テキスト処理 第6回 (2006-05-30)

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/

今日の内容

- 正規表現エンジン動作解説レポートの解説
- 前回の残りの try_rep
- 正規表現エンジンの停止性
- 正規表現エンジンの計算量
- レポート

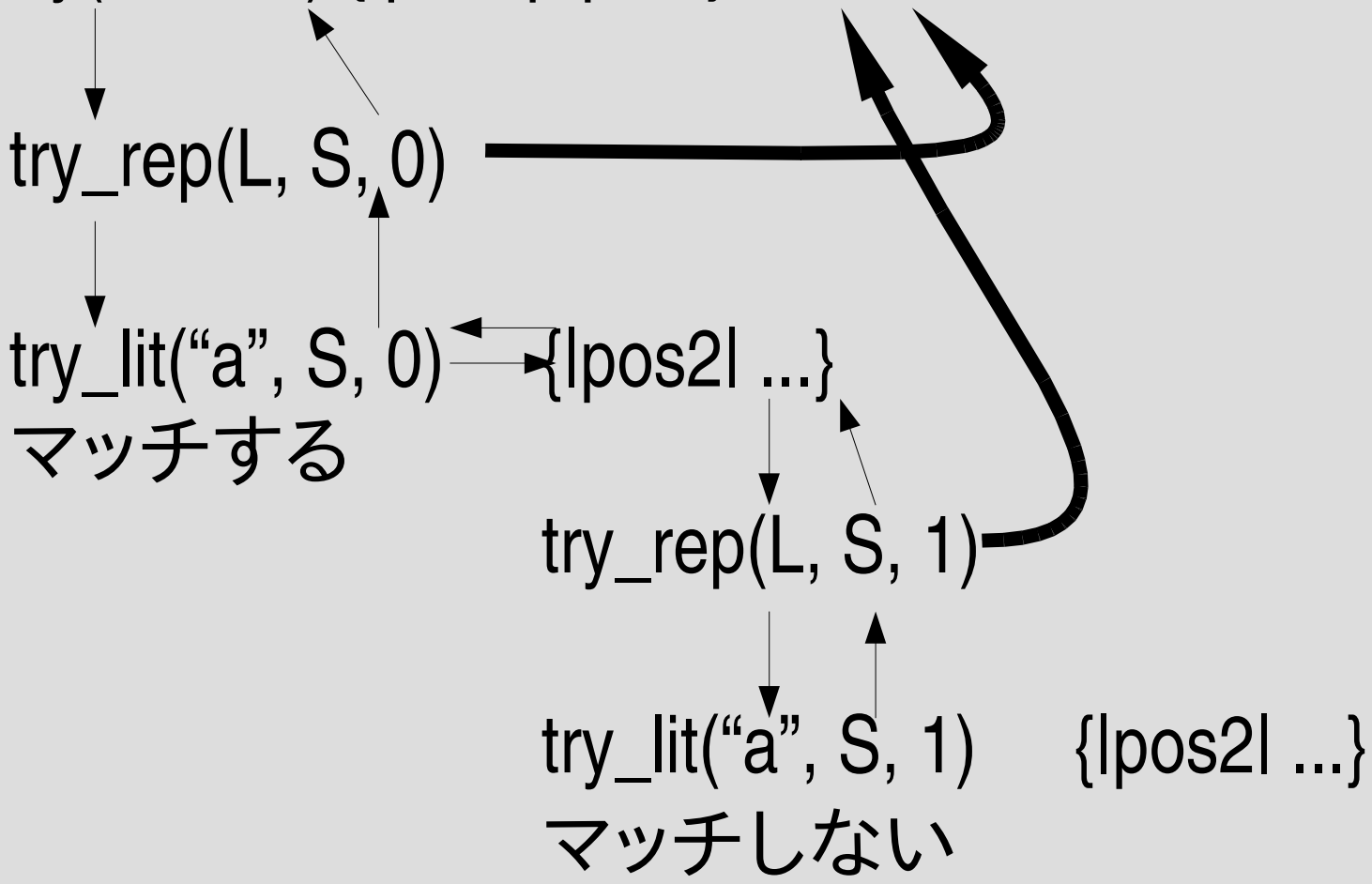
try_rep

- 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 (後で考える)

```
def try_rep(e, seq, pos, &b)
  try(e, seq, pos) {|pos2|
    try_rep(e, seq, pos2, &b)
  }
  yield pos
end
```

try_rep の実行

- L = [:lit, “a”]　　R = [:rep, L]　　S = [“a”]
- try(R, S, 0) {|pos| p pos } # 1, 0



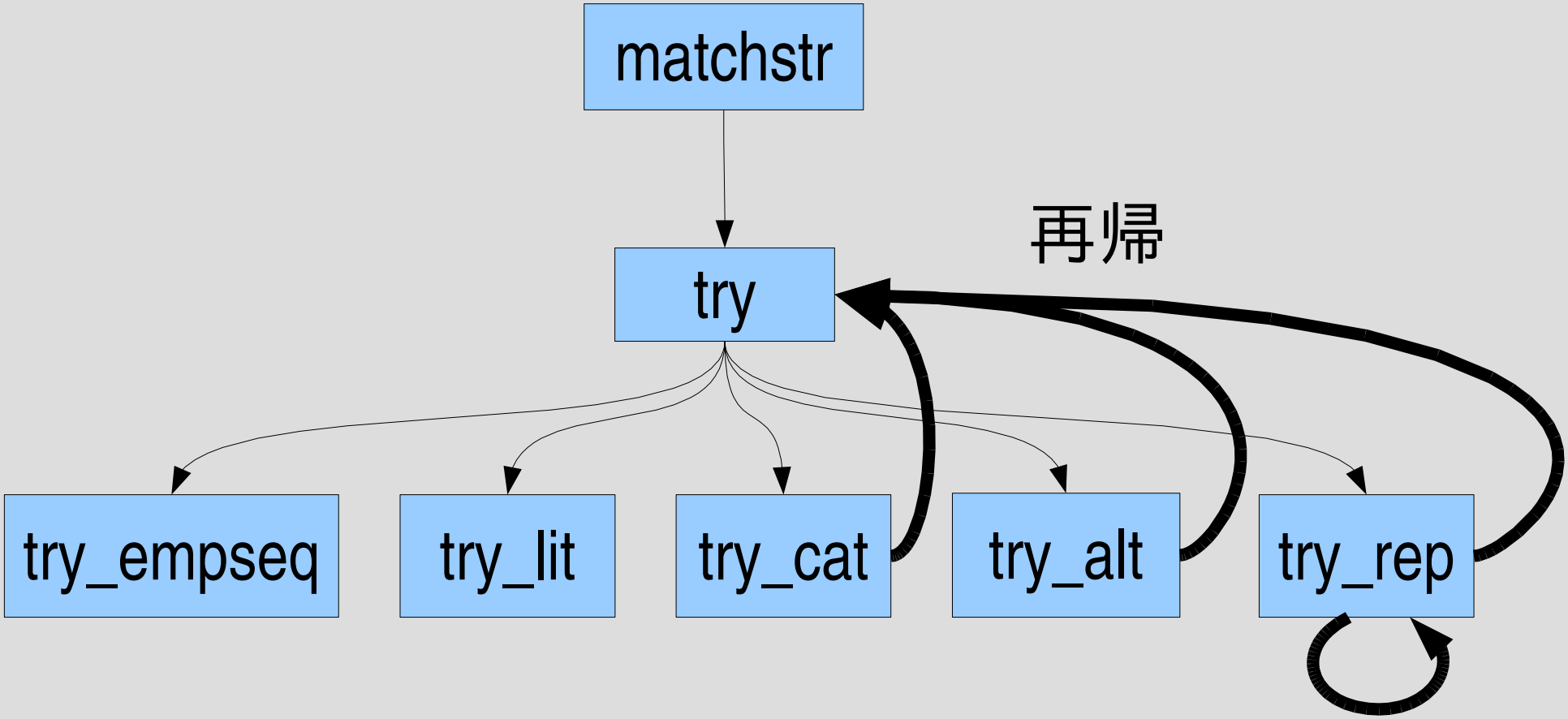
停止性

- 以下は無限に再帰して止まらない
  matchstr([:rep, [:empseq]], "")
- 実際にはスタックが溢れて止まる

```
% ruby -rrx -e 'matchstr([:rep, [:empseq]], "")'
/tmp/rx.rb:16:in `try': stack level too deep (SystemStackError)
        from /tmp/rx.rb:58:in `try_rep'
        from /tmp/rx.rb:59:in `try_rep'
        from /tmp/rx.rb:36:in `try_empseq'
        from /tmp/rx.rb:17:in `try'
        from /tmp/rx.rb:58:in `try_rep'
        from /tmp/rx.rb:59:in `try_rep'
        from /tmp/rx.rb:36:in `try_empseq'
        from /tmp/rx.rb:17:in `try'
         ... 6748 levels...
        from /tmp/rx.rb:58:in `try_rep'
        from /tmp/rx.rb:29:in `try'
        from /tmp/rx.rb:7:in `matchstr'
        from -e:1
```

