

sp-16. cons と種々のデータ構造

(Scheme プログラミング)

URL: <https://www.kkaneko.jp/pro/scheme/index.html>

金子邦彦



アウトライン

16-1 ペア

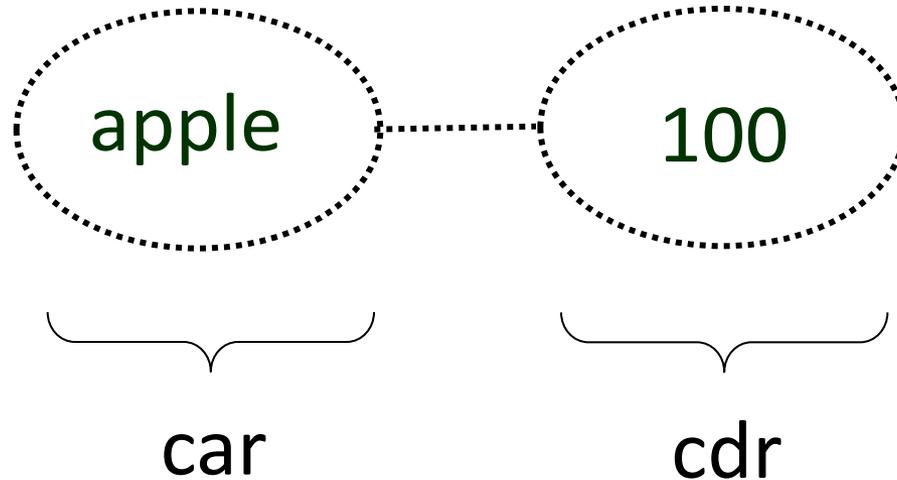
16-2 パソコン演習

16-3 課題



16-1 ペア

単純なペアの例



ペアとは： 2つの構成部分のペアのこと。
car と cdr に分かれる

- cons は 2 つの引数を取り, 2 つの引数を部分として含むような「ペア」を返す

例) `(define x (cons 'apple 100))`

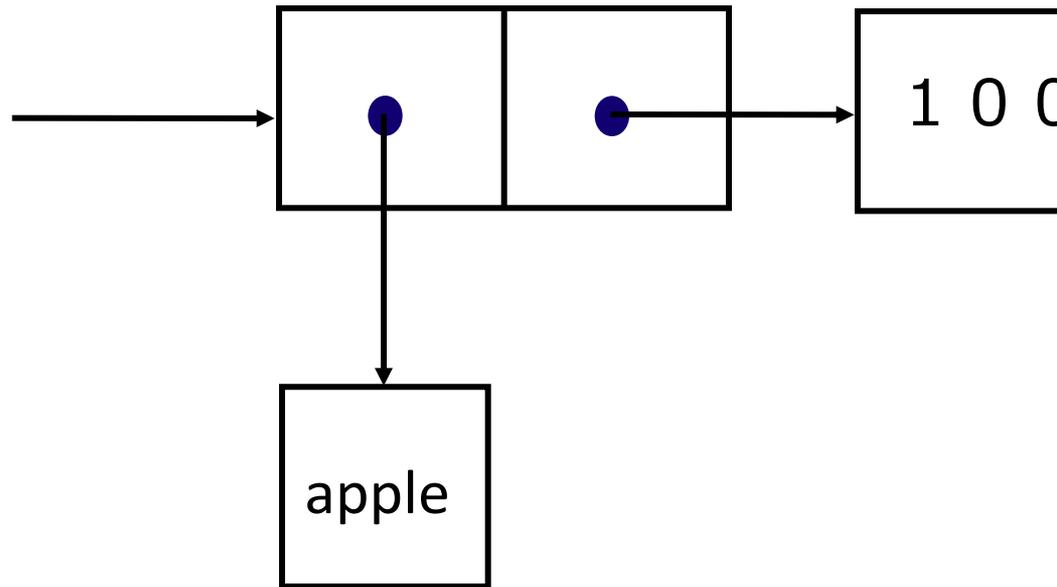
- ペアを構成する部分は, car, cdr で取り出せる

例) `(car x)` → 「apple」が得られる

`(cdr x)` → 「100」が得られる

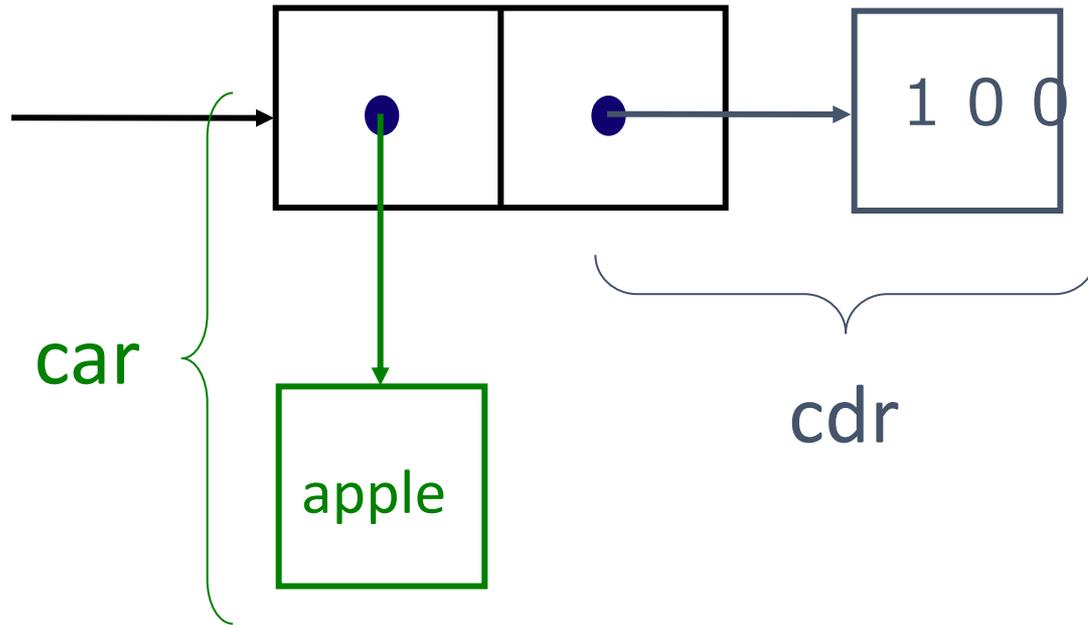
- ペアは, 「対」ともいう

(cons 'apple 100) の箱とポインタ記法

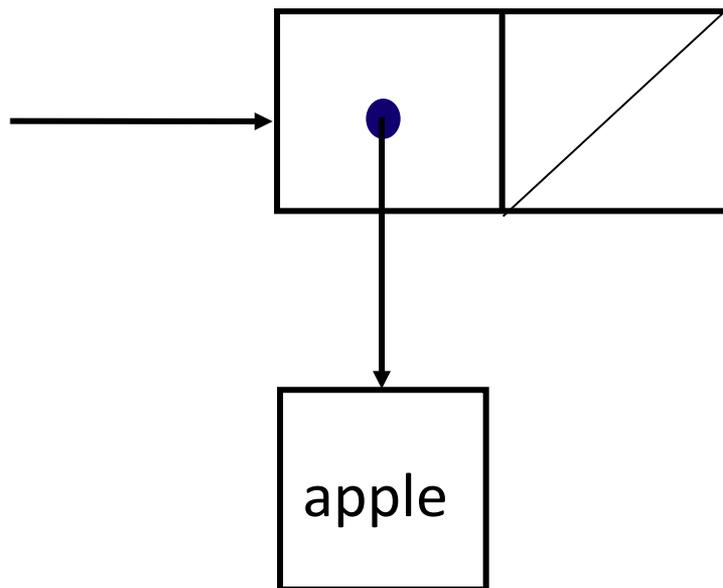


ペアとは： 2つの構成部分のペアのこと。
car と cdr に分かれる

(cons 'apple 100) の箱とポインタ記法

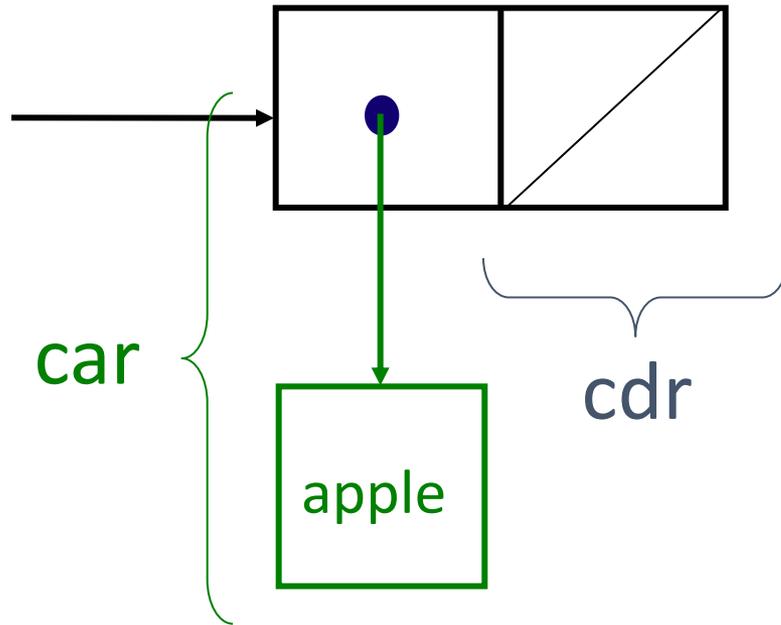


(cons 'apple empty) の箱とポインタ記法



ペアは, 上のように, 箱とポインタ表記される

(cons 'apple empty) の箱とポインタ記法



まとめ



- 「リスト」は末尾が「空リスト」であるような「ペアの並び」である
- ペアの組み合わせによって、複雑な構造を表現できる

16-2 パソコン演習

- 資料を見ながら、「例題」を行ってみる
- 各自、「課題」に挑戦する
- 自分のペースで先に進んで構いません

- DrScheme の起動
プログラム → PLT Scheme → DrScheme
- 今日の演習では「Full Scheme」
に設定
Language
→ Choose Language
→ Full Scheme
→ OK
→ 「Execute ボタン」

- ペアが、自由に扱えるようになる
- 但し、「empty」が使えなくなるので、代わりに「()'」を使う
- Full Scheme で empty を使おうとするとエラー
- リストの表示が変わる

(list 15 8 6) ⇒ (15 8 6) のように

例題 1. ペア



- シンボルと数値のペアを作る
 - ペアを生成するために `cons` を使う
 - ペアを構成する部分 (`'apple` や `100` など)を取り出すために `car`, `cdr` を使う

変数x : シンボル `'apple` と数値 `100` のペア

変数y: シンボル `'orange` と数値 `60` のペア

変数z: シンボル `'banana` と数値 `80` のペア

「例題 1. ペア」の手順



1. 次を「定義用ウィンドウ」で，実行しなさい
 - 入力した後に，Execute ボタンを押す

```
(define x (cons 'apple 100))  
(define y (cons 'orange 60))  
(define z (cons 'banana 80))
```

2. その後，次を「実行用ウィンドウ」で実行しなさい

```
x  
(car x)  
(cdr x)
```



Untitled Save Check Syntax Step Execute Break

```
(define x (cons 'apple 100))  
(define y (cons 'orange 60))  
(define z (cons 'banana 80))
```

ペアの生成

```
> x  
(apple . 100)  
> (car x)  
apple  
> (cdr x)  
100
```

表示されたペア

10:3 Unlocked not running

例題 2. リストの変数定義



- リスト 15, 8, 6 を変数として定義し, 名前Aを付ける
 - 変数を定義するために `define` を使う
 - リストを作るために `cons` を使う

