テキスト処理 第12回 (2007-07-24)

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

- 前回のレポートの説明
- ルックアラウンドアサーション
- 含まない
- 試験について

ルックアラウンドアサーション

- lookaround assertion
- 「前後読み」と訳すこともあるが、定番の訳語はまだない
- Perl5 で導入された機能
- 「プログラミング Perl 第3版 volume I」にはカタカナで書いてあった

lookaround assertion

- 以下の 4つの総称
  - positive lookahead assertion: (?=e)
  - negative lookahead assertion: (?!e)
  - positive lookbehind assertion: (?<=e)
  - negative lookbehind assertion: (?<!e)
- 行頭 (^) や行末 ($) と同様に、特定の条件を満たしている空文字列にマッチする

特定の条件

- positive lookahead assertion: (?=e)
  - 空文字列の直後に e にマッチする文字列がある
  - 訳語: 先読みの肯定
- negative lookahead assertion: (?!e)
  - 空文字列の直後に e にマッチする文字列がない
  - 訳語: 先読みの否定
- positive lookbehind assertion: (?<=e)
  - 空文字列の直前に e にマッチする文字列がある
  - 訳語: 戻り読みの肯定、後読みの肯定
- negative lookbehind assertion: (?<!e)
  - 空文字列の直前に e にマッチする文字列がない
  - 訳語: 戻り読みの否定、後読みの否定

実行例
positive lookahead assertion

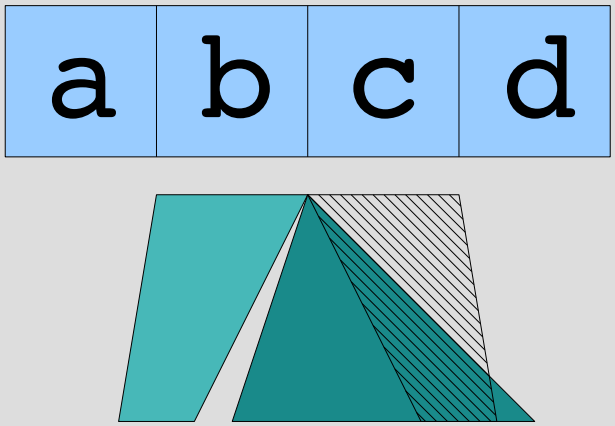
- p /b(?=c)/ =~ "abcd" #=> 1
  p $& #=> "b"
  p $~.pre_match #=> "a"
  p $~.post_match #=> "cd"
  b の後に c があるのでマッチする
  マッチした文字は b だけで、c は入っていない

- p /b(?=c)/ =~ "abxy" #=> nil
  b の後に c がないのでマッチしない

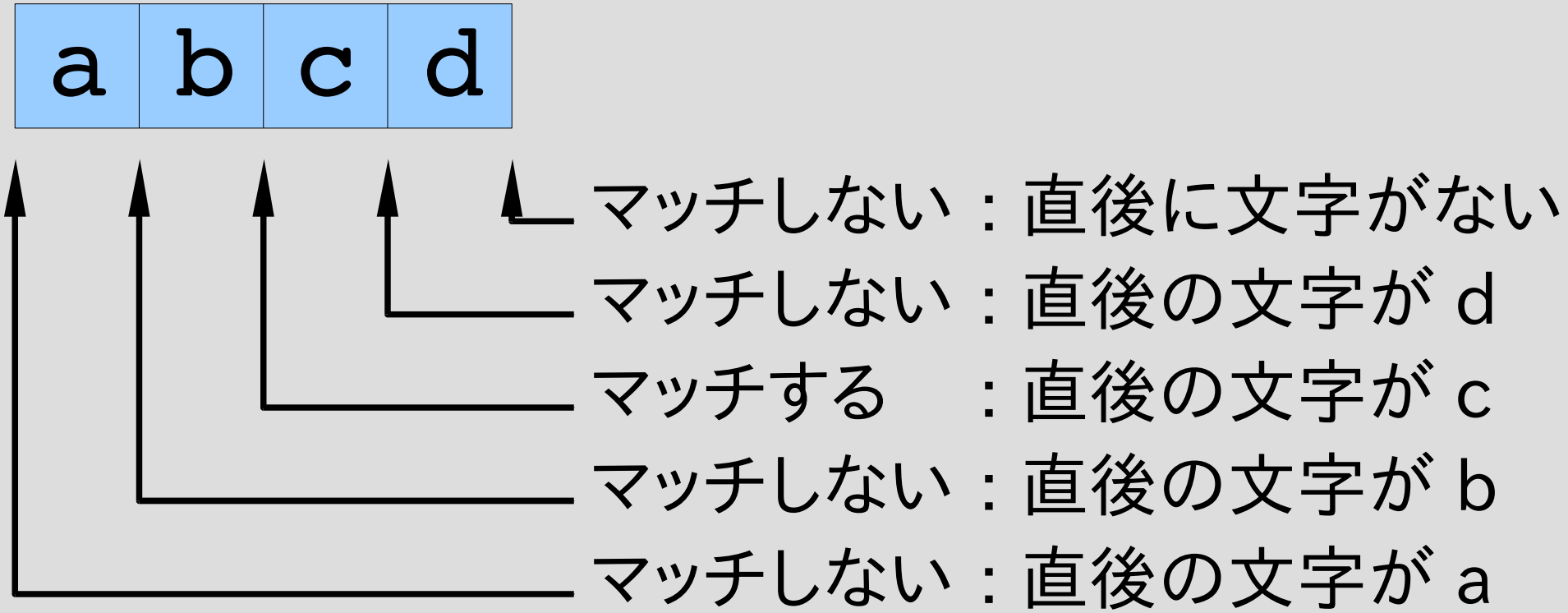
- p /b(?=c)/ =~ "ab" #=> nil
  b の後に c がないのでマッチしない

/b(?=c)/

- /b(?=c)/ は「直後に c が続く b」を意味する
- /bc/ とは、マッチする対象が b だけなところが違う



/(?=c)/ がマッチする空文字列



行末を $ を使わずに表現

- 行末とは、文字列の末尾と改行の直前のこと
- /¥z|(?=¥n)/ で行末を表現できる
- $ は行末専用だけど、(?=e) は他にも使える
- $ よりも positive lookahead assertion のほうが表現可能なことが多い
- まぁ、行末なら $ のほうがわかりやすく短くて使いやすいんだけど

文末の単語 /[A-Za-z]+(?=¥.)/

- 英語の文の末尾の単語
- ピリオドが直後にあるアルファベットの並び

正規表現エンジンに positive lookahead assertion を拡張

- /(?=e)/ に対応する抽象構文木は [:plookahead, e]

plookahead の実装 (1)

```
def try(exp, seq, pos, md, &block)
  ...
  when :plookahead
    _, e = exp
    try_plookahead(e, seq, pos, md, &block)
  ...
end
```

plookahead の実装 (2)

