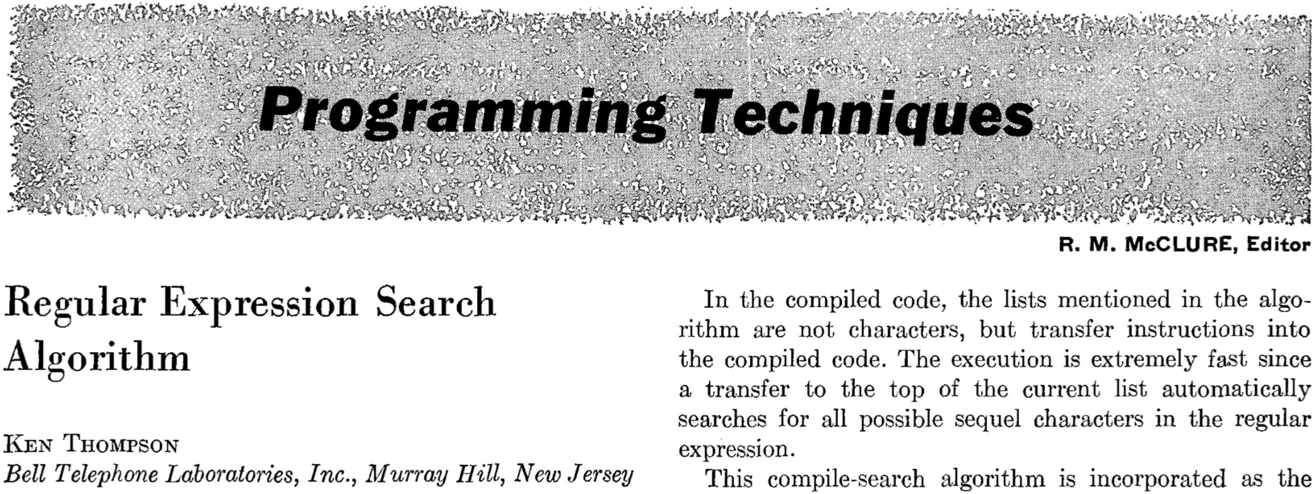
第61回 プログラミング・シンポジウム

2020-01-11

@ラフォーレ伊東

Ken Thompsonの正規表現探索アルゴリズムを解剖する

IIJ 技術研究所　和田 英一



Programming Techniques

R. M. McCLURE, Editor

Regular Expression Search Algorithm

KEN THOMPSON
Bell Telephone Laboratories, Inc., Murray Hill, New Jersey

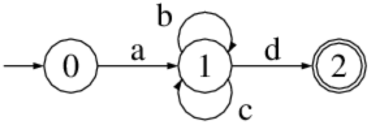
In the compiled code, the lists mentioned in the algorithm are not characters, but transfer instructions into the compiled code. The execution is extremely fast since a transfer to the top of the current list automatically searches for all possible sequel characters in the regular expression.

This compile-search algorithm is incorporated as the

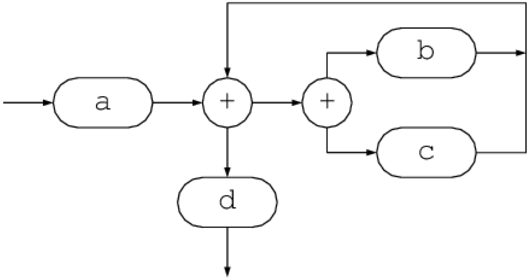
計算機考古学

パラメトロン計算機PC-1 --回路設計と方式設計-- 1996年夏
独断的プログラムリファインメント 1997年夏
中西流「目でみるGC」再現 2006年夏
セルオートマトンのプログラムハック 2012年夏
GPMとそのプログラム 2015年冬
PDP-8の再評価と再構成 2018年夏
リレーによる演算回路 2018年夏
Mercedes Euklid 2020年冬

