テキスト処理 第7回 (2007-06-19)

田中哲
産業技術総合研究所
情報技術研究部門
akr@isc.senshu-u.ac.jp
http://staff.aist.go.jp/tanaka-akira/textprocess-2007/

今日の内容

- レポートの解説
- 簡単な正規表現エンジン
- レポート

正規表現エンジン

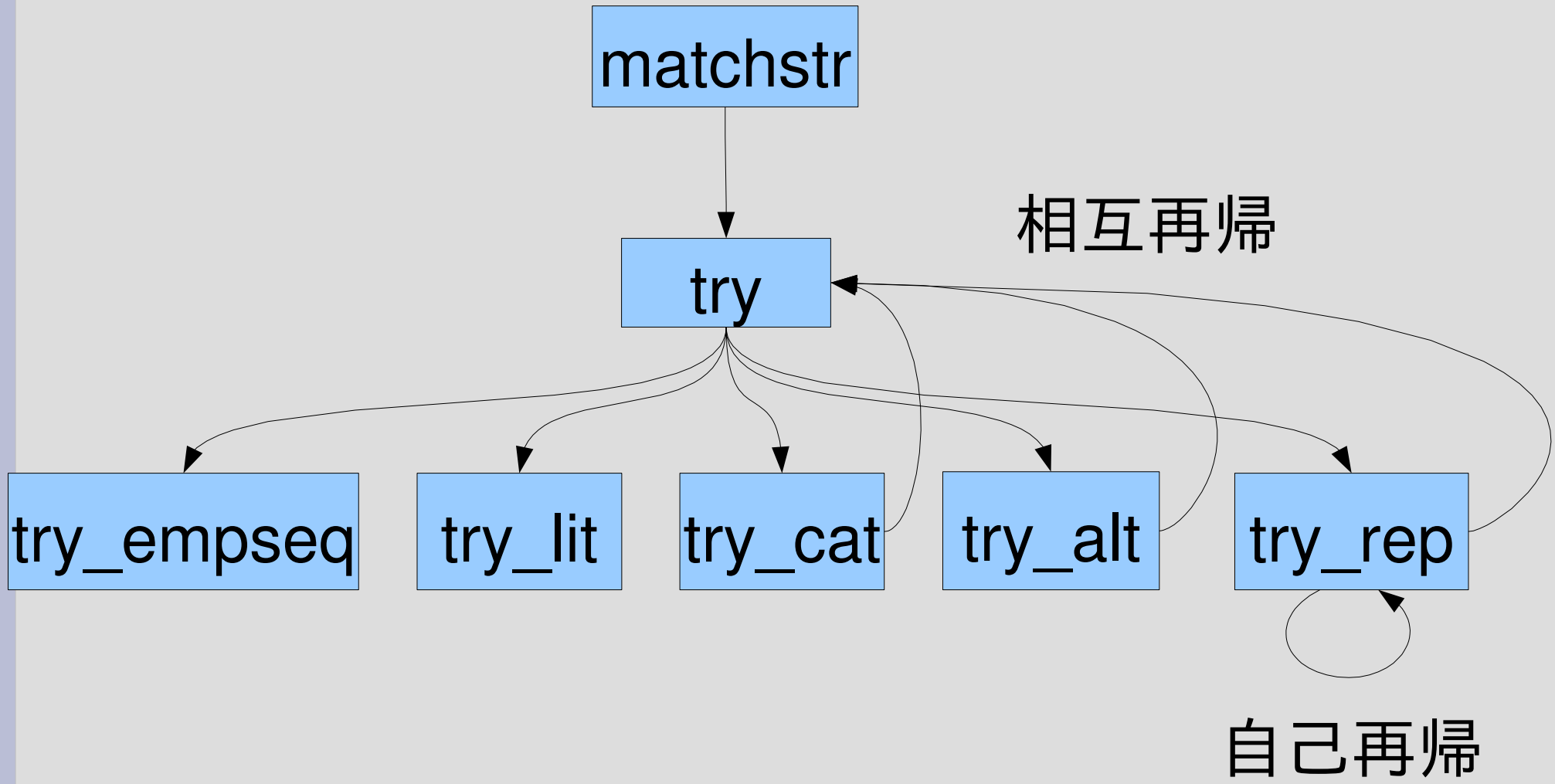
- 正規表現と文字列のマッチを行う機構

正規表現エンジン

- 6つのメソッドからなる

```
def matchstr(exp, str) ... end   # インターフェース
def try(exp, seq, pos) ... end   # ディスパッチャ
def try_empseq(seq, pos) ... end# 空文字列 //
def try_lit(sym, seq, pos) ... end # 文字 /x/
def try_cat(e1, e2, seq, pos) ... end # 連接 /XY/
def try_alt(e1, e2, seq, pos) ... end # 選択 /X|Y/
def try_rep(exp, seq, pos) ... end # 繰り返し /X*/
```

各メソッドの呼出関係


matchstr
try
相互再帰
try_empseq
try_lit
try_cat
try_alt
try_rep
自己再帰

要素技術

- 文字列の文字への分割 str.split(//)
- 多重代入
- 抽象構文木による正規表現

matchstr

```
def matchstr(exp, str)
  result = []
  try(exp, str.split(//), 0) {|pos|
    result << pos
  }
  result
end
```

try が yield した値を集めて配列として返す
空でない配列になればマッチ

matchstr

- 文字列の先頭からパターンマッチを行う
- マッチ可能な終端の位置を全て集めて配列として返す

```
p matchstr([:lit, "a"], "abc")          # [1]
p matchstr([:rep, [:lit, "a"]], "aa")# [2,1,0]
```

文字列の文字への分割

- String#split(sep)
- sep 区切りで分割して配列になる
- str.split(//) と空文字列で分割すれば文字単位
- 例

```
"".split(//)              #=> []
"a".split(//)             #=> ["a"]
"abcde".split(//)  #=>["a", "b", "c", "d", "e"]
```