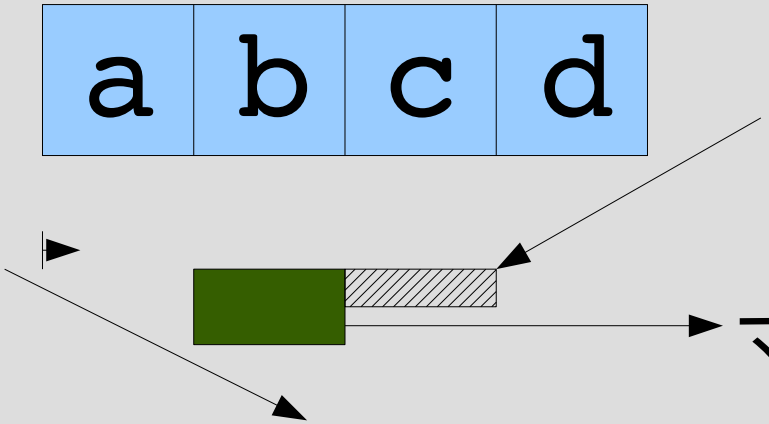
```
def try_plookahead(e, seq, pos, md)
  matched = nil
  try(e, seq, pos, md) {|pos2, md2|
    matched = md2
    break
  }
  yield pos, matched if matched
end
```

plookahead の実装 (2)

```
def try_plookahead(e, seq, pos, md)
  matched = nil       # 変数をを用意
  try(e, seq, pos, md) {|pos2, md2|     # e を挑戦
    matched = md2  # 最初の成功の md2 を記録
    break                # 最初の成功で脱出
  }
  yield pos, matched if matched    # 成功してたら
end                                # 最初の位置
                                   # を yield
```

せっかくマッチしたので、
マッチしたキャプチャを渡す

plookahead の動作 /b(?=c)/=~ "abcd"

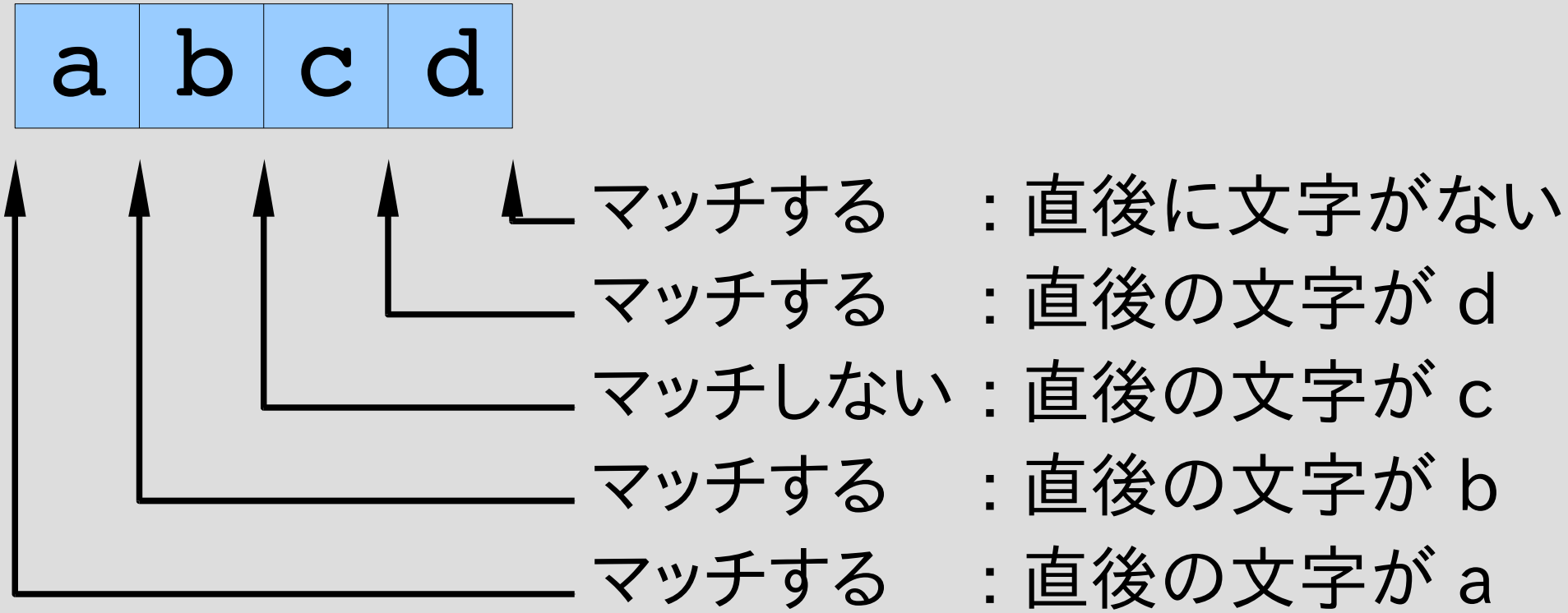


negative lookahead assertion (?!e)

- (?!e) は (?=e) がマッチしない空文字列にマッチする
- p /a(?!c)/ =~ "abcd"　　#=> 0
  p /b(?!c)/ =~ "abcd"　　#=> nil
  p /c(?!c)/ =~ "abcd"　　#=> 2
  p /d(?!c)/ =~ "abcd"　　#=> 3
  p /¥z(?!c)/ =~ "abcd"　　#=> 4

- 抽象構文木は [:nlookahead, e] とする

/(?!c)/ がマッチする空文字列



nlookahead の実装 (1)

```
def try(exp, seq, pos, md, &block)
  ...
  when :nlookahead
    _, e = exp
    try_nlookahead(e, seq, pos, md, &block)
  ...
end
```

nlookahead の実装 (2)

```
def try_nlookahead(e, seq, pos, md)
  matched = nil
  try(e, seq, pos, md) {|pos2, md2|
    matched = md2
    break
  }
  yield pos, md if !matched
end
```

plookahead と nlookahead の比較

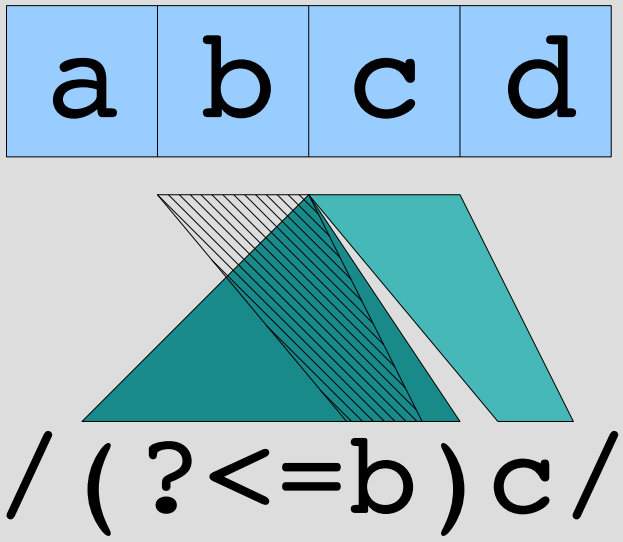
```
def try_plookahead(e,
                   seq, pos, md)
  matched = nil
  try(e, seq, pos, md) {|pos2, md2|
    matched = md2
    break
  }
  yield pos, matched if matched
end
```

```
def try_nlookahead(e,
                   seq, pos, md)
  matched = nil
  try(e, seq, pos, md) {|pos2, md2|
    matched = md2
    break
  }
  yield pos, md if !matched
end
```

