(TUT-)Scheme入門

学術情報メディアセンター
メディアコンピューティング分野
助教　　平石 拓

今日話す内容

- Lisp (Scheme)**の基本を一通り解説**
  - SICP**の**1.1**〜**1.1.6
  - 1.1.6**までに載ってないけど知っておいたほうがいいこと**
    - **リスト** (lists)
    - **局所変数** (local variables: let)
    - **コンス・セル** (cons cells)
    - **ファイル入出力** (file I/O)
    - **トレース** (tracing)

Lisp言語

- John McCarthy **によって発明**（1958**年**）
- FORTRAN（1957**年**）**に次いで**2**番目に古い**
- **特徴**
  - **リスト処理が得意**（List Processor)

Lisp言語

- John McCarthy **によって発明（**1958**年）**
- FORTRAN（1957**年）に次いで**2**番目に古い**
- **特徴**
  - **リスト処理が得意（**List Processor)
  - **対話環境**
  - **動的型付け，**closure**オブジェクト，．．．**

Lisp言語

- John McCarthy **によって発明（**1958**年）**
- FORTRAN（1957**年）に次いで**2**番目に古い**
- **特徴**
  - **リスト処理が得意（**List Processor)
  - **対話環境**
  - **動的型付け，**closure**オブジェクト，．．．**
  - **「やりたいこと」だけに集中してプログラムが書ける**
    - *rapid prototyping*

Lisp言語

- John McCarthy によって発明（1958年）
- FORTRAN（1957年）に次いで2番目に古い
- 特徴
  - リスト処理が得意（List Processor)
  - 対話環境
  - 動的型付け，closureオブジェクト，．．．
  - 「やりたいこと」だけに集中してプログラムが書ける
    - *rapid prototyping*
  - Lispのプログラムを Lisp自身で扱うことができる
    - 書きたいプログラムに合わせて言語自体をカスタマイズ可能

Lispの方言

- Common Lisp
- Scheme
- Emacs Lisp
- AutoLisp
- Lisp1.5
- MACLISP
- ISLisp
- …

Scheme

- Lispの方言の一つ
- プログラミング言語として本当に必要な部分だけをできるだけコンパクトにまとめた仕様
  - 言語仕様(R5RS) は50ページ
    （2007/09に成立した新仕様(R6RS)で3倍以上に増え，一部反発）
    - C（C99）は538ページ
    - Common Lisp（第2版）は1029ページ

Scheme

- Lispの方言の一つ
- プログラミング言語として本当に必要な部分だけをできるだけコンパクトにまとめた仕様
  - 言語仕様(R5RS) は50ページ
    （2007/09に成立した新仕様(R6RS)で3倍以上に増え，一部反発）
    - C（C99）は538ページ
    - Common Lisp（第2版）は1029ページ
- 継続オブジェクト
  - 実行中の処理の「残りの計算」をプログラムで扱える
- 真の末尾再帰呼び出しをサポート
  - 繰り返し構文（Cでいうwhileなど）を持たない

Scheme処理系の入手

- TUT-Scheme
  - **湯淺先生，小宮先生（元湯淺研）開発**
  - **利用手段**
    - **メディアセンターの端末** (Linux)
    - http://www.spa.is.uec.ac.jp/~komiya/download/ **から入手**
      - Windows（Cygwin**版もあり**），Mac OS X，Linux
- JAKLD (JAva Kumikomi-you Lisp Driver)
  - **湯淺先生開発**
  - http://ryujin.kuis.kyoto-u.ac.jp/~yuasa/jakld/index-j.html
  - JVM**が動く環境ならどこでも動く**
  - i**アプリ**（DoCoMo）**版もある！**
- MIT Scheme

起動・終了

```
% java jakld.Eval
JAKLD for SICP (October 10, 2008)
(c) Copyright Taiichi Yuasa, 2002.  All rights reserved.
> (+ 3 4)
7
> Ctrl+C
Bye.
%
```

起動

終了

処理系によっては(bye), (exit), (quit)

対話環境 (REPL: Read Eval Print Loop)

> (+ 3 4)

(+ 3 4)の評価値

> (* (+ 1 2) (- 10 7))

(* (+ 1 2) (- 10 7))の評価値

> 1234

1234の評価値

> (< (* 3 3) 10)

(< (* 3 3) 10)の評価値

対話環境 (REPL: Read Eval Print Loop)

> (+ 3 4)

7 ---------- (+ 3 4)の評価値

> (* (+ 1 2) (- 10 7))

■ ---------- (* (+ 1 2) (- 10 7))の評価値

> 1234

■ ---------- 1234の評価値

> (< (* 3 3) 10)

■ ---------- (< (* 3 3) 10)の評価値

対話環境 (REPL: Read Eval Print Loop)

> (+ 3 4)

7 ---------- (+ 3 4)の評価値

> (* (+ 1 2) (- 10 7))

9 ---------- (* (+ 1 2) (- 10 7))の評価値

> 1234

 ---------- 1234の評価値

> (< (* 3 3) 10)

 ---------- (< (* 3 3) 10)の評価値

対話環境 (REPL: Read Eval Print Loop)

> (+ 3 4)

7 ---------- (+ 3 4)の評価値

> (* (+ 1 2) (- 10 7))

9 ---------- (* (+ 1 2) (- 10 7))の評価値

> 1234

1234 ---------- 1234の評価値

> (< (* 3 3) 10)

 ---------- (< (* 3 3) 10)の評価値

対話環境 (REPL: Read Eval Print Loop)

> (+ 3 4)

7 ---------- (+ 3 4)の評価値

> (* (+ 1 2) (- 10 7))

9 ---------- (* (+ 1 2) (- 10 7))の評価値

> 1234

1234 ---------- 1234の評価値

> (< (* 3 3) 10)

#t ---------- (< (* 3 3) 10)の評価値

変数定義（１）

```
> (define x 10)
x
> x
[redacted]
> (* x (+ x 3))
[redacted]
> (set! x 24)
> x
[redacted]
```

x という名前の変数を定義

xの値を変更

変数定義（１）

```
> (define x 10)
x
> x
10
> (* x (+ x 3))
```

```
> (set! x 24)
> x
```

x という名前の変数を定義

xの値を変更

変数定義（１）

```
>(define x 10)
x
>x
10
>(* x (+ x 3))
130
>(set! x 24)
>x
```

x という名前の変数を定義

xの値を変更

変数定義（１）

```
> (define x 10)
x
> x
10
> (* x (+ x 3))
130
> (set! x 24)
> x
24
```

x という名前の変数を定義

xの値を変更

変数定義（２）：局所変数

```
> (let ((x 10)
        (y 20))
    (* x (+ x y)))
```

```
> x
```

変数定義（２）：局所変数

```
> (let ((x 10)
        (y 20))
    (* x (+ x y)))
300
> x
```

変数定義（２）：局所変数

```
> (let ((x 10)
        (y 20))
    (* x (+ x y)))
300
> x
Error:  x is an unbound symbol.
```

関数定義

```
> (define (square x) (* x x))

> (square 10)
```

```
> (square (* 3 4))
```

関数定義

```
> (define (square x) (* x x))

> (square 10)
```

```
> (square (* 3 4))
```

関数の名前

関数定義

```
> (define (square x) (* x x))

> (square 10)

> (square (* 3 4))
```

関数のパラメータ

関数の名前

関数定義

```
> (define (square x) (* x x))

> (square 10)
```

```
> (square (* 3 4))
```

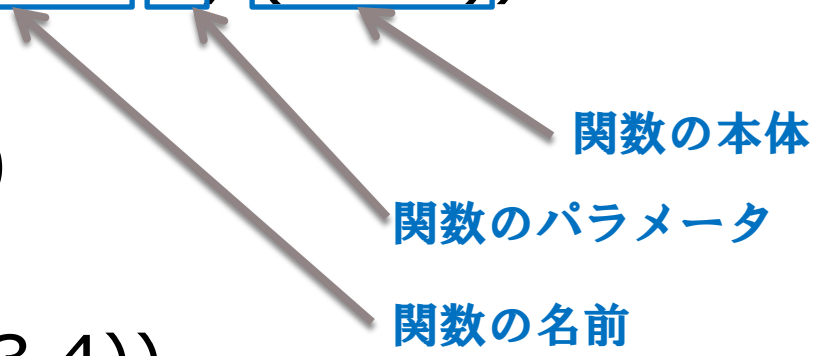
関数の本体

関数のパラメータ

関数の名前

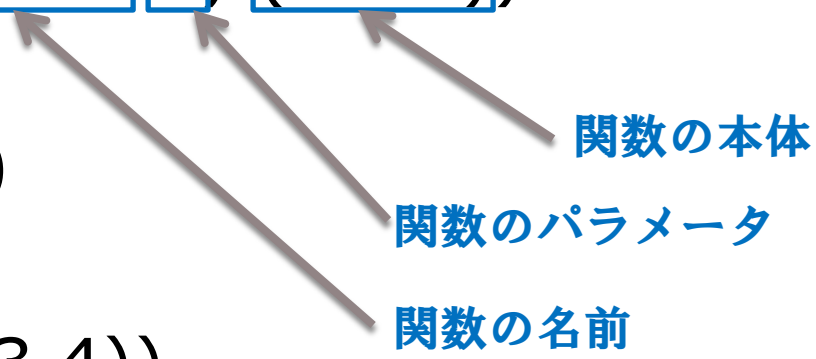
関数定義

```
> (define (square x) (* x x))

> (square 10)
100
> (square (* 3 4))
```



関数定義

```
> (define (square x) (* x x))

> (square 10)
100
> (square (* 3 4))
144
```



階乗の計算

- **n! を求める関数**

```
> (define (fact n)
```

```
> (fact 10)
```

階乗の計算

- **n! を求める関数**

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

> (fact 10)
```

n=0の時

n=0でない時

階乗の計算

- **n! を求める関数**

```
> (define (fact n)
    (if (= n 0)
        1                          ---------- n=0の時
        (* n (fact (- n 1)))))     -------- n=0でない時

> (fact 10)
362800
```

Fibonacci数の計算

- 1, 1, 2, 3, 5, 8, 13, …
- fib(n) =

```
> (define (fib n)
```

```
fib
> (fib 10)
89
```

Fibonacci数の計算

- 1, 1, 2, 3, 5, 8, 13, …
- fib(n) = fib(n-1) + fib (n-2)

```
> (define (fib n)
```

```
fib
> (fib 10)
89
```

Fibonacci数の計算

- 1, 1, 2, 3, 5, 8, 13, …
- fib(n) = fib(n-1) + fib (n-2)

```
> (define (fib n)
      (if (< n 2)          ; fib(0) = fib(1) = 1
          1
          (+ (fib (- n 1)) (fib (- n 2)))))
fib
> (fib 10)
89
```

引用符（１）

- (quote 〈式〉)・・・スペシャル・フォーム

```
>(quote x)
[redacted]
>(quote (+ 3 4))
[redacted]
>(define x (quote y))
>x
[redacted]
```

引用符（１）

- (quote 〈式〉)・・・スペシャル・フォーム

```
> (quote x)
x
> (quote (+ 3 4))
```

```
> (define x (quote y))
> x
```

引用符（１）

- (quote 〈式〉)・・・スペシャル・フォーム

```
> (quote x)
x
> (quote (+ 3 4))
(+ 3 4)
> (define x (quote y))
> x
```

引用符（１）

- (quote 〈式〉)・・・スペシャル・フォーム

