

<http://www.flickr.com/photos/dust/3603580129/> (CC Attribution)

ふつうのLisp入門

(Gauche ファンクラブ代表 吉田裕美)

自己紹介

- 吉田裕美 (ヨシダ ユウミ)
- Twitter yuumi3
- ブログ <http://d.hatena.ne.jp/yuum3/>



2000年より独立しEY-Officeを設立

- Ruby / Perl / Java 等でWebシステムを多数構築
- 最近はRuby on Railsの教育なども行っています



- ・アジアリーグアイスホケー
- ・<http://www.alhockey.jp>
- ・チーム・選手の成績集計



- ・アロハパーク (アロハストリート)
- ・<http://park.aloha-street.com/>
- ・SNS



フムフム・ヌクヌク・アブアアの絵の画像は <http://plaza.rakuten.co.jp/ohanamaster/diary/200806070000/> から

(Gaucheファンクラブ代表 吉田裕美)

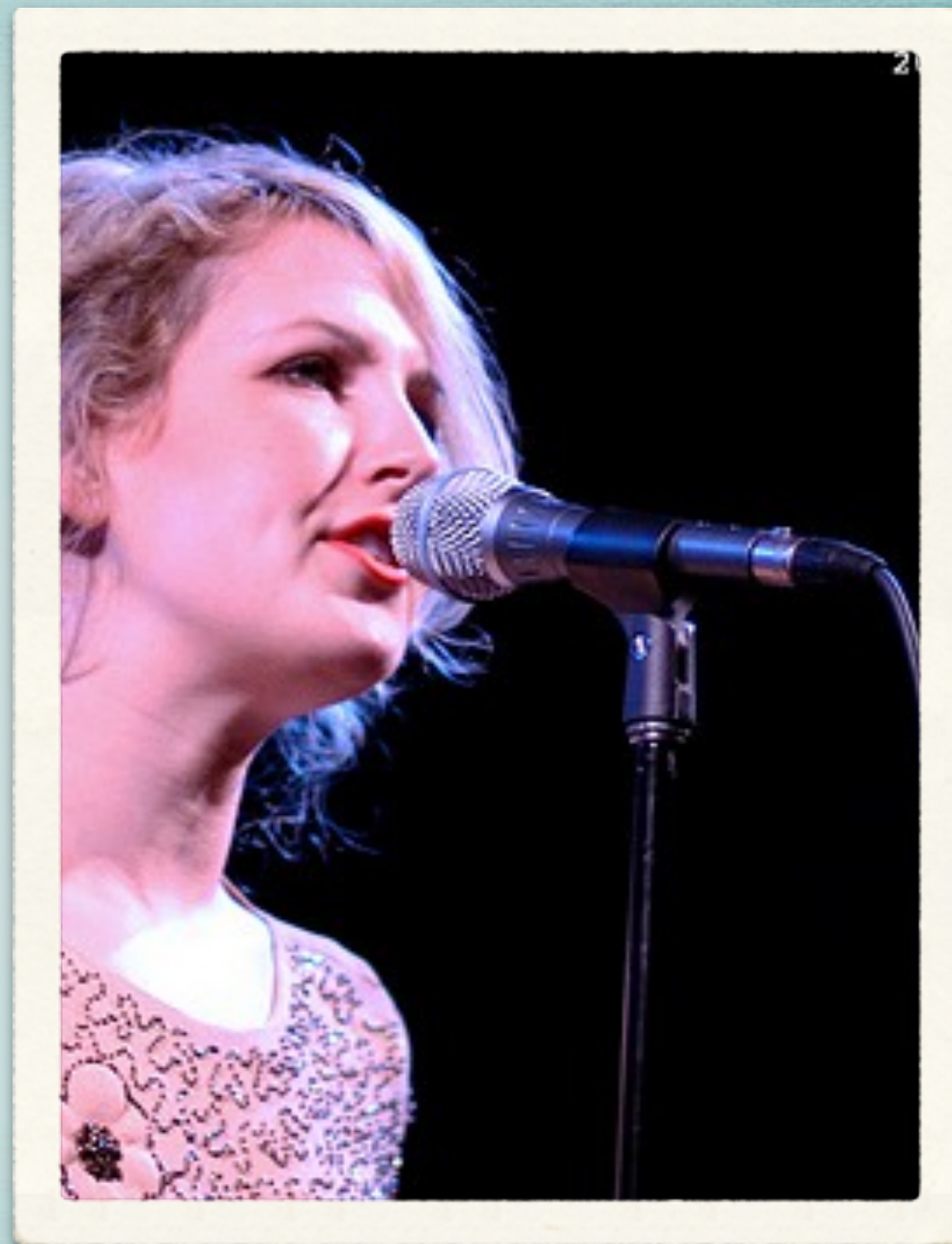
Lisp と の 出 会 い

- 大学の卒研でLispのインタプリター/コンパイラーの実装を行ったのが始まり
- 8bit CPU メモリー 64Kbyte :-)
- 雑誌に出ていた Lispインタプリターを移植・改造
- 先生・院生の作ったコンパイラーを移植 (Lisp, アセンブラー)

