テキスト処理 第9回 (2007-07-03)

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

- 正規表現エンジン計算量レポートの解説
- Ruby の正規表現の動作と実現
- 正規表現エンジンにいくつか機能を拡張する (再帰を使わない機能)
- レポート

Ruby の正規表現の動作

- マッチすることがわかったら残りの可能性は検査しない
- マッチするというのはパターンが「含まれる」こと

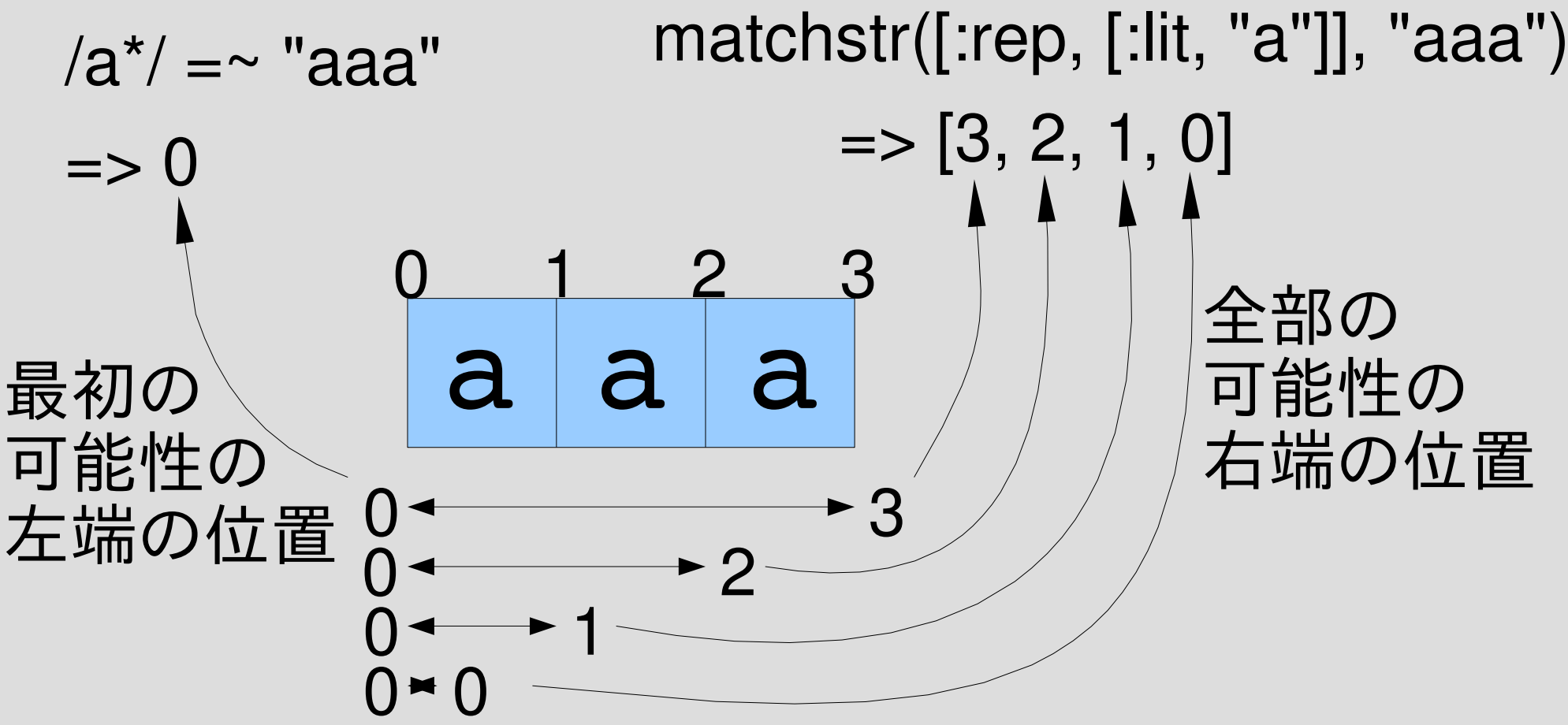
正規表現エンジンでRubyのような動作

```
def hasmatch(exp, str)
  ary = str.split(//)
  0.upto(ary.length) {|i|
    try(exp, ary, i) {
      return i
    }
  }
  nil
end
```

最初の可能性だけを返す

- matchstr はすべての可能性を列挙する
  matchstr([:rep, [:lit, "a"]], "aaa") #=> [3, 2, 1, 0]
  - 位置0から始まって位置3で終わる可能性
  - 位置0から始まって位置2で終わる可能性
  - 位置0から始まって位置1で終わる可能性
  - 位置0から始まって位置0で終わる可能性
- Ruby の =~ は最初のひとつしか求めない
  /a*/ =~ "aaa" #=> 0
  - 位置0から始まって位置3で終わる可能性

/a*/ =~ "aaa"


/a*/ =~ "aaa"
=> 0
matchstr([:rep, [:lit, "a"]], "aaa")
=> [3, 2, 1, 0]
0 1 2 3
a a a
最初の
可能性の
左端の位置
全部の
可能性の
右端の位置
0 3
0 2
0 1
0 0

最初の可能性を返す

```
def startwithmatch(exp, str)
  try(exp, str.split(//), 0) {|pos|
    return 0
  }
  nil
end
```

try が yield したら即座に
startwithmatch から return する
try はその時点で中断される

/\Apattern/ =~ "string" に類似

計算量: 速くなるか?

- 最初のマッチで中断しても、時間がかかるときには時間がかかる
- マッチしないのであれば、結局全部の可能性を試すことになる

matchstr([:rep, [:rep, [:lit, "a"]]], "aaa...aaa")　遅い

startwithmatch([:rep, [:rep, [:lit, "a"]]], "aaa...aaa")　　速い

startwithmatch([:cat, [:rep, [:rep, [:lit, "a"]]], [:lit, "b"]], "aaa...aaac")　　遅い

