テキスト処理 第6回 (2008-05-27)

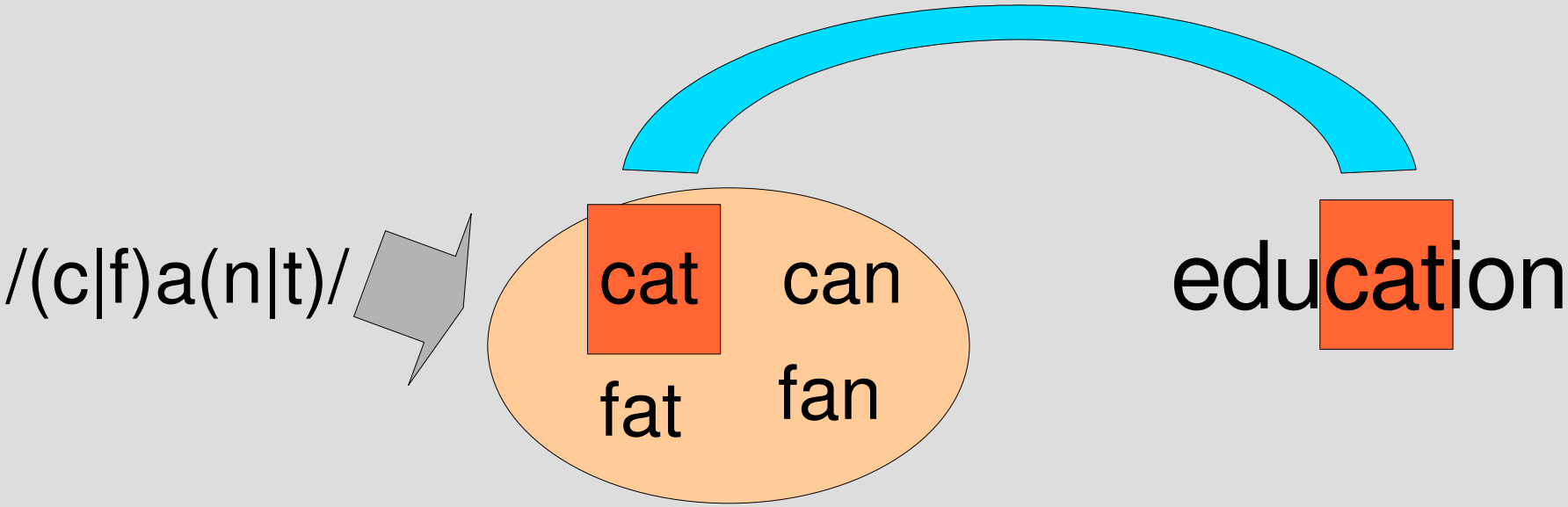
田中哲
産業技術総合研究所
情報技術研究部門
akr@isc.senshu-u.ac.jp
http://staff.aist.go.jp/tanaka-akira/textprocess-2008/

今日の内容

- まともな正規表現エンジンを作る
- レポート

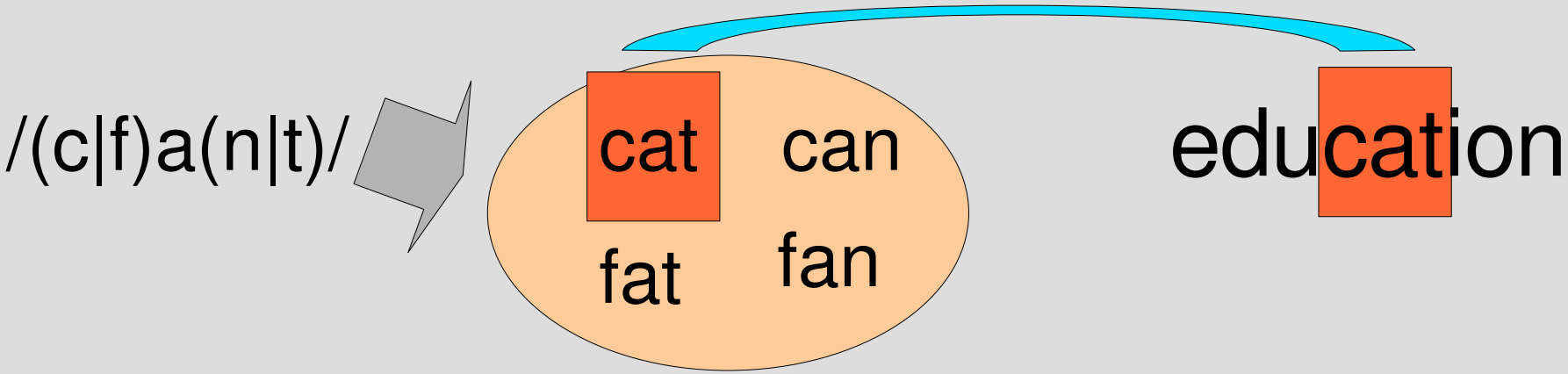
前回の正規表現エンジンの問題

- 前回の正規表現エンジンのしくみ
  1.正規表現から文字列の集合を生成
  2.その文字列集合と対象の文字列を比較



前回の正規表現エンジンの問題

- 前回の正規表現エンジンのしくみ
  1.正規表現から文字列の集合を生成
  2.その文字列集合と対象の文字列を比較
- 問題点
  - 繰り返しが扱えない
    - 文字列集合が無限になってしまう
  - 無駄が多い
    - 対象文字列を考慮せずに文字列集合を生成する