本日の内容

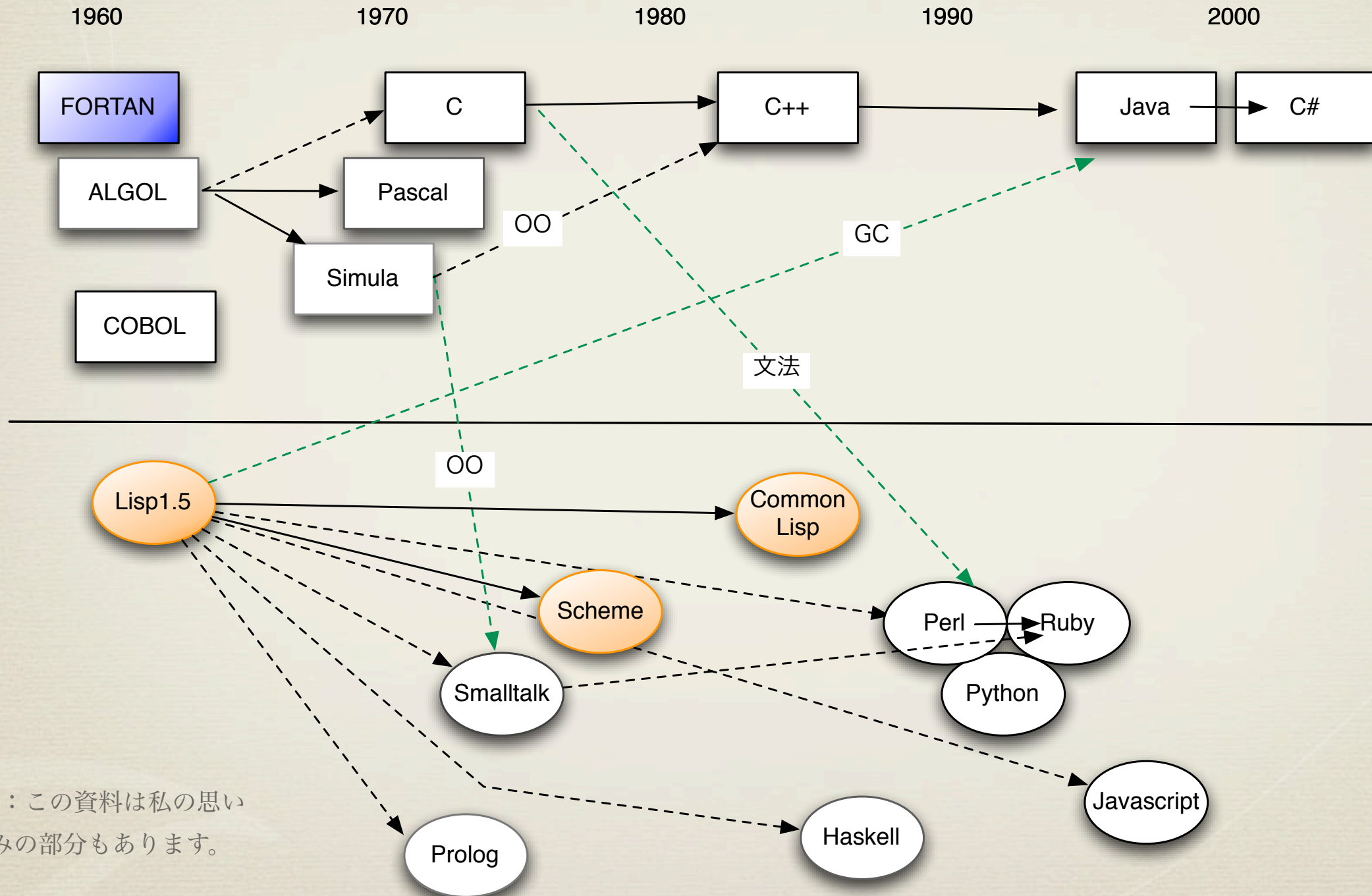
- Lispの紹介
- S式
- Lispプログラミング
- マクロ (Lisp処理系の仕組み)

Lispの紹介



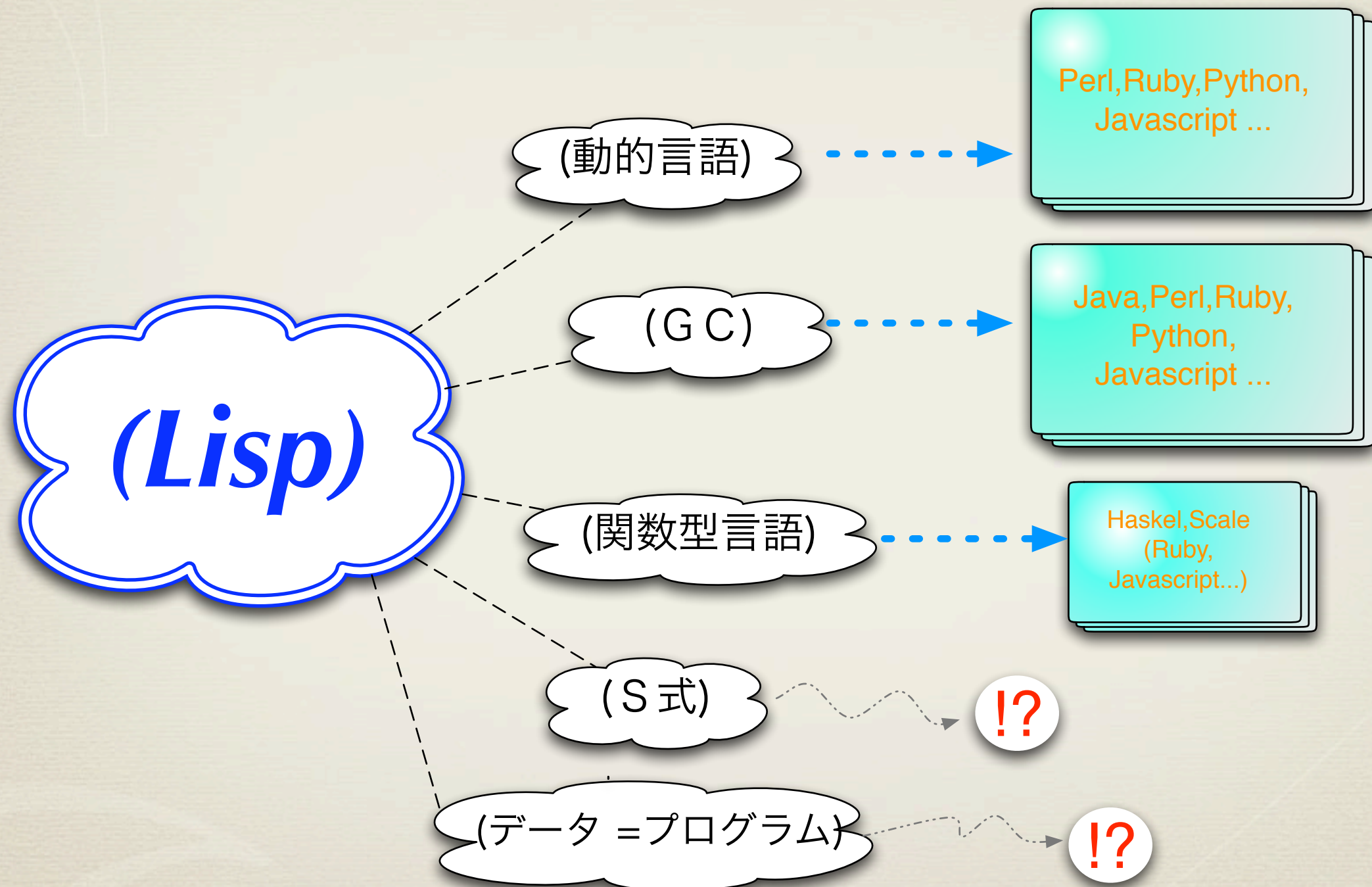
<http://www.flickr.com/photos/dust/3603580129/> (CC Attribution)

LispはLL言語のご先祖さま



注意：この資料は私の思いこみの部分もあります。

Lisp がもたらしたものの



動的言語

- 変数や関数を予め宣言しなくても良い
- 変数の型もあまり気にしなくて良い
- データの大きさもあまり気にしなくて良い
- そのコードが実行するまでに決定すれば良い

GC (Garbage collection)

- データ(メモリー)の回収は気にしないでよい
- 生産性増大
- 今では、静的言語(Java, C#)にもある
- GCのアルゴリズムは面白い
 - http://wiki.livedoor.jp/author_nari/d/GC
- でも、デバックスはたいへん (ˊˊ;)

関数型言語

- 関数のみ
- しかし、Lispは純粋な関数言語ではない
 - 副作用を持つ関数を許す
- 再帰呼び出し
- クロージャー (関数がファーストクラスオブジェクト)