```
> (quote x)
x
> (quote (+ 3 4))
(+ 3 4)
> (define x (quote y))
> x
y
```

引用符（２）

- **’〈式〉でも同じ意味**

```
> 'x
x
> '(+ 3 4)
(+ 3 4)
> (define x 'y)
> x
y
```

リスト

- **Lispにおける最も重要なデータ型の1つ**
  - Lisp = List Processor
- **データ（要素）の“並び”を表す**
  - (1 2 3 4 5 6 7 8 9 10)
  - (we eat rice)
  - ((lions 0.543) (buffaloes 0.524) (fighters 0.514) (marines 0.510) (eagles 0.461) (hawks 0.454))

リストの生成（1）

```
> (list 1 2 3 4)
```

```
> (list 'w 'x (list 'y 'z))
```

```
> (define x 4)
> (list x (* x 5))
```

```
> (list)
```

空リスト

リストの生成（1）

```
> (list 1 2 3 4)
(1 2 3 4)
> (list 'w 'x (list 'y 'z))
```

```
> (define x 4)
> (list x (* x 5))
```

```
> (list)
```

空リスト

リストの生成（1）

```
> (list 1 2 3 4)
(1 2 3 4)
> (list 'w 'x (list 'y 'z))
(w x (y z))
> (define x 4)
> (list x (* x 5))
```

```
> (list)
```

空リスト

リストの生成（1）

```
> (list 1 2 3 4)
(1 2 3 4)
> (list 'w 'x (list 'y 'z))
(w x (y z))
> (define x 4)
> (list x (* x 5))
(4 20)
> (list)
```

空リスト

リストの生成（1）

```
> (list 1 2 3 4)
(1 2 3 4)
> (list 'w 'x (list 'y 'z))
(w x (y z))
> (define x 4)
> (list x (* x 5))
(4 20)
> (list)
()
```

空リスト

リストの生成（2）

- **リストの要素がわかっている場合は、（quote 〈リストを表す式〉）でもよい。**

> '(x y) ---------- (quote (x y)) の略記

(x y)

> '((x y) 1 2 (a b c))

((x y) 1 2 (a b c))

> '(define (square x) (* x x))

(define (square x) (x x))*

リストの要素の参照

>(car '(a b c d)) ---------- リストの先頭要素

>(cdr '(a b c d)) ---------- 先頭要素を除いたリスト

>(car (cdr (cdr '(a b c d))))

>(cdr (cdr (cdr (cdr '(a b c d)))))

リストの要素の参照

> (car '(a b c d)) ---------- リストの先頭要素

a

> (cdr '(a b c d)) ---------- 先頭要素を除いたリスト

> (car (cdr (cdr '(a b c d))))

> (cdr (cdr (cdr (cdr '(a b c d)))))

リストの要素の参照

> (car '(a b c d)) ---------- リストの先頭要素

a

> (cdr '(a b c d)) ---------- 先頭要素を除いたリスト

(b c d)

> (car (cdr (cdr '(a b c d))))

> (cdr (cdr (cdr (cdr '(a b c d)))))

リストの要素の参照

> (car '(a b c d)) ---------- リストの先頭要素

a

> (cdr '(a b c d)) ---------- 先頭要素を除いたリスト

(b c d)

> (car (cdr (cdr '(a b c d))))

c

> (cdr (cdr (cdr (cdr '(a b c d)))))

リストの要素の参照

> (car '(a b c d)) ---------- リストの先頭要素

a

> (cdr '(a b c d)) ---------- 先頭要素を除いたリスト

(b c d)

> (car (cdr (cdr '(a b c d))))

c

> (cdr (cdr (cdr (cdr '(a b c d)))))

()

リストへの要素追加

- **リストの先頭に要素を追加**

```
>(cons 'we '(eat rice))
[redacted]
>(cons 'never (cdr '(we eat rice)))
[redacted]
>(cons '(a b c) '(x y z))
[redacted]
>(cons 'single '() )
[redacted]
```

リストへの要素追加

- **リストの先頭に要素を追加**

> (cons 'we '(eat rice))

(we eat rice)

> (cons 'never (cdr '(we eat rice)))

> (cons '(a b c) '(x y z))

> (cons 'single '())

リストへの要素追加

- **リストの先頭に要素を追加**

> (cons 'we '(eat rice))

(we eat rice)

> (cons 'never (cdr '(we eat rice)))

(never eat rice)

> (cons '(a b c) '(x y z))

> (cons 'single '())

リストへの要素追加

- **リストの先頭に要素を追加**

```
> (cons 'we '(eat rice))
(we eat rice)
> (cons 'never (cdr '(we eat rice)))
(never eat rice)
> (cons '(a b c) '(x y z))
((a b c) x y z)
> (cons 'single '() )
```

リストへの要素追加

- **リストの先頭に要素を追加**

```
> (cons 'we '(eat rice))
(we eat rice)
> (cons 'never (cdr '(we eat rice)))
(never eat rice)
> (cons '(a b c) '(x y z))
((a b c) x y z)
> (cons 'single '() )
(single)
```

リストの長さを求める関数

```
(define (my-length x)
```

*x*が()か？

リストの長さを求める関数

- **「リストの長さ」の定義は？**

(define (my-length x)

*x*が()か？

リストの長さを求める関数

- **「リストの長さ」の定義は？**
  - (length ()) ＝ 0

(define (my-length x)

x**が**()**か？**

リストの長さを求める関数

- **「リストの長さ」の定義は？**
  - (length ()) ＝ 0
  - (length '(*a b* …)) ＝ 1 + (length '(*b* …))

```
(define (my-length x)
```

*x*が()か？

リストの長さを求める関数

- **「リストの長さ」の定義は？**
  - (length ()) ＝ 0
  - (length '(*a b* …)) ＝ 1 + (length '(*b* …))

```
(define (my-length x)
  (if (null? x)
      0
      (+ 1 (my-length (cdr x)))))
```

x **が** () **か？**

リストを結合する関数

- (append '(a b c) '(1 2 3)) → (a b c 1 2 3)
- **「リストを結合する」の定義は？**

(define (my-append x y)

リストを結合する関数

- (append '(a b c) '(1 2 3)) → (a b c 1 2 3)
- **「リストを結合する」の定義は？**
  - (append () *y*) ＝ 
    

(define (my-append x y)

リストを結合する関数

- (append '(a b c) '(1 2 3)) → (a b c 1 2 3)
- **「リストを結合する」の定義は？**
  - (append () *y*) ＝ 
    

(define (my-append x y)

リストを結合する関数

- (append '(a b c) '(1 2 3)) → (a b c 1 2 3)
- **「リストを結合する」の定義は？**
  - (append () *y*) ＝ 
    

(define (my-append x y)

リストを結合する関数

- (append '(a b c) '(1 2 3)) → (a b c 1 2 3)
- **「リストを結合する」の定義は？**
  - (append () *y*) ＝ 
  - (append '(*a b* …) *y*) ＝ ( (append '(*b* …) *y*))

(define (my-append x y)

リストを結合する関数

- (append '(a b c) '(1 2 3)) → (a b c 1 2 3)
- **「リストを結合する」の定義は？**
  - (append () *y*) ＝ *y*
  - (append '(*a b* …) *y*) ＝ ([redacted] (append '(*b* …) *y*))

(define (my-append x y)

[redacted]

リストを結合する関数

- (append '(a b c) '(1 2 3)) → (a b c 1 2 3)
- **「リストを結合する」の定義は？**
  - (append () *y*) ＝ *y*
  - (append '(*a b* …) *y*) ＝ (cons *a* (append '(*b* …) *y*))

(define (my-append x y)

リストを結合する関数

- (append '(a b c) '(1 2 3)) → (a b c 1 2 3)
- **「リストを結合する」の定義は？**
  - (append () *y*) ＝ *y*
  - (append '(*a b* …) *y*) ＝ (cons *a* (append '(*b* …) *y*))

```
(define (my-append x y)
   (if (null? x)
      y
      (cons (car x) (my-append (cdr x) y))))
```

リストを逆順に並べ換える関数

- (reverse '(a b c)) → (c b a)

(define (my-reverse x)

リストを逆順に並べ換える関数

- (reverse '(a b c)) → (c b a)
- **「リストを逆順にする」の定義は？**

(define (my-reverse x)

リストを逆順に並べ換える関数

- (reverse '(a b c)) → (c b a)
- **「リストを逆順にする」の定義は？**

(define (my-reverse x)

リストを逆順に並べ換える関数

- (reverse '(a b c)) → (c b a)
- **「リストを逆順にする」の定義は？**
  - (reverse ()) ＝ [redacted]
  - (reverse (*a b* …)) ＝ [redacted] (reverse '(b …)) [redacted]

(define (my-reverse x)

[redacted]

リストを逆順に並べ換える関数

- (reverse '(a b c)) → (c b a)
- **「リストを逆順にする」の定義は？**
  - (reverse ()) ＝ ()
  - (reverse (*a b* …)) ＝ [redacted] (reverse '(b …)) [redacted]

(define (my-reverse x)

リストを逆順に並べ換える関数

- (reverse '(a b c)) → (c b a)
- **「リストを逆順にする」の定義は？**
  - (reverse ()) ＝ ()
  - (reverse (*a b* …)) ＝ (append (reverse '(b …)) '(*a*))

(define (my-reverse x)

リストを逆順に並べ換える関数

- (reverse '(a b c)) → (c b a)
- **「リストを逆順にする」の定義は？**
  - (reverse ()) ＝ ()
  - (reverse (*a b* …)) ＝ (append (reverse '(b …)) '(*a*))

```
(define (my-reverse x)
   (if (null? x)
      ()
      (append (my-reverse (cdr x))
                  (list (car x)))))
```

リストを逆順に並べ換える関数

- (reverse '(a b c)) → (c b a)
- **「リストを逆順にする」の定義は？**
  - (reverse ()) ＝ ()
  - (reverse (*a b* …)) ＝ (append (reverse '(b …)) '(*a*))

```
(define (my-reverse x)
   (if (null? x)
      ()
      (append (my-reverse (cdr x))
                  (list (car x)))))
```

リストを逆順に並べ換える関数

- (reverse '(a b c)) → (c b a)
- **「リストを逆順にする」の定義は？**
  - (reverse ()) ＝ ()
  - (reverse (*a b* …)) ＝ (append (reverse '(b …)) '(*a*))

```
(define (my-reverse x)
   (if (null? x)
      ()
      (append (my-reverse (cdr x))
                  (list (car x)))))
```

リストを逆順に並べ換える関数

- (reverse '(a b c)) → (c b a)
- **「リストを逆順にする」の定義は？**
  - (reverse ()) ＝ ()
  - (reverse (*a b* …)) ＝ (append (reverse '(b …)) '(*a*))

```
(define (my-reverse x)
   (if (null? x)
      ()
      (append (my-reverse (cdr x))
                  (list (car x)))))
```

リストを逆順に並べ換える関数

- (reverse '(a b c)) → (c b a)
- **「リストを逆順にする」の定義は？**
  - (reverse ()) ＝ ()
  - (reverse (*a b* …)) ＝ (append (reverse '(b …)) '(*a*))