メソッド呼び出しは戻り先などを記録する必要がある
そのような情報はスタックに記録される

呼出関係の再帰



無限に再帰するとスタックが溢れる　　再帰

再帰を停止させる方法

- ひとまわりごとに少しでも処理を進める
- 処理が有限であれば終わる

正規表現エンジンの処理の進み

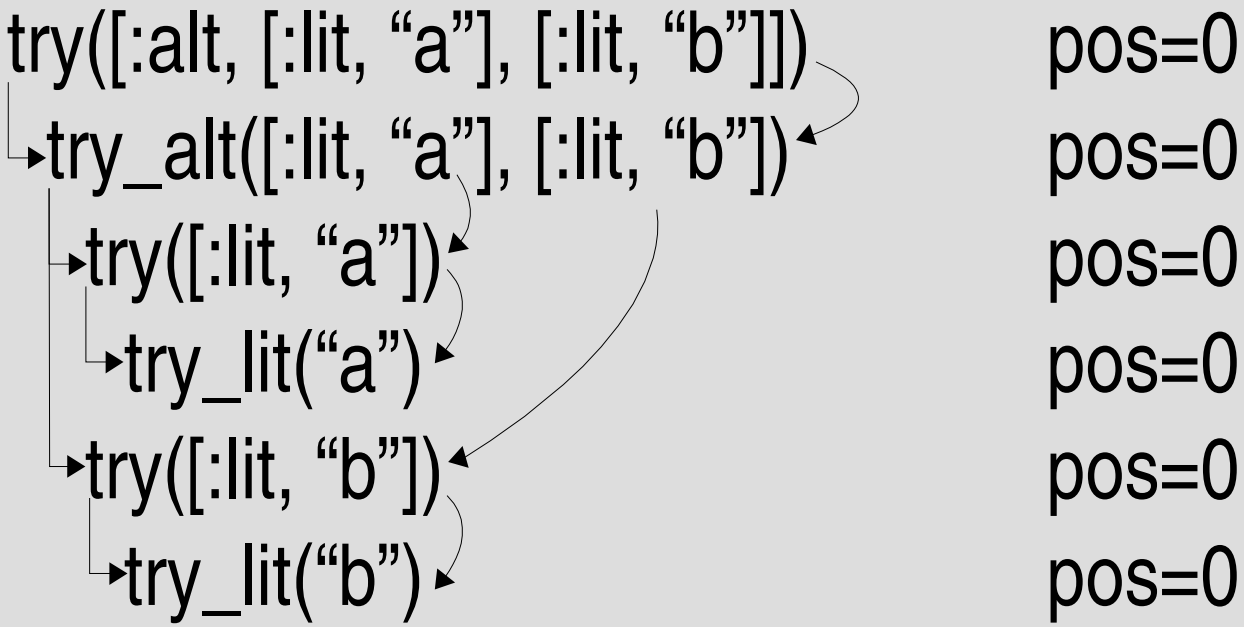
- 処理が進む:
  - 残りの文字列が短くなる
  - パターンが小さくなる
- 再帰するたびに処理が進めば、無限には再帰しない
  - 文字列の長さは有限
  - パターンの大きさは有限

try_alt

- [:alt, e1, e2] より e1 と e2 は小さい
- try([:alt, e1, e2]) は try(e1) と try(e2) を呼び出すのでパターンが小さくなっている
- 文字列の残り (pos 以降) は変わらない

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

/a|b/ =~ “z”


try([:alt, [:lit, “a”], [:lit, “b”]])
try_alt([:lit, “a”], [:lit, “b”])
try([:lit, “a”])
try_lit(“a”)
try([:lit, “b”])
try_lit(“b”)
pos=0
pos=0
pos=0
pos=0
pos=0
pos=0

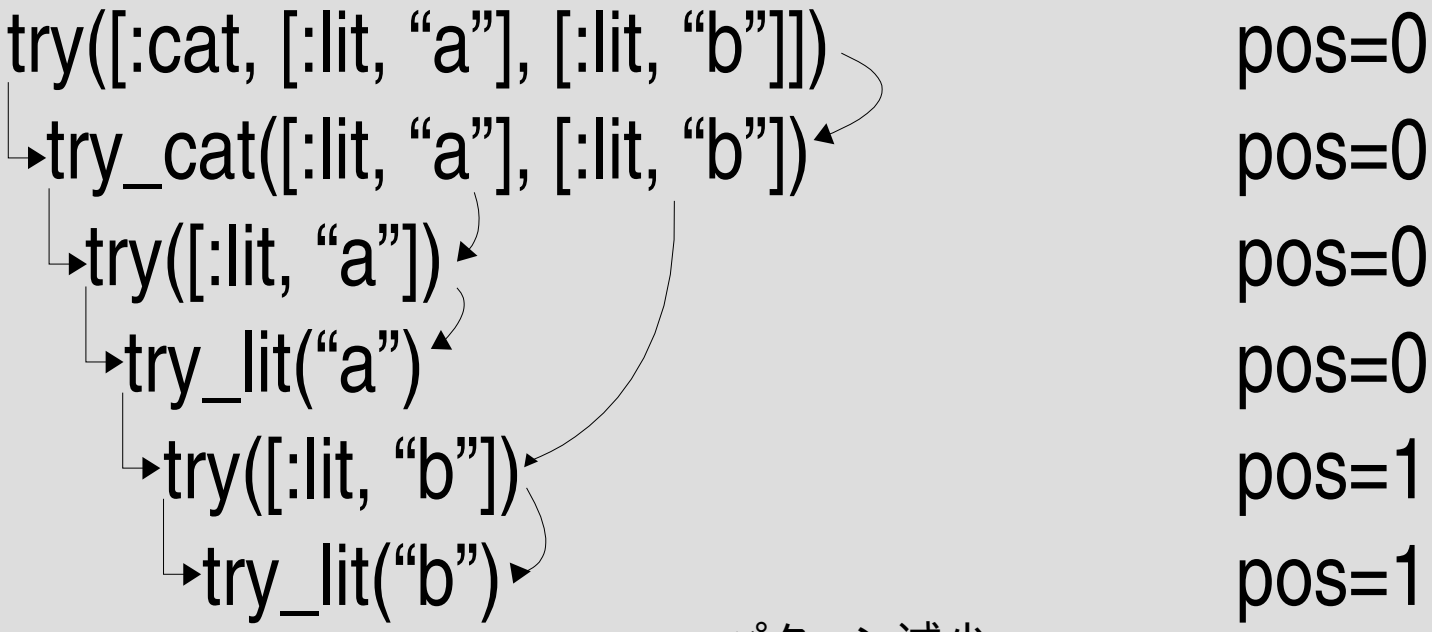
パターン減少

try_cat

- [:cat, e1, e2] より e1 と e2 は小さい
- try([:cat, e1, e2]) は try(e1) と try(e2) を呼び出し、そのときパターンは小さくなっている
- 文字列の残りは、増えることはない
  (try(e2) については減るかもしれない)

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

/ab/ =~ “abc”



