

# テキスト処理 第5回 (2007-05-22)

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](mailto:akr@isc.senshu-u.ac.jp)

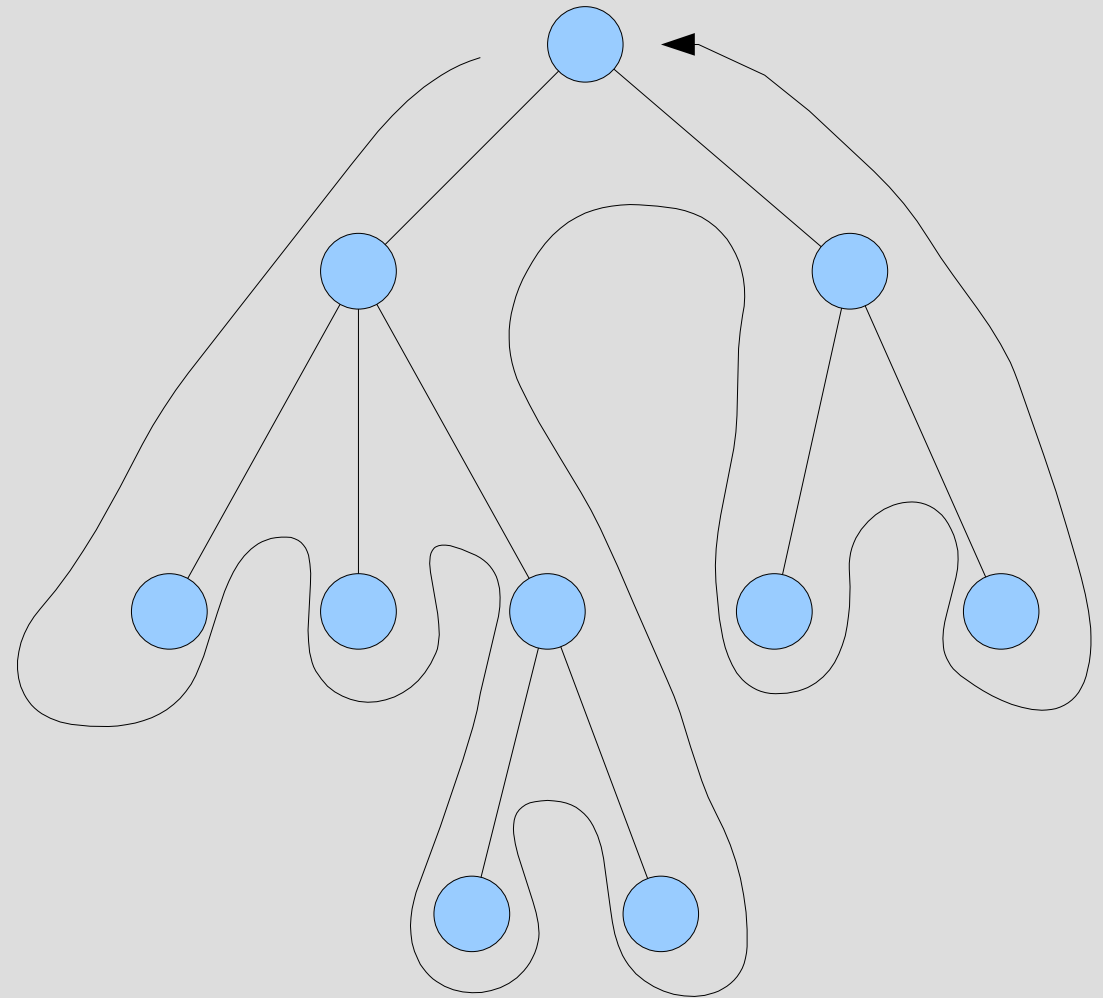
<http://staff.aist.go.jp/tanaka-akira/textprocess-2007/>

# 今日の内容

- 木構造に対する再帰
- 数式の評価
- レポート

# 木構造に対する再帰

- 木構造をたどる



# 木構造内の整数の和

- 配列内の整数の和を求める: sum
- ただし、配列はネストしていることもある
- `sum([1,2,3])` #=> 6
- `sum([1,[2,3]])` #=> 6
- `sum([1,[[[2]]],3])` #=> 6
- `sum([1,[2,3],[[5], 6, [7]])` #=> 24

# sumの定義

- ```
def sum(obj)
  if obj.respond_to? :each
    s = 0
    obj.each {|v| s += sum(v) }
    s
  else
    obj
  end
end
```

 ← 再帰

# Object#respond\_to?

- オブジェクトにメソッドがあるか調べる
- `obj.respond_to?(:each)` は `obj` に `each` メソッドがあるときに真になる

# sum の respond\_to?

- def sum(obj)

