テキスト処理 第8回 (2007-06-26)

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

- 補講について
- 試験の日程について
- 正規表現エンジン動作解説レポートの解説
- 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

補講について

- 7/24 (火) に講義 (補講) を行う
- 教務課掲示板･ポータルには 7/11 (水) に発表

- 今後の予定
  - 6/26 (火) 今日
  - 7/3 (火)
  - 7/10 (火)
  - 7/17 (火)
  - 7/24 (火) 補講
  - 7/31 (火) 試験

試験の日程

- 7/31 (火) 5時限 (16:30 〜 17:30)

- 講義時の教室とは違うかも?
- 試験時間割は 7/4 (水) に発表される

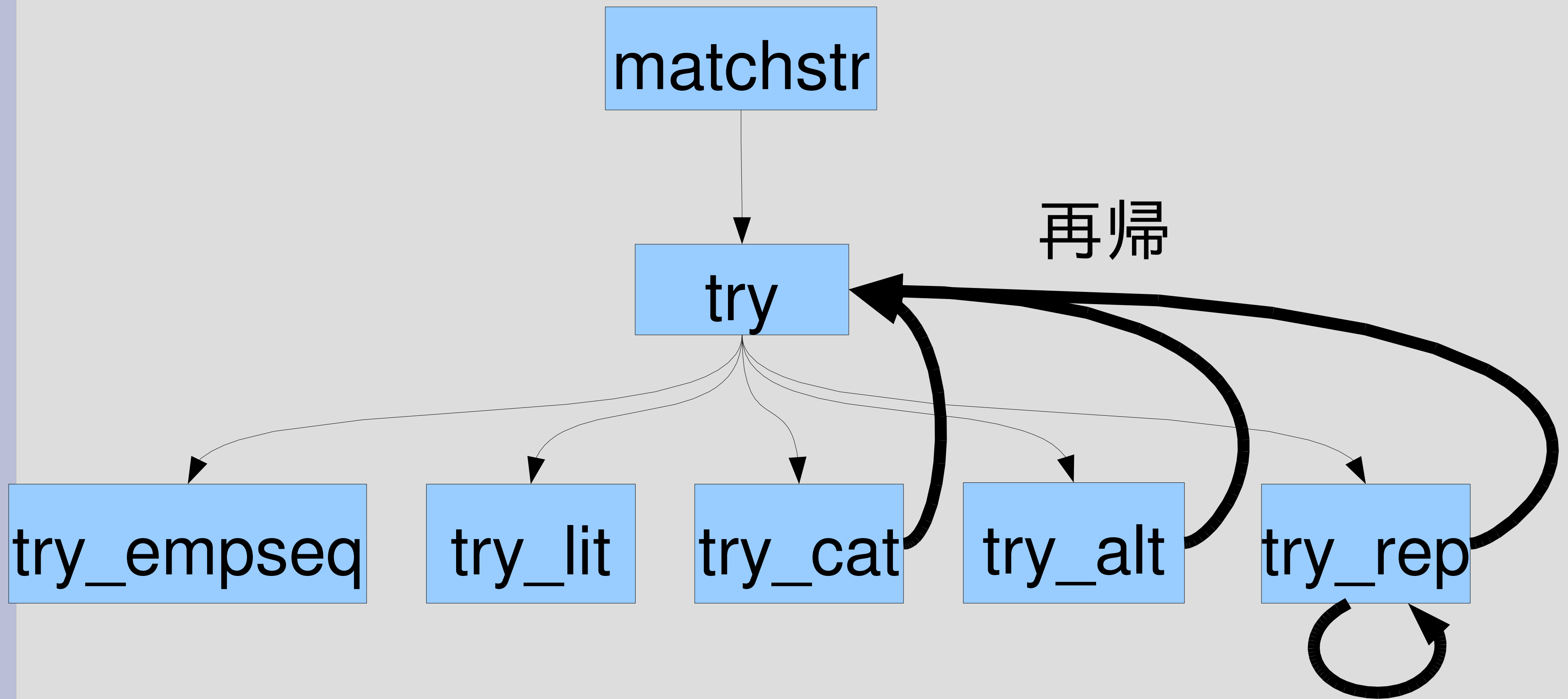
停止性

- 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]], "a")'
/tmp/rx.rb:15:in `try': stack level too deep (SystemStackError)
        from /tmp/rx.rb:56:in `try_rep'
        from /tmp/rx.rb:57:in `try_rep'
        from /tmp/rx.rb:35:in `try_empseq'
        from /tmp/rx.rb:16:in `try'
        from /tmp/rx.rb:56:in `try_rep'
        from /tmp/rx.rb:57:in `try_rep'
        from /tmp/rx.rb:35:in `try_empseq'
        from /tmp/rx.rb:16:in `try'
         ... 1420 levels...