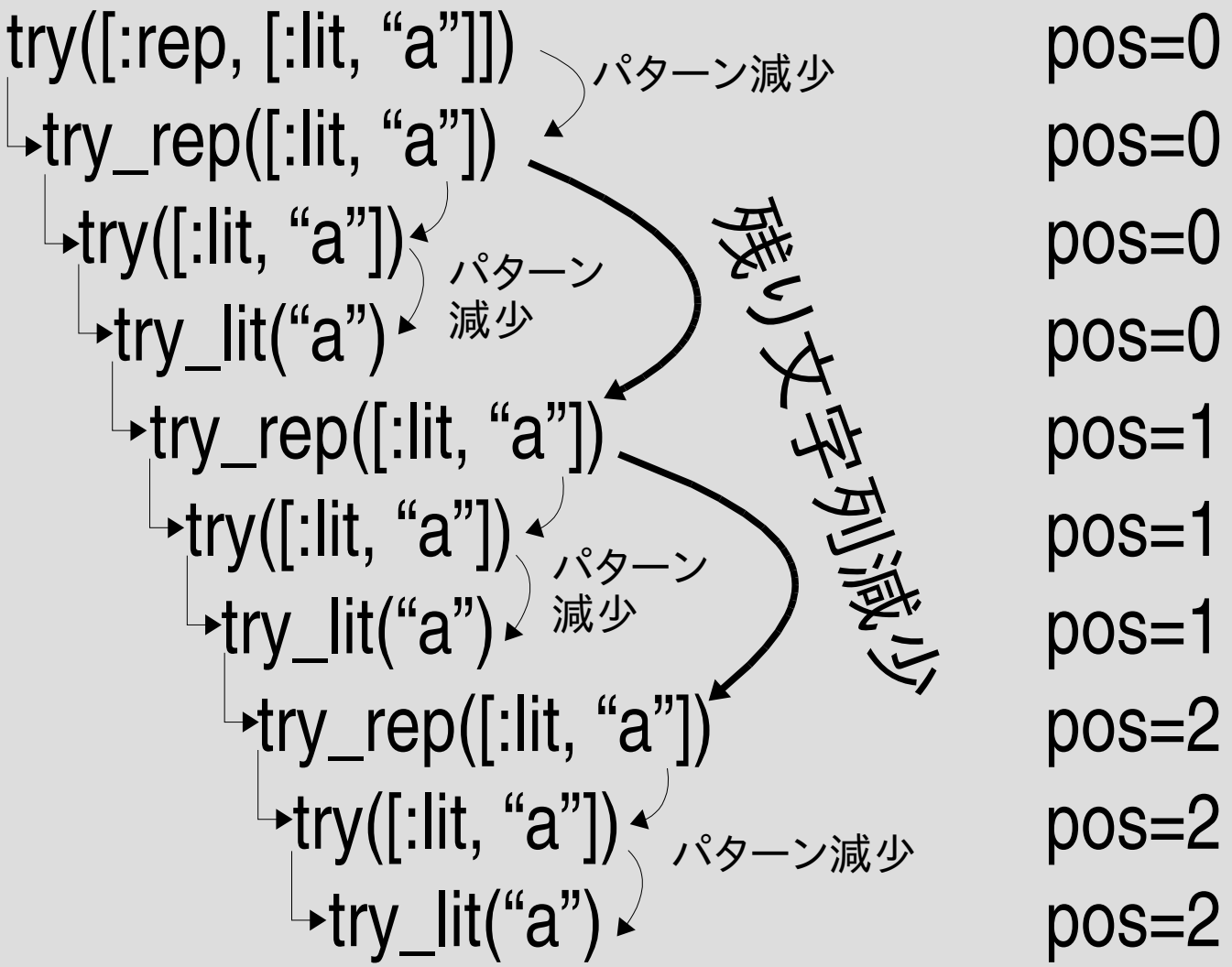
パターン減少

try_rep

- [:rep, exp] より exp のほうが小さい
- try([:rep, exp]) は try(exp) を呼び、小さくなってる
- try_rep(exp) は try_rep(exp) と同じ大きさで呼ぶ
- そのとき、pos == pos2 かもしれない (exp に依存)

```
def try_rep(exp, seq, pos, &block)
  try(exp, seq, pos) {|pos2|
    try_rep(exp, seq, pos2, &block)
  }
  yield pos
end
```

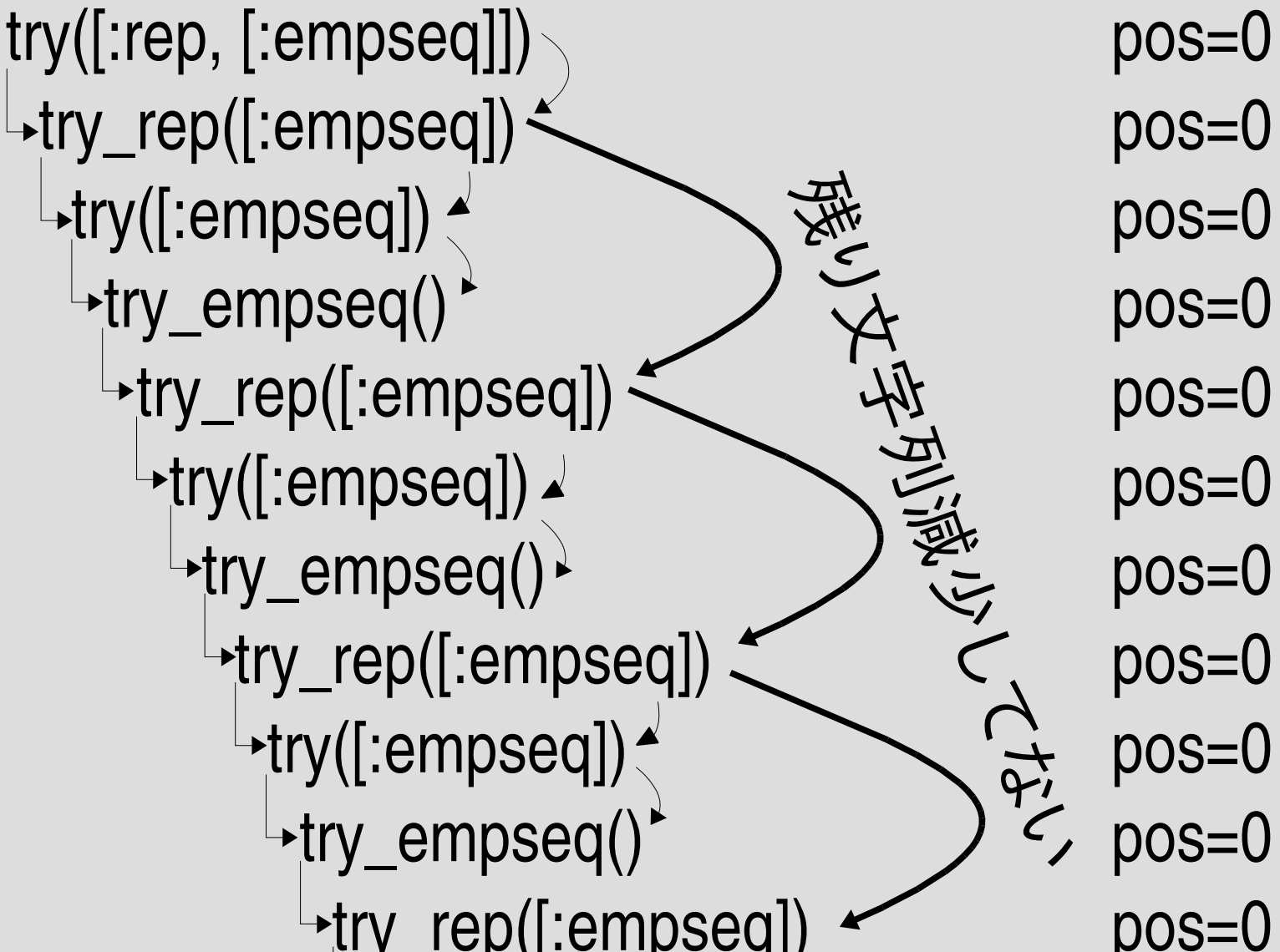
/a*/ =~ “aa”


try([:rep, [:lit, “a”]])
try_rep([:lit, “a”])
try([:lit, “a”])
try_lit(“a”)
try_rep([:lit, “a”])
try([:lit, “a”])
try_lit(“a”)
try_rep([:lit, “a”])
try([:lit, “a”])
try_lit(“a”)
パターン減少
パターン減少
パターン減少
パターン減少
残り文字列減少
pos=0
pos=0
pos=0
pos=0
pos=1
pos=1
pos=1
pos=2
pos=2
pos=2

try_repの無限再帰

- try_rep が try_rep を呼び出す
- パターンが小さくならない
  - これはいつも成り立つ
- 残りの文字列が小さくならない
  - exp が空文字列にマッチして pos == pos2 になったときに成り立つ