- if `obj.respond_to? :each`

- s = 0

- obj.each {|v| s += sum(v) }

- s

- else

- obj

- end

- end

objにeachがある  
ArrayとかRangeとか

objにはeachがない  
呼出元を信じればここではきっとInteger

# sum(100)の実行

- ```
def sum(obj)
  if obj.respond_to? :each
    s = 0
    obj.each {|v| s += sum(v) }
    s
  else
    obj
  end
end
```

100が渡ってくる

偽になる

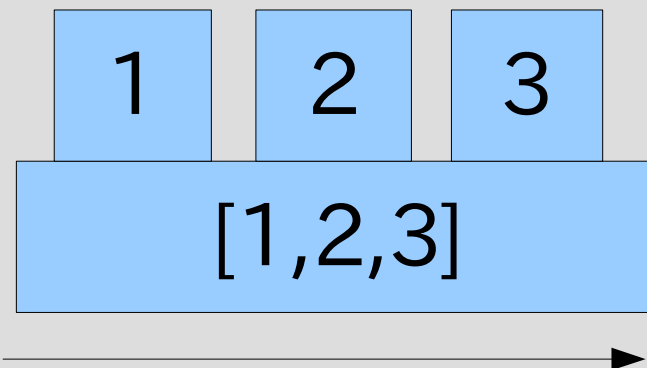
100を返す



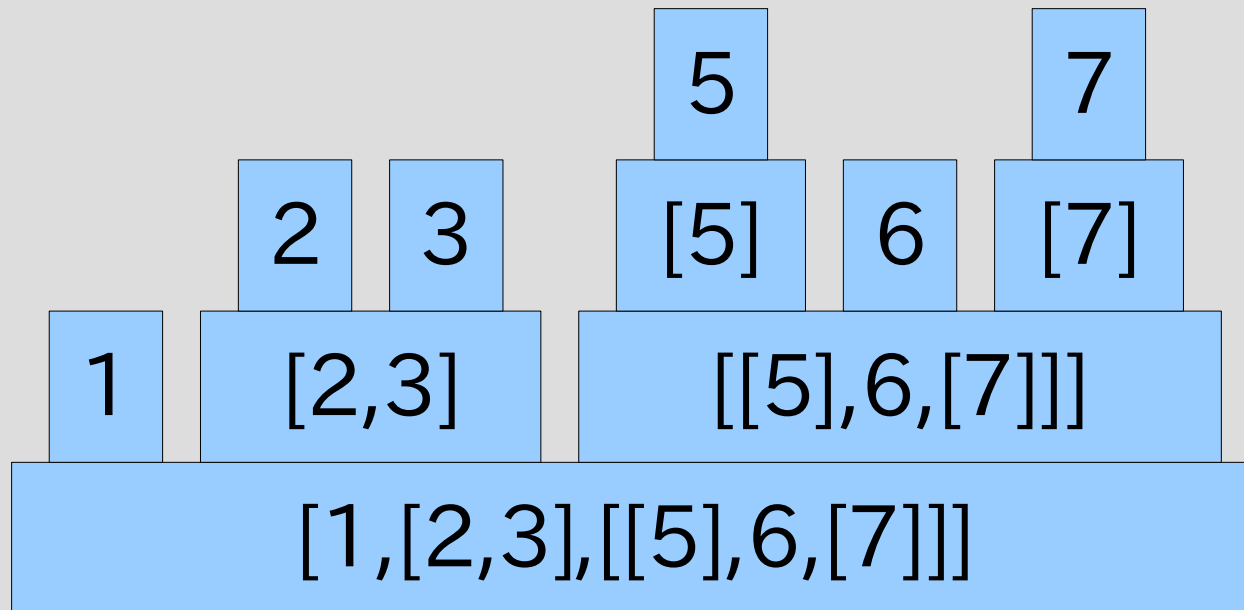
# sum([1,2,3])の実行

- def sum(obj) ← [1,2,3]が渡ってくる  
 if obj.respond\_to? :each ← 真になる  
 s = 0  
 obj.each {|v| s += sum(v)}  
 s  
 else  
 obj  
 end  
end

sum(1), sum(2),  
sum(3) を順に  
呼び出す



# もっと複雑な呼び出し時系列



→ 時間

# Array#flatten

- ネストした配列を展開して一段にする
- `[1,[2,3], [[4]]].flatten`  $\#=>$  `[1,2,3,4]`
- `[]`.flatten  $\#=>$  `[]`
- `[1,2,3].flatten`  $\#=>$  `[1,2,3]`

