Ken Thompsonの正規表現探索アルゴリズムを解剖する

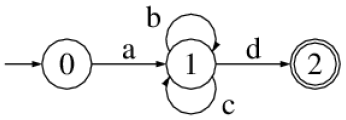
和田 英一*

概要　計算機科学の標準問題のひとつ, 正規表現探索アルゴリズムに, 1968年のComm. ACMに掲載されたKen Thompsonのプログラムがある. これはバックトラックを使わないから高速ということで知られている[0]. このプログラムは正規表現をIBM 704系の機械語に変換するコンパイラと, アセンブリ言語で記述された実行時に使うサブルーチン類で説明されている. つまり探索を機械語のプログラムで直接実行し, 速度を稼ごうという発想である.
Algol 60で記述されているコンパイラ部分はよいとして, 実行時処理では, サブルーチンと, コンパイラの生成した機械語プログラムと, これらのプログラムが実行中に生成する機械語プログラムが入り乱れて走り, 普通に読んだのでは難解至極である.
Internetで正規表現探索アルゴリズムを探しても, 機械語のThompsonアルゴリズムの説明は見付からないので, この不思議で興味深いプログラムを真面目に解読しようと思うようになった. サブルーチンはいずれも短く, 命令の種類も少ないので, 解読用にIBM 704系の計算機の, このプログラムで使う機能に限定したシミュレータを書き, いくつもの例題を実行し, そのトレース結果を丹念に読みながら調べると, その考え方が次第に分ってきた. 今回はThompsonプログラムの仕組みと, プログラムで作ったプログラムが走るvon Neumann型アーキテクチャを活用した時代のプログラムの面白さを説明する.

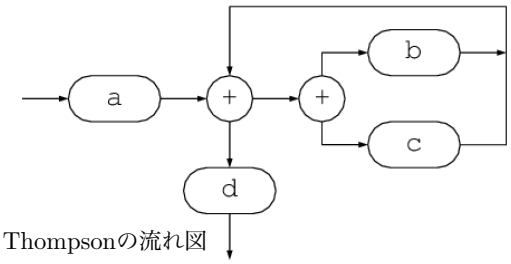
正規表現探索プログラム

正規表現探索プログラムはガーベージコレクションとともに, 計算機科学の多少難しい標準問題かと思われるが, 私には読んでみたいという気分になるプログラムが見付からない. Brian KernighanとRob Pikeの「プログラミング作法[1]」の第9章には, Pikeの書いた正規表現探索プログラムが出ているが, 食指が動かない. (このプログラムは「ビューティフルコード[2]」の第1章にも載っている.)
その私がどうしても読みたいと兼ねてから思っていたのが, Bell研にいたKen Thompsonの正規表現探索プログラムであった. しかしこのプログラムはIBM 7094のアセンブリ言語で書いてあり, しかも実行中に同計算機の機械語のプログラムを生成して走るので, 徒や疎かに読み始めてもたちどころに頓挫する.
しかし, 最近どうしても理解したくなり, IBM 7094の簡単なシミュレータも用意して読み始めたところ, プロシンで話すのも申し訳けないくらい, 簡単な構成であることが判明した. しかし一方で巧妙な仕組みなので, やはり話したくなっている.
これはComm. ACM誌の1968年6月号のProgramming Techniques欄に掲載された4ページの記事[3]である. 最近の正規表現には多くの機能があるが, Thompsonの扱うのは, ふたつの正規表現を並べて書く ‘連接’; ‘|’で表わす ‘選択’; ‘*’で表わす ‘反復’だけである. (Kleeneのsum, product, iterateに対応する[4].)
ところで最近Kernighanの「UNIX A History and a Memoir[5]」を読んでいたら, Thompsonの正規表現のアルゴリズムは特許になっていると書いてあった(p.75).
Thompsonの説明は, 正規表現 ‘`a(b|c)*d`’を例にしているので, ここでもそれを使う.
この正規表現は次ページ上左のような遷移図に対応し, 最初は状態0にいる. 読んだ文字が"a"なら状態1に移る. その後, "b"か"c"が続く限り状態1にいるが, 状態1で文字"d"が来ると状態2へ移り, 二重丸で成功する. これをThompson風の流れ図にしたのが右だ. ‘+’のある円は分岐, 文字のある小判型は文字を調べる場所で, 状態はどこかの線上にある. Thompsonはこれをプログラムにしたので, サブルーチンを呼ぶ場所などあり, 上の流れ図はすぐ下の図の左のように作った. それから無駄なジャンプを除いて, 今回のテストに使ったのがその右である. (数字は後掲のプログラムの行番号)