/()*/ =~ “aa”


try([:rep, [:empseq]])
try_rep([:empseq])
try([:empseq])
try_empseq()
try_rep([:empseq])
try([:empseq])
try_empseq()
try_rep([:empseq])
try([:empseq])
try_empseq()
try_rep([:empseq])
残り文字列減少してない
pos=0
pos=0
pos=0
pos=0
pos=0
pos=0
pos=0
pos=0
pos=0
pos=0

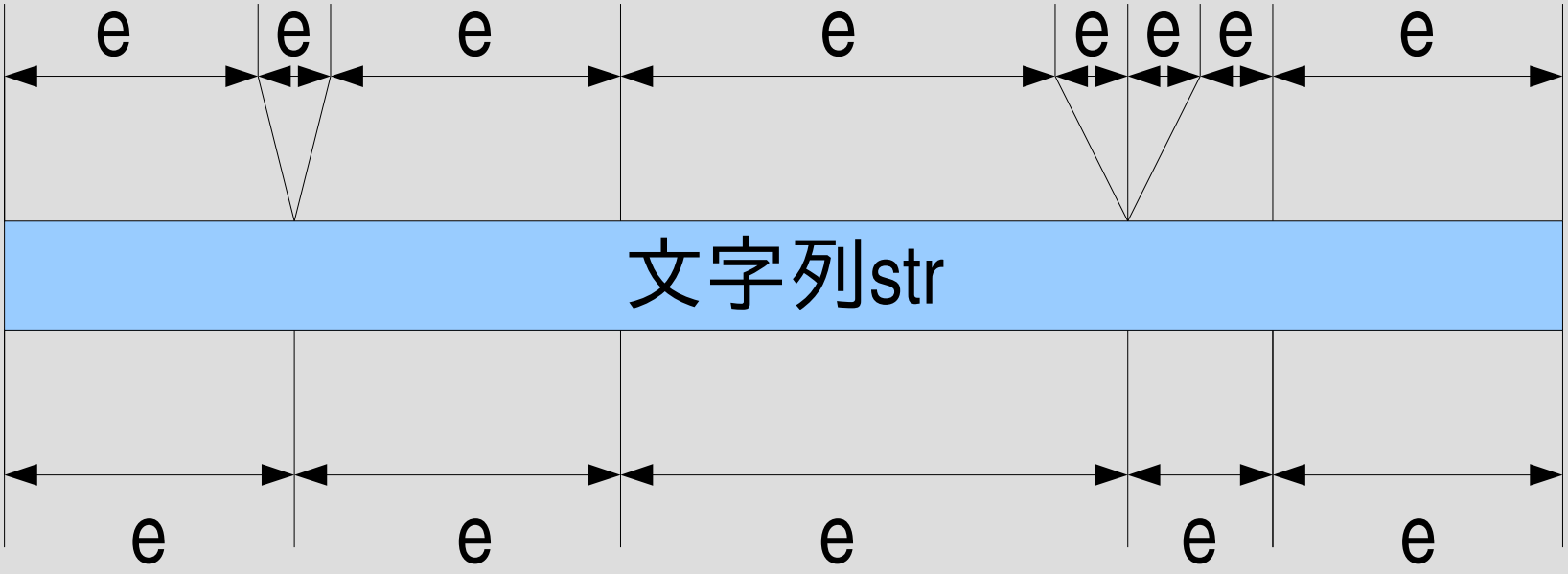
無限再帰防止

- exp が空文字列にマッチした場合は無視
- 変にマッチしなくなることはない

```
def try_rep(exp, seq, pos, &block)
  try(exp, seq, pos) {|pos2|
    try_rep(exp, seq, pos2, &block)  if pos < pos2
  }
  yield pos
end
```

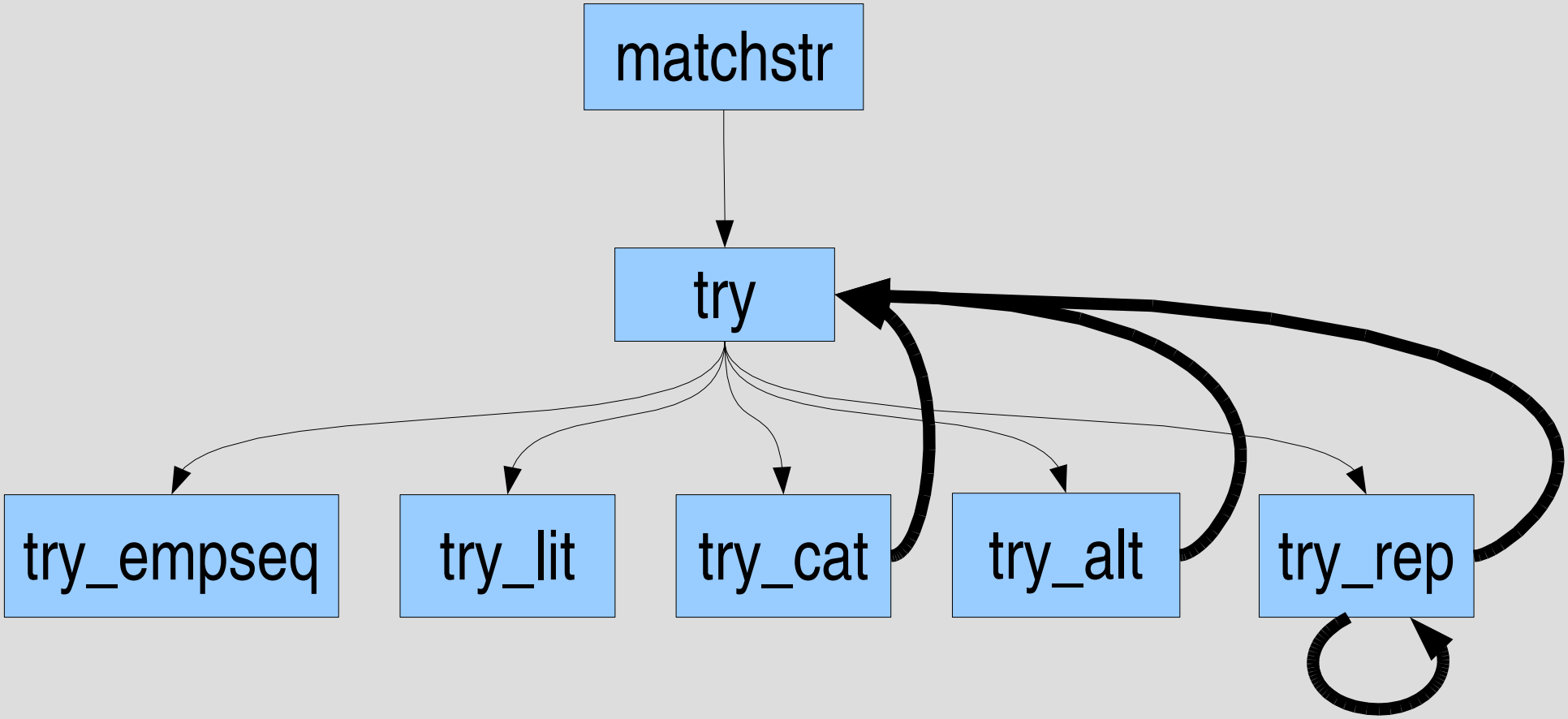
無視してもマッチするものはマッチする

/\Ae*\z/ =~ str



空文字列へのマッチを取り除いて、
全体へのマッチを作れる。
空文字列を無視しても
マッチしなくなるわけではない。

呼出関係



どう回っても問題が小さくなるのでそのうち終わる
(スタックが溢れる前に終わればうまくいく)