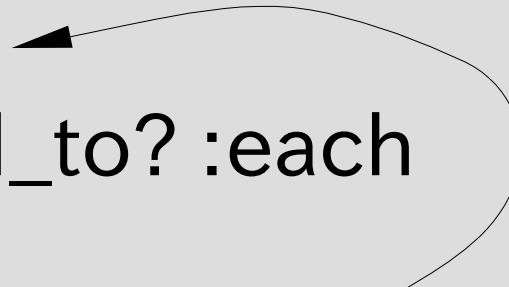
# 自前flatten

- ネストした配列を展開して一段にする
- `flatten([1,[2,3], [[4]])`  $\#=>$  `[1,2,3,4]`
- `flatten([])`  $\#=>$  `[]`
- `flatten([1,2,3])`  $\#=>$  `[1,2,3]`
- `flatten(1)`  $\#=>$  `[1]`

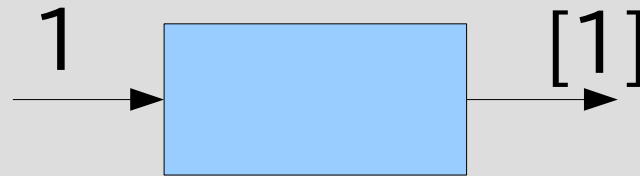
# flatten の実装例

```
def flatten(v)
  if v.respond_to? :each
    r = []
    v.each { |e| r += flatten(e) }
    r
  else
    [v]
  end
end
```

再帰



# flatten(1)の動作



`flatten(1) #=> [1]`

# flatten(1)の動作

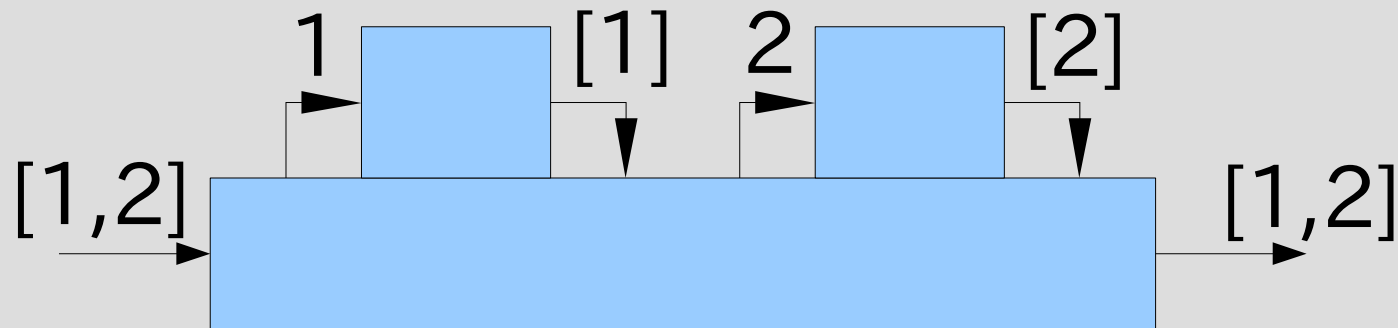
```
def flatten(v)
  if v.respond_to?(:each)
    r = []
    v.each {|e| r += flatten(e) }
    r
  else
    [v]
  end
end
```

1が渡される

偽になる

[v]を返す

# flatten([1,2])の動作



flatten([1,2]) #=> [1, 2]



# flatten([1,2])の動作

```
def flatten(v)
  if v.respond_to?(:each)
    r = []
    v.each {|e| r += flatten(e) }
    r
  else
    [v]
  end
end
```