```
(define (my-reverse x)
   (if (null? x)
      ()
      (append (my-reverse (cdr x))
                  (list (car x)))))
```

効率は悪い

リストが要素を含むか判定

- (member 4 '(1 3 5 7 9)) → #f
- (member 5 '(1 3 5 7 9)) → (5 7 9)
- **定義は？**

リストが要素を含むか判定

- (member 4 '(1 3 5 7 9)) → #f
- (member 5 '(1 3 5 7 9)) → (5 7 9)
- **定義は？**
  - (define (my-member x lst)
    (if (null? lst)
    #f
    (if (equal? x (car lst))
    lst
    (my-member x (cdr lst)))))

リスト処理の練習問題（自習用）

リスト処理の練習問題（自習用）

- `but-last` を定義せよ.

リスト処理の練習問題（自習用）

- but-last を定義せよ.
- (but-last '(1 2 3 4 5)) ⇒ (1 2 3 4)

リスト処理の練習問題（自習用）

- but-last を定義せよ.
- (but-last '(1 2 3 4 5)) ⇒ (1 2 3 4)
- assoc を定義せよ.

リスト処理の練習問題（自習用）

- but-last を定義せよ.
- (but-last '(1 2 3 4 5)) ⇒ (1 2 3 4)
- assoc を定義せよ.
- (assoc hgo '((f IP) (hgo IntroAlgDS) (yan ProLang))

リスト処理の練習問題（自習用）

- but-last を定義せよ.
- (but-last '(1 2 3 4 5)) ⇒ (1 2 3 4)
- assoc を定義せよ.
- (assoc hgo '((f IP) (hgo IntroAlgDS) (yan ProLang))
  ⇒ (hgo IntroAlgDS)

リスト処理の練習問題（自習用）

- but-last を定義せよ.
- (but-last '(1 2 3 4 5)) ⇒ (1 2 3 4)
- assoc を定義せよ.
- (assoc hgo '((f IP) (hgo IntroAlgDS) (yan ProLang))
  ⇒ (hgo IntroAlgDS)
- (assoc hgo '((ishi Gairon) (iso math) (tom HW)) ⇒ #f

リスト処理の練習問題（自習用）

- but-last を定義せよ.
- (but-last '(1 2 3 4 5)) ⇒ (1 2 3 4)
- assoc を定義せよ.
- (assoc hgo '((f IP) (hgo IntroAlgDS) (yan ProLang))
  ⇒ (hgo IntroAlgDS)
- (assoc hgo '((ishi Gairon) (iso math) (tom HW)) ⇒ #f
- length を定義せよ.

リスト処理の練習問題（自習用）

- but-last を定義せよ.
- (but-last '(1 2 3 4 5)) ⇒ (1 2 3 4)
- assoc を定義せよ.
- (assoc hgo '((f IP) (hgo IntroAlgDS) (yan ProLang))
  ⇒ (hgo IntroAlgDS)
- (assoc hgo '((ishi Gairon) (iso math) (tom HW)) ⇒ #f
- length を定義せよ.
- (length '(1 2)) ⇒ 2 (length '(1 2 3 5)) ⇒ 4

リスト処理の練習問題（自習用）

- but-last を定義せよ.
- (but-last '(1 2 3 4 5)) ⇒ (1 2 3 4)
- assoc を定義せよ.
- (assoc hgo '((f IP) (hgo IntroAlgDS) (yan ProLang))
  ⇒ (hgo IntroAlgDS)
- (assoc hgo '((ishi Gairon) (iso math) (tom HW)) ⇒ #f
- length を定義せよ.
- (length '(1 2)) ⇒ 2 (length '(1 2 3 5)) ⇒ 4
- (length nil) ⇒ 0

リスト処理の練習問題（自習用）

- but-last を定義せよ.
- (but-last '(1 2 3 4 5)) ⇒ (1 2 3 4)
- assoc を定義せよ.
- (assoc hgo '((f IP) (hgo IntroAlgDS) (yan ProLang))
  ⇒ (hgo IntroAlgDS)
- (assoc hgo '((ishi Gairon) (iso math) (tom HW)) ⇒ #f
- length を定義せよ.
- (length '(1 2)) ⇒ 2 (length '(1 2 3 5)) ⇒ 4
- (length nil) ⇒ 0
- copy を定義せよ.

リスト処理の練習問題（自習用）

- but-last を定義せよ.
- (but-last '(1 2 3 4 5)) ⇒ (1 2 3 4)
- assoc を定義せよ.
- (assoc hgo '((f IP) (hgo IntroAlgDS) (yan ProLang))
  ⇒ (hgo IntroAlgDS)
- (assoc hgo '((ishi Gairon) (iso math) (tom HW)) ⇒ #f
- length を定義せよ.
- (length '(1 2)) ⇒ 2 (length '(1 2 3 5)) ⇒ 4
- (length nil) ⇒ 0
- copy を定義せよ.
- (copy '(1 2)) ⇒ (1 2)

リスト処理の練習問題（自習用）

- but-last を定義せよ.
- (but-last '(1 2 3 4 5)) ⇒ (1 2 3 4)
- assoc を定義せよ.
- (assoc hgo '((f IP) (hgo IntroAlgDS) (yan ProLang))
  ⇒ (hgo IntroAlgDS)
- (assoc hgo '((ishi Gairon) (iso math) (tom HW)) ⇒ #f
- length を定義せよ.
- (length '(1 2)) ⇒ 2 (length '(1 2 3 5)) ⇒ 4
- (length nil) ⇒ 0
- copy を定義せよ.
- (copy '(1 2)) ⇒ (1 2)
- (copy '((1 . 2) . (3 . 4))) ⇒ ((1 . 2) 3 . 4)

リスト処理の練習問題（自習用）

- but-last を定義せよ.
- (but-last '(1 2 3 4 5)) ⇒ (1 2 3 4)
- assoc を定義せよ.
- (assoc hgo '((f IP) (hgo IntroAlgDS) (yan ProLang))
  ⇒ (hgo IntroAlgDS)
- (assoc hgo '((ishi Gairon) (iso math) (tom HW)) ⇒ #f
- length を定義せよ.
- (length '(1 2)) ⇒ 2 (length '(1 2 3 5)) ⇒ 4
- (length nil) ⇒ 0
- copy を定義せよ.
- (copy '(1 2)) ⇒ (1 2)
- (copy '((1 . 2) . (3 . 4))) ⇒ ((1 . 2) 3 . 4)
- flatten を定義せよ.

リスト処理の練習問題（自習用）

- but-last を定義せよ.
- (but-last '(1 2 3 4 5)) ⇒ (1 2 3 4)
- assoc を定義せよ.
- (assoc hgo '((f IP) (hgo IntroAlgDS) (yan ProLang))
  ⇒ (hgo IntroAlgDS)
- (assoc hgo '((ishi Gairon) (iso math) (tom HW)) ⇒ #f
- length を定義せよ.
- (length '(1 2)) ⇒ 2 (length '(1 2 3 5)) ⇒ 4
- (length nil) ⇒ 0
- copy を定義せよ.
- (copy '(1 2)) ⇒ (1 2)
- (copy '((1 . 2) . (3 . 4))) ⇒ ((1 . 2) 3 . 4)
- flatten を定義せよ.

リスト処理の練習問題（自習用）

- but-last を定義せよ.
- (but-last '(1 2 3 4 5)) ⇒ (1 2 3 4)
- assoc を定義せよ.
- (assoc hgo '((f IP) (hgo IntroAlgDS) (yan ProLang))
  ⇒ (hgo IntroAlgDS)
- (assoc hgo '((ishi Gairon) (iso math) (tom HW)) ⇒ #f
- length を定義せよ.
- (length '(1 2)) ⇒ 2 (length '(1 2 3 5)) ⇒ 4
- (length nil) ⇒ 0
- copy を定義せよ.
- (copy '(1 2)) ⇒ (1 2)
- (copy '((1 . 2) . (3 . 4))) ⇒ ((1 . 2) 3 . 4)
- flatten を定義せよ.
- (flatten '((1)(2 (3) 4) 5) ⇒ (1 2 3 4 5)

リスト処理の練習問題（自習用）

- but-last を定義せよ.
- (but-last '(1 2 3 4 5)) ⇒ (1 2 3 4)
- assoc を定義せよ.
- (assoc hgo '((f IP) (hgo IntroAlgDS) (yan ProLang))
  ⇒ (hgo IntroAlgDS)
- (assoc hgo '((ishi Gairon) (iso math) (tom HW)) ⇒ #f
- length を定義せよ.
- (length '(1 2)) ⇒ 2 (length '(1 2 3 5)) ⇒ 4
- (length nil) ⇒ 0
- copy を定義せよ.
- (copy '(1 2)) ⇒ (1 2)
- (copy '((1 . 2) . (3 . 4))) ⇒ ((1 . 2) 3 . 4)
- flatten を定義せよ.
- (flatten '((1)(2 (3) 4) 5) ⇒ (1 2 3 4 5)
- (flatten '((1 2 3))) ⇒ (1 2 3)

リスト処理の練習問題（自習用）

- but-last を定義せよ.
- (but-last '(1 2 3 4 5)) ⇒ (1 2 3 4)
- assoc を定義せよ.
- (assoc hgo '((f IP) (hgo IntroAlgDS) (yan ProLang))
  ⇒ (hgo IntroAlgDS)
- (assoc hgo '((ishi Gairon) (iso math) (tom HW)) ⇒ #f
- length を定義せよ.
- (length '(1 2)) ⇒ 2 (length '(1 2 3 5)) ⇒ 4
- (length nil) ⇒ 0
- copy を定義せよ.
- (copy '(1 2)) ⇒ (1 2)
- (copy '((1 . 2) . (3 . 4))) ⇒ ((1 . 2) 3 . 4)
- flatten を定義せよ.
- (flatten '((1)(2 (3) 4) 5) ⇒ (1 2 3 4 5)
- (flatten '((1 2 3))) ⇒ (1 2 3)
- (flatten ' (1 2 3)) ⇒ (1 2 3) (flatten nil) ⇒ nil

式の評価

- (+ (* 3 2) 5) や (define x 30) なども、それ自身は単なるリスト

  > (list '+ (list '* 3 2) 5)

  (+ (3 2) 5)*

  - システムがこのリスト（データ）を「評価」すると、“関数呼び出し式”として処理を行い、値を返す

  > (eval (list '+ (list '* 3 2) 5))

  11

- フォーム：システムが評価できるデータ
- フォームでないデータを評価しようとするとエラーになる

フォームの分類（リスト・フォーム）

- **リスト・フォーム**
  - **関数適用 ・・・ (〈関数〉 〈式1〉 … 〈式n〉)**
    1. **〈関数〉 と 〈式1〉～〈式n〉を評価**
    2. **関数を〈式1〉～〈式n〉の評価結果に適用**
  - **スペシャル・フォーム**
    - define, set!, quote, if **など特定の記号で始まるリスト**
    - **それぞれのスペシャルフォームごとに決まった評価方法**
  - **マクロ・フォーム**
    - **マクロを表す記号で始まるリスト**
    - **プログラマが新しいマクロを定義できる**
    - **詳細は省略（ただし，Lispを強力たらしめる最大の特徴）**

（関数適用とは違う）

フォームの分類（リスト以外）

- **記号**
  - **変数の値が評価値になる**

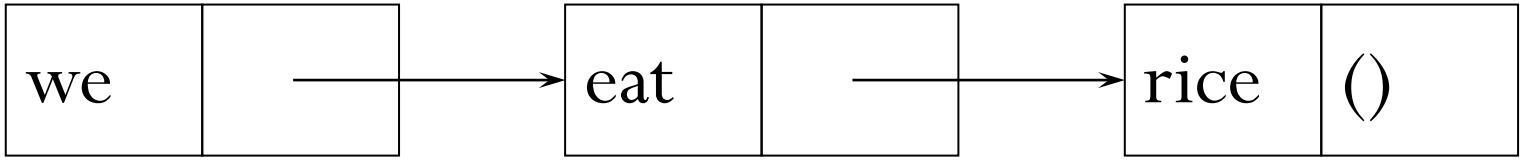
```
> (define x 30)
x
> x
30
> +
#<function +>
```

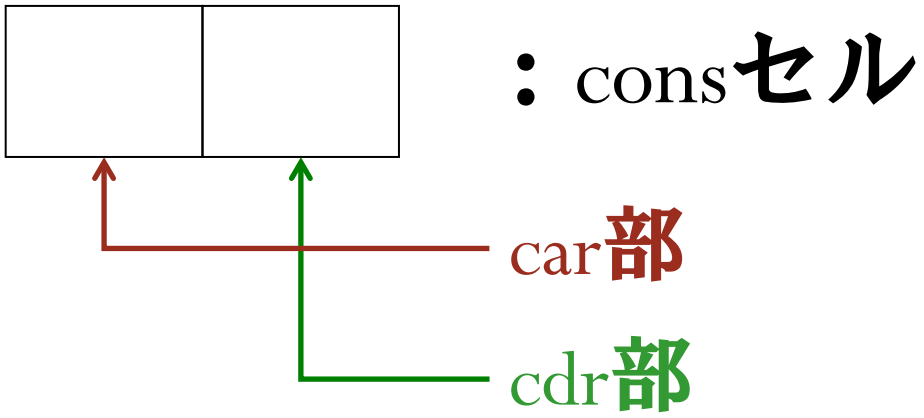
- **その他のデータ（数値，文字列など）**
  - **それ自身が評価値となる**