パターンが含まれる

- /b/ =~ "abc"　　#=> 1　マッチする
- matchstr([:lit, "b"], "abc")　　#=> []　マッチしない

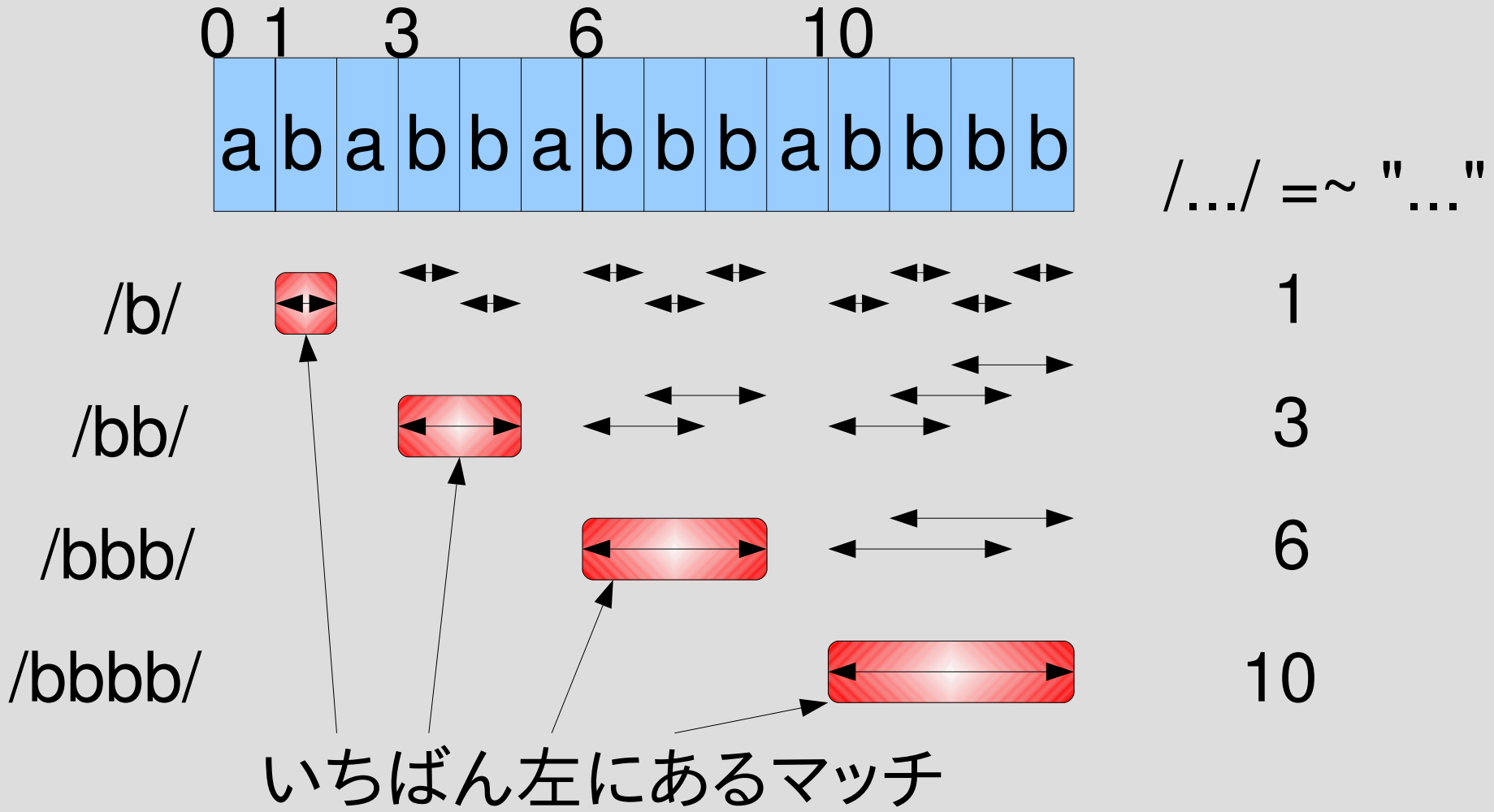
- matchstr は先頭からしか調べないが、
  Ruby のは途中からのも調べる

複数あったら?

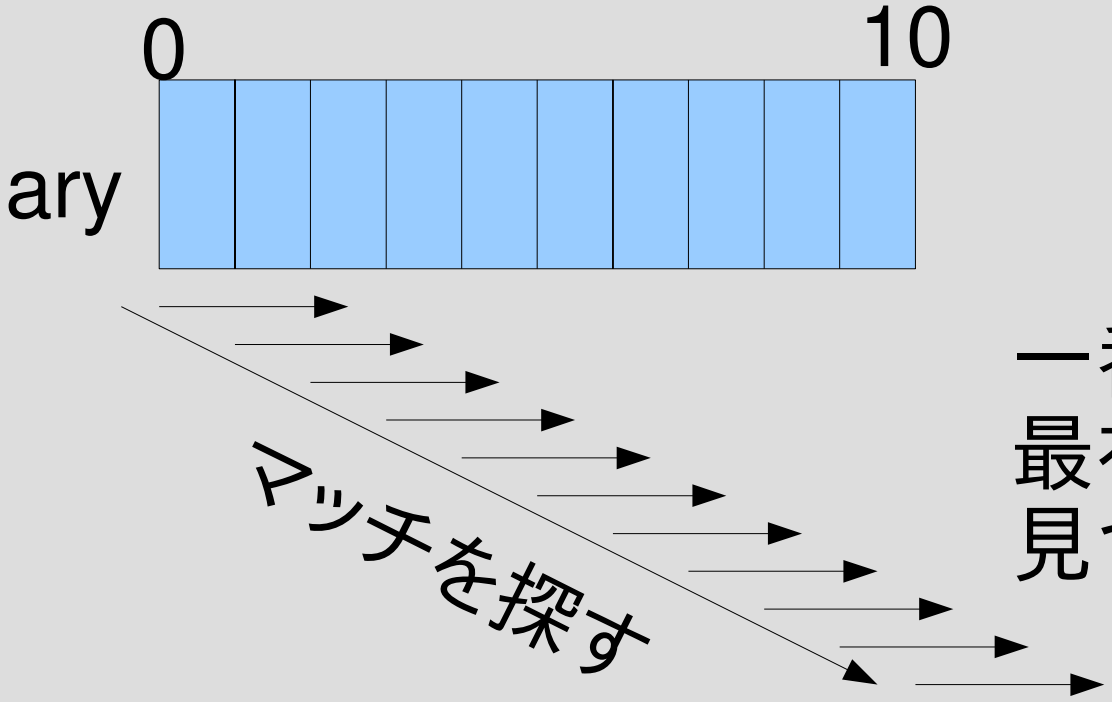
- /b/ =~ "ababbabbbabbbb" #=> 1
- /bb/ =~ "ababbabbbabbbb" #=> 3
- /bbb/ =~ "ababbabbbabbbb" #=> 6
- /bbbb/ =~ "ababbabbbabbbb" #=> 10

- いちばん左にあるマッチ (の左端) を返す

複数あったら?