計算量

- どのくらい時間がかかるか？
- try は何回呼び出されるか？

try の呼び出しを数える

```
def count_try(exp, str)
  $try_count = 0
  matchstr(exp, str)
  $try_count
end

def try(exp, seq, pos, &block)
  $try_count += 1
  case exp[0]
  ...
```

要素技術

- グローバル変数
- 文字列の式展開
- 等差数列の和

グローバル変数

- $xxx のような変数はグローバル変数
- /\A\$[a-zA-Z_][a-zA-Z_0-9]*\z/
- いままで使っていた pos とかはローカル変数
- グローバル変数は必要ないかぎり使わない

try の呼び出しを数える

```
def count_try(exp, str)
  $try_count = 0
  matchstr(exp, str)
  $try_count
end

def try(exp, seq, pos, &block)
  $try_count += 1
  case exp[0]
  ...
```

a* を a の並びにマッチするとき

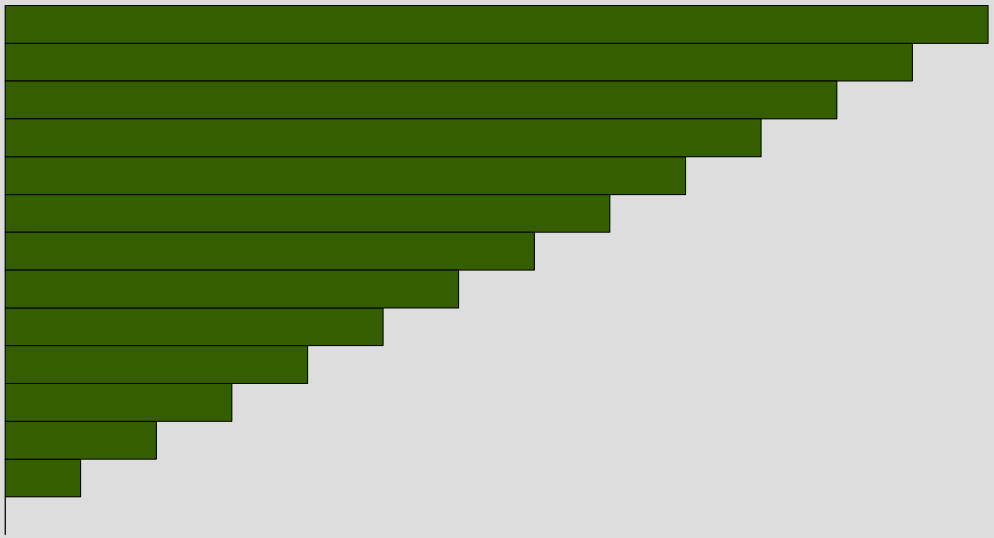
- count_try([:rep, [:lit, "a"]], "aaa") #=> 5
- count_try([:rep, [:lit, "a"]], "aaaaa") #=> 7
- count_try([:rep, [:lit, "a"]], "a"*10) #=> 12
- count_try([:rep, [:lit, "a"]], "a"*100) #=> 102
- count_try([:rep, [:lit, "a"]], "a"*1000) #=> 1002

- 文字列の長さを n として、n+2 回呼び出されている

a*の処理

- a を可能な限り繰り返しマッチ
  - マッチしなくなったら繰り返しを止める
- 長い方から yield

a	a	a	a	a	a	a	a	a	a	a	a	a



aaaa に対するマッチ

- try([:rep, [:lit, “a”]])　　pos=0
  try([:lit, “a”])　　pos=0　マッチする
  try([:lit, “a”])　　pos=1　マッチする
  try([:lit, “a”])　　pos=2　マッチする
  try([:lit, “a”])　　pos=3　マッチする
  try([:lit, “a”])　　pos=4　マッチしない

- aaaa は長さ 4
- マッチする 4回に加えて最初と最後で 4+2

0から20まで

```
0.upto(20) {|n|
  m = count_try([:rep, [:lit, "a"]], "a"*n)
  puts "#{n} #{m}"
}
=>
0 2
1 3
2 4
3 5
...
```

文字列の式展開

- “aaa#{式}bbb” というように、文字列の中に式を埋め込める
- 埋め込んだ式は毎回評価されて結果が文字列として埋め込まれる
- ダブルクオートの文字列に使える
- シングルクオートの文字列には使えない

0から20まで

```
0.upto(20) {|n|
  m = count_try([:rep, [:lit, "a"]], "a"*n)
  puts "#{n} #{m}"
}
```

=>

0 2

1 3

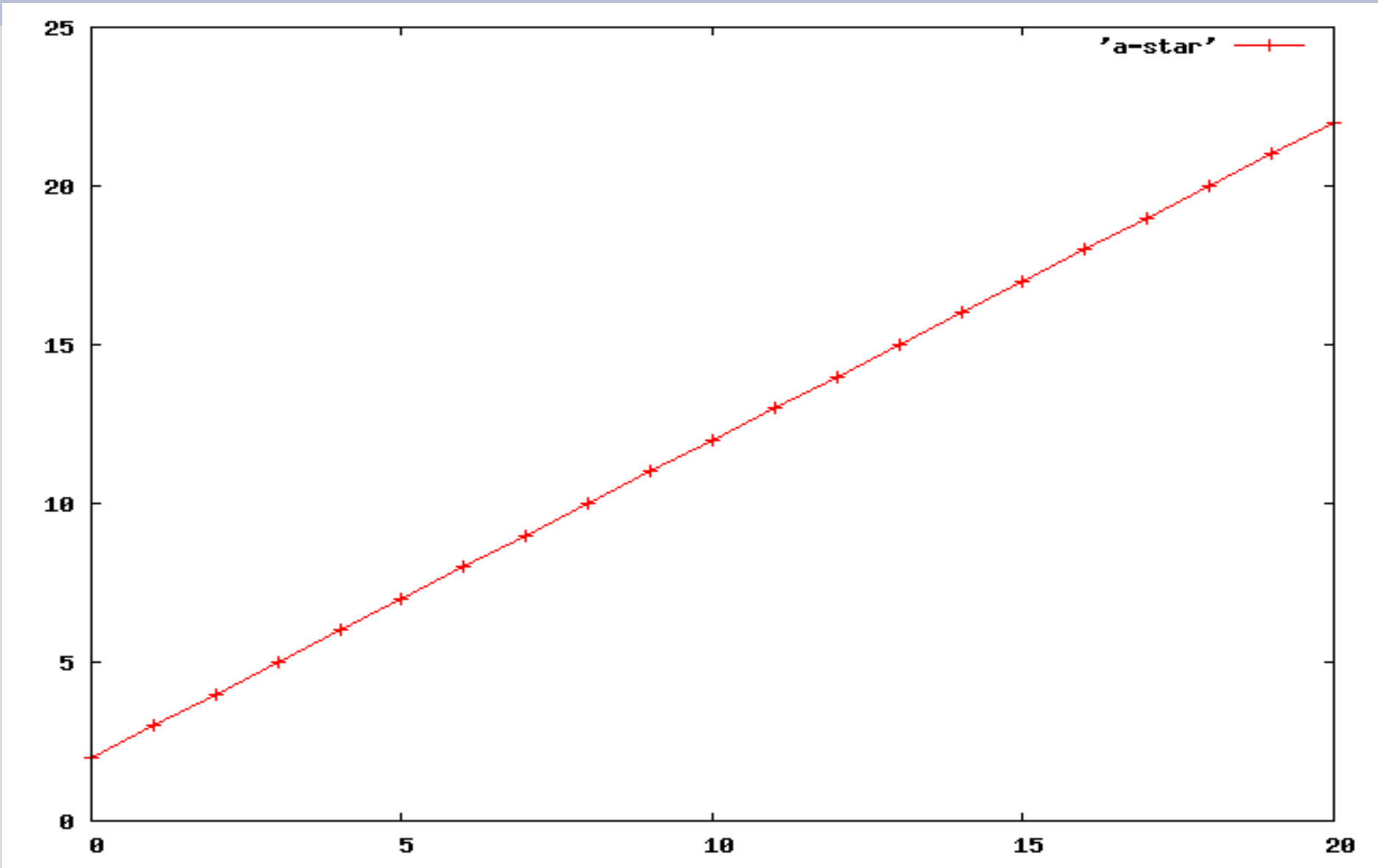
2 4

3 5

...

