《编译原理》专题1设计

目标任务

实验项目

以下为正则文法所描述的 C 语言子集单词符号的示例，请补充单词符号： ++，--， ，<<, += , -= ,*=, /= ， &&（逻辑与），||（逻辑或），！（逻辑非）等等，给 出补充后描述 C 语言子集单词符号的正则文法，设计并实现其词法分析程序。

该语言的保留字： void、int、float、double、if、else、for、do、while 等等（也可 补充）。

设计说明

1. 可将该语言设计成大小写不敏感，也可设计成大小写敏感，用户定 义的标识符最长不超过 32 个字符；
2. 字母为 a-z A-Z，数字为 0-9；
3. 可以对上述文法 进行扩充和改造；
4. “/……/”和“//”(一行内)为程序的注释部分。

设计要求

1. 给出各单词符号的类别编码；
2. 词法分析程序应能发现输入串中的 错误；
3. 词法分析作为单独一遍编写，词法分析结果为二元式序列组成的中间文件；
4. 设计两个测试用例（尽可能完备），并给出测试结果。

程序功能描述

1. 自定义单词的匹配规则和名称, 具体包括标识符, 整数, +-*/运算符, 两种注释类型
2. 根据正则表达式识别输入串的单词
3. 将识别的单词和名称以二元式的方式输出到中间文件
4. 将标识符替换成保留字

数据结构

Token保存单词的正则表达式和名称

```
type Token struct {
    ExpStr    string `json:"expStr"` //单词的正则表达式
    TokenName string `json:"token"`  //单词的名称
}
```

Rule保存所有单词的Token和保留字

```
type Rule struct {
    Tokens        []Token  `json:"tokens"`        //单词token结构体数组
    ReservedWords []string `json:"reservedWords"` //保留字
}
```

TokenCompiled保存编译过的正则表达式

```
type TokenCompiled struct {
    Token                     //单词token
    ExpCompiled regexp.Regexp //编译过的正则表达式
}
```

TokenMatchText保存匹配到的输入串和名称的二元式

```
 type TokenMatchText struct {
    TokenName string //单词名称
    text      []byte //匹配到的输入串
}
```

程序结构描述

以下仅为部分代码, 完整程序请参考源代码

主函数

```
//循环扫描输入字符串, 输出匹配到的字符和名称二元式
for {
        if lexScanner.pos == len(context) {
            break
        }
        res := lexScanner.findFromExps(context)
        outPutFile.Write([]byte(string(res.text) + ": " + res.TokenName +
";\n"))
    }
```

匹配输入字符串中的单词

```
func (lexScanner *LexScanner) findFromExps(context []byte) TokenMatchText {
    for _, item := range lexScanner.TokensCompiled {
        res := item.ExpCompiled.FindSubmatch(context[lexScanner.pos:])
        if res == nil {
            continue
        } //从开头调用正则表达式匹配式引擎
        var matchedRes []byte
        if len(res) == 1 {
            matchedRes = res[0]
        } else if len(res) == 2 {
            matchedRes = res[1]
        }
        lexScanner.pos += len(matchedRes)
        fmt.Println(string(matchedRes), ": ", item.TokenName, ";")
        tokenMatchText := TokenMatchText{
            TokenName: item.TokenName,
            text:      matchedRes}
        return tokenMatchText //返回二元式
    }
    panic("invalid lex position: " +
string(context[lexScanner.pos:lexScanner.pos+10])) //如果输入不符合语法规则报
错，显示出错的位置
}
```

单词正则表达式

确定单词的正则表达式匹配规则，以json文件的形式保存.

```
{
  "tokens": [
    {
      "expStr": "\\A[a-zA-Z]+\\w*", //正则表达式匹配规则
      "token": "IDENTIFIER"          //单词名称
    },
    {
      "expStr": "\\A[0-9]+\\w",
      "token": "UNSIGNEDINT"
    },
    //省略
    "reservedWords": ["main", "for", "if", "else", "int", "double"] //保留
字，可拓展
}
```