- 正規表現エンジンは文字の配列にしてから処理を行う

matchstrのsplit

```
def matchstr(exp, str)
  result = []
  try(exp, str.split(//), 0) {|pos|
    result << pos
  }
  result
end
```

tryの仕様

- try(exp, seq, pos1) {|pos2| ... }
- exp は正規表現の抽象構文木
- seq は文字の配列
- pos1 はマッチを始める位置
- pos2 はマッチが終わった次の位置
- マッチする可能性すべてについてブロックを呼び出す
  - まったくマッチしなければ呼び出さない
  - 可能性がひとつしかなければ1回だけ呼び出す
  - いろんな可能性があればその数だけ呼び出す

正規表現の抽象構文木

- 正規表現オブジェクトは中身にアクセスできないので違う形で表現する
- 配列、シンボル、文字の組合せ
  - 空文字列 [:empseq]　　# //
  - 文字 [:lit, "x"]　　# /x/
  - 連接 [:cat, e1, e2]　　# /e1e2/
  - 選択 [:alt, e1, e2]　　# /e1|e2/
  - 繰り返し [:rep, e]　　# /e*/
- :xxx はシンボル。種類を表現する名前に使用
- 以前の四則演算の式と似た表現方法
- 配列以外の表現も考えられる

抽象構文木の例

- [:cat, [:lit, "a"],
  [:cat, [:lit, "b"],
  [:lit, "c"]]]
- [:cat, [:rep, [:alt, [:lit "a"],
  [:lit "b"]]],
  [:lit, "c"]]
- [:alt, [:lit, "a"], [:empseq]]
- [:cat,[:cat,[:lit,"a"],
  [:rep,[:lit,"a"]]],
  [:cat,[:lit,"b"],
  [:rep,[:lit,"b"]]]]

- /abc/
- /(a|b)*c/
- /a|/
- /aa*bb*/

tryの実行例

- try([:lit, "a"], ["a", "b", "c"], 0) {|pos| p pos } # 1
- try([:lit, "z"], ["a", "b", "c"], 0) {|pos| p pos } # 無し
- try([:lit, "a"], ["a", "b", "c"], 1) {|pos| p pos } # 無し
- try([:rep, [:lit, "a"]], ["a", "b", "c"], 0) {|pos| p pos } #1, 0
- try([:rep, [:lit, "a"]], ["a", "b", "c"], 2) {|pos| p pos } # 2
- try([:rep, [:lit, "a"]], ["a", "b", "c"], 3) {|pos| p pos } # 3
- try([:rep, [:lit, "a"]],["a","a","a"], 0) {|pos| p pos}
  # 3,2,1,0

文字列の配列の記法

- ["a", "b", "c"] を %w[a b c] と書ける

  p %w[a b c]　　#=> ["a", "b", "c"]

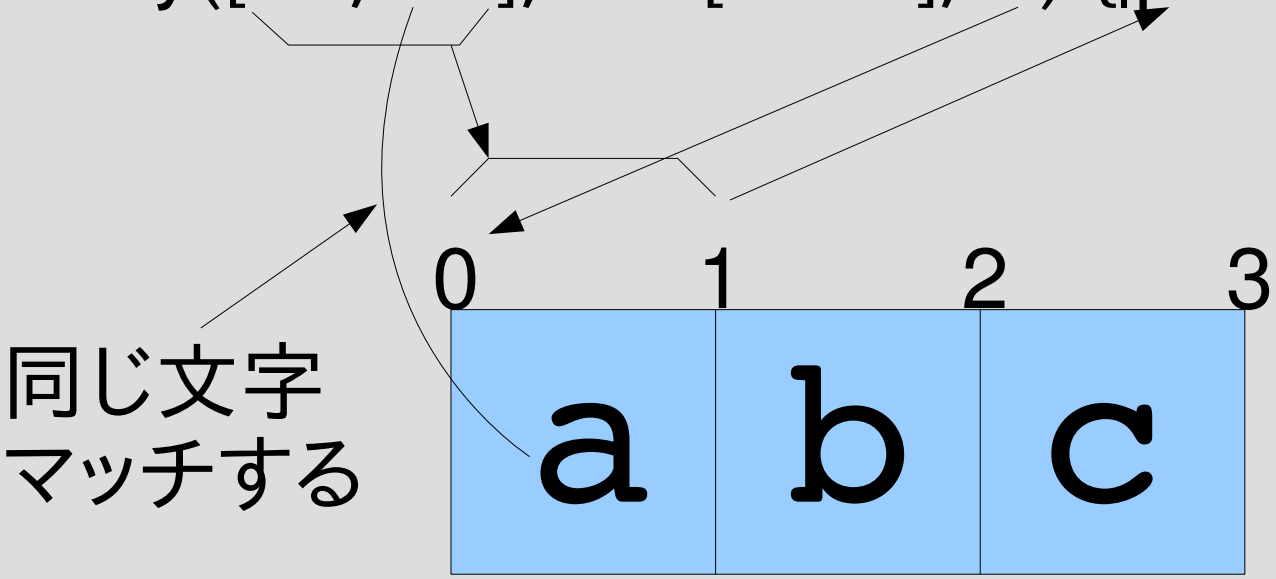
- w は word の頭文字

tryの実行例 (%w を使う)

- try([:lit, "a"], %w[a b c], 0) {|pos| p pos } # 1
- try([:lit, "z"], %w[a b c], 0) {|pos| p pos } # 無し
- try([:lit, "a"], %w[a b c], 1) {|pos| p pos } # 無し
- try([:rep, [:lit, "a"]], %w[a b c], 0) {|pos| p pos } #1, 0
- try([:rep, [:lit, "a"]], %w[a b c], 2) {|pos| p pos } # 2
- try([:rep, [:lit, "a"]], %w[a b c], 3) {|pos| p pos } # 3
- try([:rep, [:lit, "a"]],%w[a b c], 0) {|pos| p pos}
  # 3,2,1,0

