テキスト処理 第10回 (2006-06-27)

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

- 前回のレポートの説明
- 正規表現エンジンを再度説明
- 再帰を使う拡張
  - 存在するかもしれない: e?
  - 1回以上の繰り返し: e+
  - 怠惰な繰り返し: e*?
  - 存在しないかもしれない: e??
- レポート

正規表現エンジン

- try(exp, seq, spos) {leposl ... }
- seq の spos 番目から exp へのマッチを探す
- マッチごとに最後の位置を epos として yield する
- 再帰
  - exp, spos を変えながら try を再帰
  - (seq は変わらない。単に渡していくだけ)

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
```

分岐してるだけ

try_lit

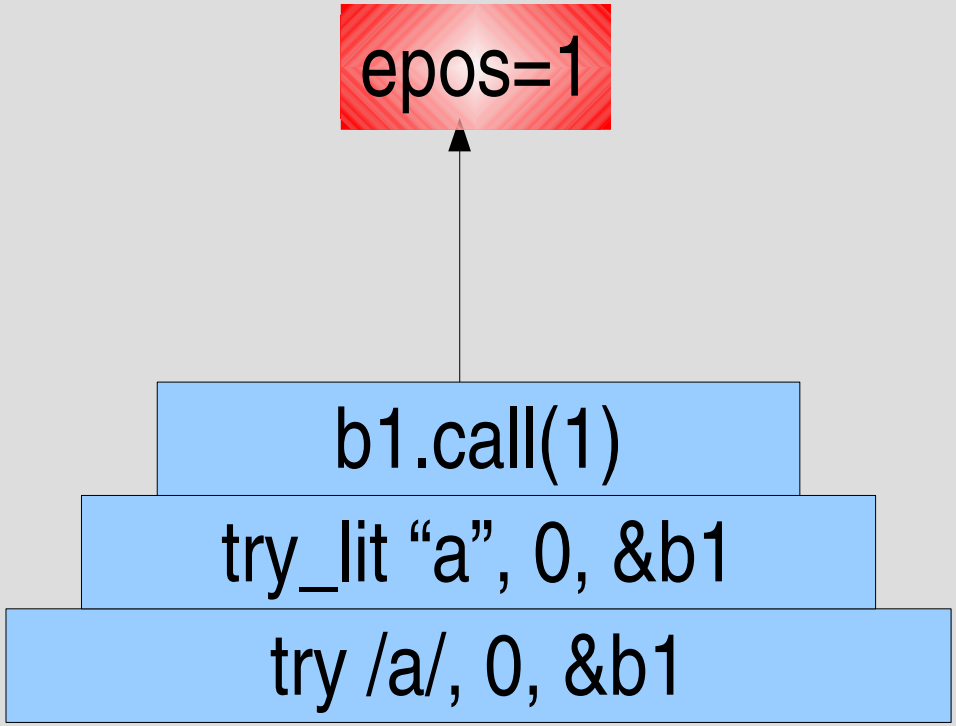
- 一文字進められれば、進んだ所を yield

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

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

/a/ =~ “abcd”

抽象構文木は長いので
普通の正規表現にしてある
seq も省略してある



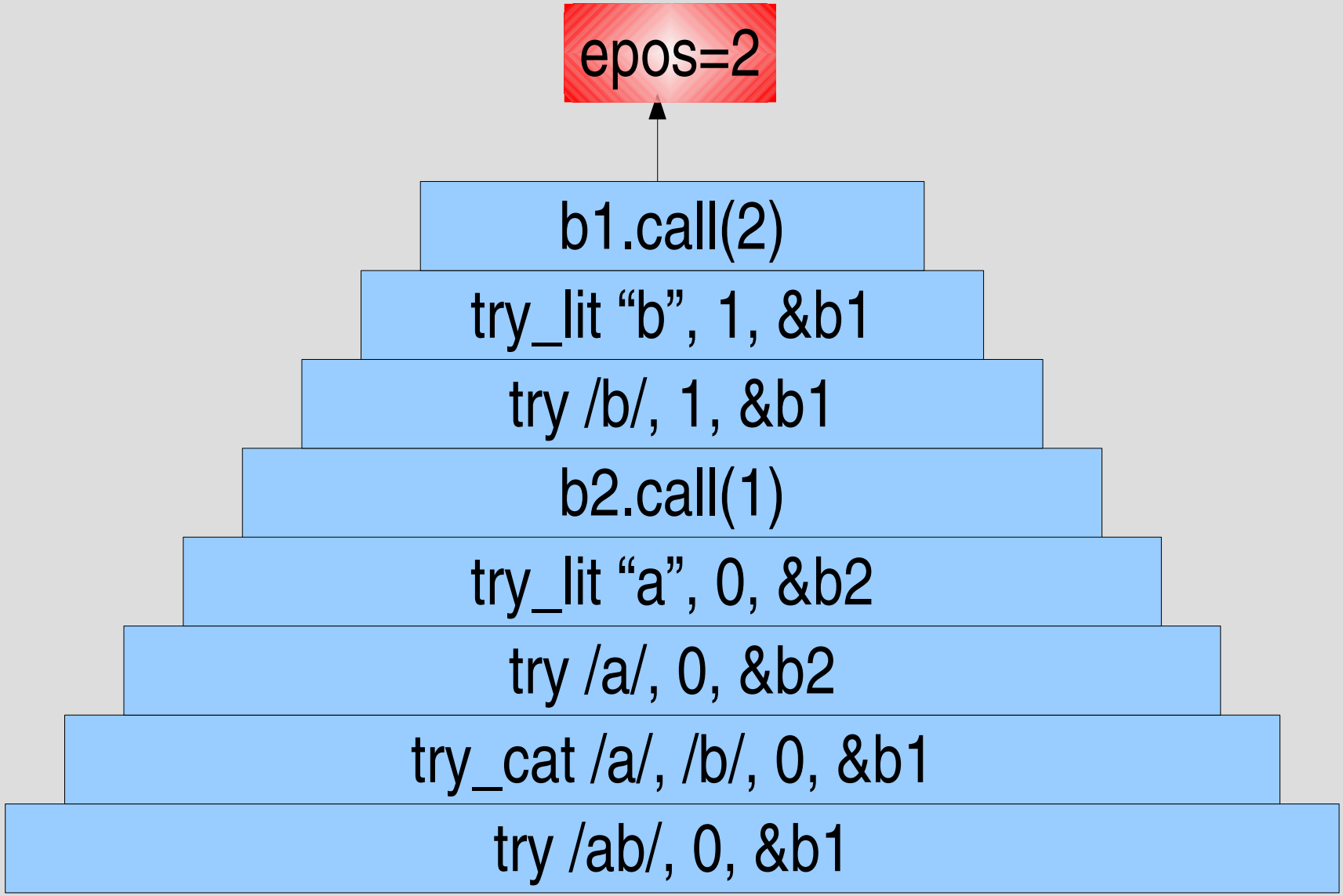
try_cat

- 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”]], [“a”, “b”], 0) {|posl p pos} # 2
```