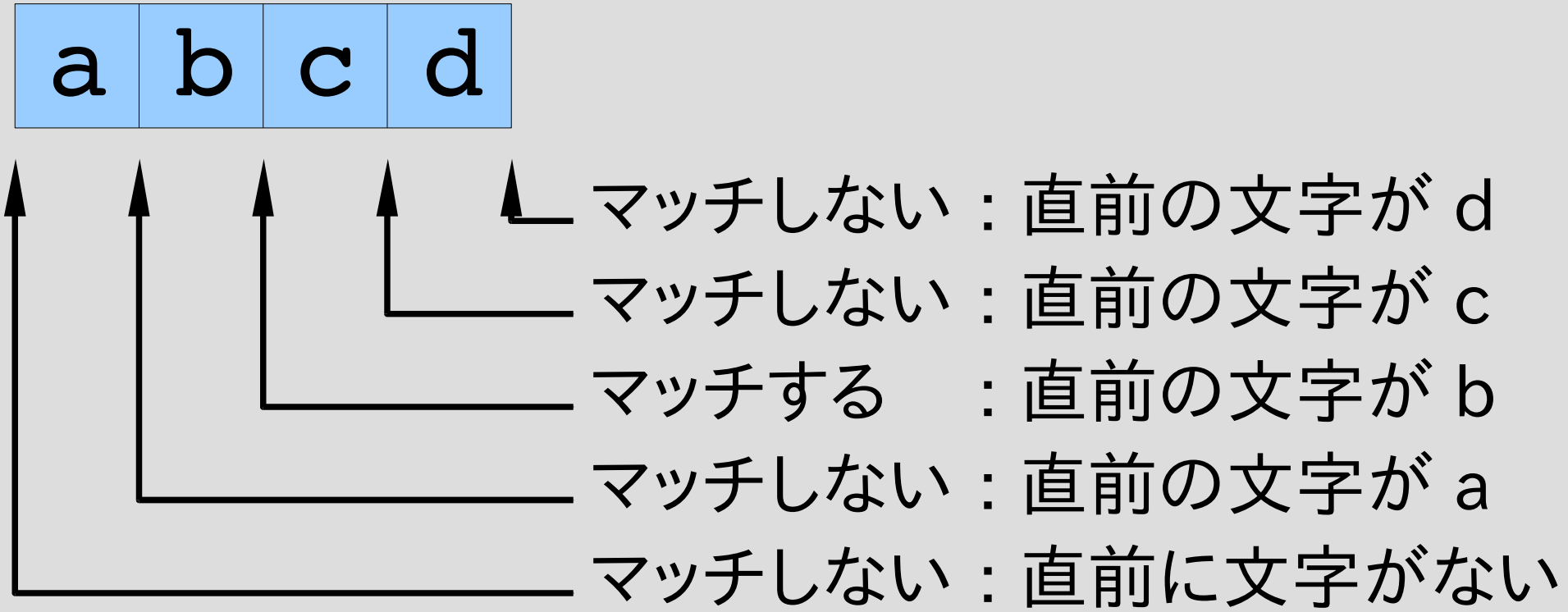
- yield する条件が逆
- e のマッチに成功していないので
  キャプチャ情報は渡さない (渡しようがない)

positive lookbehind assertion (?<=e)

- 空文字列の直前に e にマッチする文字列がある
- 抽象構文木は [:plookbehind, e]



/(?<=b)/ がマッチする空文字列



plookbehind の実装 (1)

```
def try(exp, seq, pos, md, &block)
  ...
  when :plookbehind
    _, e = exp
    try_plookbehind(e, seq, pos, md, &block)
  ...
end
```

try_plookbehind の実装に挑戦

```
def try_plookbehind(e, seq, pos, md)
  matched = nil
  try(e, seq, どこから?, md) {|pos2, md2|
    if pos2 == pos     # 終了位置が pos だったら
      matched = md2
      break
    end
  }
  yield pos, matched if matched
end
```

try でどこから始めるか?

try でどこから始めるか?

- e にマッチする長さがわからないのでマッチ開始位置もわからない
- 考えられる対策
  - 0〜pos の範囲をぜんぶ試す
    - 遅い
  - マッチする長さが固定のパターンしか許さない
    - perl, ruby1.9
    - 左右非対称な制約は美しくない
  - 逆方向にマッチを進める
    - gauche
    - 非対称な制約がなくて美しい
    - それでも微妙に変わってしまう: (?<=a*a*) とか

長さ固定

- ここではマッチする長さが固定のパターンしか許さないことにする
  - 新しい再帰関数が必要で教材として面白い
  - 逆方向の実装に try 全体をコピーするのは面白くない。コピーしないで済ますのは講義時間に収まらない
- パターンがマッチする長さを求める関数 pat_matchlen をつくる
- 長さが固定でないとき pat_matchlen は nil を返す
- try の開始位置は pos - pat_matchlen(e) とする

plookbehind の実装 (2)

```
def try_plookbehind(e, seq, pos, md)
  s = pat_matchlen(e)
  raise "長さが固定でない" if !s
  return if pos-s < 0     # 直前に十分な長さがあるか
  matched = nil
  try(e, seq, pos-s, md) {|pos2, md2|
    # pos2 == pos のはずなので検査は省略
    matched = md2
    break
  }
  yield pos, matched if matched
end
```

raise は例外を発生させる (エラー処理)

pat_matchlen 実装方針

- try と同様に抽象構文木を再帰的にたどる

- empseq だったら 0 (string_start とかも)
- lit だったら 1 (anysym も)
- cat だったら両方の和
- alt だったら、両方が一致しればそれ
- rep だったら、中身が 0 なら 0
- capture だったら中身と同じ
- times だったら、n=m だったら中身の n倍

pat_matchlen 実装 (1)

```
def pat_matchlen(exp)
  case exp[0]
  when :empseq, :string_start, :string_end,
            :line_start, :line_end,
            :plookahead, :nlookahead,
            :plookbehind, :nlookbehind
    0
...
```