```
> 123
123
> "abcde"
"abcde"
```

consセル（１）

- **例1：** (we eat rice) **の内部表現**





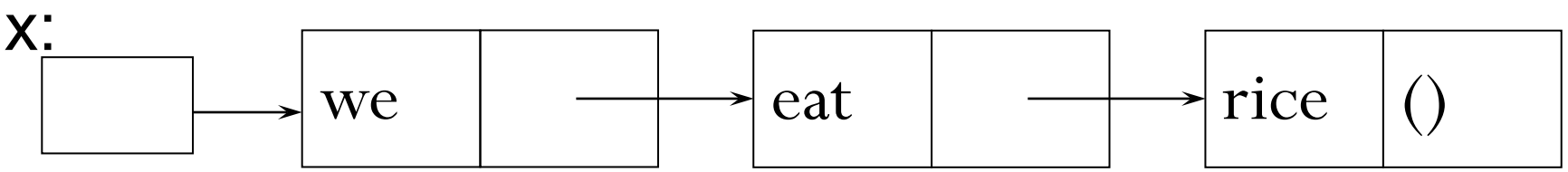
：consセル

consセル（２）

- **例２：**

(define x '(we eat rice))

(define y (cons 'they (cdr x))

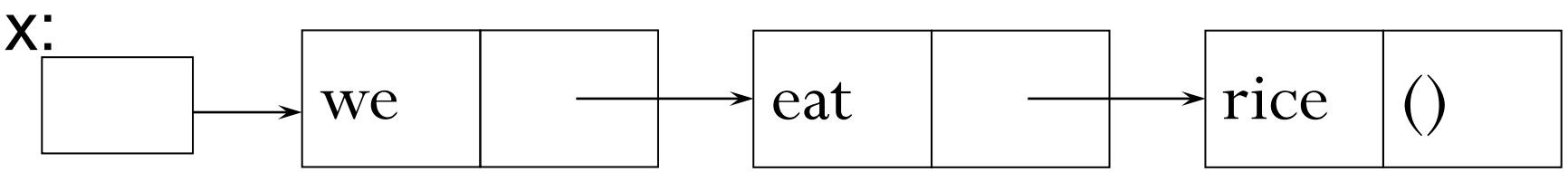


consセル（２）

- **例２：**

(define x '(we eat rice))

(define y (cons 'they (cdr x))

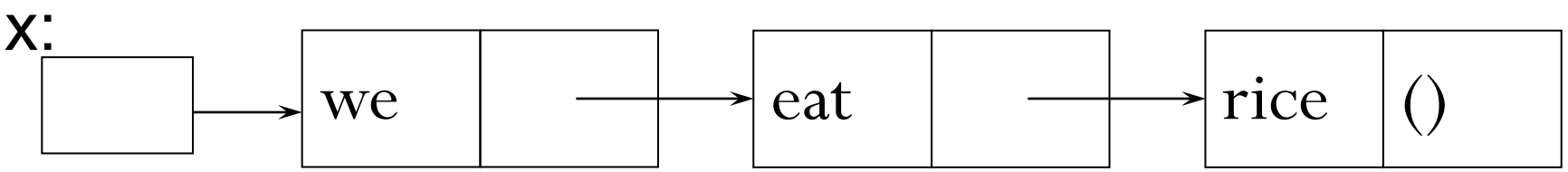


consセル（２）

- **例２：**

(define x '(we eat rice))

(define y (cons 'they (cdr x))

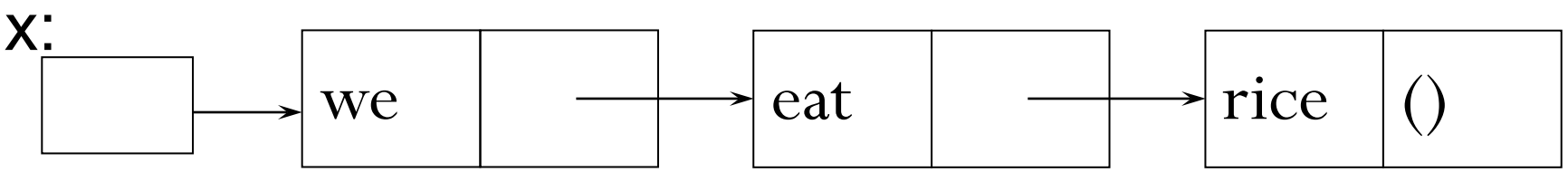


consセル（２）

- **例２：**

(define x '(we eat rice))

(define y (cons 'they (cdr x))

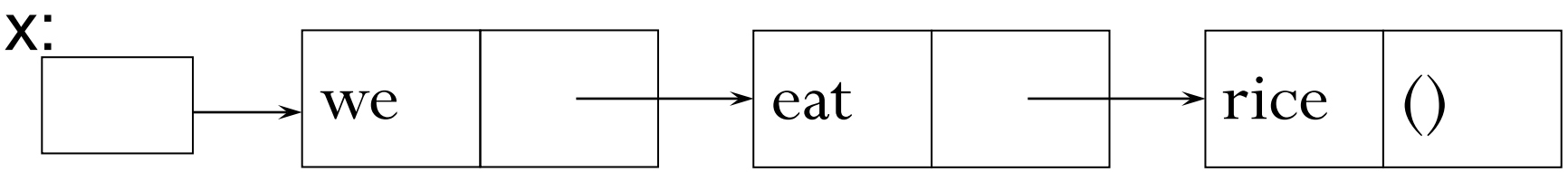


consセル（２）

- **例２：**

(define x '(we eat rice))

(define y (cons 'they (cdr x))

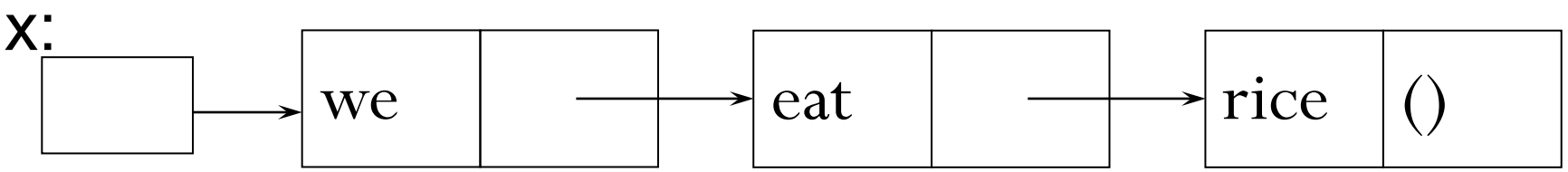


consセル（２）

- **例２：**

(define x '(we eat rice))

(define y (cons 'they (cdr x))

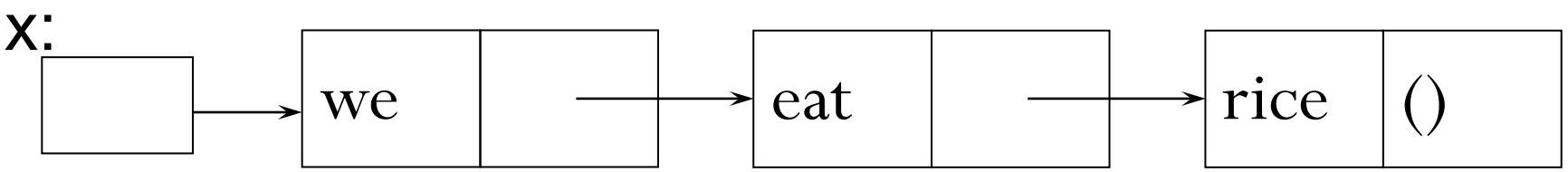


consセル（２）

- **例２：**

(define x '(we eat rice))

(define y (cons 'they (cdr x))

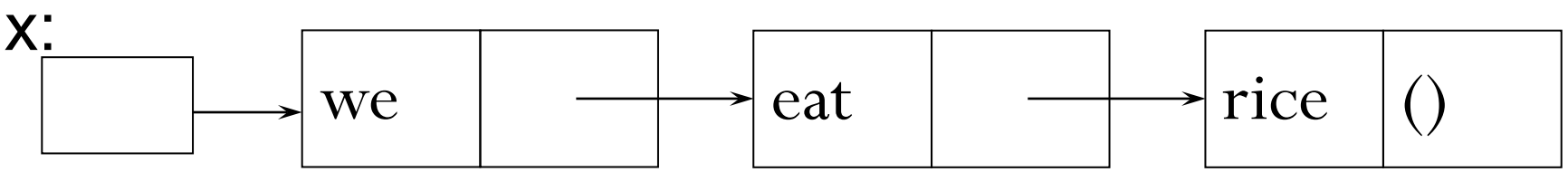


consセル（２）

- **例２：**

(define x '(we eat rice))

(define y (cons 'they (cdr x))

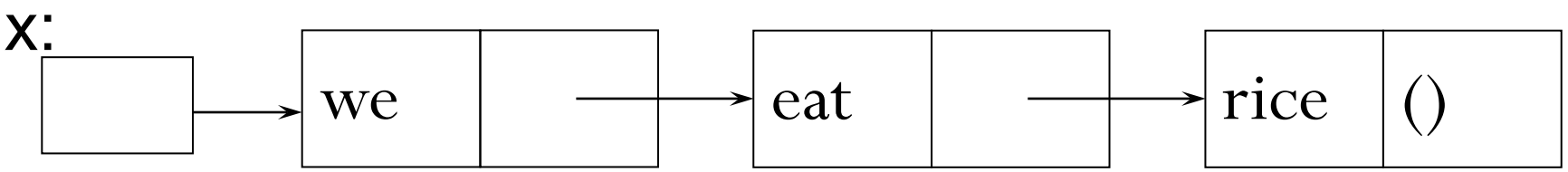


consセル（２）

- **例２：**

(define x '(we eat rice))

(define y (cons 'they (cdr x))

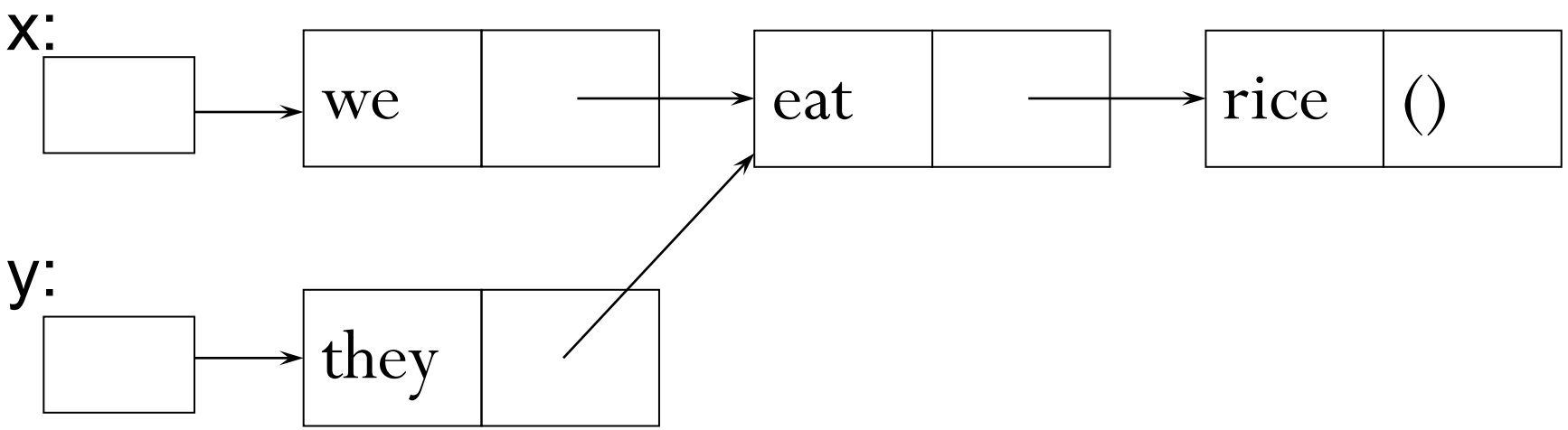


consセル（２）

- **例２：**

(define x '(we eat rice))

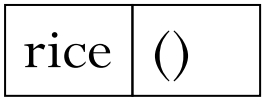
(define y (cons 'they (cdr x))



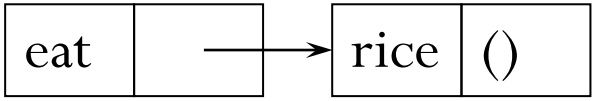
consセル（３）

- (cons 〈データ１〉 〈データ2〉)：
  car部とcdr部がそれぞれ〈データ１〉，〈データ2〉であるconsセルを作る

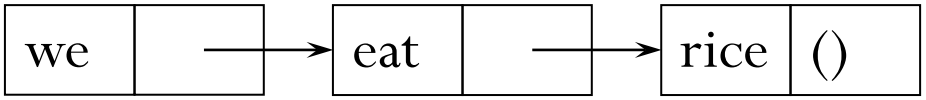
> (cons 'rice '())



> (cons 'eat (cons 'rice '()))



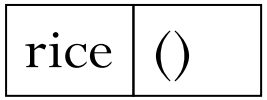
> (cons 'we (cons 'eat (cons 'rice '())))



consセル（３）

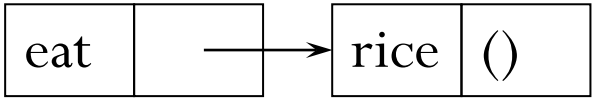
- (cons 〈データ１〉 〈データ2〉)：
  car部とcdr部がそれぞれ〈データ１〉，〈データ2〉であるconsセルを作る

> (cons 'rice '())

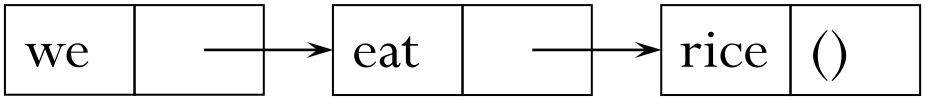


(rice)

> (cons 'eat (cons 'rice '()))



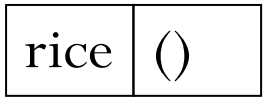
> (cons 'we (cons 'eat (cons 'rice '())))



consセル（３）