/ab/ =~ “abcd”


epos=2
b1.call(2)
try_lit “b”, 1, &b1
try /b/, 1, &b1
b2.call(1)
try_lit “a”, 0, &b2
try /a/, 0, &b2
try_cat /a/, /b/, 0, &b1
try /ab/, 0, &b1

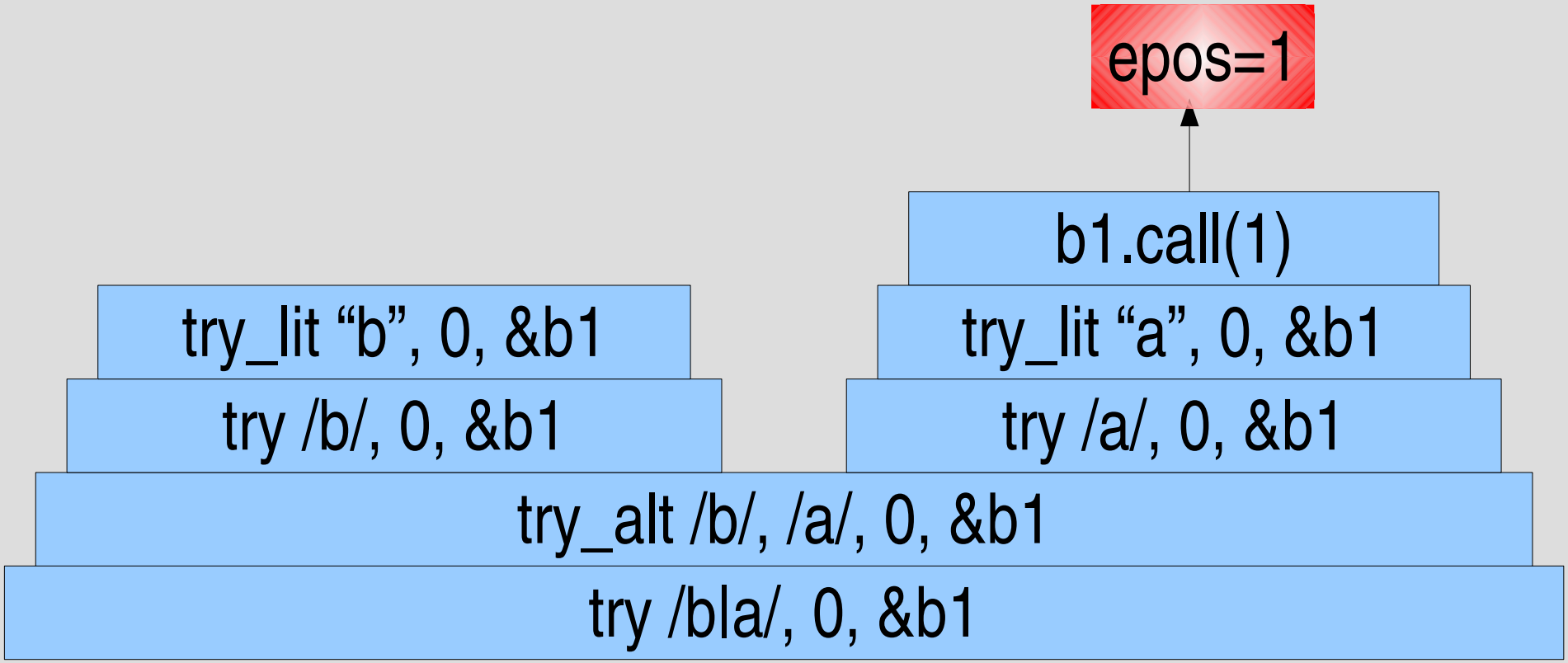
try_alt

- 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”]], [“a”, “b”], 0) {|posl p pos } # 1
```

/bla/ =~ “abcd”

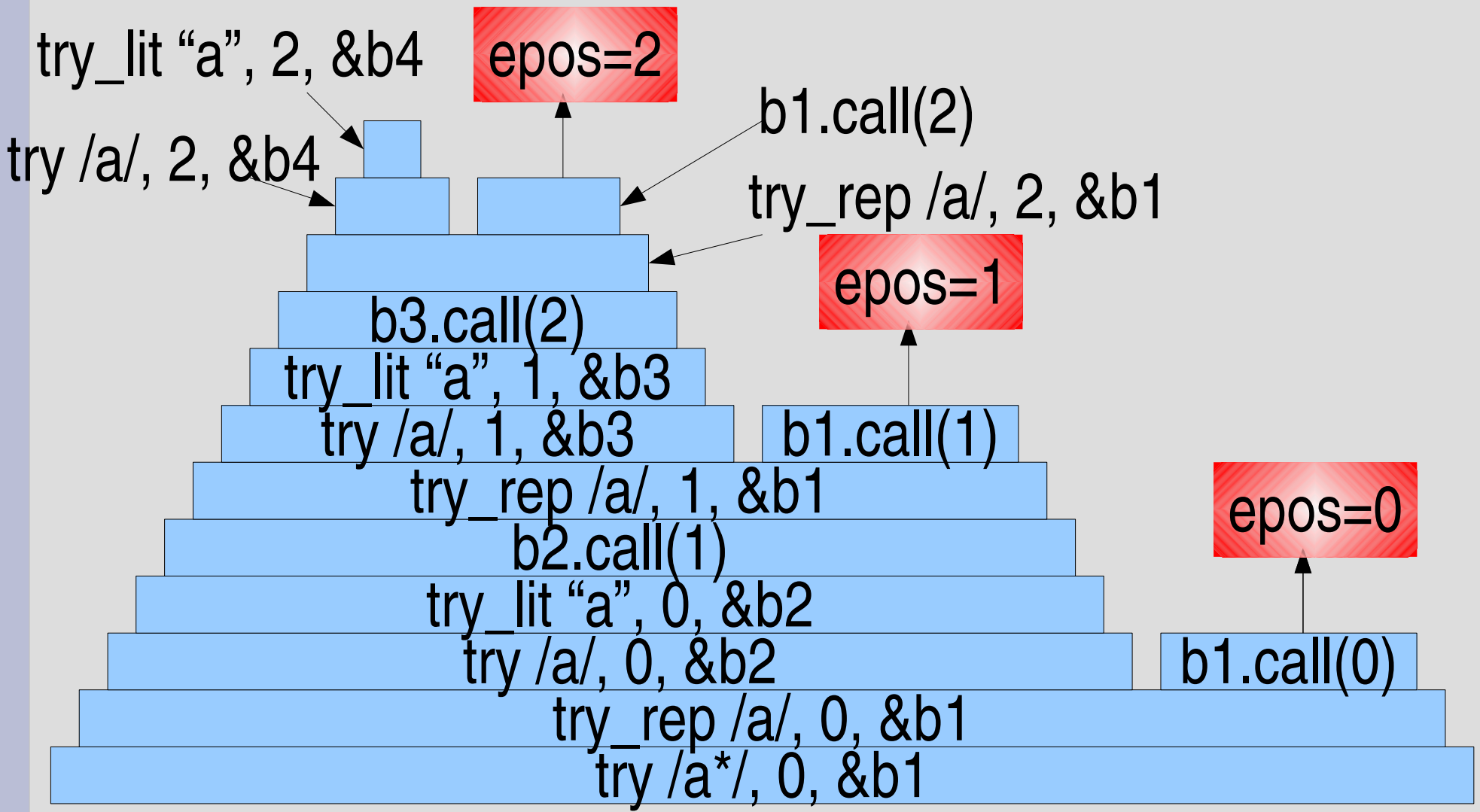

epos=1
b1.call(1)
try_lit “b”, 0, &b1
try_lit “a”, 0, &b1
try /b/, 0, &b1
try /a/, 0, &b1
try_alt /b/, /a/, 0, &b1
try /bla/, 0, &b1

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

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