"abc" の 0文字目から /a/

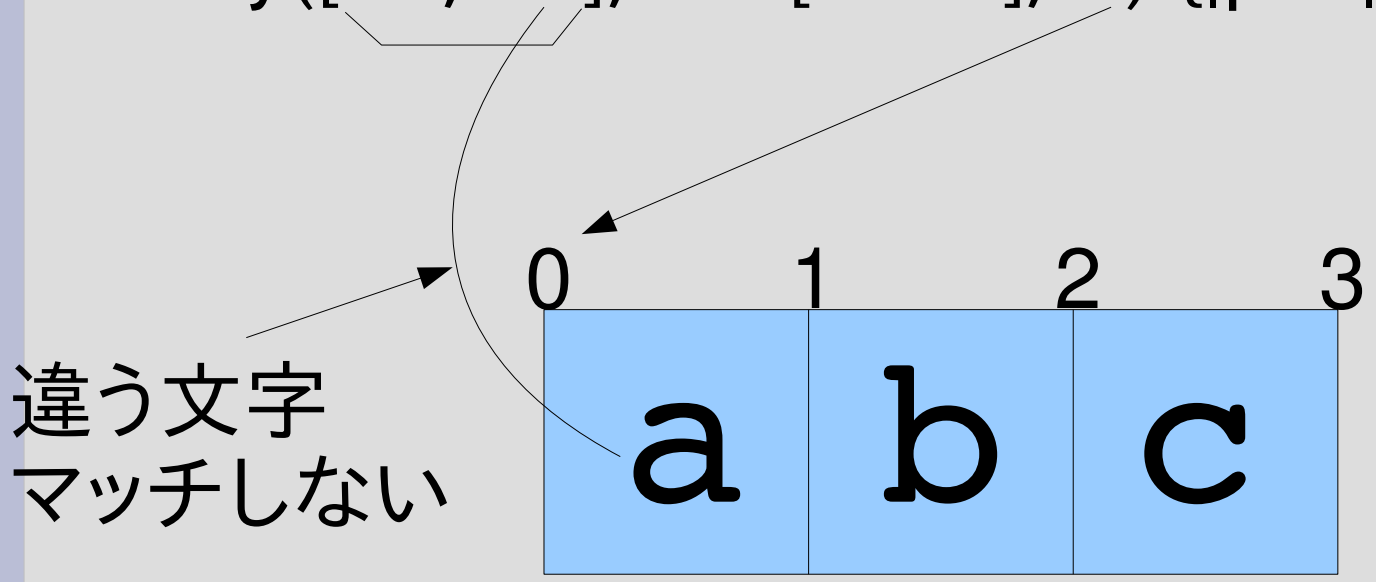
- try([:lit, "a"], %w[a b c], 0) {|pos| p pos }



- 1 が一回 yield される

"abc" の 0文字目から /z/

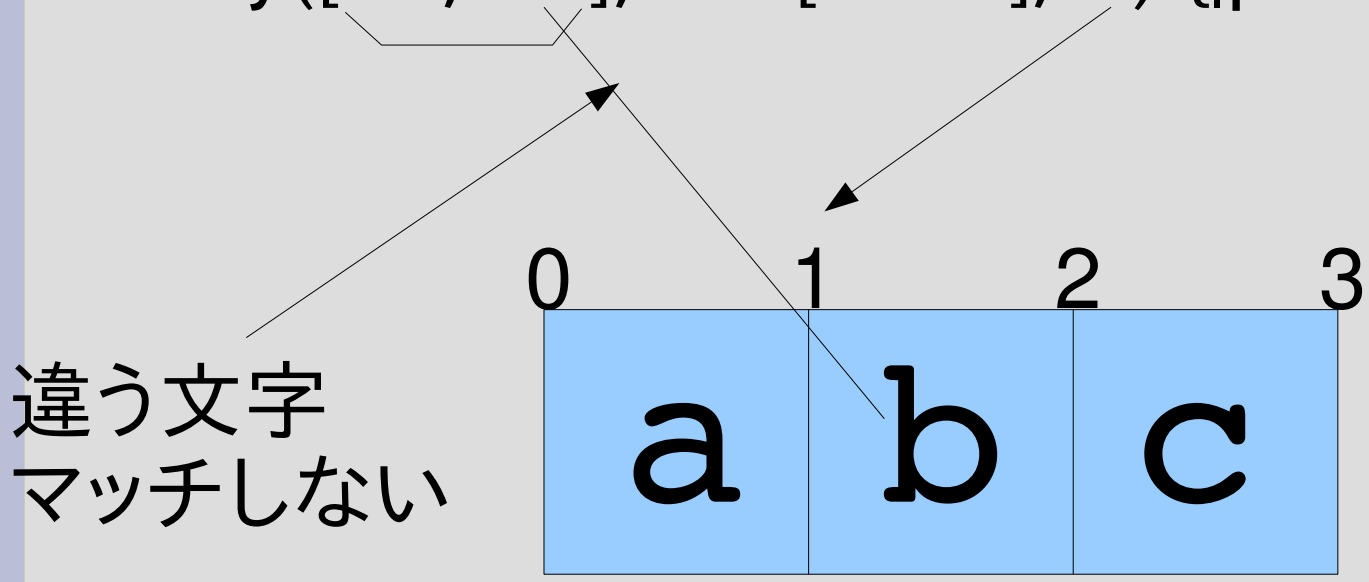
- try([:lit, "z"], %w[a b c], 0) {|pos| p pos }