解析正则表达式

```
    ruleByte, err := ioutil.ReadFile("rule.json") //打开json文件
    // error处理
    err = json.Unmarshal(ruleByte, &rule) //解析json格式
    // error处理
    var tokensCompiled []TokenCompiled
    for _, item := range rule.Tokens { //将正则表达式和对应的token名称保存
        //赋值语句
    }
```

输入输出

```
    context, err := ioutil.ReadFile("input.txt") //读取输入串
    //error处理
    err = ioutil.WriteFile("output.txt", []byte{}, 0777) //打印输出二元式序列
组
    outPutFile, err := os.OpenFile("output.txt", os.O_RDWR, 0777)
    //error处理
```

测试用例

测试用例1

- 输入文件input.txt:

```
void main my_word
!= /
//你好 , 0p
/*
fa
fa*/
```

- 输出文件:output.txt

```
void: IDENTIFIER;
 : SPACE;
main: IDENTIFIER;
 : SPACE;
my_word: IDENTIFIER;

: SPACE;
!=: NE;
 : SPACE;
/: SLASH;
```

```
: SPACE;
//你好 , 0p
: COMMENT;
/*
fa
fa*/: COMMENT;
```

测试用例2

- 输入test.c

```
    void main(){
    int num1 = 123;
    int num2 = 123;
    float num3 = 1234;
    double num4 = 12345;
    for(int i = 0;i \< 100;++i){
    do{
    if(num1==num2\|\|num3!=num4){
    \--num2;
    return 0;
    }else{
    return num1\&=num2;
    }
    }while(num1&&num2)
    }
    //单行注释
    int d = count1\<\<5;
    int c = count2\ \ 6;
    int  x = count1+count2;
    int y = count1-count2;
    int c = count1\*count2;
    int d = count1/count2;
     x+=y;
     x-=y;
     x\*=y;
     x/=y;
     x\|=y;
     x\&=y;
}
/\*\*
    左右注释
    \* \*/
```

- 输出文件output.txt

```
     :  SPACE ;
void :  IDENTIFIER ;
  :  SPACE ;
main :  IDENTIFIER ;
( :  SINGLEDELIMITER ;
) :  SINGLEDELIMITER ;
{ :  SINGLEDELIMITER ;

     :  SPACE ;
int :  IDENTIFIER ;
  :  SPACE ;
num1 :  IDENTIFIER ;
  :  SPACE ;
--- FAIL: TestScan (0.01s)
panic: invalid lex position: = 123;
   [recovered]
        panic: invalid lex position: = 123;
```