「例題 2. リストの変数定義」の手順



1. 次を「定義用ウィンドウ」で, 実行しなさい
 - 入力した後に, Execute ボタンを押す

```
(define A (cons 15 (cons 8 (cons 6 '()))))
```

2. その後, 次を「実行用ウィンドウ」で実行しなさい

```
A  
(car A)  
(cdr A)
```

☆ 次は, 例題 3 に進んでください

Untitled Save Check Syntax Step Execute Break

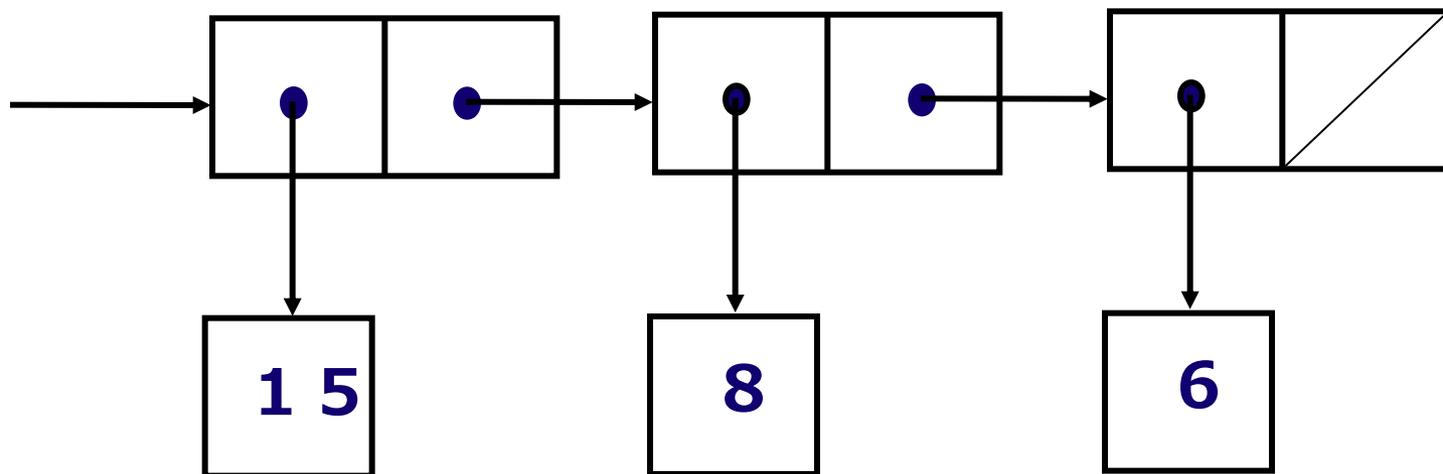
```
(define A (cons 15 (cons 8 (cons 6 '()))))
```

「'()'」は、空リストの意味

```
> A  
(15 8 6)  
> (car A)  
15  
> (cdr A)  
(8 6)  
> (cadr A)  
8
```

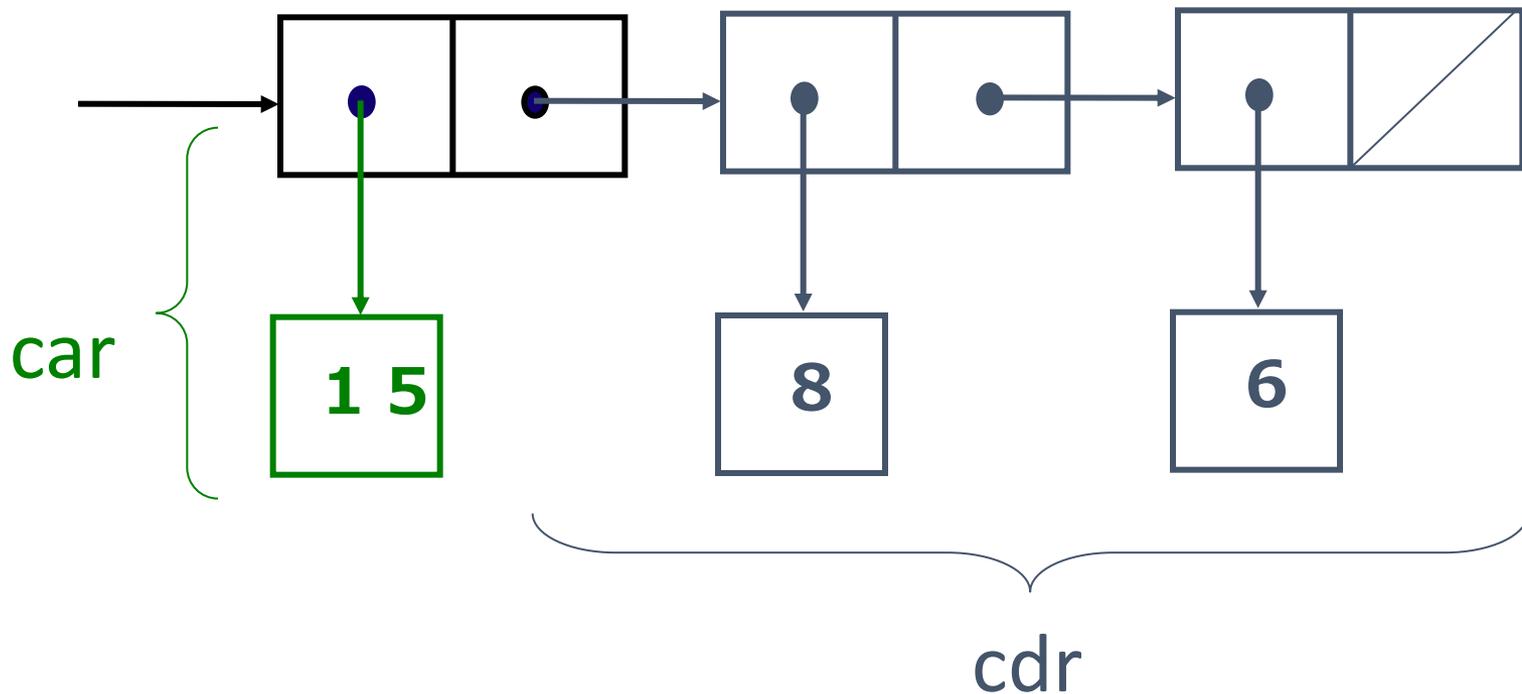
12:3 Unlocked not running

(cons 15 (cons 8 (cons 6 '()))) の箱とポインタ記法

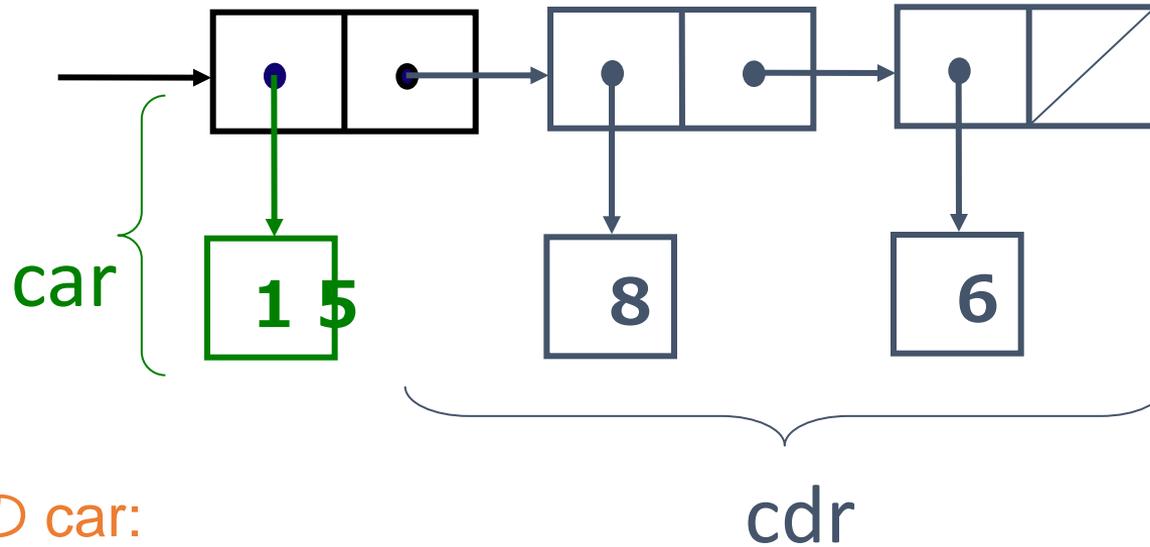


- リストの要素が、「ペアの並び」として順順につながる

(cons 15 (cons 8 (cons 6 '()))) の箱とポインタ記法

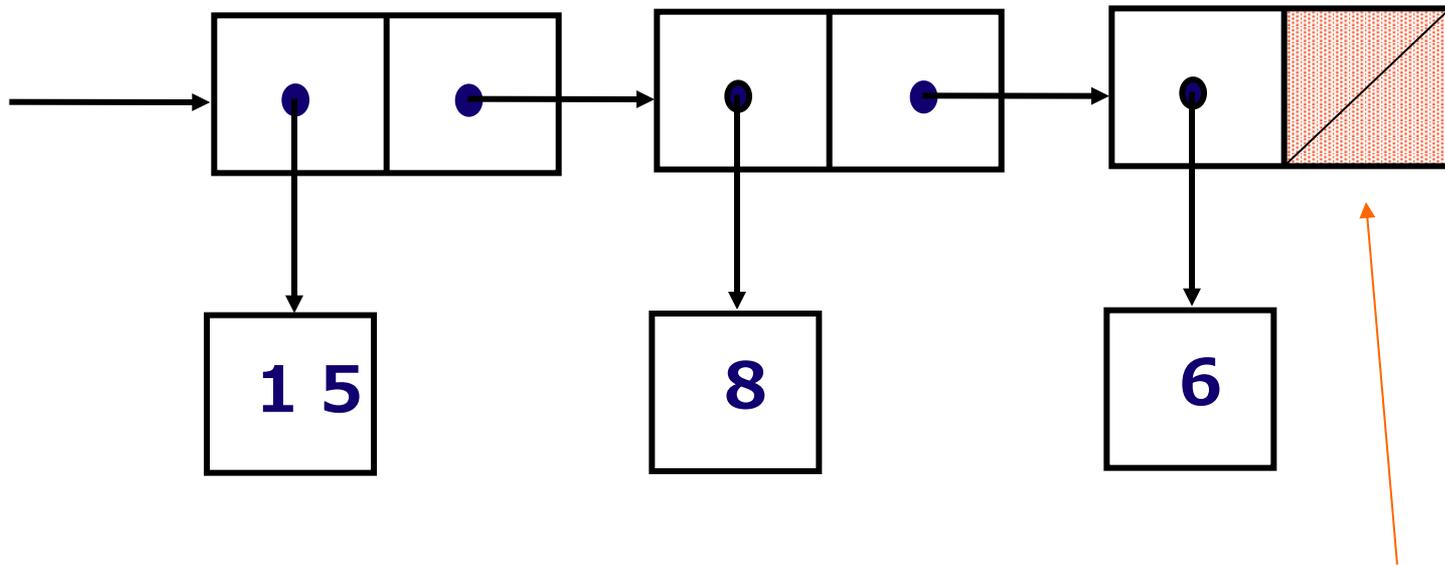


リストの car と cdr



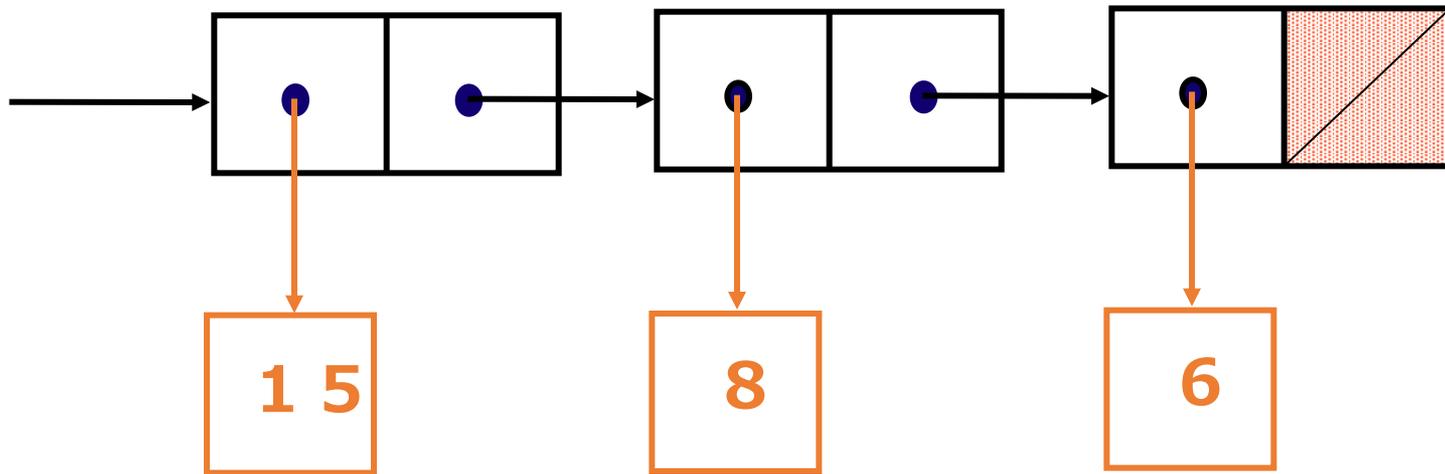
- リストの car:
 - リストの先頭要素
- リストの cdr:
 - リストから先頭要素を取り除いた残り (やはりリスト)

