

sp-15. リスト処理とクイックソート

(Scheme プログラミング)

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

金子邦彦



アウトライン

15-1 クイックソート

15-2 パソコン演習

15-3 課題



1. リストへの要素の挿入

2. インサーションソート

- 要素の挿入によるソート
- 「すでにソートされたリストへの要素の挿入」を繰り返すことで、ソートを行う

3. クイックソート

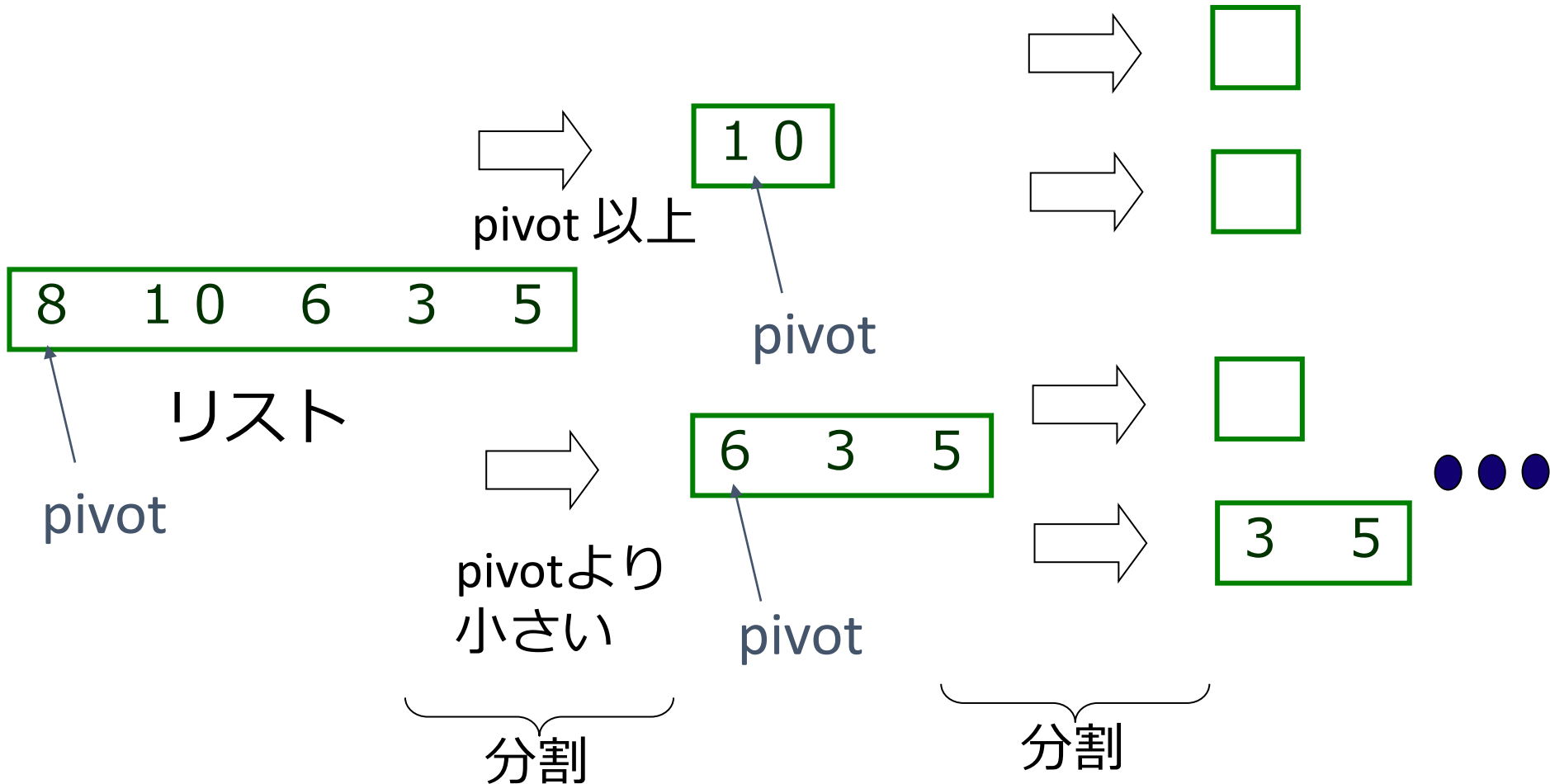
- 手順
- 再帰プログラム
- 分割統治法の考え方

4. 繰り返しでのステップ数

- ソートすべきデータサイズとステップ数の関係

15-1 クイックソート

クイックソートの考え方