[1,2]が渡される

真になる

flatten(1),  
flatten(2) を  
順に呼び出す

# flatten の実装例の問題

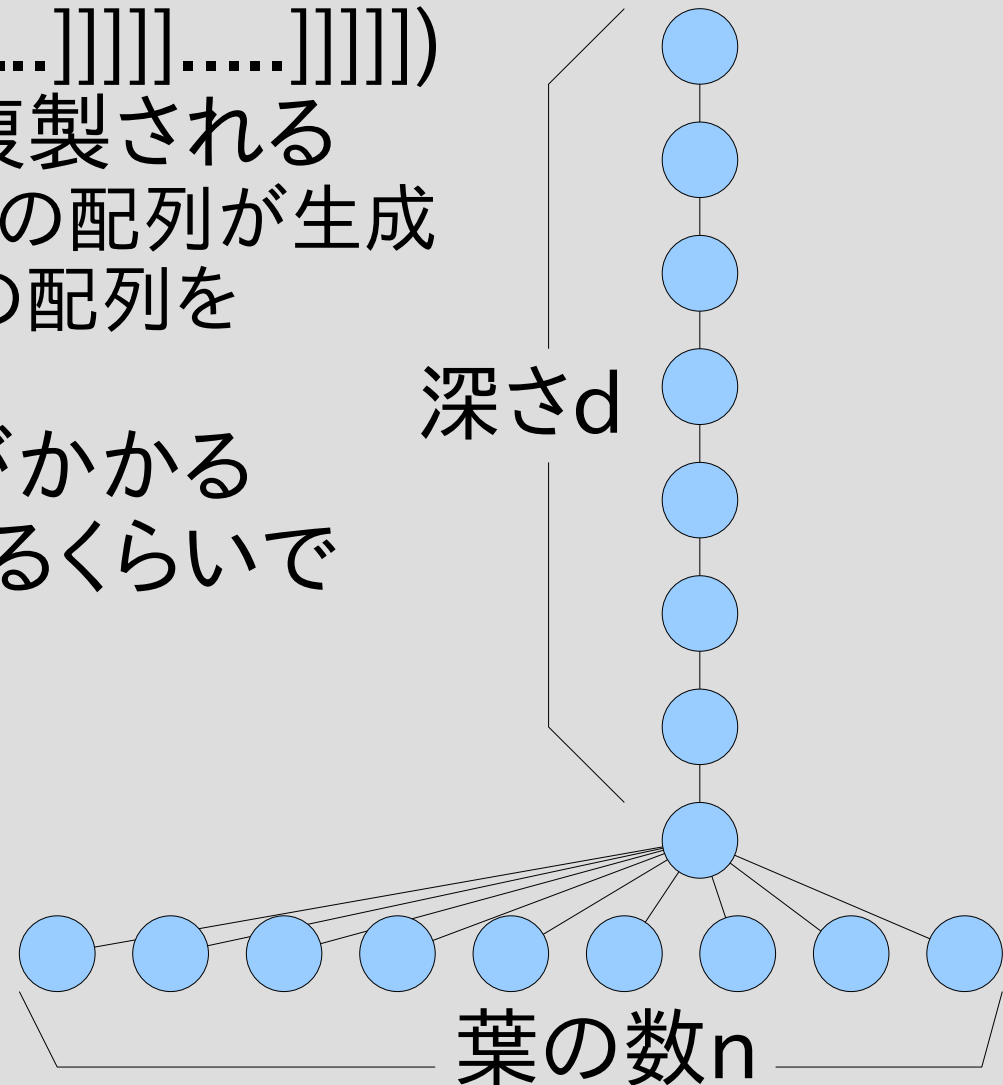
```
def flatten(v)
  if v.respond_to? :each
    r = []
    v.each { |e| r += flatten(e) }
    r
  else
    [v]
  end
end
```

問題点:

- Array#+ は遅い
- 生成される配列の長さに比例する時間
- 全体で最悪、深さ×長さくらいかかる
- そもそも配列を生成しすぎ

# 最悪ケース

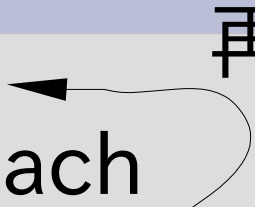
- `flatten([[[[...[[[1, 2, ...]]]]].....]]])`
- 長さ  $n$  の配列が  $d$  回複製される
  - 再帰の一番奥で長さ  $n$  の配列が生成
  - 再帰から返るたびにその配列を空配列に  $+$  で連結
- $n \times d$  に比例する時間がかかる
- ノード数  $n+d$  に比例するくらいで済んでほしい



# flatten の実装例 (高速版)

```
def flatten_acc(v, r) ← 再帰
  if v.respond_to?(:each)
    v.each {|e| flatten_acc(e, r) }
  else
    r << v
  end
end

def flatten(v)
  r = []
  flatten_acc(v, r)
  r
end
```



# Array#<<

- 配列の末尾に要素を破壊的に追加する
- 平均して定数時間で動作する:  $O(1)$   
動作時間は配列の長さに関係しない (平均で)
- 例  
ary = [1,2,3]  
ary << 4 # この時点で ary は [1,2,3,4] になる  
ary << 5  
ary << 6  
p ary #=> [1,2,3,4,5,6]

# flatten\_acc の Array#<<

```
def flatten_acc(v, r) ← 再帰
  if v.respond_to?(:each)
    v.each {|e| flatten_acc(e, r) }
  else
    r << v
  end
end
```

```
def flatten(v)
  r = []
  flatten_acc(v, r)
  r
end
```

# flatten の実装例 (高速版)

```
def flatten_acc(v, r) ← 再帰
  if v.respond_to?(:each)
    v.each {|e| flatten_acc(e, r) }
  else
    r << v
  end
end

def flatten(v)
  r = []
  flatten_acc(v, r)
  r
end
```