/a*/ =~ “aa”


try_lit “a”, 2, &b4
epos=2
try /a/, 2, &b4
b1.call(2)
try_rep /a/, 2, &b1
epos=1
b3.call(2)
try_lit “a”, 1, &b3
try /a/, 1, &b3
b1.call(1)
try_rep /a/, 1, &b1
epos=0
b2.call(1)
try_lit “a”, 0, &b2
try /a/, 0, &b2
b1.call(0)
try_rep /a/, 0, &b1
try /a*/, 0, &b1

エンジンの拡張: /e?/

- /e?/ は、e がある場合とない場合にマッチする
- e がある場合を先にためし、ない場合を後に試す
- /el/ と同じ
- [:opt, e] で表現する (optional の意)

- /behaviou?r/ =~ “behavior”　　#=> 0
- /behaviou?r/ =~ “behaviour”　　#=> 0

- matchstr([:opt, [:lit, "a"]], "a")　　#=> [1,0]
- matchstr([:opt, [:lit, "a"]], "b")　　#=> [0]

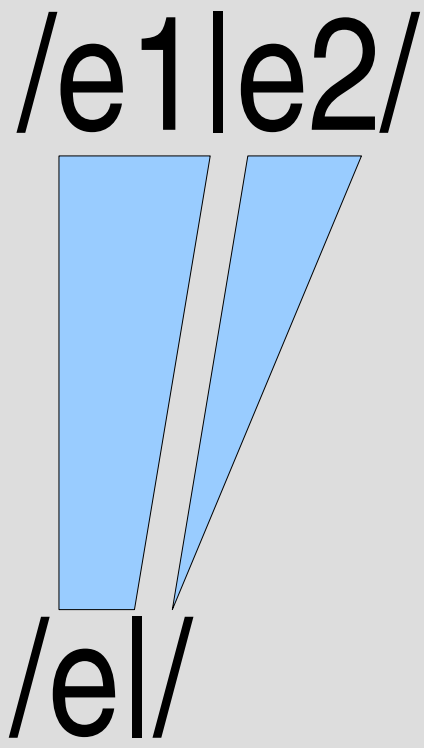
[:opt, e] の実装