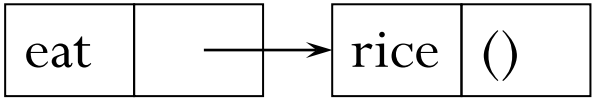
- (cons 〈データ１〉 〈データ2〉)：
  car部とcdr部がそれぞれ〈データ１〉，〈データ2〉
  であるconsセルを作る

> (cons 'rice '())



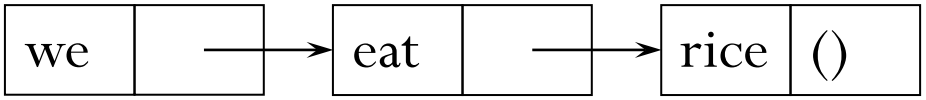
(rice)

> (cons 'eat (cons 'rice '()))



(eat rice)

> (cons 'we (cons 'eat (cons 'rice '())))

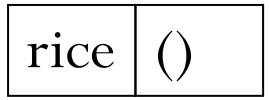


consセル（３）

- (cons 〈データ１〉 〈データ2〉)：
  car部とcdr部がそれぞれ〈データ１〉，〈データ2〉
  であるconsセルを作る

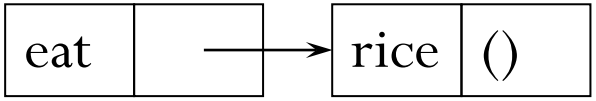
> (cons 'rice '())

(rice)



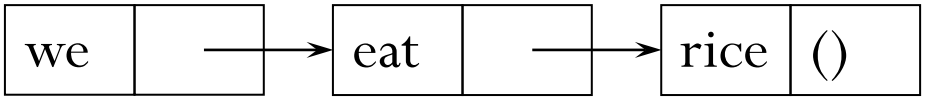
> (cons 'eat (cons 'rice '()))

(eat rice)



> (cons 'we (cons 'eat (cons 'rice '())))

(we eat rice)

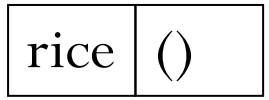


consセル（３）

- (cons 〈データ１〉 〈データ2〉)：
  car部とcdr部がそれぞれ〈データ１〉，〈データ2〉
  であるconsセルを作る

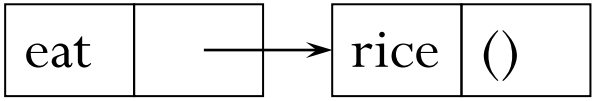
> (cons 'rice '())

(rice)



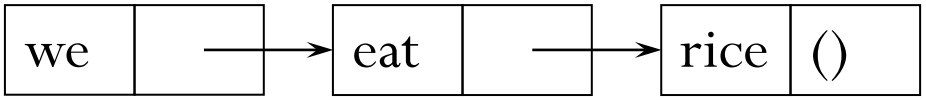
> (cons 'eat (cons 'rice '()))

(eat rice)



> (cons 'we (cons 'eat (cons 'rice '())))

(we eat rice)

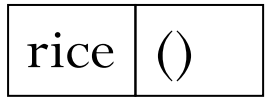


consセル（３）

- (cons 〈データ１〉 〈データ2〉)：
  car部とcdr部がそれぞれ〈データ１〉，〈データ2〉
  であるconsセルを作る

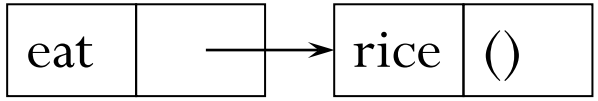
> (cons 'rice '())

(rice)



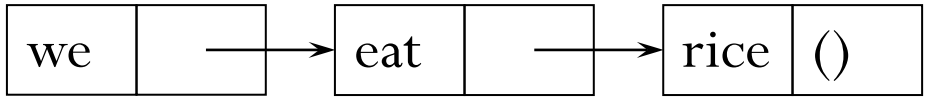
> (cons 'eat (cons 'rice '()))

(eat rice)



> (cons 'we (cons 'eat (cons 'rice '())))

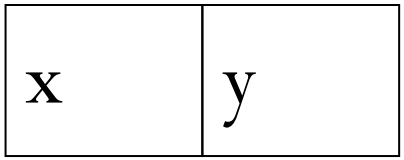
(we eat rice) = (list 'we 'eat 'rice)



consセル（４）

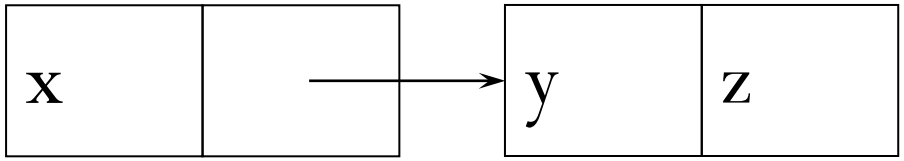
- **ドット・ペア**

> (cons 'x 'y)



- **ドット・リスト**

> (cons 'x (cons 'y 'z))

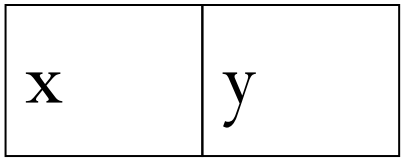


consセル（４）

- **ドット・ペア**

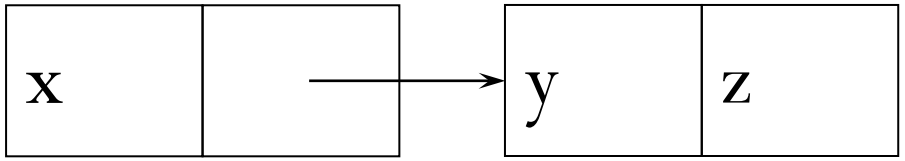
> (cons 'x 'y)

(x . y)



- **ドット・リスト**

> (cons 'x (cons 'y 'z))

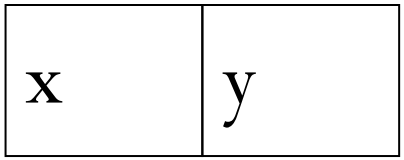


consセル（４）

- **ドット・ペア**

> (cons 'x 'y)

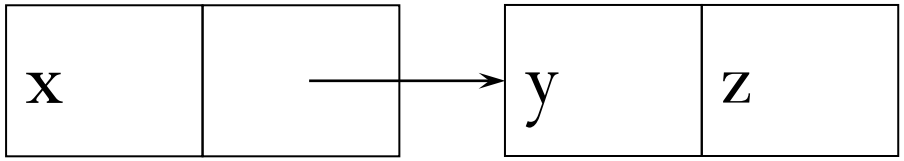
(x . y)



- **ドット・リスト**

> (cons 'x (cons 'y 'z))

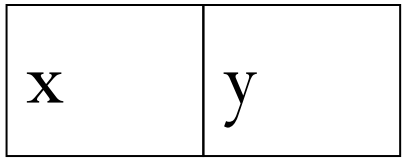
(x y . z)

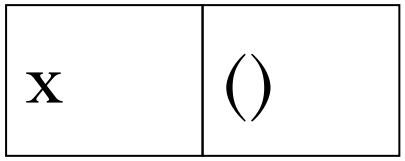


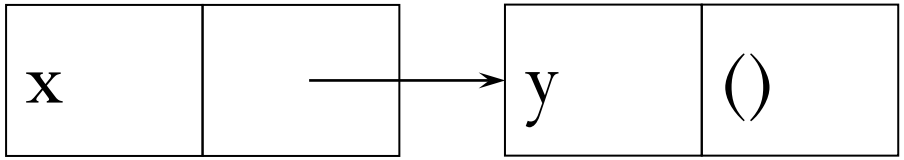
consセル（5）

- **ドット・ペア**