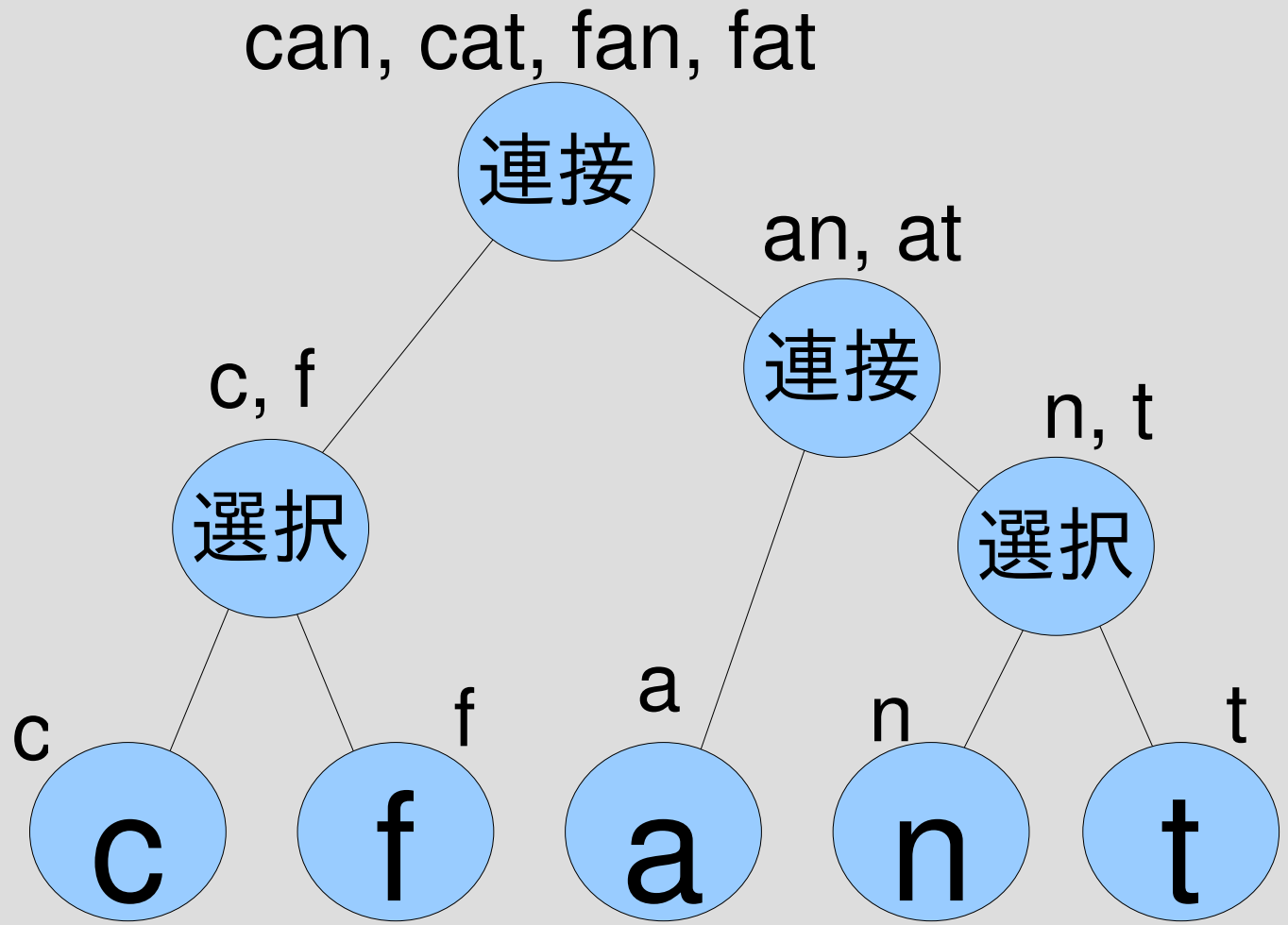
部分文字列だけを扱う

- 対象文字列の部分文字列だけに注目する
  - 部分文字列でない文字列は決してマッチしない
  - 部分文字列は有限しかない

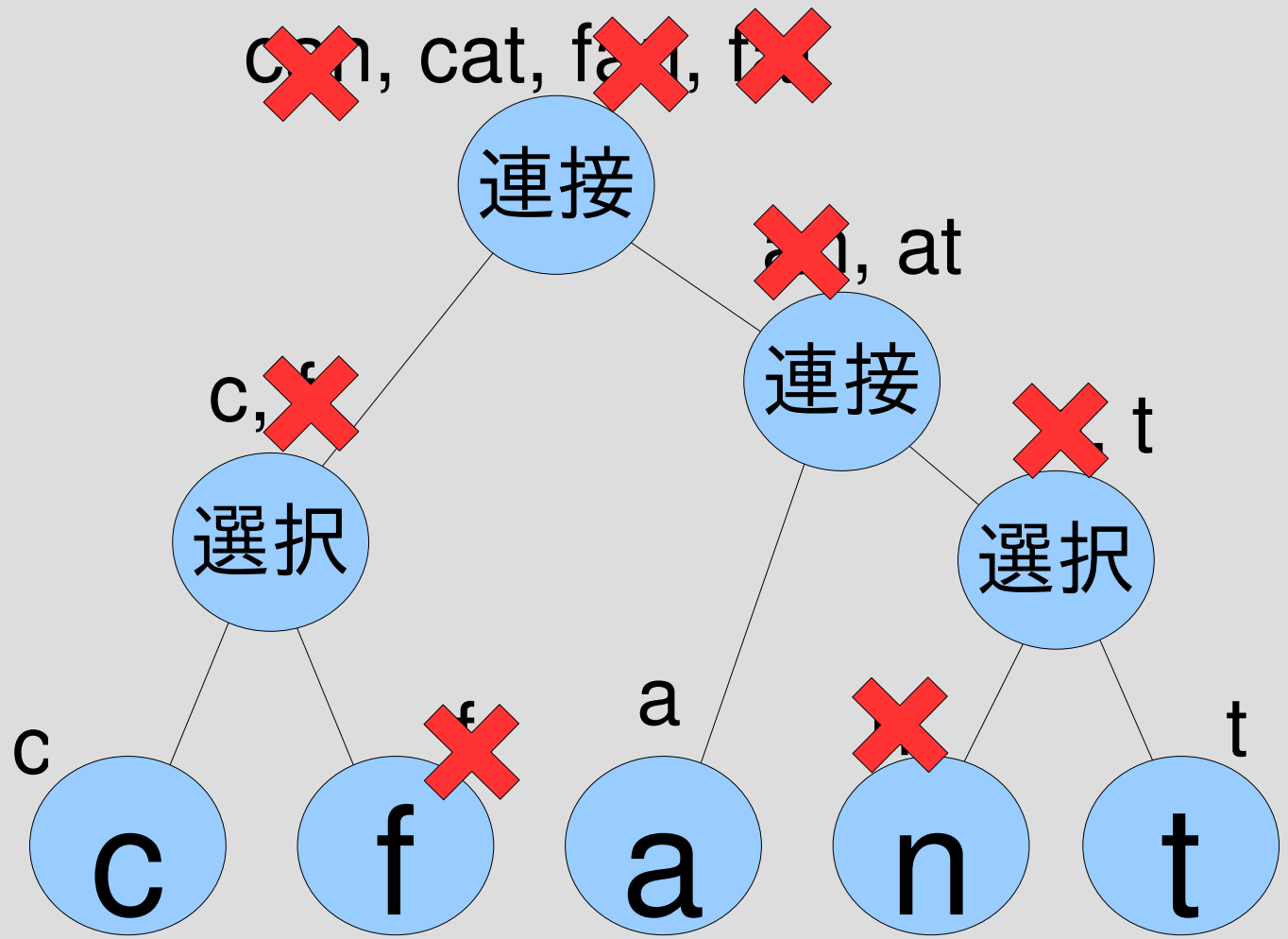
cat の部分文字列は10種類

(4つの空文字列も含む)

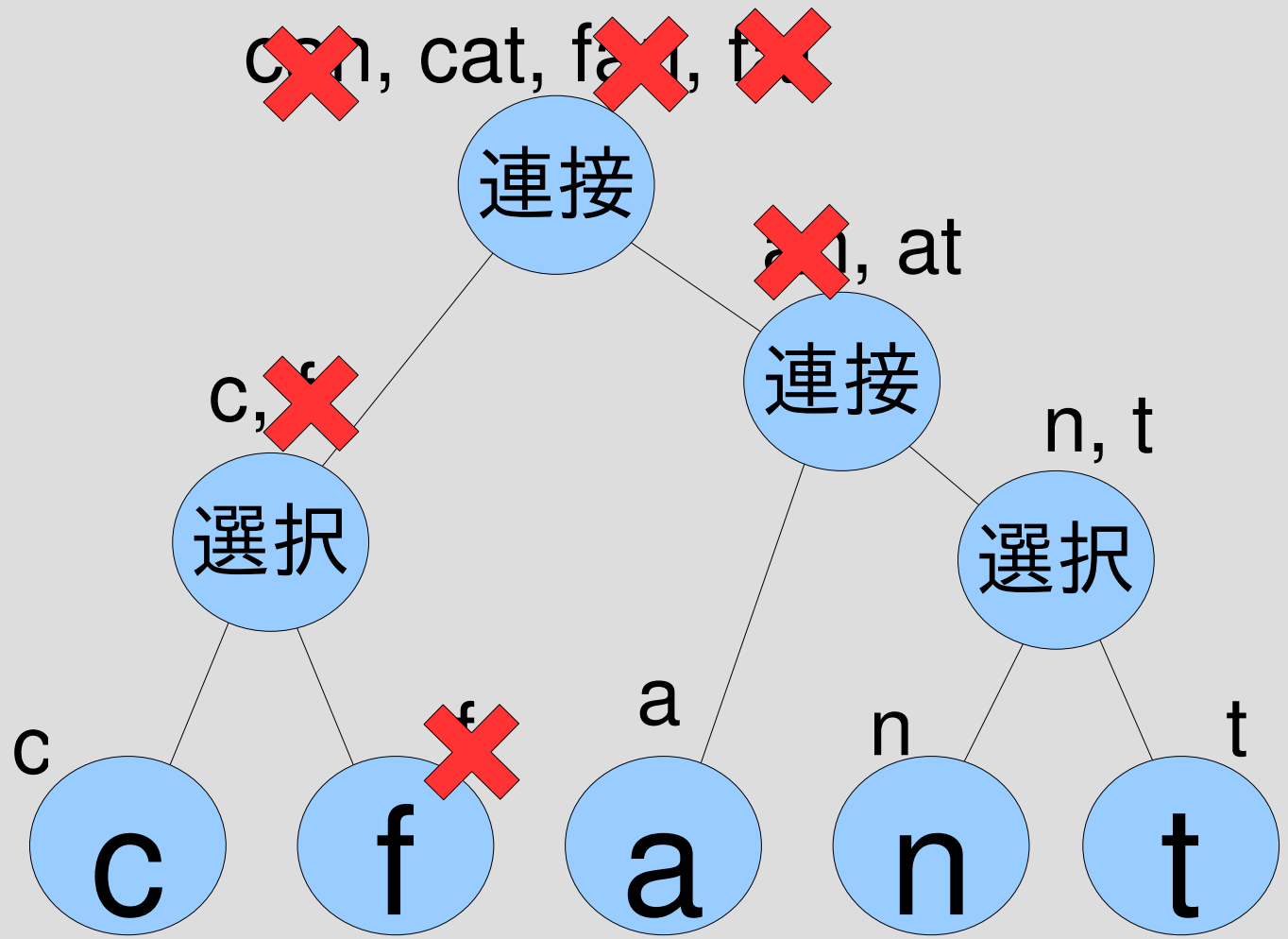
/(c|f)a(n|t)/



/(c|f)a(n|t)/ =~ "cat"