正規表現探索プログラム

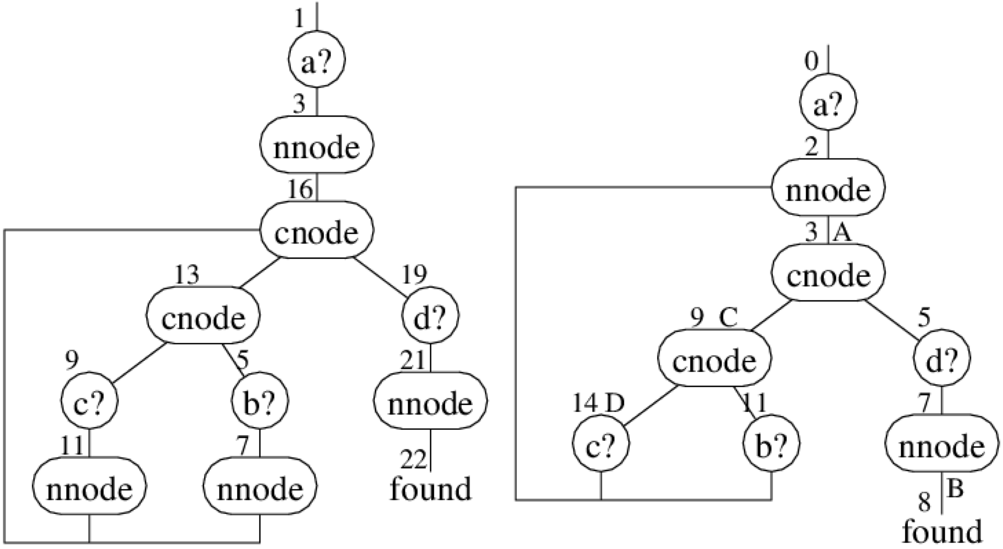


a(b|c)*dの遷移図

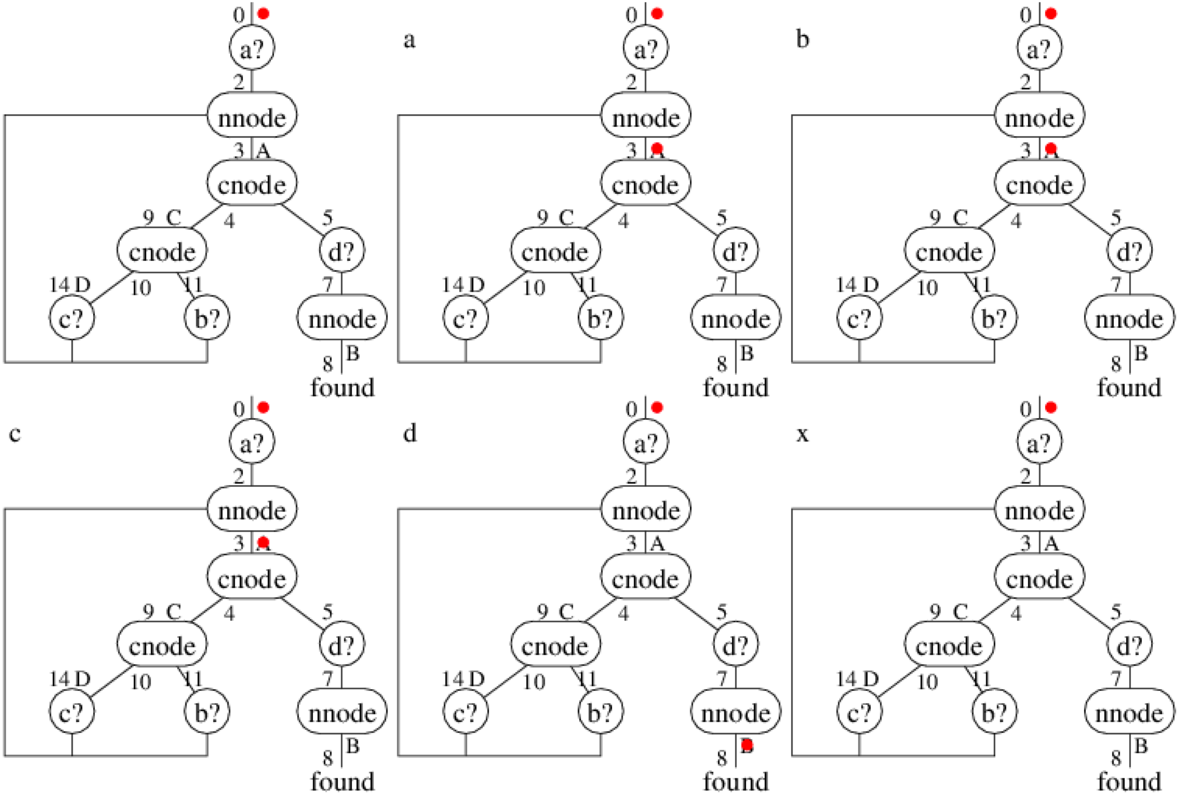


Thompsonの流れ図

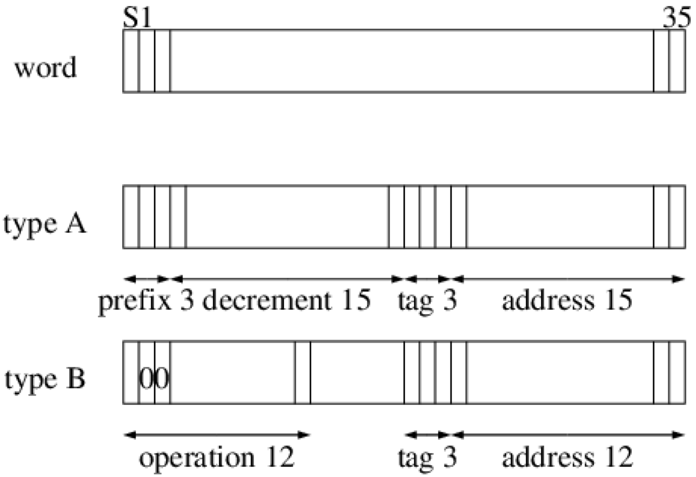
R.McNaughton, H.Yamada,
Regular Expressions and State Graphs for Automata,
IRE Trans. on EC, EC-9,No.1,(Mar.1960), pp.39-47.