        from /tmp/rx.rb:56:in `try_rep'
        from /tmp/rx.rb:28:in `try'
        from /tmp/rx.rb:7:in `matchstr'
        from -e:1
```

Segmentation Fault で
異常終了することもある

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

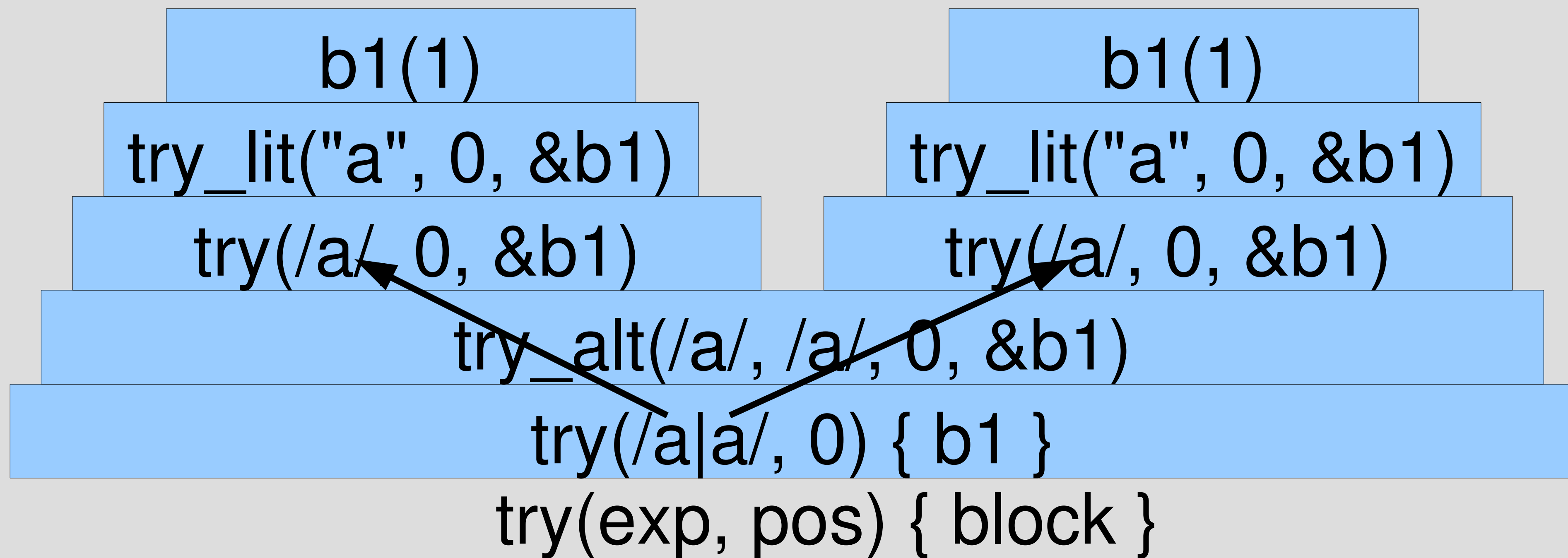
- 処理が進む:
  - 残りの文字列が短くなる
  - パターンが小さくなる
- 再帰するたびに処理が進めば、無限には再帰しない
  - 文字列の長さは有限
  - パターンの大きさは有限

try_alt

- [:alt, e1, e2] より e1 と e2 は小さい
(e1, e2 は [:alt, e1, e2] の一部分)
- 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

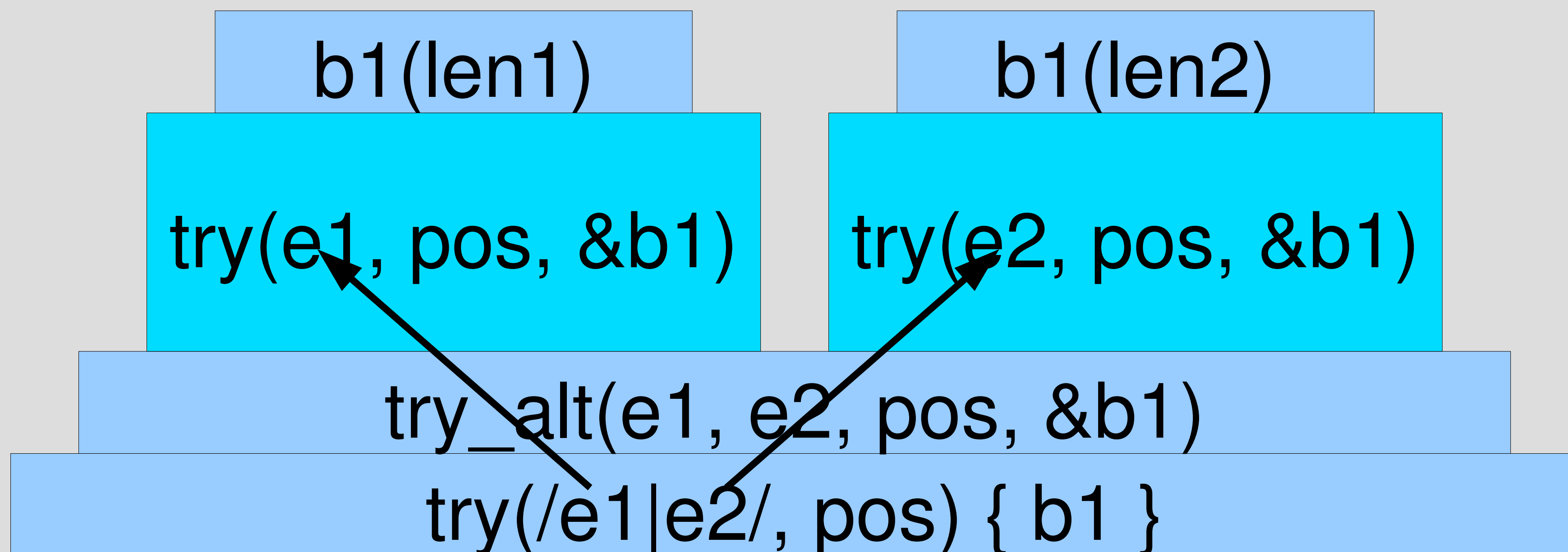
```
def try_alt(e1, e2, seq, pos, &block)
  try(e1, seq, pos, &block)
  try(e2, seq, pos, &block)
end
```

/a|a/ =~ "a"



try_alt を通るとパターンが小さくなる

/e1|e2/ =~ str



try_alt を通ると
パターンが小さくなる

len1 は e1 に
マッチした長さ

len2 は e2 に
マッチした長さ

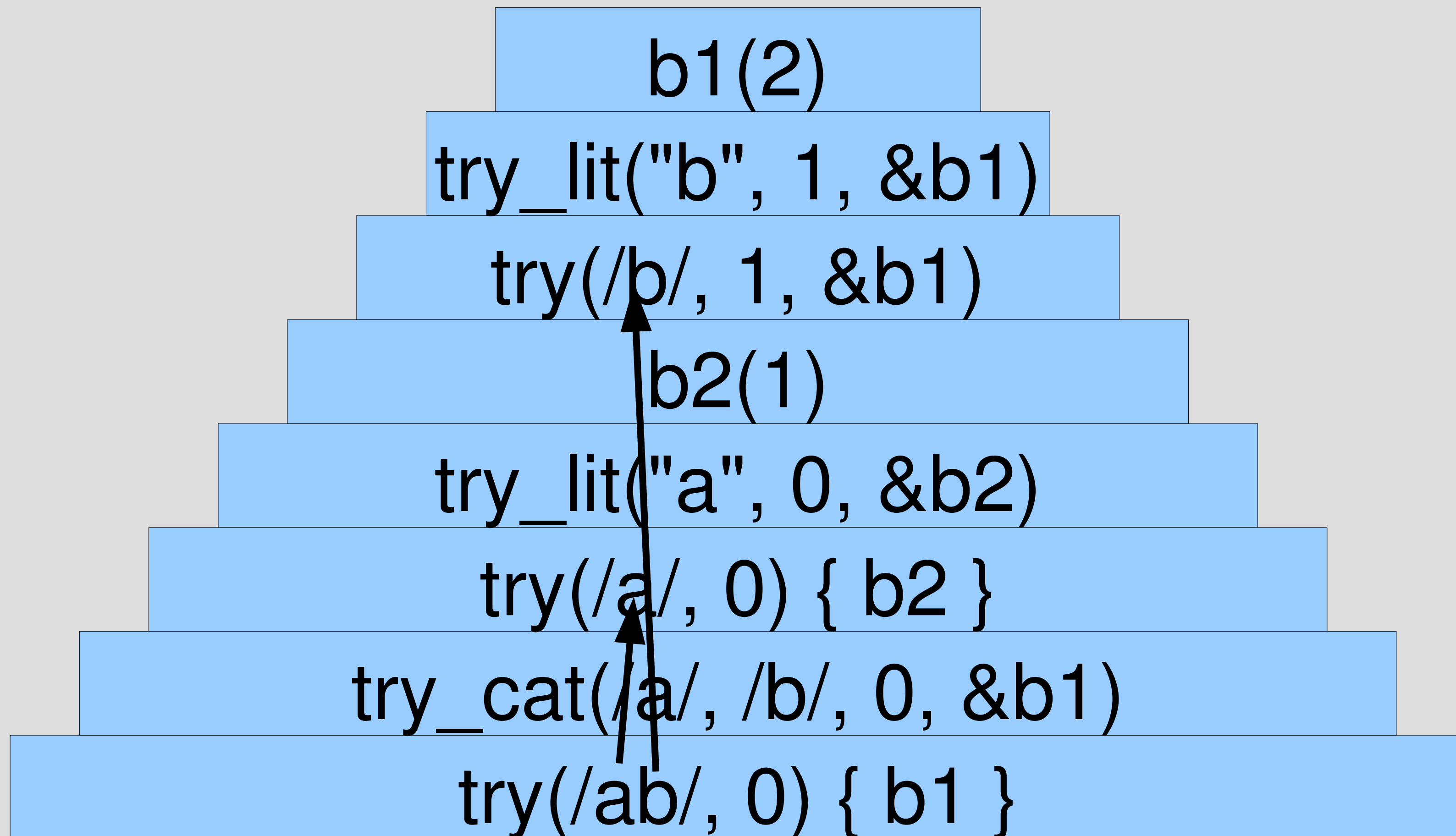
e1, e2 は
一回だけ
マッチする
ことを仮定

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"



try_cat を通るとパターンが小さくなる

/e1e2/ =~ str

b1(pos+len1+len2)