源代码

```go
//lex.go
package lab1

import (
    "encoding/json"
    "fmt"
    "io/ioutil"
    "os"
    "regexp"
)

func scan() {
    // context := []byte("abc 12 d34 {")
    // token := []string{"IDENTIFIER"}
    // var context []byte
    ruleByte, err := ioutil.ReadFile("rule.json") //打开json文件
    if err != nil {
        fmt.Println("open rule.json err")
    }
    var rule Rule
    err = json.Unmarshal(ruleByte, &rule) //解析json格式
    if err != nil {
        fmt.Println("unmarshal json err")
    }
    var tokensCompiled []TokenCompiled
    for _, item := range rule.Tokens { //将正则表达式和对应的token名称保存
        fmt.Println(item.ExpStr)
        exp := regexp.MustCompile(item.ExpStr)
```

```
        tokenCompiled := TokenCompiled{item, *exp}
        tokensCompiled = append(tokensCompiled, tokenCompiled)
    }
    lexScanner := &LexScanner{}
    lexScanner.TokensCompiled = tokensCompiled
    lexScanner.pos = 0

    context, err := ioutil.ReadFile("input.txt") //读取输入串
    if err != nil {
        fmt.Println("read file error!")
    }
    err = ioutil.WriteFile("output.txt", []byte{}, 0777) //打印输出二元式序列
组
    outPutFile, err := os.OpenFile("output.txt", os.O_RDWR, 0777)
    if err != nil {
        fmt.Println("open output err")
    }

    for {
        if lexScanner.pos == len(context) {
            break
        }
        res := lexScanner.findFromExps(context)
        outPutFile.Write([]byte(string(res.text) + ": " + res.TokenName +
";\n"))
    }
}

type LexScanner struct {
    pos            int
    TokensCompiled []TokenCompiled
}

func (lexScanner *LexScanner) findFromExps(context []byte) TokenMatchText {
    for _, item := range lexScanner.TokensCompiled {
        res := item.ExpCompiled.FindSubmatch(context[lexScanner.pos:])
        if res == nil {
            continue
        }
        var matchedRes []byte
        if len(res) == 1 {
            matchedRes = res[0]
        } else if len(res) == 2 {
            matchedRes = res[1]
        }
        lexScanner.pos += len(matchedRes)
        fmt.Println(string(matchedRes), ": ", item.TokenName, ";")
        tokenMatchText := TokenMatchText{
            TokenName: item.TokenName,
```

```
            text:      matchedRes}
        return tokenMatchText
    }
    panic("invalid lex position: " +
string(context[lexScanner.pos:lexScanner.pos+10]))
}

func (lexScanner *LexScanner) replaceReservedWord(reservedWords *[]string) {
    _, err := ioutil.ReadFile("input.txt")
    if err != nil {
        fmt.Println("read file error!")
    }
    fmt.Println(reservedWords)
}

type TokenMatchText struct {
    TokenName string //单词名称
    text      []byte //匹配到的输入串
}

type Rule struct {
    Tokens        []Token  `json:"tokens"`        //单词token结构体数组
    ReservedWords []string `json:"reservedWords"` //保留字
}
type Token struct {
    ExpStr    string `json:"expStr"` //单词的正则表达式
    TokenName string `json:"token"`  //单词的名称
}

type TokenCompiled struct {
    Token                    //单词token
    ExpCompiled regexp.Regexp //编译过的正则表达式
}

//lex_test.go
package lab1

import (
    "fmt"
    "io/ioutil"
    "regexp"
    "testing"
)

func TestScan(t *testing.T) {
    t.Log("testing...")
    scan()
}
```

```
func TestReg(t *testing.T) {
    s := []byte("/* \nad*/")
    re := regexp.MustCompile("\\A/\\*[\\s\\S]*\\*/")
    res := re.FindSubmatch(s)
    fmt.Println(res)
    t.Log(res)
    for _, item := range res {
        fmt.Println(string(item))
    }
}

func TestReservedWords(t *testing.T) {
    _, err := ioutil.ReadFile("rule.txt")
    if err != nil {
        t.Log("error!")
    }

}

//rule.json
{
  "tokens": [
    {
      "expStr": "\\A[a-zA-Z]+\\w*",
      "token": "IDENTIFIER"
    },
    {
      "expStr": "\\A[0-9]+\\w",
      "token": "UNSIGNEDINT"
    },
    {
      "expStr": "\\A[\\+\\-\\*\\;\\(\\)\\,\\{\\}]",
      "token": "SINGLEDELIMITER"
    },
    {
      "expStr": "^\\s+",
      "token": "SPACE"
    },
    {
      "expStr": "(\\A<)(?:[^=>])",
      "token": "LT"
    },
    {
      "expStr": "(\\A>)(?:[^=])",
      "token": "GT"
    },
    {
      "expStr": "\\A<>",
      "token": "NE"
```

```
    },
    {
      "expStr": "\\A!=",
      "token": "NE"
    },
    {
      "expStr": "(\\A/)(?:[^/*])",
      "token": "SLASH"
    },
    {
      "expStr": "\\A//.*\\r\\n",
      "token": "COMMENT"
    },
    {
      "expStr": "\\A/\\*[\\s\\S]*\\*/",
      "token": "COMMENT"
    }
  ],
  "reservedWords": ["main", "for", "if", "else", "int", "double"]
}
```