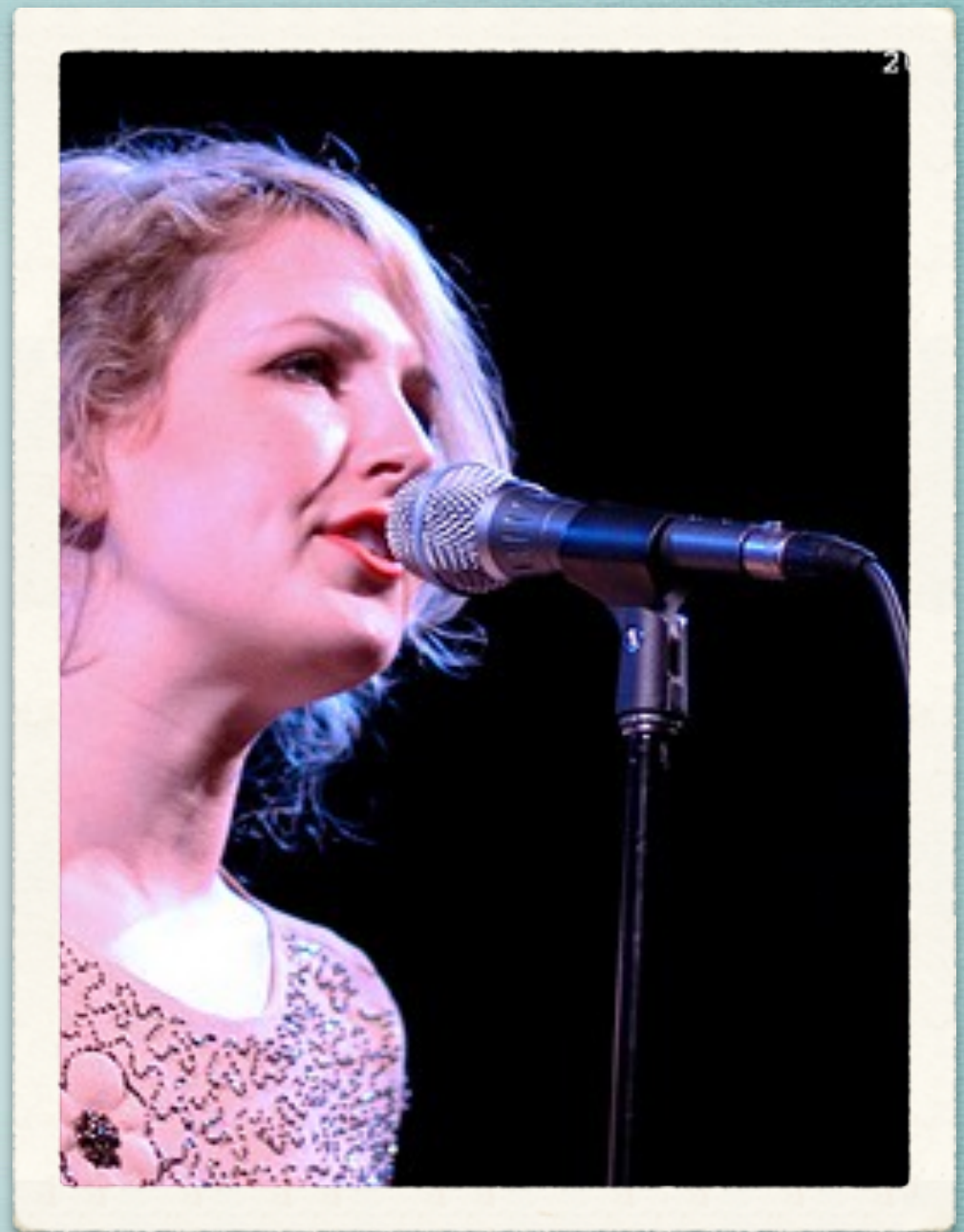
S 式 (Symbolic Expression)

- ((カッコ) が (た)(く)(さ)(ん))
- シンプルで強力
- なればカッコは気にならない
- いや、美しいと思うようになる (*^_^*)
- カッコの面倒は Emacs(vim)におまかせ

データ = プログラム

- Lispではデータもプログラムも S 式
- プログラムを使ってプログラムを生成できる
- 強力なマクロ機能

S式



<http://www.flickr.com/photos/dust/3603580129/> (CC Attribution)

S式は便利

- S式はいろいろなデータ構造を表す事が出来る
- データの挿入、削除のコストが低い
- アクセスは必ずしも高速ではない

S式とデータ構造

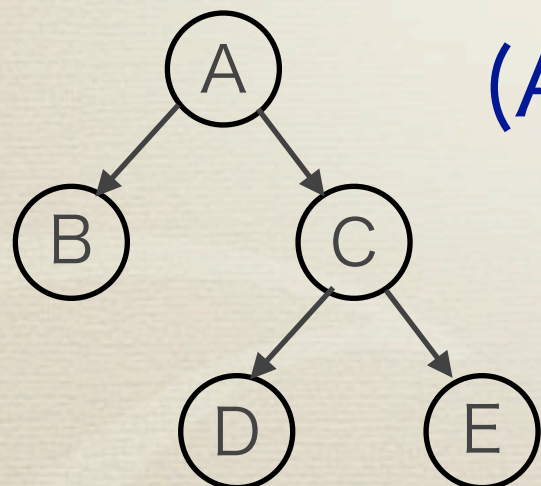
- 並び(リスト)

(gauche ypsilon mosh)

- 連想リスト

((gauche . kawai) (ypsilon . fujita) (mosh . minowa))

- 木構造



(A (B NIL NIL) (C (D NIL NIL) (E NIL NIL)))