- マッチしないので一回も yield されない

"abc" の 1文字目から /a/

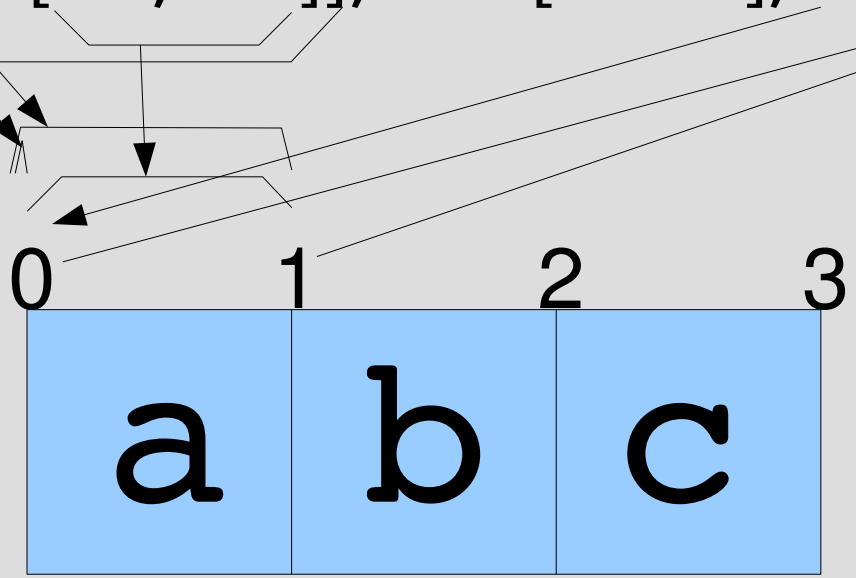
- try([:lit, "a"], %w[a b c], 1) {|pos| p pos }



- マッチしないので一回も yield されない

"abc" の 0文字目から /a*/

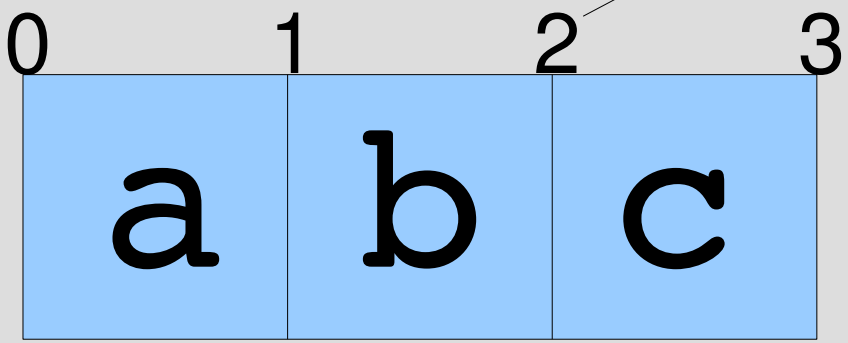
- try([:rep, [:lit, "a"]], %w[a b c], 0) {|pos| p pos }



- /a*/ は "a" と "" にマッチする
- 1 と 0 が順に yield される
- たくさん繰り返した方が先 (最長一致)

"abc" の 2文字目から /a*/

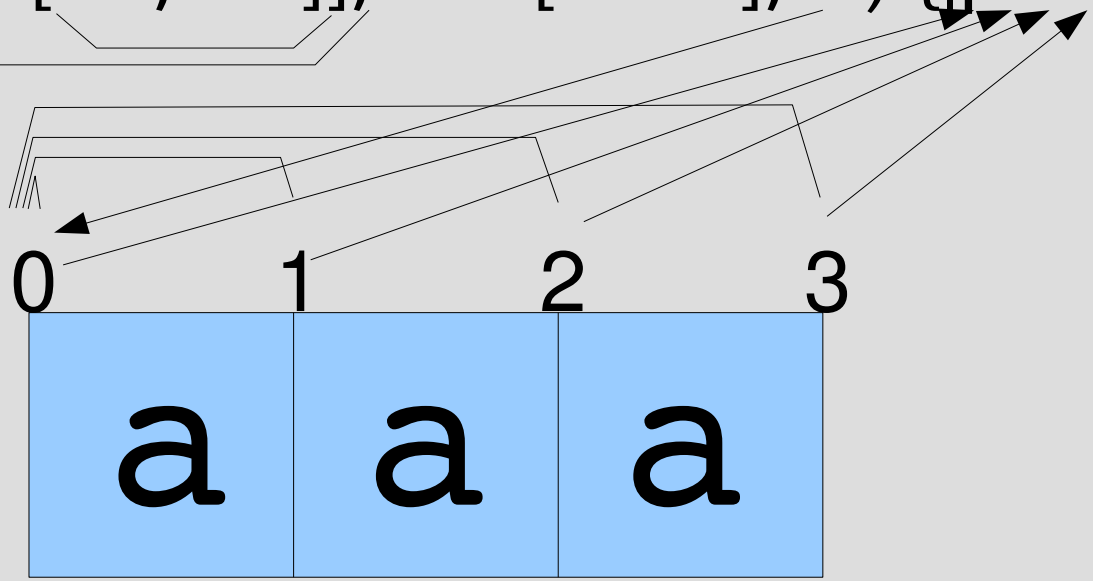
- try([:rep, [:lit, "a"]], %w[a b c], 2) {|pos| p pos }