リストの末尾としての空リスト



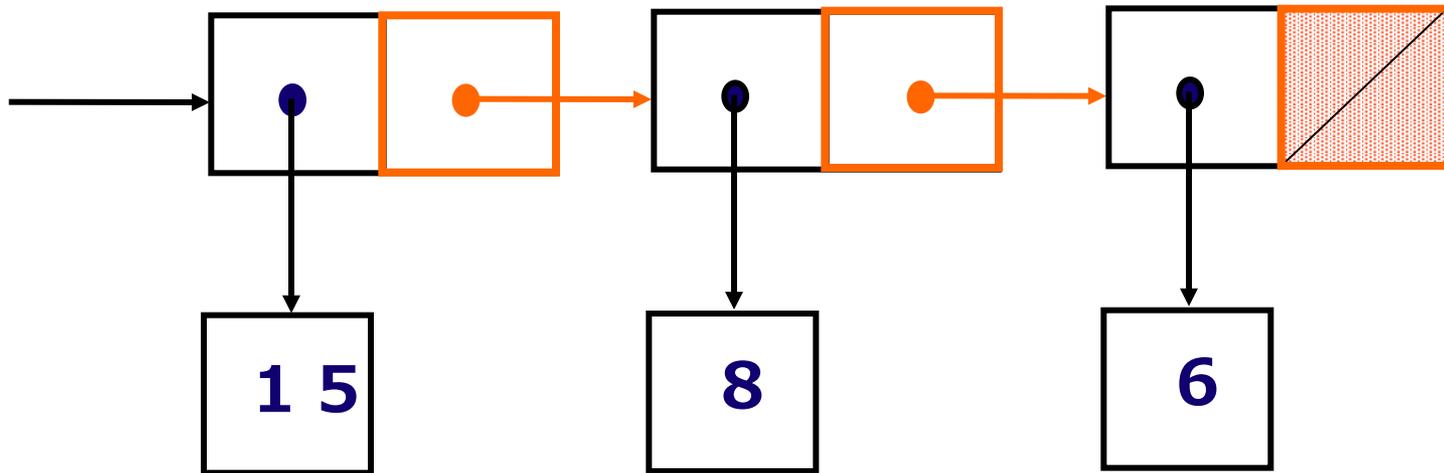
「空リストである」ことを示す特別な値が入っている

リストを構成するペアの性質 (1/2)



- リストを構成するペアの個数 : リストの要素数に等しい
- リストを構成するペアの car : リストの要素が入る

リストを構成するペアの性質 (2/2)



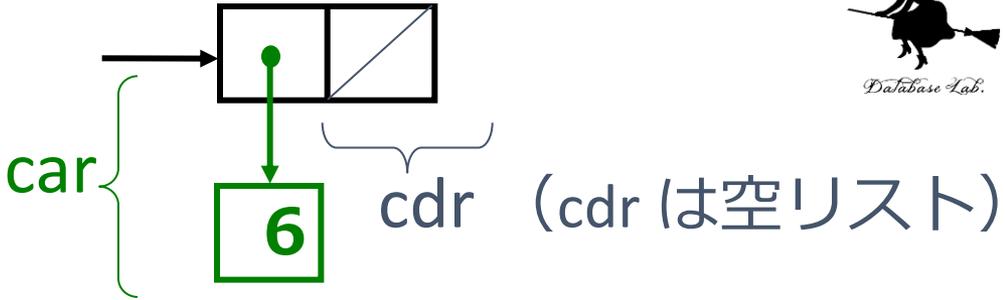
- リストを構成するペアの右側のセル(cdr フィールド)
 - 次の要素へのポインタか, 「空リスト」であることを示す特別な値 (リストの末端であることを示す) が入る

- Scheme では

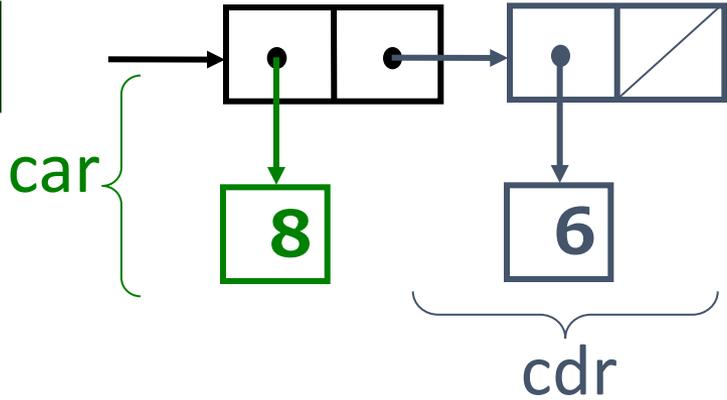
末尾が「空リスト」であるようなペアの並び

- 並びの最後のペアの cdr フィールドに空リスト が入っている
- 行儀の良いリスト (proper list) と呼ぶこともある

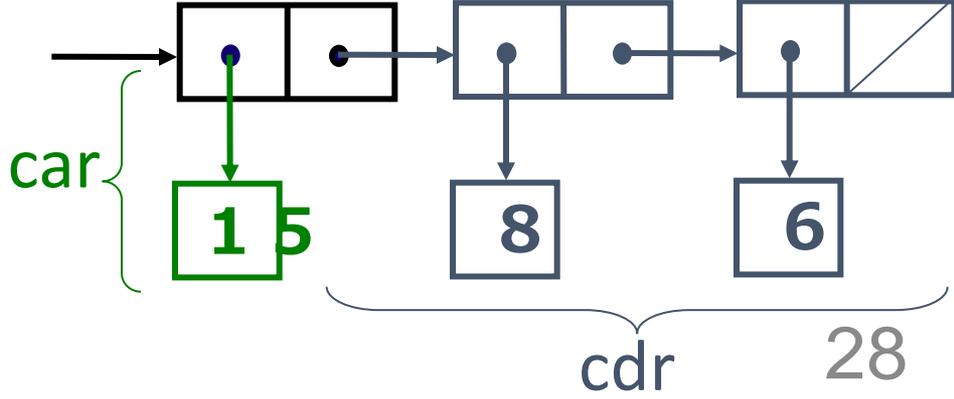
(cons 6 '())



(cons 8 (cons 6 '()))



(cons 15 (cons 8 (cons 6 '())))

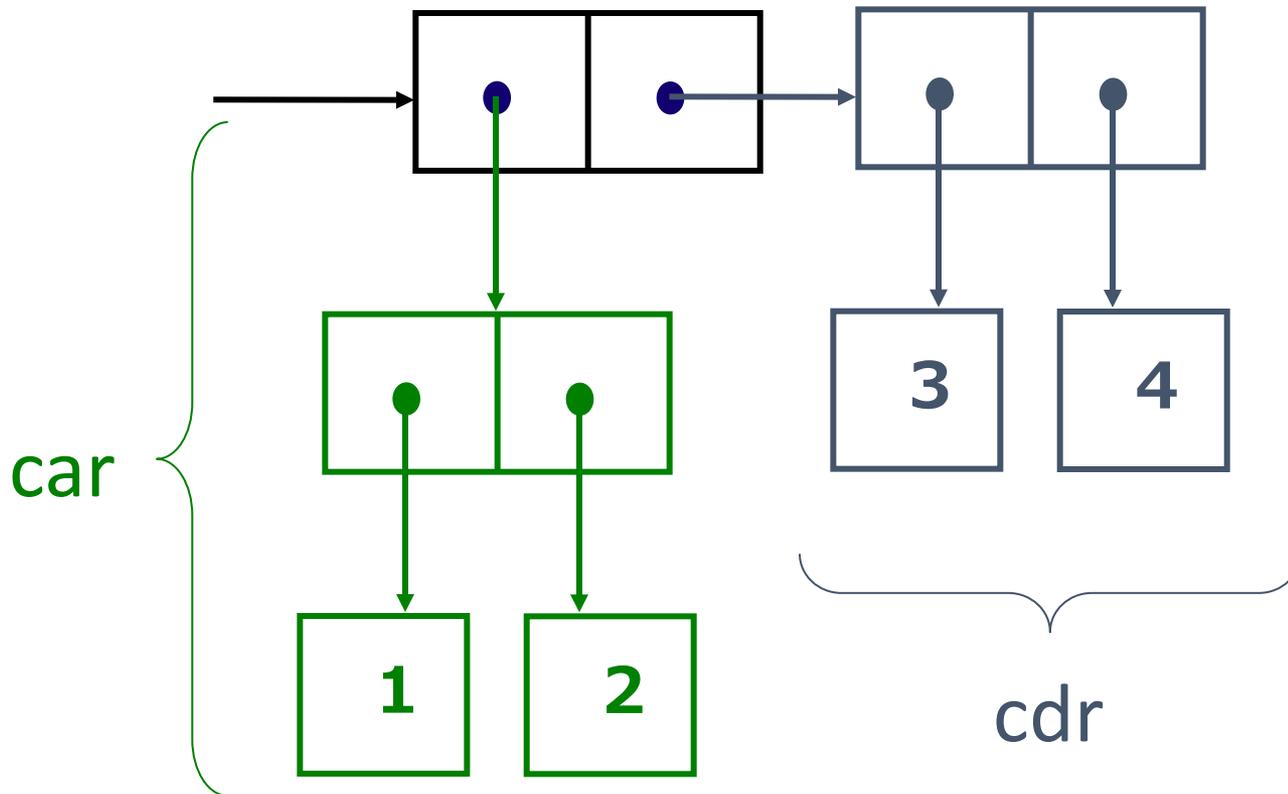


- リストでは：
 - リストと要素をつなげて，新しいリストを作る
- 一般的には：
 - データとデータをつなげて，新しいペアを作る

例題 3. ペアから構成されたペア



- 下記のペアを，変数aとして定義する



「例題 3. ペアから構成されたペア」の手順



1. 次を「定義用ウィンドウ」で、実行しなさい
 - 入力した後に、Execute ボタンを押す

```
(define a (cons (cons 1 2)
                 (cons 3 4)))
```