ここでは 2分木を (ノード 左の子 右の子)
で表している

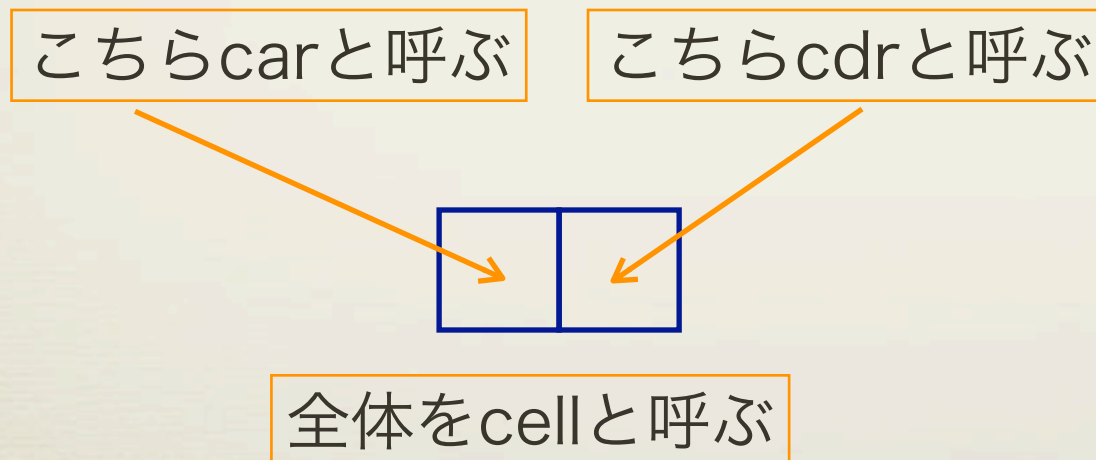
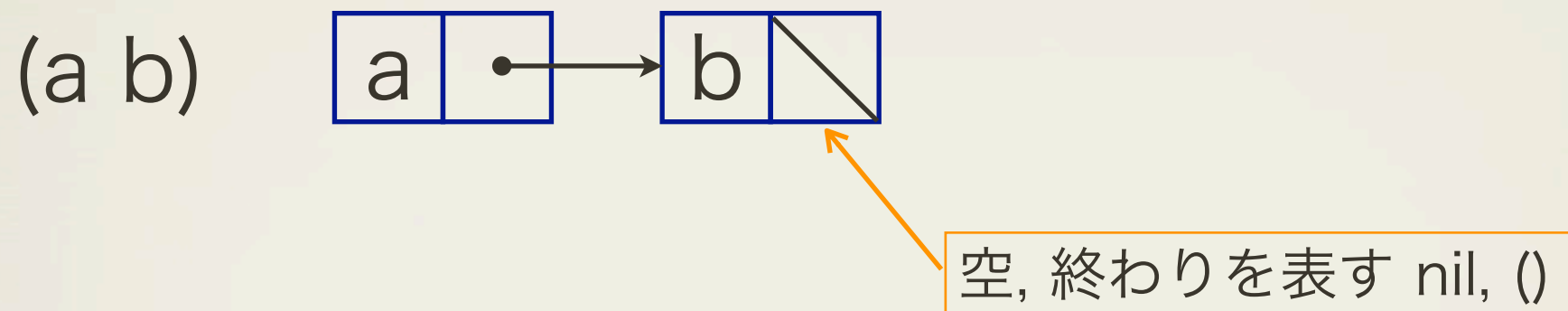
S式とデータ構造

■ HTMLをS式で表す

```
<html>
<body>
  <h1> Lisp </h1>
  <table>
    <tr><td>1</td> <td>lisp</td></tr>
    <tr><td>2</td> <td>scheme</td></tr>
  </table>
</body>
</html>
```

```
(html
 (body
  (h1 "Lisp")
  (table
   (tr (td 1) (td "lisp"))
   (tr (td 2) (td "scheme")))))
```


S式の内部表現



S式の内部表現

(a . b)

a	b
---	---

(a . (b . nil)) = (a b)

a	• → <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td style="padding: 2px 10px;">b</td><td style="padding: 2px 10px;">/</td></tr></table>	b	/
b	/		

(a b c)

a	• →
---	-----

b	• →
---	-----

c	/
---	---

(+ (* a b) (* c d))