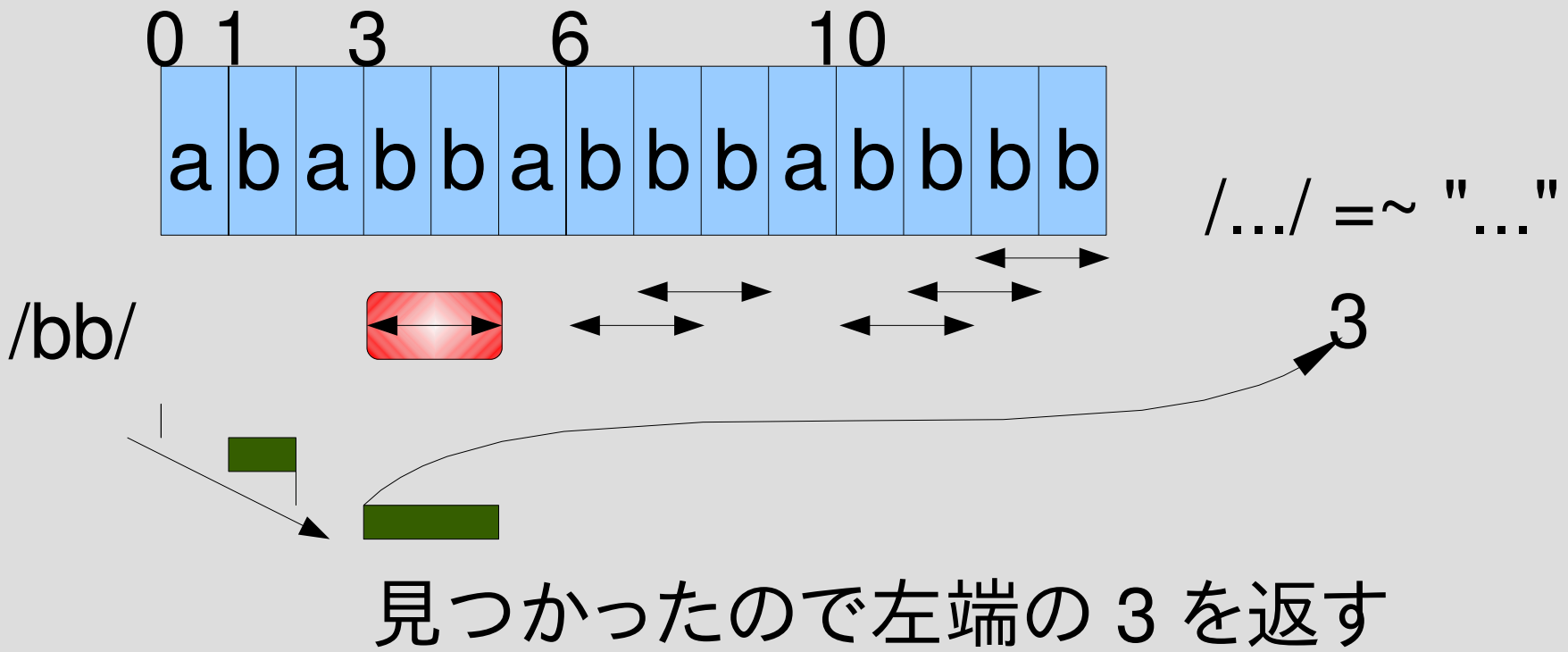


文字列先頭から順に試す



一番左にあるものが
最初に見つかる
見つかったらそれを返す

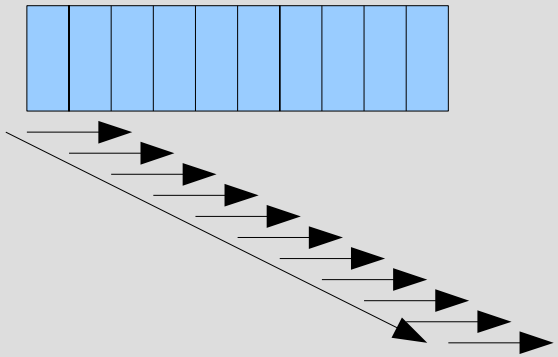
文字列先頭から順に試す



見つかったので左端の 3 を返す

文字列先頭から順に試す

```
def hasmatch(exp, str)
  seq = str.split(//)
  0.upto(seq.length) {|i|
    try(exp, seq, i ) {
      return i
    }
  }
  nil
end
```



いくつか機能を拡張する

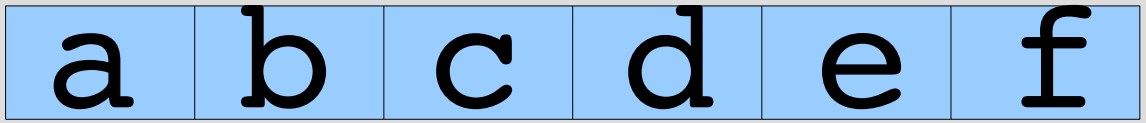
- 任意の一文字
- ¥A　文字列の先頭の位置
- ¥z　文字列の末尾の位置
- ^　　行頭
- $　　行末

任意の一文字

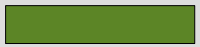
- 任意の一文字にマッチするパターン [:anysym]
- Ruby の正規表現には単純には対応しない
  - . (ドット) に似ているが ¥n にもマッチする
  - 強いていえば (.|¥n) に対応する
- matchstr([:anysym], "abc")　　#=> [1]
- matchstr([:anysym], "")　　#=> []
- matchstr([:rep, [:anysym]], "ab¥ncd¥n")
  #=> [6,5,4,3,2,1,0]
- matchstr([:rep, [:anysym]], "abcdef")
  #=> [6,5,4,3,2,1,0]

「任意の一文字」のマッチ

matchstr(exp, "abcdef")

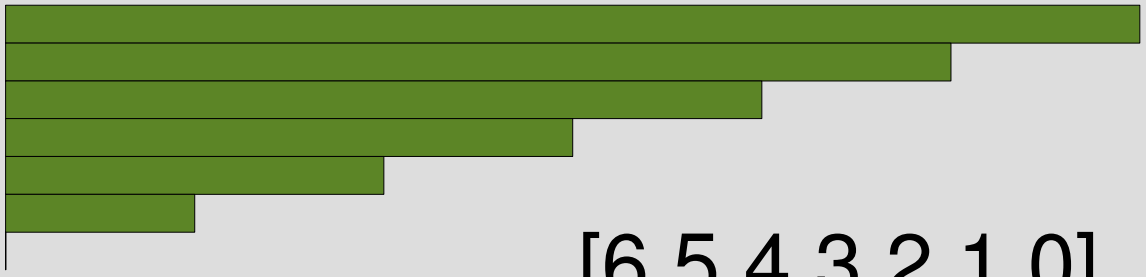


[:anysym]



[1]

[:rep, [:anysym]]



[6,5,4,3,2,1,0]

「任意の一文字」の実装 (1)

```
def try(exp, seq, pos, &block)
  ...
  when :anysym
    try_anysym(seq, pos, &block)
  ...
end
```

「任意の一文字」の実装 (2)

```
def try_anysym(seq, pos)
  if pos < seq.length
    yield pos + 1
  end
end
```

try_lit との比較

```
def try_anysym(seq, pos)
  if pos < seq.length
    yield pos + 1
  end
end
```

文字の等価性判定がない

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

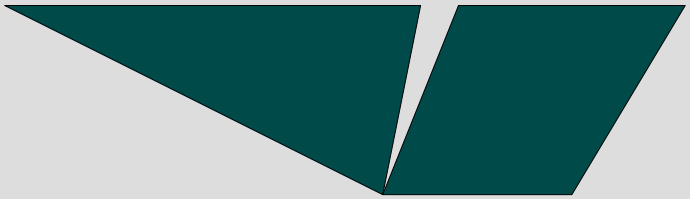
文字の等価性判定がある

文字列の先頭の位置 ¥A

- 文字列の最初にマッチするパターン [:string_start]
- Ruby の正規表現では ¥A
- このパターン自身は文字を消費しない
  先頭の文字の直前にマッチする
- matchstr([:string_start], "abc")　#=> [0]
- matchstr([:string_start], "")　　#=> [0]
- hasmatch([:cat, [:string_start], [:lit, "a"]], "abc")
  #=> 0
- hasmatch([:cat, [:string_start], [:lit, "b"]], "abc")
  #=> nil