try(/e2/, pos+len1, &b1)

b2(pos+len1)

try(/e1/, pos) { b2 }

try_cat(/e1/, /e2/, pos, &b1)

try(/e1e2/, pos) { b1 }

len1 は e1 に
マッチした長さ

len2 は e2 に
マッチした長さ

e1, e2 が
一回だけ
マッチする
ことを仮定

try_cat を通るとパターンが小さくなる

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_lit("a", 2, &b4)

try(/a/, 2) { b4 }

b1(2)

try_rep(/a/, 2, &b1)

b3(2)

try_lit("a", 1, &b3)

try(/a/, 1) { b3 }

b1(1)

try_rep(/a/, 1, &b1)

b2(1)

try_lit("a", 0, &b2)

try(/a/, 0) { b2 }

b1(0)

try_rep(/a/, 0, &b1)

try(/a*/, 0) { b1 }

try_rep(/a/) が
何回も重なる

でも pos は
増えていくので
止まる

/a*/ =~ "aa"

- try_rep(/a/) はいくらでも重なる
- でも毎回 pos が進む
- ただし、pos を進めるのは try_lit("a")

/e*/ =~ str

try(/e/, p2) { b4 }

b1(p2)

try_rep(/e/, p2, &b1)

b3(p1+len2)

try(/e/, p1) { b3 }

b1(p1)

try_rep(/e/, pos+len1, &b1)

b2(pos+len1)

try(/e/, pos) { b2 }

b1(pos)

try_rep(/e/, pos, &b1)

try(/e*/, pos) { b1 }

p1 = pos+len1

p2 = p1 + len2

e は一回だけマッチすることを仮定

try_repは無限に再帰するかも