- /a*/ は空文字列 "" にマッチする
- 2 が yield される

"aaa" の 0文字目から /a*/

- try([:rep, [:lit, "a"]], %w[a a a], 0) {|pos| p pos }



- /a*/ は "aaa", "aa", "a", "" にマッチする
- 3,2,1,0 の順で yield される

try

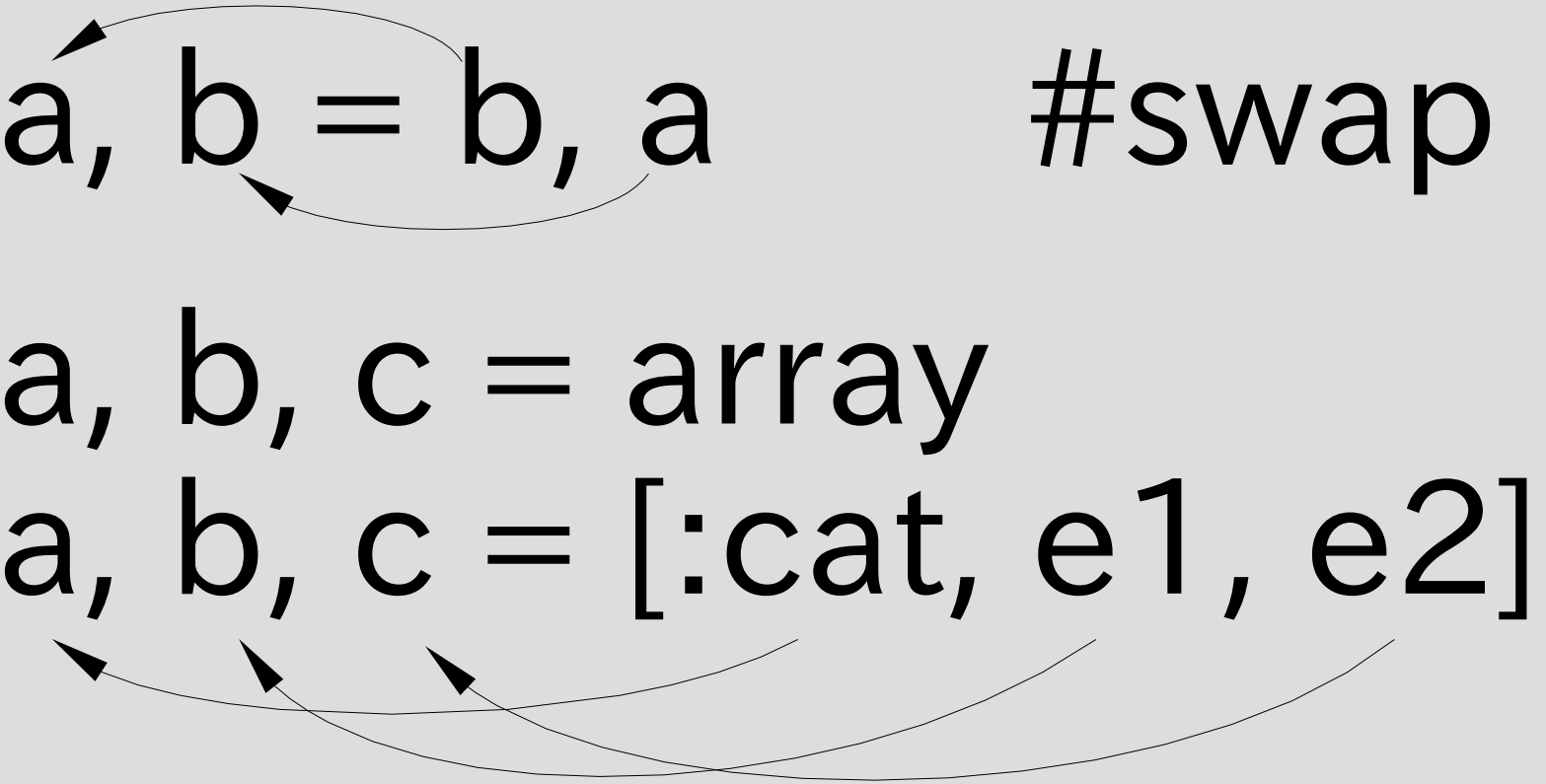
```
def try(exp, seq, pos, &b)
  case exp[0]
  when :empseq
    try_empseq(seq, pos, &b)
  when :lit
    _, sym = exp
    try_lit(sym, seq, pos, &b)
  when :cat
    _, e1, e2 = exp
    try_cat(e1, e2, seq, pos, &b)
  when :alt
    _, e1, e2 = exp
    try_alt(e1, e2, seq, pos, &b)
  when :rep
    _, e = exp
    try_rep(e, seq, pos, &b)
  end
end
```

多重代入

- 代入の左辺、右辺には複数の式や配列を書ける

- a, b = b, a　　#swap

- a, b, c = array
  a, b, c = [:cat, e1, e2]



tryの多重代入

```
def try(exp, seq, pos, &b)
  case exp[0]
  when :empseq
    try_empseq(seq, pos, &b)
  when :lit
    _, sym = exp
    try_lit(sym, seq, pos, &b)
  when :cat
    _, e1, e2 = exp
    try_cat(e1, e2, seq, pos, &b)
  when :alt
    _, e1, e2 = exp
    try_alt(e1, e2, seq, pos, &b
  when :rep
    _, e = exp
    try_rep(e, seq, pos, &b)
  end
end
```

tryでのブロックの引きわたし

```
def try(exp, seq, pos, &b)
  case exp[0]
  when :empseq
    try_empseq(seq, pos, &b)
  when :lit
    _, sym = exp
    try_lit(sym, seq, pos, &b)
  when :cat
    _, e1, e2 = exp
    try_cat(e1, e2, seq, pos, &b)
  when :alt
    _, e1, e2 = exp
    try_alt(e1, e2, seq, pos, &b)
  when :rep
    _, e = exp
    try_rep(e, seq, pos, &b)
  end
end
```

tryの内容

- 与えられた exp の種類を case で判別
- exp の内容を多重代入で取り出し
- 種類に応じて try_xxx を呼ぶ

try_empseq

- 空文字列だけ進めて yield
- 空文字列ということはぜんぜん進まないのでそのまま yield

```
def try_empseq(seq, pos)
  yield pos
end

try([:empseq], [], 0) {|pos| p pos }   # 0
```

try_lit

- 一文字進められれば、進んだ所を yield

```
def try_lit(sym, seq, pos)
  if pos < seq.length && seq[pos] == sym
    yield pos + 1
  end
end

try([:lit, "a"], ["a"], 0) {|pos| p pos }   # 1
```

try_lit の中身

- 一文字進められれば、進んだ所を yield

```
def try_lit(sym, seq, pos)
  if pos < seq.length && seq[pos] == sym
    yield pos + 1
  end
end
```