「文字列の先頭の位置」のマッチ

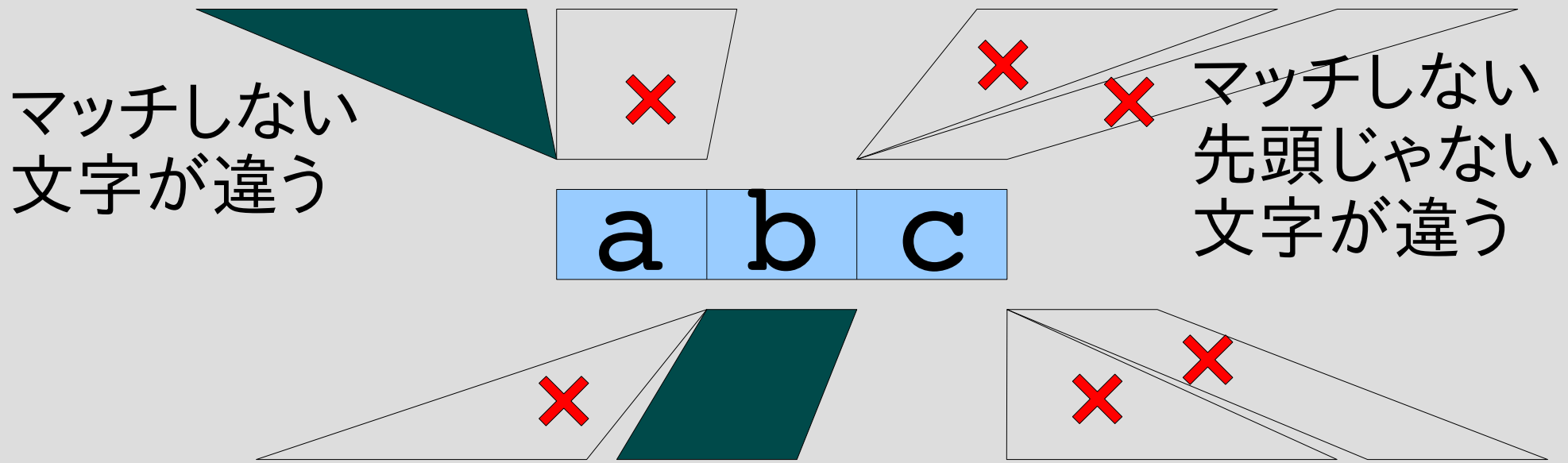
[:cat, [:string_start], [:lit, "a"]]　マッチする



a	b	c	d	e	f

「文字列の先頭の位置」のマッチ

hasmatch([:cat, [:string_start], [:lit, "b"]], "abc")



「文字列の先頭の位置」の実装

```
def try(exp, seq, pos, &block)
  ...
  when :string_start
    try_string_start(seq, pos, &block)
  ...
end
def try_string_start(seq, pos)
  yield pos if pos == 0
end
```

try_empseq との比較

```
def try_string_start(seq, pos)
  yield pos if pos == 0
end
def try_empseq(seq, pos)
  yield pos
end
```

位置の条件がある

位置の条件がない

matchstr と hasmatch

- [:string_start] を使うと hasmatch で matchstr に似た動作を行える
- ただし返り値は異なる
- matchstr(exp, str)
- hasmatch([:cat, [:string_start], exp], str)

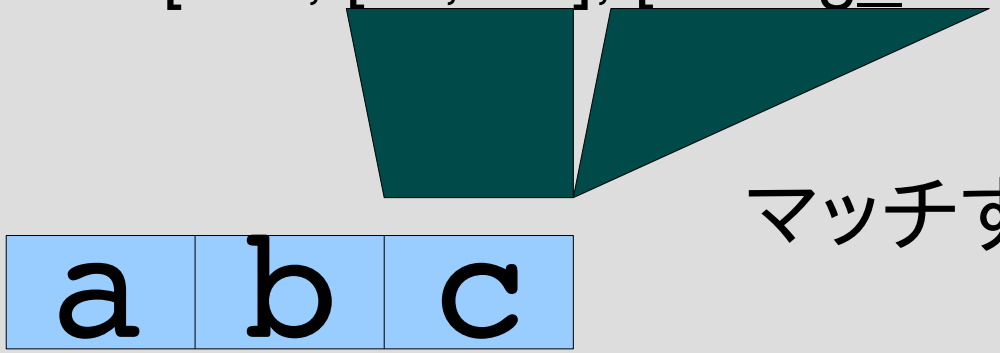
文字列の末尾の位置 ¥z

- 文字列の最後にマッチするパターン [:string_end]
- Ruby の正規表現では ¥z
- このパターン自身は文字を消費しない
  末尾の文字の直後にマッチする
- matchstr([:string_end], "abc")#=> []
- matchstr([:string_end], "")　　#=> [0]
- hasmatch([:cat, [:lit, "c"], [:string_end]], "abc")
  #=> 2
- hasmatch([:cat, [:lit, "b"], [:string_end]], "abc")
  #=> nil

「文字列の末尾の位置」のマッチ

hasmatch([:cat, [:lit, “c"], [:string_end]], “abc")

[:cat, [:lit, “c"], [:string_end]]



「文字列の末尾の位置」の実装

```
def try(exp, seq, pos, &block)
  ...
  when :string_end
    try_string_end(seq, pos, &block)
  ...
end
def try_string_end(seq, pos)
  yield pos if pos == seq.length
end
```

try_empseq との比較

```
def try_string_end(seq, pos)
  yield pos if pos == seq.length
end
def try_empseq(seq, pos)
  yield pos
end
```

