テキスト処理 第4回 (2008-05-13)

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

- 端末とコマンドライン
- 再帰と木構造
- メソッド呼び出しのしくみ
- レポート

端末とコマンドライン

- GUI でなく、文字だけでコンピュータと対話する方法がある
- キーボードからコマンドを入力して画面に結果を出力する
- GUI 環境で、そういう方法を提供するソフトウェアを端末エミュレータという
  - Windows のコマンドプロンプト
  - Unix (X Window System) の xterm
- エミュレータではない本物の端末は現在ではまず使われない

VT100

(Unix初期によく使われた端末)

端末での対話

- プロンプトの後にコマンドラインを入力する
- コマンドが実行される
- コマンドが終わるとまたプロンプトが出る

- この対話を行うプログラムをシェルという
  - sh
  - csh
  - zsh

```
prompt% コマンドライン入力
コマンドの出力
prompt% 
```

コマンドライン

- コマンドラインはコマンド名と引数からなる

  prompt% コマンド名 引数1 引数2 ...

  例:
  prompt% egrep foo filename

- 出力をファイルに向けたり、複数のコマンドを接続するパイプなどの機能もある

シングルクォート

- 引数に空白を含めるには引用符で括る
- パイプとかと解釈されないようにするときにも使う

- egrep 'foo bar' filename

  引数1

  f, o, o, 空白, b, a, r の 7文字

- シングルクォート自身は中に入れられない
- この規則は Unix のシェルの規則

ダブルクォート

- 引数に空白を含めるには引用符で括る
- パイプとかと解釈されないようにするときにも使う

- egrep "foo bar" filename

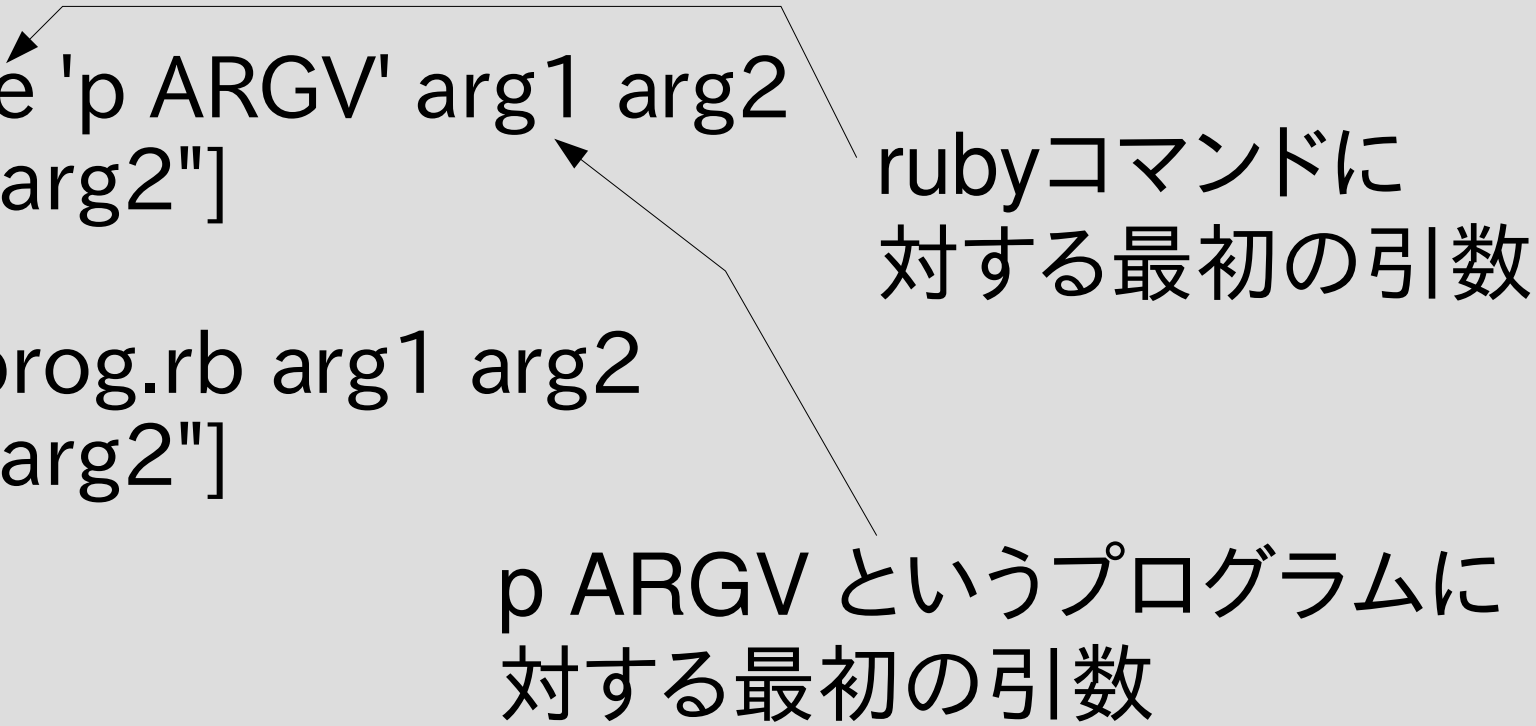
  引数1

  f, o, o, 空白, b, a, r の 7文字

- $, `, ", ￥を入れるときは￥を前置する
- たとえば " 自身を入れるときは "...¥"..." とする
- この規則は Unix のシェルの規則

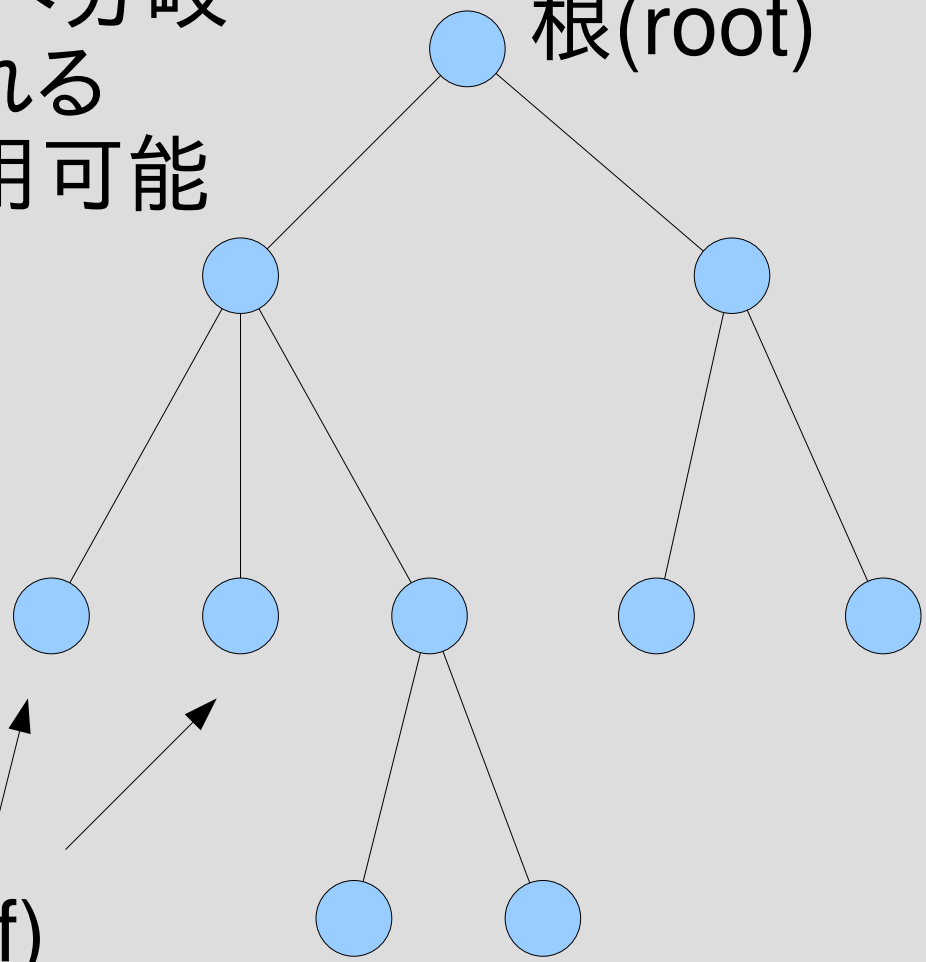
Ruby とコマンドライン引数

- コマンドラインの引数のうち Ruby でかかれたプログラムの指定を除いた残りが ARGV になる
- % ruby -e 'p ARGV' arg1 arg2
  ["arg1", "arg2"]
- % ruby prog.rb arg1 arg2
  ["arg1", "arg2"]



木構造

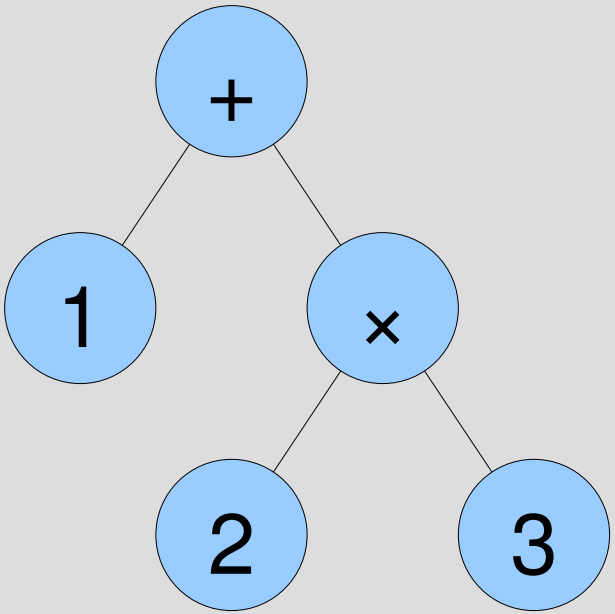
- ノードが根から葉へ分岐していく構造
- 右だと根から6つの葉へ分岐
- ノードに情報をつけられる
- 数式や正規表現に利用可能
- 伝統的に根を上に描く