- 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
- 残りの文字列も小さくならない、かも
  - exp が空文字列にマッチして pos == pos2 になったとき
  - 例えば /()*/ とか
    抽象構文木だと [:rep, [:empseq]]

/()*/ =~ str

無限に続く

try(/()/, pos) { b4 }

b1(pos)

これらには
たどり着かない

try_rep(/()/, pos, &b1)

b3(pos)

try(/()/, pos) { b3 }

b1(pos)

try_rep(/()/, pos, &b1)

b2(pos)

try(/()/, pos) { b2 }

b1(pos)

try_rep(/()/, pos, &b1)

try(/()*/, pos) { b1 }

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

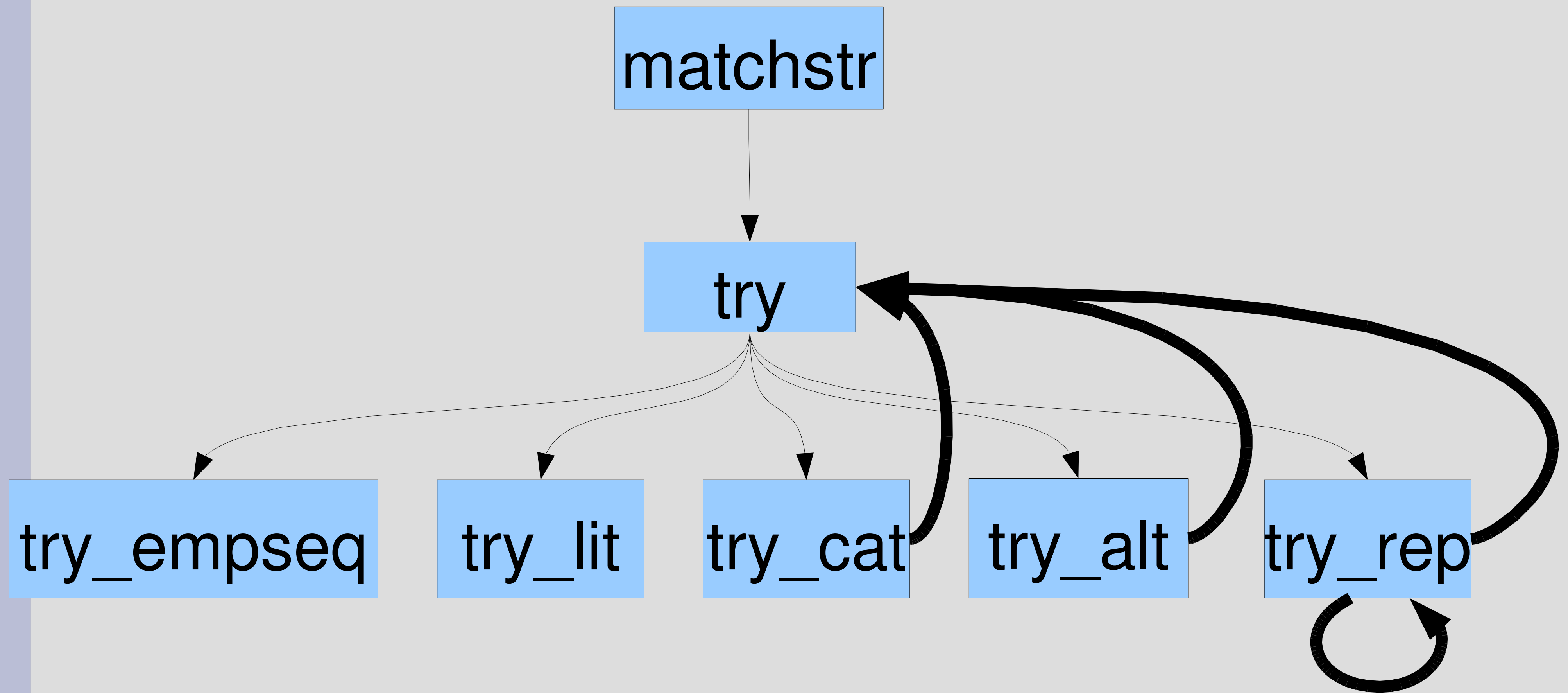
```
def try_rep(exp, seq, pos, &block)
  try(exp, seq, pos) {|pos2|
    try_rep(exp, seq, pos2, &block) if pos < pos2
  }
  yield pos
end
```

無視してもマッチするものはマッチする

/e*/ =~ 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

- 有限時間で終わることを保証できた
- では、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
- グローバル変数はなるべく避ける

try の呼び出しを数える

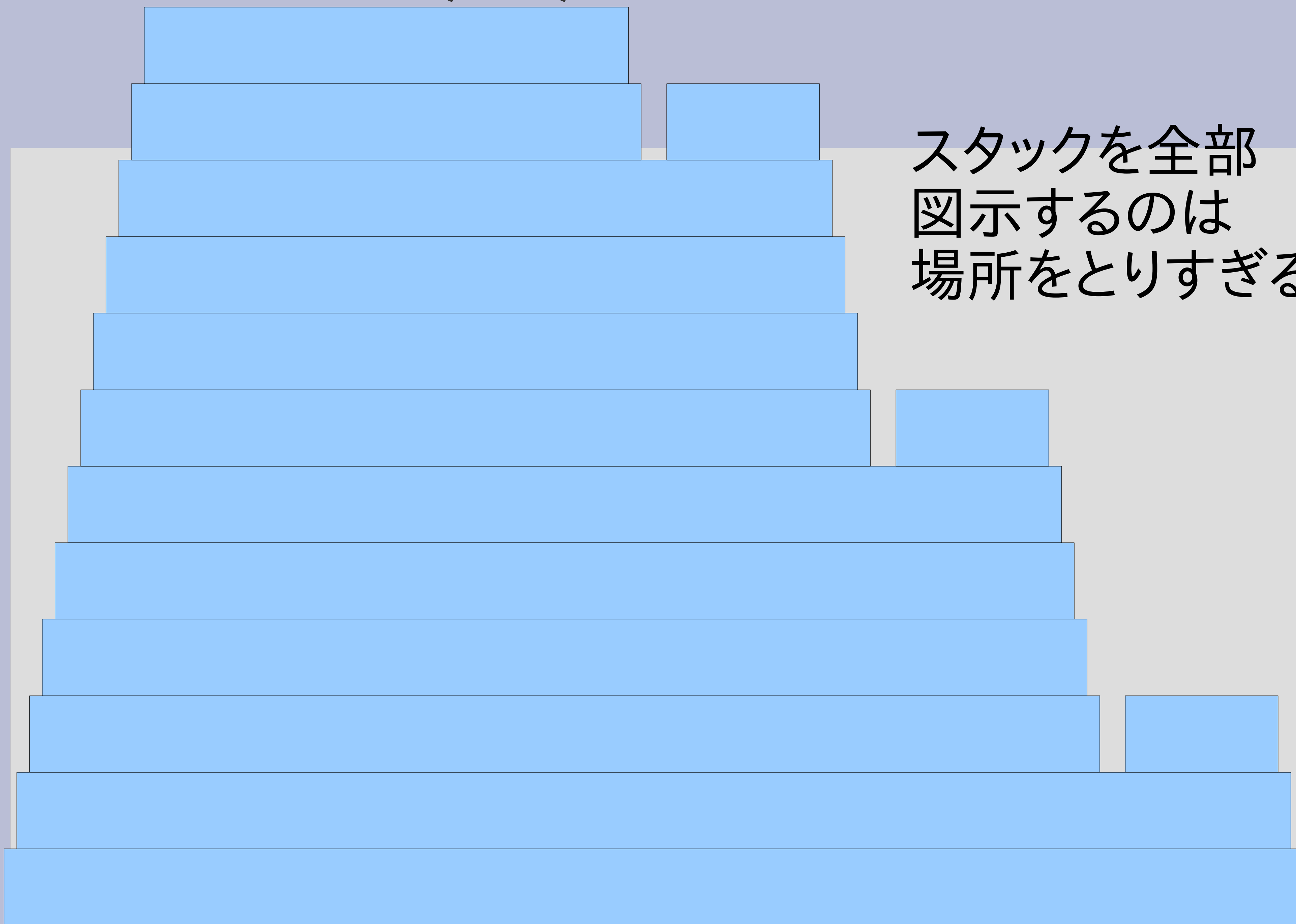
```
def count_try(exp, str)
  $try_count = 0
  matchstr(exp, str)
  $try_count
end