位置の条件がある

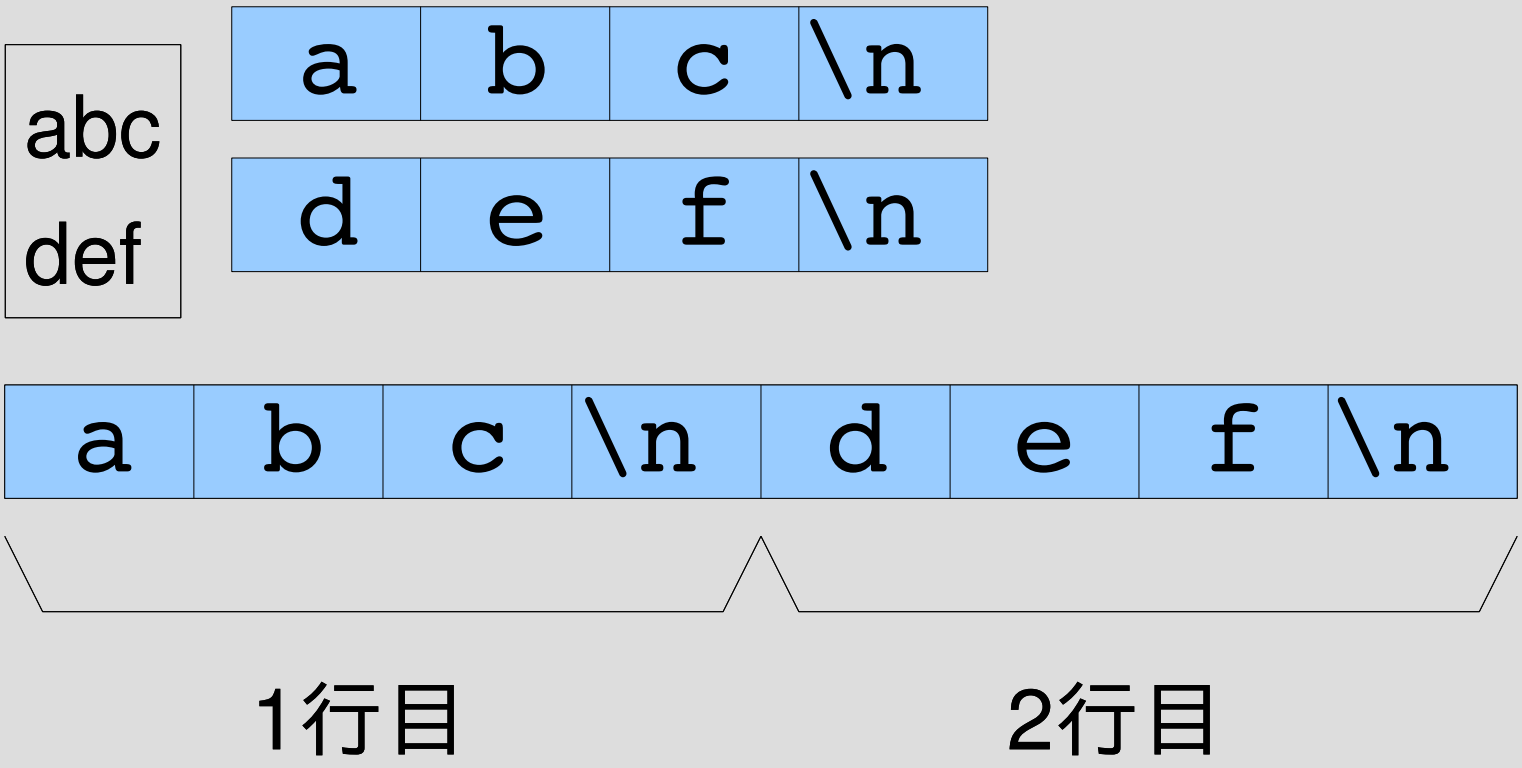
位置の条件がない

行頭 ^

- 行の先頭にマッチするパターン [:line_start]
- Ruby の正規表現では ^
- このパターン自身は文字を消費しない
- 文字列の先頭と改行の直後にマッチする