”#{n} #{count_try(...)}” と直接埋め込んでもいい

0から20までのグラフ



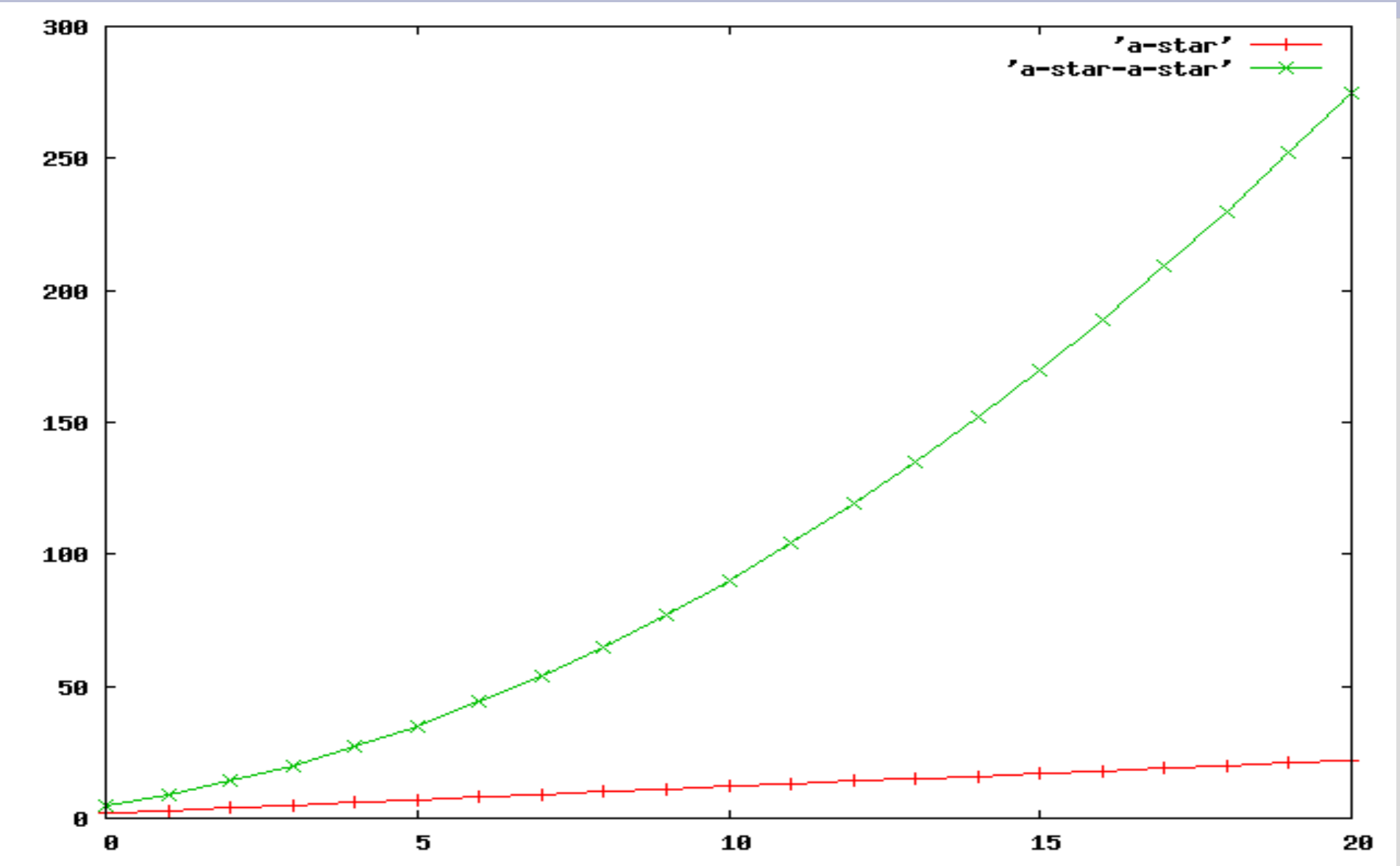
a*a* を a の並びにマッチ

- a* と a*a* はどちらも a の並びにマッチする

```
0.upto(20) {|n|
  m = count_try([:cat, [:rep, [:lit, "a"]], [:rep, [:lit, "a"]]], "a"*n)
  puts "#{n} #{m}"
}
```

```
=>    0 5     5 35    10 90    15 170   20 275
      1 9     6 44    11 104   16 189
      2 14    7 54    12 119   17 209
      3 20    8 65    13 135   18 230   実は (n+1)(n+6)/2 + 2
      4 27    9 77    14 152   19 252
```

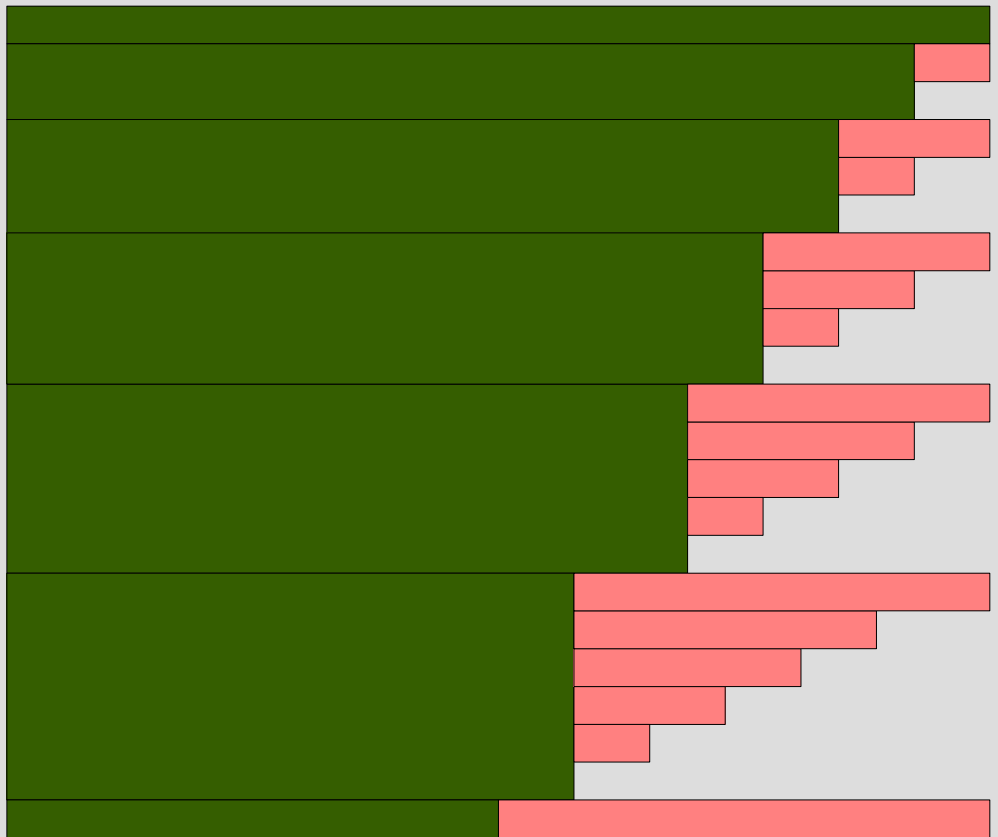
a*a* と a のグラフ



a*a*の効率が悪い理由

- 最初の a* と後の a* の境目が曖昧だから曖昧ないろんな可能性すべてを検査するのは時間がかかる

aaaaaaaaaaaaaaa

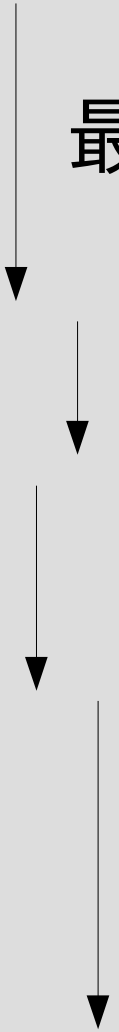


aaに対するマッチ

- 最初の a* が aa にマッチする
  - 後の a* が空文字列にマッチする
- 最初の a* が a にマッチする
  - 後の a* が a にマッチする
  - 後の a* が 空文字列にマッチする
- 最初の a* が空文字列にマッチする
  - 後の a* が aa にマッチする
  - 後の a* が a にマッチする
  - 後の a* が空文字列にマッチする