```
> (cons 'x  'y)
(x . y)
> (cons 'x  ())
(x)                    の略記
> (cons 'x (cons 'y ()))
(x  y)                 の略記
```



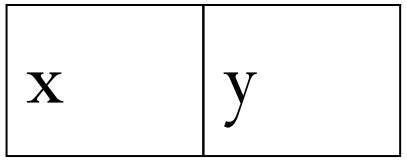


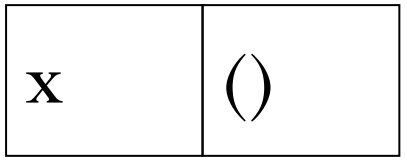


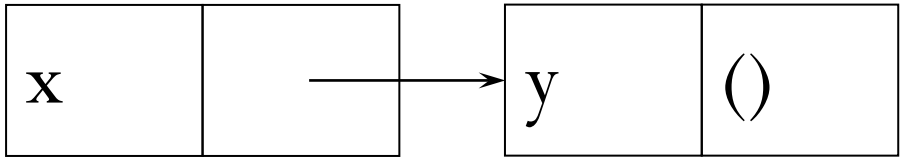
consセル（5）

- **ドット・ペア**

```
> (cons 'x  'y)
(x . y)
> (cons 'x  ())
(x)                    (x . ())の略記
> (cons 'x (cons 'y ()))
(x  y)                 の略記
```



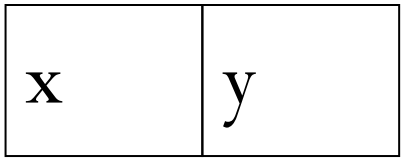


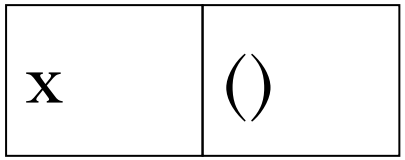


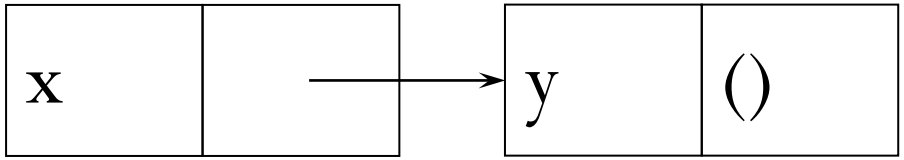
consセル（5）

- **ドット・ペア**

```
> (cons 'x  'y)
(x . y)
> (cons 'x  ())
(x)                    (x . ())の略記
> (cons 'x (cons 'y ()))
(x  y)                 (x . (y . ()))の略記
```







consセル（6）

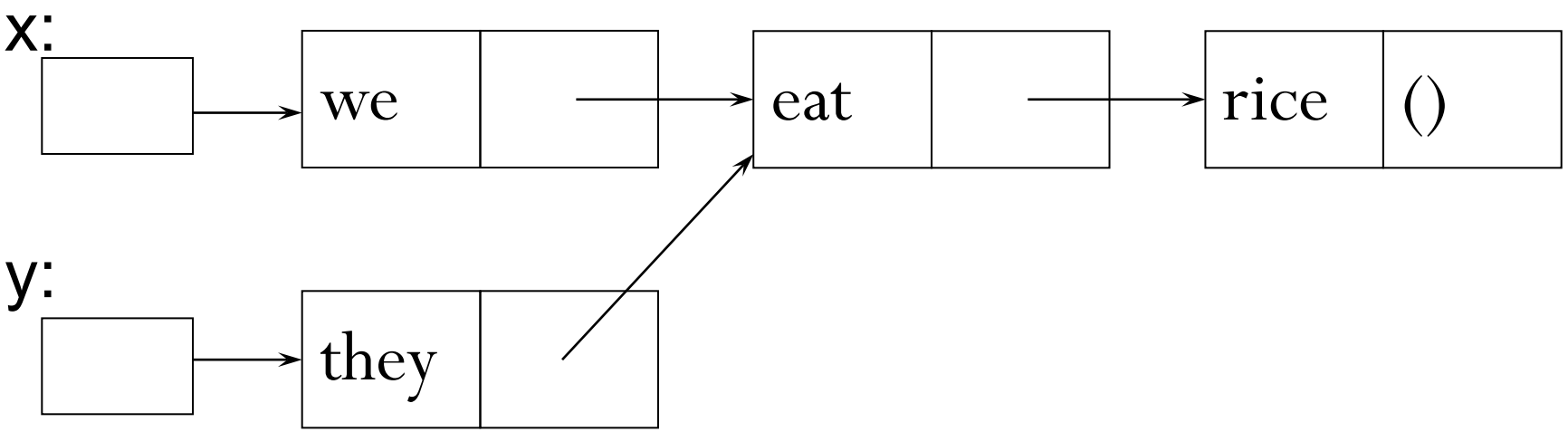
- **破壊的操作**

> (set-cdr! x 123)

→ x: [redacted]

> (set-car! x y)

→ x: [redacted]



consセル（6）

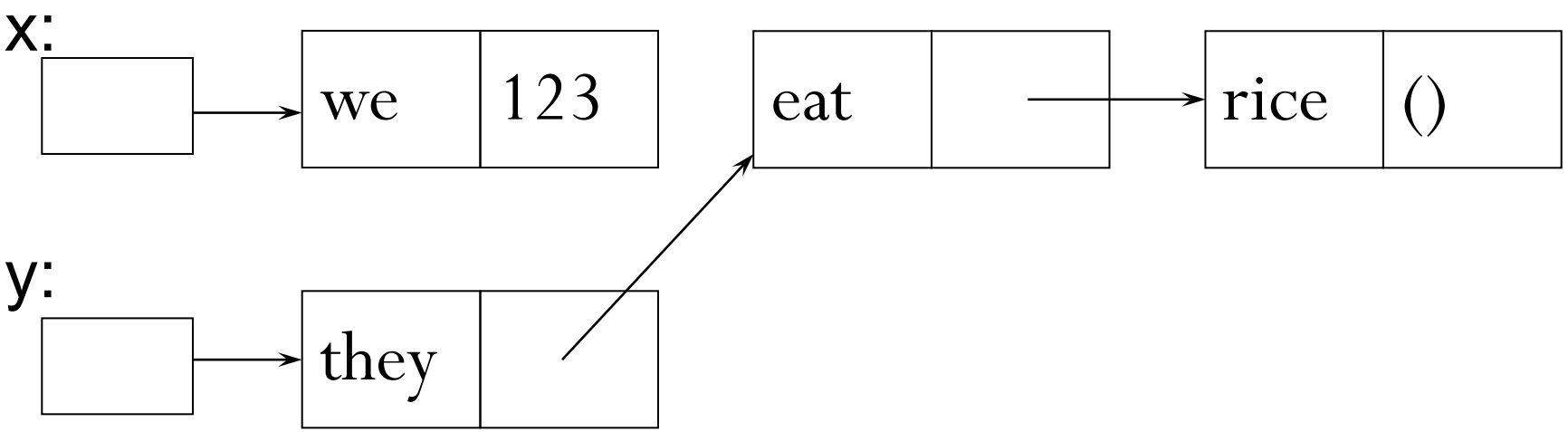
- **破壊的操作**

> (set-cdr! x 123)

→ x: [redacted]

> (set-car! x y)

→ x: [redacted]



consセル（6）

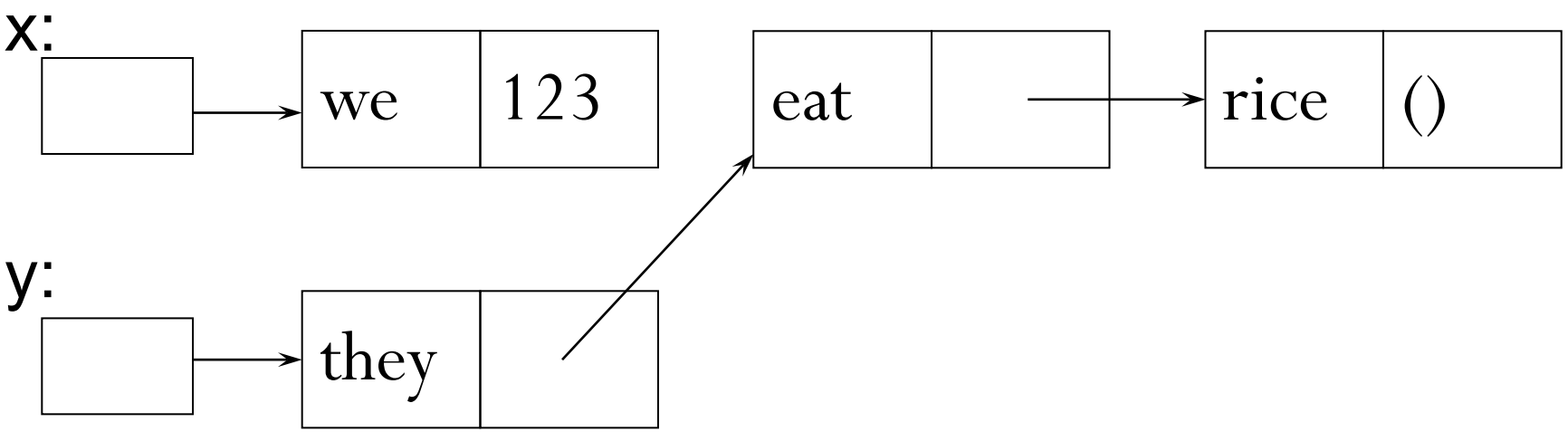
- **破壊的操作**

> (set-cdr! x 123)

→ x: (we . 123)

> (set-car! x y)

→ x:



consセル（6）

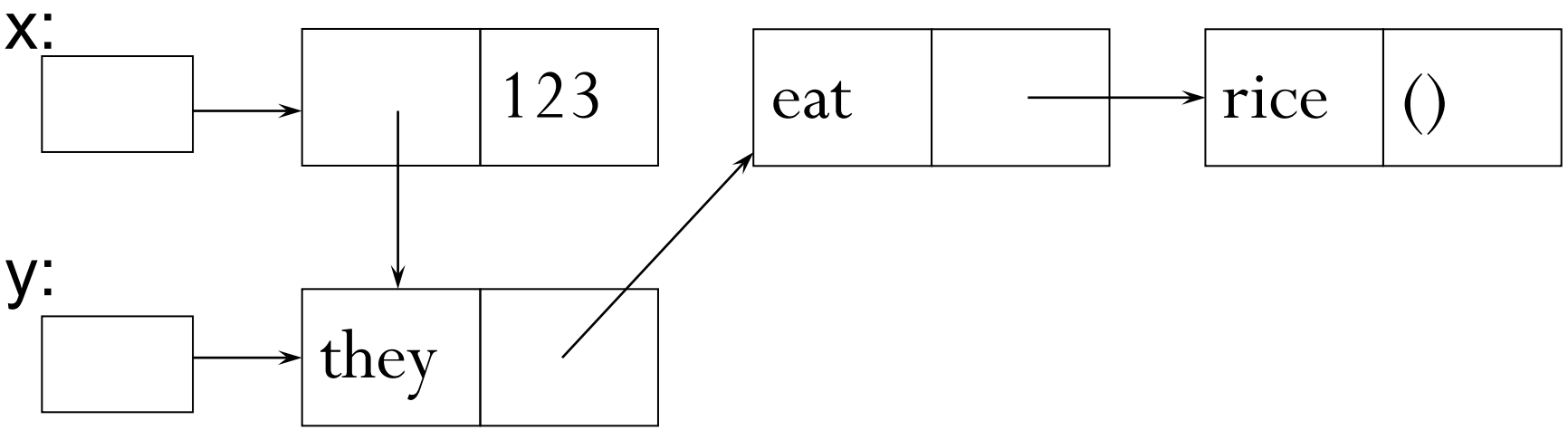
- **破壊的操作**

> (set-cdr! x 123)

→ x: (we . 123)

> (set-car! x y)

→ x:



consセル（6）

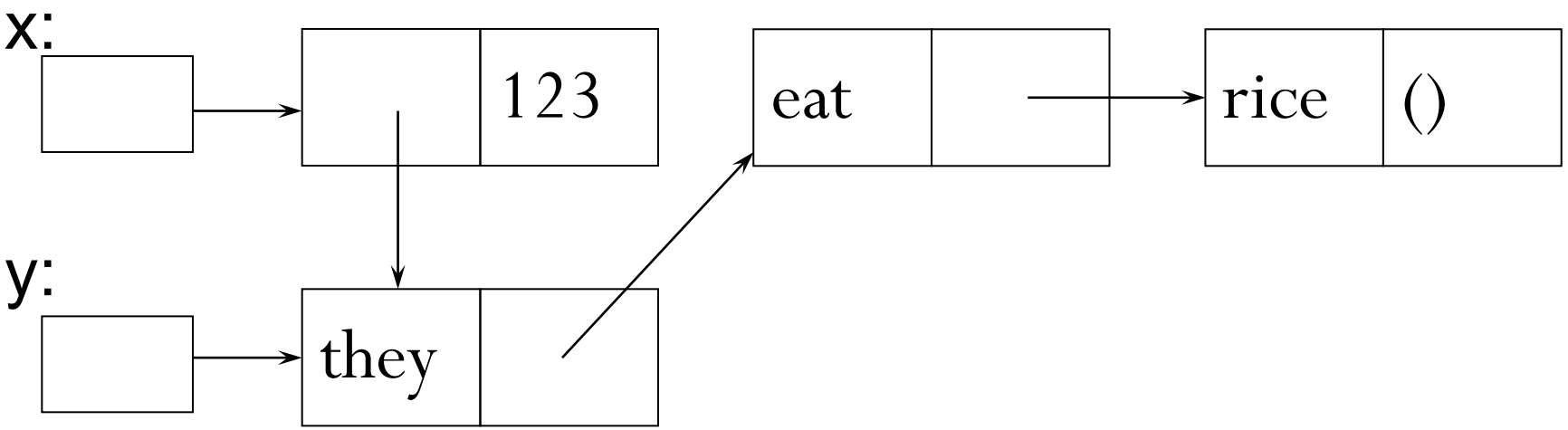
- **破壊的操作**

> (set-cdr! x 123)

→ x: (we . 123)

> (set-car! x y)

→ x: ((they eat rice) . 123)



データの比較

- **数値どうしの比較は** =, <, >, <=, >= **を使うべき**
  - (< 3 10) → #t
  - (= 4 5) → #f
- **リストどうしの比較**
  - (eq? <*e1*> <*e2*>)
    - <*e1*>**と** <*e2*>**が同じオブジェクトなら**#t
  - (eqv? <*e1*> <*e2*>)
    - <*e1*> **と**<*e2*>**が**eq?**の関係か，同じ数値，文字，**etc. **なら**#t
  - (equal? <*e1*> <*e2*>)
    - <*e1*> **と**<*e2*>**が同じ内容（のリスト）なら**#t

eq?とequal?の違い