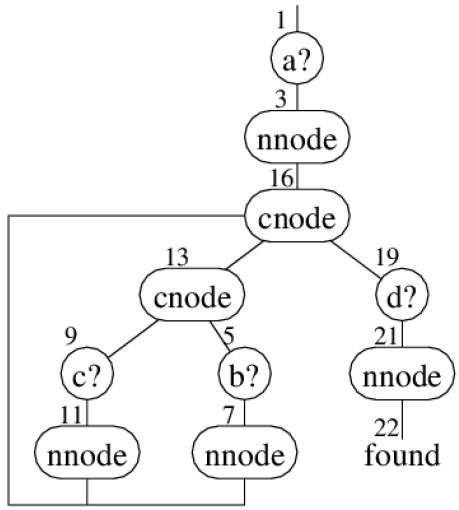
* IIJ技術研究所 ew@iij.ad.jp

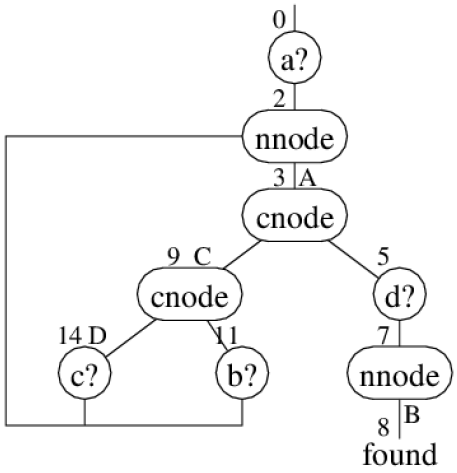


a(b|c)*dの遷移図

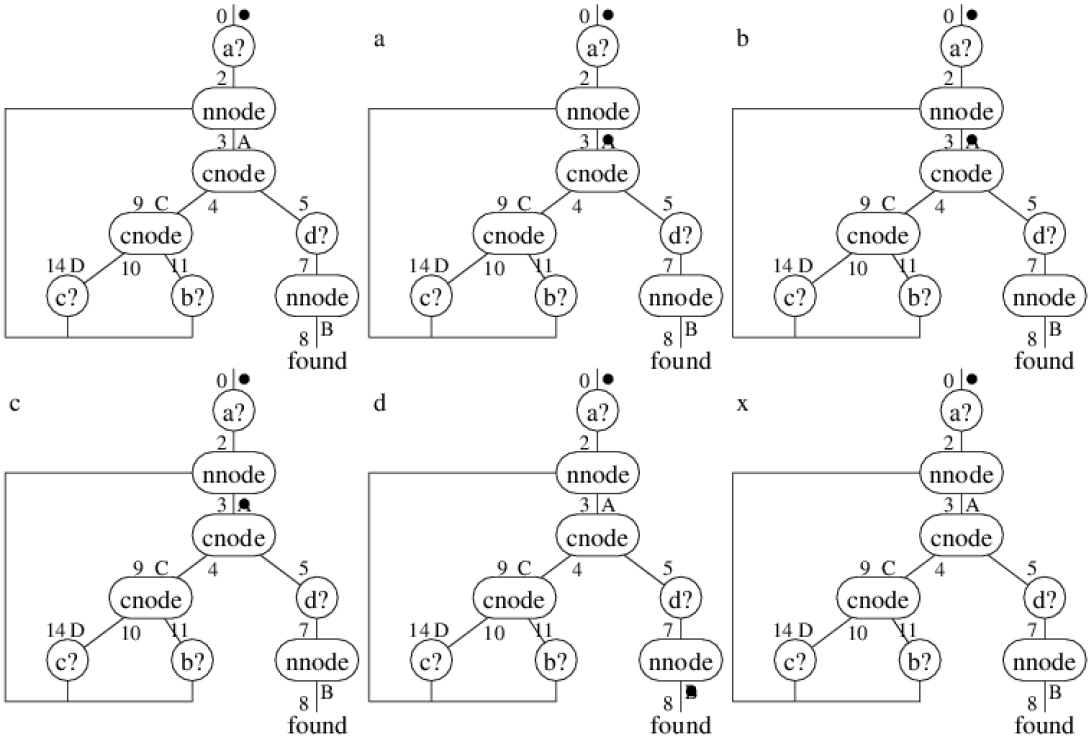


Thompsonの流れ図





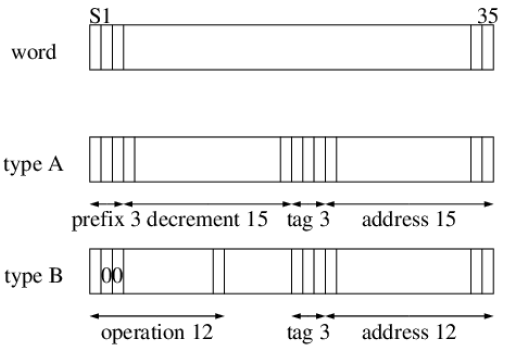
Thompsonのプログラム



探される文字列が"abcd"であったとして, このプログラムの動きが前のページの下の図で, 現在の状態を線の近くの黒丸で示す. まず上左でり黒丸は"a"を調べる円の上にいる. 上左の図と上中の図の間の"a"は, 読み込んだ文字が"a"であることを示す. 文字が一致したので黒丸はnnodeを通過した場所へ移る(上中). cnodeは黒丸を分岐するので, 黒丸は"d", "b", "c"を調べる円の上にいるのと同じである.
上中と上右の間で"b"を読み込み, 上右へ来るが, "b"の経路が一致したので, 状態は'b?'の円の下からnnodeを経て先程の場所へ戻る(上右). "c" でも同じ(下左).
下左と下中の間で"d"を読むと, "d"が一致してその下のnnodeの下へ来る(下中). 最後に何か次の1文字を読み(図には"x"としてある), それでfoundになる(下右).
図全体を通じて, "a"の上にある黒丸は, 読み込む文字列が"ababcbcd"のようであった場合, 途中の"a"から探索を始める準備である. 分岐の下に同じ文字を調べる場所が複数あって, そのいずれでも成功すれば, 黒丸は成功したすべての下に出来る. (この図にはそういうのはない.)