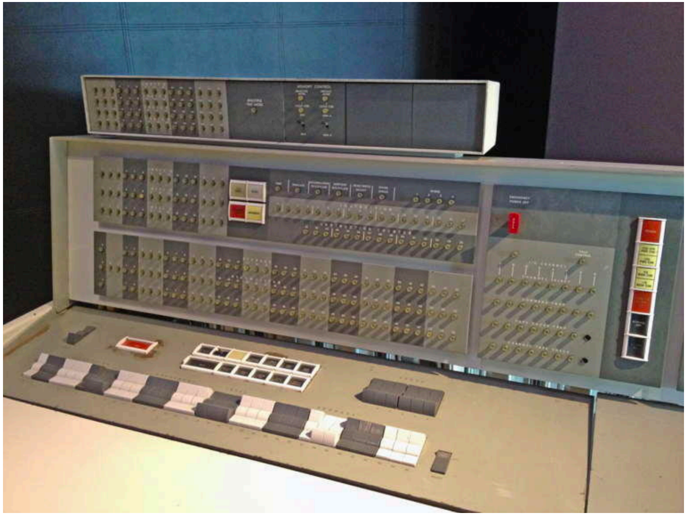


Thompsonのプログラム

0
a?
2
nnode
3
A
cnode
9
C
4
5
cnode
d?
14
D
10
11
7
c?
b?
nnode
8
B
found
a
b
c
d
x

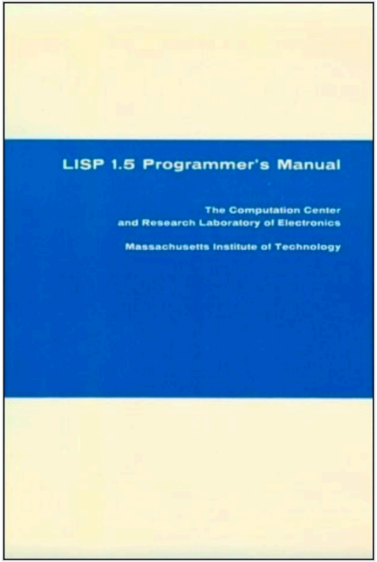
IBM 7094


S1
35
word
type A
prefix 3
decrement 15
tag 3
address 15
type B
00
operation 12
tag 3
address 12



carとcdr

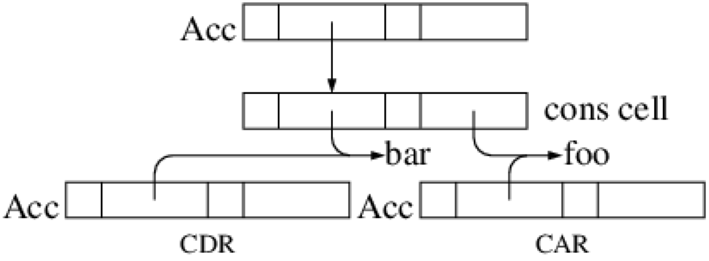
```
CAR     SXA     CARX,4  帰り番地をCARXのAdr部へ
        PDX     0,4     AccのDcr部をIR4へ
        CLA     0,4     記憶場所0,4の語をAccへ
        PAX     0,4     AccのAdr部をIR4へ
        PXD     0,4     IR4をAccのDcr部へ
CARX    AXT     **,4    帰り番地をIR4へ
        TRA     1,4     戻る
CDR     SXA     CDRX,4  帰り番地をCDRXのAdr部へ
        PDX     0,4     AccのDcr部をIR4へ
        CLA     0,4     記憶場所0,4の語をAccへ
        PDX     0,4     AccのDcr部をIR4へ
        PXD     0,4     IR4をAccのDcr部へ
CDRX    AXT     **,4    帰り番地をIR4へ
        TRA     1,4     戻る
```



