

编译原理 - 实验 1 - 词法分析

61519322 杨哲睿

日期: October 26, 2021

目录

1 引言	1
1.1 实验要求	1
2 词法规则描述、Minimized-DFA 简化过程	2
3 词法分析程序	6
3.1 简述	6
3.2 实现细节	7
3.2.1 标识符-关键字的特殊处理	7
3.2.2 数组大小约定的数字识别 – 向前看	7
3.3 词法分析函数 <code>get-token</code>	7
3.4 测试用例和测试结果	8
3.4.1 测试 1 – 数字、字符串、标识符	8
3.4.2 测试 2 – <code>relop</code> 的识别	10
3.4.3 测试 3 – 数组大小约定识别	11
4 总结	12

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（reference 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：

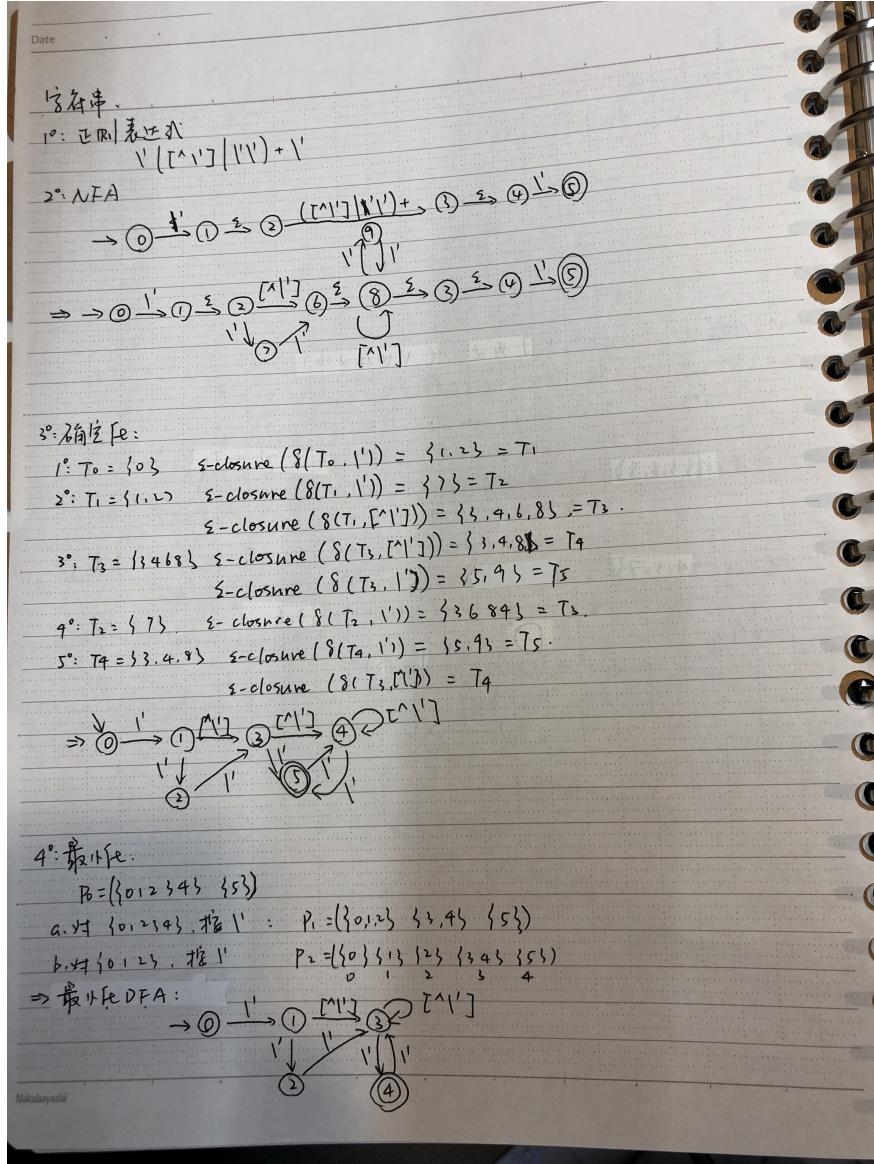


图 1: 字符串

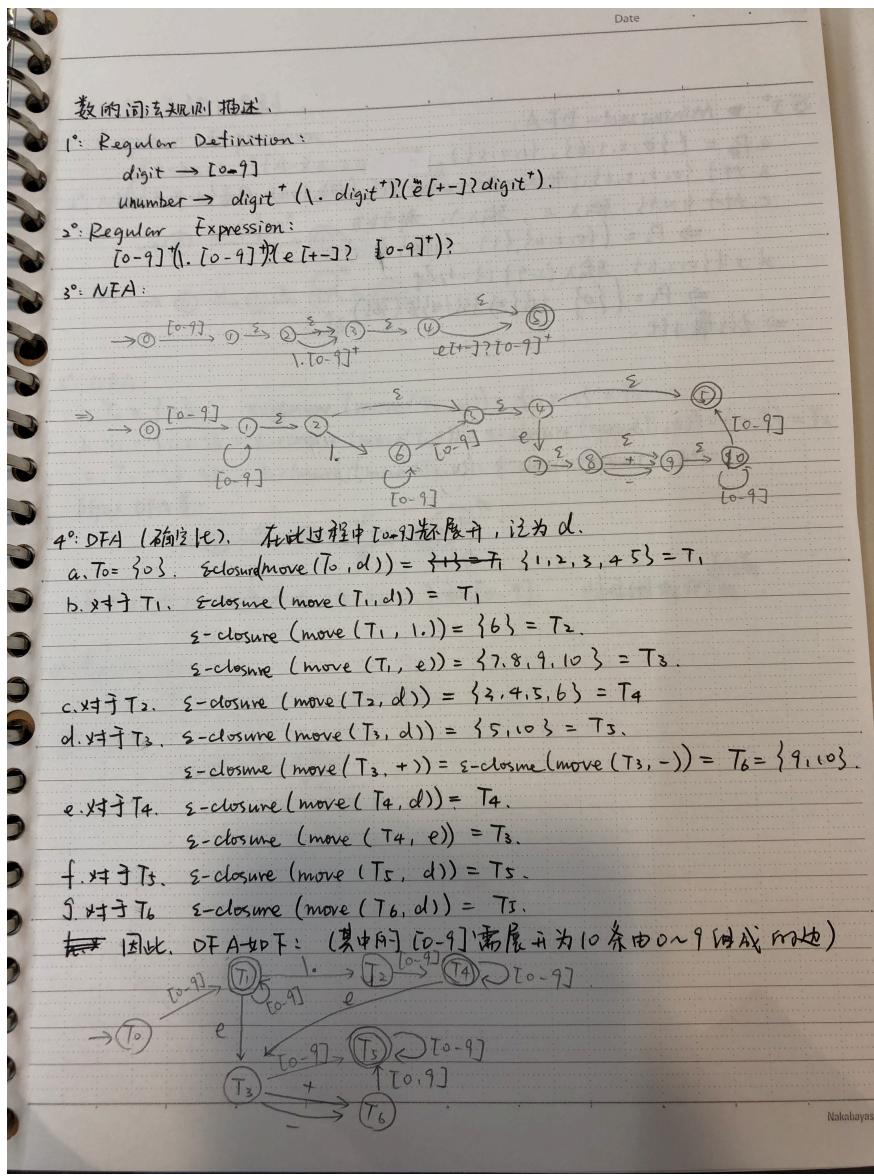


图 2: 数字识别 1

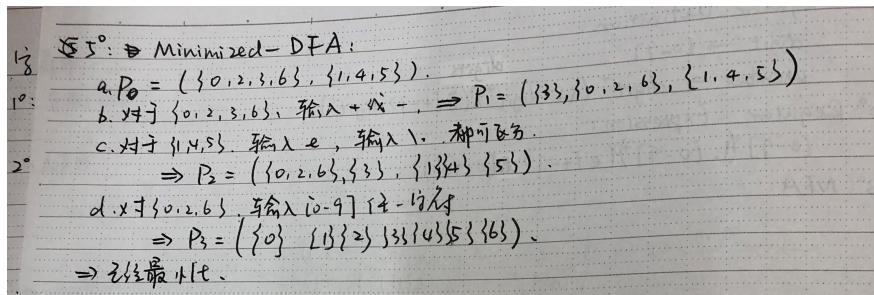


图 3: 数字识别 2

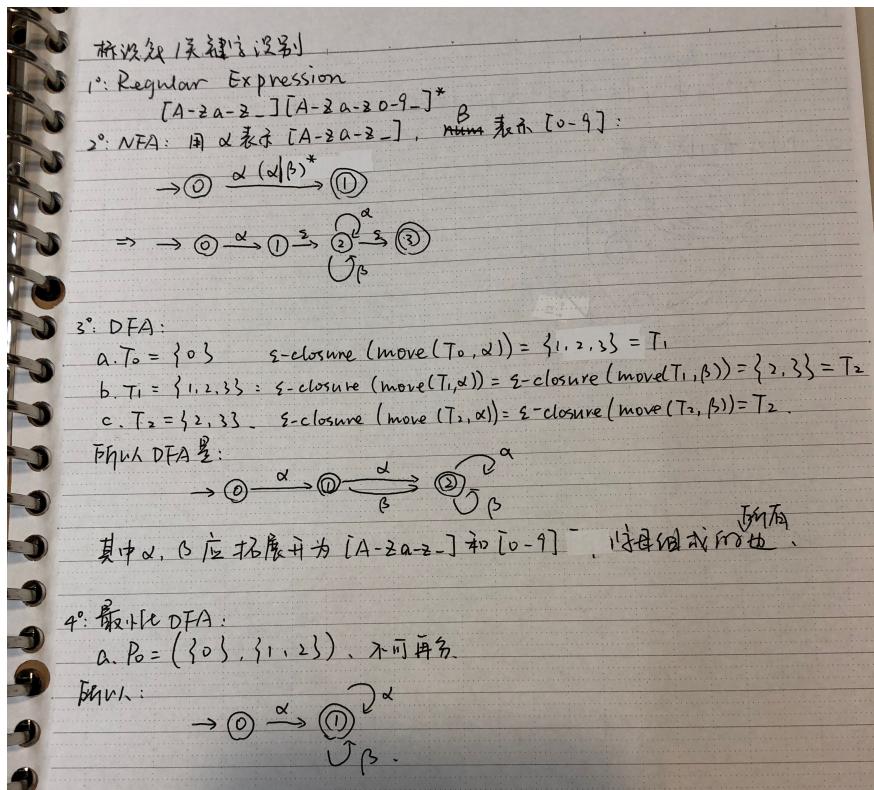


图 4: 标识符/关键字识别

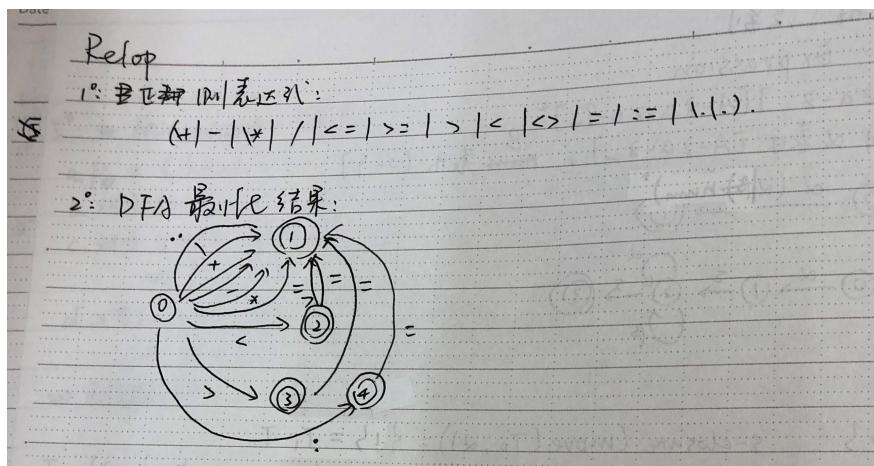


图 5: Relop 识别

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：

$$change_S(ch) = f(S, ch)$$

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 $change_S(ch)$ 做出如下的约定：

$change_S(ch) =$

1. 若 DFA 中存在 $S' = f(S, ch)$ 那么输出为对应状态（的状态转换函数）即输出为 $change_{S'}$