/(c|f)a(n|t)/ =~ "education"



正規表現エンジンの実装方針

- 前回
  - 正規表現を受け取る
  - 文字列集合を返す
  - 対象文字列と比較する
- 今回
  - 正規表現と対象文字列と開始位置を受け取る
  - 終端位置をひとつずつ yield する

try

- try(正規表現, 文字列, 開始位置) {|終端位置|
  ...
  }
- 正規表現、文字列、開始位置を受け取る
- 文字列の開始位置から始まる部分文字列で正規表現にマッチするものを探し、その終端位置を引数にしてブロックを呼ぶ (yield する)
- 正規表現は配列で表現
- 文字列は単一文字の文字列の配列で表現

ブロックの復習

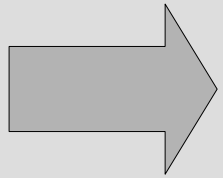
- def countup(first, last)
  i = first
  while i <= last
    yield i　　# ブロックの呼出し
    i += 1
  end
end
- countup(3,6) {|x| p x }
  ブロックは4回呼び出される (yield される)
  そのとき、x はそれぞれ 3, 4, 5, 6 になる

try(re, str, pos) {|pos2| } の動作

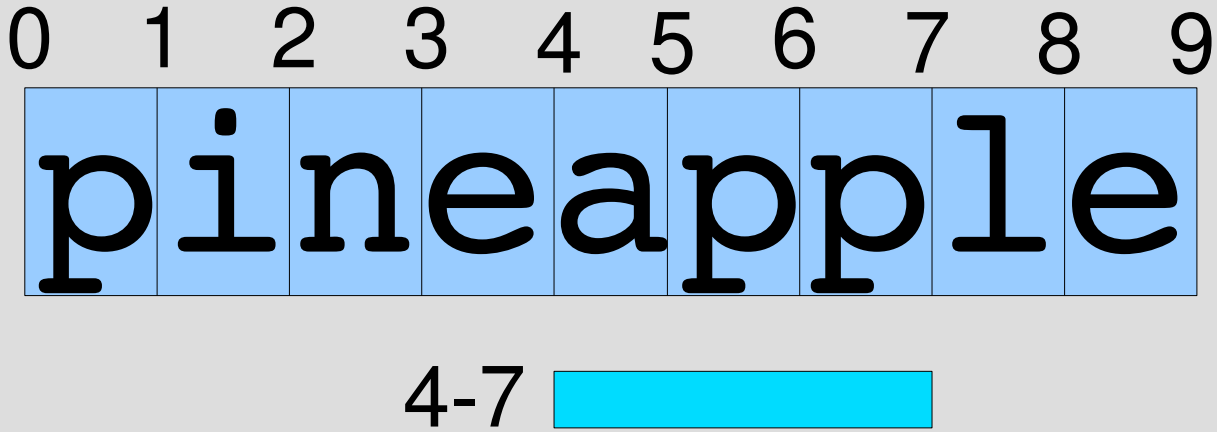
re=[:cat, "a", [:cat, "p", "p"]]

str=["p","i","n","e","a","p","p","l","e"]

pos=4



pos2=7

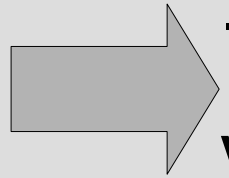


try(re, str, pos) {|pos2| } の動作

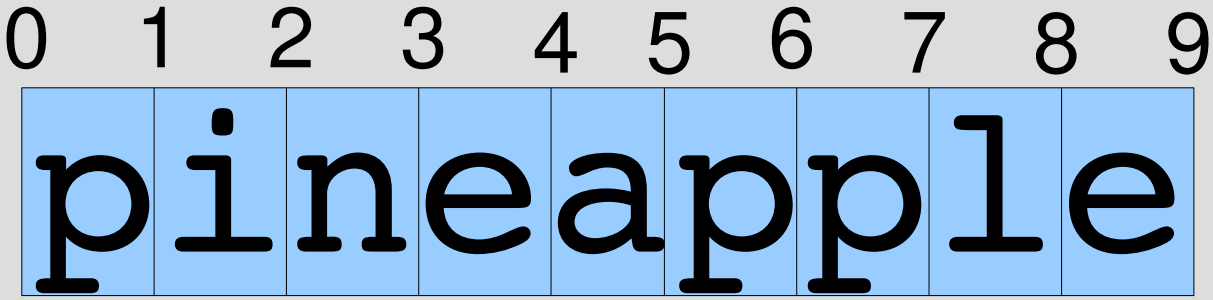
re=[:cat, "a", [:cat, "p", "p"]]

str=["p","i","n","e","a","p","p","l","e"]

pos=3



一回も
yield
されない

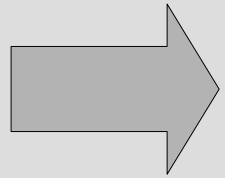


try(re, str, pos) {|pos2| } の動作

re=[:rep, "p"]

str=["p","i","n","e","a","p","p","l","e"]