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

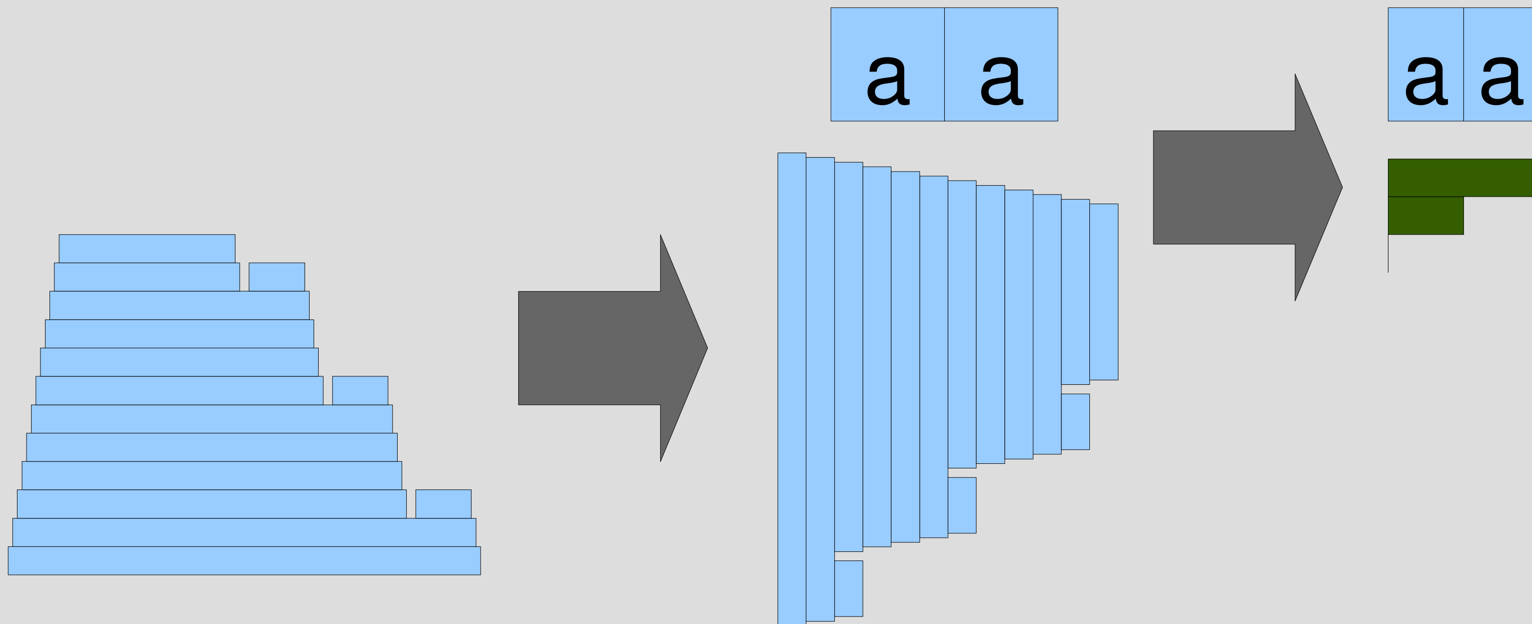
/a*/ =~ "aa"



スタックを全部
図示するのは
場所をとりすぎる

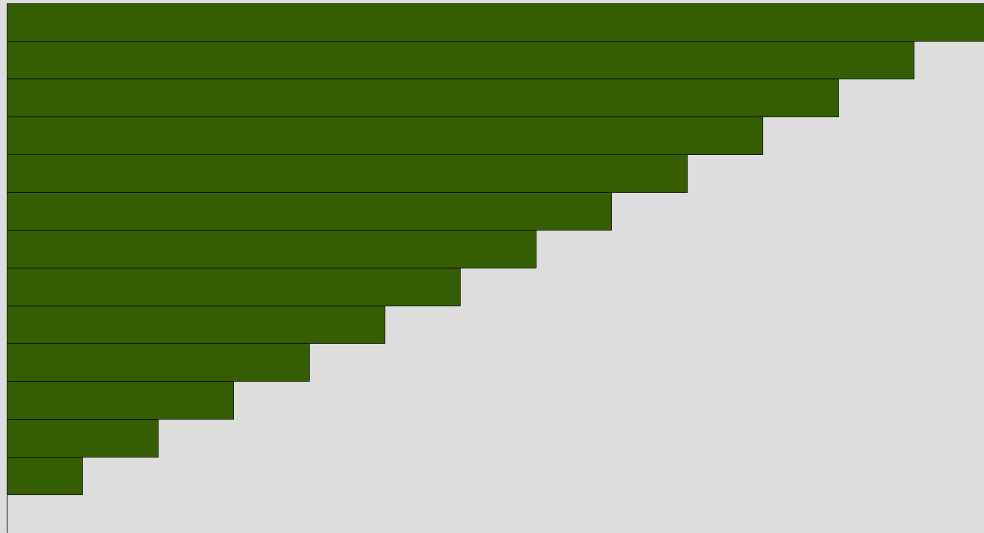
/a*/ =~ "aa"