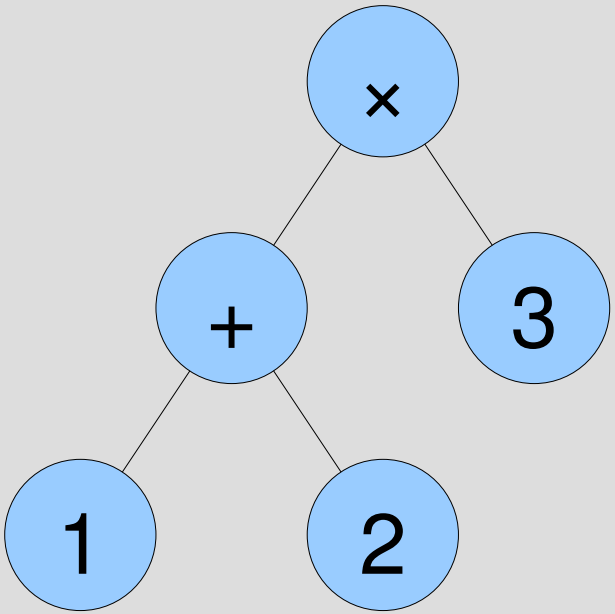
数式と木構造

- 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+2×3

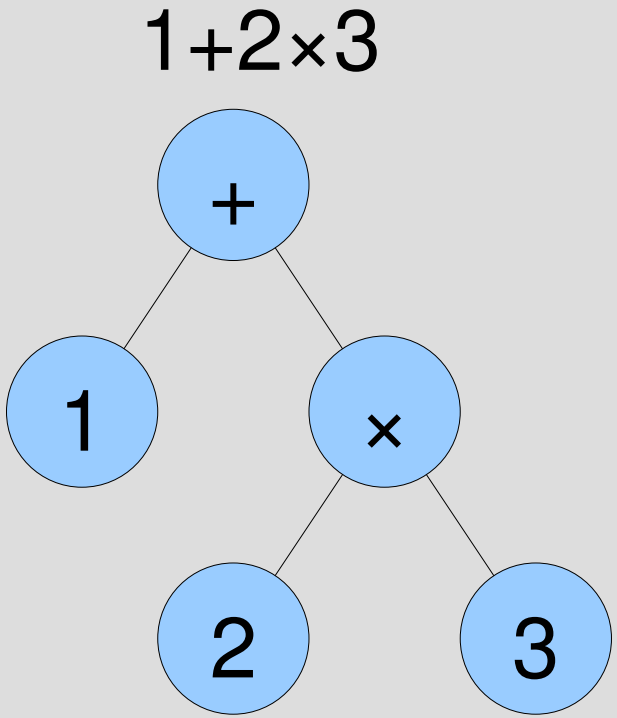


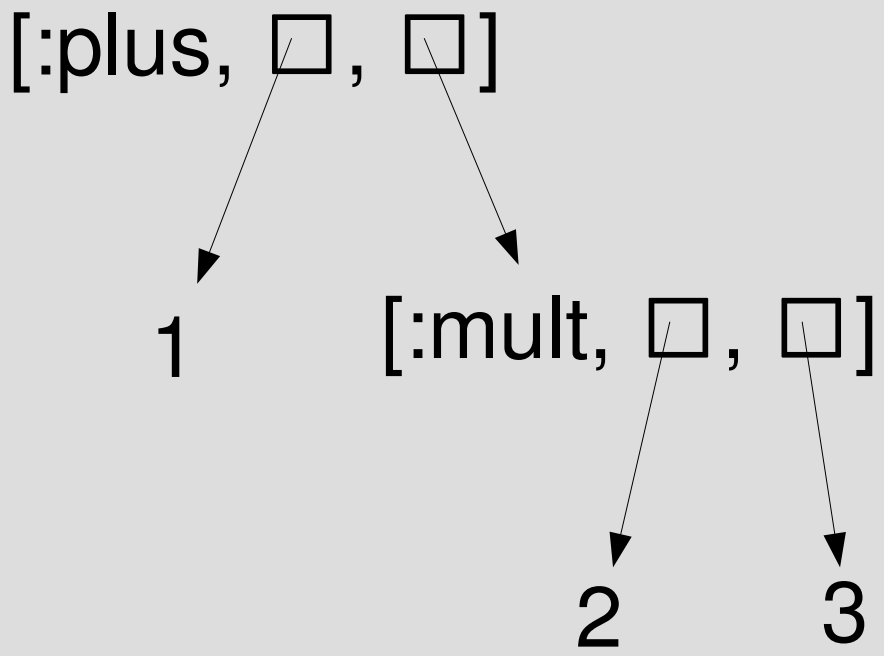
(1+2)×3



木構造を配列で表現

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

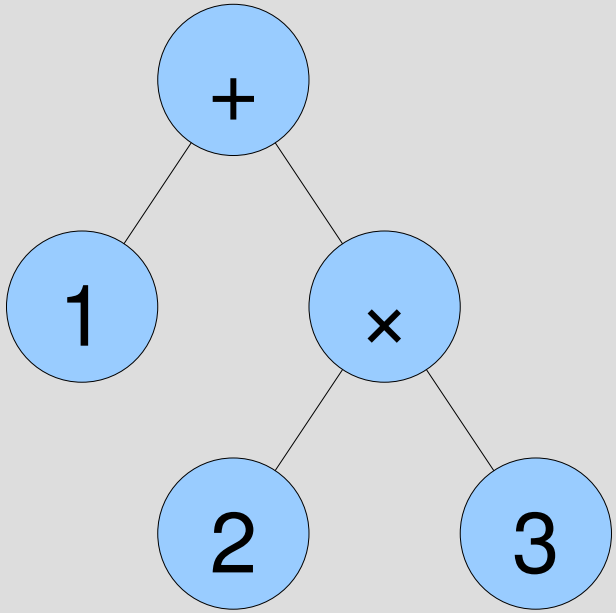




木構造を配列で表現

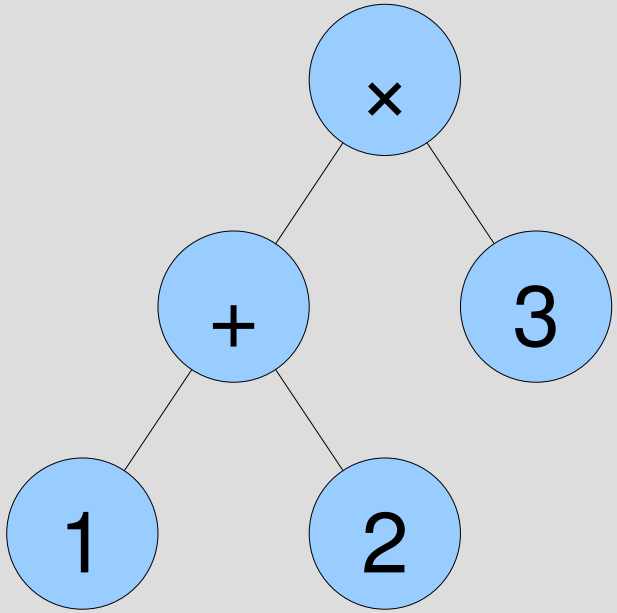
- [:plus, 1, [:mult, 2, 3]]