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) (= ch #\'))])
    (letrec ([s0 (lambda (ch) (cond
                                [(q? ch) s1]
                                [else #f]))]
            [s1 (lambda (ch) (cond
                                [(q? ch) s1]
                                [(= ch #\nul) #f]
                                [else s3]))]
            [s2 (lambda (ch) (cond
                                [(q? ch) s3]
                                [else #f]))]
            [s3 (lambda (ch) (cond
                                [(q? ch) s4]
                                [(= ch #\nul) #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 1)]) ; 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 1))])
  (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

正如龙书中所说的，一般而言，关键字的识别可以通过在识别出一个标识符之后进行后处理，判断各个标识符是否是关键字，如果是关键字那么就返回关键字，反之判定为标识符。

3.2.2 数组大小约定的数字识别 – 向前看

一个特情况是这样的：

```
var a: array [1..4] of int;
```

考虑 1..4 的判断，首先看1，得到的是需要做数字的匹配。在第二个 . 处无法进行识别。且第一个识别出的1. 实际语义是错误的。因此需要另外识别，即如果判断到数字、以 . 结尾，且下一个紧接的是 .，那么识别为数组大小约定的 ...。

3.3 词法分析函数 get-token

```

(define (get-token str)
  (let* ([buffer (string->list str)]
         [s (first buffer)]) ; s: the first char in buffer
  (cond
   ; Matching whitespace
   [(ws? s)
    (let ([matched (Whitespace! buffer)])
      (token (list->string matched) token-whitespace (void)))]
   ; Matching separator
   [(try-sep s)
    (token (list->string (list s)) token-sep (try-sep s))]
   ; Matching numbers
   [(digit? s)
    (let ([matched (Number! buffer)])
      (if (void? matched)
          ; Look Forward.

```

```

(let ([matched (ArrayIndex! buffer)])
  (if (void? matched)
      (void)
      (if (and
            (>= (string-length str) (+ 2 (length matched)))
            (string=? ".." (substring str (length matched) (+ 2 (length matched)
                                                               )))))
          (token (list->string matched) token-number (void))
          (void))))
  (token (list->string matched) token-number (void))
  ))
[(rellop-letter? s)
 (let ([matched (Relop! buffer)])
  (if (void? matched)
      (void)
      (let ([content (list->string matched)])
       (token content token-rellop (try-rellop content))))]
 ; Matching identifier/keyword
 [(letter_? s)
  (let ([matched (Identifier! buffer)])
   (if (void? matched)
       (void)
       (let* ([content (list->string matched)]
              [kw (try-keyword content)])
        (if kw
            (token content token-keyword kw)
            (token content token-identifier content))
        ))))]
 [(char=? #\' s)
  (let ([matched (String! buffer)])
   (if (void? matched)
       (void)
       (let ([content (list->string matched)])
        (token content token-string content))))]
 [else (println "Error."))])

```

3.4 测试用例和测试结果

3.4.1 测试 1 – 数字、字符串、标识符

测试用例如下：

Listing 3: test1.pas

```

a := b + 1;
c = '123';
i0123_abc
'3124','123',
1.3245
1e3