IBM 7094

IBM 7094は, 1964年にSystem 360が発表になるまでの, IBM社の科学用計算機 IBM 704から始まる二進法36ビットの計算機の系列の最後のモデルであった.
私が1958年の秋にMITの計算機センターを訪れると, そこにあったのはIBM 704で, 同じ計算機は間もなく東京の気象庁に納入されるとのこと. 当時の記憶容量は, 1語36ビットの4K語で, 記憶装置の最後の200_8語くらいにFortran Monitorを常駐させてバッチ処理を行っていた.
MITのIBM 704は, この上でLispやCompatible Time Sharing System(CTSS)が開発された, 計算機史上記憶されるべき計算機である.
IBM 704系のアーキテクチャを簡単に記述すると次のようだ.

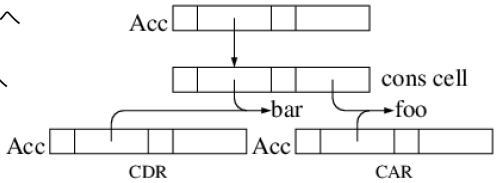


左の図の上は数値語で, 左端が符号ビット. 正は0, 負は1. 残り35ビットは整数の絶対値である. 演算用にそれぞれ36ビットのアキュムレータ(Acc)と乗除算に使うMQレジスタがある.
命令語にはtype Aとtype Bがあり, インデックスレジスタ(以下IRと記す) 関係の命令はtype Aで, 左から3ビット, 15ビット, 3ビット, 15ビットのプレフィックス, デクレメント(Dcr), タグ, アドレス(Adr)部である. type B命令は左から12ビット, 6ビット, 3ビット, 15ビットのそれぞれオペレーション, 不使用, タグ, アドレス(Adr)部である.