リストが空になるまで, pivot の選択と, pivot による要素の分割を続ける

クイックソートの処理手順

1. 基準となるピボット (pivot) の選択

- ソートする範囲の中から pivot を1つ選ぶ

2. ピボットによるリストの分割

- smaller-items, larger-items を使用

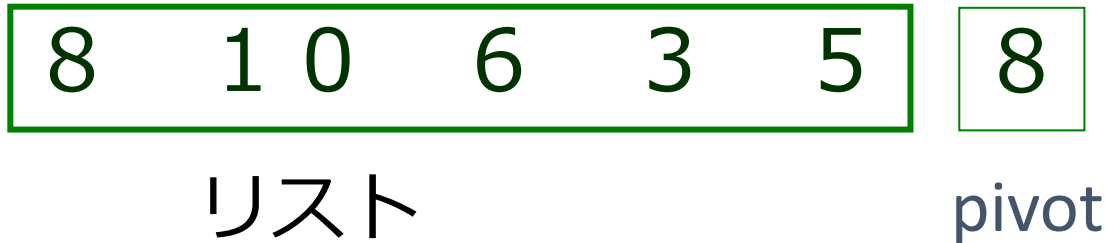
3. 1 と 2. を繰り返す

- 2. で出来た「pivot より大きい要素のリスト」と「pivot より小さい要素のリスト」のそれぞれを独立にソート

4. できた分割 (木構造) を使ってソートを実行

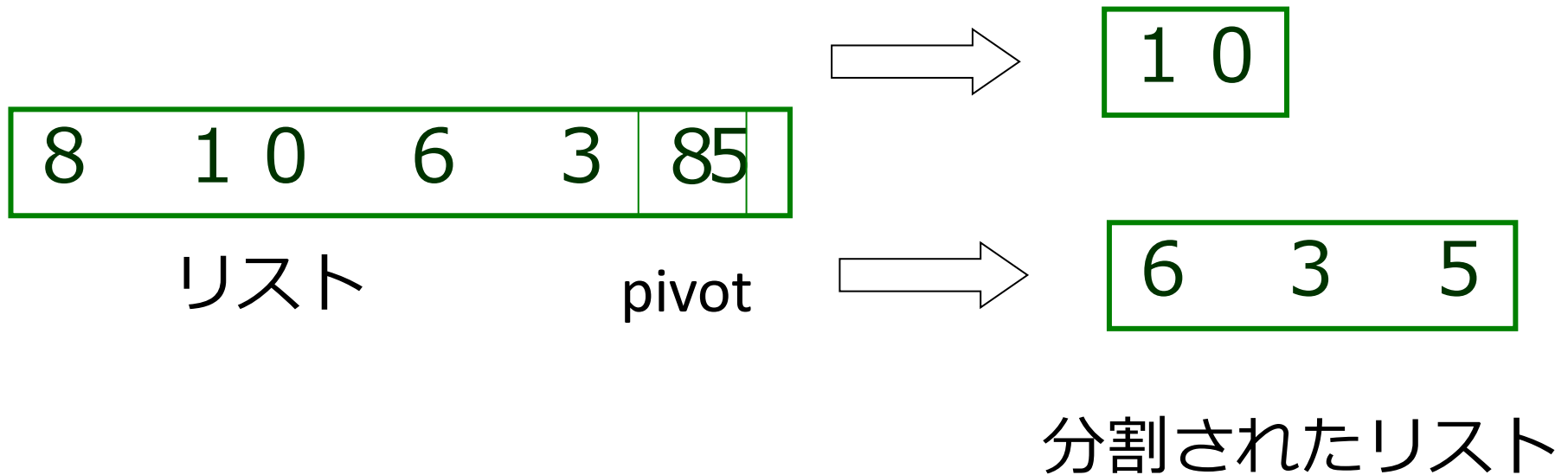
- 最後に、2つの部分と pivot をつなげる. 全体がソートできたことになる
- append を使用