- empseq, string_start, ... なら 0 を返す
- lookaround 自身もマッチする長さは 0

pat_matchlen 実装 (2)

```
when :lit, :anysym
  1
...
```

- lit, anysym なら 1 を返す

pat_matchlen 実装 (3) cat

```
  when :cat
    _, e1, e2 = exp
    s1 = pat_matchlen(e1)
    s2 = pat_matchlen(e2)
    if s1 && s2
      s1 + s2
    else
      nil
    end
...
```

- cat で、e1, e2 が両方とも固定長なら和を返す
- そうでなければ nil を返す

pat_matchlen 実装 (4) alt

```
  when :alt
    _, e1, e2 = exp
    s1 = pat_matchlen(e1)
    s2 = pat_matchlen(e2)
    if s1 && s2 && s1 == s2
      s1
    else
      nil
    end
...
```

- alt で、e1, e2 が両方とも固定長かつ等しければそれを返す
- そうでなければ nil を返す

pat_matchlen 実装 (5) rep

```
  when :rep, :opt, :plus, ¥
          :rep_lazy, :opt_lazy, :plus_lazy
    _, e = exp
    s = pat_matchlen(e)
    if s == 0
      0
    else
      nil
    end
...
```

- rep などの繰り返しで、e が固定長かつその長さが 0 であれば 0 を返す
- そうでなければ nil を返す

pat_matchlen 実装 (6) times

```
  when :times, :times_lazy
    _, e, m, n = exp
    s = pat_matchlen(e)
    if s && (s == 0 || m == n)
      s * m
    else
      nil
    end
...
```

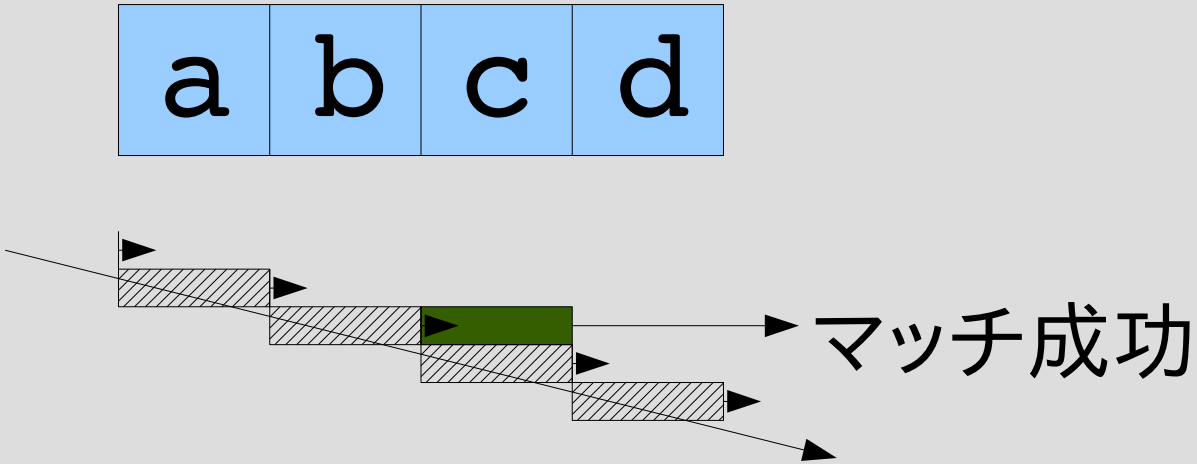
- times による e が固定長な繰り返しで、以下のいずれかであれば 0 を返す
  - 繰り返し回数が一定 (m=n)
  - e の長さが 0
- そうでなければ nil を返す

pat_matchlen 実装 (7) capture

```
  when :capture
    _, n, e = exp
    pat_matchlen(e)
  end
end
```

- capture で e が固定長ならそれを返す
- そうでなければ nil を返す

plookbehind の動作
/(?<=b)c/ =~ "abcd"

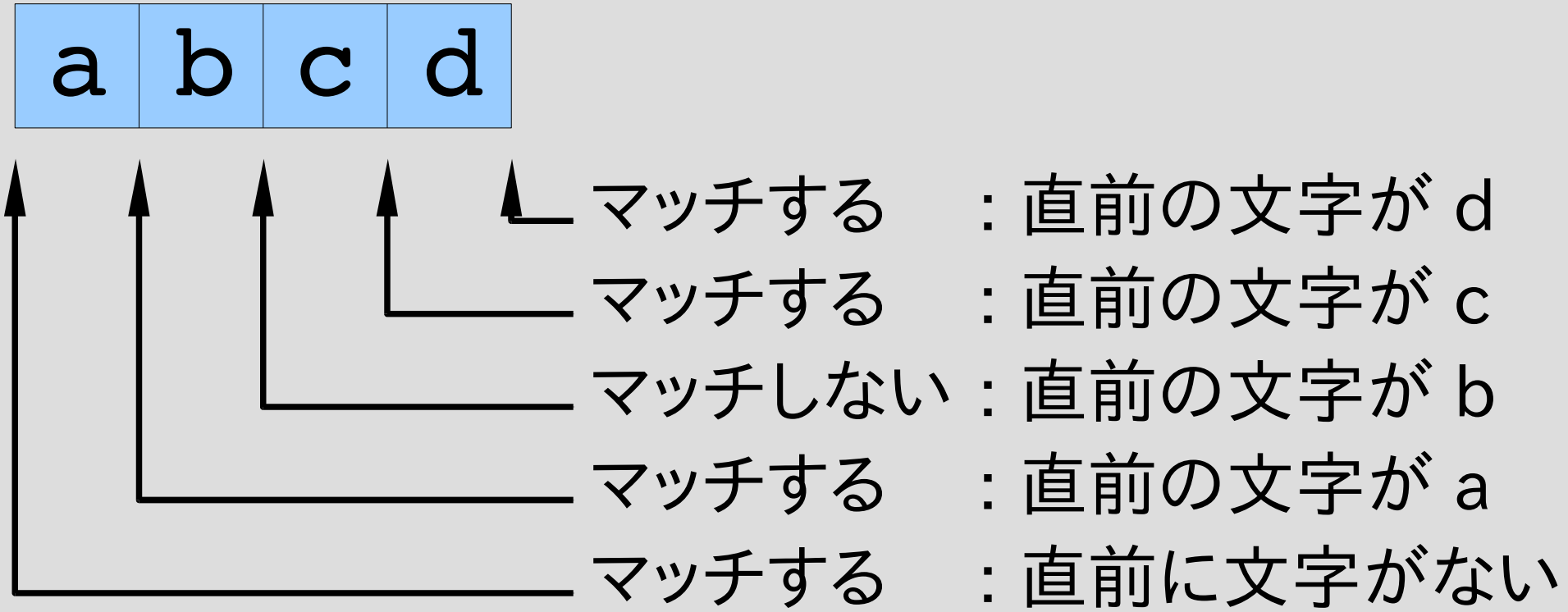


negative lookbehind assertion (?<!e)