- 配列をひとつしか生成しない
- その配列に要素を付け加えていく
- 前の最悪ケースだと  $n+d$  に比例で済む

# 前回のクイックソートも遅い

- def qsort(ary)  
 return ary if ary.length <= 1  
 pivot = ary[0]  
 smaller, bigger = ary[1..-1].partition { |v| v < pivot }  
 qsort(smaller) + [pivot] + qsort(bigger)  
end

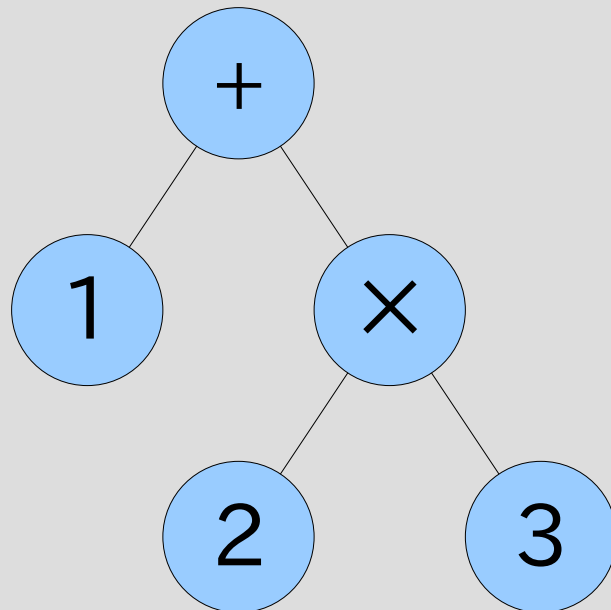
Array#+ を使っていて実は遅い  
Array#partition も遅い  
ary[1..-1] も遅い  
どれも配列の長さに比例する  
でも比較回数の定数倍程度