+	• →
---	-----

•	• →
---	-----

•	/
---	---

*	• →
---	-----

c	• →
---	-----

d	/
---	---

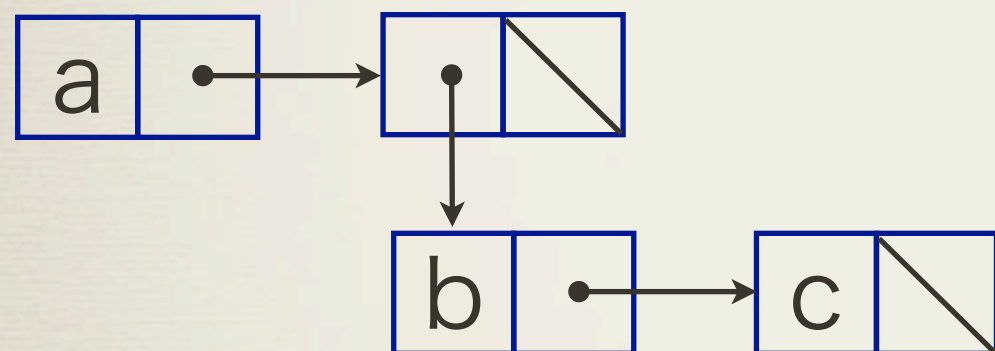
*	• →
---	-----

a	• →
---	-----

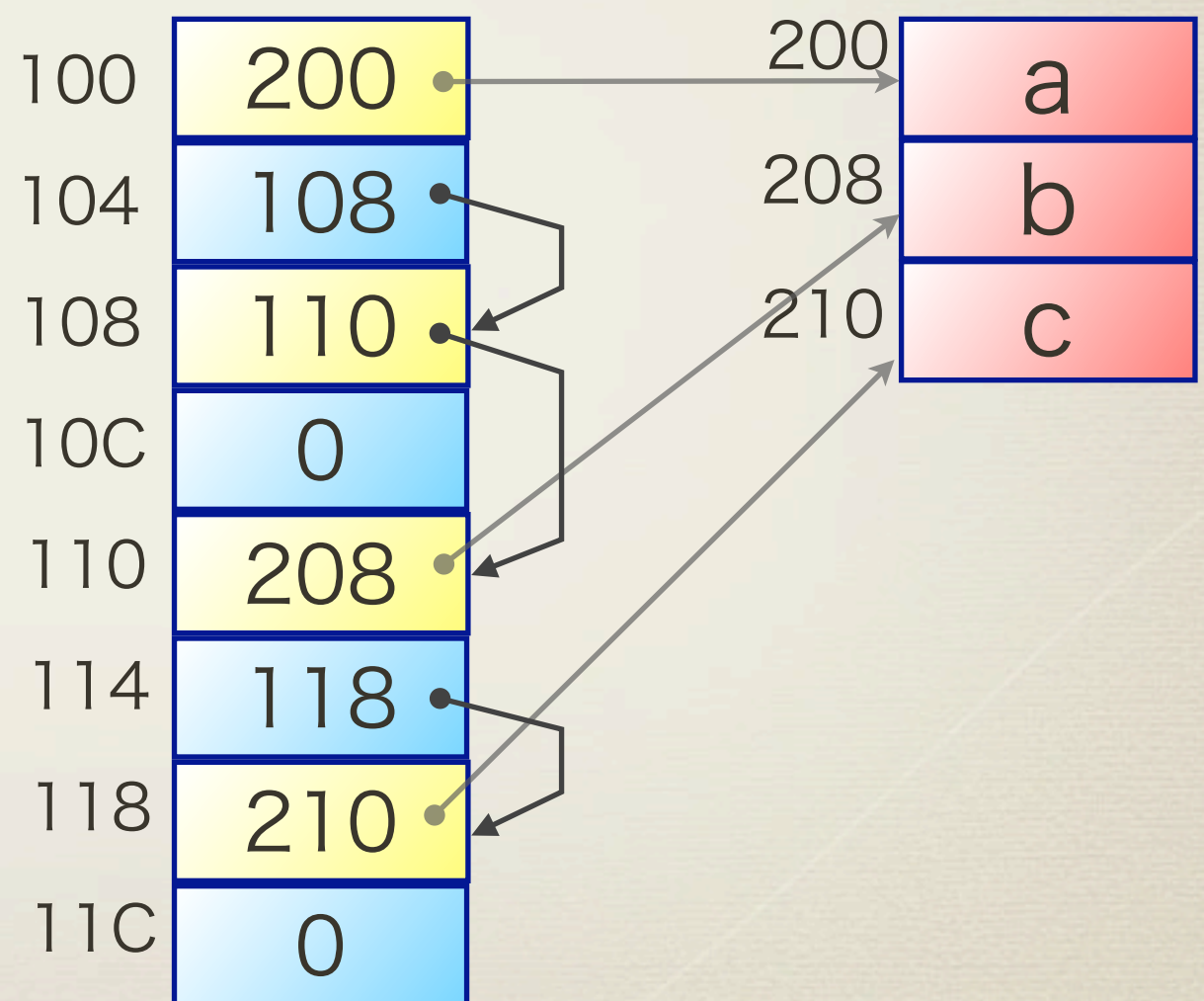
b	/
---	---

S式の内部表現

(a (b c))

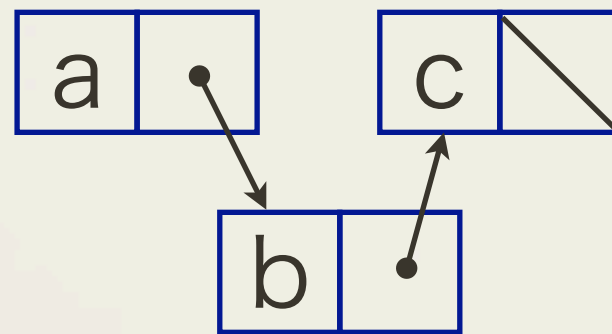
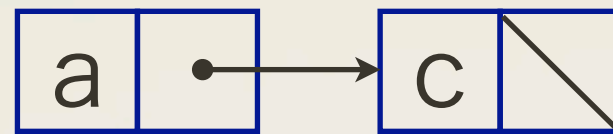


メモリー上



リストへの挿入

(a c)
↑
b



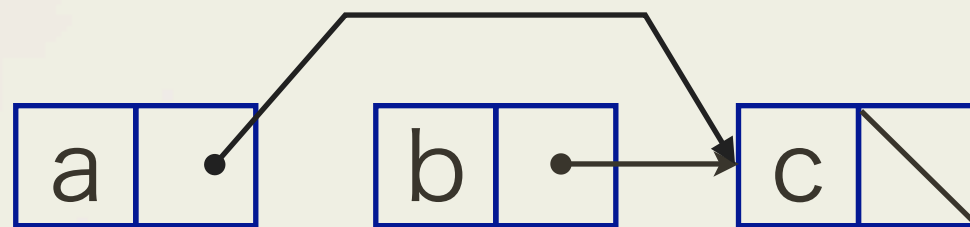
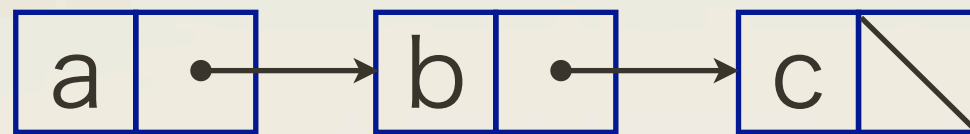
b用の箱(cell)を
用意しポインター
を書き換える

(a b c)



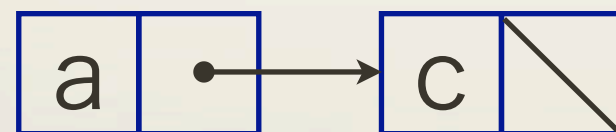
リストからの削除

(a ~~b~~ c)

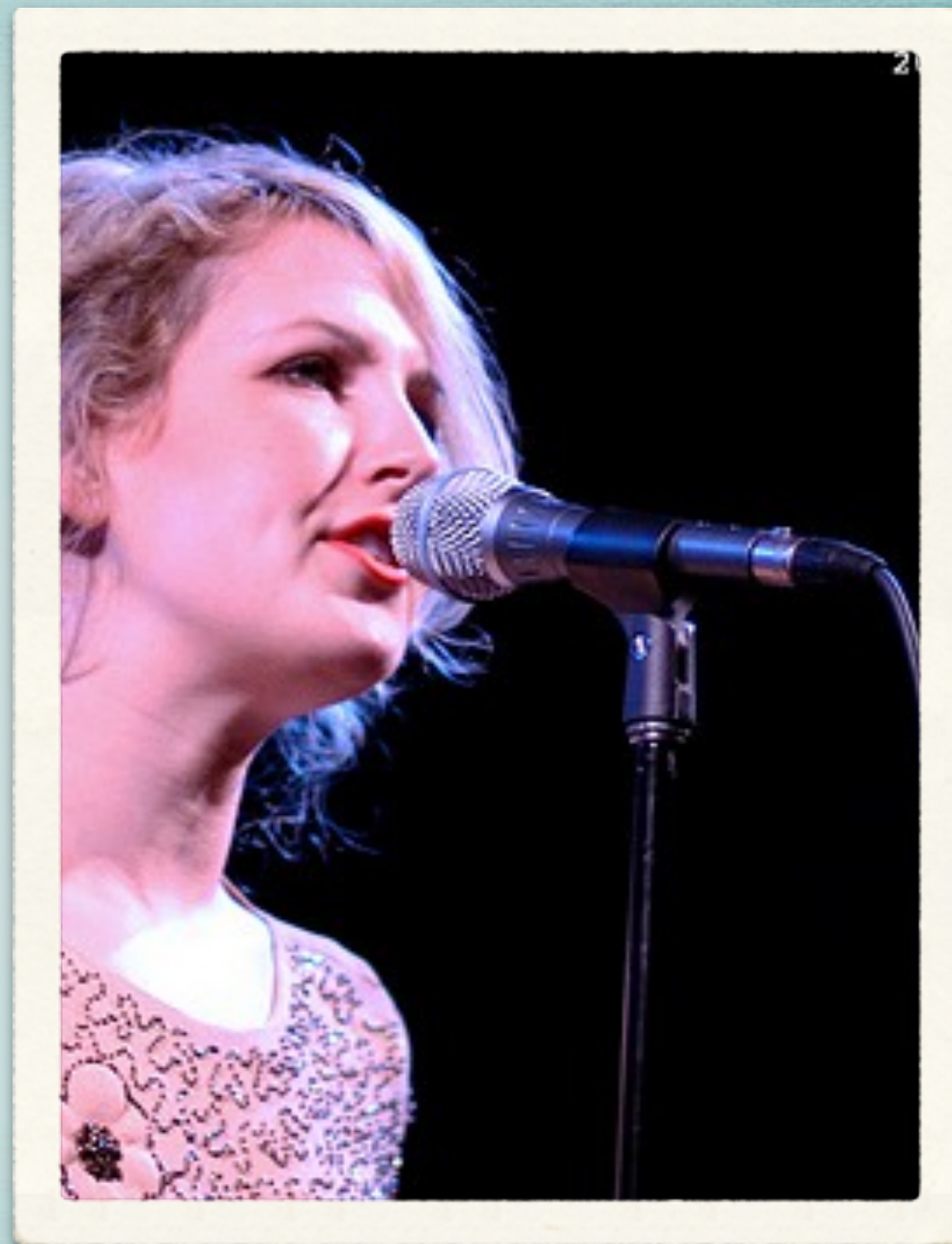


aのcdrのポインター
を書き換える。
bのcellがどこからも
参照されてなければ
GCで回収される

(a c)