```

```
1e-3  
1.0e3  
13  
-13
```

测试结果如下：

```
Content of test1.pas is:  
a := b + 1;  
c = '123';  
i0123_abc  
'3124','123'  
1.3245  
1e-3  
1e-3  
1.0e3  
13  
-13  
  
##### RESULT OF LEXER #####  
  
#(struct:token a #<token-kind:token-identifier> a)  
#(struct:token <whitespace> #<token-kind:token-whitespace> #<void>)  
#(struct:token := #<token-kind:token-relop> assign)  
#(struct:token <whitespace> #<token-kind:token-whitespace> #<void>)  
#(struct:token b #<token-kind:token-identifier> b)  
#(struct:token <whitespace> #<token-kind:token-whitespace> #<void>)  
#(struct:token + #<token-kind:token-relop> plus)  
#(struct:token <whitespace> #<token-kind:token-whitespace> #<void>)  
#(struct:token 1 #<token-kind:token-number> #<void>)  
#(struct:token ; #<token-kind:token-sep> semicolon)  
#(struct:token <whitespace> #<token-kind:token-whitespace> #<void>)  
#(struct:token c #<token-kind:token-identifier> c)  
#(struct:token <whitespace> #<token-kind:token-whitespace> #<void>)  
#(struct:token = #<token-kind:token-relop> eq)  
#(struct:token <whitespace> #<token-kind:token-whitespace> #<void>)  
#(struct:token '123' #<token-kind:token-string> '123')  
#(struct:token ; #<token-kind:token-sep> semicolon)  
#(struct:token <whitespace> #<token-kind:token-whitespace> #<void>)  
#(struct:token i0123_abc #<token-kind:token-identifier> i0123_abc)  
#(struct:token <whitespace> #<token-kind:token-whitespace> #<void>)  
#(struct:token '3124','123' #<token-kind:token-string> '3124','123')  
#(struct:token <whitespace> #<token-kind:token-whitespace> #<void>)  
#(struct:token 1.3245 #<token-kind:token-number> #<void>)  
#(struct:token <whitespace> #<token-kind:token-whitespace> #<void>)  
#(struct:token 1e3 #<token-kind:token-number> #<void>)  
#(struct:token <whitespace> #<token-kind:token-whitespace> #<void>)  
#(struct:token 1e-3 #<token-kind:token-number> #<void>)  
#(struct:token <whitespace> #<token-kind:token-whitespace> #<void>)  
#(struct:token 1.0e3 #<token-kind:token-number> #<void>)  
#(struct:token <whitespace> #<token-kind:token-whitespace> #<void>)
```

```

#(struct:token 13 #<token-kind:token-number> #<void>)
#(struct:token <whitespace> #<token-kind:token-whitespace> #<void>)
#(struct:token - #<token-kind:token-relop> minus)
#(struct:token 13 #<token-kind:token-number> #<void>)
#(struct:token <whitespace> #<token-kind:token-whitespace> #<void>)

```

可以看到，我的词法分析程序对于空白字符和各个标识符、数字、赋值:= 和判断=都进行了正确的判断。(assign 和 eq)

3.4.2 测试 2-relop的识别

第二个测试主要测试对于符号的识别：

Listing 4: test2.pas

```

> >= < <= = <>
:= ..
+ - * /
[ ] ( )
, ; .

```

测试结果如下：

```

Content of test2.pas is:
> >= < <= = <>
:= ..
+ - * /
[ ] ( )
, ; .

#####
##### RESULT OF LEXER #####
#####

#(struct:token > #<token-kind:token-relop> gt)
#(struct:token <whitespace> #<token-kind:token-whitespace> #<void>)
#(struct:token > #<token-kind:token-relop> gt)
#(struct:token = #<token-kind:token-relop> eq)
#(struct:token <whitespace> #<token-kind:token-whitespace> #<void>)
#(struct:token < #<token-kind:token-relop> lt)
#(struct:token <whitespace> #<token-kind:token-whitespace> #<void>)
#(struct:token <= #<token-kind:token-relop> leq)
#(struct:token <whitespace> #<token-kind:token-whitespace> #<void>)
#(struct:token = #<token-kind:token-relop> eq)
#(struct:token <whitespace> #<token-kind:token-whitespace> #<void>)
#(struct:token <> #<token-kind:token-relop> neq)
#(struct:token <whitespace> #<token-kind:token-whitespace> #<void>)
#(struct:token := #<token-kind:token-relop> assign)
#(struct:token <whitespace> #<token-kind:token-whitespace> #<void>)
#(struct:token .. #<token-kind:token-relop> double-dot)
#(struct:token <whitespace> #<token-kind:token-whitespace> #<void>)
#(struct:token + #<token-kind:token-relop> plus)
#(struct:token <whitespace> #<token-kind:token-whitespace> #<void>)
#(struct:token - #<token-kind:token-relop> minus)