- (?<!e) は (?<=e) がマッチしない空文字列にマッチする
- p /(?<!b)¥A/ =~ "abcd" #=> 0
  p /(?<!b)a/ =~ "abcd" #=> 0
  p /(?<!b)b/ =~ "abcd" #=> 1
  p /(?<!b)c/ =~ "abcd" #=> nil
  p /(?<!b)d/ =~ "abcd" #=> 3

- 抽象構文木は [:nlookabehind, e] とする

/(?<!b)/ がマッチする空文字列



nlookbehind の実装 (1)

```
def try(exp, seq, pos, md, &block)
  ...
  when :nlookbehind
    _, e = exp
    try_nlookbehind(e, seq, pos, md, &block)
  ...
end
```

nlookbehind の実装 (2)

```
def try_nlookbehind(e, seq, pos, md)
  s = pat_matchlen(e)
  raise "長さが固定でない" if !s
  return if pos-s < 0
  matched = nil
  try(e, seq, pos-s, md) {|pos2, md2|
    matched = md2
    break
  }
  yield pos, md if !matched
end
```

plookbehind と nlookbehind の比較

```
def try_plookbehind(e,
    seq, pos, md)
  s = pat_matchlen(e)
  raise "長さが固定でない" if !s
  return if pos-s < 0
  matched = nil
  try(e, seq, pos-s, md) {|pos2, md2|
    matched = md2
    break
  }
  yield pos, matched if matched
end
```

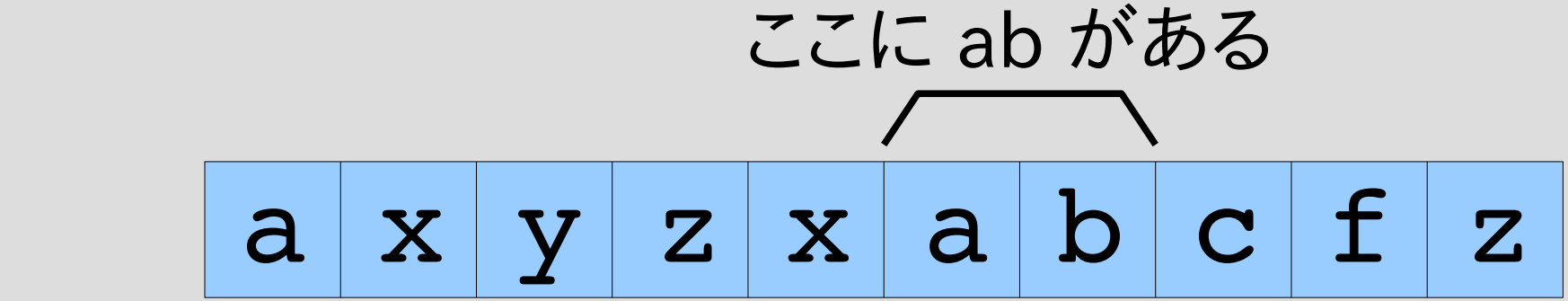
```
def try_nlookbehind(e,
    seq, pos, md)
  s = pat_matchlen(e)
  raise "長さが固定でない" if !s
  return if pos-s < 0
  matched = nil
  try(e, seq, pos-s, md) {|pos2, md2|
    matched = md2
    break
  }
  yield pos, md if !matched
end
```

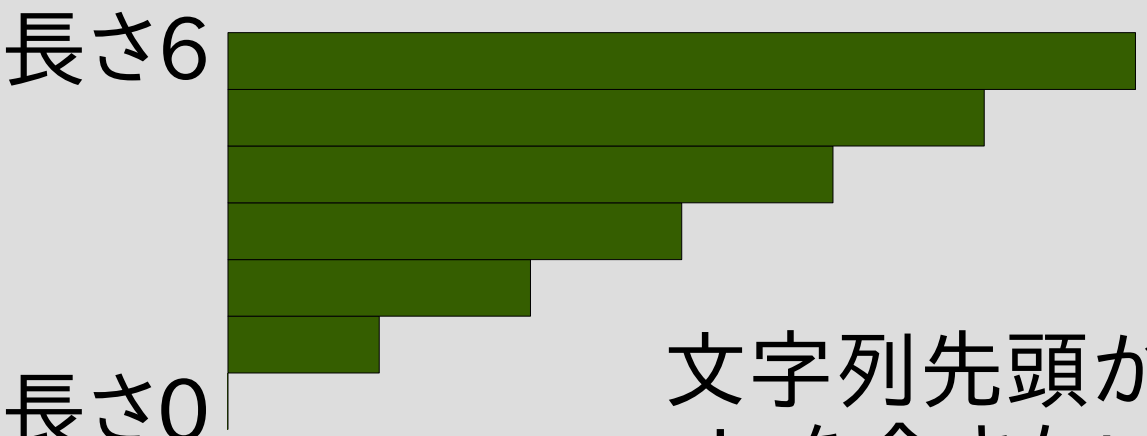
- yield する条件が逆
- e のマッチに成功していないので
  キャプチャ情報は渡さない (渡しようがない)

「含まない」

- C のコメントを正規表現で書くのはたいへん
- */ を含まない文字列、というのが難しい
- ある正規表現にマッチする文字列を「含まない」文字列にマッチする、という機能を拡張する
- それがあれば ¥/¥*「*/ を含まない文字列」¥*¥/ というように記述できる
- 既存の正規表現エンジンはこの機能をもっていない (知っている範囲では)
- 抽象構文木で [:absent, e] とする