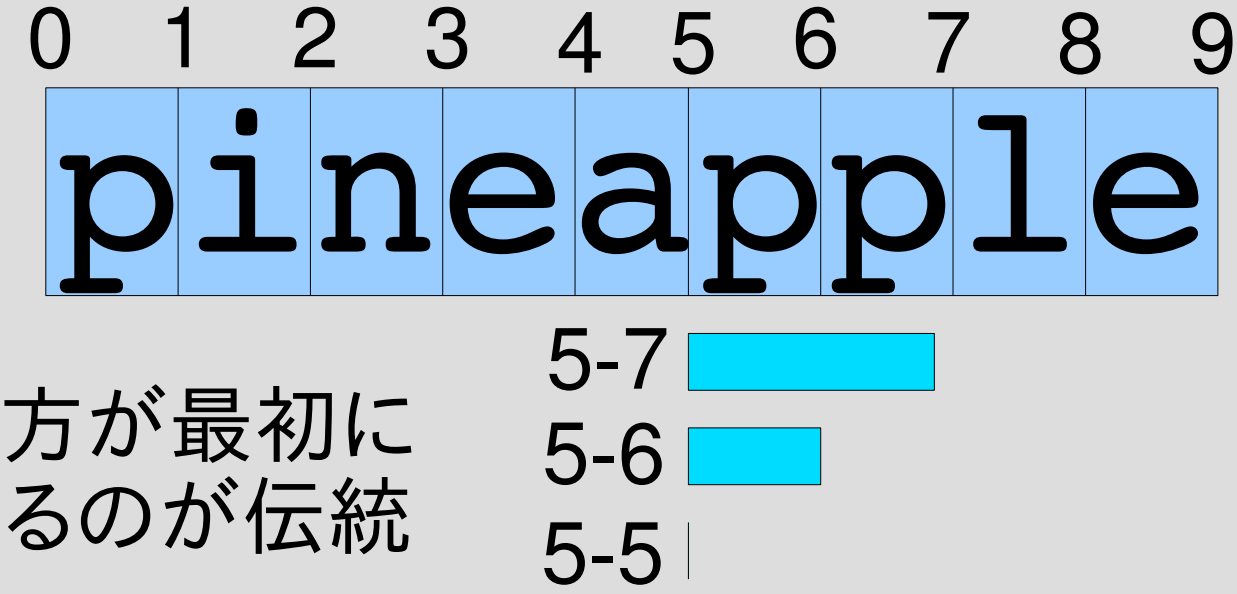
pos=5



pos2=7

pos2=6

pos2=5



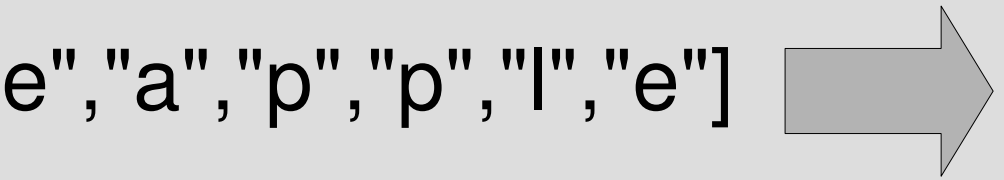
長い方が最初に
表れるのが伝統

try(re, str, pos) {|pos2| } の動作

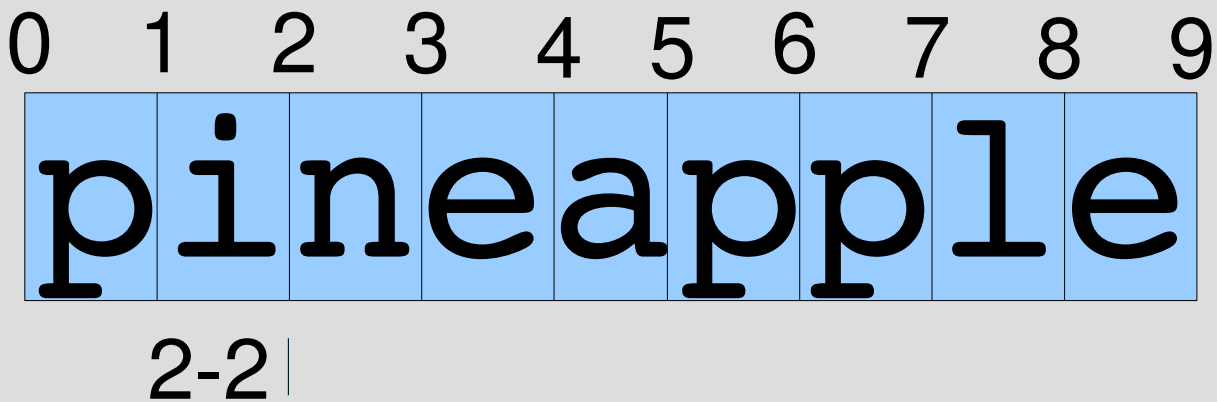
re=[:rep, "p"]

str=["p","i","n","e","a","p","p","l","e"]

pos=2



pos2=2



%w[...]

- 文字列の配列を記述する記法
- ... を空白区切りとした文字列の配列
- %w[apple orange] == ["apple", "orange"]
- %w[p i n e a p p l e] == ["p","i","n","e","a","p","p","l","e"]
- %w の w は word (単語) の w

try(re, str, pos) {|pos2| } の動作

re=[:rep, "a"]

str=%w[a a a a a a a a a a]

pos=0

pos2=9

pos2=8

...

pos2=0

try による正規表現エンジン

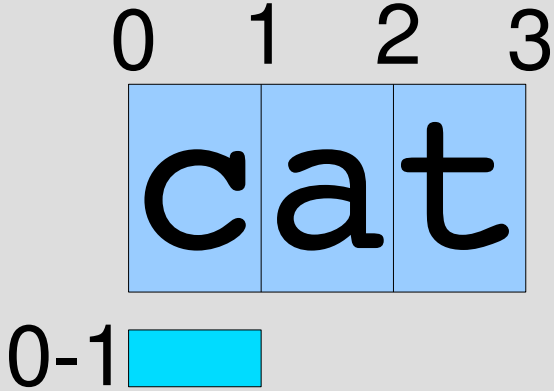
- try を使って
  - 開始位置を先頭として文字列末尾が yield されるかどうか調べれば文字列全体がマッチするかどうか分かる
  - 開始位置をずらしながら文字列の各位置から繰り返せばマッチする部分文字列が存在するかどうか分かる
- try によって正規表現マッチが実現できる

try の実装

```
def try(re, str, pos)
  if re.respond_to? :to_str
    [redacted]
  else
    case re[0]
    when :empset
      [redacted]
    when :empstr
      [redacted]
    when :cat
      [redacted]
    when :alt
      [redacted]
    when :rep
      [redacted]
    end
  end
end
```

文字とのマッチ

- try("c", %w[c a t], 0) {|pos2| p pos2 }
  #=> 1

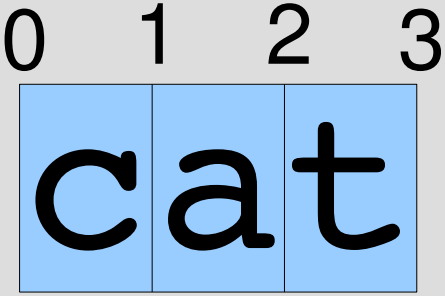


実装:

```
if re.respond_to? :to_str
  yield pos+1 if str[pos] == re
else ...
```

空集合 [:empset]

- try([:empset], %w[c a t], 0) {|pos2| p pos2 }
  #=> ブロックは呼び出されずなにも表示されない

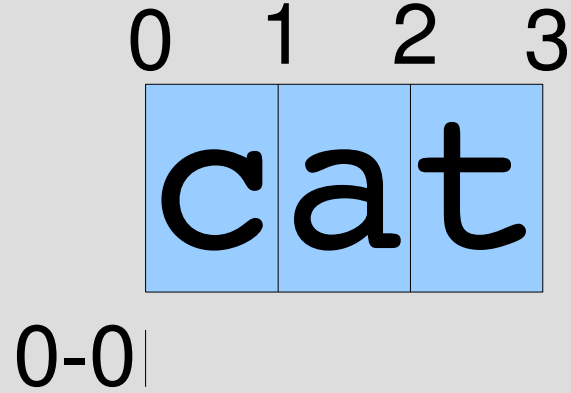


実装:

```
case re[0]
when :empset
  # nothing to do
when ...
```

空文字列 [:empstr]

- try([:empstr], %w[c a t], 0) {|pos2| p pos2 }
  #=> 0

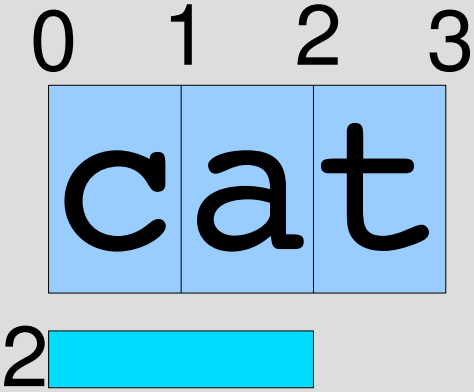


実装:

```
when :empstr
  yield pos
when ...
```

連接 [:cat, r1, r2]

- try([:cat, "c", "a"], %w[c a t], 0) {|pos2| p pos2 }
  #=> 2

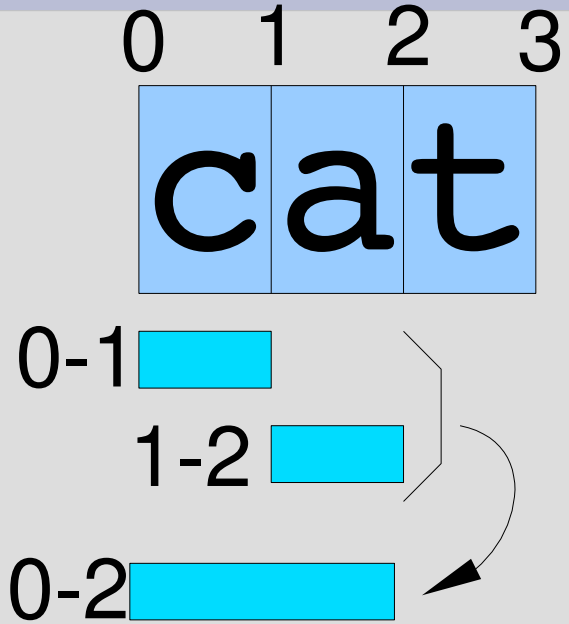


実装:

```
when :cat
  try_cat(re, str, pos) {|pos2| yield pos2 }
when ...
```

try_catの実装

```
def try_cat(re, str, pos)
  try(re[1], str, pos) {|pos2|
    try(re[2], str, pos2) {|pos3|
      yield pos3
    }
  }
end
```