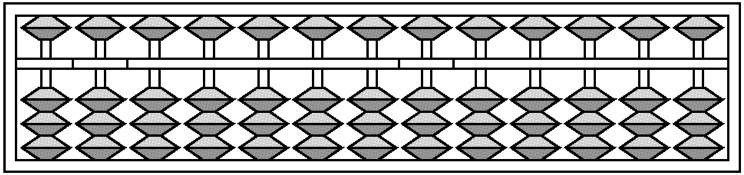
LISP 1.5 Programmer's Manual

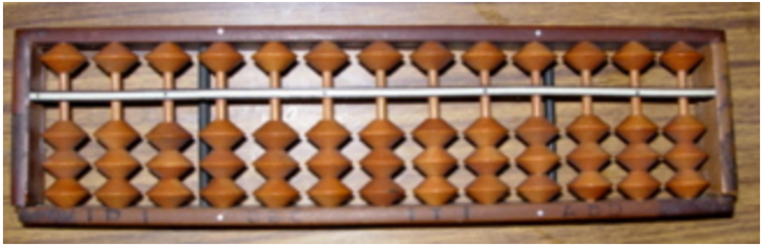
The Computation Center
and Research Laboratory of Electronics

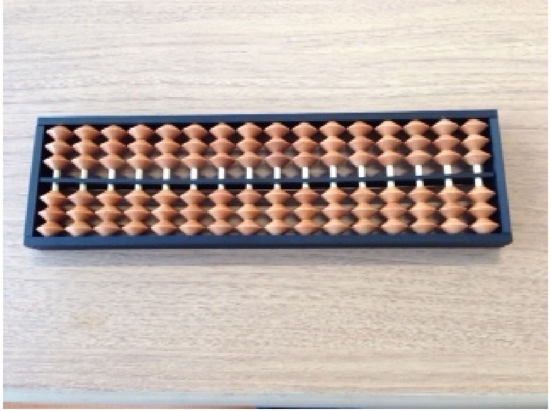
Massachusetts Institute of Technology



八進そろばん

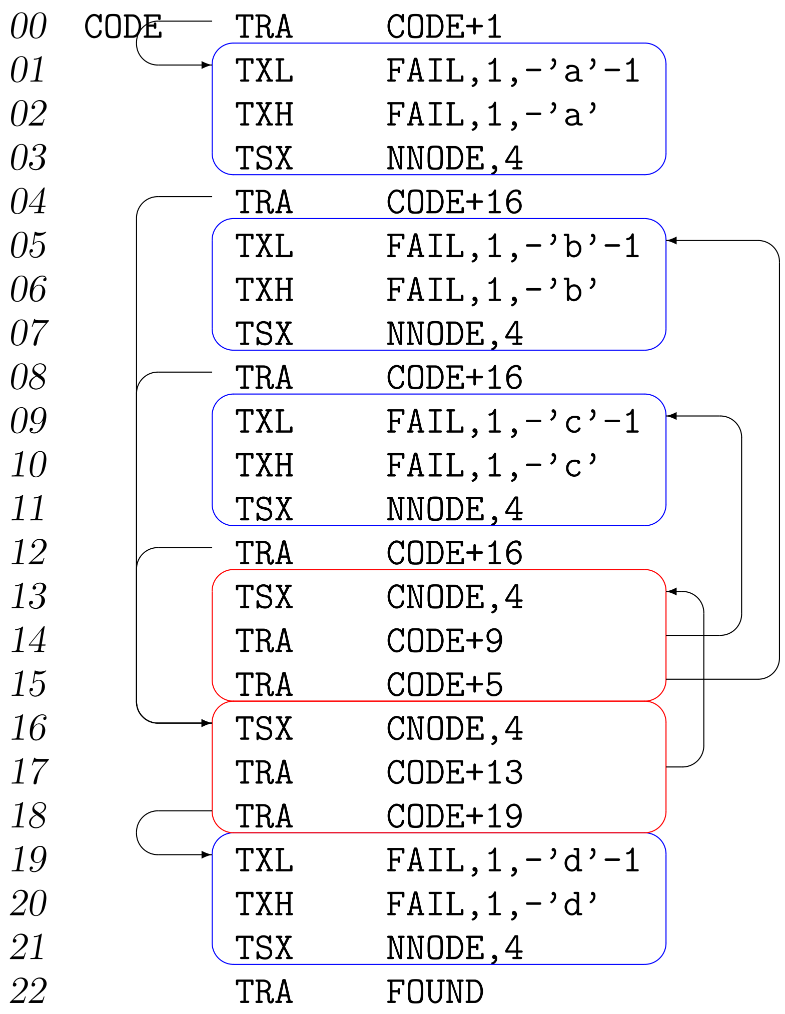




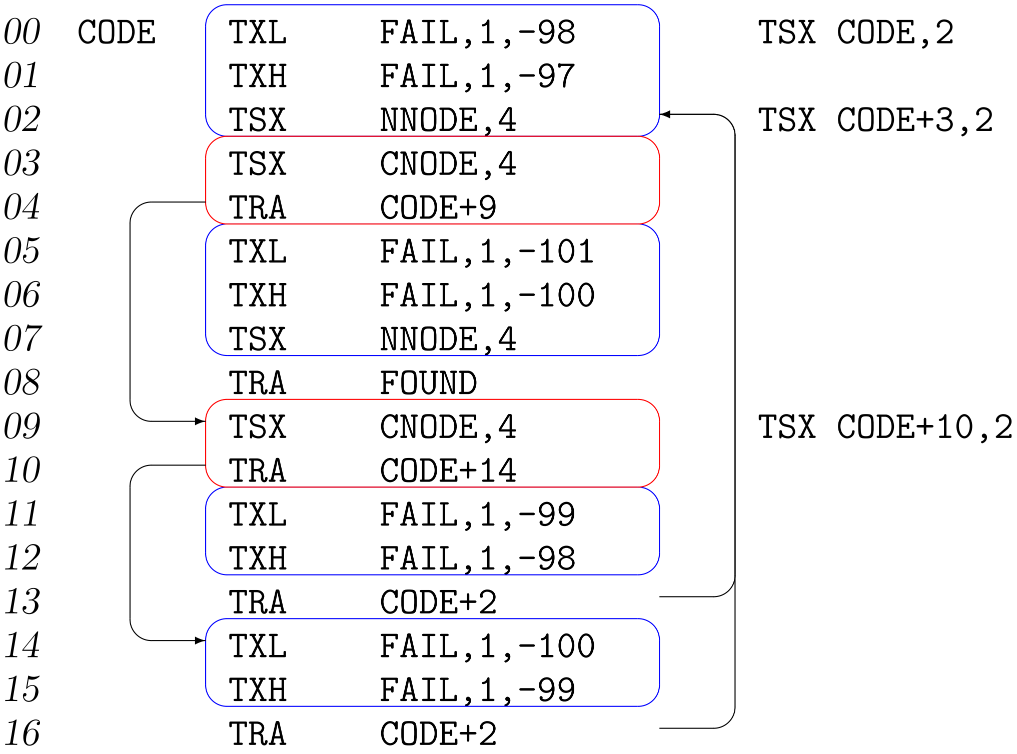


第58回報告集　参照

CODE



```
00   CODE   TRA   CODE+1
01          TXL   FAIL,1,-'a'-1
02          TXH   FAIL,1,-'a'
03          TSX   NNODE,4
04          TRA   CODE+16
05          TXL   FAIL,1,-'b'-1
06          TXH   FAIL,1,-'b'
07          TSX   NNODE,4
08          TRA   CODE+16
09          TXL   FAIL,1,-'c'-1
10          TXH   FAIL,1,-'c'
11          TSX   NNODE,4
12          TRA   CODE+16
13          TSX   CNODE,4
14          TRA   CODE+9
15          TRA   CODE+5
16          TSX   CNODE,4
17          TRA   CODE+13
18          TRA   CODE+19
19          TXL   FAIL,1,-'d'-1
20          TXH   FAIL,1,-'d'
21          TSX   NNODE,4