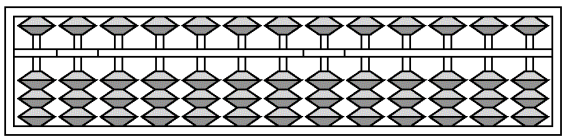
IBM 704の場合, タグの3ビットの100, 010, 001でIRの4番, 2番, 1番を指定した. IBM 7094ではIRは1番から7番に増えた.
type B命令にタグtの指定があると, アドレスの値からIRtを '引いた'値が実効アドレス値になる. この他に実行中の命令の番地を保持する命令カウンタIcがある. IBM 704のアーキテクチャは基本の形は変えることなく, 機能が少しずつ拡張され, 最後のIBM 7094にまで受け継がれた.
その機械語の雰囲気が分るのが, McCarthy他の著した「LISP 1.5 Programmer's Manual[6]」のAppendix Aにあるcarとcdrの説明だ.
(foo bar)のcarを取りたいとすると, そのconsセルの番地(の負数)をアキュムレータのDcr部に置き, IR4に帰り番地を置いてCARのサブルーチンへ来る. 帰り番地を一旦CARXに退避し, Dcrの内容をIR4へ入れ, consセ

ルをAccに取り出し, Adr部の情報(fooの番地の負数)をIR4経由でAccのDcr部へ置く. そのプログラム

```
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
```



のような記述があり, 当時はこれらを夢中で解読したものである.
その2,3年後, 1960年頃に, 私のいた小野田セメントの計算機室にも同じアーキテクチャのIBM 7074が設置され, 私もこのシリーズの計算機のアセンブリ言語でプログラムを書いたりしていた.



36ビットの機械語の情報はすべて八進法で表示され, 八進法の加減算が必要になったので, 八進法算盤を作ったのもその頃であった(上の図と写真).

Ken Thompsonの正規表現探索アルゴリズム

Thompsonの扱いでは, まず通常は何の記号も存在しない連接の場所に二項演算子'.'を入れて, 'a(b|c)*d'を'a.(b|c)*.d'にする. 次にかっこをはずして逆ポーランド記法 'abc|*.d.'にする.
この式を後述するコンパイラが読み込み次の機械語の列に変換する. 文字(a,b,c,d)と演算子(*, |)の対応する部分は, 正規表現に現れる順に出てくる. それを後で繋げ変えるから, ジャンプ命令TRAが沢山使われている.
文字xに対応する命令列は

```
TXL     FAIL,1,-'x'-1⌉
TXH     FAIL,1,-'x'  |
TSX     NNODE,4      ⌋,
```

この命令列に来たとき, 調べる文字のASCIIコードの値を負にしたもの("a"なら−97)がIR1に入っている. 最初のTXL命令はIR1<=−98ならFAILへ行け; 次のTXH命令はIR1>−97ならFAILへ行けで,丁度−97ならFAILへは行かず, 次へ進んでNNODEへサブルーチンジャンプする.
演算子に対応するのは, その分岐後の飛び先をs, tとして

```
TSX     CNODE,4⌉
TRA     CODE+s |
TRA     CODE+t ⌋
```

である. これらとTRA命令でCODEが作られる. 次の左側は前の流れ図の左に対応するもので, 右側は左の命令列から余計なTRA命令を削除し並べ変えたもので, 定数も計算してある.

NNODEは文字が一致した場合の処理ルーチン; CNODEは分岐処理ルーチンで,これらについては後で述べる.

