1.词法分析

1.1 词法分析概述

词法分析实际上的作用是将原本并无意义的字符串（我们的输入）转化为词素，生成并输出一个个词法单元。

不仅仅是一门语言需要用到词法分析，实际上我们在更多地方都能看见它，比如我们的数据库，SQL语句其实也是一个个的词素的组合，
还有其他可交互平台中，我们在Shell中的输入，实际上也同样会被转化为对应的词法单元，然后再通过语法分析，最后再交由机器执行。

所以词法分析其实就相当于我们阅读文言文的时候，是不是会对大部分的词语进行标注，从而才能为接下来翻译的过程做好准备。

1.2 词法单元

所以让我们先来畅想一下BY-Monkey语言吧。

```
let age = 1;
let b = 1 - 1 + 1 * 1 + 1 / 1;

let t = true;
let name = "BY-Monkey";

let myArray = [1, 2, 3, 4];
let BY = {"name":"炳源","age":21};

puts(age);
puts(t);
puts(name);

puts(myArray[0]);
puts(BY["name"])

let newAdder = fn(x) { fn(y){x + y} };
let addTwo = newAdder(2);

addTwo(3);
```

上述代码中，我们能够大体把遇到的词法单元分为以下几类：

- 操作符：`+,-,*,/,=`

- 标识符，也就是我们的变量名和函数名，我们把他们归为一类
- 关键字：`let,fn`
- 内置函数：`puts`上述例子中只展示了这一个内置函数，不过没事，我们会逐步添加的！
- 特殊字符：`(,),[,],{,},:,EOF`
- 字面量：字面量即`1234`,`"1234"`,`ture`这类数值or字符串，他们也是我们词法单元的一类

确立了我们需要的词法单元，下一步就是编写对应的`token`文件。

这玩意有点像我们的词典，但是不是给我们读的，而是要交给下一个步骤要实现的模块——词法分析器来读的。

`./token/token.go`

```
package token

type TokenType string

const (
    ILLEGAL = "ILLEGAL"
    EOF     = "EOF"

    PLUS     = "+"
    MINUS    = "-"
    ASTERISK = "*"
    SLASH    = "/"

    LPAREN   = "("
    RPAREN   = ")"
    LBRACE   = "{"
    RBRACE   = "}"
    LBRACKET = "["
    RBRACKET = "]"

)

type Token struct {
    Type    TokenType
    Literal string
}
```

上述的`Token`就是我们的词法单元，并将每个基础的符号先做了标记，先不必着急其他的标识符关键字，我们先从它来上手，
了解词法分析器的机制，我们就能很快扩展出我们的各个词法单元了。

大体上确立了我们的 *词典* ，接下来就要来编写我们真正的解释器的第一个模块了。

1.3 词法分析器

词法分析器实际上就相当于一个能够自动帮我们把词典上的注释搬到原本的书本——也就是源代码的一个功能。

经过了这个步骤，我们至少能把原本都看似无规则的语句划分称为一个个的单元，并且能知道这个单元表示的是什么。

我们先来定义一下基础的词法分析器结构：

`./lexer/lexer.go`

```
type Lexer struct {
    input        string
    position     int  // 所输入字符串中的当前位置
    readPosition int  // 所输入字符串中的当前读取位置
    ch           byte // 当前正在读取的字符
}

func New(input string) *Lexer {
    l := &Lexer{input: input}
    l.readChar()
    return l
}

func (l *Lexer) readChar() {
    if l.readPosition >= len(l.input) {
        l.ch = 0
    } else {
        l.ch = l.input[l.readPosition]
    }
    l.position = l.readPosition
    l.readPosition += 1
}
```

虽然注释已经写了，但是还是再解释一遍，首先`input`是我们正在处理的整段代码，`position`则是我们正在分析的位置，
`readPosition`则是一个类似于缓冲区的位置，是我们已经读取但是还没有分析的位置，我们可以通过这个来实现“偷窥”下一个字符的功能，
从而能够更好地判断当前的词法单元是什么。