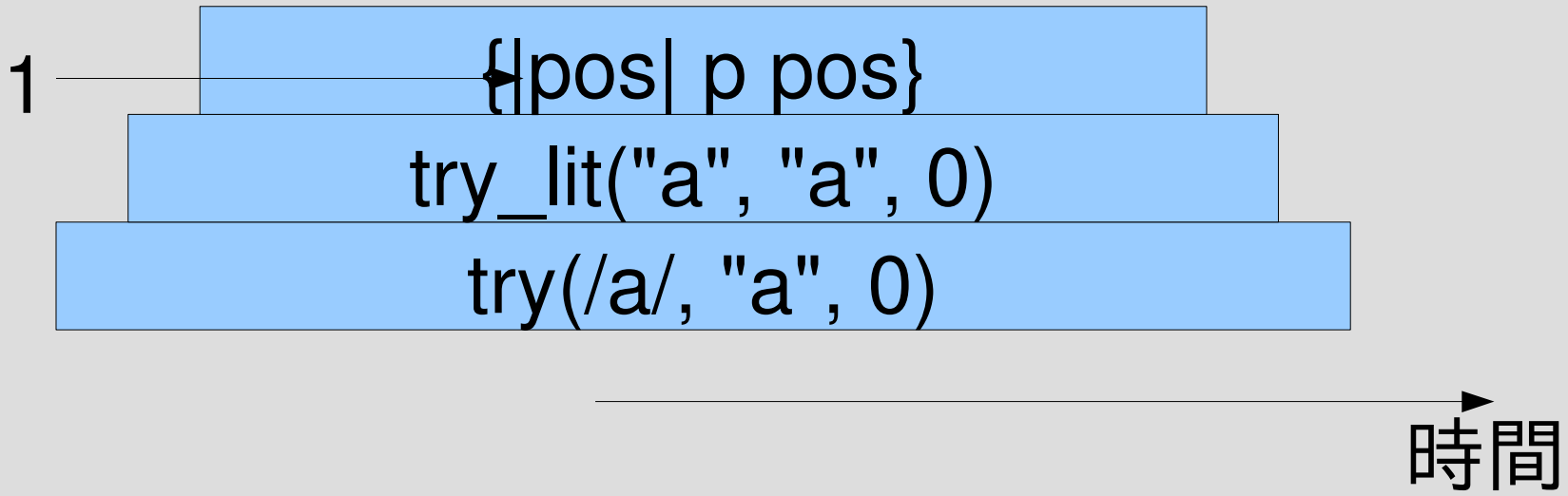
まだ文字が存在する

その文字が等しい

一文字進めたインデックス

try_litの動作 (マッチする場合)

try([:lit, "a"], ["a"], 0) {|pos| p pos } # 1

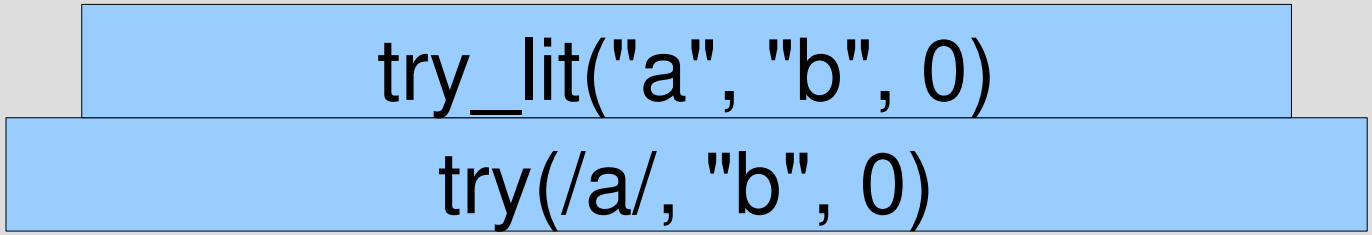


抽象構文木、文字の配列は繁雑なので
正規表現、文字列で表現してある

try_litの動作 (マッチしない場合)