a*a* を aa にマッチしたときの try

```
pos   exp
0     [:cat, [:rep, [:lit, "a"]], [:rep, [:lit, "a"]]]
0     [:rep, [:lit, "a"]]
0     [:lit, "a"]
1     [:lit, "a"]
2     [:lit, "a"]
2     [:rep, [:lit, "a"]]
2     [:lit, "a"]
1     [:rep, [:lit, "a"]]
1     [:lit, "a"]
2     [:lit, "a"]
0     [:rep, [:lit, "a"]]
0     [:lit, "a"]
1     [:lit, "a"]
2     [:lit, "a"]
```



最上位の try 呼び出し

最初の a* を伸ばせるだけ伸ばす

後の a* を位置 2 から伸ばす

後の a* を位置 1 から伸ばす

後の a* を位置 0 から伸ばす

a*a* を a の並びにマッチ

- 最上位の try 呼び出しで 1
- 最初の a* を伸ばすのに 1+n+1=n+2
  - [:rep, [:lit, “a”]] で 1
  - 残り長さ n なのでマッチする [:lit, “a”] で n
  - マッチしない [:lit, “a”] で 1
- ここまでで n+3

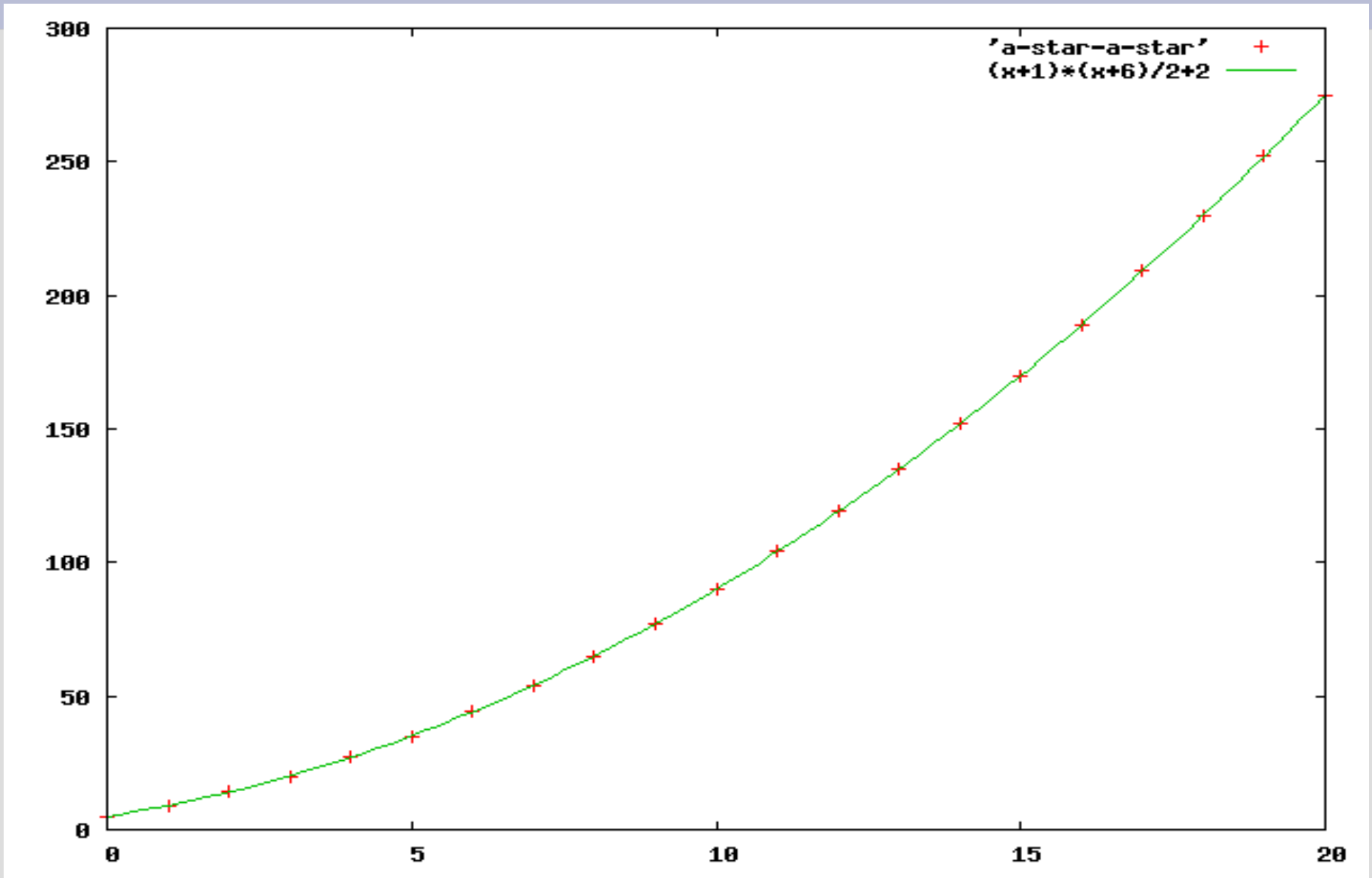
a*a* を a の並びにマッチ

- 前ページで n+3
- 後の a* で、2 + 3 + ... + (n+2)=(n+1)(n+4)/2
  - 位置2から伸ばすのに 2
    - [:rep, [:lit, “a”]] で 1
    - 残り長さ0なのでマッチする [:lit, “a”] は無し
    - マッチしない[:lit, “a”] で1
  - 位置1から伸ばすのに 3
    - [:rep, [:lit, “a”]] で 1
    - 残り長さ1なのでマッチする [:lit, “a”] で 1
    - マッチしない [:lit, “a”] で 1
  - ...
- 計 n+3+(n+1)(n+4)/2=(n+1)(n+6)/2+2