1+2×3



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

(1+2)×3



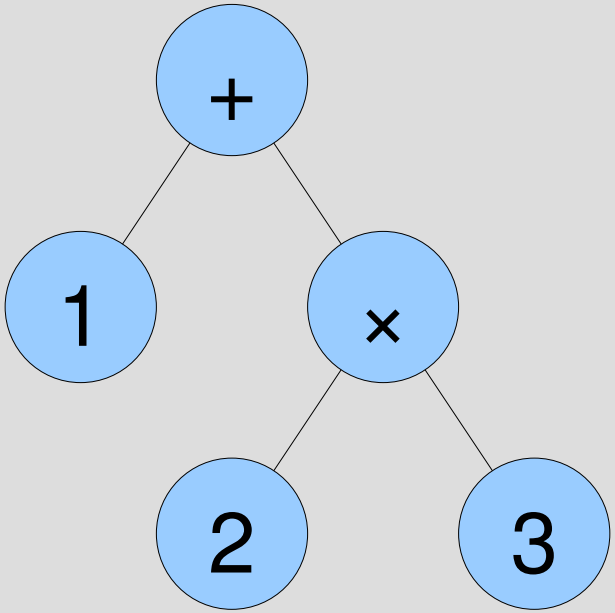
シンボル

- :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

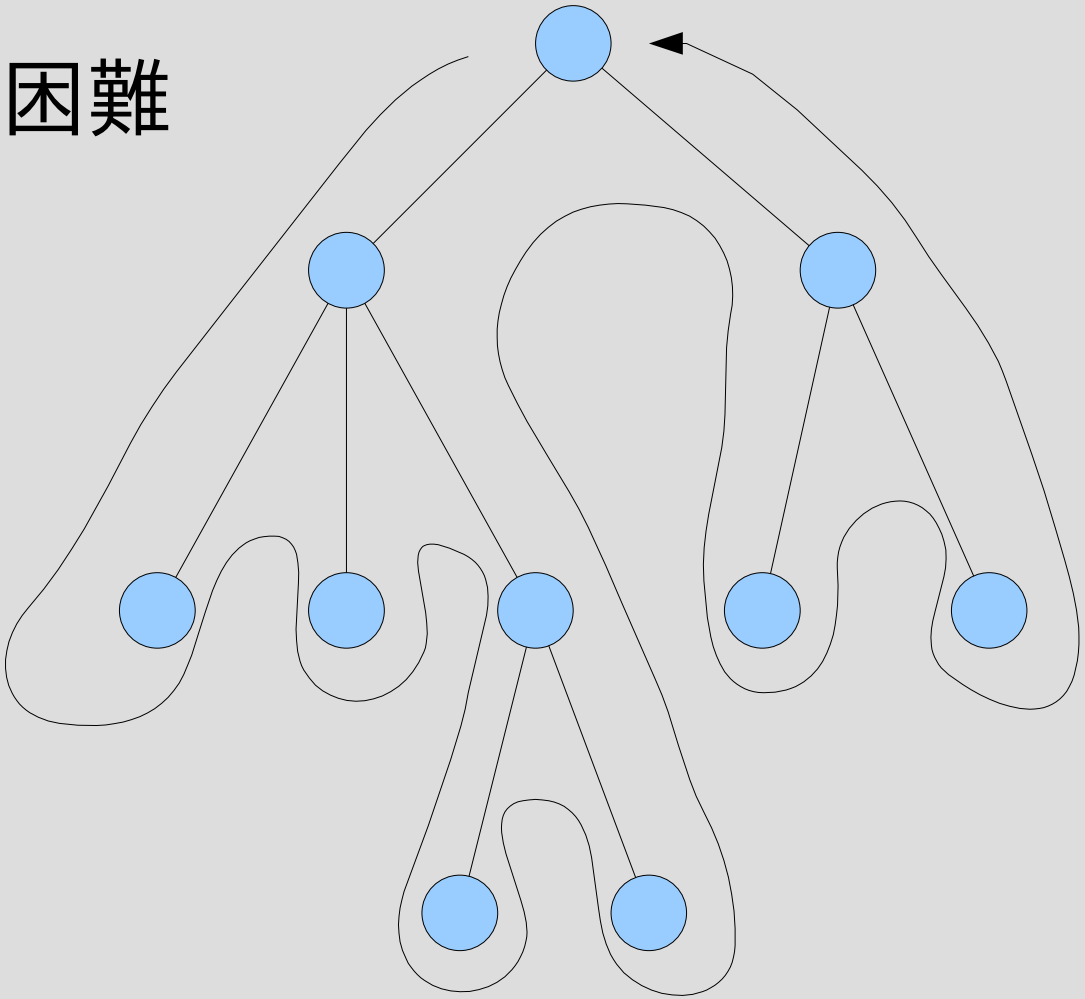
- 数値はそれ自身で表す
- a+b は [:plus, A, B] で表す
- a−b は [:minus, A, B]で表す
- a×b は [:mult, A, B]で表す
- a÷b は [:div, A, B]で表す
- A, B は a, b の式を配列に変換して表現したもの

再帰

- ある関数がその関数自身を呼ぶこと
- 木構造を処理するのに必須

木構造に対する再帰

- 木構造をたどる
- 再帰を使わないと困難



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
```

calc の再帰

```
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
```

Object#respond_to?

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

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 で整数かどうか判定する

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

case式とswitch文

- Ruby
- case 式
  when 式
    式
  ...
  else
    式
  end
- else 節は省略可能
- 選ばれた節の値がcase式の値になる
- 次の選択肢に移ることはない(break 不要)

- C
- switch (式) {
    case 定数:
      文; break;
    ...
    default:
      文; break;
  }
- default 部分は省略可能
- switch 文は式ではなく、値を持たない

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)の実行

```
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])
```

偽になる

:plus になる

一致する

2回再帰して
両辺を計算した後
それらを足し算して

結果の 3が返る

再帰するプログラムを読む注意

- とりあえず calc が式を計算して値を返すという動作を認めてしまうこと
- calc の動作をたどるときには再帰した先はたどらないこと
- calc の呼出し一回の動作をたどるのは難しくない
- 再帰が無限に続くかどうかは別の話として気にせず、とりあえず終わると想定して考える

calc([:mult, 2, 3])

```
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])
```

偽になる

:mult になる

一致しない

一致する

再帰した後
掛算して
結果の 6が返る

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 と記述して計算できる
- 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])
...
```

符号反転のサポート

- calc([:neg, 0])　　#=> 0
- calc([:neg, 1])　　#=> -1
- calc([:neg, 2])　　#=> -2
- calc([:neg, 5])　　#=> -5
- calc([:neg, 10])　　#=> -10

符号反転のサポート

```
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 :mult
      calc(exp[1]) * calc(exp[2])
    when :div
      calc(exp[1]) / calc(exp[2])
    when :pow
      calc(exp[1]) ** calc(exp[2])
    when :neg
      -calc(exp[1])
    end
  end
end
```

符号反転のサポート (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 :neg
  -calc(exp[1])
...
```

メソッド呼び出し

- 定義

```
def func(arg1, arg2, ...)
  式1
  式2
  ...
  式n
end
```

引数1の値がarg1,
引数2の値がarg2, ...
となった状態で
式1, 式2, ... が順に実行される

- 呼び出し