有了基础结构显然也是不够的，我们还需要准备将这个词法分析器初始化的函数，以及“阅读原文”的对应函数，才能够使这个词法分析器具有能跑起来的能力。

这里我们的`New()`方法和`readChar()`则分别对应着初始化和阅读。

这就是我们伟大的词法分析器的入口了！

但是它现在的能力还不足以识别词法单元。我们还需要对其进行完善，我们继续来编写一个`nextToken()`函数，这个函数能够返回当前正在阅读的词法单元。

我们还是从比较简单的情况开始吧。首先我们来尝试识别一下括号和加减乘除。

`./lexer/lexer.go`

```
func (l *Lexer) skipWhitespace() {
    for l.ch == ' ' || l.ch == '\t' || l.ch == '\n' || l.ch == '\r' {
        l.readChar()
    }
}
```

```
func (l *Lexer) NextToken() token.Token {
    var tok token.Token

    l.skipWhitespace()

    switch l.ch {
    case '+':
        tok = newToken(token.PLUS, l.ch)
    case '-':
        tok = newToken(token.MINUS, l.ch)
    case '/':
        tok = newToken(token.SLASH, l.ch)
    case '*':
        tok = newToken(token.ASTERISK, l.ch)
    case '(':
        tok = newToken(token.LPAREN, l.ch)
    case ')':
        tok = newToken(token.RPAREN, l.ch)
    case '{':
        tok = newToken(token.LBRACE, l.ch)
    case '}':
        tok = newToken(token.RBRACE, l.ch)
    case '[':
        tok = newToken(token.LBRACKET, l.ch)
    case ']':
        tok = newToken(token.RBRACKET, l.ch)
    case 0:
        tok.Literal = ""
        tok.Type = token.EOF

    l.readChar()
    return tok
}
```

首先是`skipWhitespace()`这个函数，我们稍微看一下便能明白其功能，就是将空格和换行等等我们并不需要的，为了让代码更具有可读性的部分跳过。

然后则是`NextToken()`，我们在这个函数中才真正实现了读取并加上“注释”的这一功能，其实这一宏大的功能这样子一看，并不是那么难以理解。

我们对于当前`lexer`正在解析的字符进行判断，并把不同种类的括号与运算符和他们对应的词法单元，然后返回对应的`Token`，便完成了标注。

好的，恭喜！我们已经实现了一个简陋的词法分析器（即便它只能分析括号和加减乘除），但是我们还是需要将对应的测试样例完善好（这也是我的比较不好的习惯？先写代码后写测试代码）

`./lexer/lexer_test.go`

```
package lexer

import (
    "Monkey/token"
    "testing"
```

```
)

func TestNextToken(t *testing.T) {
    input := `()[]{}`

    tests := []struct {
        expectedType    token.TokenType
        expectedLiteral string
    }{
        {token.LPAREN, "("},
        {token.RPAREN, ")"},
        {token.LBRACE, "{"},
        {token.RBRACE, "}"},
        {token.LBRACKET, "["},
        {token.RBRACKET, "]"},
    }

    l := New(input)
    for i, tt := range tests {
        tok := l.NextToken()

        if tok.Type != tt.expectedType {
            t.Fatalf("tests[%d] - tokentype wrong. expected=%q, got=%q",
                i, tt.expectedType, tok.Type)
        }

        if tok.Literal != tt.expectedLiteral {
            t.Fatalf("tests[%d] - literal wrong. expected=%q, got=%q",
                i, tt.expectedLiteral, tok.Literal)
        }
    }
}
```

如预期一样，我们的测试通过了！

但是还没办法开始休息，毕竟哪门语言只有括号啊？！

我们继续添加新的词法单元和识别方法。

1.4 这个解释器需要更多词法单元！

首先如果要有变量，自然要先能够识别到`let`这个变量声明词法单元。

我们先将其添加到`token.go`中

```
package token

type TokenType string

const (
    ILLEGAL = "ILLEGAL"
    EOF     = "EOF"
```

```
    PLUS     = "+"
    MINUS    = "-"
    ASTERISK = "*"
    SLASH    = "/"


    LPAREN   = "("
    RPAREN   = ")"
    LBRACE   = "{"
    RBRACE   = "}"
    LBRACKET = "["
    RBRACKET = "]"

    LET      = "LET"
)

type Token struct {
    Type    TokenType
    Literal string
}
```