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

上のプログラムは実はサブルーチンの本体である. サブルーチンの入口は前の図の黒丸に対応し, CODEの先頭かNNODEの下か, CNODEのすぐ下で, 出口はFAILかNNODEへ行ってから; 呼び主はプログラムが実行中に生成するCLISTの中からで, そこには 'TSX CODE+n,2'のサブルーチンジャンプ命令列が並んでいる.
NNODEは文字の照合が成功した後なので, 次の文字用の黒丸への呼出し命令をNLISTに置いて準備をする. サブルーチン本体の中を走ってCNODEへ来ると, CNODEの次へのTSX命令をCLISTに追加して次の次の命令から再開する. CLISTの呼出し命令がなくなると, CLISTの最後にあるTRA XCHG命令でXCHGへ行き, そこでNILSTをCLISTへコピーし, 次の文字を読み, IR1へ置いて, 新しいCLISTの実行を開始する.
以上を理解しておくと, Thompsonの書いたプログラム, CNODE, NNODE, XCHGを読むのは簡単だ.

```
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
```

上右のCODEの*03*行目のCODE+3からCNODEへ来たとすると, IR4はCODE+3を負にしたものである. AXC **,7でCLISTの長さの負数をIR7に置き, 続く2命令でCLISTの最後の 'TRA XCHG'命令を1番地下へ移す. PCAでCODE+3をアキュムレータに置き, 'TSX 1,2'を足すからアキュムレータは 'TSX CODE+4,2'になり, これをさき程 'TRA XCHG'命令のあった場所にいれ, TXIでIR7を1減らす. CLISTの長さは1増える. この値を先頭のAXC命令のAdr部に戻し, 'TRA 2,4'でCODE+5へ帰る. つまり分岐の一方sをCLISTに保存し, もう一方tに戻ったわ

けだ.

```
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
```

CODEの*02*行目のCODE+2からNNODEへ来たとする. AXC命令でIR7をNLISTの長さの負数にする. ジャンプ元のCODE+2をアキュムレータに置き, 'TSX 1,2'を足して 'TSX CODE+3,2'を作り,次回のCLIST用の要素としてNLISTに入れ, NLISTの長さを1増やして, 'TRA 1,2'でサブルーチンから戻る.

```
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
```

XCHGはある入力文字との比較が終った時, つまりCLISTを使い切った時に来る. NLISTの長さをIR7に置き, IR6を零にする. このあとX1とX2の間のループで, NLISTをCLISTに逆順に移す. 逆順に意味はない. 次にCLISTの最後にTRA XCHGを置き, CNODEとNNODEの先頭のAXC命令のAdr 部の値を更新し, 1文字読み, 流れ図の先頭の黒丸に対応する呼出し 'TSX CODE,2'を実行してから, 'TRA CLIST'へ進む. Thompsonのプログラムは'TXL X2,7,0'と間違っていた.

この他に

```
    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
```

がある.

インデックスレジスタの使い方を整理すると,

IR1 文字の比較用
IR2 CODEの列中へのサブルーチンジャンプの帰り番地用
IR4 NNODE, CNODEへのサブルーチンジャンプの帰り番地
IR6 XCHGでCLISTの長さを記憶
IR7 XCHGでNLISTの長さを記憶とNNODE, CNODE で使う.

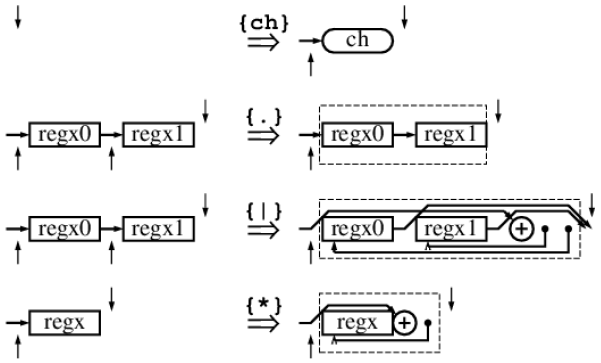
実行は最後にあったINITから始まる. SCAはレジスタの値の負数をアドレス部の示す番地のアドレス部に入れる. IR0は常に0だから, NNODEのアドレス部は0になる. NNODE, CNODEのアドレス部は, プログラムが実行時に生成するプログラムリストのNLIST, CLIST(最後のTRA XCHG命令を除いた)長さが正の値で格納されている. そしてXCHGへ飛ぶ. XCHGにはこの時と, CLIST実行の最後に来る.