```
def try(exp, seq, pos, &block)
  ...
  when :opt
    _, e = exp
    try_opt(e, seq, pos, &block)
  ...
end
```

```
def try_opt(e, seq, pos, &block)
  try(e, seq, pos, &block)
  yield pos
end
```

try_alt と try_opt の比較

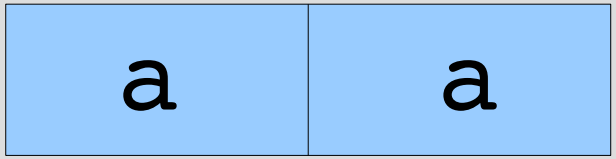
```
def try_alt(e1, e2, seq, pos, &block)
  try(e1, seq, pos, &block)
  try(e2, seq, pos, &block)
end
```

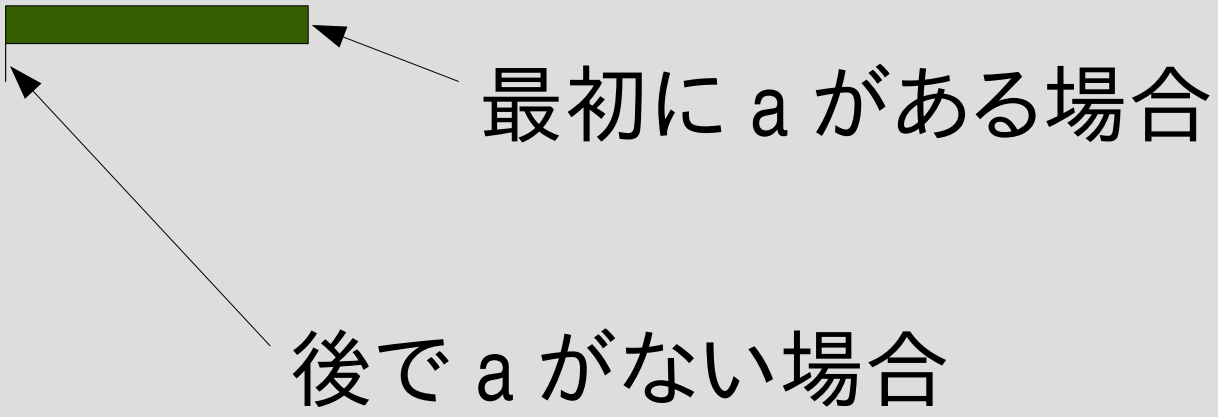