但是另外一个棘手的问题也出现了，按照我们刚刚的`lexer`的逻辑，我们得在`switch`中添加一个分支来识别`let`。

可是我们只能识别一个字符！

而且这个问题还必须解决，因为我们基本上所有的关键字，都并非单个字符，所以我们没办法在`case`对`l`这个字符单独识别，然后看看之后几个是不是`et`吧。

同时我们还有例如`==、!=`等符号不仅仅由一个字符组成，所以我们就必须得对其进行特殊处理。

于是我们引入了`peekChar()`用于偷窥下一个字符的函数，有了这个函数，在遇到`=`、`!`时，我们就可以通过调用这个函数，从而判断它是否属于`==`和`!=`的情况了。

但是对更多的字符的关键字，我们就没办法通过`peekChar()`来解决了，所以我们只能对`default`分支做手脚。

我们先编写一个`map`用于存放我们的关键字，不仅仅是`let`，还有其他的我们所需要用到的关键字：

```
// 关键词集合
var keywords = map[string]TokenType{
    "fn":     FUNCTION,
    "let":    LET,
    "true":   TRUE,
    "false":  FALSE,
    "if":     IF,
    "else":   ELSE,
    "return": RETURN,
}
```

编写关键字对应Token环节这里便按下不表了，相信有了之前的各种操作，大伙都能够将其复现出来。

完成了这个，我们就能大大方方地对`default`分支做手脚了：

- 首先如果我们遇到的是字母字符，则它大概率是个字面量or关键字，所以我们将其读取直到 遇到第一个非字母/数字字符。
  这样子我们对于BY-Monkey语言的变量声明就有了一个基础的规定—— **禁止以数字开头**！！！
  - 接着再将其放入刚刚的`keywords`，看其是否在其中，就可以将其从变量与关键字分开。
- 其次如果我们遇到的是数字字符，则它大概率是个数字，我们就一直读取到第一个非数字字符即可停止。

完成了上述两个，如果还有其他的，那么就只剩下非法的情况了。

至此，我们便完成了我们基础的词法分析器。

1.5 必要的测试

完成了基础代码的编写，我非常不好的习惯出现了，我总是习惯于先写代码再编写测试样例，实际上往往是先设计对应的测试样例，再来编写我们的代码会更具针对性一些，
没事，就让我们任性这一次。

我们继续试着编写一小段的`BY-Monkey`语言，然后写出其对应的每个词法单元，再将这段语言输入进我们刚刚编写的`lexer`，获得`lexer`所产出的词法单元，再挨个比较，我们就可以判断我们的代码是否出现纰漏。

```
let five = 5;
let ten = 10;

let add = fn(x, y){
    x + y;
};

let result = add(five, ten);

!-/*5;
5 < 10 > 5;

if (5 < 10) {
    return true;
} else {
    return false;
}

10 == 10;
10 != 9;

"foobar"
"foo bar"

[1,2];
{"foo": "bar"}
```

上面是一段看起来非常奇怪的代码，尽管其中有些部分在之后根本通过不了解释，但是没关系，我们所需要的并非是一段真正的代码，我们只需要
将其对应的各个词法单元进行标注即可。