2. その後、次を「実行用ウィンドウ」で実行しなさい

```
a
(car a)
(cdr a)
```



Untitled



(define ...)

```
(define a (cons (cons 1 2) (cons 3 4)))
```

ペアの生成

> a

```
((1 . 2) 3 . 4)
```

表示されたペア

> (car a)

```
(1 . 2)
```

> (cdr a)

```
(3 . 4)
```

14:3

Unlocked



not running

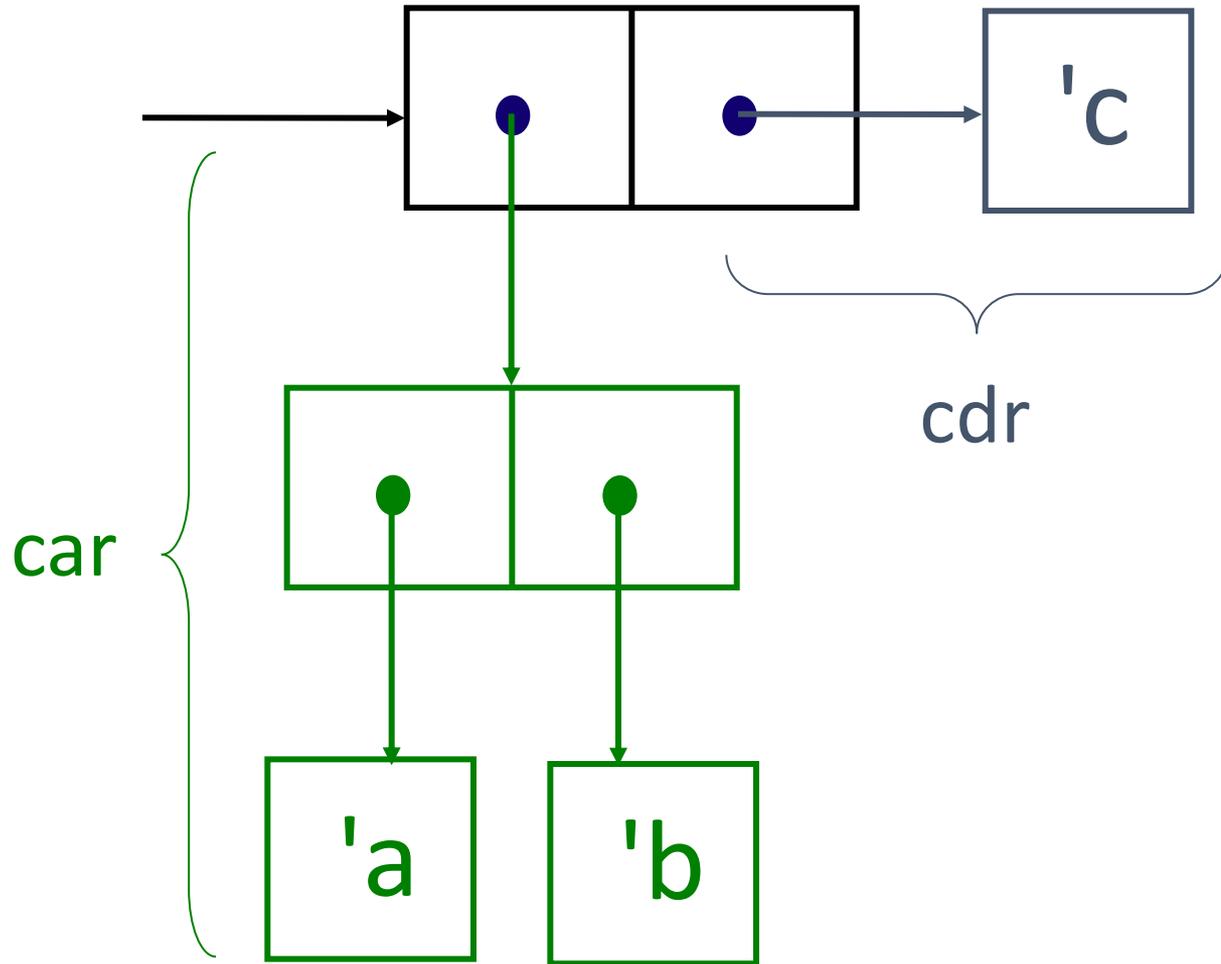
例題 3 のプログラム例



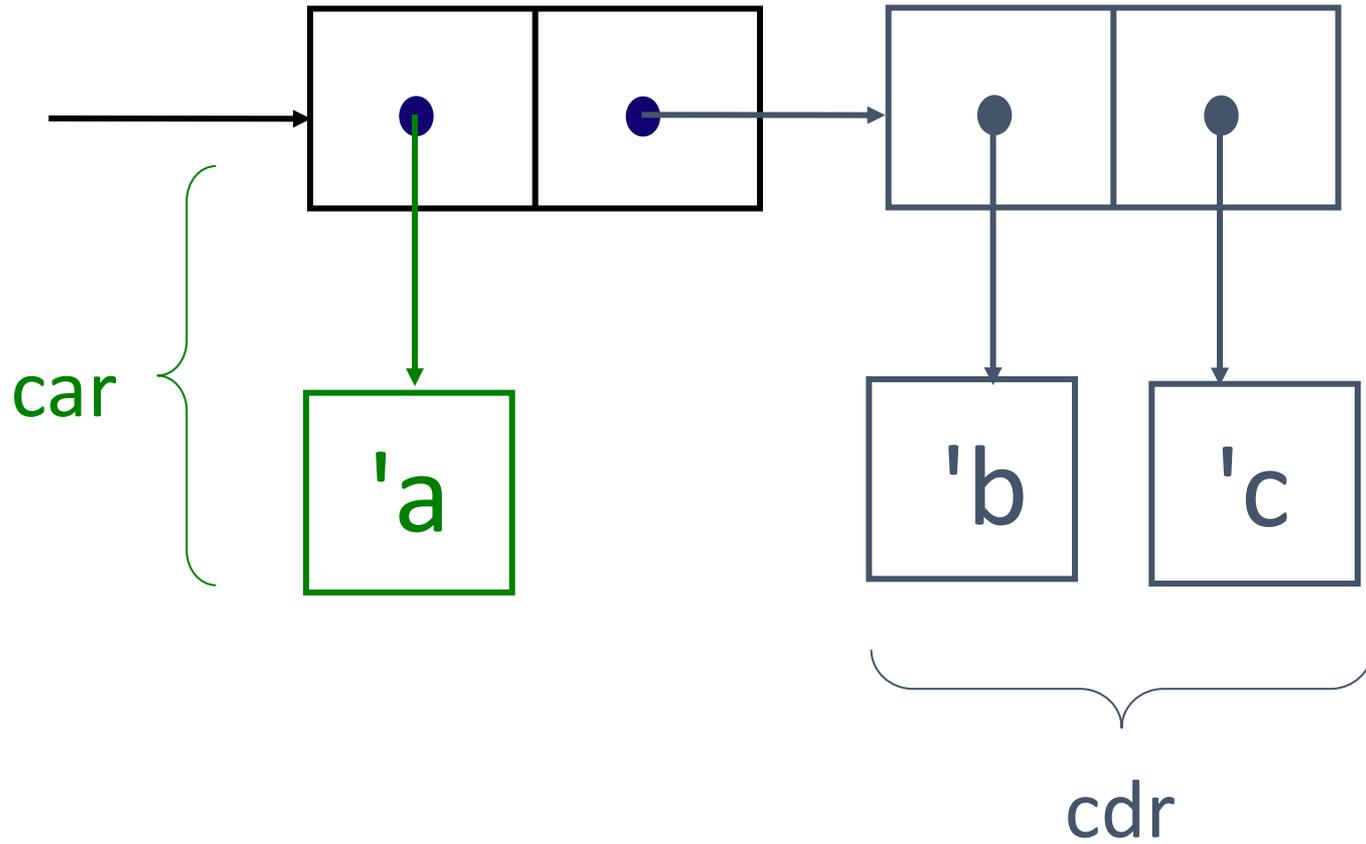
```
(define a (cons (cons 1 2)
                (cons 3 4)))
```

ペアから構成されたペアは、
「cons の入れ子」で書ける

(cons (cons 'a 'b) 'c) の相とポイント記法



(cons 'a (cons 'b 'c)) の相とポインタ記法



cons によるペアの表記

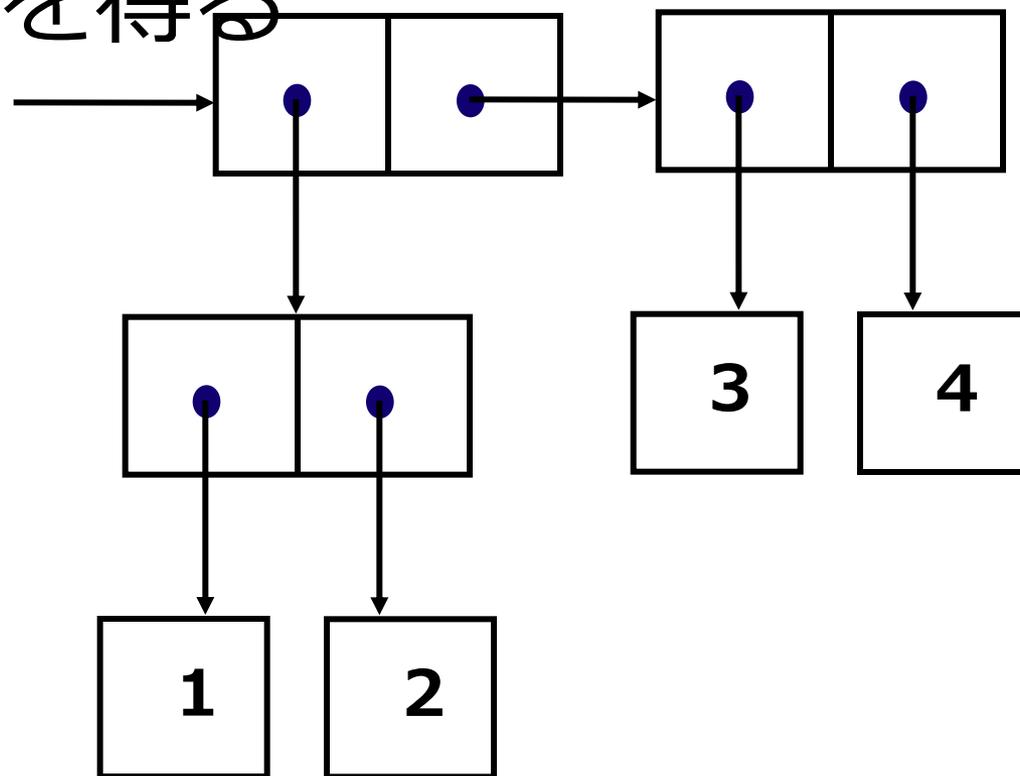
A screenshot of a Scheme REPL window. The window title is "Untitled". The toolbar contains buttons for "Check Syntax", "Step", "Execute", and "Break". The main text area shows four REPL interactions. The first interaction shows the command `(cons 'a 'b)` resulting in `(a . b)`. The second shows `(cons (cons 'a 'b) 'c)` resulting in `((a . b) . c)`. The third shows `(cons 'a (cons 'b 'c))` resulting in `(a b . c)`. The fourth shows `(cons (cons 'a 'b) (cons 'c 'd))` resulting in `((a . b) c . d)`. The bottom status bar shows "43:3", "Unlocked", and "not running".

```
Untitled  
(define ...)  
Check Syntax Step Execute Break  
  
> (cons 'a 'b)  
(a . b)  
> (cons (cons 'a 'b) 'c)  
((a . b) . c)  
> (cons 'a (cons 'b 'c))  
(a b . c)  
> (cons (cons 'a 'b) (cons 'c 'd))  
((a . b) c . d)  
43:3 Unlocked not running
```