- 右90°回転
- 文字を進めるところだけに注目してあとは省略



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



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

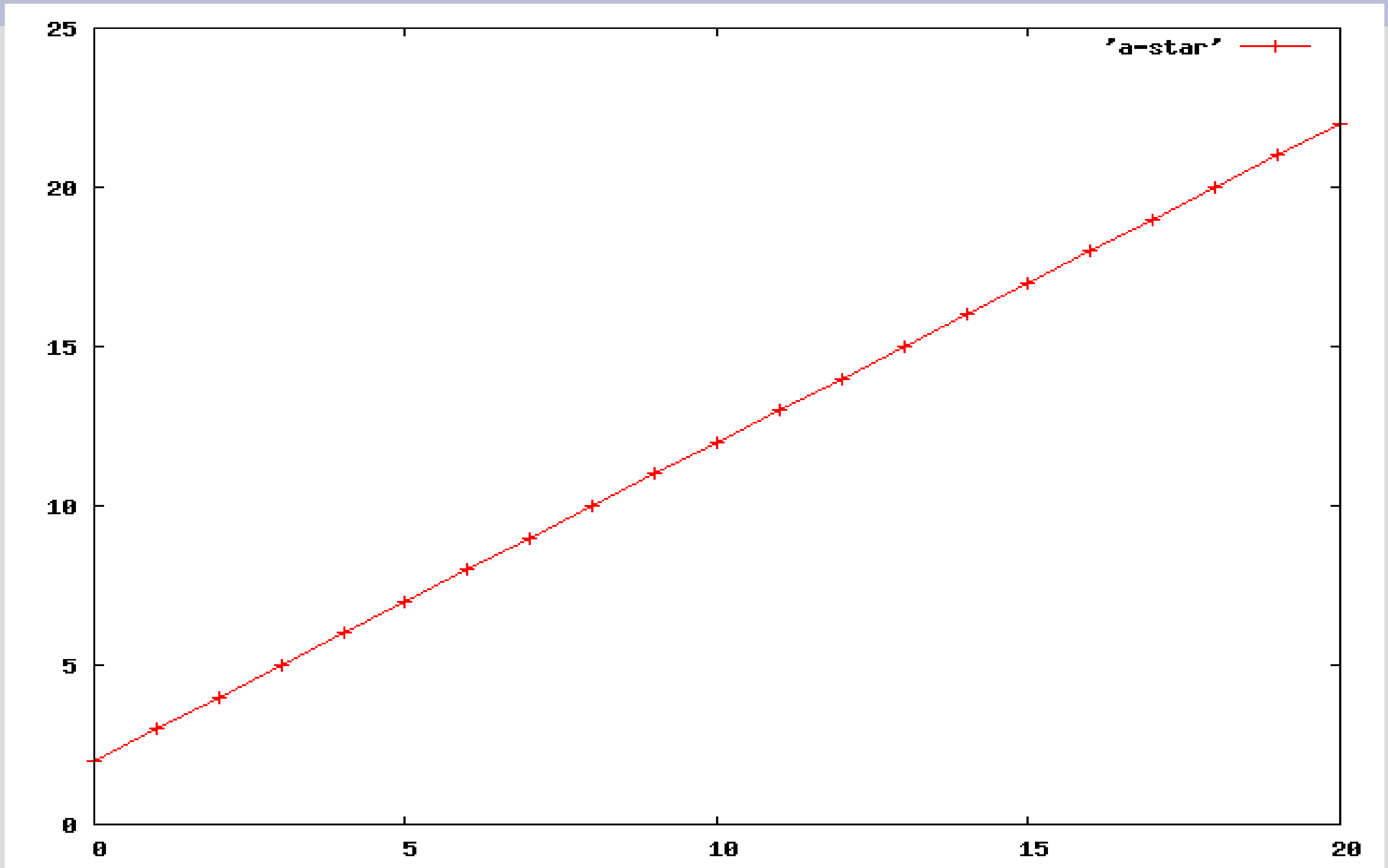
- "aaa#{式}bbb" というように、文字列の中に式を埋め込める
- 埋め込んだ式は毎回評価されて結果が文字列として埋め込まれる
- ダブルクオートの文字列に使える
- シングルクオートの文字列には使えない

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
直接埋め込んでもよい

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

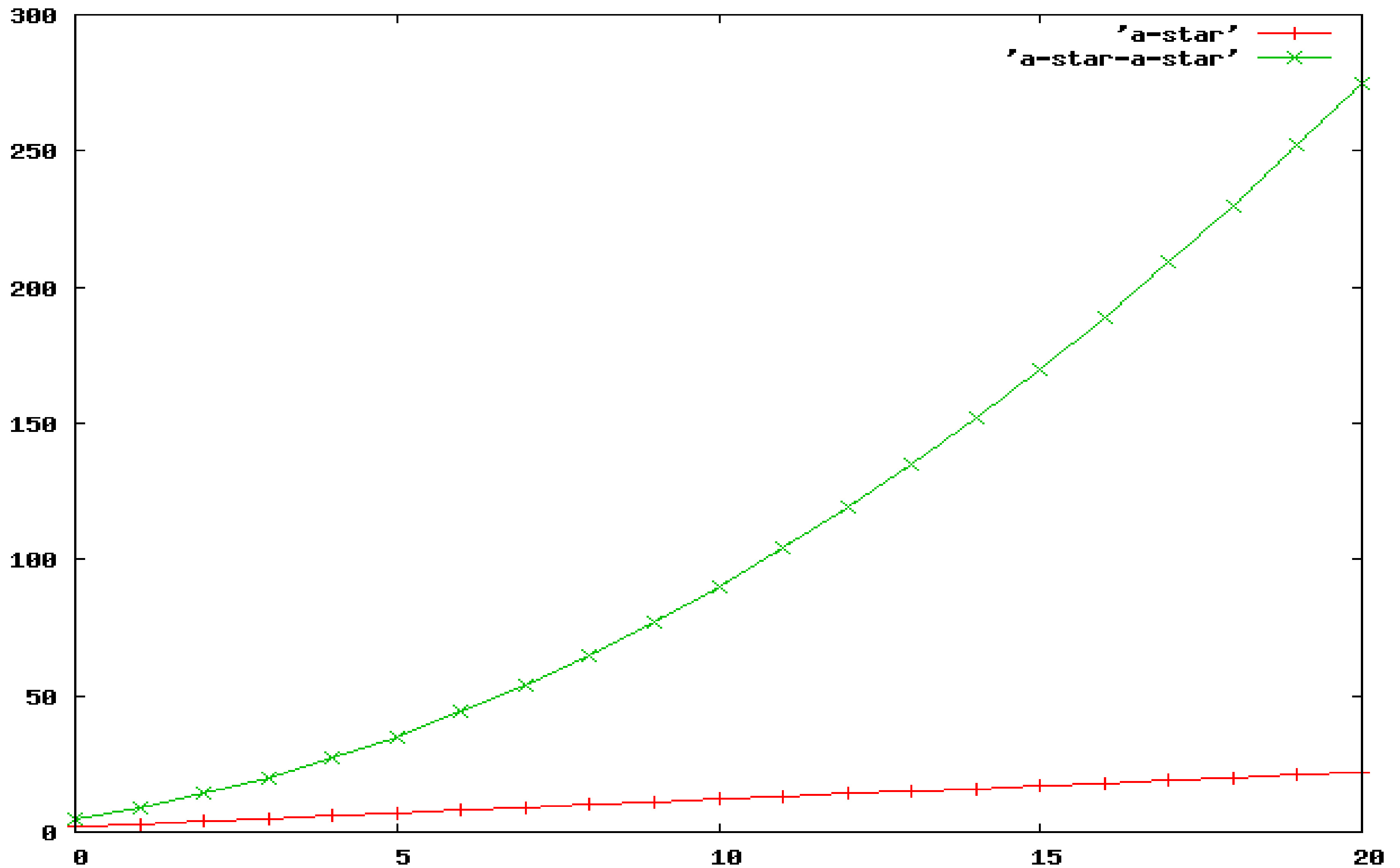
```
0.upto(20) {|n|
  m = count_try([:cat, [:rep, [:lit, "a"]],
                       [:rep, [:lit, "a"]]], "a"*n)
  puts "#{n} #{m}"
}
=>
```