try([:lit, "a"], ["b"], 0) {|pos| p pos }

マッチしないので
ブロックは呼ばれない



抽象構文木、文字の配列は繁雑なので
正規表現、文字列で表現してある

try_cat

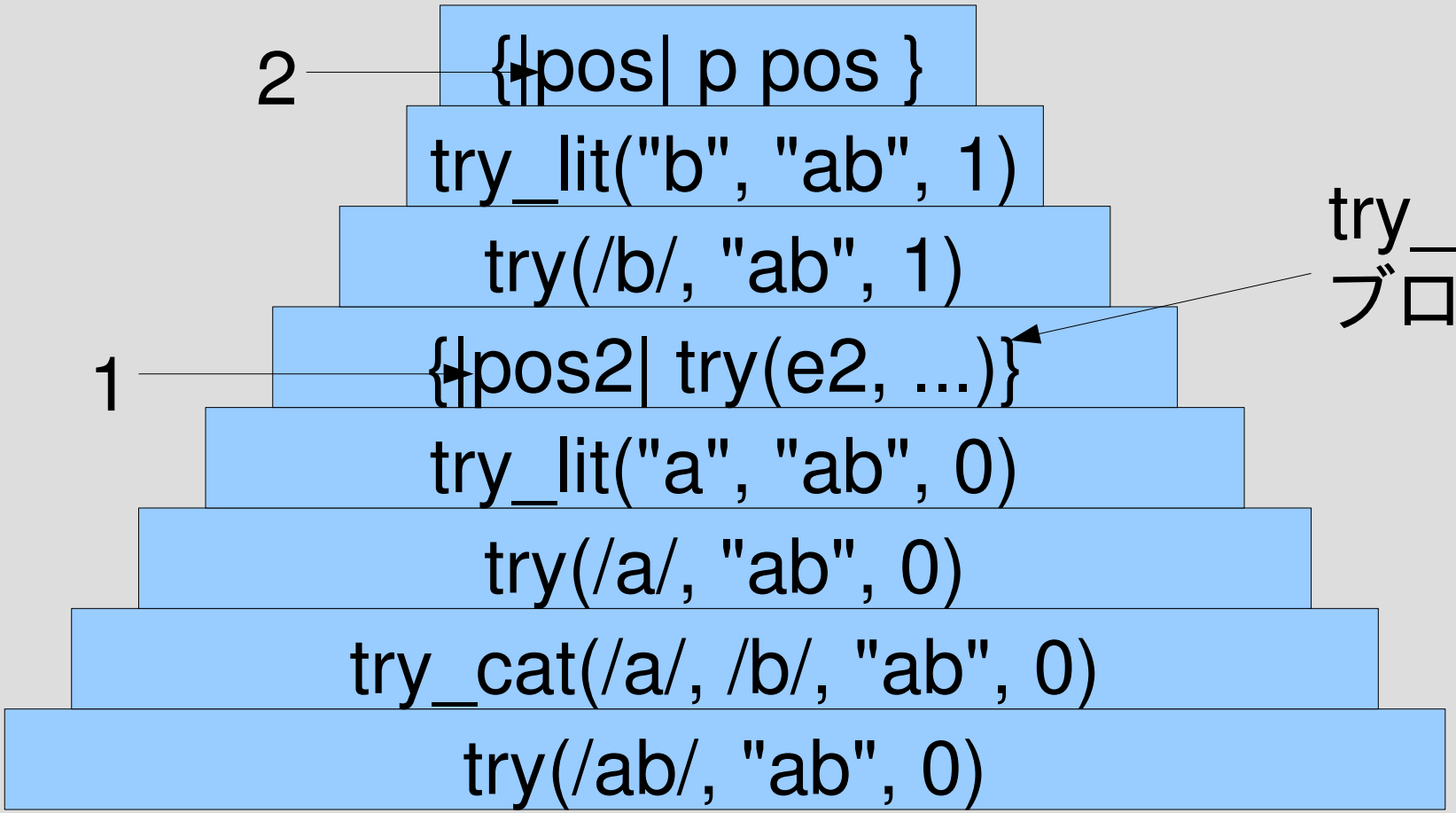
- e1 を try で進めて、進んだ所から e2 をさらに進める

```
def try_cat(e1, e2, seq, pos, &b)
  try(e1, seq, pos) {|pos2|
    try(e2, seq, pos2, &b)
  }
end

try([:cat, [:lit, "a"], [:lit, "b"]], %w[a b], 0) {|pos| p pos}
# 2
```

try_catの動作

try([:cat, [:lit, "a"], [:lit, "b"]], %w[a b], 0) {|pos| p pos}
2



try_alt

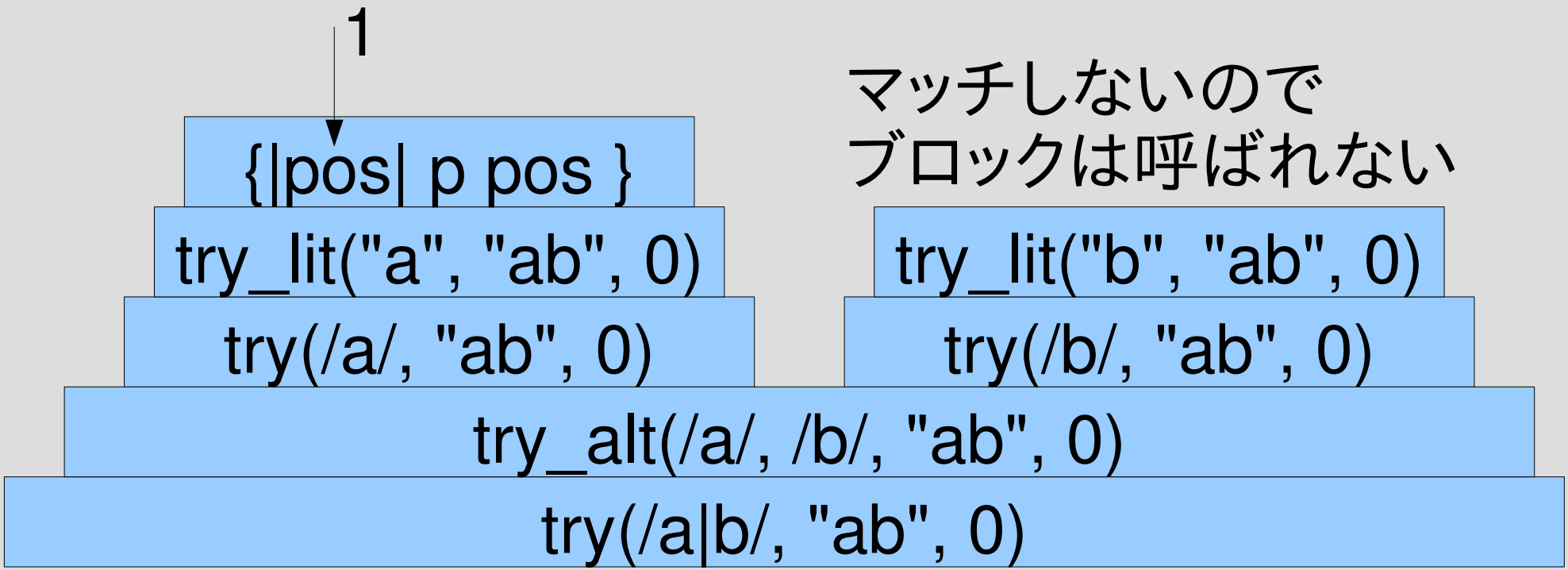
- e1 進めるのを試して、また、e2 進めるのを試す

```
def try_alt(e1, e2, seq, pos, &b)
  try(e1, seq, pos, &b)
  try(e2, seq, pos, &b)
end

try([:alt, [:lit, "a"], [:lit, "b"]], %w[a b], 0) {|pos| p pos }
# 1
```

try_altの動作

```
try([:alt, [:lit, "a"], [:lit, "b"]], %w[a b], 0) {|pos| p pos}
   # 1
```