コンパイラ

Thompsonのコンパイラは, 逆ポーランド記法の正規表現を読み, CODEの機械語命令列を作り出すものだ. Algol 60で記述してある[7]. 正規表現は入れ子になるから, 再起呼出しを使ってもよさそうだが, スタックで処理していて, KnuthのTAOCPのアルゴリズムを読む気分だ. つまり元はFORTRANだったのか.
処理は1文字読み, それが文字か(*alpha*), 連接(‘.’,*juxta*)か, 反復(‘*’,*closure*)か, 選択(‘|’, *or*)か, eof(*eof*)かでそれぞれの処理へ飛び,スタックを使ってCODEの要素を生成する. (プログラムの後の図を参照)

```
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
```

各処理によるCODE生成の様子, TRA命令の繋ぎ変えは次のようだ.



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.

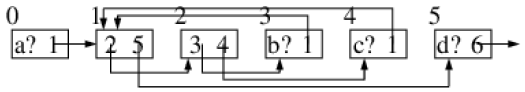
反復の場合 反復処理をCODEに積む. CODEは2増え, 単項演算だからスタックは変らない.

Schemeでもっと手軽に

IBM7094のアセンブリ言語のアルゴリズムは理解できたので, この方法をもっと簡単に実験しようと考え, Schemeでインタプリティブ方式の実装した.

探索エンジン

'abc|*.d.'のCODEに相当するものは, たとえば下の図のようにし, codeの範囲から出たらfoundとする.



下がプログラムだ. 文字を次々と処理し, 各文字についてclistを次々と処理し, clistの要素について文字の比較に来るまでcodeを次々とたどるから3重のループになっている. 書いてみると, 機械語命令を作ったりしないから, Thompsonのよりずっと簡単であった.

```
(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))
```

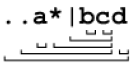
実行すると

```
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

次に正規表現からcodeを生成するコンパイラだが, これも以下のようだ. Thompsonは逆ポーランド記法をコンパイルしたが, 私はLispらしく, 逆でないポーランド記法, つまり '..a*|bcd'を読み込んで処理する形にした.

コンパイラの考え方はこうだ. 文字や記号には正規表現として支配の範囲があるとする. その支配の範囲を改めて正規表現という. (右図の下線部分)



文字はそれ自体が1個の正規表現である. 連接と選択の記号はそれとそれに続く2個の正規表現で正規表現とする. 反復の記号はそれとそれに続く1個の正規表現で正規表現とする. 上に例は右の図のような正規表現の入れ子になる. コンパイラは記号を見るとletを使ってそれに続く正規表現を先にコンパイルしてから自分のコンパイル結果を作る. (comp q r)のqはコンパイル結果を入れる位置, rは帰り先, () なら次へという指示.

```
(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)))))         ;.
```

上の例のコンパイル結果

```
(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

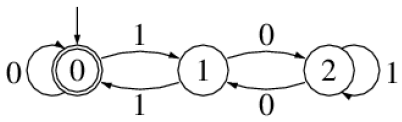
```
(define str "..x*|0..1*...0*1*.0001y") (comp 0 '()) =>
((#\x 1) (2 14) (3 4) (#\0 1) (#\1 5) (6 13) (#\0 7) (8 9)
 (#\1 7) (10 12) (#\0 11) (#\0 9) (#\0 5) (#\1 1) (#\y 15))
```

3で整除出来る二進数

あるホームページ[8]に, 正規表現

(0|(1(01*(00)*0)*1)*)*

の二進数は3で整除できると書いてあった. これは次のオートマトンを正規表現にしたものである.



これが3で整除できる理由だが, 状態0はスタートで値は0だから3で整除できる. その後に0が付くと値は2倍になる. 3の倍数は2倍しても3の倍数だから, 状態0のまま. 状態0で1が来れば, 3で割った剰余は1だから, 状態1へ移る. 状態1で1が来れば, 剰余1の2倍足す1で剰余0になり状態0へ移る. 後も同じだ.