22          TRA   FOUND
```



```
00  CODE  TXL  FAIL,1,-98      TSX  CODE,2
01        TXH  FAIL,1,-97
02        TSX  NNODE,4         TSX  CODE+3,2
03        TSX  CNODE,4
04        TRA  CODE+9
05        TXL  FAIL,1,-101
06        TXH  FAIL,1,-100
07        TSX  NNODE,4
08        TRA  FOUND
09        TSX  CNODE,4         TSX  CODE+10,2
10        TRA  CODE+14
11        TXL  FAIL,1,-99
12        TXH  FAIL,1,-98
13        TRA  CODE+2
14        TXL  FAIL,1,-100
15        TXH  FAIL,1,-99
16        TRA  CODE+2
```

CNODE， NNODE， XCHG

```
00  CNODE   AXC   **,7        命令語のAdr部の負数をIR7へ;IR7は負
01          CAL   CLIST,7     CLISTの最後の語を
02          SLW   CLIST+1,7   その次へ移す
03          PCA   ,4          IR4の負数をAccへ;Accは正
04          ACL   TSXCMD      ‘TSX 1,2’を足す
05          SLW   CLIST,7     CLISTの最後の前へ
06          TXI   *+1,7,-1    IR7を1減らす
07          SCA   CNODE,7     その値の負をCNODEのAdr部へ;CNODEのAdrは正
08          TRA   2,4         TSX CNODE,4命令の次の次へ戻る
09  TSXCMD  TSX   1,2         定数
```