```

```

#(struct:token <whitespace> #<token-kind:token-whitespace> #<void>)
#(struct:token * #<token-kind:token-relop> multi)
#(struct:token <whitespace> #<token-kind:token-whitespace> #<void>)
#(struct:token / #<token-kind:token-relop> divide)
#(struct:token <whitespace> #<token-kind:token-whitespace> #<void>)
#(struct:token [ #<token-kind:token-sep> square-lbracket)
#(struct:token <whitespace> #<token-kind:token-whitespace> #<void>)
#(struct:token ] #<token-kind:token-sep> square-rbracket)
#(struct:token <whitespace> #<token-kind:token-whitespace> #<void>)
#(struct:token ( #<token-kind:token-sep> lbracket)
#(struct:token <whitespace> #<token-kind:token-whitespace> #<void>)
#(struct:token ) #<token-kind:token-sep> rbracket)
#(struct:token <whitespace> #<token-kind:token-whitespace> #<void>)
#(struct:token , #<token-kind:token-sep> comma)
#(struct:token ; #<token-kind:token-sep> semicolon)
#(struct:token <whitespace> #<token-kind:token-whitespace> #<void>)
#(struct:token . #<token-kind:token-relop> dot)

```

可见对于rellop和界符、运算符的词法分析的正确性。

3.4.3 测试3 – 数组大小约定识别

对于前文提到的数组大小约定，我们用这个特殊的例子来证明我的词法分析程序的正确性：

Listing 5: test3.pas

```
var a: array[1..4, 3 .. 5] of integer;
```

可以从下面的输出看到，词法分析程序对于..都做出了正确的判断。

```

Content of test3.pas is:
var a: array[1..4, 3 .. 5] of integer;
#####
#<token-kind:token-keyword> var
#<token-kind:token-whitespace> #<void>
#<token-kind:token-identifier> a
#<token-kind:token-relop> #
#<token-kind:token-whitespace> #<void>
#<token-kind:token-identifier> array
#<token-kind:token-sep> square-lbracket)
#<token-kind:token-number> #<void>
#<token-kind:token-relop> double-dot)
#<token-kind:token-number> #<void>
#<token-kind:token-sep> comma)
#<token-kind:token-whitespace> #<void>
#<token-kind:token-number> #<void>
#<token-kind:token-whitespace> #<void>
#<token-kind:token-relop> double-dot)
#<token-kind:token-whitespace> #<void>
#<token-kind:token-number> #<void>

```

```
#(struct:token ] #<token-kind:token-sep> square-rbracket)
#(struct:token <whitespace> #<token-kind:token-whitespace> #<void>)
#(struct:token of #<token-kind:token-keyword> of)
#(struct:token <whitespace> #<token-kind:token-whitespace> #<void>)
#(struct:token integer #<token-kind:token-identifier> integer)
#(struct:token ; #<token-kind:token-sep> semicolon)
```

4 总结

说说为啥用了 Racket：我参考过很多往期的报告，他们大多都用了 c 作为实现语言，实现的也是对于 c 的词法分析，但我觉得词法分析程序中的自动机应用并不应该用 c++ 的变量标识，然后依靠判断语句来对自动机来模拟运行。所以我第一个排除的实现语言就是面向对象、面向过程等范式来实现词法分析，首先也排除的也是如 C/C++、Java 这些强调面向过程、面向对象的语言。原本我想的是用 python 这些脚本语言，他们有函数式编程的基本的支持，但这依然不足以能够使得我脑子里面的状态即函数的想法实现，因为它们依然不将函数作为第一公民中的”核心”。在最后我选择了一个相对陌生的语言 – SICP 中的开发工具 Racket。

实际上在纸上实现完 DFA 状态转换图所有算法的核心就已经结束了。其他的工作都是一些细枝末节的处理、语法分析的控制程序和输入输出接口。这是一个很有意思的大作业，不得不说，因此我也将该代码发布到了 [我的个人 github](#) 上进行长期的存档。

Functional Programming – 舒服了！