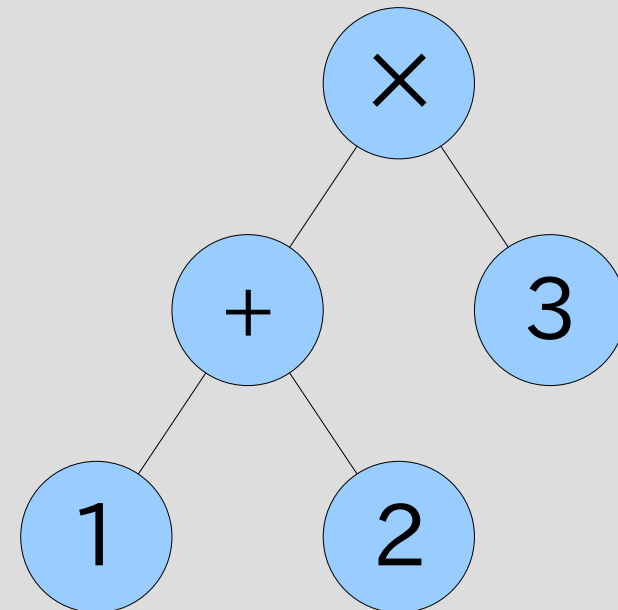
# 数式と木構造

- 数式は木構造で表現できる
- 木構造で表現するときには括弧は不要

$1+2\times 3$



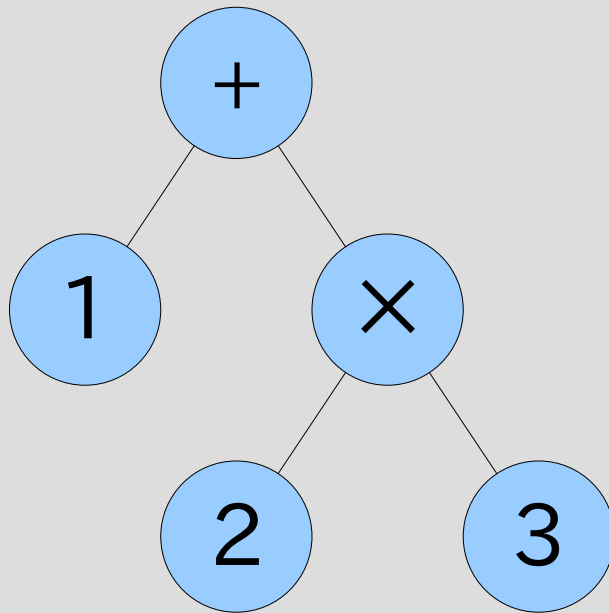
$(1+2)\times 3$



# 木構造を配列で表現

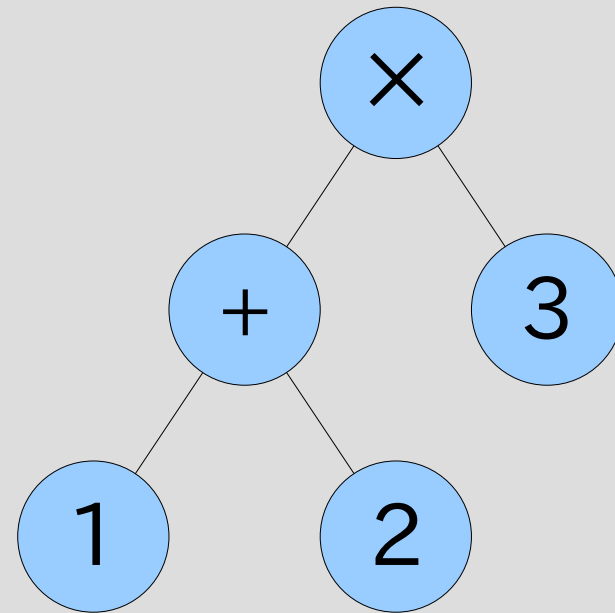
- `[[:plus, 1, [:mult, 2, 3]]]`

$1+2\times 3$



- `[[:mult, [:plus, 1, 2], 3]]`

$(1+2)\times 3$



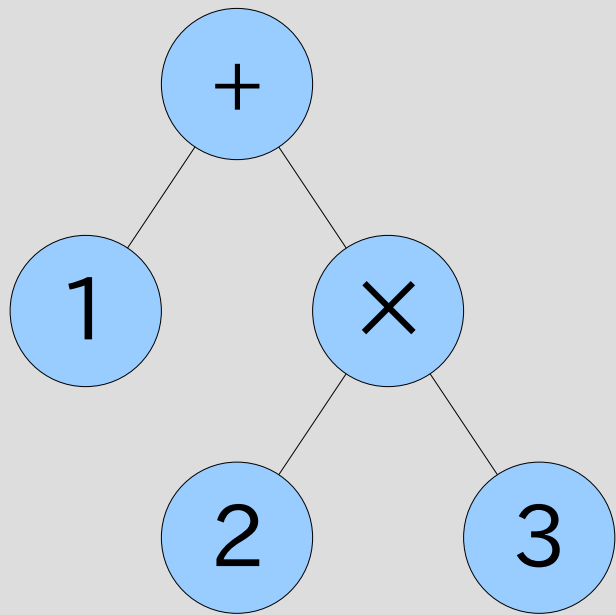
# シンボル

- :foo
- 名前を表現するオブジェクト
- コロンに続いて名前を記述する
- 同じ名前かどうか比較できる
- C の enum に類似した用途
- 文字列とは別種のオブジェクト

`[:plus, 1, [:mult, 2, 3]]`

# 数式と配列の対応

- `[[:plus, 1, [:mult, 2, 3]]]`



1+2×3

- 数値はそれ自身で表す
- $a+b$  は `[[:plus, A, B]]` で表す
- $a-b$  は `[[:minus, A, B]]` で表す
- $a\times b$  は `[[:mult, A, B]]` で表す
- $a\div b$  は `[[:div, A, B]]` で表す
- $A, B$  は  $a, b$  の式を配列に変換して表現したもの

# 式を計算する関数 calc

- `calc(1)` `#=> 1`
- `calc([:plus, 1, 2])` `#=> 3`
- `calc([:minus, 1, 2])` `#=> -1`
- `calc([:plus, 1, [:mult, 2, 3]])` `#=> 7`
- `calc([:mult, [:plus, 1, 2], [:minus, 3, 4]])` `#=> -3`

# calc の実装

```
def calc(exp)
  if exp.respond_to? :to_int
    exp
  else
    case exp[0]
    when :plus
      calc(exp[1]) + calc(exp[2])
    when :minus
      calc(exp[1]) - calc(exp[2])
    when :mult
      calc(exp[1]) * calc(exp[2])
    when :div
      calc(exp[1]) / calc(exp[2])
    end
  end
end
```

# obj.respond\_to? :to\_int

- Ruby で整数は to\_int メソッドを持つ
- 整数以外は持たない
- respond\_to? :to\_int で整数を判定できる

```
def calc(exp)
  if exp.respond_to? :to_int
    exp
  else
    ...
  end
end
```

# case文

- Ruby
- case 式  
when 式  
文  
...  
else  
文  
end
- 次の選択肢に移ることはない(break 不要)
- else 節は省略可能
- C
- switch (式) {  
case 定数:  
文; break;  
...  
default:  
文; break;  
}
- default 部分は省略可能