例題 4 . car と cdr の組み合わせ



- 例題 3 のペアについて, car と cdr を組み合わせせて, 「1」, 「2」, 「3」, 「4」を得る



「例題 3. car と cdr の組み合わせ」の手順

1. 次を「定義用ウィンドウ」で、実行しなさい
 - 入力した後に、Execute ボタンを押す

```
(define a (cons (cons 1 2)
                 (cons 3 4)))
```

2. その後、次を「実行用ウィンドウ」で実行しなさい

```
(car (car a))
(cdr (car a))
(car (cdr a))
(cdr (cdr a))
(caar a)
(cdar a)
(cadr a)
(cddr a)
```

☆ 次は、例題 4 に進んでください



Untitled Save Check Syntax Step Execute Break

```
(define a (cons (cons 1 2)
                (cons 3 4)))
```

```
> (car (car a))
1
> (cdr (car a))
2
> (car (cdr a))
3
> (cdr (cdr a))
4
```

12:3 Unlocked not running

Untitled



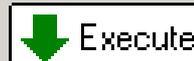
Save



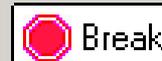
Check Syntax



Step



Execute



Break



Database Lab

(define ...)

```
(define a (cons (cons 1 2)
                (cons 3 4)))
```

> (caar a)

1

> (cdar a)

2

> (cadr a)

3

> (cddr a)

4

20:3

Unlocked

not running

例題 4 のプログラム例



```
(define a (cons (cons 1 2)
                (cons 3 4)))
```

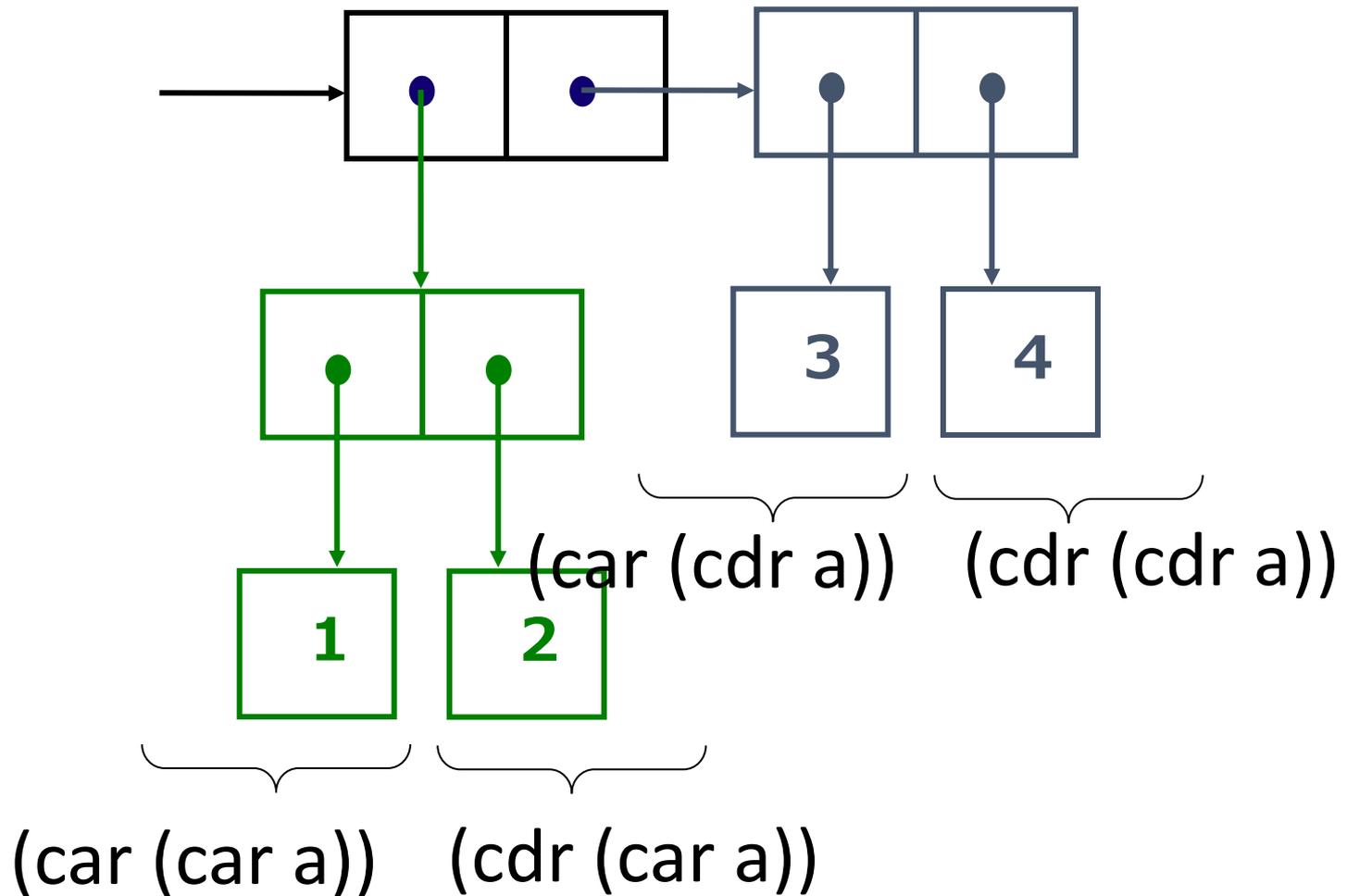
```
(car (car a))
```

```
(cdr (car a))
```

```
(car (cdr a))
```

```
(cdr (cdr a))
```

car, cdr の組み合わせ



car, cdr の組み合わせ



- (caar pair)
- (cadr pair)
- ...
- (cdddr pair)

car, cdr を 4 つまで組み合わせることができる

ペアに関する関数

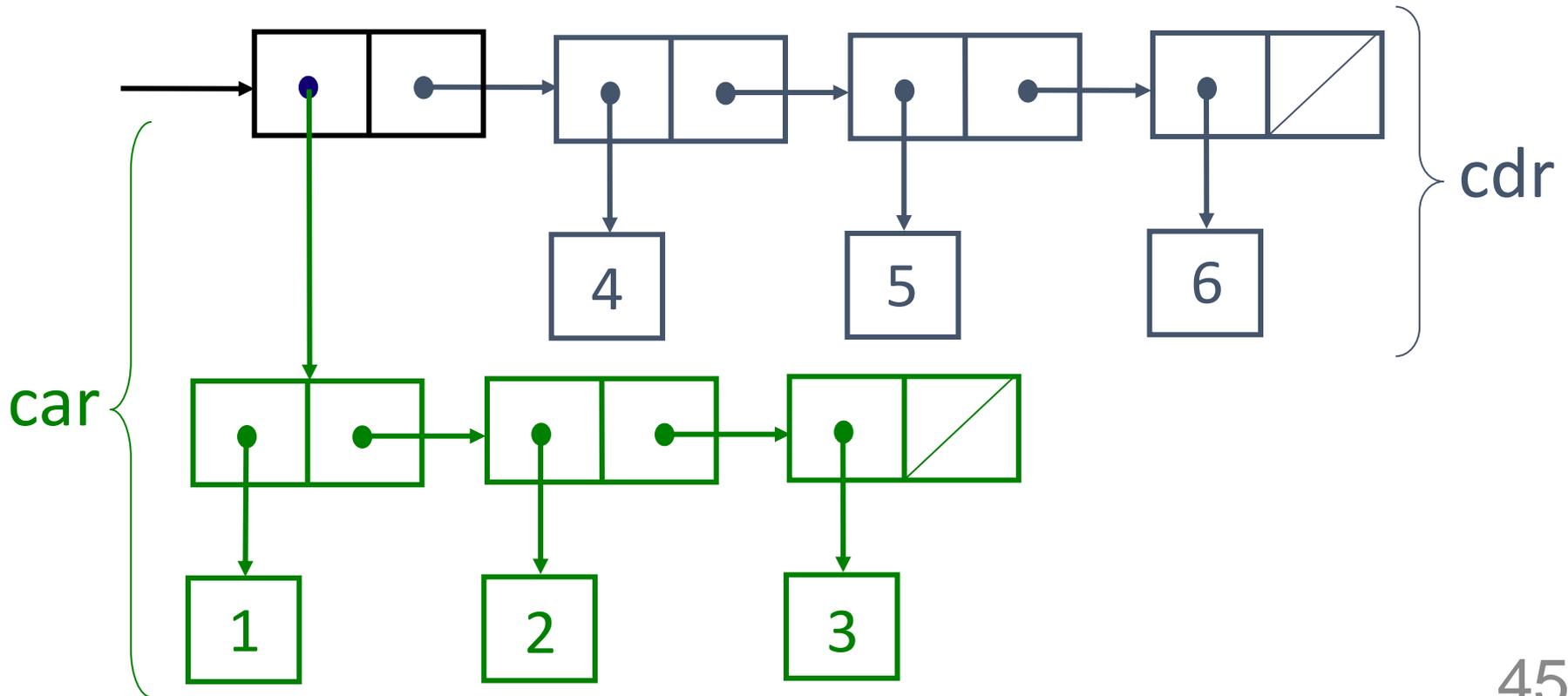


- (cons obj1 obj2)
ペアの生成
- (car pair)
car の取り出し
- (cdr pair)
cdr の取り出し
- (caar pair)
- (cadr pair)
- ...
- (caddr pair)
car, cdr を4つまで組み合わせることができる

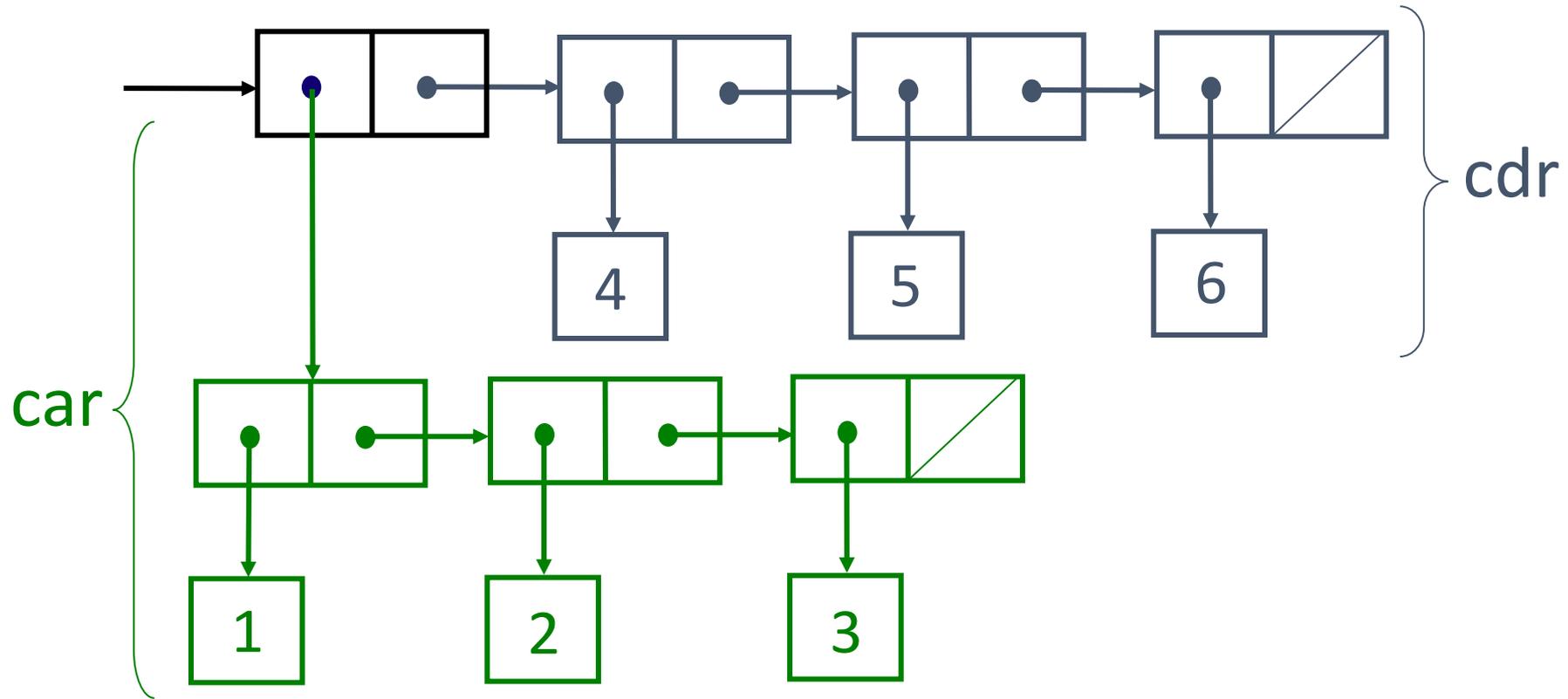
例題 5. cons と list の組み合わせ(1)