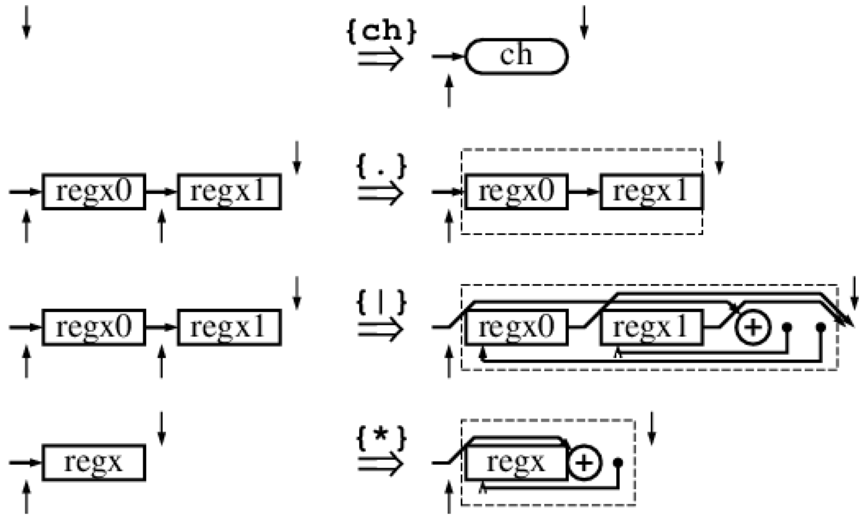
```
00  NNODE   AXC   **,7        命令語のAdr部の負数をIR7へ;IR7は負
01          PCA   ,4          IR4の負数をAccへ;Accは正
02          ACL   TSXCMD      ‘TSX 1,2’を足す
03          SLW   NLIST,7     NLISTの最後へ
04          TXI   *+1,7,-1    IR7を1減らす
05          SCA   NNODE,7     その値の負をNNODEのAdr部へ;NNODEのAdrは正
06          TRA   1,2         CODEのサブルーチンからCLISTへ戻る
```

```
00  XCHG    LAC   NNODE,7     NNODEのAdr部の負数をIR7へ;IR7は負
01          AXC   0,6         命令語のAdr部をIR6へ;IR6は0
02  X1      TXH   X2,7,-1     IR7>-1なら,IR7=0ならX2へ
03          TXI   *+1,7,1     IR7を1増やす
04          CAL   NLIST,7     NLIST-IR7の内容をAccへ
05          SLW   CLIST,6     AccをCLIST-IR6へ
06          TXI   X1,6,-1     IR6を1減らしてX1へ
07  X2      CLA   TRACMD      ‘TRA XCHG’をAccへ
08          SLW   CLIST,6     AccをCLIST-IR6へ
09          SCA   CNODE,6     IR6の負数をCNODEのAdr部へ;CNODEのAdrは正
10          SCA   NNODE,0     0をNNODEのAdr部へ
11          TSX   GETCH,4     次の文字を読みAccのAdr部へ
12          PAC   ,1          Adr部の負数をIR1へ
13          TSX   CODE,2      帰り番地をIR2に入れCODE+0へ
14          TRA   CLIST       CLISTへ
15  TRACMD  TRA   XCHG        定数

    FAIL    TRA   1,2         IR2を使って戻る

    INIT    SCA   NNODE,0     0をNNODEのAdr部へ
            TRA   XCHG        XCHGへ
```

コンパイラ `abc|*.d.`

```
begin                                                   codeの宣言は省略
  integer $char, lc, pc$;
  integer array $stack[0 : 10], code[0 : 300]$;
  switch $switch := alpha, juxta, closure, or, eof$;
  $lc := pc := 0$;                                      スタックカウンタ，プログラムカウンタ
$advance$ :                                             1文字読むとここに戻る
  $char := get\ character$ :                            1文字読む
  go to $switch[index(char)]$;                          文字の種類に応じて分岐する
$alpha$ :                                               文字の場合
  $code[pc] := instruction(`tra', value(`code') + pc + 1, 0, 0)$;
  $code[pc + 1] := instruction(`txl', value(`fail'), 1, -char - 1)$;
  $code[pc + 2] := instruction(`txh', value(`fail'), 1, -char)$;
  $code[pc + 3] := instruction(`tsx', value(`nnode'), 4, 0)$;
  $stack[lc] := pc$;
  $pc = pc + 4$;
  $lc = lc + 1$;
  go to $advance$;
$juxta$ :                                               連接の場合
  $lc := lc - 1$;
  go to $advance$;
```

```
$closure$ :                                    反復の場合
    $code[pc] := instruction(`tsx', value(`cnode'), 4, 0);$
    $code[pc + 1] := code[stack[lc - 1]];$
    $code[stack[lc - 1]] := instruction(`tra', value(`code') + pc, 0, 0);$
    $pc := pc + 2;$
    **go to** $advance;$
$or$ :                                         選択の場合
    $code[pc] := instruction(`tra', value(`code') + pc + 4, 0, 0);$
    $code[pc + 1] := instruction(`tsx', value(`cnode'), 4, 0);$
    $code[pc + 2] := code[stack[lc - 1]];$
    $code[pc + 3] := code[stack[lc - 2]];$
    $code[stack[lc - 2]] := instruction(`tra', value(`code') + pc + 1, 0, 0);$
    $code[stack[lc - 1]] := instruction(`tra', value(`code') + pc + 4, 0, 0);$
    $pc := pc + 4;$
    $lc := lc - 1;$
    **go to** $advance;$
$eof$ :                                        eofの場合
    $code[pc] := instruction(`tra', value(`found'), 0, 0);$
    $pc := pc + 1;$
**end**
```