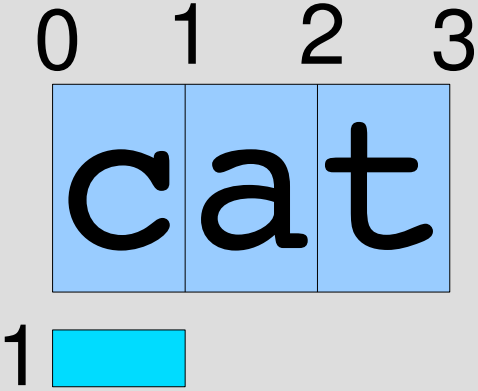
try("c", %w[c a t], 0) が 1 を発見

try("a", %w[c a t], 1) が 2 を発見

発見した 2 を yield

選択 [:alt, r1, r2]

- try([:alt, "c", "a"], %w[c a t], 0) {|pos2| p pos2 }
  #=> 1

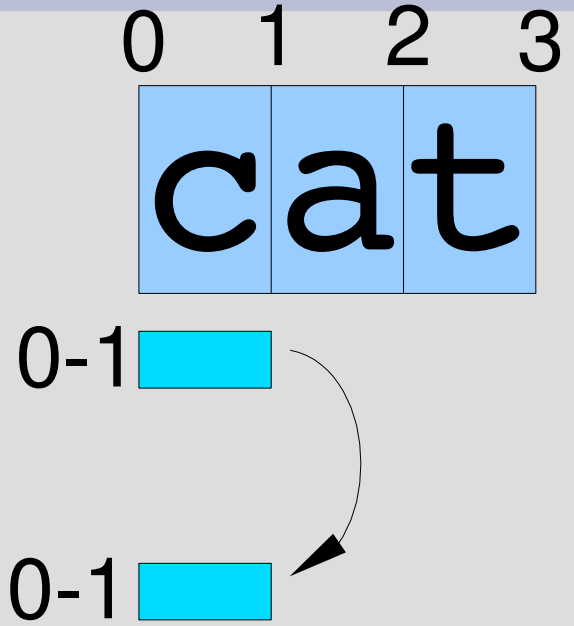


実装:

```
when :cat
  try_alt(re, str, pos) {|pos2| yield pos2 }
when ...
```

try_altの実装

```
def try_alt(re, str, pos)
  try(re[1], str, pos) {|pos2|
    yield pos2
  }
  try(re[2], str, pos) {|pos2|
    yield pos2
  }
end
```

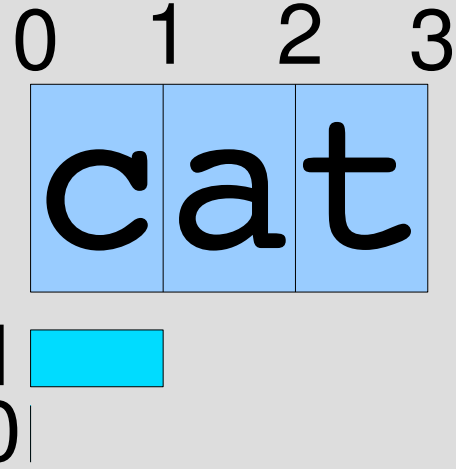


try("c", %w[c a t], 0) が 1 を発見してyield

try("a", %w[c a t], 0) はなにも発見しない

繰り返し [:rep, r]

- try([:rep, "c"], %w[c a t], 0) {|pos2| p pos2 }
  #=> 1, 0

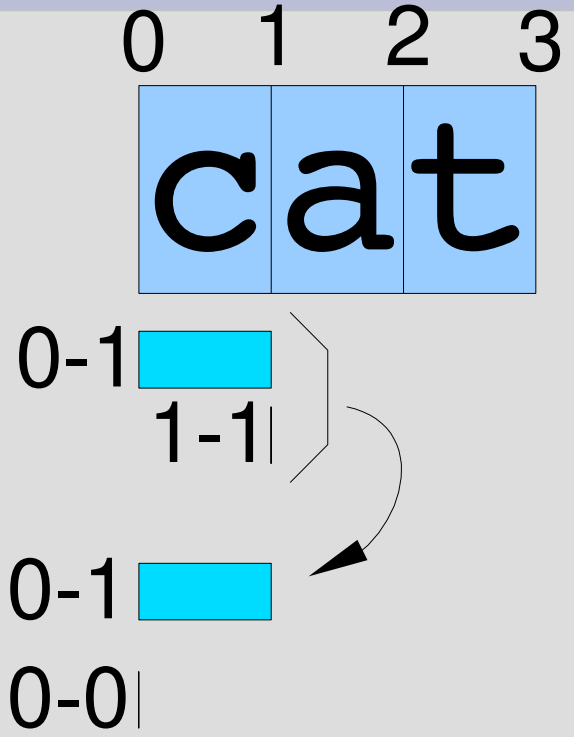


実装:

```
when :cat
  try_rep(re, str, pos) {|pos2| yield pos2 }
when ...
```

try_repの実装

```
def try_rep(re, str, pos)
  try(re[1], str, pos) {|pos2|
    try(re, str, pos2) {|pos3|
      yield pos3
    }
  }
  yield pos
end
```



try("c", %w[c a t], 0) が 1 を発見
  try([:rep, "c"], %w[c a t], 1) が 1 を発見
0 を発見

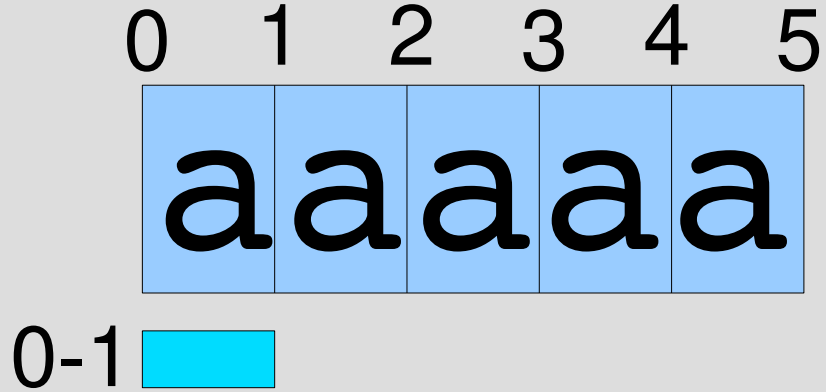
/¥A(a|aa)(a|aa)(a|aa)/ =~ "aaaaa"

- try([:cat, [:alt, "a", [:cat, "a", "a"]],
  [:cat, [:alt, "a", [:cat, "a", "a"]],
  [:alt, "a", [:cat, "a", "a"]]]],
  "aaaaa",
  0) {|pos| p pos }

  #=>
  3, 4, 4, 5, 4, 5, 5

/¥Aa/ =~ "aaaaa"