pivot の選択

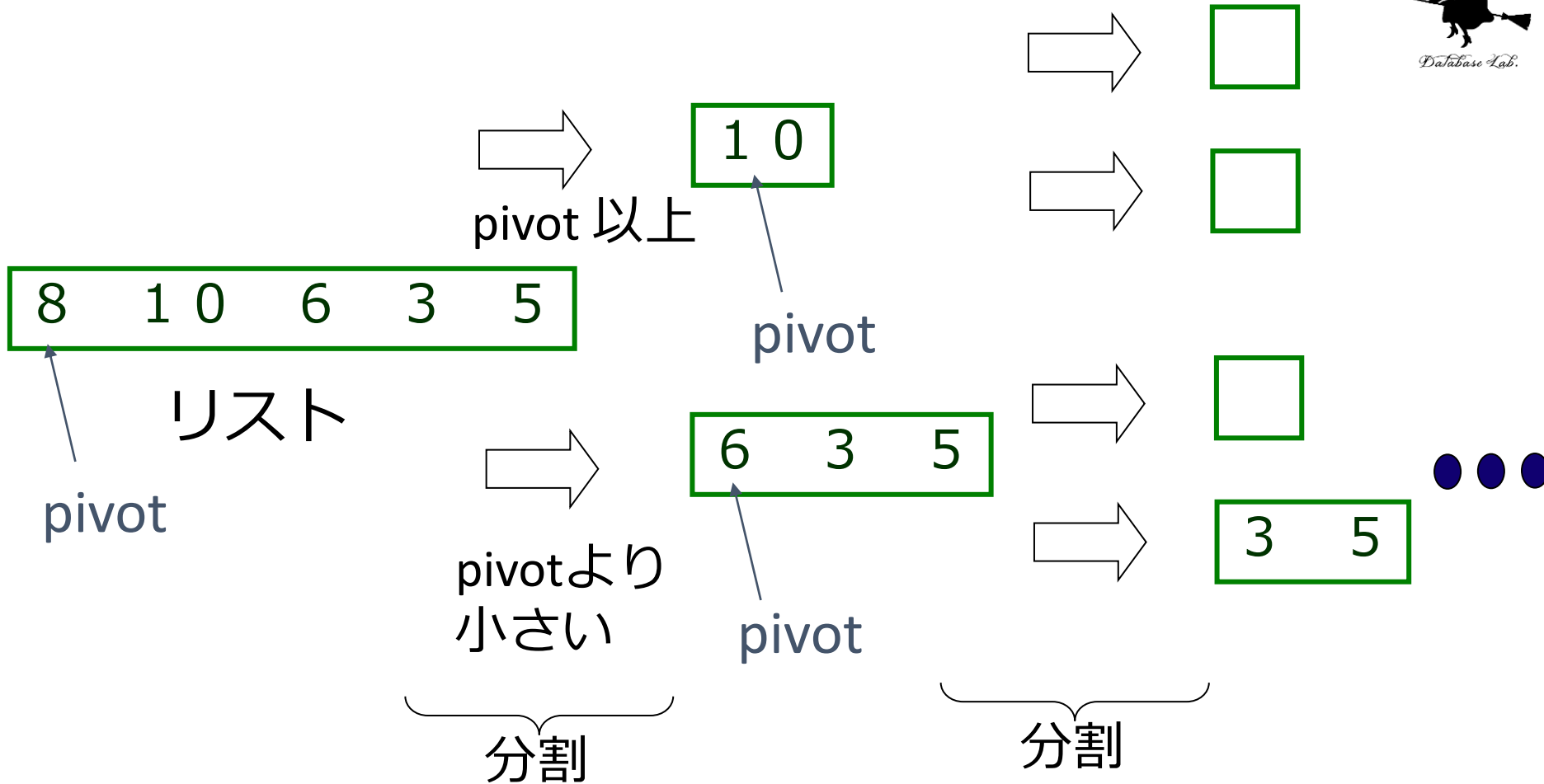


ソートする範囲の中から pivot を選ぶ
(ここでは, リストの先頭要素を pivot として
選んでいる)

pivot による要素の分割



要素を1つずつ調べて、pivot より
小さい要素と、より大きい要素を分ける



リストが空になるまで, pivot の選択と, pivot による要素の分割を続ける

15-2 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「Intermediate Student」
に設定
Language
→ Choose Language
→ Intermediate Student
→ Execute ボタン