`lexer_test.go`

```
package lexer

import (
    "Monkey/token"
    "testing"
)

func TestNextToken(t *testing.T) {
    input := `let five = 5;
let ten = 10;

let add = fn(x, y){
    x + y;
};

let result = add(five, ten);

!-/*5;
5 < 10 > 5;

if (5 < 10) {
    return true;
} else {
    return false;
}

10 == 10;
10 != 9;

"foobar"
"foo bar"

[1,2];
{"foo": "bar"}
`

    tests := []struct {
        expectedType    token.TokenType
        expectedLiteral string
    }{
        {token.LET, "let"},
        {token.IDENT, "five"},
        {token.ASSIGN, "="},
        {token.INT, "5"},
        {token.SEMICOLON, ";"},
        {token.LET, "let"},
        {token.IDENT, "ten"},
        {token.ASSIGN, "="},
        {token.INT, "10"},
        {token.SEMICOLON, ";"},
        {token.LET, "let"},
        {token.IDENT, "add"},
        {token.ASSIGN, "="},
```

```
        {token.FUNCTION, "fn"},
        {token.LPAREN, "("},
        {token.IDENT, "x"},
        {token.COMMA, ","},
        {token.IDENT, "y"},
        {token.RPAREN, ")"},
        {token.LBRACE, "{"},
        {token.IDENT, "x"},
        {token.PLUS, "+"},
        {token.IDENT, "y"},
        {token.SEMICOLON, ";"},
        {token.RBRACE, "}"},
        {token.SEMICOLON, ";"},
        {token.LET, "let"},
        {token.IDENT, "result"},
        {token.ASSIGN, "="},
        {token.IDENT, "add"},
        {token.LPAREN, "("},
        {token.IDENT, "five"},
        {token.COMMA, ","},
        {token.IDENT, "ten"},
        {token.RPAREN, ")"},
        {token.SEMICOLON, ";"},
        {token.BANG, "!"},
        {token.MINUS, "-"},
        {token.SLASH, "/"},
        {token.ASTERISK, "*"},
        {token.INT, "5"},
        {token.SEMICOLON, ";"},
        {token.INT, "5"},
        {token.LT, "<"},
        {token.INT, "10"},
        {token.GT, ">"},
        {token.INT, "5"},
        {token.SEMICOLON, ";"},
        {token.IF, "if"},
        {token.LPAREN, "("},
        {token.INT, "5"},
        {token.LT, "<"},
        {token.INT, "10"},
        {token.RPAREN, ")"},
        {token.LBRACE, "{"},
        {token.RETURN, "return"},
        {token.TRUE, "true"},
        {token.SEMICOLON, ";"},
        {token.RBRACE, "}"},
        {token.ELSE, "else"},
        {token.LBRACE, "{"},
        {token.RETURN, "return"},
        {token.FALSE, "false"},
        {token.SEMICOLON, ";"},
        {token.RBRACE, "}"},
        {token.INT, "10"},
```

```
        {token.EQ, "=="},
        {token.INT, "10"},
        {token.SEMICOLON, ";"},
        {token.INT, "10"},
        {token.NOT_EQ, "!="},
        {token.INT, "9"},
        {token.SEMICOLON, ";"},
        {token.STRING, "foobar"},
        {token.STRING, "foo bar"},
        {token.LBRACKET, "["},
        {token.INT, "1"},
        {token.COMMA, ","},
        {token.INT, "2"},
        {token.RBRACKET, "]"},
        {token.SEMICOLON, ";"},
        {token.LBRACE, "{"},
        {token.STRING, "foo"},
        {token.COLON, ":"},
        {token.STRING, "bar"},
        {token.RBRACE, "}"},
        {token.EOF, ""},
    }

    l := New(input)
    for i, tt := range tests {
        tok := l.NextToken()

        if tok.Type != tt.expectedType {
            t.Fatalf("tests[%d] - tokentype wrong. expected=%q, got=%q",
                i, tt.expectedType, tok.Type)
        }

        if tok.Literal != tt.expectedLiteral {
            t.Fatalf("tests[%d] - literal wrong. expected=%q, got=%q",
                i, tt.expectedLiteral, tok.Literal)
        }
    }
}
```

上述便是我们的测试代码了，很幸运，我们一次就通过了这个测试：

```
=== RUN   TestNextToken
--- PASS: TestNextToken (0.00s)
PASS
```