# calc の case

```
def calc(exp)
  if exp.respond_to? :to_int
    exp
  else
    case exp[0]
    when :plus
      calc(exp[1]) + calc(exp[2])
    when :minus
      calc(exp[1]) - calc(exp[2])
    when :mult
      calc(exp[1]) * calc(exp[2])
    when :div
      calc(exp[1]) / calc(exp[2])
    end
  end
end
```

exp の最初の要素が  
:plus か、  
:minus か、  
:mult か、  
:div によって分岐

# calc(100)

```
def calc(exp)
  if exp.respond_to? :to_int
    exp
  else
    case exp[0]
    when :plus
      calc(exp[1]) + calc(exp[2])
    when :minus
      calc(exp[1]) - calc(exp[2])
    when :mult
      calc(exp[1]) * calc(exp[2])
    when :div
      calc(exp[1]) / calc(exp[2])
    end
  end
end
```

expが100なら真になる

100が返る

# calc([:plus, 1, 2])

```
def calc(exp)
  if exp.respond_to? :to_int
    exp
  else
    case exp[0]
    when :plus
      calc(exp[1]) + calc(exp[2])
    when :minus
      calc(exp[1]) - calc(exp[2])
    when :mult
      calc(exp[1]) * calc(exp[2])
    when :div
      calc(exp[1]) / calc(exp[2])
    end
  end
end
```

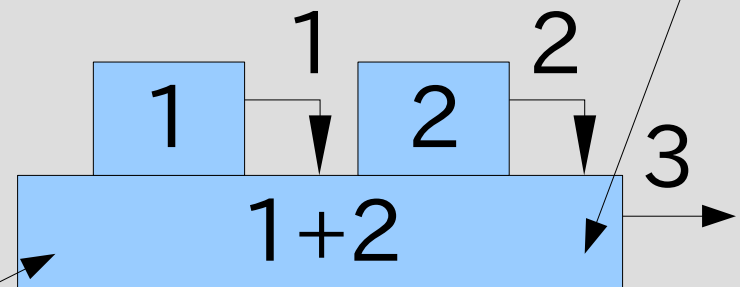
偽になる

:plus になる

一致する

再帰した後  
足し算して  
結果の 3 が返る

このあたりでで足し算



このあたりで分岐

時間

# calc([:mult, 2, 3])

```
def calc(exp)
```

```
  if exp.respond_to? :to_int
```

偽になる

```
    exp
```

```
  else
```

:mult になる

```
    case exp[0]
```

```
      when :plus
```

```
        calc(exp[1]) + calc(exp[2])
```

一致しない

```
      when :minus
```

```
        calc(exp[1]) - calc(exp[2])
```

一致する

```
      when :mult
```

```
        calc(exp[1]) * calc(exp[2])
```

再帰した後

掛算して

結果の 6 が返る

```
      when :div
```

```
        calc(exp[1]) / calc(exp[2])
```

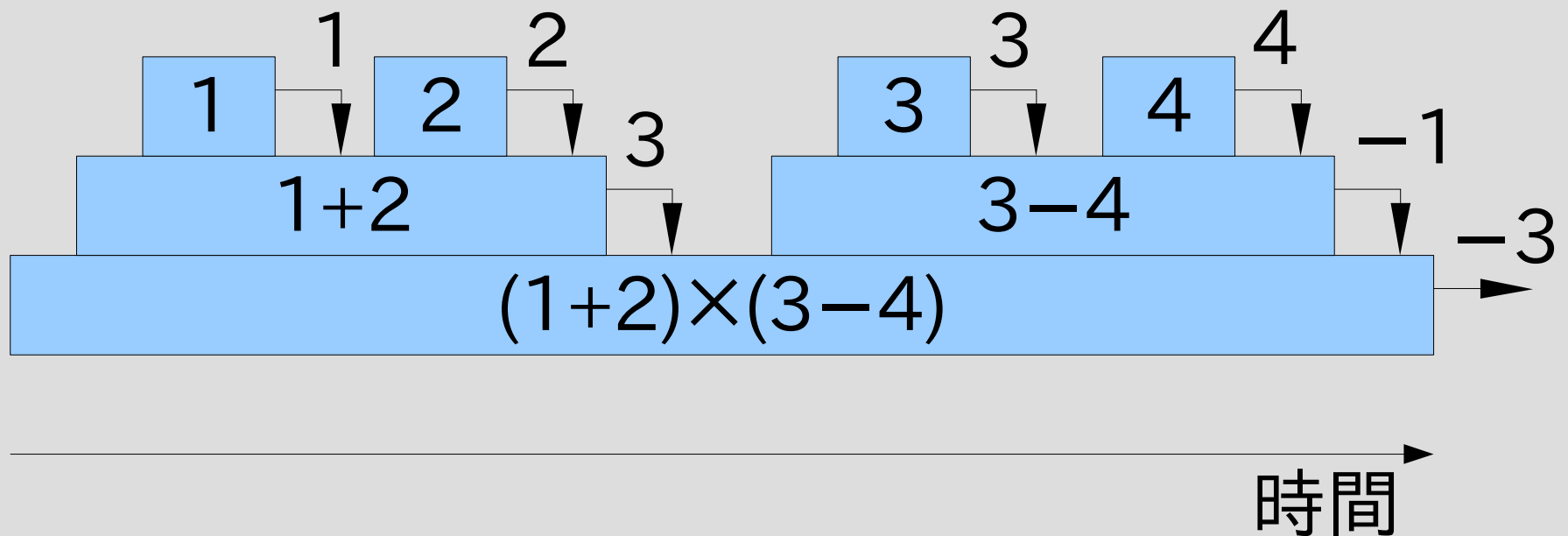
```
    end
```

```
  end
```

```
end
```