等差数列の和

- $1+2+\ldots+n = n(n+1)/2$

a*a*の実測値と理論値



(a*)* を a の並びにマッチ

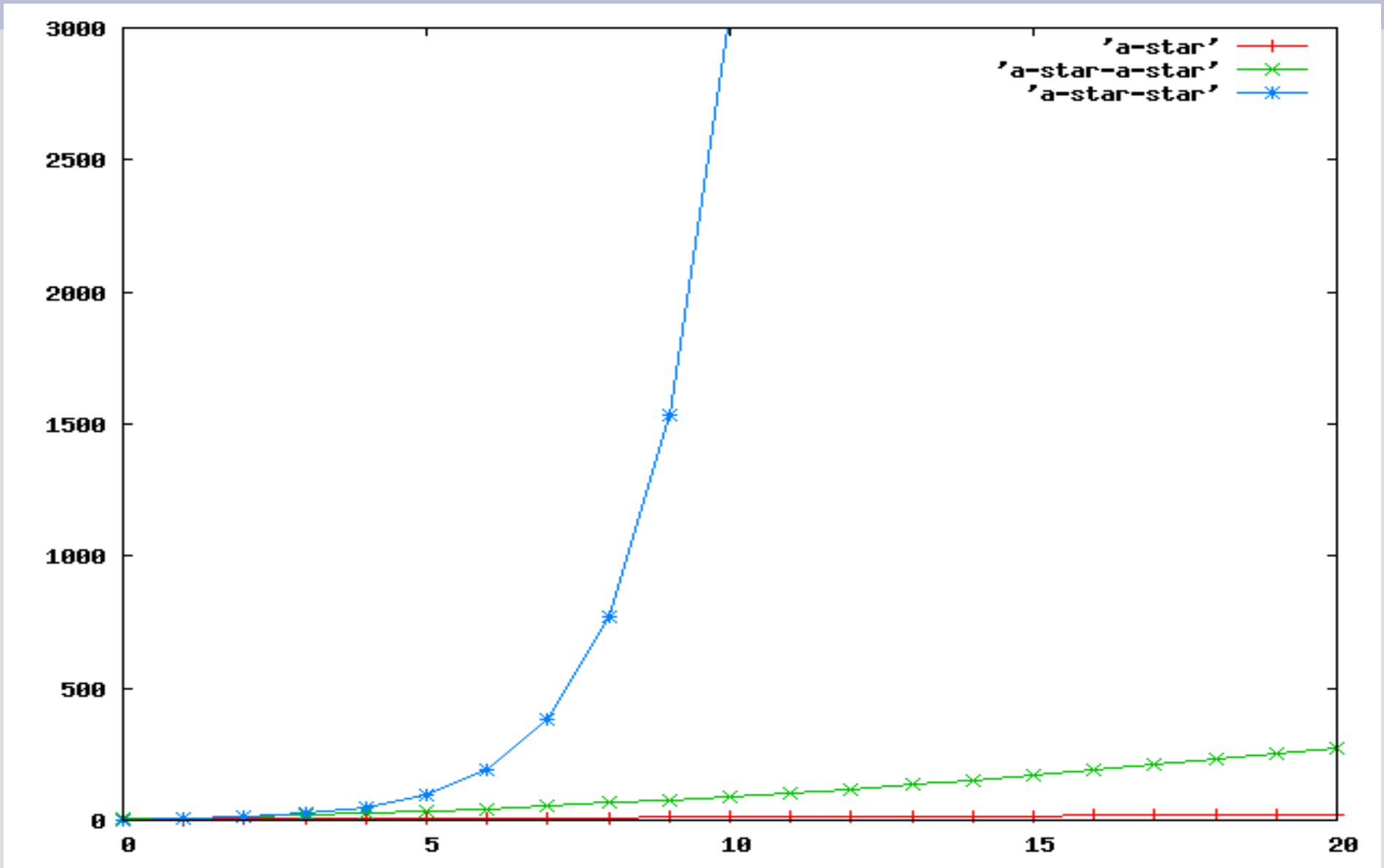
- (a*)* も aの並びにマッチするのは a*、a*a* と同じ
- (a*)* は a*a* よりさらに効率が悪い

```
0.upto(20) {|n|
  m = count_try([:rep, [:rep, [:lit, "a"]]], "a"*n)
  puts "#{n} #{m}"
}
```

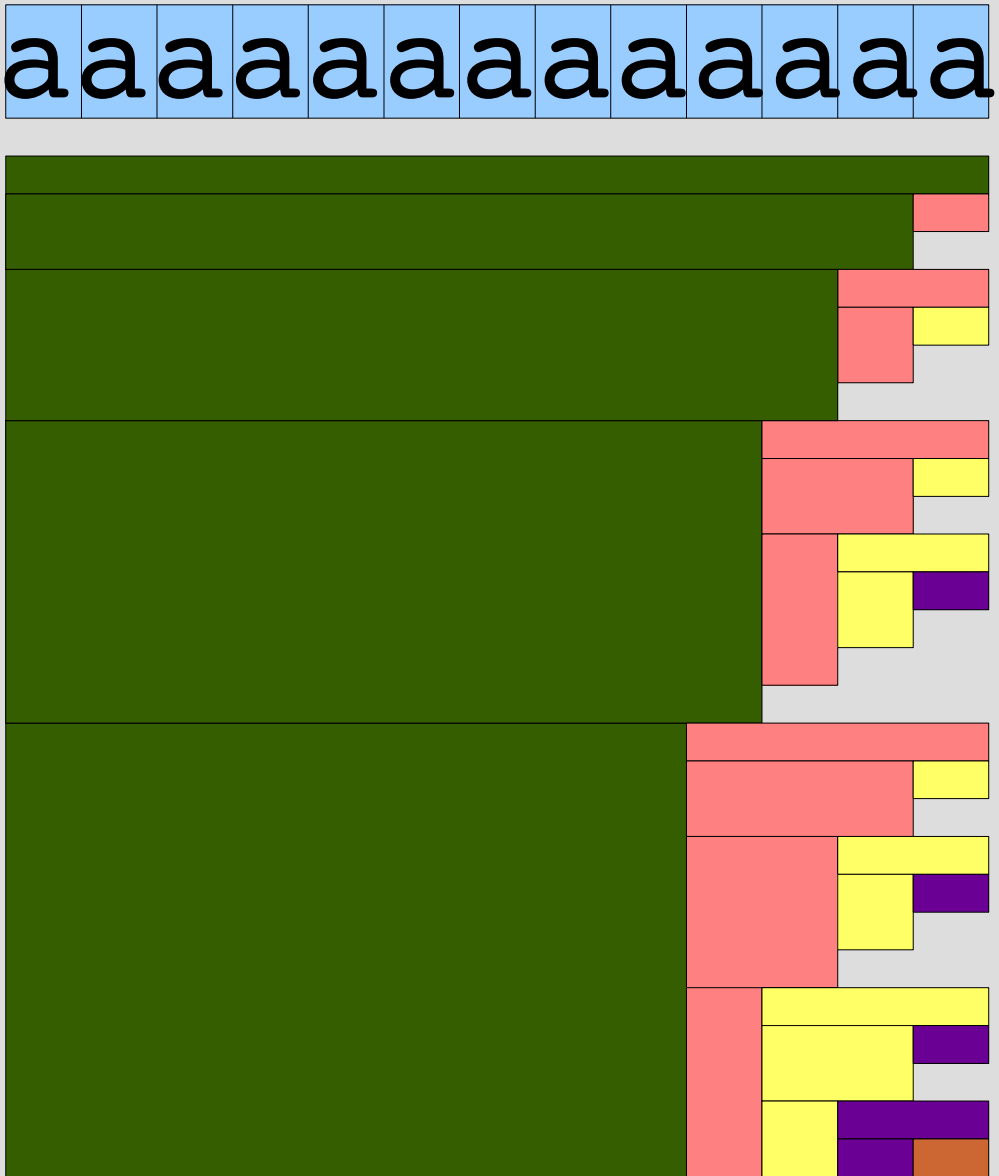
実は$3*2^n$

```
=>  0 3     4 48     8 768     12 12288
    1 6     5 96     9 1536    13 24576
    2 12    6 192    10 3072   14 49152
    3 24    7 384    11 6144   15 98304
```

(a*)*, a*a*, a* のグラフ


'a-star'
'a-star-a-star'
'a-star-star'
3000
2500
2000
1500
1000
500
0
0
5
10
15
20

(a*)*の効率が悪い理由



繰り返しの効率

- a* は n+2 回 try を呼び出す
- a*a* は(n+1)(n+6)/2 + 2 回 try を呼び出す
- (a*)* は $3*2^n$回 try を呼び出す
- aの並びという同じ対象にマッチするパターンでも、曖昧なものは遅くなる
- 繰り返しがネストしていると、とくに (指数関数的に) 遅い

レポート

- a が n個、b が n 個並んでいる文字列に /a*b*/ をマッチさせたときに try が呼び出される回数を n を使用した式として求めよ
- ある特定の n に対する回数は以下のようにして求められる

  count_try([:cat, [:rep, [:lit, “a”]], [:rep, [:lit, “b”]]],
  “a” * n + “b” * n)
- ∠切 2006-06-06 16:20
- IT's class
- 拡張子が txt なテキストファイル希望

まとめ

- 前回のレポートの解説
- try_repの解説
- 正規表現エンジンをちゃんと停止するようにした
- aの並びについて効率が悪いケースを述べた
- レポートを出した