例題 1. 要素の挿入



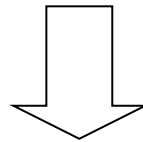
- ソート済みのリストに, 要素を挿入する関数 `insert` を作り, 実行する
 - ここでは, 要素が大きい順並んでいるものとする
 - 挿入を行うために, `cons` を使う

ソート済みのリスト

(80 21 10 7 5 4 3 1)

要素

40



(80 40 21 10 7 5 4 3 1)

「例題 1. 要素の挿入」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 (insert n alon)
  (cond
    [(empty? alon) (cons n empty)]
    [else (cond
      [(<= (first alon) n) (cons n alon)]
      [(> (first alon) n)
       (cons (first alon) (insert n (rest alon)))]))]))
```

2. その後、次を「実行用ウィンドウ」で実行しなさい

```
(insert 40 (list 80 21 10 7 5 4))
```

実行結果の例



Untitled - DrScheme

File Edit Windows Show Language Scheme Help

Untitled Save Check Syntax Step Execute Break

```
(define (insert n alon)
  (cond
    [(empty? alon) (cons n empty)]
    [else (cond
      [(<= (first alon) n) (cons n alon)]
      [(> (first alon) n)
        (cons (first alon) (insert n (rest alon)))]))]))
```

関数 insert の定義

Welcome to [DrScheme](#), version 103p1.
Language: ~~Intermediate Student~~.