文字の場合　文字処理をCODEに積む．CODEは4増え，スタックも1増える．

連接の場合　2項演算だからスタックは1減る．

選択の場合　選択処理をCODEに積む．CODEは4増え，2項演算だからスタックは1減る．

反復の場合　反復処理をCODEに積む．CODEは2増え，単項演算だからスタックは変らない．

Schemeで再試

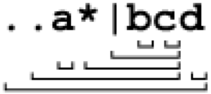
探索エンジン

```
(define (nextchar str clist nlist)      ;文字のループ
 (define (nextclist clist)              ;clistのループ
  (define (nextcode clist)              ;codeのループ
   (let ((n (car clist)))
    (if (>= n (length code)) 'found     ;codeの範囲を超えた->found
     (let ((s (list-ref code n)))
      (cond ((number? (car s))  (display (list 'code n s))
        (nextcode (append s (cdr clist))))
       ((char=? (car s) (string-ref str 0))
        (set! nlist (cons (cadr s) nlist))
        (nextclist (cdr clist)))
       (else (nextclist (cdr clist))))))))
  (display (list 'clist clist))
  (if (null? clist) (nextchar (string-tail str 1) nlist '(0))
   (nextcode clist)))
 (newline)(display str)
 (if (= (string-length str) 0) 'fail (nextclist clist))
;文字列がなくなるとfail
```

```
(define code '((#\a 1) (2 5) (3 4) (#\b 1) (#\c 1) (#\d 6)))
(define str "abcdx")
(nextchar str '(0) '(0))
```

```
abcdx(clist (0))(clist ())
bcdx(clist (1 0))(code 1 (2 5))(code 2 (3 4))(clist (4 5 0))(clist (5 0))
   (clist (0))(clist ())
cdx(clist (1 0))(code 1 (2 5))(code 2 (3 4))(clist (4 5 0))(clist (5 0))
   (clist (0))(clist ())
dx(clist (1 0))(code 1 (2 5))(code 2 (3 4))(clist (4 5 0))(clist (5 0))
   (clist (0))(clist ())
x(clist (6 0)) =>found
```

コンパイラ



```
(define (comp q r)
 (define (chr ch r) (list (list ch (if (null? r) (+ q 1) r))))
 (define (ast r) (let ((c (comp (+ q 1) q)));
    続く1個の正規表現をコンパイルしcに置く
  (cons (list (+ q 1) (+ q 1 (length c))) c)))
 (define (mid r)       ;続く2個の正規表現をコンパイルしcとdに置く
  (let* ((c (comp (+ q 1) r)) (d (comp (+ q 1 (length c)) r)))
   (cons (list (+ q 1) (+ q 1 (length c))) (append c d))))
 (define (dot r)       ;続く2個の正規表現をコンパイルしcとdに置く
  (let* ((c (comp q '())) (d (comp (+ q (length c)) r)))
   (append c d)))
  (let ((ch (string-ref str 0))) (set! str (string-tail str 1))
  (cond ((and (char<=? #\a ch) (char<=? ch #\z)) (chr ch r));英字
   ((and (char<=? #\0 ch) (char<=? ch #\9)) (chr ch r))     ;数字
   ((char=? ch #\*) (ast r))            ;*
   ((char=? ch #\|) (mid r))            ;|
   ((char=? ch #\.) (dot r)))))         ;.
```

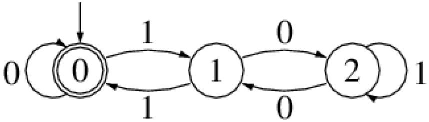
```
(define str "..a*|bcd") (comp 0 '()) =>
((#\a 1) (2 5) (3 4) (#\b 1) (#\c 1) (#\d 6))
```

下の3で整除出来る二進数のコンパイル結果

```
(define str "..x*|0..1*...0*1*.0001y") (comp 0 '()) =>
((#\x 1) (2 14) (3 4) (#\0 1) (#\1 5) (6 13) (#\0 7) (8 9)
 (#\1 7) (10 12) (#\0 11) (#\0 9) (#\0 5) (#\1 1) (#\y 15))
```