Lisp プログラミング



<http://www.flickr.com/photos/dust/3603580129/> (CC Attribution)

簡単な演算

```
% r\lwrap gosh
```

Windowsは goshのみ

```
gosh> (+ 1 2)
```

```
3
```

```
gosh> (* 2 (+ 3 4))
```

```
14
```

```
gosh>
```


広域変数

```
gosh> (define a 3)
```

```
a
```

```
gosh> a
```

```
3
```

```
gosh> (define b (+ 2 5))
```

```
b
```

```
gosh> b
```

```
7
```

```
gosh> (+ a b)
```

```
10
```


広域変数

```
gosh> (set! a (+ a 1))  
4  
gosh> a  
4  
gosh> c  
*** ERROR: unbound variable: c  
gosh>
```


ローカル変数

```
gosh> (let ((a 2)
              (b 3))
        (set! a (+ a 1))
        (+ a b))
```

6

```
gosh>
```


関数定義

```
gosh> (define (add3 n) (+ n 3))
```

```
add3
```

```
gosh> (add3 5)
```

```
8
```

```
gosh> (add3 (add3 4))
```

```
10
```


クオート quote

- シンボルやS式を評価せず、そのまま値とする

```
gosh> c
```

```
*** ERROR: unbound variable: c
```

```
Stack Trace:
```

```
-----
```

```
gosh> (quote c)
```

```
c
```

```
gosh> 'c
```

```
c
```

```
gosh>
```


クオート quote

- シンボルやS式を評価せず値とする

```
gosh> (x y z)
*** ERROR: unbound variable: y
Stack Trace:
```

```
-----
gosh> '(x y z)
(x y z)
gosh> (append '(a b) '(c d))
(a b c d)
gosh>
```


S式の演算

```
gosh> (car '(a b c))
```

```
a
```

```
gosh> (cdr '(a b c))
```

```
(b c)
```

```
gosh> (cons 'a '(b c))
```

```
(a b c)
```

```
gosh>
```


再帰的定義

```
gosh>
(define (fact n)
  (if (= n 0)
      1
      (* n (fact (- n 1)))))

fact
gosh> (fact 3)
6
gosh> (fact 10)
3628800
gosh>
```

fact関数の定義
はエディターで
書きコピペする

例題

- S式で表したHTMLをHTMLに変換する

```
gosh> (define sample
' (html
  (body
    (h1 "Lisp")
    (table
      (tr (td 1) (td "lisp"))
      (tr (td 2) (td "scheme"))))))
```

sample

```
gosh> (print-html sample)
```

```
<html><body><h1>Lisp</h1><table><tr><td>1</td><td>lisp</td></tr><tr><td>2</td><td>scheme</td></tr></table></body></html>#<undef>
gosh>
```


考え方

- * htmlが文字列等なら、そのまま出力すればよい
- * htmlが (タグ 内容) なら <タグ> 内容 </タグ> と出力すればよい
- * 内容が空なら何もしない
- * 内容が複数ある場合は最初の内容を出力し、残りを処理する

* の関数

```
(define (print-html e)
  (if (list? e)
      (begin
        (format #t "<~a>" (car e))
        (print-html-list (cdr e))
        (format #t "</~a>" (car e)))
      (display e)))
```


* の関数

```
(define (print-html-list l)
  (if (null? l)
      #f
      (begin
        (print-html (car l))
        (print-html-list (cdr l))))))
```

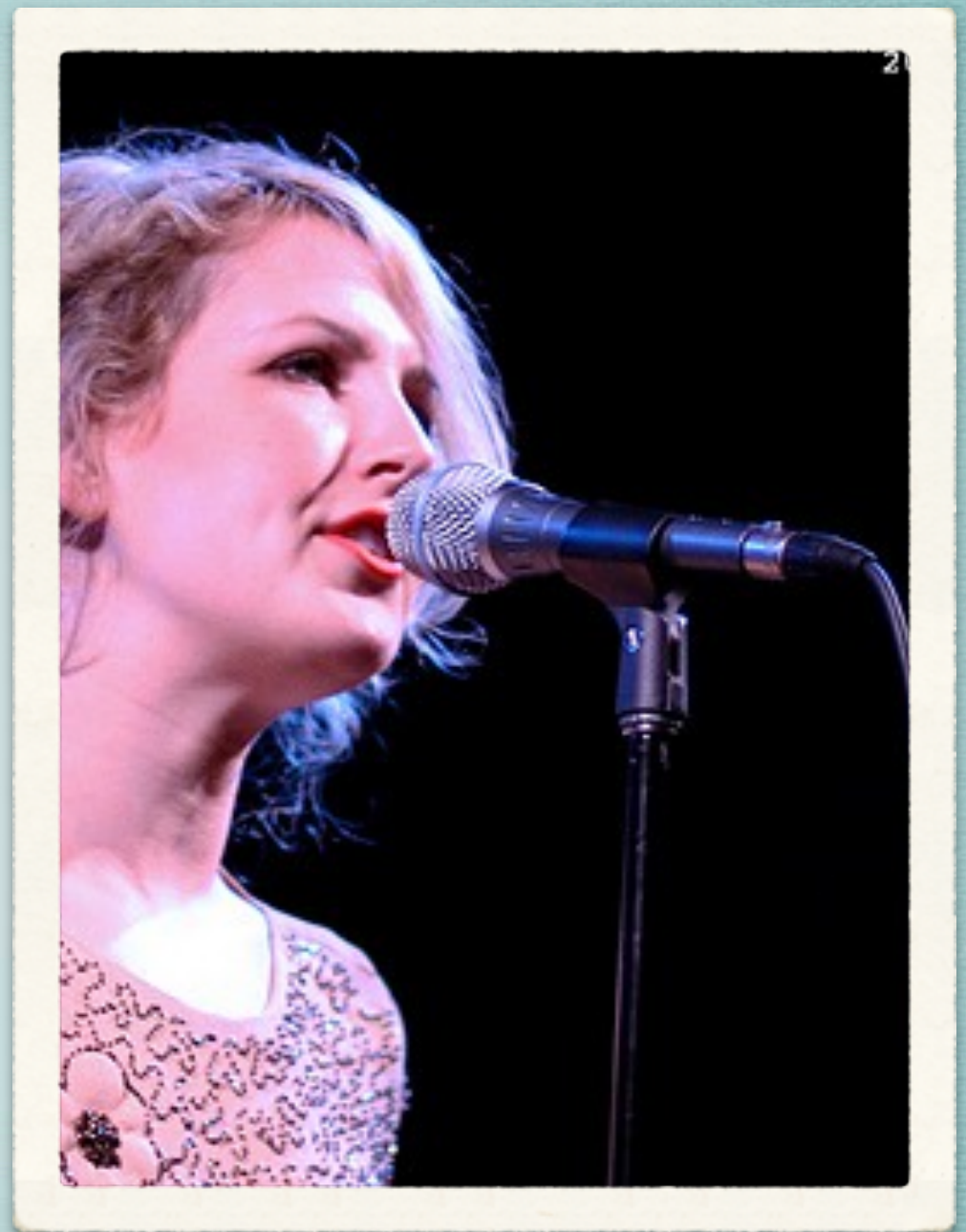

* の関数

- ループを使った例

```
(define (print-html-list l)
  (until (null? l)
    (print-html (car l))
    (set! l (cdr l))))
```