- 下記のようなペアの集まりを，変数 x として定義する



cons と list の組み合わせ (1/2)



```
(define x (cons (list 1 2 3)
                (list 4 5 6)))
```

Untitled - DrScheme

File Edit Windows Show Language Scheme Help

Untitled Save Check Syntax Step Execute Break

```
(define x (cons (list 1 2 3)
                (list 4 5 6)))
```

```
> x
((1 2 3) 4 5 6)
> (car x)
(1 2 3)
> (cdr x)
(4 5 6)
```

11:3 not running

Untitled - DrScheme

File Edit Windows Show Language Scheme Help

Untitled Save Check Syntax Step Execute Break

```
(define x (cons (list 1 2 3)
                (list 4 5 6)))
```

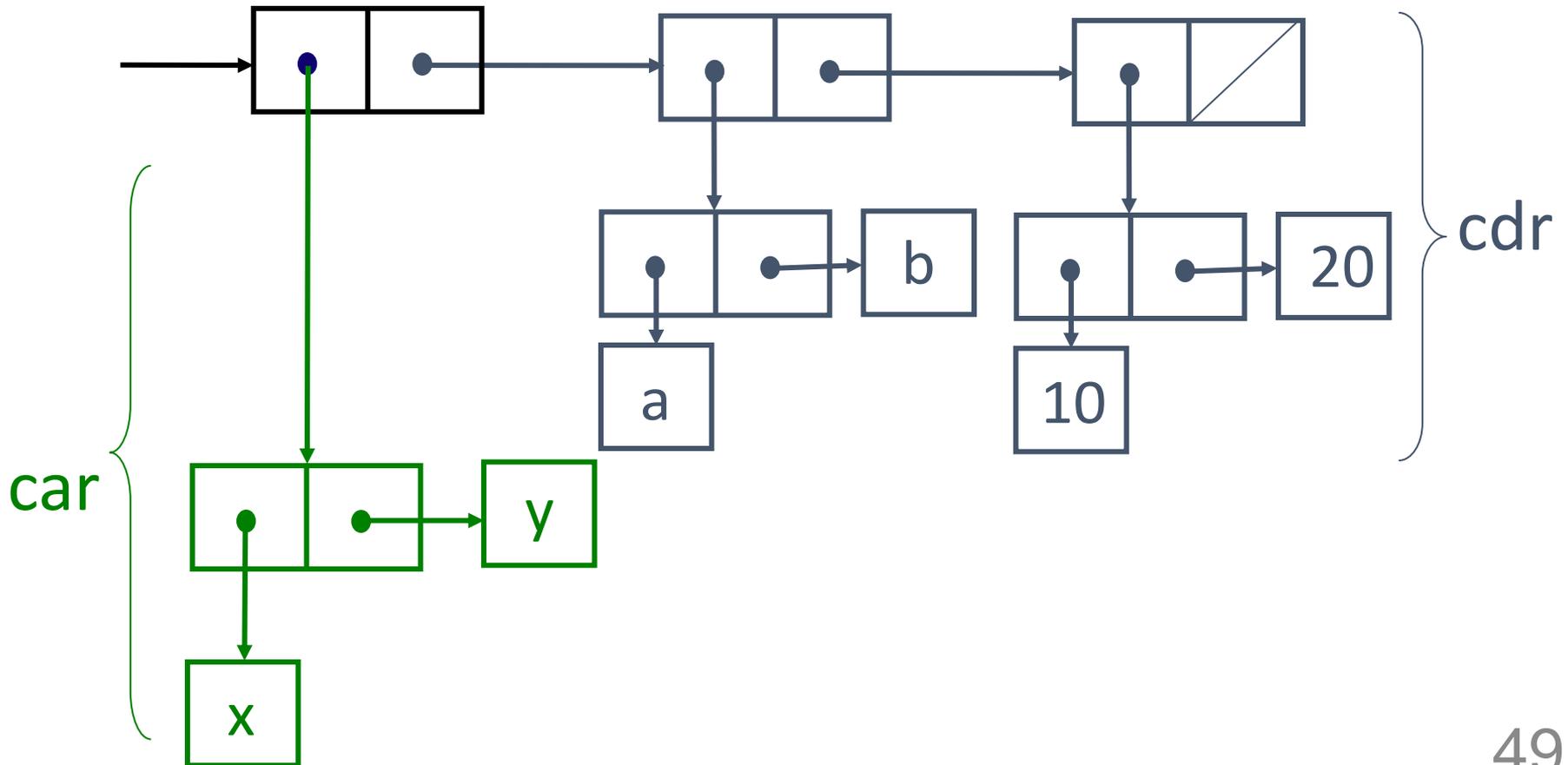
```
> x
((1 2 3) 4 5 6)
> (car x)
(1 2 3)
> (cdr x)
(4 5 6)
```

11:3 not running

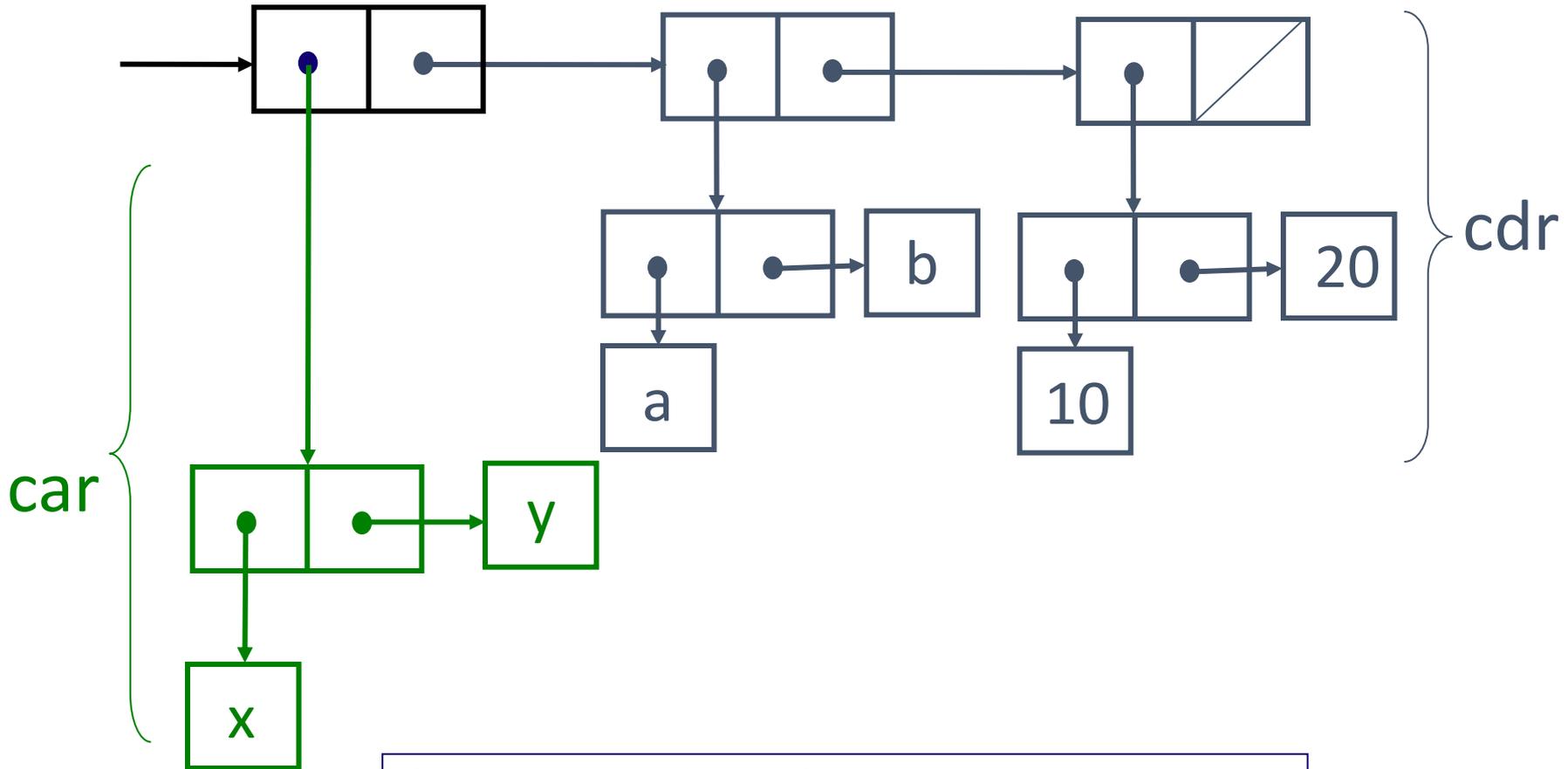
例題 6. cons と list の組み合わせ(2)



- 下記のようなペアの集まりを，変数 x として定義する



cons と list の組み合わせ (2/2)



```
(define x (list (cons 'x 'y)
                (cons 'a 'b)
                (cons 10 20)))
```

Untitled - DrScheme

File Edit Windows Show Language Scheme Help

Untitled Save Check Syntax Step Execute Break

```
(define x (list (cons 'x 'y)
                (cons 'a 'b)
                (cons 10 20)))
```