```
> (insert 40 (list 80 21 10 7 5 4))
(list 80 40 21 10 7 5 4)
```

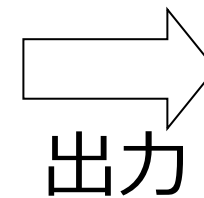
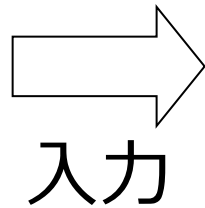
実行結果

5:3 Unlocked not running

入力と出力



(list 80 21 10 7 5 4)
40



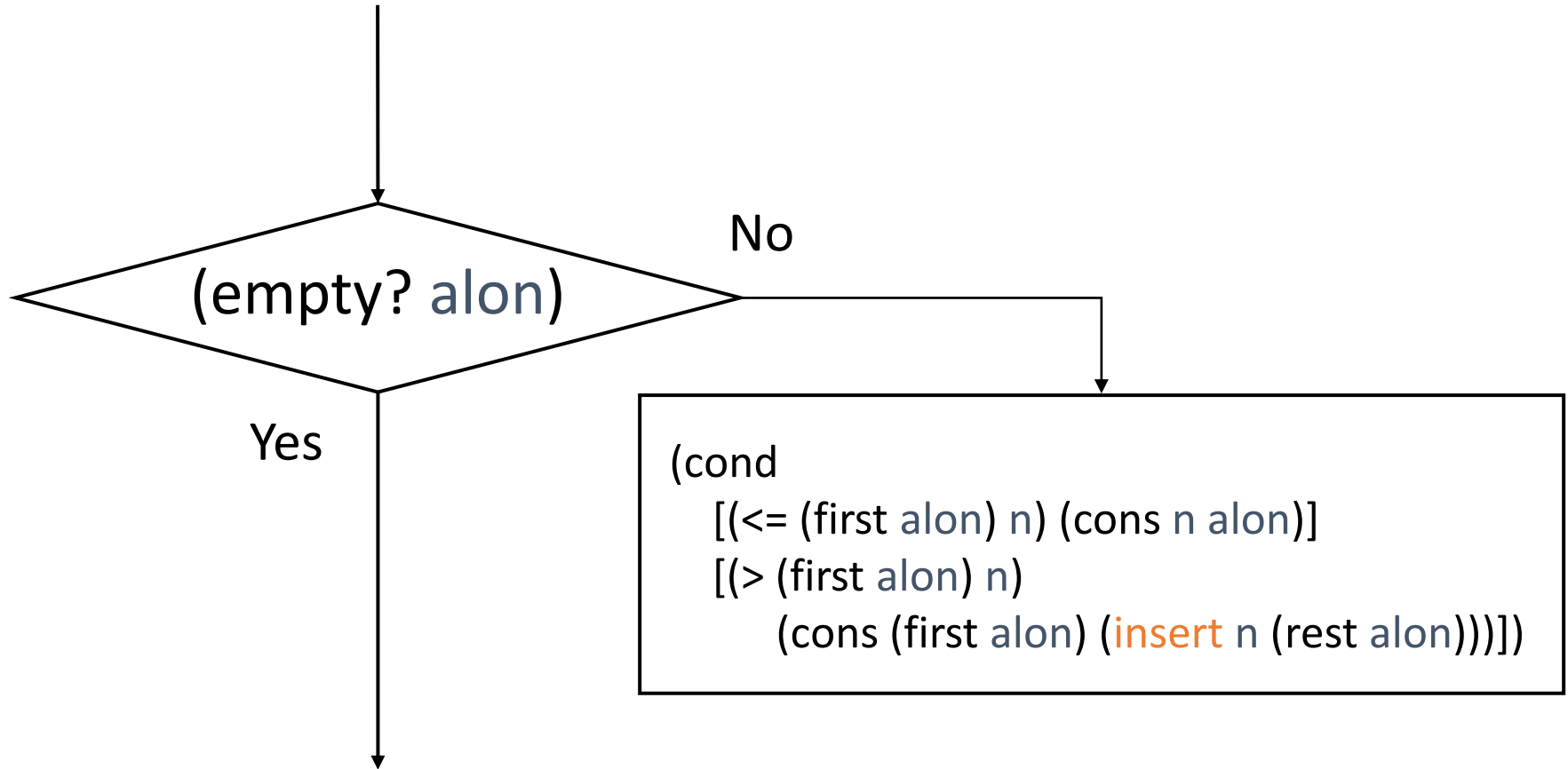
(list 80 40 21 10 7 5 4)

入力は、ソート済みのリスト
と数値

出力はソート済みの
リスト



```
;; insert: number list-of-numbers->list-of-numbers
;; to create a list of numbers from n and the numbers
;; on alon that is sorted in descending order; alon is
;; already sorted
;; insert: number list-of-numbers(sorted)
;;          -> list-of-numbers(sorted)
(define (insert n alon)
  (cond
    [(empty? alon) (cons n empty)]
    [else (cond
              [(<= (first alon) n) (cons n alon)]
              [(> (first alon) n)
               (cons (first alon) (insert n (rest alon)))]))]))
```

(cons n empty) が
自明の解である

要素の挿入



- insert の内部に insert が登場

```
(define (insert n alon)
  (cond
    [(empty? alon) (cons n empty)]
    [else (cond
      [(<= (first alon) n) (cons n alon)]
      [(> (first alon) n)
       (cons (first alon) (insert n (rest alon)))]))]))
```

- insert の実行が繰り返される

例 : (insert 40 (list 80 21 10 7 5 4))

= (cons 80 (insert 40 (list 21 10 7 5 4)))

(insert 40 (list 80 21 10 7 5 4)) から
(list 80 40 21 10 7 5 4) が得られる過程の概略

(insert 40 (list 80 21 10 7 5 4))

= ...

= (cons 80 (insert 40 (rest (list 80 21 10 7 5 4))))

= (cons 80 (insert 40 (list 21 10 7 5 4)))

= ...

= (cons 80 (cons 40 (list 21 10 7 5 4)))

= (list 80 40 21 10 7 5 4)



(insert 40 (list 80 21 10 7 5 4)) から
(list 80 40 21 10 7 5 4) が得られる過程の概略

(insert 40 (list 80 21 10 7 5 4))

= ...

= (cons 80 (insert 40 (rest (list 80 21 10 7 5 4))))

= (cons 80 (insert 40 (list 21 10 7 5 4)))

これは,