マクロ

Lisp処理系の仕組み



<http://www.flickr.com/photos/dust/3603580129/> (CC Attribution)

マクロ

- Lisp最強の武器と呼ばれている
- マクロの用途
 - 構文の拡張
 - DSL
 - ...

マクロ：構文の拡張

■ not-if を作る

```
> (if (not (= 0 1))  
      (print "not-eq")  
      (print "eq"))  
not-eq
```

```
> (not-if (= 0 1)  
          (print "not-eq")  
          (print "eq"))  
not-eq
```


マクロ：構文の拡張

■ not-if を作る

```
(define-macro (not-if test then else)
  (list 'if (list 'not test) then else))
```

```
gosh> (not-if (= 0 1) (print "not-eq") (print "eq"))
not-eq
#<undef>
gosh> (not-if (= 0 0) (print "not-eq") (print "eq"))
eq
#<undef>
```


マクロの原理

- マクロはマクロ定義を実行して出来たS式を再度実行する

```
(define-macro (not-if test then else)  
  (list 'if (list 'not test) then else))
```

```
> (let ((test '(= 0 1))  
      (then '(print "not-eq"))  
      (else '(print "eq")))  
  (list 'if (list 'not test) then else))  
  
(if (not (= 0 1)) (print "not-eq") (print "eq"))
```


マクロの原理

- マクロ定義はバッククオートを使うと簡単、読みやすい

```
(define-macro (not-if test then else)  
  `(if (not ,test) ,then ,else))
```

```
gosh> (let ((a 2) (b 3))  
        `(add ,a (sub 5 ,(+ b 1))))  
(add 2 (sub 5 4))  
gosh>
```


マクロ：DSL

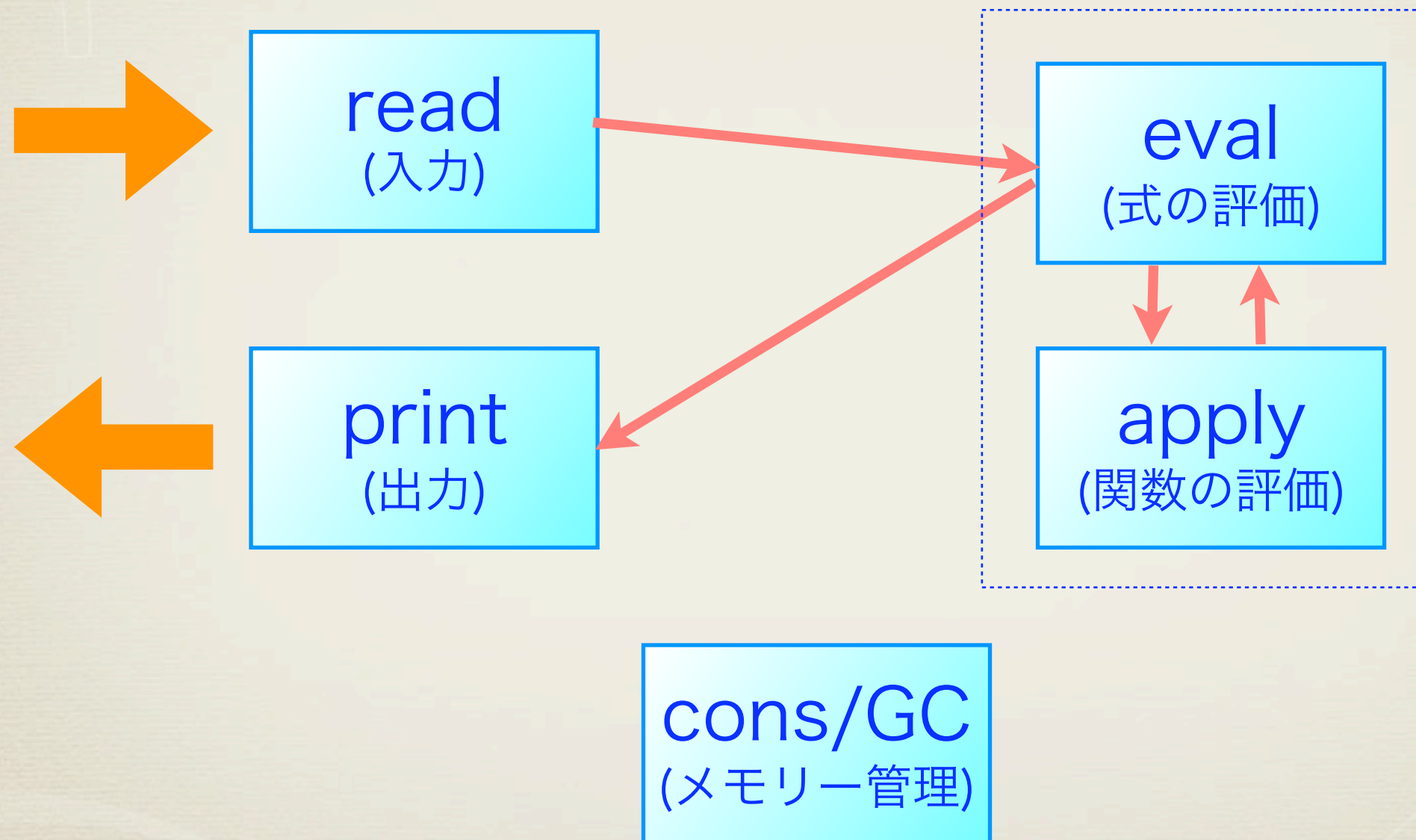
Gauche on Rails の コントローラーの一部

```
(define-action index
  (set! @todos (find <todo> :all)))

(define-action show
  (set! @todo (find <todo> (params :id))))

(define-action edit
  (set! @todo (find <todo> (params :id)))
  (define-continuation edit
    (when (update_attributes @todo (params :todo))
      (flash :notice "todo was successfully updated.")
      (redirect :action 'index))))
```


Lisp処理系



REPL

- Read Eval Print Loop

```
int main(int argc, char *argv[]) {  
    init();  
  
    while (1) {  
        print_e(eval(read_e()), NIL);  
    }  
}
```


eval (式の評価)