```
func(引数1, 引数2, ...)
```

メソッドの返り値

- def func(arg1, arg2, ...)
  式1
  式2
  ...
  式n
  end

最後の式の値が返値になる

- def func(...)
  return 式 if ...
  ...
  end

途中で値を返したいときは return も使える

メソッド呼び出しのしくみ

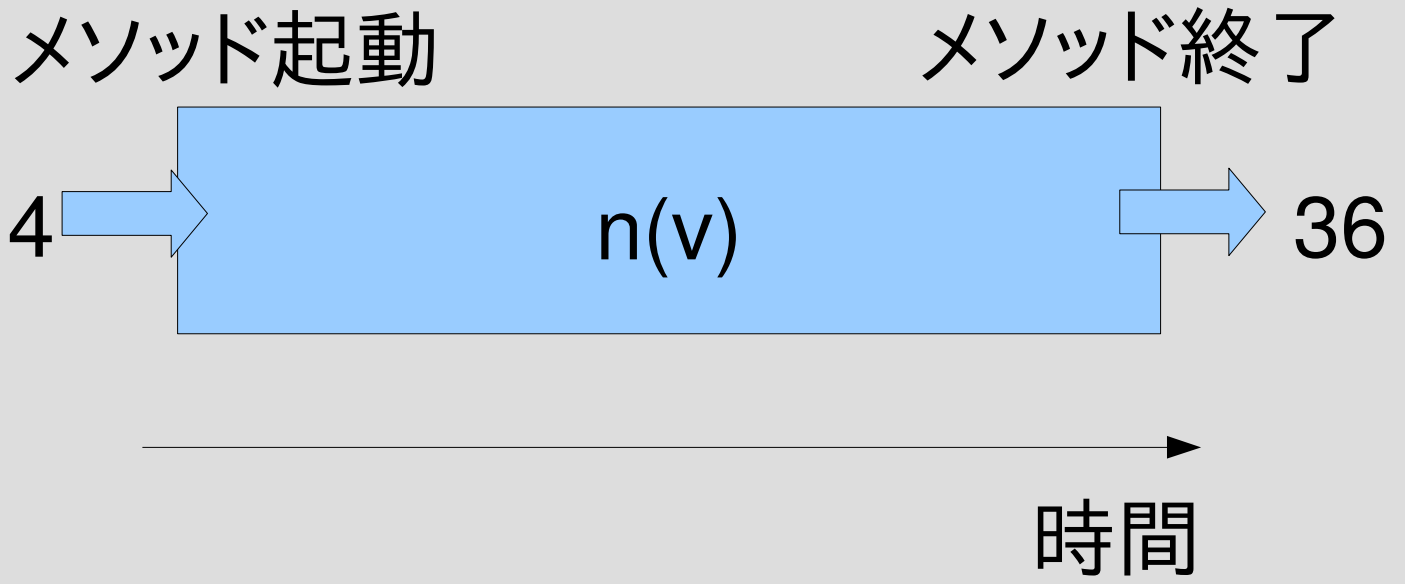
- def m(v)
  v + 1
  end
  def n(v)
  m(v*2)*v
  end

- p n(4) #=> 36

- n(4)=
  m(4*2)*4=
  m(8)*4=
  (8+1)*4=
  9*4=
  36

メソッド呼び出しのしくみ

- def m(v)
  v + 1
  end
  def n(v)
  m(v*2)*v
  end

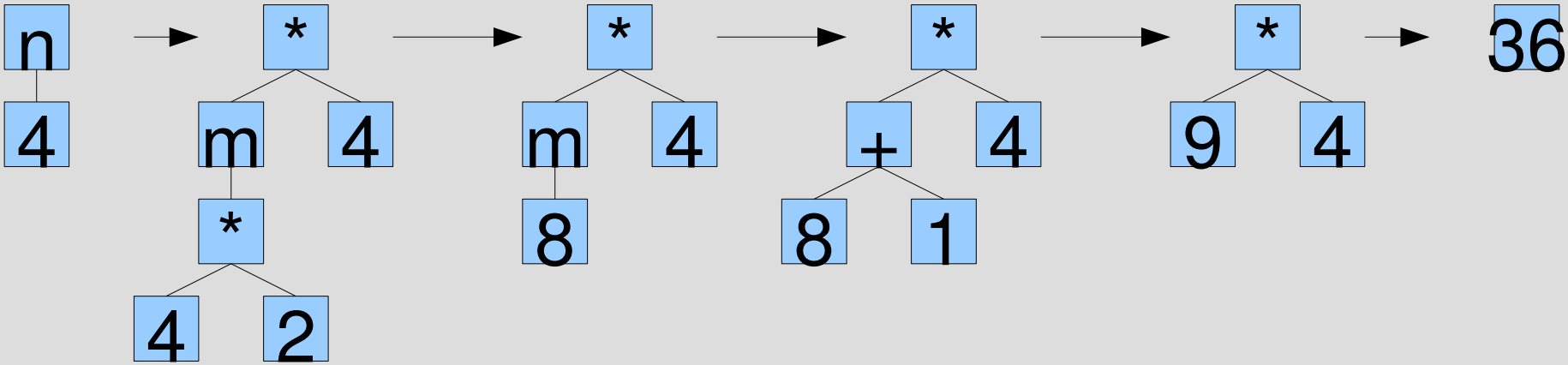


n(v) の中身は実際にはどう動作するのか？

- p n(4) #=> 36

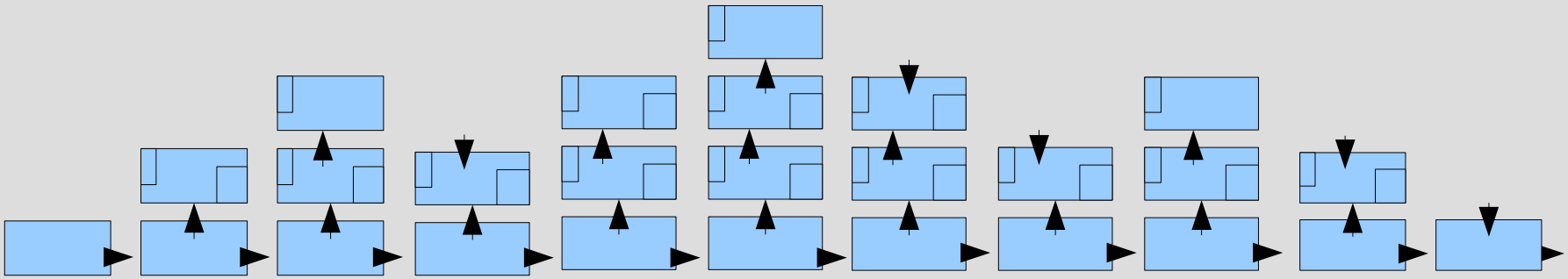
n(4)の計算: 項書換え

- 案1: 式をデータ構造で表現して変形していく
  - コンピュータ上には木構造を表現できる
  - 式の変形のとおりに木構造を書き換えていく