ab を含まない文字列

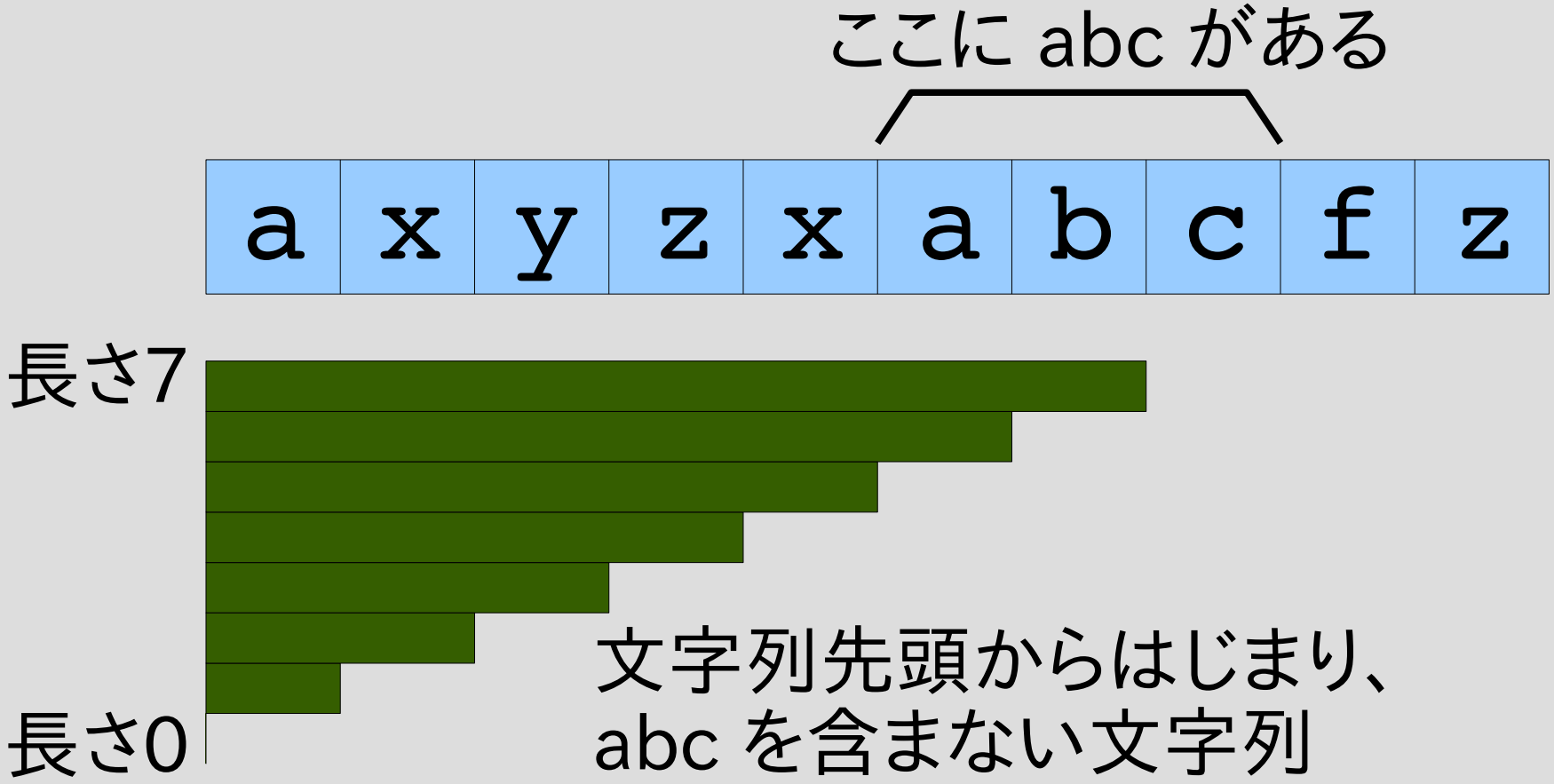




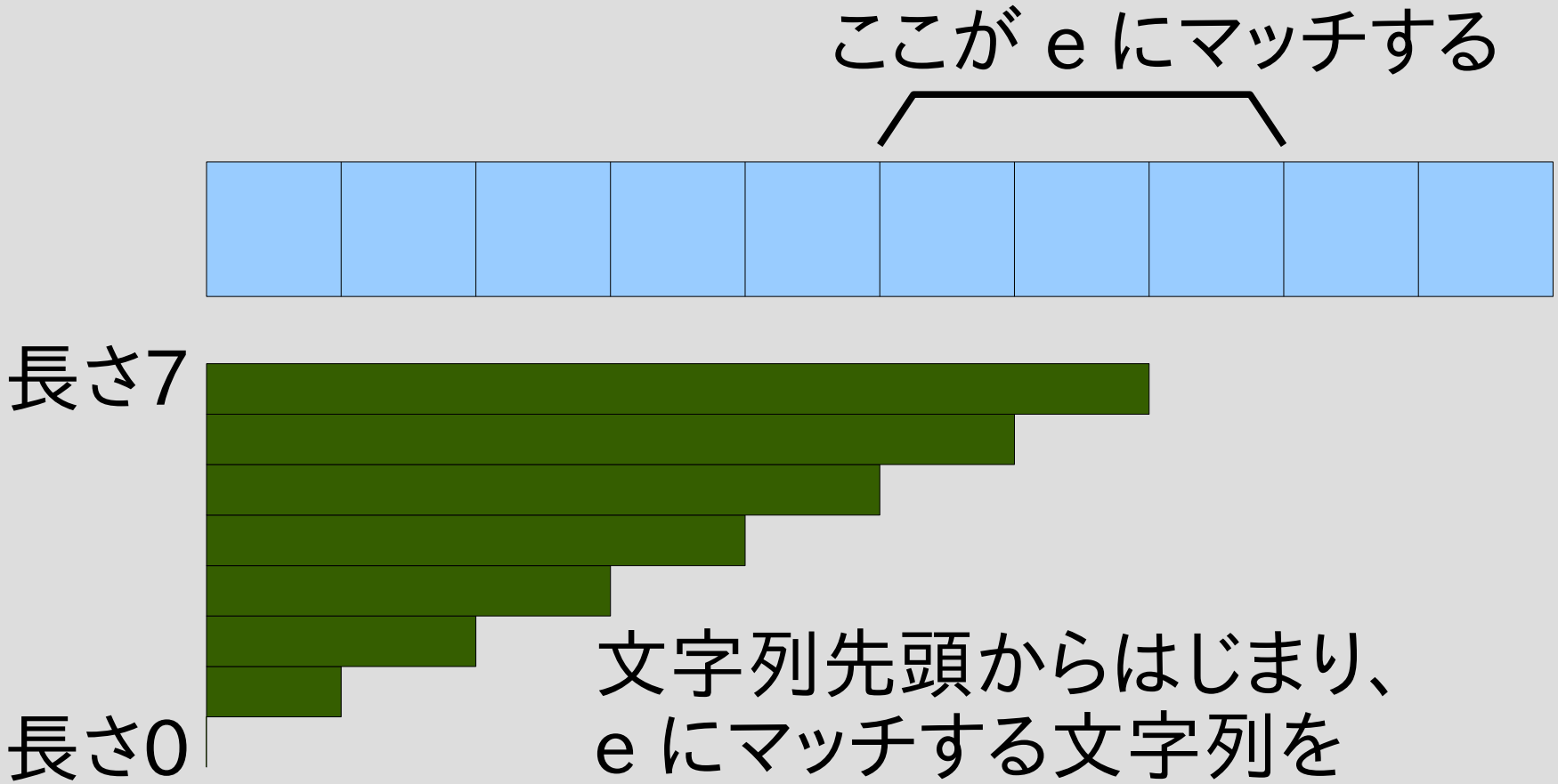
文字列先頭からはじまり、
ab を含まない文字列

rep と似た形だが、繰り返しの終端位置が違う

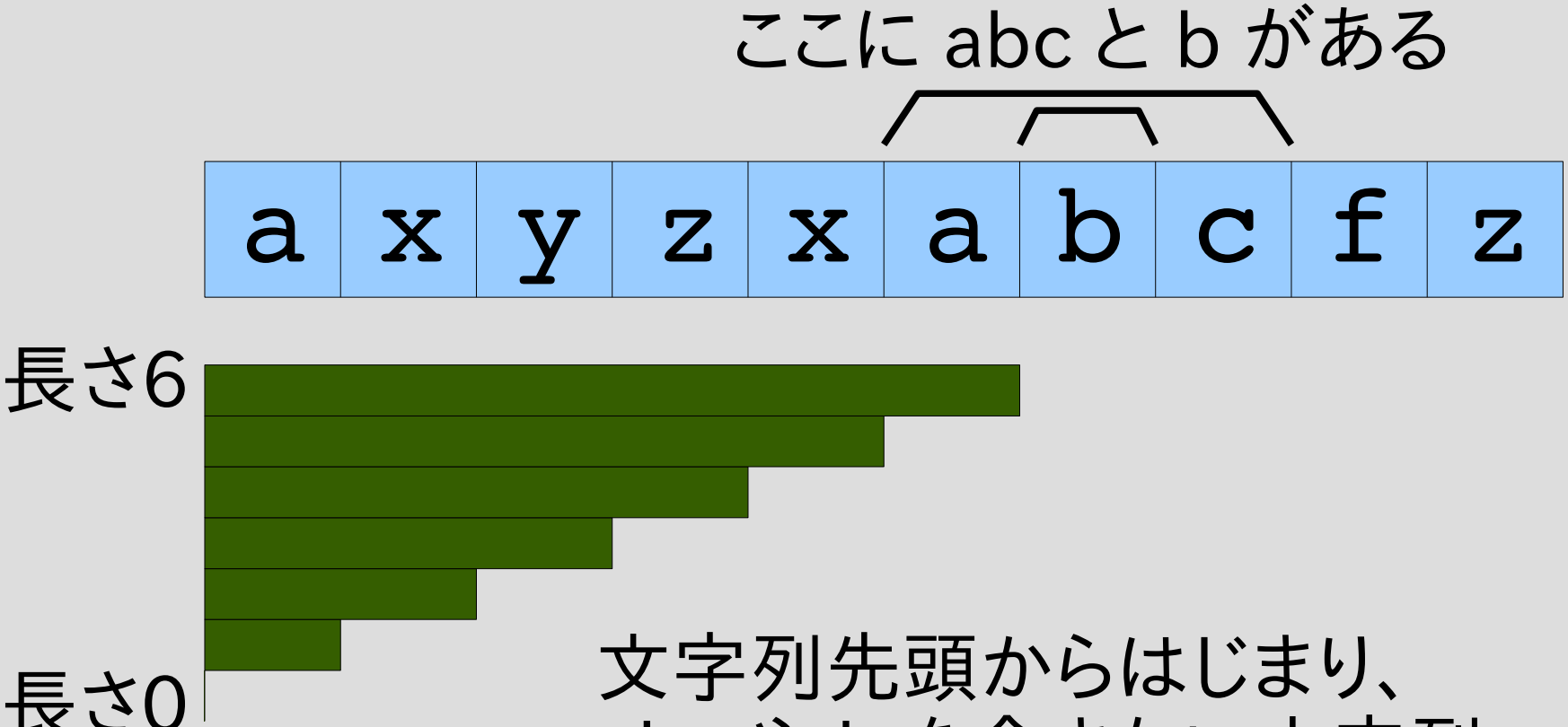
abc を含まない文字列



/e/ を含まない文字列


ここが e にマッチする
長さ7
長さ0
文字列先頭からはじまり、
e にマッチする文字列を
含まない文字列
マッチ終端のひとつ前まで

/abc|b/ を含まない文字列



文字列先頭からはじまり、
abc や b を含まない文字列

マッチ終端のうち、最左のひとつ前まで

[:absent, e] の実装 (1)

```
def try(exp, seq, pos, md, &block)
...
  when :absent
    _, e = exp
    try_absent(e, seq, pos, md, &block)
...
end
```

[:absent, e] の実装 (2)

```
def try_absent(e, seq, pos, md)
  limit = seq.length
  pos2 = pos
  while pos2 <= limit
    try(e, seq, pos2, md) {|pos3, md3|
      limit = pos3-1 if pos3-1 < limit
    }
    pos2 += 1
  end
  limit.downto(pos) {|pos4| yield pos4, md}
end
```

どこまで伸ばしていいか調べる

```
def try_absent(e, seq, pos, md)
  limit = seq.length
  pos2 = pos
  while pos2 <= limit
    try(e, seq, pos2, md) {|pos3, md3|
      limit = pos3-1 if pos3-1 < limit
    }
    pos2 += 1
  end
  limit.downto(pos) {|pos4| yield pos4, md}
end
```

どこまで伸ばしていいか調べる

```
def try_absent(e, seq, pos, md)
  limit = seq.length      # マッチしないなら最後まで
  pos2 = pos
  while pos2 <= limit # 右にずらしながらマッチ探索
    try(e, seq, pos2, md) {|pos3, md3|
      limit = pos3-1 if pos3-1 < limit # マッチ終端の
    }                                  # ひとつ前
    pos2 += 1
  end
  limit.downto(pos) {|pos4| yield pos4, md}
end
```