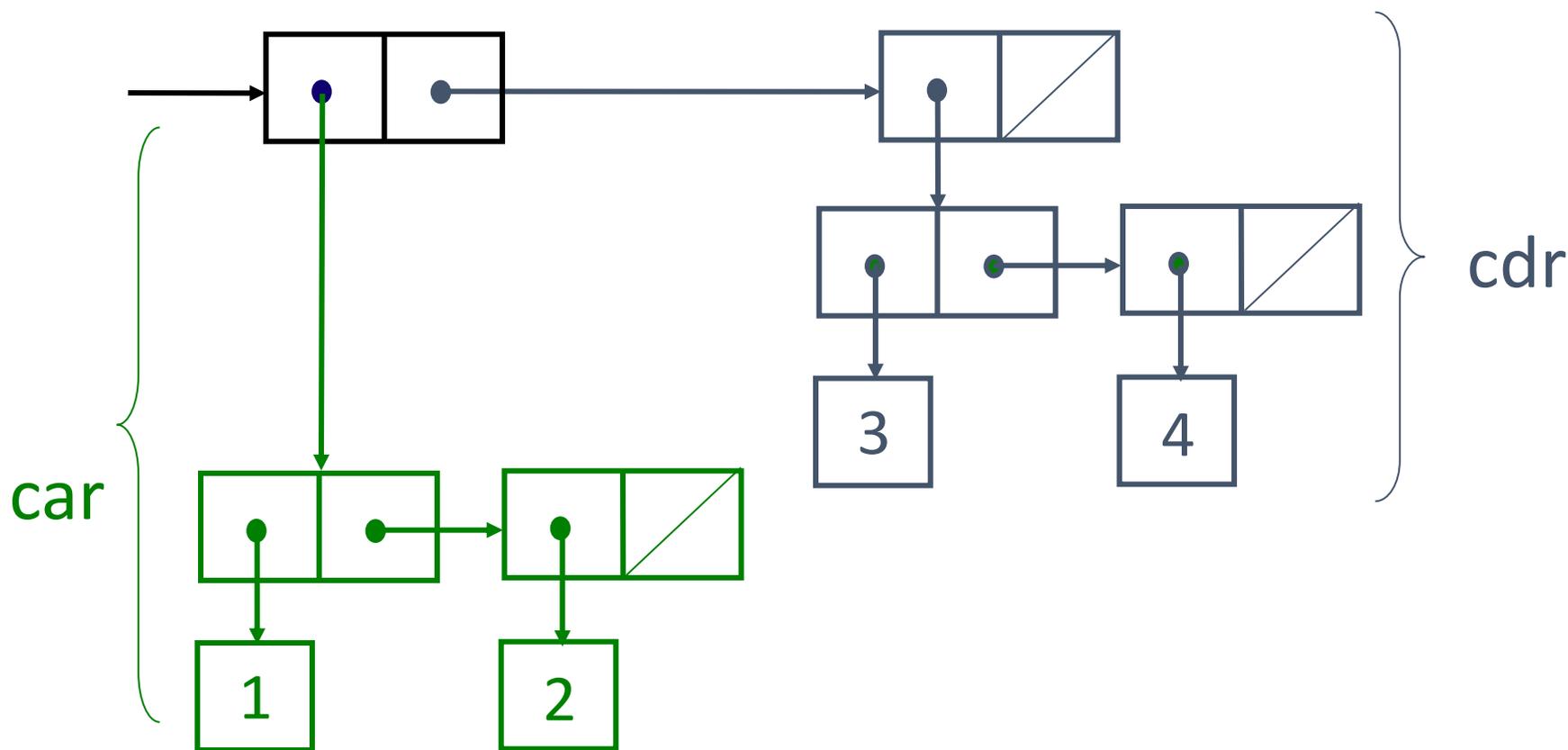
```
> x
((x . y) (a . b) (10 . 20))
> (car x)
(x . y)
> (cdr x)
((a . b) (10 . 20))
```

11:3 Unlocked not running

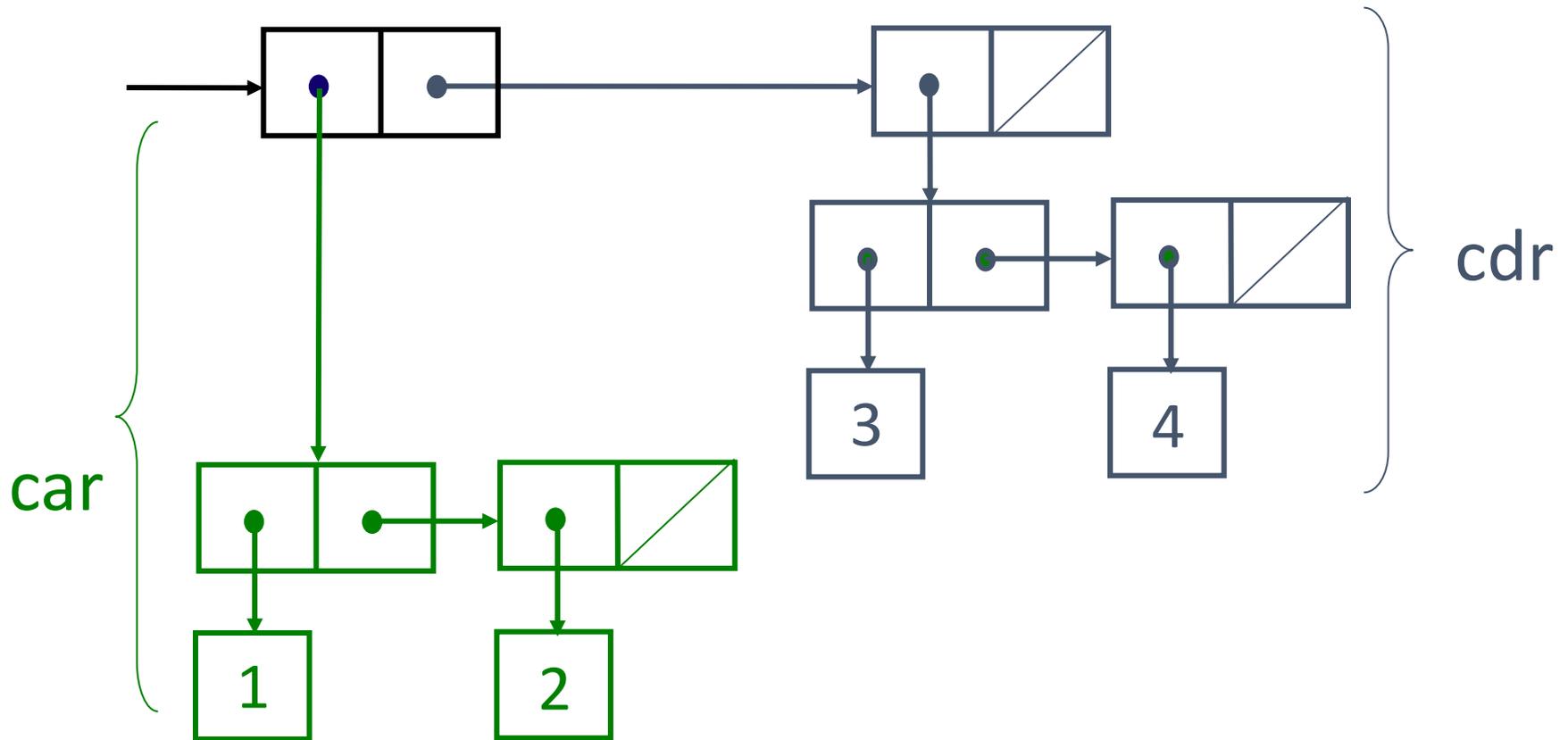
例題 7. list と list の組み合わせ



- 下記のようなペアの集まりを，変数 x として定義する



list と list の組み合わせ



```
(define x (list (list 1 2)
                (list 3 4)))
```

Untitled - DrScheme

File Edit Windows Show Language Scheme Help

Untitled Save Check Syntax Step Execute Break

```
(define x (list (list 1 2)
                (list 3 4)))
```

> x
((1 2) (3 4))

> (car x)
(1 2)

> (cdr x)
((3 4))

11:3 Unlocked not running

ドット対の例

(cons 'a 'b)

⇒ (a . b) と表示される

(cons (cons 'a 'b) 'c)

⇒ ((a . b) . c) と表示される

(cons 'a (cons 'b 'c))

⇒ (a b . c) と表示される

(cons (cons 'a 'b) (cons 'c 'd))

⇒ ((a . b) c . d) と表示される

Untitled

(define ...)

Check Syntax

Step

Execute

Break



```
> (cons 'a 'b)
(a . b)
> (cons (cons 'a 'b) 'c)
((a . b) . c)
> (cons 'a (cons 'b 'c))
(a b . c)
> (cons (cons 'a 'b) (cons 'c 'd))
((a . b) c . d)
```

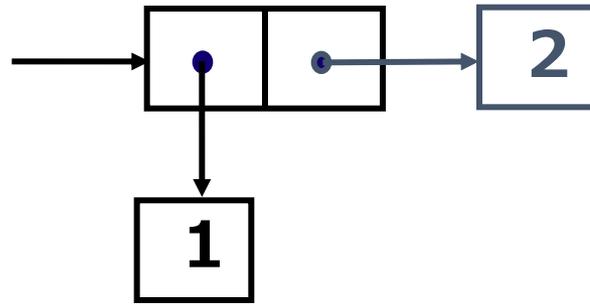
43:3

Unlocked

not running

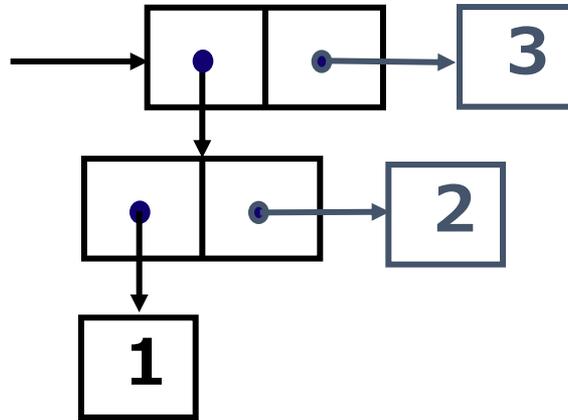
- ペアの cdr がリストになっていない場合
 - つまり, cdr 方向にペアの並びをみたときに, 末尾が「空リスト」になっていなければ
- ⇒ ドットを, 末尾の要素の前に追加

(cons 1 2)



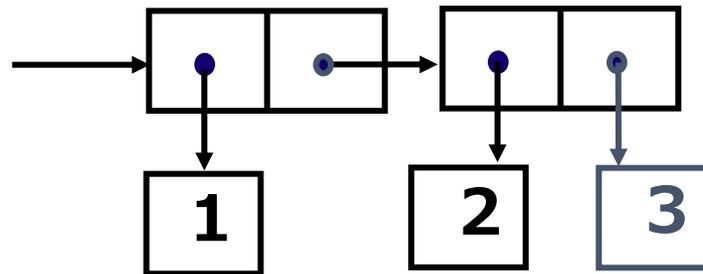
(1 . 2)

(cons (cons 1 2) 3)



((1 . 2) . 3)

(cons 1 (cons 2 3))



(1 2 . 3)

16-3 課題

課題①



- 実行結果を報告しなさい
 - 実行上の注意： DrScheme で、必ず「Full Scheme」を選んでから実行すること。

	<code>(list (cons 1 2) (cons 3 4))</code>	<code>(list (list 1 2) (list 3 4))</code>
<code>(car (list ...))</code> の実行結果		
<code>(cdr (list ...))</code> の実行結果		
<code>(cadr (list ...))</code> の実行結果		
<code>(caddr (list ...))</code> の実行結果		

さらに勉強したい人への 補足説明事項

二分探索木

例題 8. 二分探索木

例題 9. 二分探索木による探索

- ・ 入れ子になった構造

- 幾つかの節点(node)と、それらを結ぶ枝(branch)から構成
 - 節点がデータに対応
 - 枝がデータ間の親子関係に対応
 - 子：節点の中で下方に分岐する枝の先にあるもの
 - 親：分岐元の節点