```
> (define x1 (list 10 20))
(10 20)
> (define x2 x1)
> (define y (list 10 20))
(10 20)
> (eq? x1 x2)
```

```
> (eq? x1 y)
```

```
> (equal? x1 y)
```

eq?とequal?の違い

```
> (define x1 (list 10 20))
(10 20)
> (define x2 x1)
> (define y (list 10 20))
(10 20)
> (eq? x1 x2)
```

```
> (eq? x1 y)
```

```
> (equal? x1 y)
```

eq?とequal?の違い

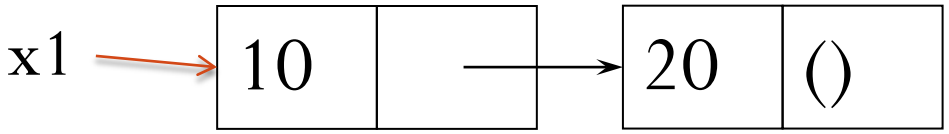
```
> (define x1 (list 10 20))
(10 20)
> (define x2 x1)
> (define y (list 10 20))
(10 20)
> (eq? x1 x2)
```

```
> (eq? x1 y)
```

```
> (equal? x1 y)
```

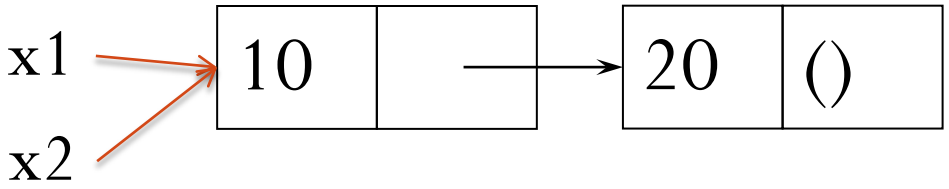
eq?とequal?の違い

```
> (define x1 (list 10 20))
(10 20)
> (define x2 x1)
> (define y (list 10 20))
(10 20)
> (eq? x1 x2)
#t
> (eq? x1 y)
#f
> (equal? x1 y)
#t
```



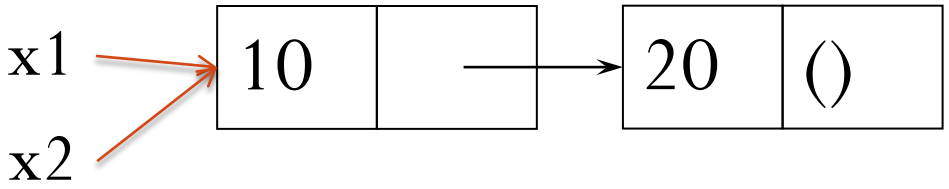
eq?とequal?の違い

```
> (define x1 (list 10 20))
(10 20)
> (define x2 x1)
> (define y (list 10 20))
(10 20)
> (eq? x1 x2)
#t
> (eq? x1 y)
#f
> (equal? x1 y)
#t
```



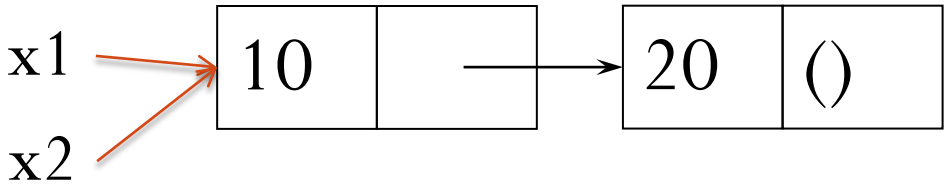
eq?とequal?の違い

```
> (define x1 (list 10 20))
(10 20)
> (define x2 x1)
> (define y (list 10 20))
(10 20)
> (eq? x1 x2)
#t
> (eq? x1 y)
#f
> (equal? x1 y)
#t
```



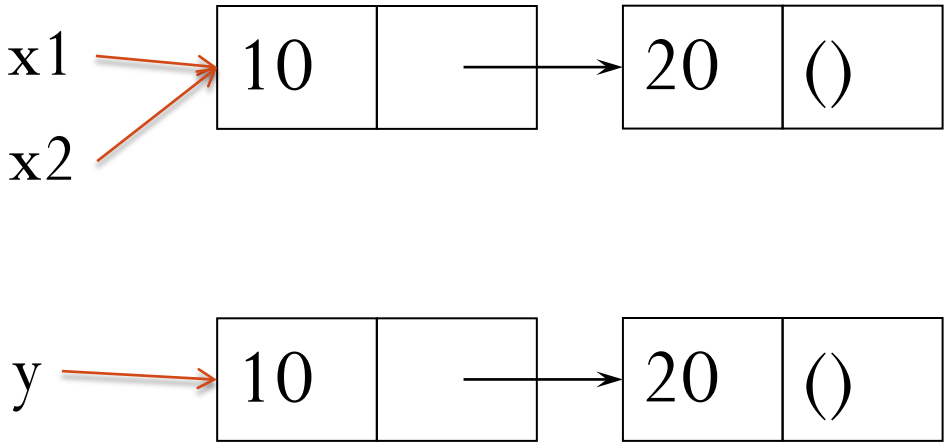
eq?とequal?の違い

```
> (define x1 (list 10 20))
(10 20)
> (define x2 x1)
> (define y (list 10 20))
(10 20)
> (eq? x1 x2)
#t
> (eq? x1 y)
#f
> (equal? x1 y)
#t
```



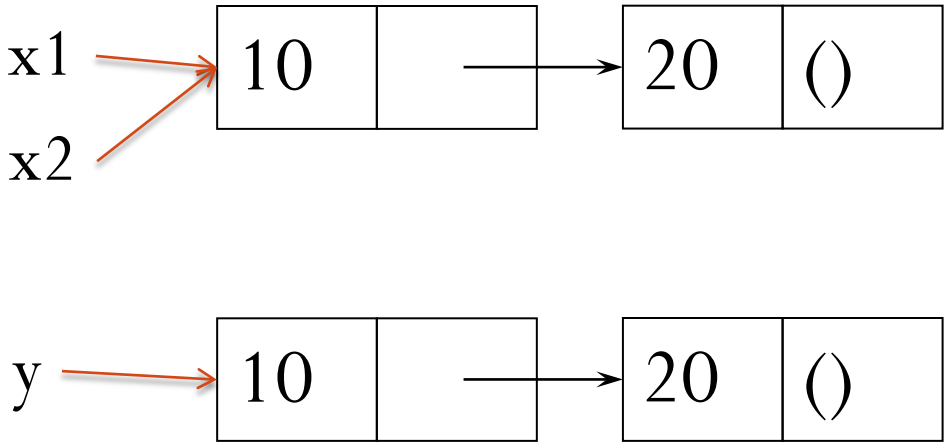
eq?とequal?の違い