```
def try_opt(e, seq, pos, &block)
  try(e, seq, pos, &block)
  yield pos
end
```

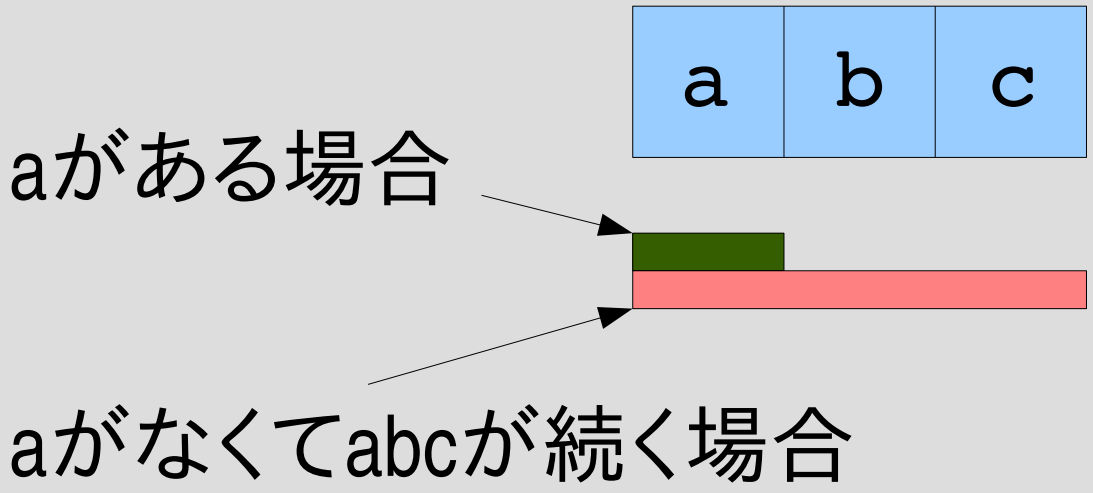
try(//) を展開した形になっている

matchstr([:opt, [:lit, “a”]], “aa”)

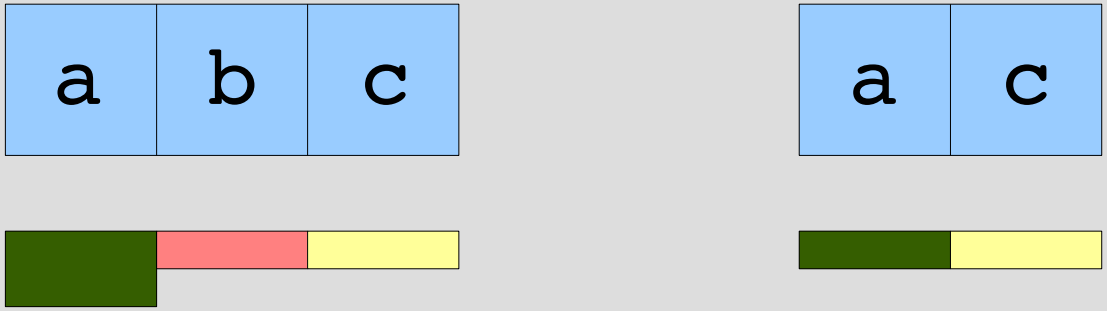




/a?abc/ =~ “abc”



/ab?c/ =~ “abc”, /ab?c/ =~ “ac”



エンジンの拡張: /e+/

- /e+/ は、e の 1つ以上の繰り返し
- [:plus, e] で表現する
- /ee*/ と同じ

- /ab+c/ =~ “ac”　　#=> nil
- /ab+c/ =~ “abc”　　#=> 0
- /ab+c/ =~ “abbbc”　　#=> 0

- matchstr([:plus, [:lit, "a"]], "aaa")　#=> [3,2,1]
- matchstr([:rep, [:lit, "a"]], "aaa")　#=> [3,2,1,0]

[:plus, e] の実装

```
def try(exp, seq, pos, &block)
  ...
  when :plus
    _, e = exp
    try_plus(e, seq, pos, &block)
  ...
end
```

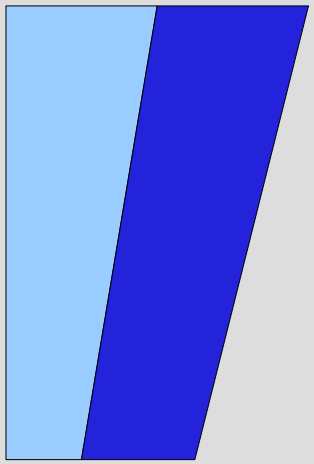
```
def try_plus(e, seq, pos, &block)
  try(e, seq, pos) {|pos2|
    try_rep(e, seq, pos2, &block)
  }
end
```

try_cat と try_plus の比較

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

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

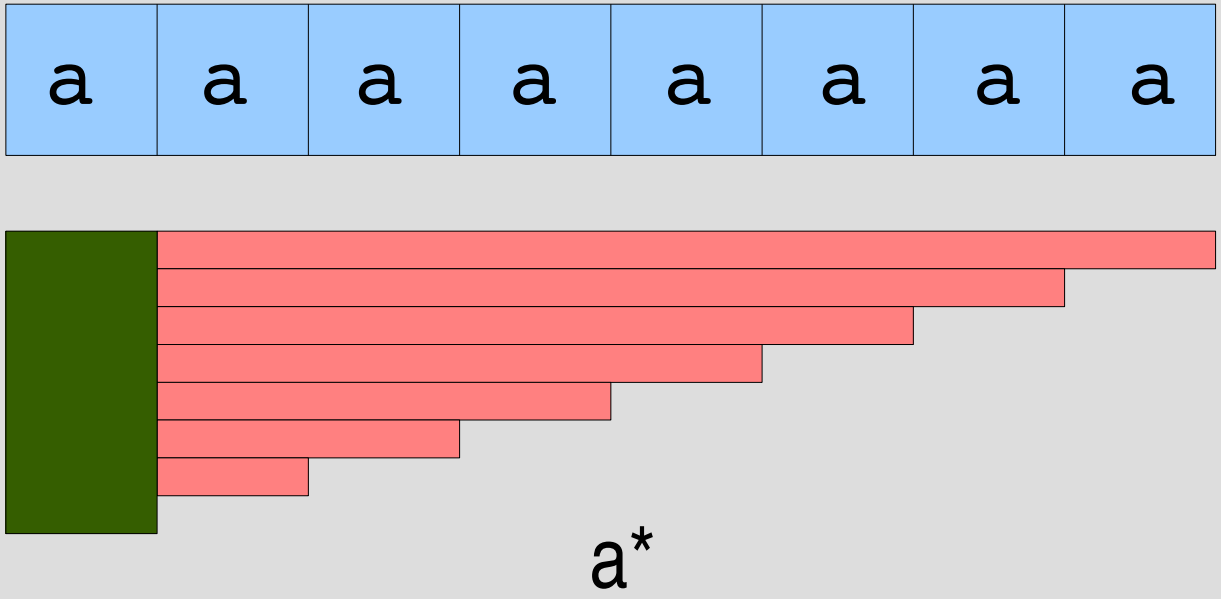
/e1e2/



/ee*/

try(/e*/) を展開した
構造になっている

a+ の動作



エンジンの拡張: /e*?/

- /e*?/ は、e の 0個以上の繰り返し
- [:rep_lazy, e] で表現する
- e* とは逆に、少ない繰り返しから試す
- いままでの組合せでは表現できない

- matchstr([:rep_lazy, [:lit, "a"]], "aaa") #=> [0,1,2,3]
- matchstr([:rep, [:lit, "a"]], "aaa") #=> [3,2,1,0]

e*? と (e*)? の違い

- e*? は [:rep_lazy, e]
- (e*)? は [:opt, [:rep, e]]
- *? はひとつの機能で、* と ? の組合せではない

lazy

- a* は繰り返しが多い場合から続きを試す
  この順序を greedy (貪欲) という
  とりあえずたくさん食べてみる、というイメージ

- a*? は繰り返しが少ない場合から続きを試す
  この順序を lazy (怠惰) もしくは nongreedy (非貪欲) という
  なるべくなら食べないで済ます、というイメージ

e*? の用途

- C のコメントを取り出す
  - /*.**\// =~ “ab /* ccc */ de /* xxx */”
  - /*.*?*\// =~ “ab /* ccc */ de /* xxx */”
- HTML のタグの対を取り出すのにも使われる
  - /.*<\/b>/ =~ “aabbbcccdddee”
  - /.*?<\/b>/ =~ “aabbbcccdddee”
- HTML のはあまり正しいやりかたではない
  - ネストしていたらうまくいかない
  - /.*?<\/b>/ =~ “aabbbcccdddee”
  - 閉じタグがないとうまくいかない
  - /.*?<\/b>/ =~ “
bbbccc
dddee”

[:rep_lazy, e] の実装 (1)