  - そういう言語もある: Haskell とか
  - でも、Ruby を含む多くの言語は直接にはそうしない
  - C も Java も Python も Perl も PHP も

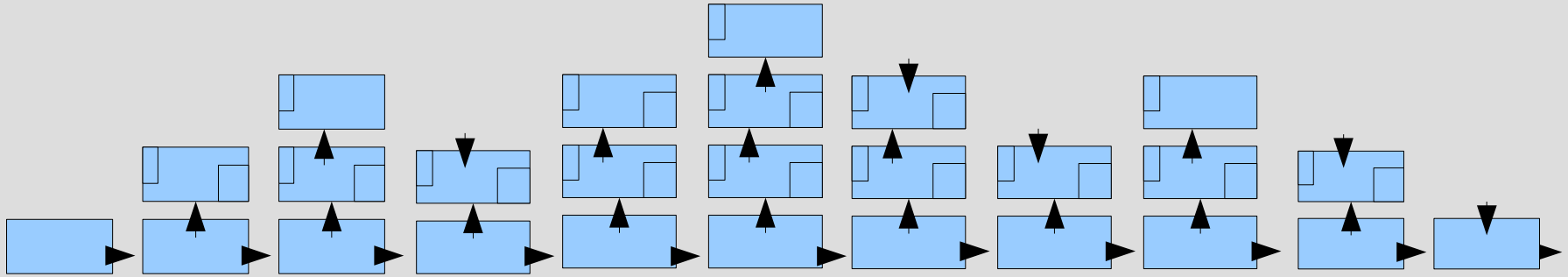


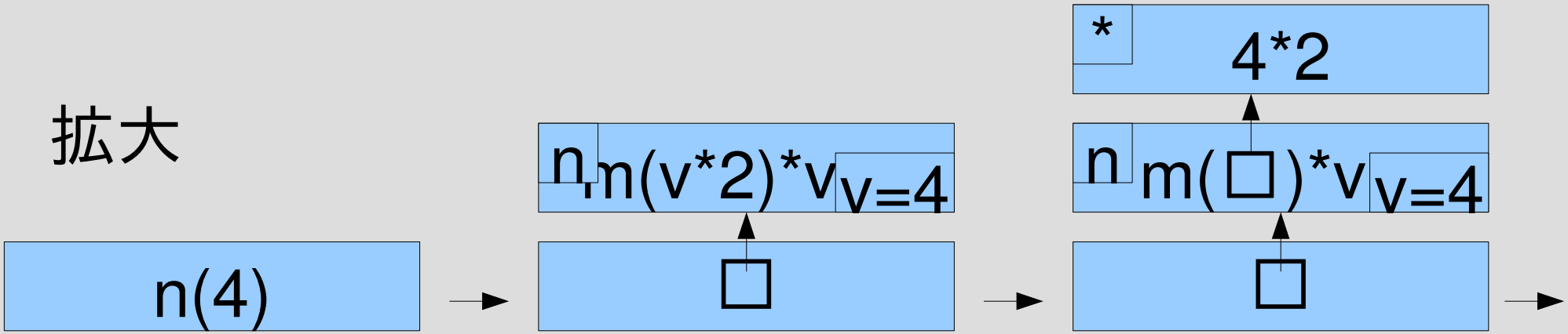
n(4) の計算: スタック

- 案2: スタックを使って計算する
  - メソッドが呼び出されるたびにスタックフレームというメモリを確保してスタックにプッシュする
  - スタックフレームに戻り先とローカル変数を記録する
  - メソッドが終わったらその戻り先に戻ってスタックフレームをスタックからポップする
  - こういう最後に入れたものを最初に使うデータ構造を一般にスタックという
  - メソッド呼び出し用のスタックは制御スタックなどと呼ぶこともあるが、ここでは単にスタックと呼ぶ

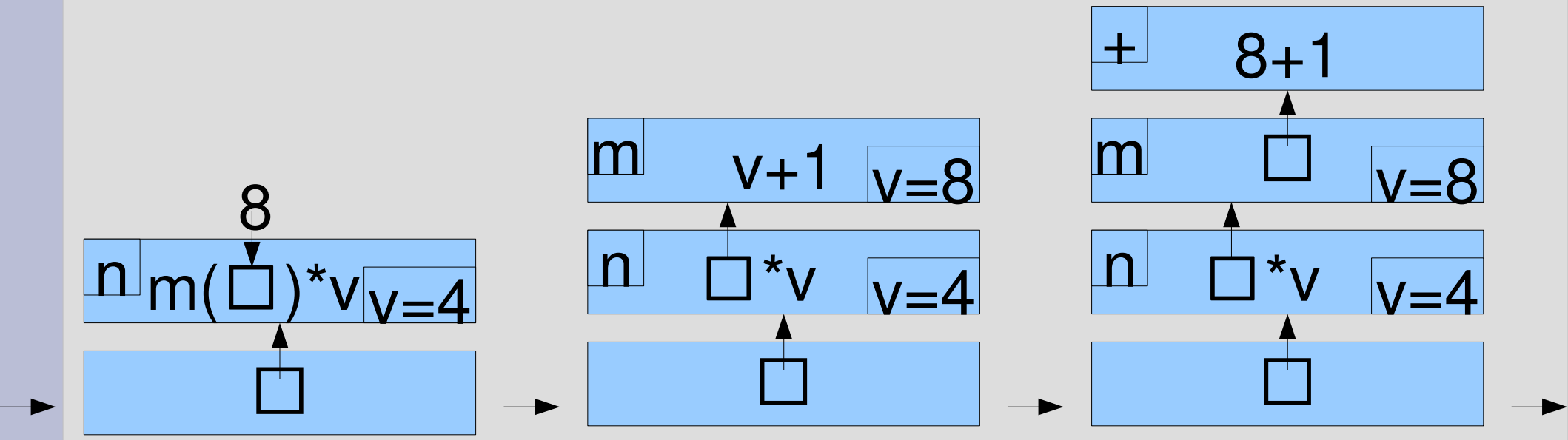


スタックの動作 (1)

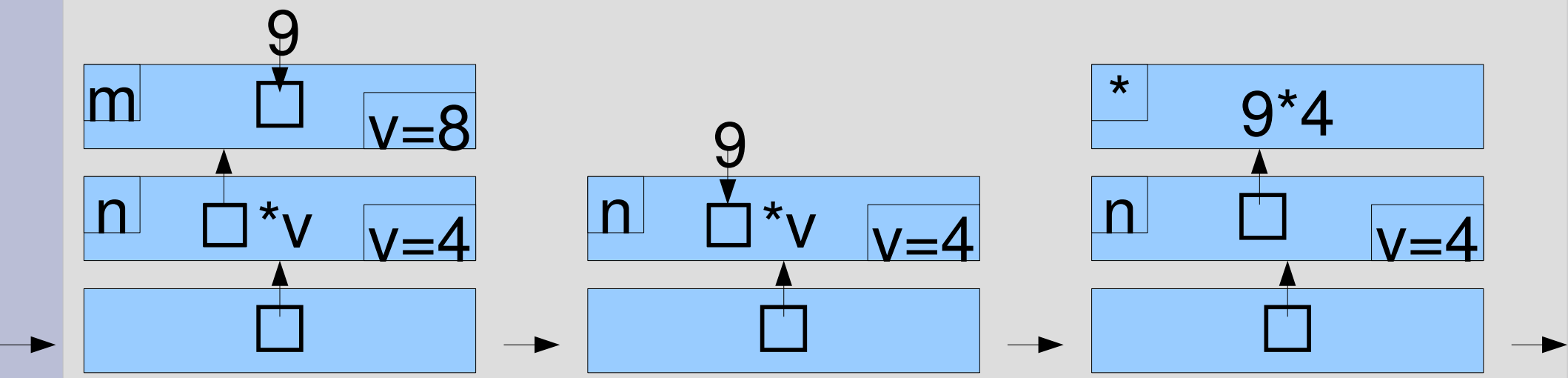




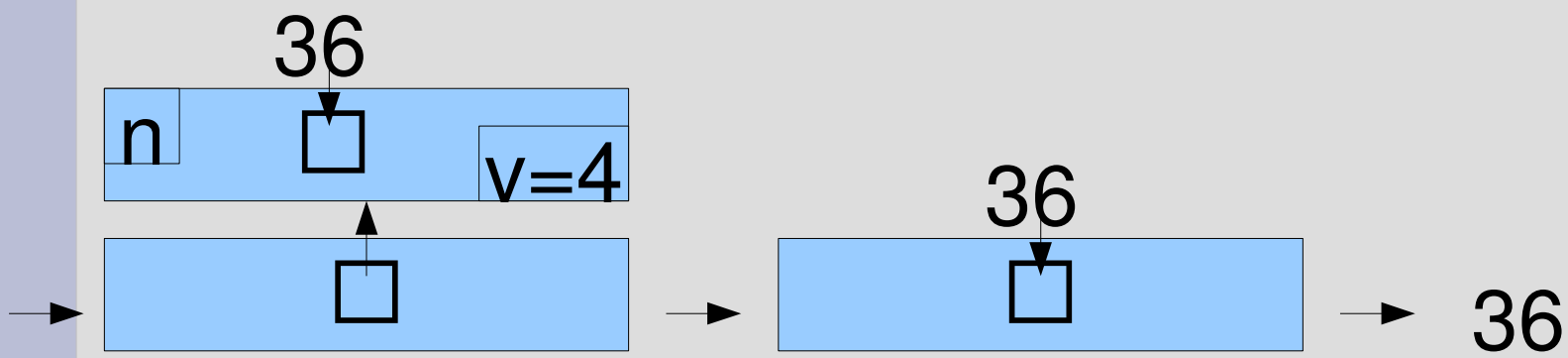
スタックの動作 (2)



スタックの動作 (3)

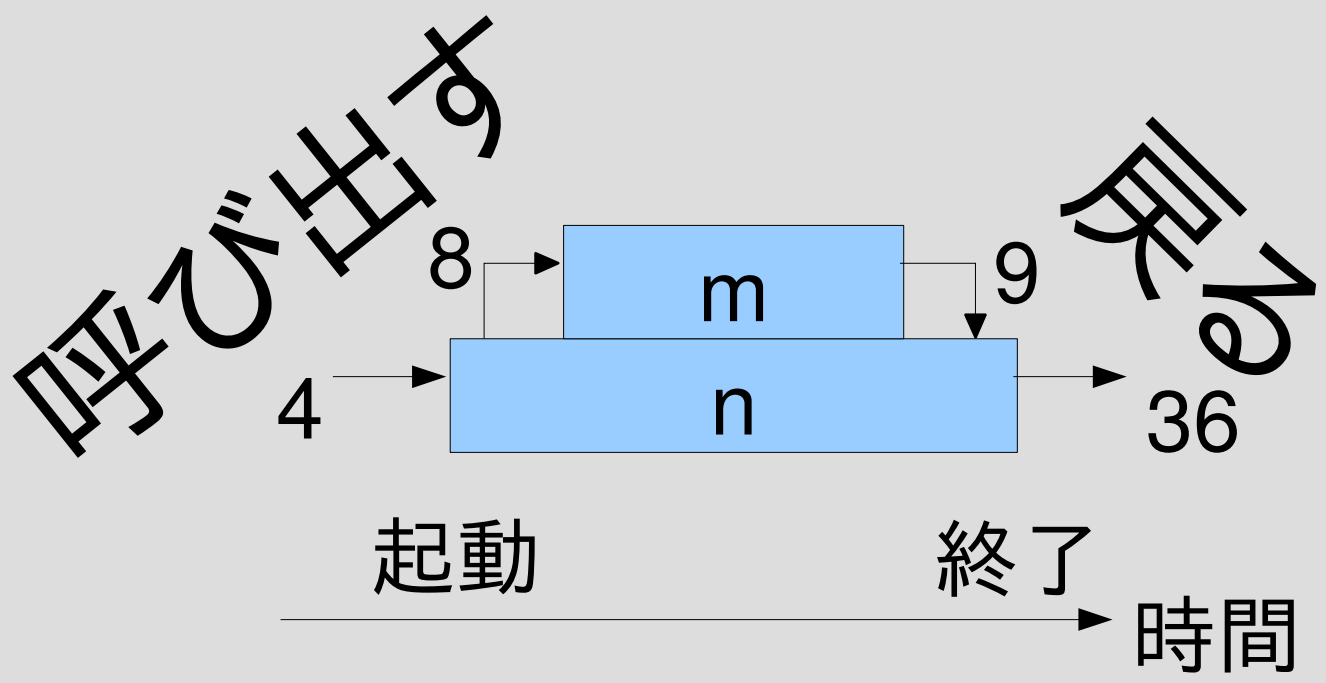


スタックの動作 (4)



おおざっぱな呼び出しの時系列