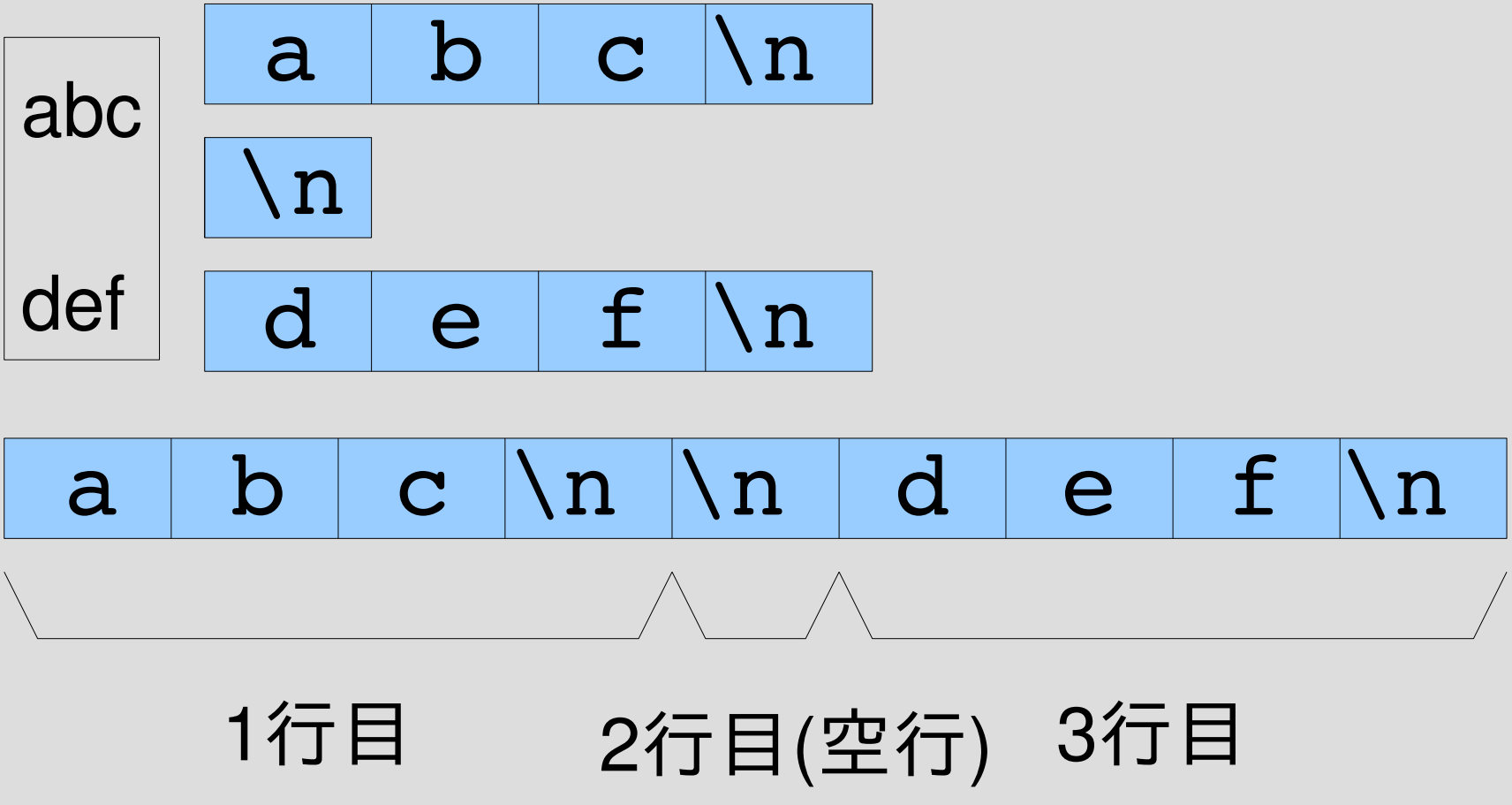
行の構造

- 文字列は文字の並び
- 文字のひとつに改行文字 ¥n というものがある
- 改行文字で行の終端を表す



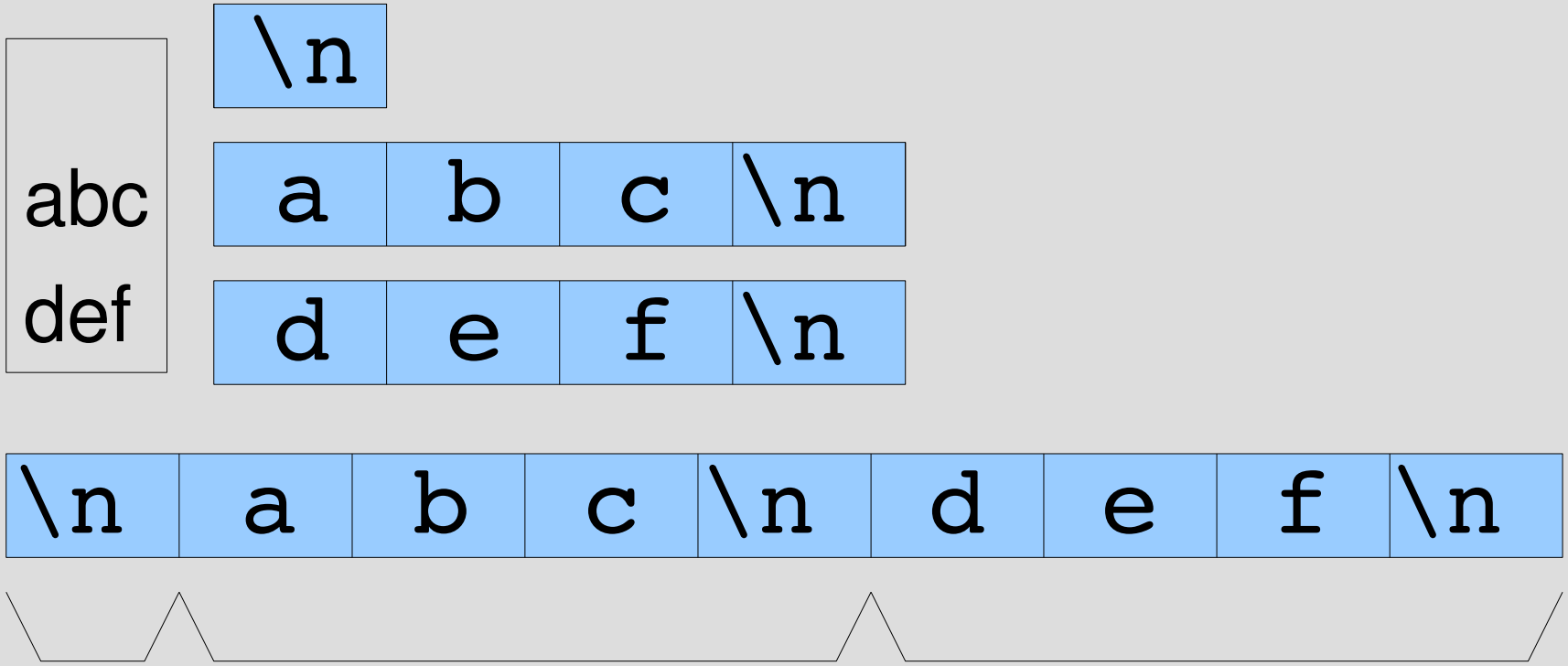
空行

- 改行以外に文字がない行が空行
- 改行が連続する



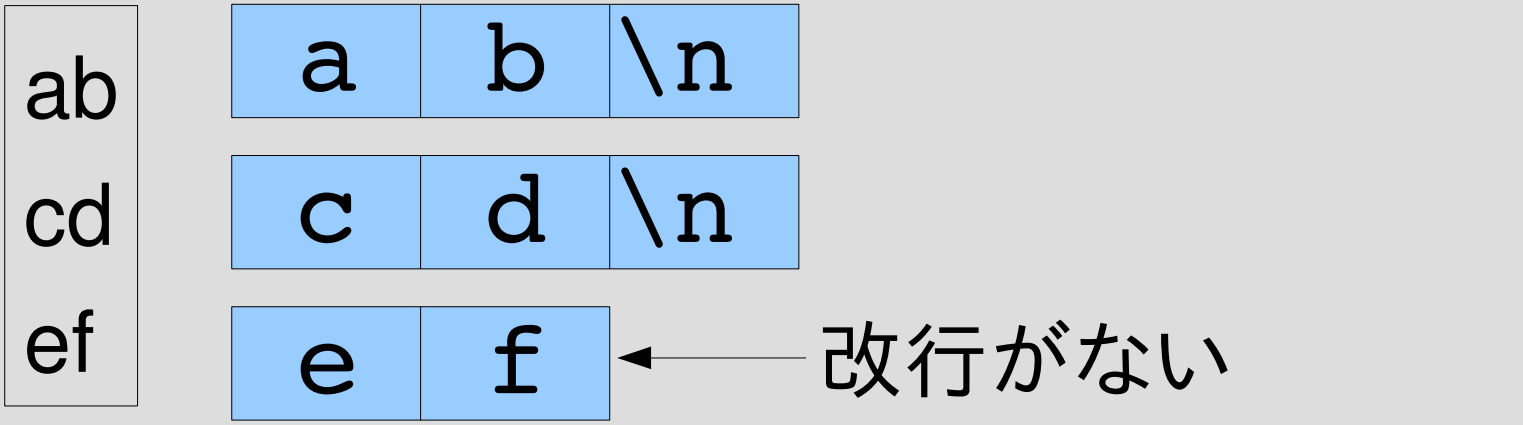
空行 (文字列先頭)

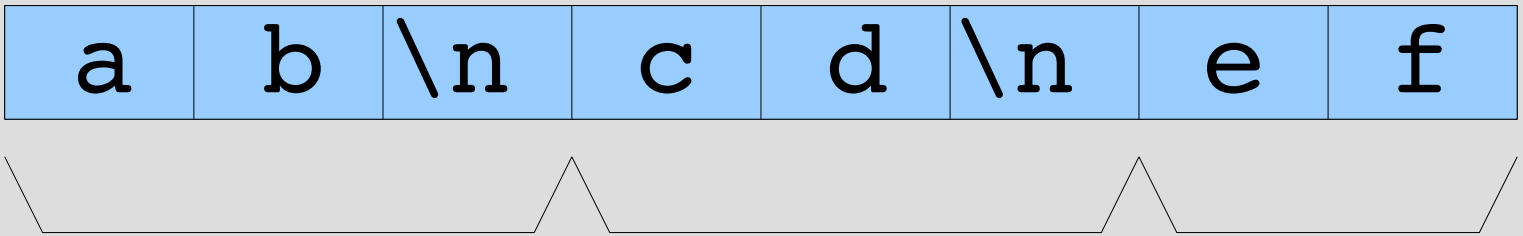
- 改行以外に文字がない行が空行
- 文字列の先頭に改行



不完全な行

- 文字列末尾の改行で終わらない行
- 文字列が改行で終わっているときには行がない とみなす (この場合不完全な行は存在しない)





1行目　　2行目　　3行目(不完全な行)