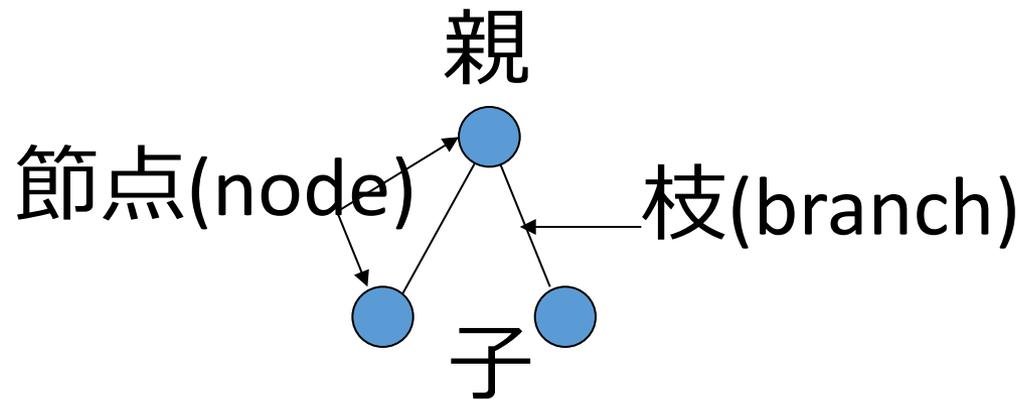
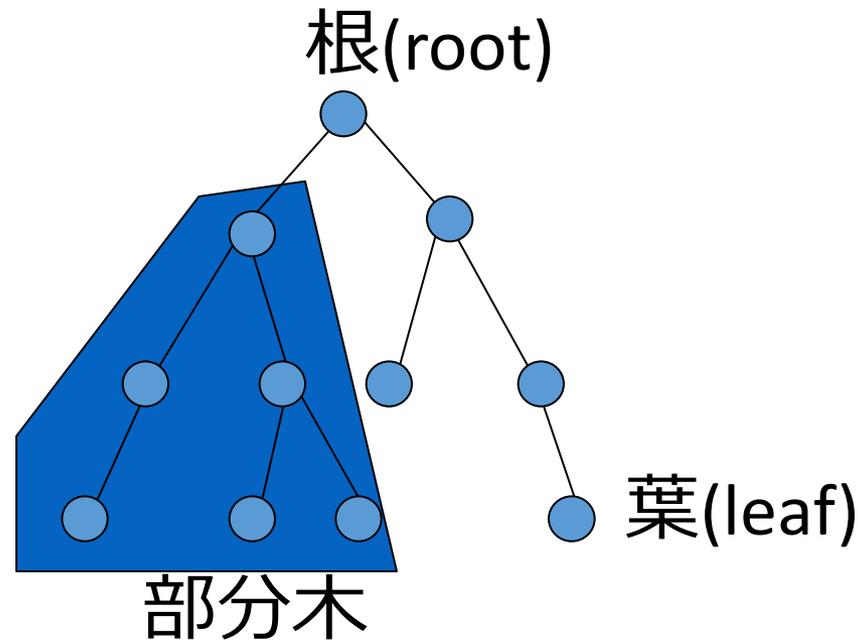


図. 単純な木構造



- **根(root)** : 木の一番上の節点を根(root)
- **葉(leaf)** : 子を持たない節点
- **部分木** : 木の中のある節点を相対的な根と考えたときの, そこから枝分かれした枝と節点の集合

二分木 (binary tree)

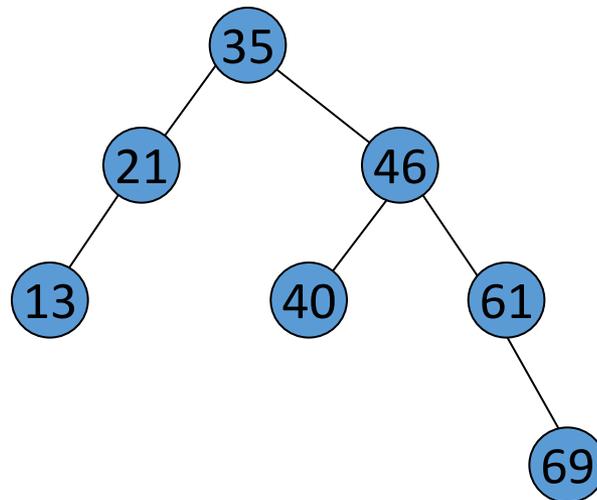


- 木構造で，各節点から出る枝が二本以下のもの
- 木構造に関するアルゴリズムの中で，中心的なデータ構造

二分探索木 (binary search tree)



- 二分木の種類
- データの配置に規則あり
 - 左側のすべての子は親より小さい
 - 右側のすべての子は親より大きい
- データの探索のためのデータ構造



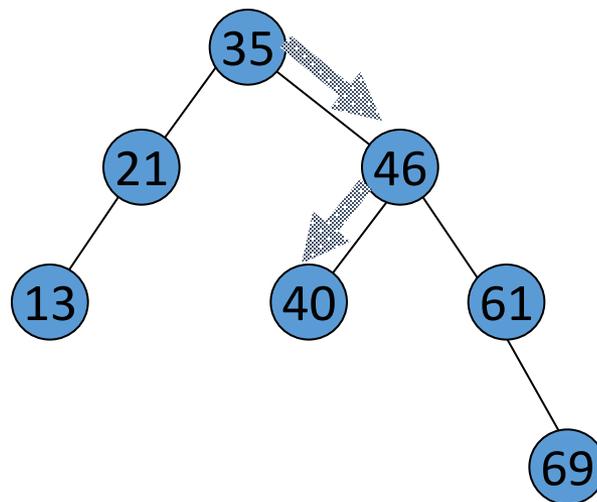
- 根(root)から始める
- 探索キーの値と、各節点のデータを比較し、目標となるデータを探す
 - 探索キーよりも節点のデータが小さいときは、右側の子をたどる
 - 探索キーよりも節点のデータが大きいときは、左側の子をたどる

二分探索木による探索の例



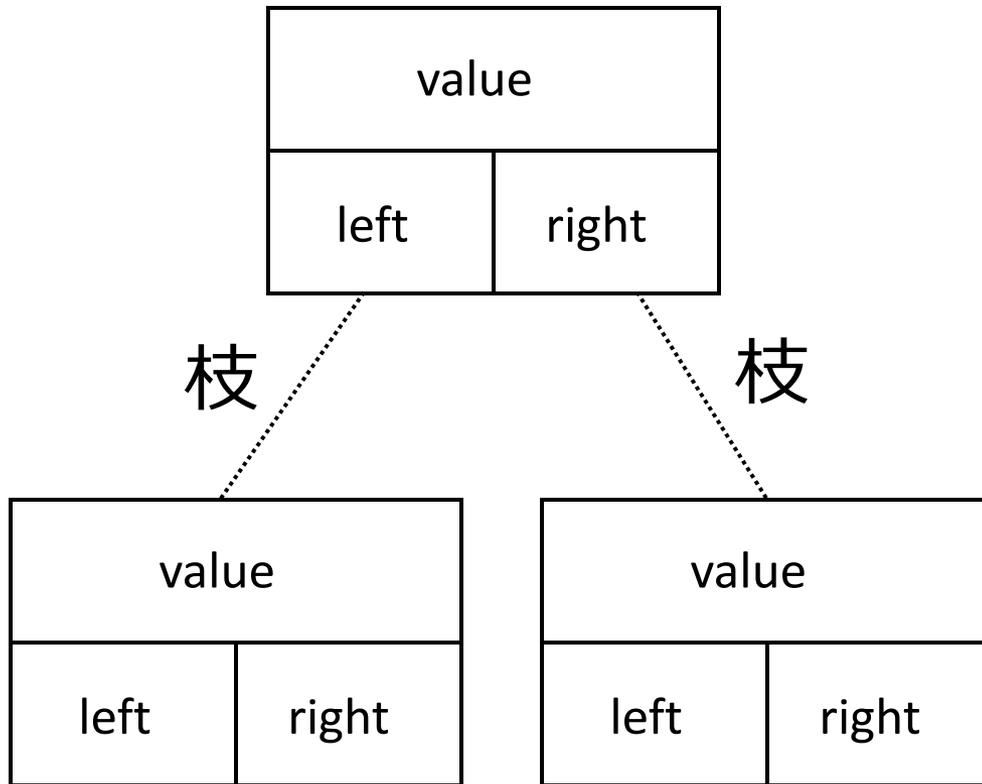
(例) 40である節点を探す場合

1. 根の値(35)と, 探索キー(40)を比較
2. 探索キーの方が大きいので, 右側の子節点へ移る
3. 次に移った節点の値(46)と探索キー(40)を比較し
4. 探索キーの方が小さいので, 左の子節点へ移る
5. 次に移った節点(40)が, 目標の節点である



- 複雑なプログラムを作成する時，データ構造について考える必要がある
- データ構造
 - アルゴリズムを容易にするために工夫されたデータの並び
 - 基本的なデータ構造は，配列，キュー，スタック，リスト構造，木構造など

二分探索木のための node structure



(define-struct node
 (value left right))

例題 8 . 二分探索木



- 二分探索木のプログラムを作り，実行する
 - 二分探索木の節点を扱うために，`define struct` 文を使って，`node structure` を定義する
 - 1つの二分探索木は，節点が集まって，入れ子の構造になる

二分探索木の節点



- 二分探索木の節点を, `define-struct` 文を使って定義する

名前

```
(define-struct node  
  (value left right))
```

属性の並び

(それぞれの属性にも名前がある)

```
(define-struct node  
  (value left right))
```

- 上記のプログラムの実行によって
 - make-node
 - node-value
 - node-left
 - node-rightが使えるようになる
- } 属性 value, left, right の取得

(make-node 35

(make-node 21

(make-node 13 false false)
false)

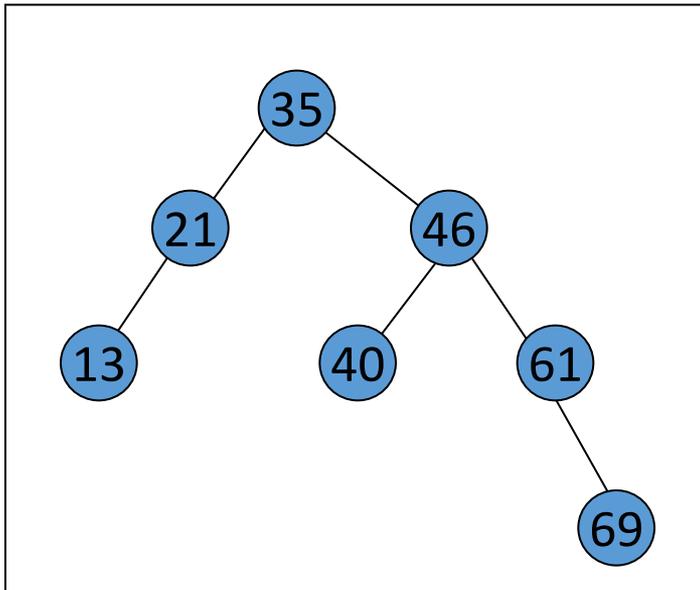
(make-node 46

(make-node 40 false false)

(make-node 61

false

(make-node 69 false false))))



上のプログラムが表現する
2分探索木

(「false」は「データが無い」ことを示す特別な値)

例題 9 . 二分探索木による探索



- 二分探索木の探索のプログラムを作り、実行する
 - データが二分探索木の中にあれば
 - true
 - 無ければ
 - false

```
(define-struct node
```

```
  (value left right))
```

```
(define (search x a-tree)
```

```
(cond
```

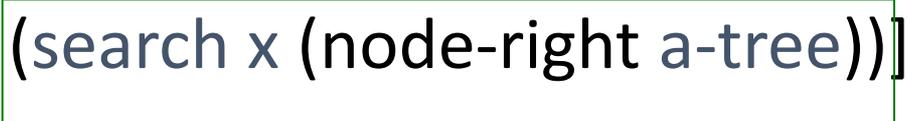
```
  [(eq? a-tree false) false]
```

```
  [(< x (node-value a-tree)) (search x (node-left a-tree))]
```

```
  [(< (node-value a-tree) x) (search x (node-right a-tree))]
```

```
  [else true]))
```

左を探す



右を探す

```
(define-struct node
  (value left right))
(define (search x a-tree)
  (cond
    [(eq? a-tree false) false]
    [(< x (node-value a-tree)) (search x (node-left a-tree))]
    [(< (node-value a-tree) x) (search x (node-right a-tree))]
    [else true]))
```

```
> (define tree
  (make-node 35
    (make-node 21
      (make-node 13 false false)
      false)
    (make-node 46
      (make-node 40 false false)
      (make-node 61
        false
        (make-node 69 false false)))))

> (search 20 tree)
false
> (search 21 tree)
```

- 2分探索木を「入れ子になった node structure」
として表現した