```
def try(exp, seq, pos, &block)
  ...
  when :rep_lazy
    _, e = exp
    try_rep_lazy(e, seq, pos, &block)
  ...
end
```

[:rep_lazy, e] の実装 (2)

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

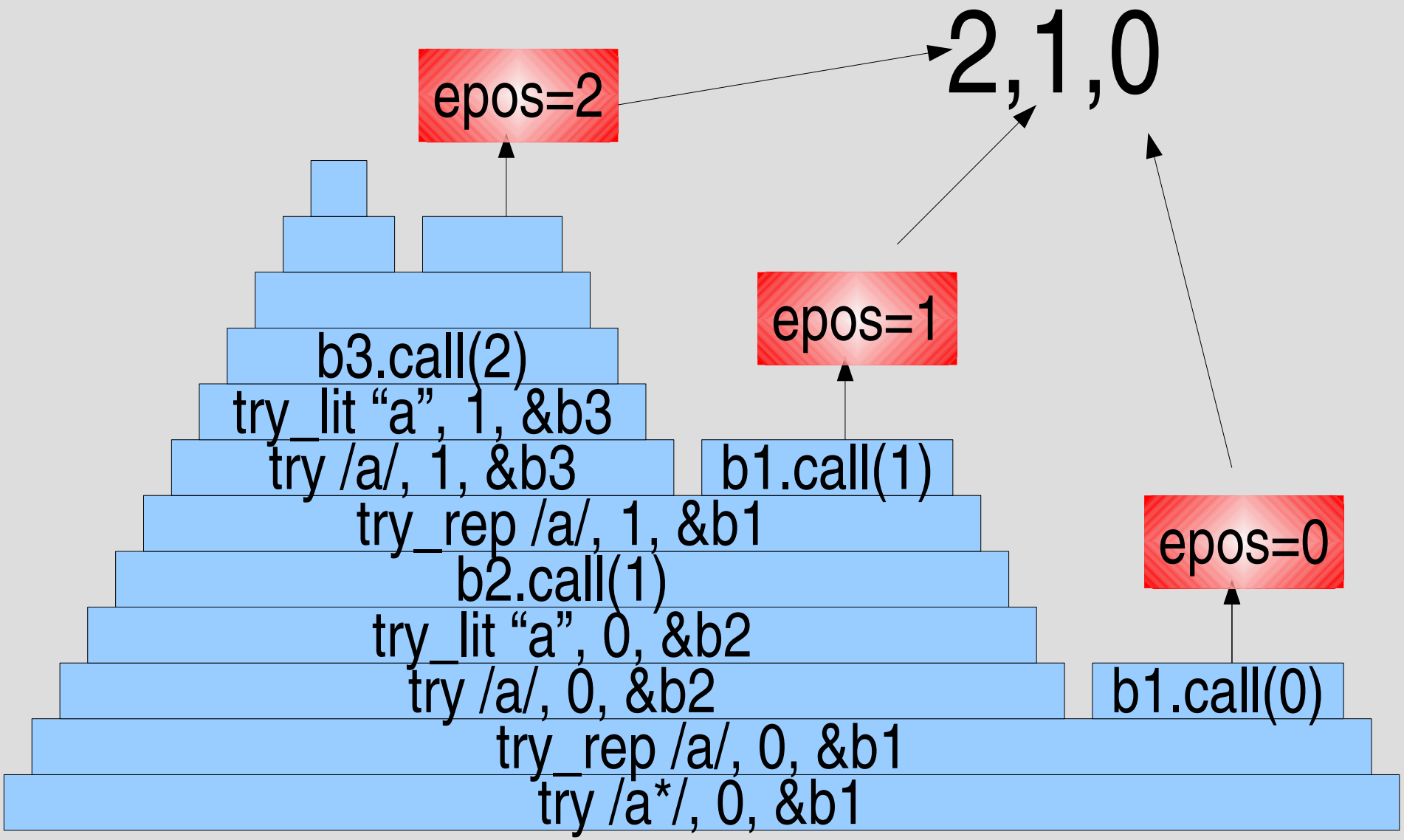
rep と rep_lazy

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

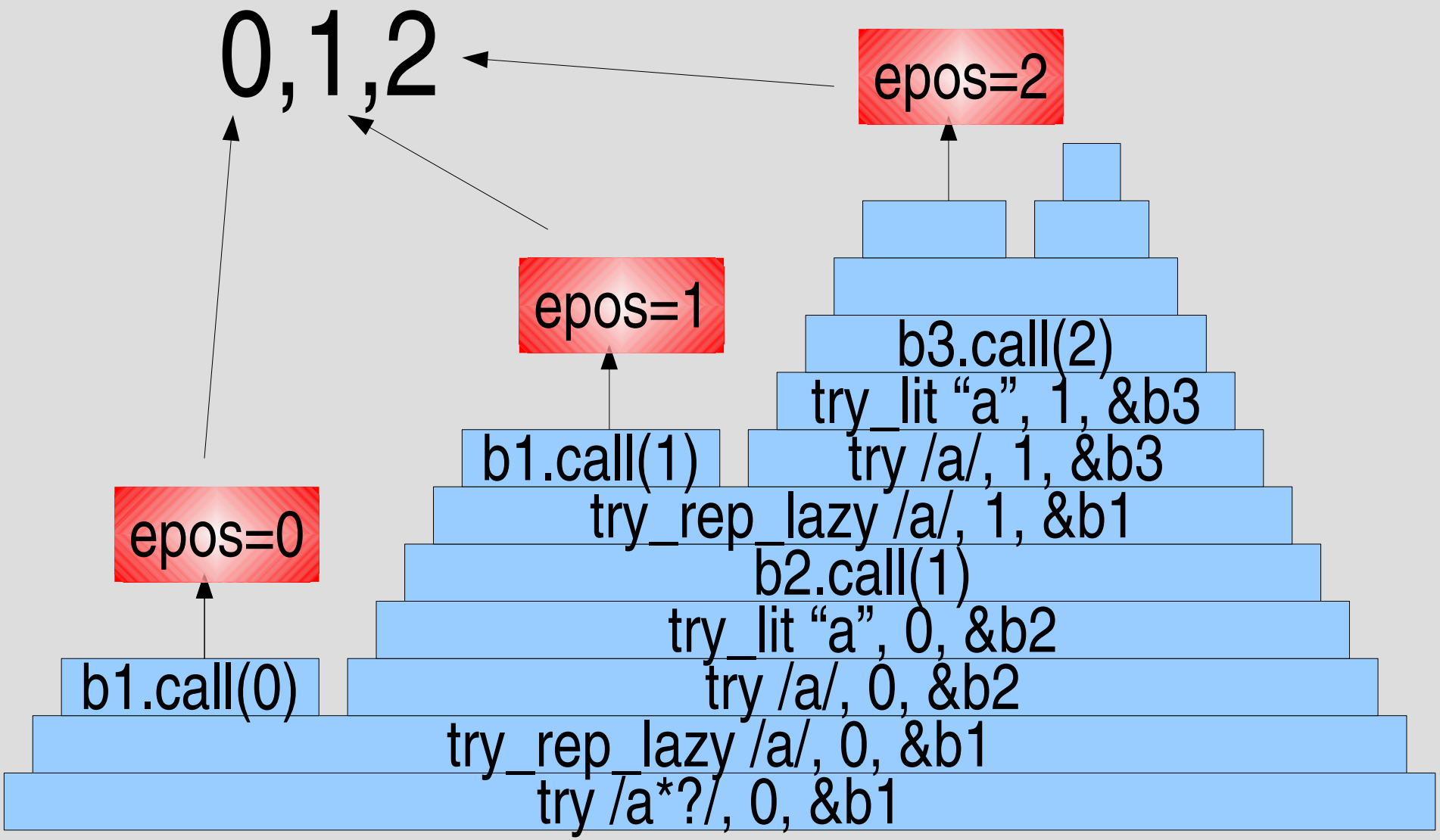
後に yield

先に yield

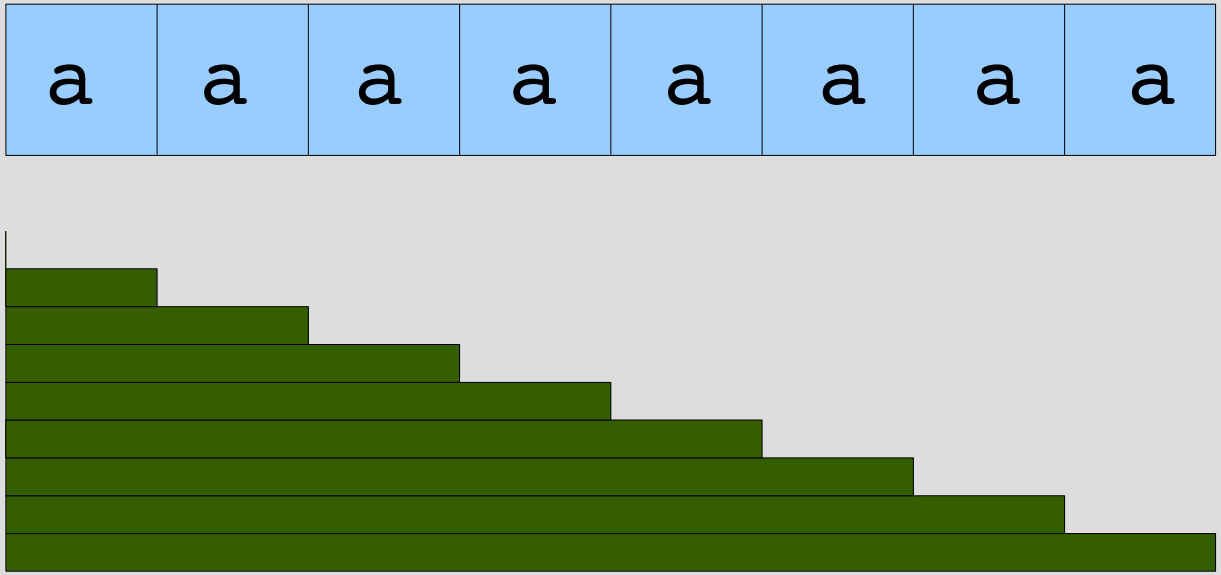
greedy: /a*/ =~ “aa”


2,1,0
epos=2
epos=1
epos=0
b3.call(2)
try_lit “a”, 1, &b3
try /a/, 1, &b3
b1.call(1)
try_rep /a/, 1, &b1
b2.call(1)
try_lit “a”, 0, &b2
try /a/, 0, &b2
b1.call(0)
try_rep /a/, 0, &b1
try /a*/, 0, &b1

lazy: /a*?/ =~ “aa”


0,1,2
epos=2
epos=1
epos=0
b3.call(2)
try_lit “a”, 1, &b3
try /a/, 1, &b3
b1.call(1)
try_rep_lazy /a/, 1, &b1
b2.call(1)
try_lit “a”, 0, &b2
try /a/, 0, &b2
b1.call(0)
try_rep_lazy /a/, 0, &b1
try /a*?/, 0, &b1

a*? の動作



エンジンの拡張: /e??/

- /e??/ は、e がない場合とある場合にマッチする
- e がない場合を先にためし、ある場合を後に試す
- /le/ と同じ
- [:opt_lazy, e] で表現する

- /behaviou??r/ =~ “behavior”　　#=> 0
- /behaviou??r/ =~ “behaviour”　　#=> 0

- matchstr([:opt_lazy, [:lit, "a"]], "aa")　#=> [0,1]
- matchstr([:opt_lazy, [:lit, "a"]], "b")　　#=> [0]

e? と e??

- e? は e がある場合を先に試す: greedy
- e?? は e がない場合を先に試す: lazy

	0～∞	0～1	1～∞
greedy	e*	e?	e+
lazy	e*?	e??	e+?

[:opt_lazy, e] の実装

