc=<17:4>
identifier: 'Fib1' Loc=<17:8>
semicolon: ';' Loc=<17:12>
procedure: 'PROCEDURE' Loc=<19:4>
identifier: 'Fib2' Loc=<19:14>
lparen: '(' Loc=<19:18>
var: 'VAR' Loc=<19:19>
identifier: 'n' Loc=<19:23>
colon: ':' Loc=<19:24>
integer: 'INTEGER' Loc=<19:26>
rparen: ')' Loc=<19:33>
semicolon: ';' Loc=<19:34>
const: 'CONST' Loc=<20:4>
identifier: 'c' Loc=<21:8>
equal: '=' Loc=<21:10>
Unexpected exception:
exceptions.IllegalOctalException: Illegal Octal Detected
    at OberonScanner.yylex(OberonScanner.java:1017)
    at OberonScanner.main(OberonScanner.java:1236)

~/minijava_sysu/exp3/ex2 on main !7 > ./run.sh testcases/fib.012 py base at 01:39:34
comment: '(* Unmatched comment *)' Loc=<0:0>
Unexpected exception:
exceptions.MismatchedCommentException: Mismatched Comment Detected
    at OberonScanner.yylex(OberonScanner.java:1012)
    at OberonScanner.main(OberonScanner.java:1236)

~/minijava_sysu/exp3/ex2 on main !8 > py base at 01:40:00
```

符合报错预期

013

```
..sysu/exp3/ex2 x + v
lparen: '(' Loc=<5:18>
identifier: 'n' Loc=<5:19>
colon: ':' Loc=<5:20>
integer: 'INTEGER' Loc=<5:22>
semicolon: ';' Loc=<5:29>
var: 'var' Loc=<5:31>
identifier: 'result' Loc=<5:35>
colon: ':' Loc=<5:41>
integer: 'INTEGER' Loc=<5:43>
rparen: ')' Loc=<5:50>
semicolon: ';' Loc=<5:51>
var: 'VAR' Loc=<6:4>
identifier: 'i' Loc=<6:8>
comma: ',' Loc=<6:9>
identifier: 'j' Loc=<6:11>
colon: ':' Loc=<6:12>
integer: 'INTEGER' Loc=<6:14>
semicolon: ';' Loc=<6:21>
begin: 'BEGIN' Loc=<7:4>
if: 'IF' Loc=<8:8>
identifier: 'n' Loc=<8:11>
equal: '=' Loc=<8:13>
Unexpected exception:
exceptions.IllegalIntegerException: Illegal Integer Detected
    at OberonScanner.yylex(OberonScanner.java:1022)
    at OberonScanner.main(OberonScanner.java:1236)

~/minijava_sysu/exp3/ex2 on main !9 > py base at 01:40:19
```

符合报错预期

014

```
..sysu/exp3/ex2 x + v
integer: 'INTEGER' Loc=<5:43>
rparen: ')' Loc=<5:50>
semicolon: ';' Loc=<5:51>
var: 'VAR' Loc=<6:4>
identifier: 'i' Loc=<6:8>
comma: ',' Loc=<6:9>
identifier: 'j' Loc=<6:11>
colon: ':' Loc=<6:12>
integer: 'INTEGER' Loc=<6:14>
semicolon: ';' Loc=<6:21>
begin: 'BEGIN' Loc=<7:4>
if: 'IF' Loc=<8:8>
identifier: 'n' Loc=<8:11>
equal: '=' Loc=<8:13>
Octal: '0' Loc=<8:15>
then: 'THEN' Loc=<8:17>
identifier: 'result' Loc=<9:12>
assign: ':= ' Loc=<9:19>
Octal: '0' Loc=<9:22>
elsif: 'ELSIF' Loc=<10:8>
identifier: 'n' Loc=<10:14>
equal: '=' Loc=<10:16>
Unexpected exception:
exceptions.IllegalIntegerRangeException: Integer Is Out of Range
    at OberonScanner.yylex(OberonScanner.java:949)
    at OberonScanner.main(OberonScanner.java:1236)

~/minijava_sysu/exp3/ex2 on main !10 > py base at 01:40:32
```

符合报错预期

015

```
integer: 'INTEGER'          Loc=<6:14>
semicolon: ';'              Loc=<6:21>
begin: 'BEGIN'              Loc=<7:4>
if: 'IF'                     Loc=<8:8>
identifier: 'n'              Loc=<8:11>
equal: '='                   Loc=<8:13>
Octal: '0'                  Loc=<8:15>
then: 'THEN'                 Loc=<8:17>
identifier: 'result'         Loc=<9:12>
assign: ':= '                Loc=<9:19>
Octal: '0'                  Loc=<9:22>
elsif: 'ELSIF'              Loc=<10:8>
identifier: 'n'              Loc=<10:14>
equal: '='                   Loc=<10:16>
Unexpected exception:
exceptions.IllegalIntegerRangeException: Integer Is Out of Range
    at OberonScanner.yylex(OberonScanner.java:949)
    at OberonScanner.main(OberonScanner.java:1236)

~/minijava_sysu/exp3/ex2 on main !10 > ./run.sh testcases/fib.015          py base at 01:40:32
comment: '(* Illegal Identifier Length *)'                                Loc=<0:0>
module: 'MODULE'                  Loc=<2:0>
Unexpected exception:
exceptions.IllegalIdentifierLengthException: Identifier Is Too Long
    at OberonScanner.yylex(OberonScanner.java:987)
    at OberonScanner.main(OberonScanner.java:1236)

~/minijava_sysu/exp3/ex2 on main !11 >                                     py base at 01:40:42
```

符合报错预期

通过以上的测试，可以基本判断词法分析器的正确性

同时还提供了 `test.sh` 用来对所有 `testcases` 目录下的进行测试，并将对应结果写入到 `results` 文件夹下

## 对比

之前曾经有使用过 GNU Flex，这里简单说下这两者的异同点，

1. 最明显的，GNU Flex 的目标语言是 C 语言，后续使用 `gcc` 来完成编译，而 JFlex 生成的是 Java 代码，需要使用 `javac` 来完成编译
2. 两者的词法文件都是分割为三个部分，且分隔符都是用的是 `%%`
3. 一些扩展的正则表达式是相同的，例如 `[a-zA-Z]` 在两者都可以使用
4. 配置文件的不同，Flex 的三部分依次为声明(例如一些用户自定义的变量，预处理指令等)，规则(正则表达式)和用户自定义程序(如函数)，而 JFlex 的则是用户程序，选项，规则
5. 由于 GNU Flex 产生的是 C 代码，所以在用户程序中可以直接定义函数入口即 `main` 函数，而 JFlex 则不可以
6. 两者都可以使用 `yytext` 来获取当前匹配的字符串，也都是一些以 `yy` 开头的内置的函数/变量，但在细节上可能有所不同

其它的地方其实都基本相似

而 JLex 和 JFlex 一样，都是生成 Java 代码，且文件分割和 JFlex 相同，个人感觉这两者关系很像是 `flex` 和 `lex`，基本是相同的工具，配置写法也一样，区别主要在于平台上

## 参考文献

1. 扩展的正则表达式，<https://www.runoob.com/regexp/regexp-syntax.html>
2. JFlex 文档，<https://www.jflex.de/manual.html>