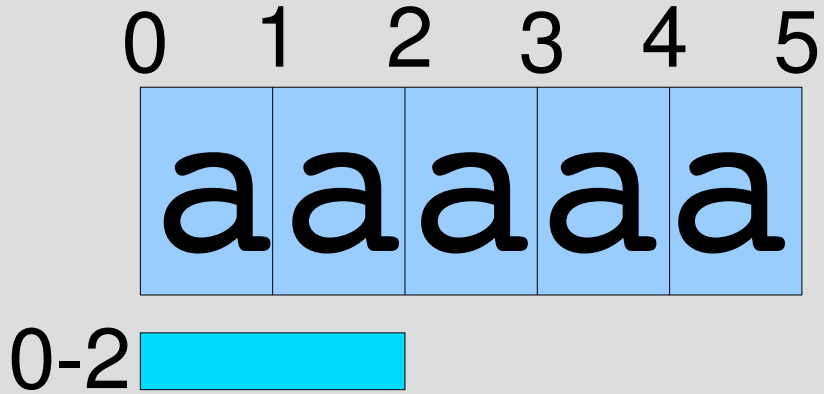
- try(/a/, "aaaaa", 0) {|pos| }
- pos は 1 になる



re, str は読みやすさのため配列でなく書いてある
実際には配列で書かないと動かない

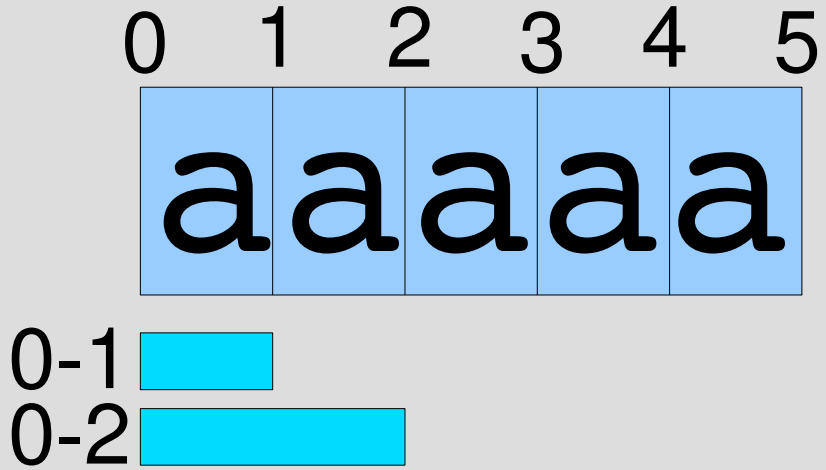
/¥Aaa/ =~ "aaaaa"

- try(/aa/, "aaaaa", 0) {|pos| }
- pos は 2 になる



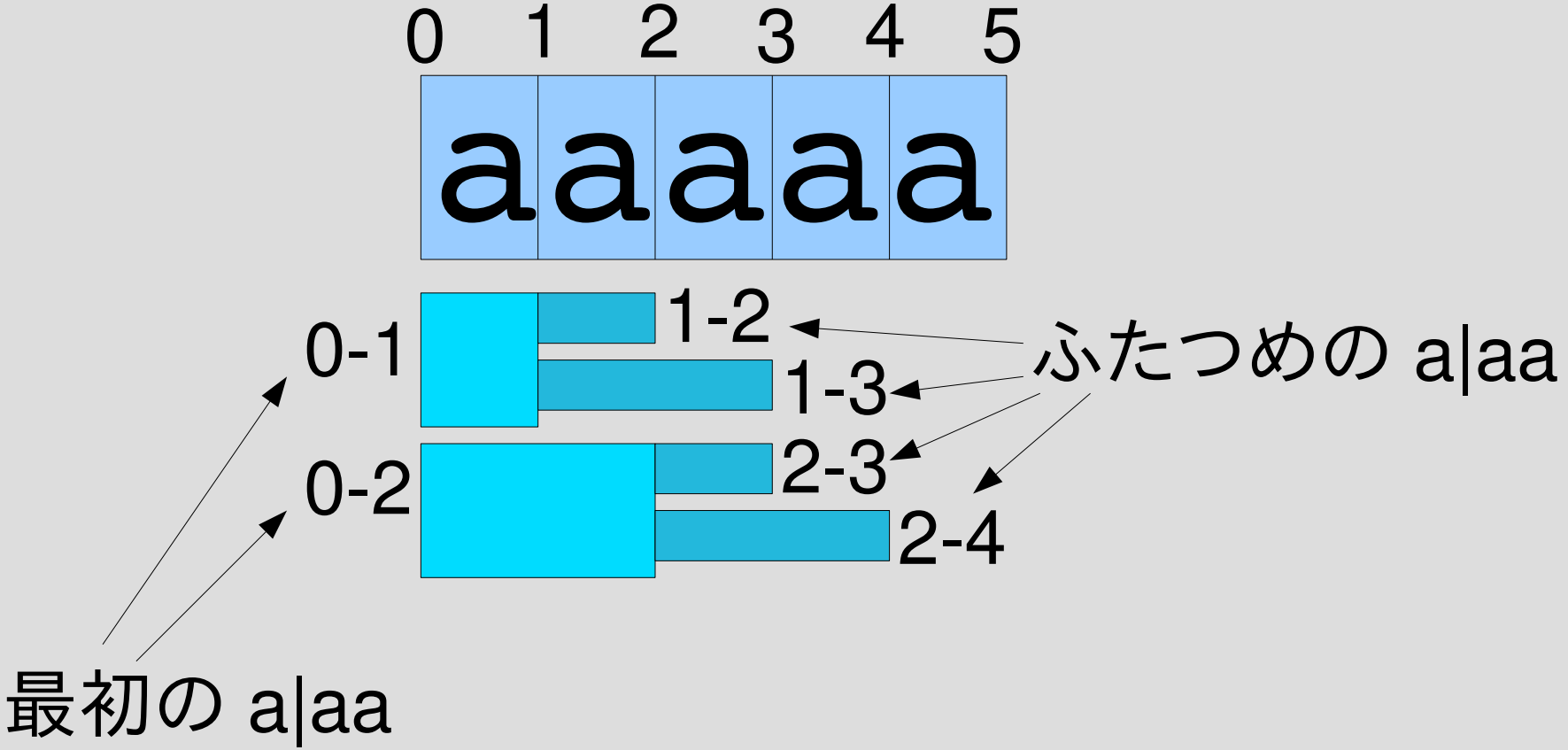
/¥A(a|aa)/ =~ "aaaaa"

- try(/(a|aa)/, "aaaaa", 0) {|pos| }
- pos は 1, 2 になる



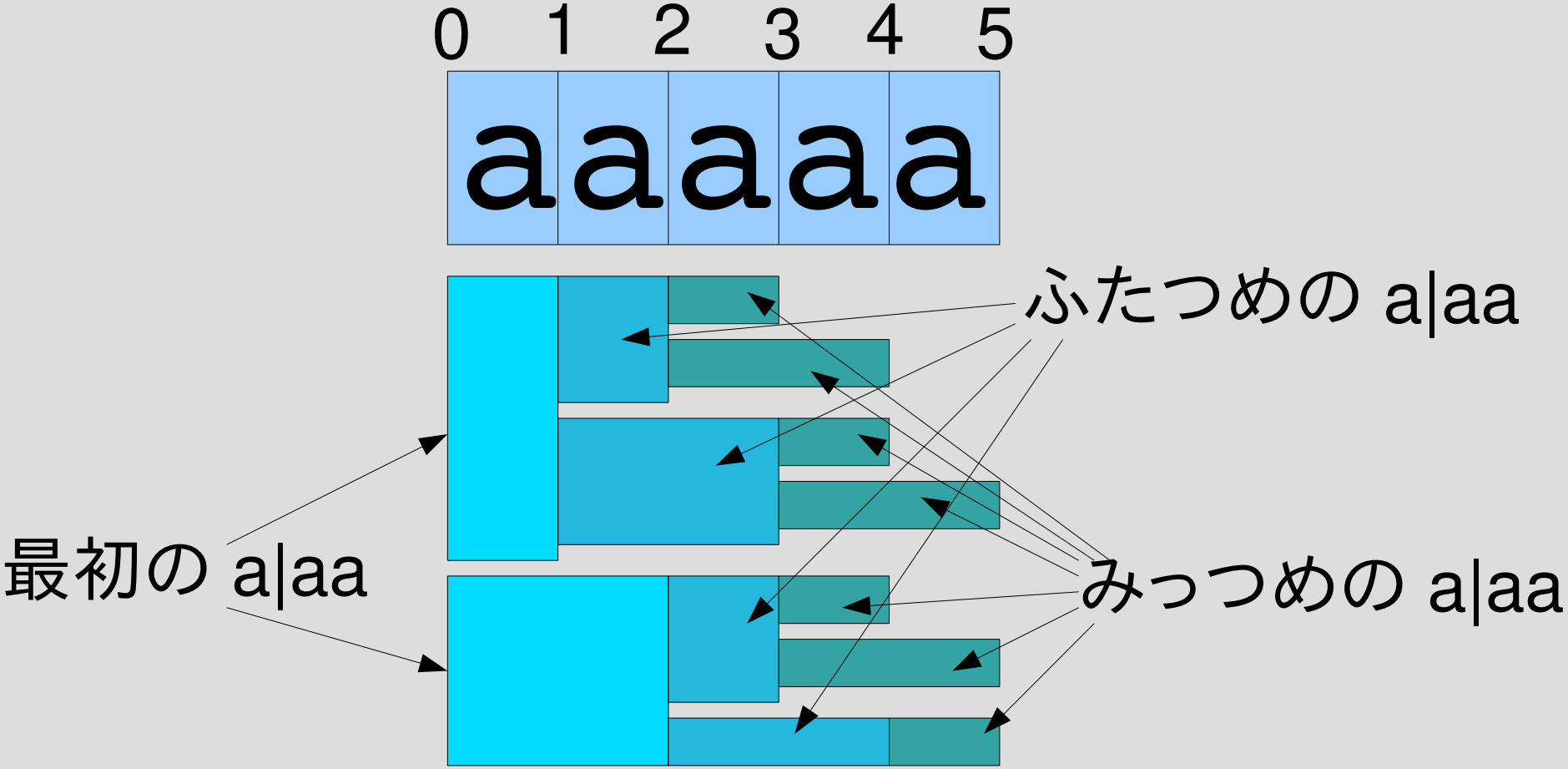
/¥A(a|aa)(a|aa)/ =~ "aaaaa"