```
> (define x1 (list 10 20))
(10 20)
> (define x2 x1)
> (define y (list 10 20))
(10 20)
> (eq? x1 x2)
#t
> (eq? x1 y)
#f
> (equal? x1 y)
#t
```



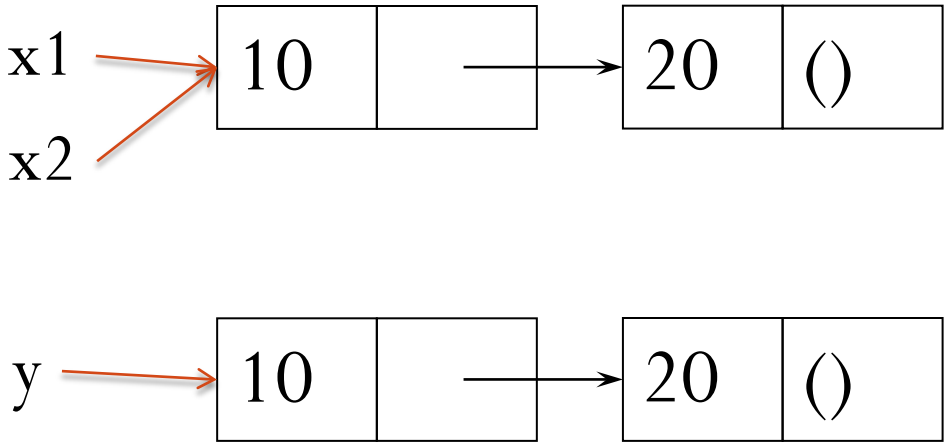
eq?とequal?の違い

```
> (define x1 (list 10 20))
(10 20)
> (define x2 x1)
> (define y (list 10 20))
(10 20)
> (eq? x1 x2)
#t
> (eq? x1 y)
#f
> (equal? x1 y)
#t
```



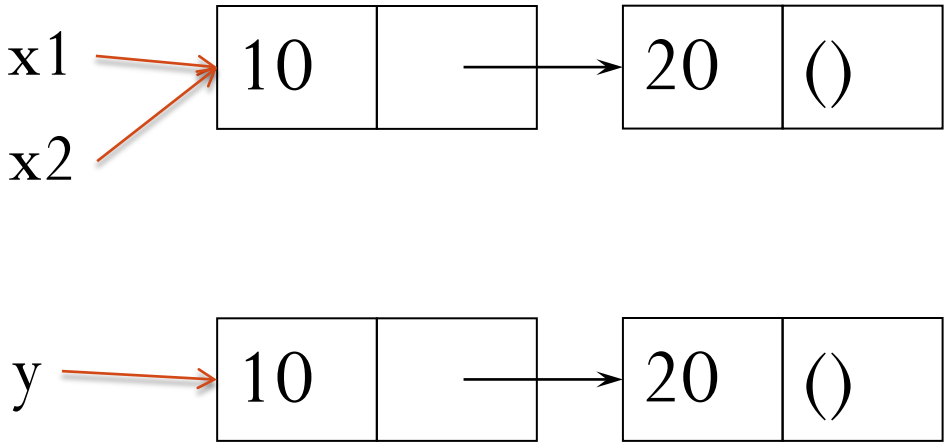
eq?とequal?の違い

```
> (define x1 (list 10 20))
(10 20)
> (define x2 x1)
> (define y (list 10 20))
(10 20)
> (eq? x1 x2)
#t
> (eq? x1 y)
#f
> (equal? x1 y)
```



eq?とequal?の違い

```
> (define x1 (list 10 20))
(10 20)
> (define x2 x1)
> (define y (list 10 20))
(10 20)
> (eq? x1 x2)
#t
> (eq? x1 y)
#f
> (equal? x1 y)
#t
```



equal?の定義

- (define (my-equal? e1 e2)
  (if (pair? e1)
  (and (pair? e2)
  (my-equal? (car e1) (car e2))
  (my-equal? (cdr e1) (cdr e2)))
  (eqv? e1 e2)))
- **本物のequal?は，文字列や配列にも対応**

画面への出力（1）

- **出力関数**

```
> (write '(a b c))
(a b c)(a b c)
```

システムの出力（評価値）

write関数の出力

```
> (begin (write '(a b c)) (newline))
(a b c)
#t
```

write関数の出力

システムの出力（評価値）

画面への出力（2）

```
> (define (square x)
      (write (list 'x x))
      (newline)
      (* x x))
square
> (square (* 3 4))
(x 12)
144
```

キーボードからの入力

- **入力関数**

```
> (read)
abcde
abcde
> (define (prompt-fact)
    (display "input: ")
    (fact (read)))
prompt-fact
> (prompt-fact)
input: 11
39916800
```

キーボードから入力

システムの出力（read関数の返り値）

キーボードから入力

プロンプト（display関数が出力）

ファイルへ出力

```
> (define out (open-output-file "outfile"))
out
> out
#<port to outfile>
> (write (fact 7) out) ------ 結果を "outfile"に書き込む
5040
> (newline out)
#t
> (close-output-port out)
#t
```

ファイルから入力

```
> (define in (open-input-file "infile"))
in
> (read in)
data
> (read in)
#<end-of-file>
> (close-input-port in)
#t
```

"infile"からデータを1つ読み込む

ファイルの終端かチェック

ファイル入出力のための関数

- (call-with-output-file **〈ファイル名〉 〈関数〉**)**:**
  指定されたファイルへの出力ポートを 引数として〈関数〉を呼び出し，その返り値を返す.
- (call-with-input-file **〈ファイル名〉 〈関数〉**)**:**
  指定されたファイルへの入力ポートを 引数として〈関数〉を呼び出し，その返り値を返す.

- **〈関数〉は1引数の関数でなければならない。**
- **実行終了後、ファイルは自動的に閉じられる。**

call-with-output-file

- **例：** $1^2, 2^2, \ldots, 99^2$ **の値をファイルに書き出す。**

```
(call-with-output-file "square99.out"
  (lambda (out)
    (let loop ((n 1))
      (if (< n 100)
          (begin
            (write (* n n) out)
            (newline out)
            (loop (+ n 1)))))))
```

call-with-input-file

- **例：ファイルから全てのデータを順に読み込んで画面に表示する。**

```
(call-with-input-file ”infile”
  (lambda (in)
    (let loop ((dat (read in)))
      (if (not (eof-object? dat))
          (begin
            (write dat)
            (newline)
            (loop (read in)))))))
```

プログラムファイルのロード

- (load **〈ファイル名〉**) **：ファイルに書かれている フォームを順に、全て評価する。**

```
> (load "square.scm")
Loading square.scm...
Finished.
"square.scm"
> (square 4)
16
```

関数実行のトレース

- (trace **〈関数名〉**)：**関数のトレースを開始**
- (untrace **〈関数名〉**)：**トレースをやめる**

標準ではないが，たいていの処理系で使える．

```
> (trace fact)
> (fact 2)
1>(fact 2)
  2>(fact 1)
    |3>(fact 0)
    |3<(fact 1)
  2<(fact 1)
1<(fact 2)
2
```

主な組み込み関数（数値）

- **加減乗除**
  - (+ **〈数値1〉** … **〈数値n〉**)
  - (− **〈数値1〉** … **〈数値n〉**)
  - (* **〈数値1〉** … **〈数値n〉**)
  - (/ **〈数値1〉** … **〈数値n〉**)
  - (remainder **〈整数1〉** **〈整数2〉**)：**割り算の余り** (cf. modulo)
- **比較**
  - (= **〈数値1〉** … **〈数値n〉**)
  - (< **〈数値1〉** … **〈数値n〉**)
  - (> **〈数値1〉** … **〈数値n〉**)
  - (<= **〈数値1〉** … **〈数値n〉**)
  - (>= **〈数値1〉** … **〈数値n〉**)

主な組み込み関数（リスト）

- (length **〈リスト〉**)
- (append **〈リスト1〉** … **〈リストn〉**)
- (reverse **〈リスト〉**)

主な組み込み関数（等号・論理演算）

- **等号**
  - (eq? **〈データ1〉** **〈データ2〉**)
  - (eqv? **〈データ1〉** **〈データ2〉**)
  - (equal? **〈データ1〉** **〈データ2〉**)
- **論理演算**
  - (not **〈データ1〉**)
  - (and **〈データ1〉** … **〈データn〉**)**【特殊フォーム】**
  - (or **〈データ1〉** … **〈データn〉**) **【特殊フォーム】**

主な組み込み関数（データ型述語）

- (number? **〈データ〉**)
- (integer? **〈データ〉**)
- (symbol? **〈データ〉**)
- (pair? **〈データ〉**)
- (list? **〈データ〉**)
- (null? **〈データ〉**)
- (string? **〈データ〉**)

その他の組み込み関数

- **ヘルプ機能**（tus, guile**など**）
  - (apropos 〈**文字列**〉) **：〈文字列〉を含む組み込み関数、スペシャルフォーム、マクロの一覧を表示する.**

```
> (apropos "list")
dolist
get-host-list
list
list*
list-ref
list-tail
list->string
...
```

レポートについて

- ソースコードの手書きはなるべくやめて
- 適当なところで改行．インデント．
- コメント（‘;’を使う）もできるだけつけること．

```
(define (square x)  (write (list 'x x))
                    (newline)  (* x x))
```

レポートについて

- ソースコードの手書きはなるべくやめて
- 適当なところで改行．インデント．
- コメント（‘;’を使う）もできるだけつけること．

```
(define (square x)
        (write (list 'x x))    ; 追加
        (newline)              ; 追加
        (* x x))
```

おすすめエディタ

- **最低限，括弧の対応付けをハイライトしてくれるものでないと話にならない**
  - **メモ帳とか（なぜか）Wordで頑張っている人も…**
- **Lispのインデントに対応しているものがよい**
  - Windows
    - Meadow（**「設定済みMeadow」がお手軽**）
      http://www.bookshelf.jp/soft/meadow_8.html
    - xyzzy（**作者が日本人．Meadowは癖がありすぎてちょっと…という人にはよいかも**）
      http://www.jsdlab.co.jp/~kamei/
  - Linux, Mac
    - Emacs

あたりが無難

参考資料

- **講義のページ**
  http://winnie.kuis.kyoto-u.ac.jp/~okuno/Lecture/08/IntroAlgDs/
- TUT Scheme**のマニュアル**
  http://www.spa.is.uec.ac.jp/~komiya/tus-man/tus/
- TUT Scheme Tips
  http://www.spa.is.uec.ac.jp/~komiya/download/tus-tips.html