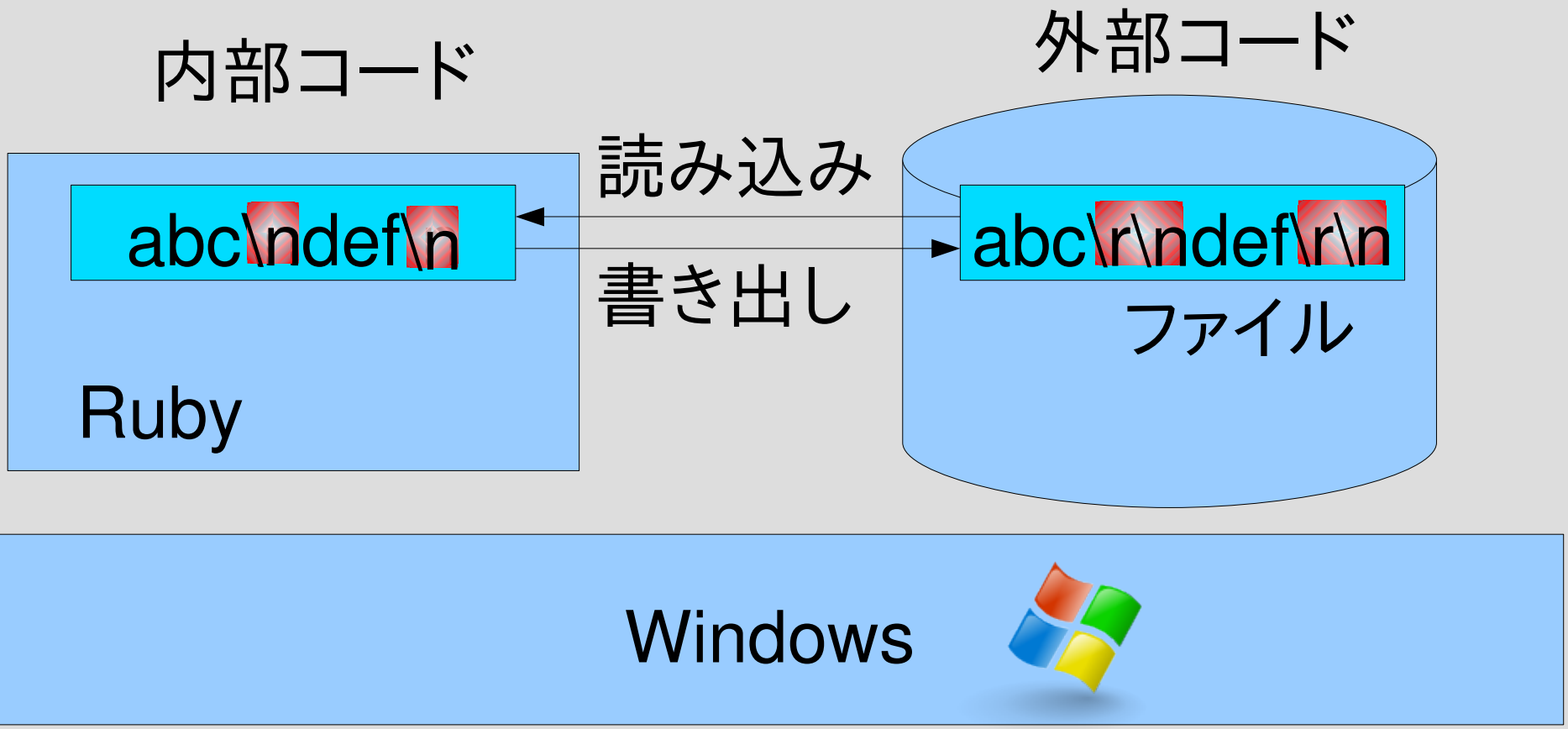
空文字列

- 空文字列 "" に行は入っているか？
- おそらく入っていないと考えるのが自然
  - 入っているとしたら、何行入っている？
- でも、伝統的に、行頭、行末は空文字列にマッチする
- この講義では伝統に従う

改行は環境依存

- 改行の標準的な表現は OS によって違う
  - Windows では¥r¥nの 2文字の並びがひとつの改行
  - Unix では¥nが改行
  - 昔のMacintoshでは¥rだった
  - Unicode にはIBMの大型機由来のとかも入っている
- 違いをいちいち気にするのは面倒なので、
  読み込むときに Unix の形式に変換し、
  書き出すときに環境依存の改行に変換する

内部コードと外部コード



テキストモードとバイナリモード

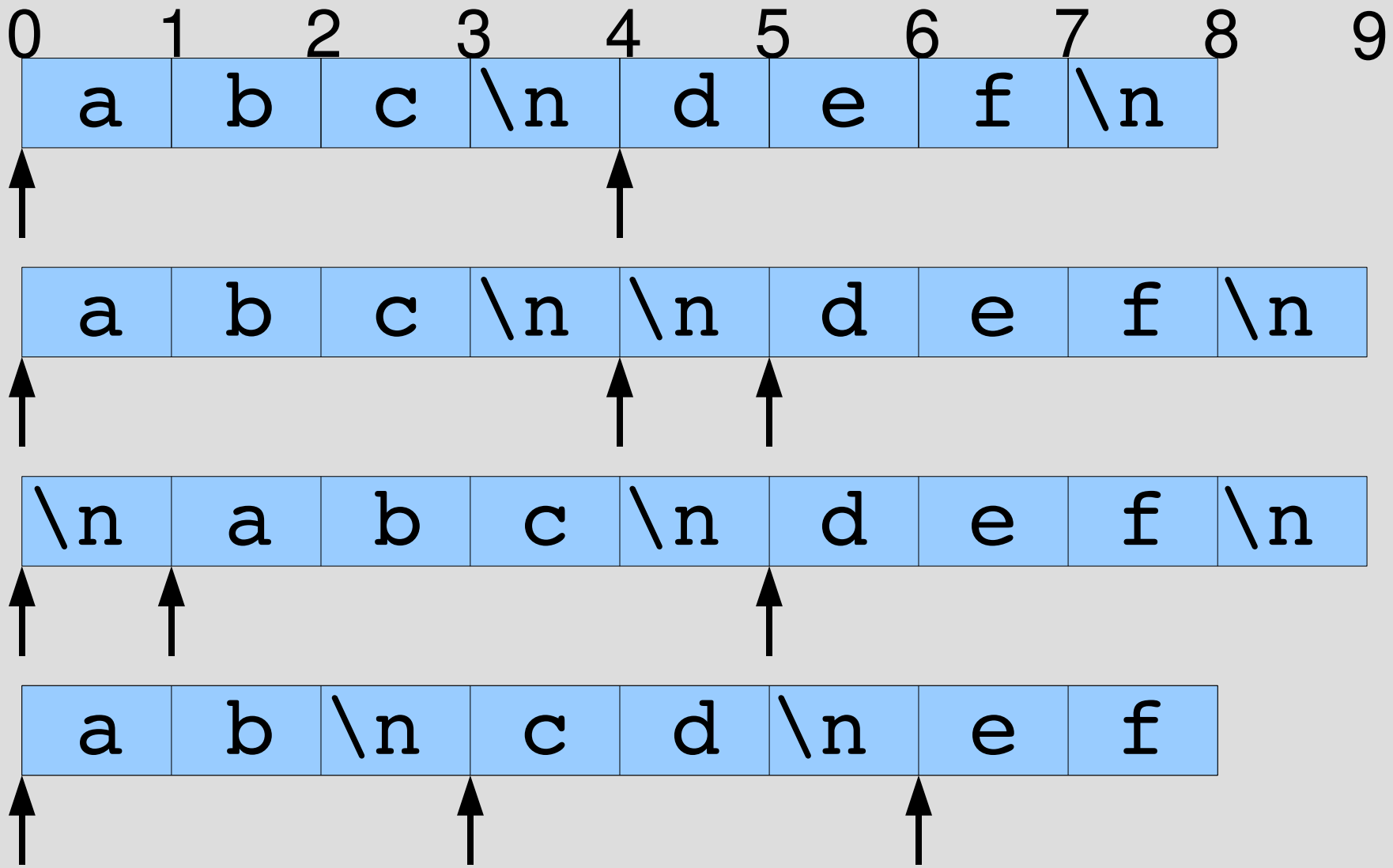
- 画像など、行構造ではないファイルを扱うときに変換が行われるとデータが壊れて困る
- 変換するかどうかのモードがある
- デフォルトは変換するテキストモード