- try(/(a|aa)(a|aa)/, "aaaaa", 0) {|pos| }
- pos は 2, 3, 3, 4 になる

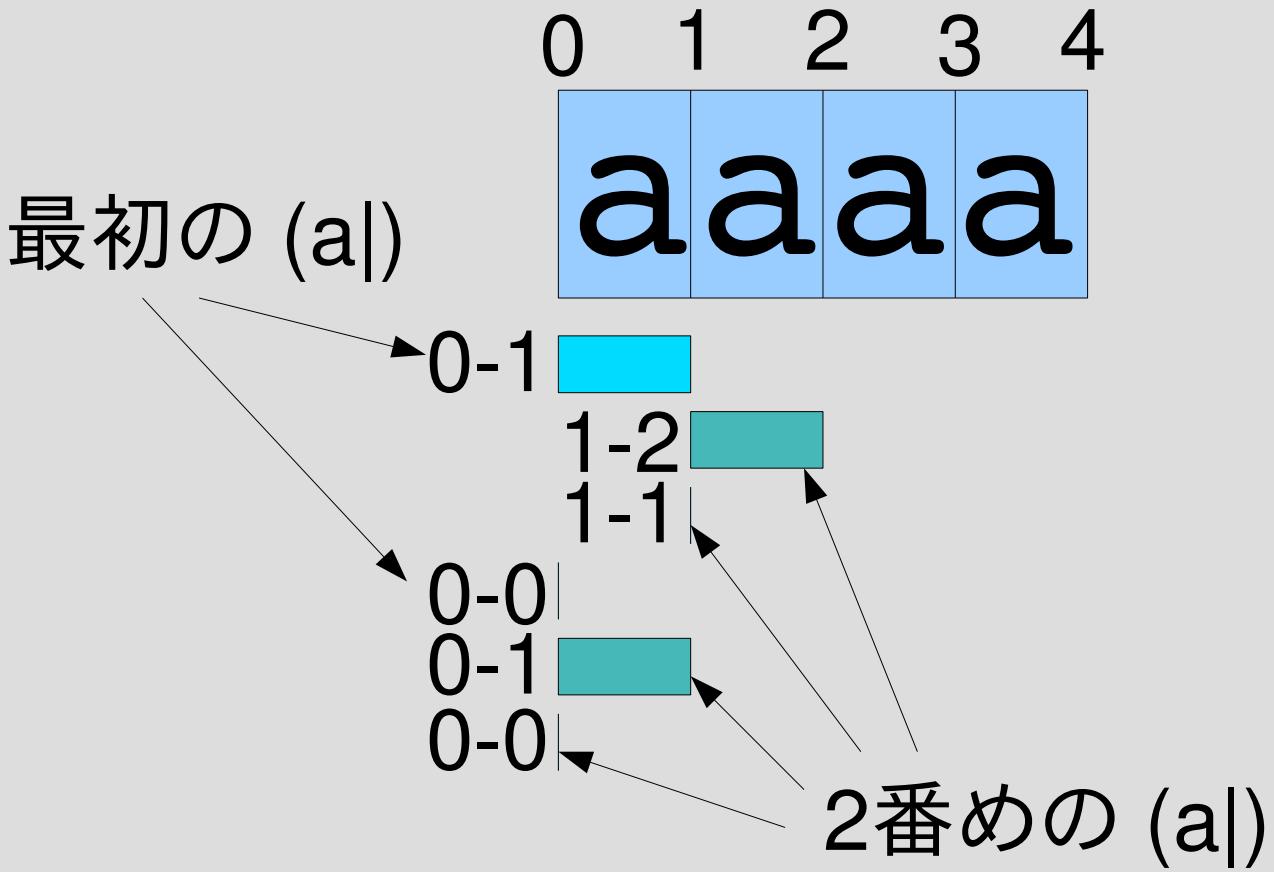


/¥A(a|aa)(a|aa)(a|aa)/ =~ "aaaaa"

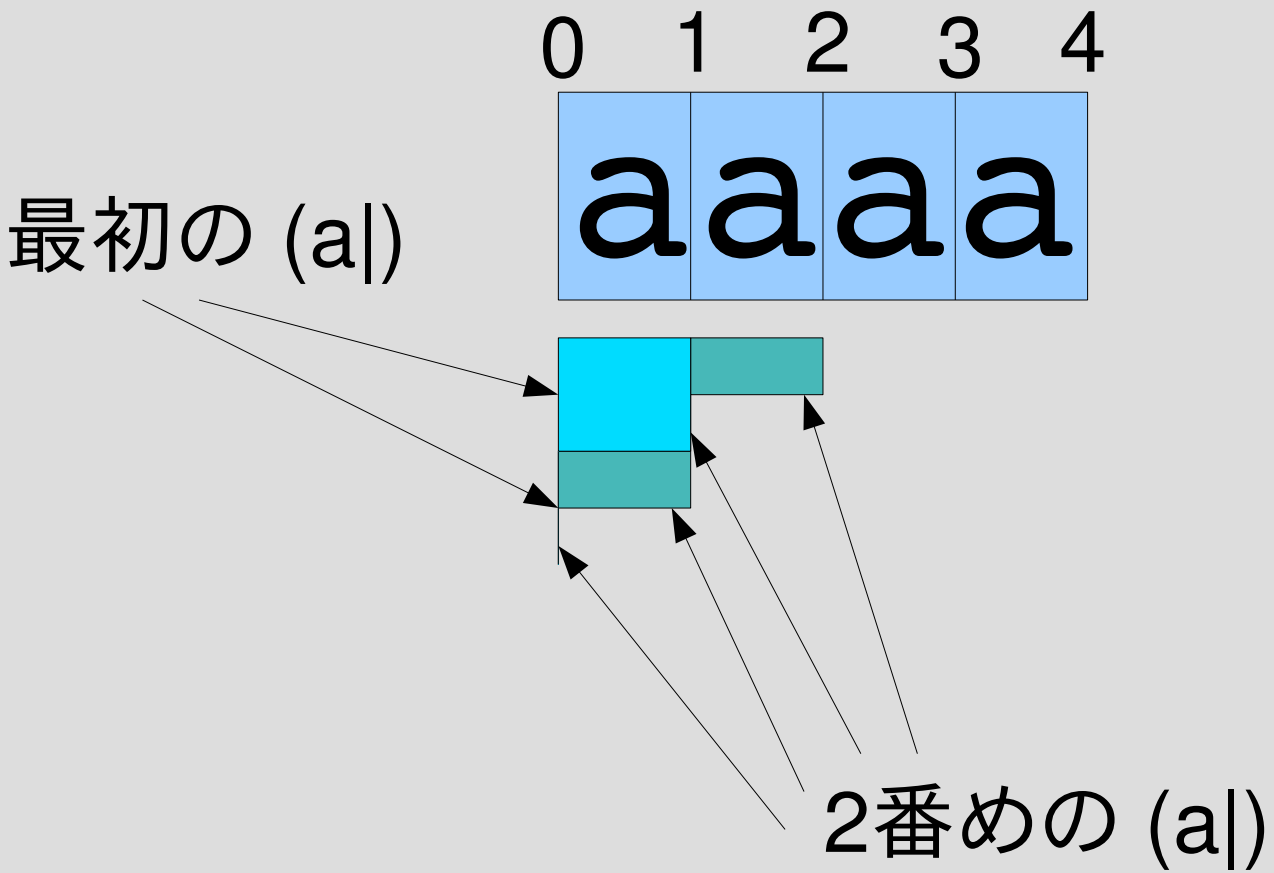
- try(/(a|aa)(a|aa)(a|aa)/, "aaaaa", 0) {|pos| }
- pos は 3, 4, 4, 5, 4, 5, 5 になる