わかった範囲で **yield**

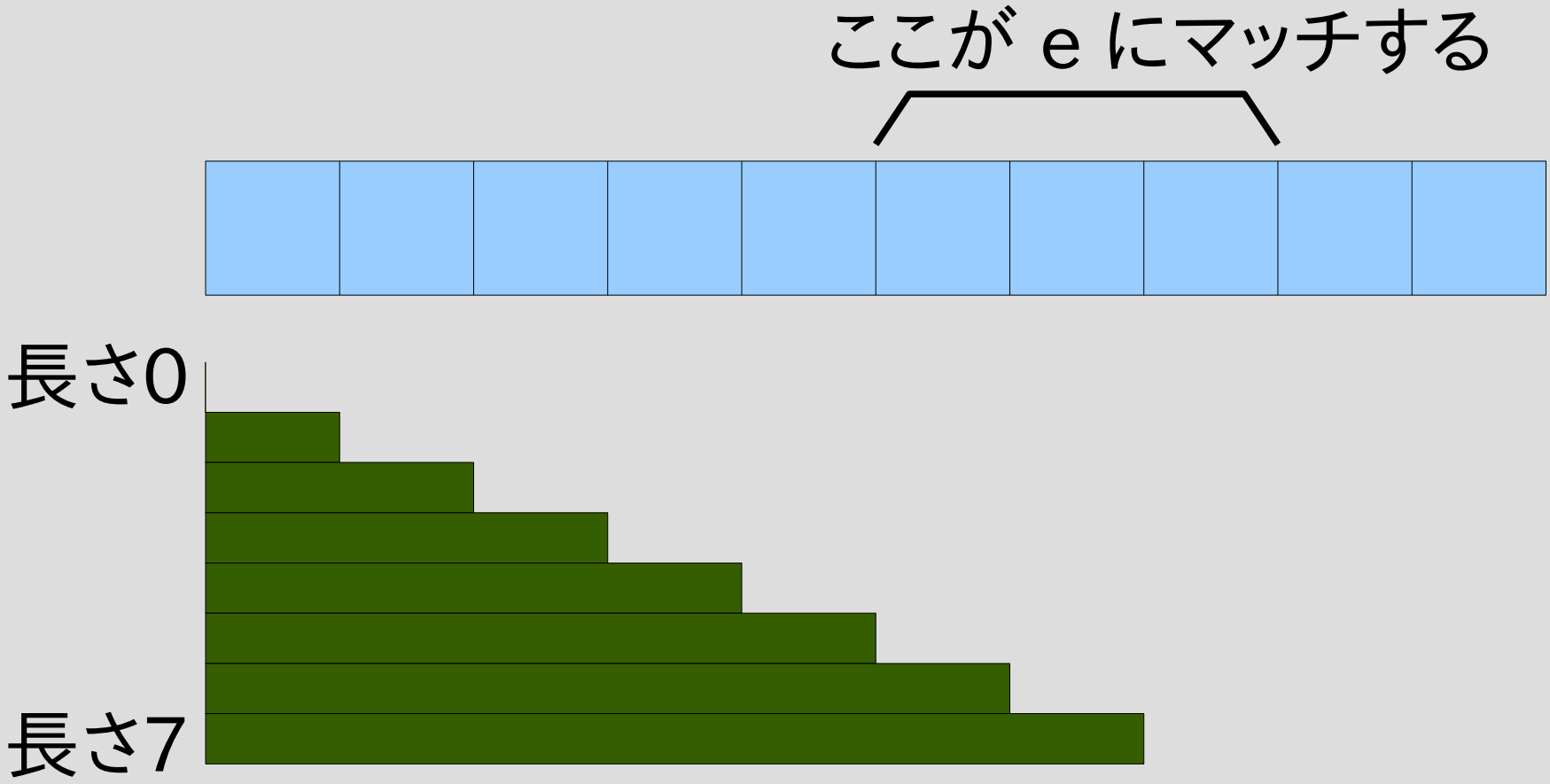
```
def try_absent(e, seq, pos, md)
  limit = seq.length
  pos2 = pos
  while pos2 <= limit
    try(e, seq, pos2, md) {|pos3, md3|
      limit = pos3-1 if pos3-1 < limit
    }
    pos2 += 1
  end
  limit.downto(pos) {|pos4| yield pos4, md}
end
```

downto は upto と逆で降順に整数を生成

absent_lazy

- absent の lazy 版
- 短い方から長い方へ

/e/ を含まない文字列



マッチ終端のうち、最左のひとつ前まで

[:absent_lazy, e] の実装 (1)

```
def try(exp, seq, pos, md, &block)
...
  when :absent_lazy
    _, e = exp
    try_absent_lazy(e, seq, pos, md, &block)
...
end
```

[:absent_lazy, e] の実装 (2)

```
def try_absent_lazy(e, seq, pos, md)
  limit = seq.length
  pos2 = pos
  while pos2 <= limit
    try(e, seq, pos2, md) {|pos3, md3|
      limit = pos3-1 if pos3-1 < limit
    }
    yield pos2, md if pos2 <= limit
    pos2 += 1
  end
end
```

探している途中で **yield**

```
def try_absent_lazy(e, seq, pos, md)
  limit = seq.length
  pos2 = pos
  while pos2 <= limit
    try(e, seq, pos2, md) {|pos3, md3|
      limit = pos3-1 if pos3-1 < limit
    }
    yield pos2, md if pos2 <= limit
    pos2 += 1
  end
end
```

探し終わらなくても
limit はもう pos2 未満にはならない
ことがわかるので yield できる

後からやってもいけなくはない

```
def try_absent_lazy(e, seq, pos, md)
  limit = seq.length
  pos2 = pos
  while pos2 <= limit
    try(e, seq, pos2, md) {|pos3, md3|
      limit = pos3-1 if pos3-1 < limit
    }
    pos2 += 1
  end
  pos.upto(limit) {|pos4| yield pos4, md}
end
```

try_absent から順序だけ変更

C のコメントにマッチするパターン

```
[:cat, [:cat, [:lit, "/"], [:lit, "*"]],
    [:cat, [:absent, [:cat, [:lit, "*"], [:lit, "/"]]],
     [:cat, [:lit, "*"], [:lit, "/"]]]]
```

/* にマッチするパターン
[:cat, [:lit, "/"], [:lit, "*"]]

*/ を含まないパターン
[:absent,[:cat, [:lit, "*"], [:lit, "/"]]]

*/ にマッチするパターン
[:cat, [:lit, "*"], [:lit, "/"]]

連結

試験

- 2007-07-31 (火) 5時限 16:30 〜 17:30
- 125教室 (講義の教室とは異なるので注意)
- 持込: 一切可
- 学生証を携帯すること
- このあたりの情報は以下でも確認できる
  専修大学ポータル → ライブラリ →【教務課】定期試験関連情報 → 前期試験時間割等（ネットワーク情報学部）

- 試験はレポート 2回相当程度の予定
- 試験の重みは低いので、レポートをまともに出していない限り単位は難しい

試験問題

1. 正規表現 X を講義で述べた抽象構文木に変換せよ
2. 上で変換した抽象構文木を e として、講義で述べた matchstr を matchstr(e, Y) として実行した結果の値を示せ
3. matchstr(e, Y) 内部の正規表現エンジンの動作を解説せよ

- X, Y は試験および追試で変化する

まとめ

- レポート解説
- ルックアラウンドアサーション
  - plookahead: (?=e)
  - nlookahead: (?!e)
  - plookbehind: (?<=e)
  - nlookbehind: (?<!e)
- 正規表現を含まない正規表現
  - absent
  - absent_lazy
- 試験について