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	
4 27	9 77	14 152	19 252	

実は

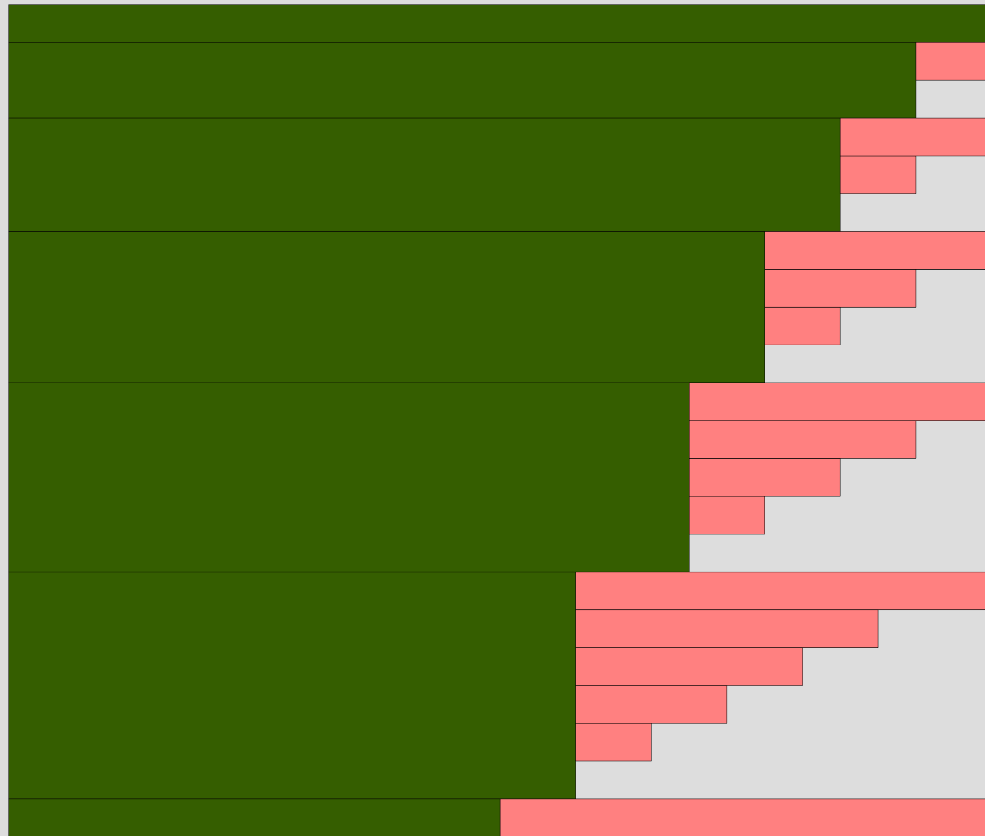
$(n+1)(n+6)/2 + 2$

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



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"]]]	最上位の try 呼び出し
0	[:rep, [:lit, "a"]]	最初の a* を伸ばせるだけ伸ばす
0	[:lit, "a"]	
1	[:lit, "a"]	
2	[:lit, "a"]	
2	[:rep, [:lit, "a"]]	後の a* を位置 2 から伸ばす
2	[:lit, "a"]	
1	[:rep, [:lit, "a"]]	後の a* を位置 1 から伸ばす
1	[:lit, "a"]	
2	[:lit, "a"]	
0	[:rep, [:lit, "a"]]	後の a* を位置 0 から伸ばす
0	[:lit, "a"]	
1	[:lit, "a"]	
2	[:lit, "a"]	