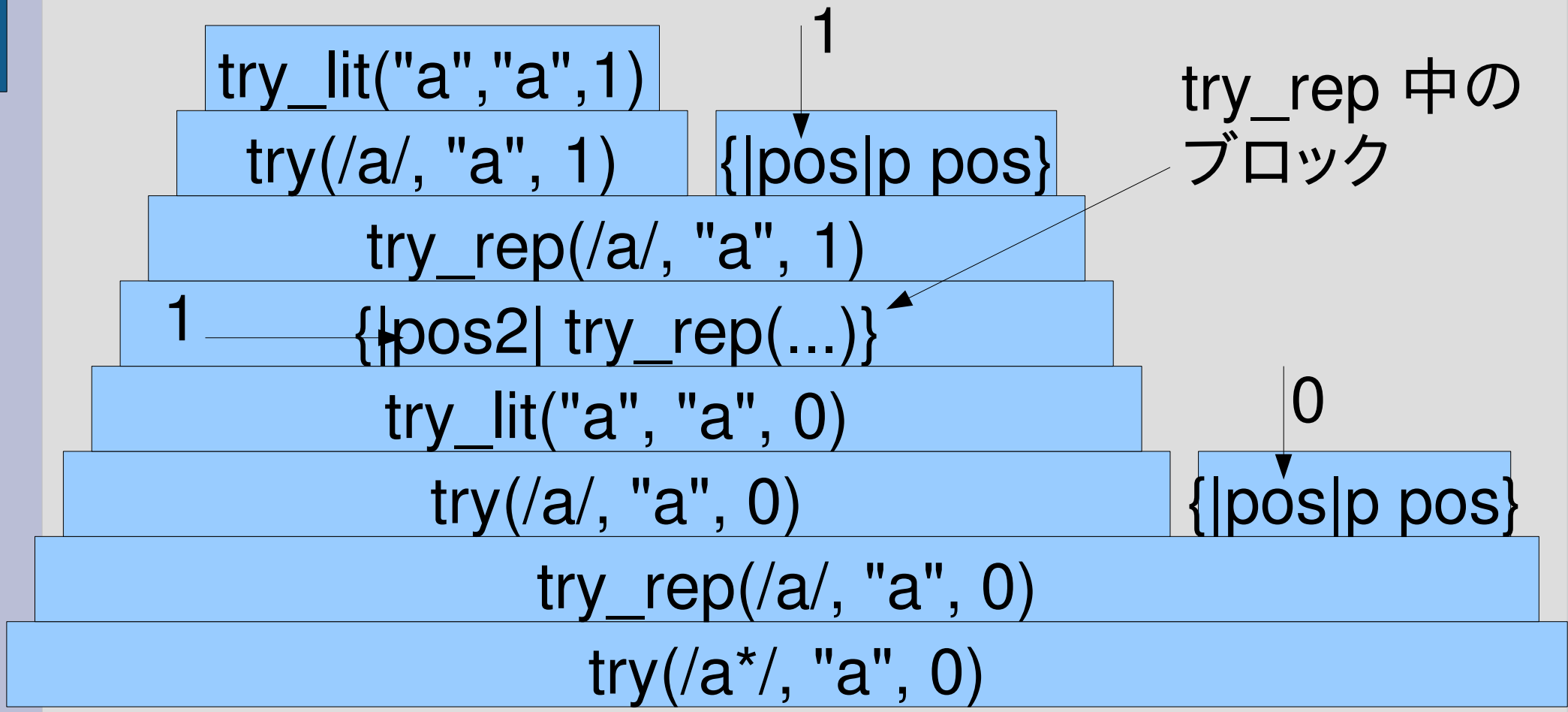
try_rep

- e を進められるだけ進める
  - とりあえず try でひとつ進める
  - ひとつ進めた後に try_rep で進められるだけ進める
- 無限再帰の可能性は気にしない (今は)

```
def try_rep(e, seq, pos, &b)
  try(e, seq, pos) {|pos2|
    try_rep(e, seq, pos2, &b)
  }
  yield pos
end
try([:rep, [:lit, "a"]], "a", 0) {|pos| p pos } # 1,0
```

try_repの動作

try([:rep, [:lit, "a"]], %w[a], 0) {|pos| p pos} # 1,0



レポート: 正規表現エンジンの解説

- 以下の正規表現を抽象構文木に変換し、今回の正規表現エンジンを用いて動かして動作を解説せよ
  - /(abc)*abc/
  - /abc(abc)*/
  - /(a|b)*abc/
- ∠切 2007-06-26 16:20
- HIPLUS
- 拡張子が txt なテキストファイルにしてほしい

解説のしかた

- 正規表現の動作がわかるような文字列を適切に選ぶ
- その文字列を正規表現にマッチさせたときの動作を調べて説明する

- 注意
  - とくにどういうふうに再帰しているか述べる
  - matchstr(..., ...) を実行して返り値だけを示すことは動作の解説にはならない

まとめ

- レポートの解説
- 簡単な正規表現エンジン
  - empseq, lit, cat, alt, rep
- レポートを出した