- e がリテラルならその値を返す。
- e がシンボルなら対応する値を返す。
- e がリストなら (car e) の値を関数、(cdr e) を引数とし (cdr e)を引数リストとして、applyで評価してもらう。

```
cell eval(cell e, cell env) {  
    cell v;  
    if (e == NIL || IS_INT(e)) return e;  
    if (IS_SYMBOL(e)) {  
        v = assoc(e, env);  
        if (v != NIL) return CDR(v);  
        throw_error("Undefined symbol '%s'",  
SYMBOL_NAME(e));  
    }  
    return apply(CAR(e), CDR(e), env);  
}
```


env (環境、変数の値)

- このLispでは実装の簡単な連想リストa-list を使って変数のbindを管理しています。
- 変数の値は(シンボル. 値)でbindされている。
- 変数の値は、毎回a-listの先頭から一致するシンボルを検索する。
- 関数呼び出し時には新しい値をa-listの先頭に追加する。
- a-listは環境としてインタプリタ内部を渡り歩く。

関数定義

```
(defun nlist(a n)
  (if (= n 0)
      nil
      (cons a (nlist a (- n 1)))))
```

実行経過

```
> (nlist 'x 2)
-->(cons 'x (nlist 'x 1))
-->(cons 'x (cons 'x (nlist 'x 0)))
<--(cons 'x (cons 'x nil))
<--(cons 'x ('x))
<--('x 'x)
```

a-listの内容

```
()
((a.x)(n.2))
((a.x)(n.1)(a.x)(n.2))
((a.x)(n.0)(a.x)(n.1)(a.x)(n.2))
((a.x)(n.1)(a.x)(n.2))
((a.x)(n.2))
()
```


関数

■ 関数の種類

- function : C言語で書かれた関数、引数は呼び出し前に評価する。 car, +, list
- special form : C言語で書かれた関数、引数は評価呼び出し前にしない。 setq, if
- lambda式 : Lispで書かれた関数
- macro : マクロの定義

関数

- (define (f x)...) は (define f (lambda(x) ...)) の構文糖

```
gosh> (define (add2 n) (+ n 2))
```

```
add2
```

```
gosh> (add2 3)
```

```
5
```

```
gosh> (define add2 (lambda(n) (+ n 2)))
```

```
add2
```

```
gosh> (add2 3)
```

```
5
```

```
gosh>
```


Lisp定義関数の評価

- 関数の評価： (apply func arg-list)
 - func の値が lambda式なら、 引数リストの各要素を eval で評価した値をlambda変数に対応付け、lambdaの式を eval で評価する。

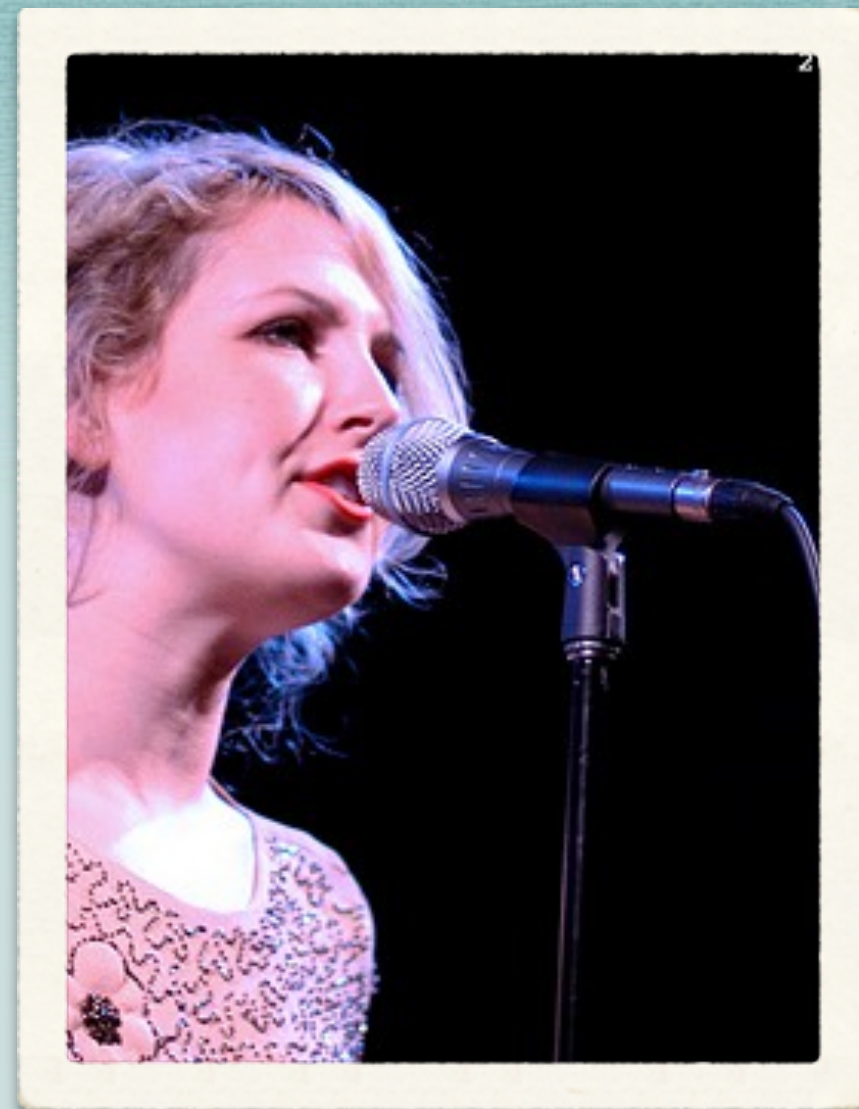
```
cell apply(cell func, cell args, cell env) {  
  cell fbody = eval(func, env);  
  cell ftype = CAR(fbody);  
  if (ftype == lambda_sym) {  
    cell params = CADR(fbody);  
    cell e = CADDR(fbody);  
    check_arg_count(func, list_len(params),  
                     list_len(args));  
    return eval(e, concat2(pairlis(params,  
                                   eval_list(args, env)),  
                           env));  
  }  
}
```

```
cell eval_list(cell list, cell env) {  
  cell values = NIL;  
  while (list != NIL) {  
    values = concat2(values,  
                     cons(eval(CAR(list), env),  
                           NIL));  
    list = CDR(list);  
  }  
  return values;  
}
```


マクロの評価

- マクロは定義を評価して出来た S 式を実行する

```
cell apply(cell func, cell args, cell env) {  
  cell fbody = eval(func, env);  
  cell ftype = CAR(fbody);  
  if (ftype == macro_sym) {  
    cell params = CADDR(fbody);  
    cell e = CADDR(fbody);  
    check_arg_count(func, list_len(params), list_len(args));  
  
    // macro展開  
    cell r = eval(e, concat2(pairlis(params, args), env));  
    return eval(r, env);  
  }  
}
```

<http://www.flickr.com/photos/dust/3603580129/> (CC Attribution)

おつかれさまでした！