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

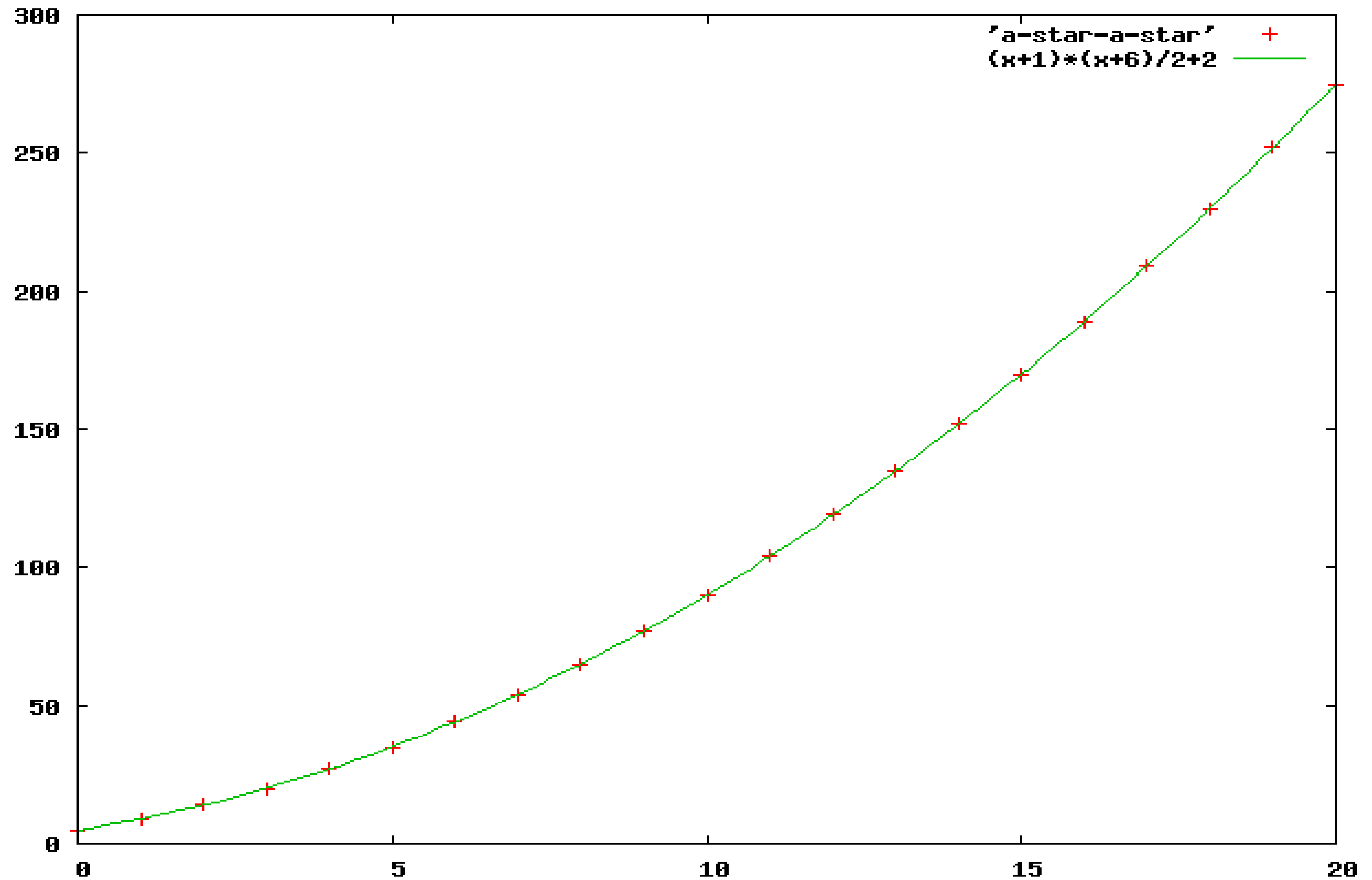
a*a* を a の並びにマッチ

- 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
  - 位置0から伸ばすのに 4
- 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

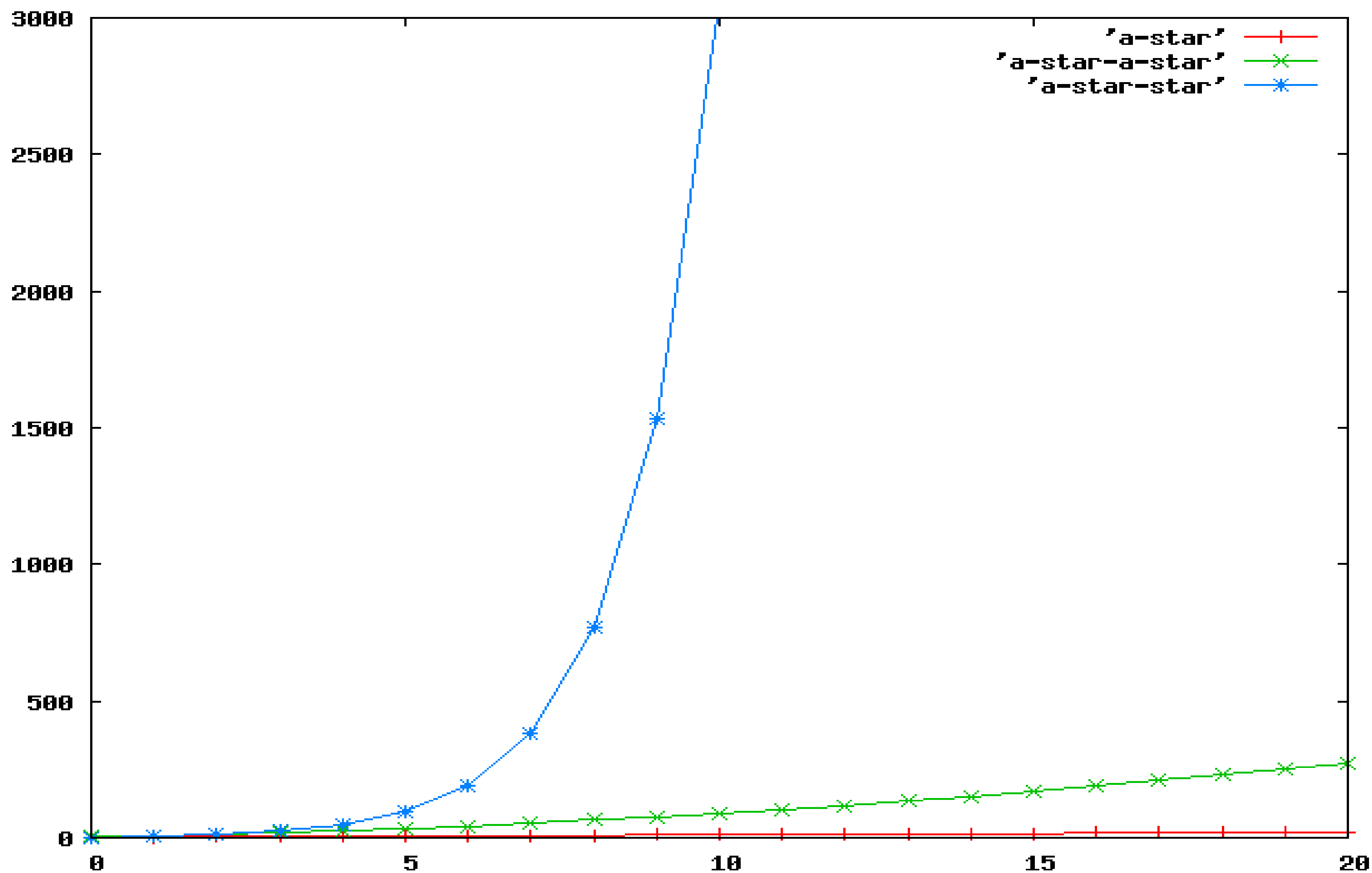
- (a*)* も aの並びにマッチするのは a*、a*a* と同じ
- (a*)* は a*a* よりさらに効率が悪い

```
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*)*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 個並んでいる文字列に /a*aaa/ をマッチさせたときに try が呼び出される回数を n に対する関数として求めよ
- 〆切 2007-07-03 16:20
- HIPLUS
- 拡張子が txt なテキストファイル希望

/a*aaa/ =~ "a" * n

- ある特定の n に対し、try の呼び出し回数は以下で求められる
- count_try(
  [:cat, [:rep, [:lit, "a"]],
  [:cat, [:lit, "a"],
  [:cat, [:lit, "a"], [:lit, "a"]]]],
  "a" * n)

```
count_try(
  [:cat, [:rep, [:lit, "a"]],
         [:cat, [:lit, "a"],
                [:cat, [:lit, "a"], [:lit, "a"]]]],
  "a" * n)
```

- 注意: /a*(a(aa))/ という構造とする

想定されるレポートの内容

- 求めた式
- その式が求まった理由
  - グラフから求めるのではなく、動作をたどって数えることを想定

ヒント

- わからなければ try の先頭に p exp とか p [pos, exp] とか入れて動作をたどる

まとめ

- 前回のレポートの解説
- 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