$$(1+2) \times (3-4)$$

- `calc([:mult, [:plus, 1, 2], [:minus, 3, 4]]) #=> -3`



# 冪乗

- 冪 (べき), 冪乗 (べきじょう), 累乗 (るいじょう)
- $2^3 = 2 \times 2 \times 2 = 8$
- 数学の記法では  $2^3$  というように右上に書く
- Ruby では `2**3` と書く
- 英語だと power

# 冪乗のサポート

```
def calc(exp)
  if exp.respond_to? :to_int
    exp
  else
    case exp[0]
    when :plus
      calc(exp[1]) + calc(exp[2])
    when :minus
      calc(exp[1]) - calc(exp[2])
    when :mult
      calc(exp[1]) * calc(exp[2])
    when :div
      calc(exp[1]) / calc(exp[2])
    when :pow
      calc(exp[1]) ** calc(exp[2])
    end
  end
end
```

## 冪乗のサポート (追加部分)

- 追加する演算の名前に対応する分岐を追加する
- 分岐の中でその演算を実行する

...

```
when :pow
```

```
  calc(exp[1]) ** calc(exp[2])
```

...



# 階乗のサポート

- `calc([:fact, 0])`      `#=> 1`
- `calc([:fact, 1])`      `#=> 1`
- `calc([:fact, 2])`      `#=> 2`
- `calc([:fact, 5])`      `#=> 120`
- `calc([:fact, 10])`     `#=> 3628800`

# 階乗のサポート

```
def calc(exp)
  if exp.respond_to? :to_int
    exp
  else
    case exp[0]
    when :plus
      calc(exp[1]) + calc(exp[2])
    when :minus
      calc(exp[1]) - calc(exp[2])
    when :mult
      calc(exp[1]) * calc(exp[2])
    when :div
      calc(exp[1]) / calc(exp[2])
    when :pow
      calc(exp[1]) ** calc(exp[2])
    when :fact
      fact(calc(exp[1]))
    end
  end
end
```

```
def fact(n)
  ret = 1
  1.upto(n) {|i| ret *= i }
  ret
end
```

# 階乗のサポート (追加部分)

fact 用の分岐を追加

...

when :fact

fact(calc(exp[1]))

...

fact の実装を追加

```
def fact(n)
```

```
  ret = 1
```

```
  1.upto(n) {|i| ret *= i }
```

```
  ret
```

```
end
```

# レポート

- 足し算が引数を任意個とれるよう calc を拡張せよ
- 足し算の引数がひとつもないときの動作について考えよ
- ✖切 2007-05-29 16:20
- HIPLUS
- 拡張子が txt なファイルが望ましい

# 足し算の任意個引数

- `calc([:plus, 1, 2])`  $\#=> 3$
  - `calc([:plus, 1, 2, 3])`  $\#=> 6$
  - `calc([:plus, 1, 2, 3, 4])`  $\#=> 10$
  - `calc([:plus, 1, 2, 3, 4, 5])`  $\#=> 15$
- 
- 上記のような動作が可能なよう `calc` を拡張する
  - レポートには実装、動作の様子、解説を含める

# 足し算の引数がない場合

- `calc([:plus])` #=> ???
- 以下のことをレポートで述べる
  - どんな値にすべきか?
  - その理由は何か?

# まとめ

- 木構造に対する再帰
  - sum
  - flatten
- 数式の評価
  - 四則演算
  - 冪乗
  - 階乗
- レポートを出した