正規表現探索では, 両端をはっきりさせるためにxとyで囲むことにする.

前節にあったように, コンパイルして得られたcodeは

```
(define code ' ((#\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)))
```

で, (range 0 16)の二進数の整除性を調べる次のプログラムを走らせると, その下のような結果が出る.

```
(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

正規表現が 'abc'で, 探される文字列が"abcabc"なら2回見付けたい. 正規表現が 'aaaa'で, 探される文字列が"aaaaaaa"なら4回見付たい. そういうことは出来るか. Thompsonは, FOUNDはマッチする度に呼ばれると書いているから, それを試みる.

Thompsonの方法は, 文字の比較で一致したら, その次の状態をNLISTに置き, 次の文字でその状態からFOUNDへ行くのであった. 今回の変更は, 次の状態を置く時に, それがFOUNDへ進むなら, NILSTに置かずにFOUNDを表示し, 次のCLIST処理を続ける.

下のプログラムはそのように出来ている. 後半は"a"の7個の列から"a"の4個の列を探した結果である.

この方式では最後に余計な文字を付ける必要はない.

```
(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))
```

以下が実行結果で, foundの後は, その時残っている文字列である. 実行の最後はfailになる.

```
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

Thompsonのアルゴリズムとよく比較されるのが, バックトラックするものである. 今回序でにそういうプログラムも書いてみた. 結果が分った途端に入れ子の内部から脱出したいので, call-with-cc を使う.

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

実行結果は以下のようだ.

```
(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へバックトラック
(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
```

IBM7090命令の説明

本稿に出てきたIBM704系の命令の簡単な説明は以下の通り．詳細な解説は文献[9] などにある．m[Adr−IRt] はAdr−IRt番地の記憶場所の内容の意．IRtはt番のインデックスレジスタ．

ACL	m[Adr−IRt]をAccに足す
AXC	命令語のAdr部の負数をIRtへ
CAL	m[Adr−IRt]をAccへ
CLA	m[Adr−IRt]をAccへ
LAC	命令語のAdr部の負数をIRtへ
PAC	AccのAdr部の負数をIRtへ
PAX	AccのAdr部をIRtへ
PCA	IRtの負数をAccAdr部へ
PXD	IRtをAccのCdr部へ
SCA	IRtをm[Adr]のAdr部へ
SLW	Accをm[Adr]へ
TRA	Adr−IRtへ進む
TSX	IcをIRtへ; Adr部へ進む
TXH	Dcr部>IRtならAdr部へ進む
TXI	IRt=IRt+DcrとしAdr部へ進む
TXL	Dcr部<=IRtならAdr部へ進む

参考文献

[0] Russ Cox, Regular Expression Matching Can Be Simpler And Fast, https;//swtch.com/~rsc/regexp/regexp1.html.

[1] B.W.Kernighan, R.Pike, 福崎俊博訳, プログラミング作法,アスキー, 2000.

[2] A.Oram他編, 久野靖他訳, ビューティフルコード, O'Reilly, 2008.

[3] Ken Thompson, Regular Exprssion Search Algorithm, Comm. ACM, Vol.11,No.6 (June 1968),pp.419-422.

[4] S.C.Kleene, Realization of events in nerve nets and finite automata, in C.E.Shannon and J.McCarthy ed., Automata Studies,pp.3-42,1956.

[5] Brian Kernighan, UNIX A History and a Memoir.

[6] John McCarthy他, LISP 1.5 Programmer's Manual, 1965, MIT Press, http://www.softwarepreservation.org/projects/LISP/book/LISP%201.5%20Programmers%20Manual.pdf.

[7] J.W.Backus他, Revised report on the algorithm language ALGOL 60, Comm.ACM,Vol.6,No.1,(Jan.1963),pp.1-17. https://web.eecs.umich.edu/~weimerw/2018-590/reading/NauerAlgol60.pdf

[8] https://en.wikipedia.org/wiki/Thompson's_construction

[9] IBM 704 Electronic Data-Processing Machine, Manual of Operation, http://www.bitsavers.org/pdf/ibm/704/24-6661-2_704_Manual_1955.pdf.