```
def try(exp, seq, pos, &block)
  ...
  when :opt
    _, e = exp
    try_opt_lazy(e, seq, pos, &block)
  ...
end
```

```
def try_opt_lazy(e, seq, pos, &block)
  yield pos
  try(e, seq, pos, &block)
end
```

try_opt と try_opt_lazy の比較

```
def try_opt(e, seq, pos, &block)
  try(e, seq, pos, &block)
  yield pos
end
```

後に yield

```
def try_opt_lazy(e, seq, pos, &block)
  yield pos
  try(e, seq, pos, &block)
end
```

先に yield

レポート

- n回の繰り返しを表現する [:ntimes, n, e] を実装して解説せよ
- n は非負整数、e は抽象構文木
- Ruby の正規表現では /e{n}/
- /a{3}/ は /aaa/ と等しい
- def try_ntimes(n, e, &b) ... end
- 実装したらユニットテストで確認すること
- ∠切 2006-07-04 16:20
- IT's class
- 拡張子が txt なテキストファイル希望

まとめ

- 前回のレポートの説明
- 正規表現エンジンを再度説明
- 再帰を使う拡張
  - 存在するかもしれない: e?
  - 1回以上の繰り返し: e+
  - 怠惰な繰り返し: e*?
  - 存在しないかもしれない: e??
- レポートを出した: e{n}