3で整除出来る二進数

正規表現 (0|(1(01*(00)*0)*1)*)* の二進数は3で整除できる.



```
(do ((i 0 (+ i 1))) ((= i 16) 'ok)
 (let ((str (string-append "x" (number->string i 2) "yy")))
  (display (list i (nextchar str '(0) '(0))))))
```

```
(0 found)(1 fail)(2 fail)(3 found)(4 fail)(5 fail)(6 found)(7 fail)
(8 fail)(9 found)(10 fail)(11 fail)(12 found)(13 fail)(14 fail)(15 found)
```

複数回探すには

```
(define (nextchar str clist nlist)
 (define (nextclist clist)
  (define (nextcode clist)
   (let* ((n (car clist)) (s (list-ref code n)))
    (cond ((number? (car s)) (display (list ’code n s))
       (nextcode (append s (cdr clist))))
      ((char=? (car s) (string-ref str 0))
        (let ((m (cadr s)))
         (if (>= m (length code))       ;codeの範囲を超えた
          (display (list ’found (string-tail str 1))) ;found
          (set! nlist (cons m nlist)))  ;nlistに入れる
         (nextclist (cdr clist))))
      (else (nextclist (cdr clist))))))
  (display (list ’clist clist))
  (if (null? clist)                ;clistが無くなったら
   (nextchar (string-tail str 1) nlist ’(0))
           ;nlistをclistにし次の文字から探す
   (nextcode clist)))
 (newline)(display str)
 (if (= (string-length str) 0) ’fail (nextclist clist)))
```

```
(define code ’((#\a 1) (#\a 2) (#\a 3) (#\a 4)))
      ;連続する4個の"a"を探すcode
(define str "aaaaaaa")
(nextchar str ’(0) ’(0))
```

```
aaaaaaa(clist (0))(clist ())
aaaaaa(clist (1 0))(clist (0))(clist ())
aaaaa(clist (1 2 0))(clist (2 0))(clist (0))(clist ())
aaaa(clist (1 3 2 0))(clist (3 2 0))(found aaa)(clist (2 0))
    (clist (0))(clist ())
aaa(clist (1 3 2 0))(clist (3 2 0))(found aa)(clist (2 0))
    (clist (0))(clist ())
aa(clist (1 3 2 0))(clist (3 2 0))(found a)(clist (2 0))
    (clist (0))(clist ())
a(clist (1 3 2 0))(clist (3 2 0))(found )(clist (2 0))
    (clist (0))(clist ())
```

バックトラックでは

```
(define (search str)
 (call-with-current-continuation
  (lambda (return)
   (define (try n str)   ;n番目のcodeをstrで調べる
    (newline) (display (list ’try n str))
    (cond ((>= n (length code)) (return ’found))
     ((string=? str "") (return ’fail))
     (else
      (let ((c (list-ref code n))) (display c)
       (cond ((number? (car c)) (try (car c) str) (try (cadr c) str))
        ((char=? (car c) (string-ref str 0))
         (try (cadr c) (string-tail str 1))))))))
   (let ((x (try 0 str)))    ;先頭から調べ
    (if (or (eq? x ’found) (eq? x ’fail)) x;foundかfailが返れば終わる
     (search (string-tail str 1)))))))     ;1文字先から調べる
```

実行結果

```
(define code ’((#\a 1) (2 5) (3 4) (#\b 1) (#\c 1) (#\d 6)))
          ;thompson example
(define str "aabbccdd")
(search str)
```

```
(try 0 aabbccdd)(a 1) ;"a"だから1へ
(try 1 abbccdd)(2 5) ;分岐を先へ進む
(try 2 abbccdd)(3 4) ;文字列の先頭が
(try 3 abbccdd)(b 1) ;"b"でも
(try 4 abbccdd)(c 1) ;"c"でも
(try 5 abbccdd)(d 6) ;"d"でもないから0へ
(try 0 abbccdd)(a 1) ;"a"だから1へ
(try 1 bbccdd)(2 5)
(try 2 bbccdd)(3 4)
(try 3 bbccdd)(b 1) ;"b"だから1へ
(try 1 bccdd)(2 5)
(try 2 bccdd)(3 4)
(try 3 bccdd)(b 1) ;"b"だから1へ
(try 1 ccdd)(2 5)                    ↗
```

```
(try 2 ccdd)(3 4)
(try 3 ccdd)(b 1)
(try 4 ccdd)(c 1) ;"c"だから1へ
(try 1 cdd)(2 5)
(try 2 cdd)(3 4)
(try 3 cdd)(b 1)
(try 4 cdd)(c 1) ;"c"だから1へ
(try 1 dd)(2 5)
(try 2 dd)(3 4)
(try 3 dd)(b 1)
(try 4 dd)(c 1)
(try 5 dd)(d 6) ;"d"だから6へ
(try 6 d)
=>found
```