/¥A(a|)(a|)/ =~ "aaaa"

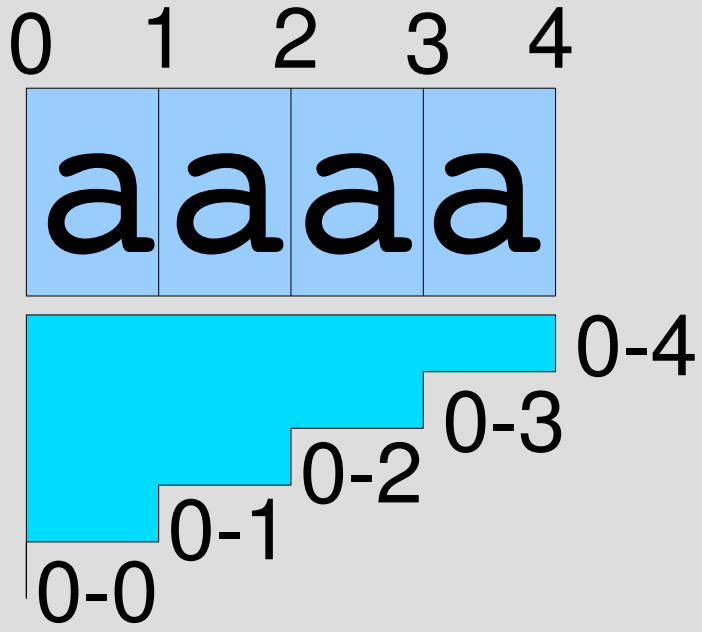


/¥A(a|)(a|)/ =~ "aaaa"



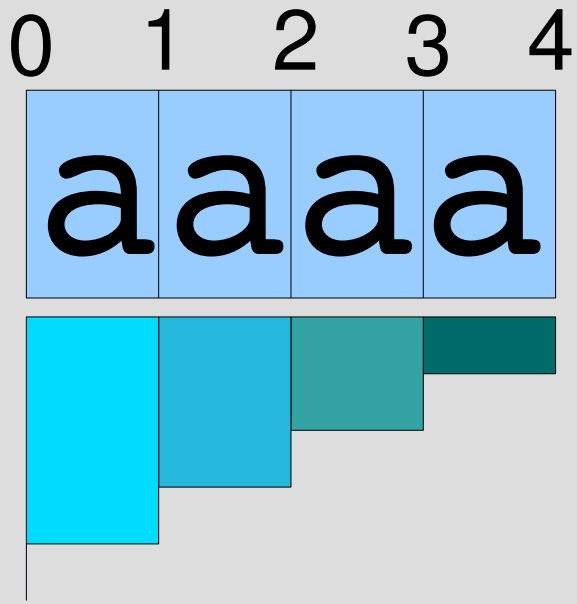
/¥Aa*/ =~ "aaaa"

- 4, 3, 2, 1, 0 の順で終端を生成する



/¥Aa*/ =~ "aaaa"

- 内部的には4段階の再帰が行われる



try_repの中身

```
def try_rep(re, str, pos)
  try(re[1], str, pos) {|pos2|  最初のひとつを発見
    try(re, str, pos2) {|pos3|
      yield pos3     最初のひとつの後に続いた
    }                0回以上の繰り返しを発見
  }
  yield pos       0回の繰り返しを発見
end
```

try_rep の別実装

- /a*/ と /(aa*)|/ は等しい

```
def try_rep(re, str, pos)
  try([:alt, [:cat, re[1], re], [:empstr]], str, pos) {|pos2|
    yield pos2
  }
end
```

try を展開すると元の実装と一致する

レポート

- 選択(:alt) を任意個引数に拡張せよ
- 連接(:cat) を任意個引数に拡張せよ

- 〆切 2008-06-03 12:00
- RENANDI
- 拡張子が txt なプレインテキストが望ましい

[:alt, r1, r2, r3, ...]の動作例

- try([:alt, "a", "b", "c"], %w[a], 0) {|pos| p pos }
  #=> 1
- try([:alt, "a", "b", "c"], %w[b], 0) {|pos| p pos }
  #=> 1
- try([:alt, "a", "b", "c"], %w[c], 0) {|pos| p pos }
  #=> 1
- try([:alt, "a", [:cat, "a", "a"]], %w[a a], 0) {|pos|
    p pos
  }
  #=> 1, 2

ユニットテスト

- 例のとおりに動作しているかどうか確かめるのは面倒くさい
- ユニットテストとしてコードとその結果を書いておけば、そのコードがその結果になるかどうかを確認してくれる
- 今回のレポートに対するユニットテストを配布する
- ユニットテストが成功するようにプログラムを拡張すること

ユニットテストの実行のしかた

```
% ruby test-rx.rb
Loaded suite test-rx
Started
.F.F....
Finished in 0.03028669 seconds.


  1) Failure:
test_alt_multiarg(TestRX) [test-rx.rb:88]:
<[1]> expected but was
<[]>.


  2) Failure:
test_cat_multiarg(TestRX) [test-rx.rb:99]:
<[4]> expected but was
<[2]>.


8 tests, 12 assertions, 2 failures, 0 errors
```

- ruby test-rx.rb で実行
- 0 failures, 0 errors と出てきたら成功
- 失敗したときは場所と理由が出てくる
- レポートに出題した部分は実装していないので当然失敗する

test-rx.rb

```
def try... # try など講義で述べたコード
require 'test/unit'  # ユニットテストのライブラリを使用
def rx_ends(re, str, pos)
   # テストのためのユーティリティ関数
  a = []
  try(re, str.split(//), pos) {|pos2| a << pos2 }
  a
end
```

str.split(//) は文字列を
文字の配列に分解する

test-rx.rb (続き)

```
class TestRX < Test::Unit::TestCase
  def test_empset # 空集合のテスト
    assert_equal([], rx_ends([:empset], "", 0))
  end

  def test_empstr # 空文字列のテスト
    assert_equal([0], rx_ends([:empstr], "", 0))
  end
```

test-rx.rb (抜粋)

```
def test_cat # 連接のテスト
  assert_equal([2], rx_ends([:cat, "a", "b"], "ab", 0))
  assert_equal([2,1,1,0],
    rx_ends([:cat, [:alt, "a", [:empstr]],
                   [:alt, "a", [:empstr]]],
            "aaaa", 0))
end
```

[:cat, r1, r2, r3, ...]の動作例

- try([:cat, "a", "b", "c"], %w[a b c], 0) {|e| p e }
  #=> 3
- try([:cat, [:alt, "a", [:cat, "a", "a"]],
  [:alt, "a", [:cat, "a", "a"]],
  [:alt, "a", [:cat, "a", "a"]]],
  %w[a a a a a a], 0) {|pos| p pos }
  #=> 3,4,4,5,4,5,5,6

:cat の任意個引数のヒント

- 再帰を使う
- 難しいかも?
- 出来なかった場合、3引数、4引数を作ってみよ
  - [:cat3, r1, r2, r3]
  - [:cat4, r1, r2, r3, r4]

まとめ

- 正規表現エンジンを作ってみた
  - 繰り返しも扱える
  - やっぱり再帰
  - ブロックをたくさん使った
- レポートを出した
  - ユニットテストを配布するので実装したら確かめること