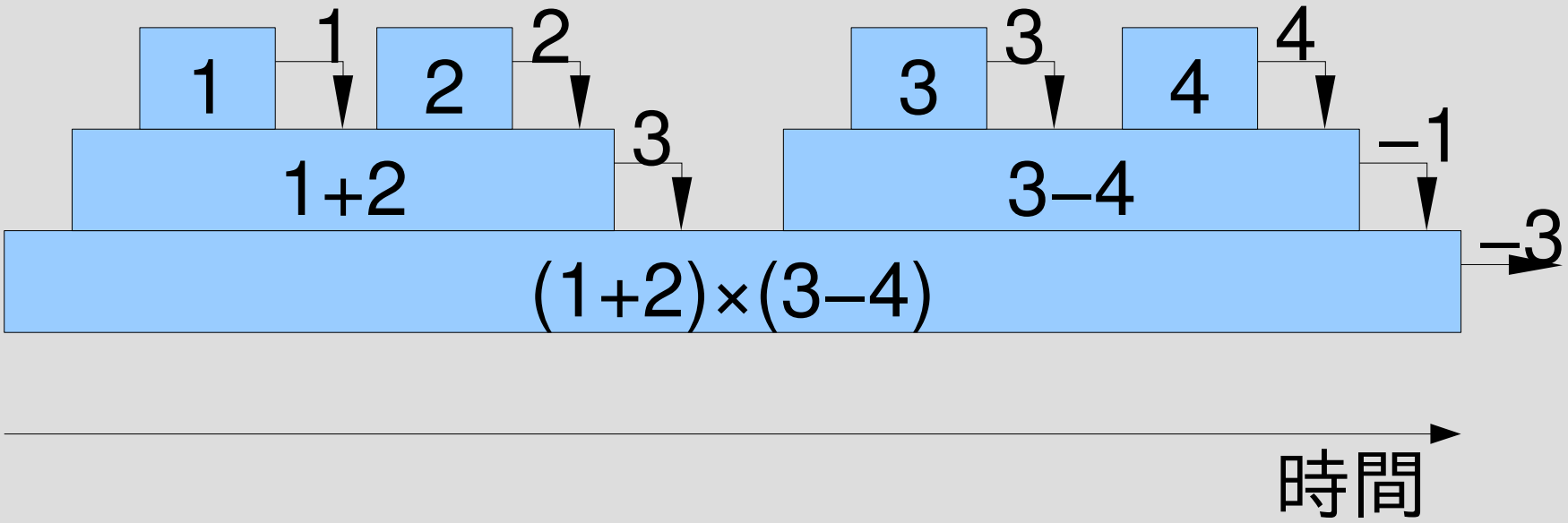
- def m(v)
  v + 1
  end
  def n(v)
  m(v*2)*v
  end



- p n(4) #=> 36

calc で (1+2)×(3−4) を計算

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



レポート

- 乗算が引数を任意個とれるように calc を拡張せよ
- 乗算の引数がひとつもないときの動作について考察せよ
- レポートには以下の内容を含むものとする
  - 拡張についての実装・動作の様子・解説
  - 無引数のときの考察
- 〆切 2008-05-20 12:00
- RENANDI
- 拡張子が txt なファイルが望ましい

任意個引数の乗算

- 2×3×4 ＝2×(3×4) なので
  [:mult, 2, [:mult, 3, 4]] と記述できるが面倒
- [:mult, 2, 3, 4] と記述できるとちょっと楽
- なのでそうできるように calc を拡張せよ

例

- calc([:mult, 2, 3])　　#=> 6
- calc([:mult, 2, 3, 4])　　#=> 24
- calc([:mult, 2, 3, 4, 5])　　#=> 120

もちろん他の演算子とも組み合わせられる

- calc([:div, [:mult, [:plus, 1, 2],
[:minus, 3, 4],
[:pow, 5, 6]],
75])

#=> -625

引数がないとき?

- calc([:mult])　　#=> ???

- どんな値にすべきか?
- その理由は何か?

想定されるレポートの書き方

- 題意に沿ってプログラムを拡張する
- レポートに記載するもの
  - プログラム自体
    (読んだ後実際に実行して試します)
  - どのように考えて変更を行ったか
    (理解できているかどうかの判断に必要)
  - 動作の様子
- うまくいかなくても考えた方針などを書くこと

トラブルが起こったとき

- 症状を再現可能なように書いてあれば可能な範囲で対応する
- 良い書き方
  - やったこと・起きたことをそのまま記述する (重要)
  - 期待される挙動を記述する
- 良い書き方の例
  「1+1 を計算しようとして以下のようにしたら No such file or directory と出てきた
  % ruby 'p 1 + 1'
  ruby: No such file or directory -- p 1 + 1 (LoadError)」
- 悪い書き方の例
  「1+1 を計算しようとしたがうまくいかなかった」

まとめ

- 端末とコマンドライン
  - 端末における対話のしかた
- 木構造と再帰
  - 数式の計算
- メソッド呼び出しのしくみ
  - スタック
- レポートを出題した