编译原理 – 实验 1 - 词法分析

61519322 杨哲睿

日期：October 23, 2021

目录

1 引言

1.1 实验要求

选择一个你熟悉的程序设计语言，找到它的规范（referrence or standard）。在规范中找到其词法的 BNF 或正规式描述。

评论. 选择的程序设计语言为：`pascal`（源语言），具体规范参考为 Github-pascal 词法的 lex 描述。

选择该语言的一个子集（能够构成一个 mini 的语言，该语言至少能够进行函数调用、控制流语句（分支或循环）、简单的运算和赋值操作。）给出该 mini 语言的词法的正规文法或正规式。

在这里，选择的 mini 语言关键字为：

```
and
begin
div
do
else
end
for
```

```
function
if
in
nil
not
of
procedure
program
repeat
set
then
to
until
var
while
```

与标识符一同识别。

relop 如下：

```
<= >= < > <> =
:=
..
+ - * /
```

界符 sep 如下：

```
[ ] ( )
, ; .
```

其他需要的基本词法单元如下：

1. 数
2. 字符串

下面给出具体的词法规则描述。

2 词法规则描述、Minimized-DFA 简化过程

如图所示：

3 词法分析程序

3.1 简述

正如 DFA 的定义：描述 DFA 只需要五元组 $S = (K, \Sigma, f, S, Z)$。那么描述 DFA 中的状态只需要状态转换函数：

$$change_S(\text{ch}) = f(S, \text{ch})$$

这适合于将函数作为第一公民的**函数式编程**来实现。同时，对于 $change_S(\text{ch})$ 做出如下的约定：

$change_S(\text{ch}) =$

1. 若 DFA 中存在 $S' = f(S, \text{ch})$ 那么输出为对应状态（的状态转换函数）即输出为 $changeS'$
2. 若 DFA 中不存在这样的转换，那么若

   $S \in Z$ 输出 `true`；

   $S \notin Z$ 输出 `false`。

   例如：

例 3.1. 字符串的识别：

正如上文所提到的，将手工简化好的 DFA 转换为代码，即为：

Listing 1: Racket-lang 实现-用于匹配字符串的 DFA 定义

```
; DFA -- string
(define StringSM
(let ([q? (lambda (ch) (char=? ch #\'))])
    (letrec ([s0 (lambda (ch) (cond
                                [(q? ch) s1]
                                [else #f]))]
             [s1 (lambda (ch) (cond
                                [(q? ch) s1]
                                [(char=? #\nul ch) #f]
                                [else s3]))]
             [s2 (lambda (ch) (cond
                                [(q? ch) s3]
                                [else #f]))]
             [s3 (lambda (ch) (cond
                                [(q? ch) s4]
                                [(char=? #\nul ch) #f]
                                [else s3]))]
             [s4 (lambda (ch) (cond
                                [(q? ch)s3]
                                [else #t]))])
    s0)))
```

与此同时，为了运行该自动机，完成了`iter-until-fail`函数，该函数的作用是在给定的符号列上运行 DFA，直到 DFA 无法识别当前符号时停止。

Listing 2: 在给定的符号列上，运行该状态机

```
(define (iter-until-fail s l)
  (let* ([ch (if (empty? l)
                 #\null
                 (first l))] ; getch --> ch
         [t (s ch)])         ; transition {s --[ch]-> t}
    (if (boolean? t)         ; stop?
        (if t
            (list)           ; '()  -> accept
            (void))          ; void -> fail
        (let ([result (iter-until-fail t (rest l))])
          (if (void? result)
              (void)         ; -> fail matching
              (cons ch result))
```

```
        ))))
```

3.2 实现细节

3.2.1 标识符-关键字的特殊处理

3.2.2 数组大小约定的数字识别 – 向前看

3.3 测试用例和测试结果

4 总结