- テキストモード
  - ファイルの読み書きで改行の変換を行う
- バイナリモード
  - ファイルの読み書きで改行の変換を行わない

行頭 ^

- 行の先頭にマッチするパターン [:line_start]
- Ruby の正規表現では ^
- このパターン自身は文字を消費しない
- 文字列の先頭と文字列末尾でない改行の直後にマッチする
- 改行は変換済みで、¥n になっているとする
- matchstr([:line_start], "abc") #=> [0]
- matchstr([:cat, [:rep, [:anysym]], [:line_start]], "a¥nb¥n") #=> [2,0]

行頭



「行頭」の実装

```
def try(exp, seq, pos, &block)
  ...
  when :line_start
    try_line_start(seq, pos, &block)
  ...
end
```

「行頭」の実装

```
def try_line_start(seq, pos)
  if pos == 0 || (pos < seq.length && seq[pos-1] == "\n"
    yield pos
  end
end
```

try_string_start との比較

条件が緩くなっている

```
def try_line_start(seq, pos)
  if pos == 0 || (pos < seq.length && seq[pos-1] == "\n")
    yield pos
  end
end

def try_string_start(seq, pos)
  yield pos if pos == 0
end
```

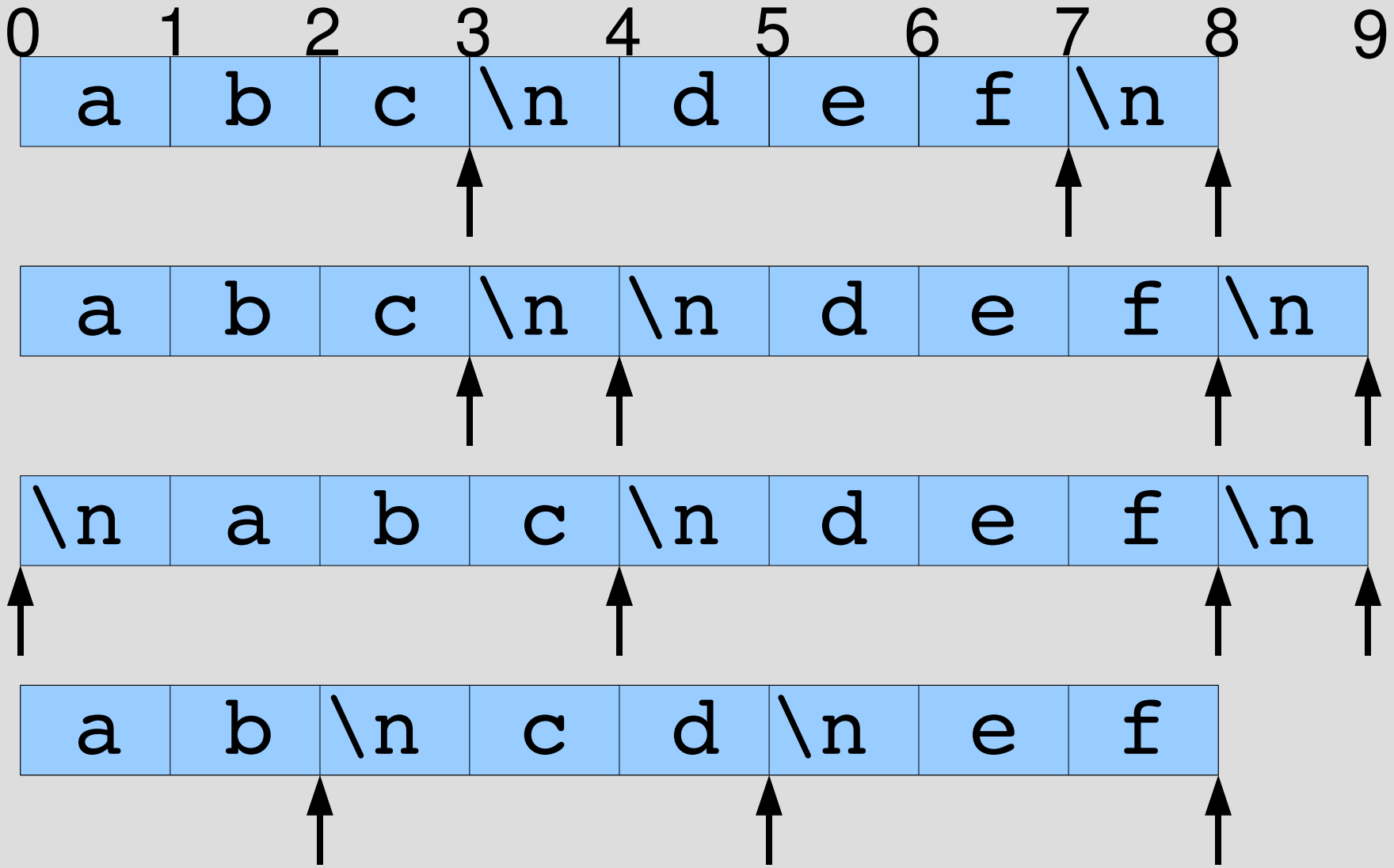
文字列の末尾でない

直前の文字が\n

行末 $

- 行の末尾にマッチするパターン [:line_end]
- Ruby の正規表現では $
- このパターン自身は文字を消費しない
- 文字列の最後と改行の直前にマッチする
- 文字列が改行で終わっていても文字列の最後にマッチする (歴史的慣習)

行末



レポート

- 文字列の絶対位置を指定する指示 abspos を実装して解説せよ
- たとえば [:abspos, 3] は位置3 を示す。
- [:abspos, 0] は [:string_start] と同じ
- ユニットテストを提供するので、実装したらテストして確認すること
- 〆切 2007-07-10 16:20
- HIPLUS
- 拡張子が txt なテキストファイルを望む

abspos の使用例

- hasmatch([:abspos, 3], "abcdef") #=> 3
- hasmatch(
  [:cat, [:lit, "c"], [:abspos, 3]], "abcdef")
  #=> 2
- matchstr(
  [:cat, [:rep, [:lit, "a"]], [:abspos, 3]], "aaaaa")
  #=> [3]

まとめ

- 前回のレポートの解説
- Ruby の =~ の動作にあわせた hasmatch を定義
- いくつか機能拡張
  - [:anysym]
  - [:string_start]
  - [:string_end]
  - [:line_start]
  - [:line_end]
- レポートを出した