```
(define (insert n alon)
```

```
  (cond
```

```
    [(empty? alon) (cons n empty)]
```

```
    [else (cond
```

```
      [(<= (first alon) n) (cons n alon)]
```

```
      [(> (first alon) n)
```

```
        (cons (first alon) (insert n (rest alon)))]])])])
```

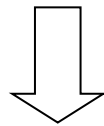
の alon を (list 80 21 10 7 5 4) で, n を 40 で置き換えたもの 22

例題 2. インサージョンソート



- 例題 1 の `insert` を使って, リストをソートする関数 `sort` を作り実行する
 - ここでは, 大きい順にソートする

(list 1 3 5 7 10 21 4 80)



(list 80 21 10 7 5 4 3 1)

「例題 2. インサージョンソート」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

```
;; sort: list-of-numbers -> list-of-numbers  
(define (sort alon)  
  (cond  
    [(empty? alon) empty]  
    [else (insert (first alon) (sort (rest alon)))]))
```

```
(define (insert n alon)  
  (cond  
    [(empty? alon) (cons n empty)]  
    [else (cond  
      [(<= (first alon) n) (cons n alon)]  
      [(> (first alon) n)  
       (cons (first alon) (insert n (rest alon)))]))])
```

例題 1
と同じ

2. その後、次を「実行用ウィンドウ」で実行しなさい

```
(sort (list 3 5 1 4))
```


実行結果の例



The screenshot shows the DrScheme IDE with a Scheme program for sorting a list. The code defines a `sort` function that uses a helper `insert` function. The `insert` function places a number `n` into a sorted list `alon` at the correct position. The `sort` function recursively sorts the rest of the list and inserts the first element.

```
(define (sort alon)
  (cond
    [(empty? alon) empty]
    [(cons? alon)
     (insert (first alon) (sort (rest alon)))]))
;; insert: number list-of-numbers(sorted) -> list-of-numbers(sorted)
(define (insert n alon)
  (cond
    [(empty? alon) (cons n empty)]
    [else (cond
             [(<= (first alon) n) (cons n alon)]
             [(> (first alon) n)
              (cons (first alon) (insert n (rest alon)))]))])
```

The execution results show the function being called with three different lists, and the output is the sorted list for each:

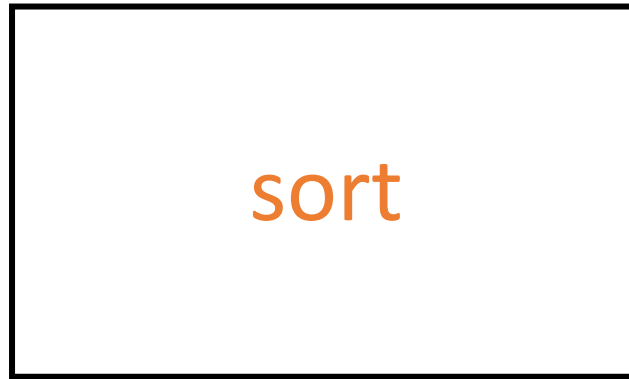
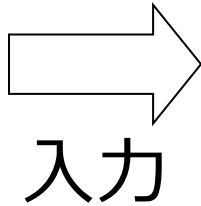
```
> (sort (list 3 5 1 4))
(list 5 4 3 1)
> (sort (list 4 3 2 1))
(list 4 3 2 1)
> (sort (list 5 4 3 5))
(list 5 5 4 3)
```

The text "実行結果" (Execution Results) is overlaid in green on the bottom right of the IDE window.

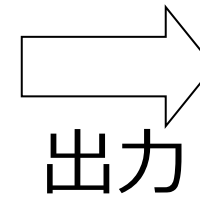
入力と出力



(list 3 5 1 4)



(list 5 4 3 1)



入力はリスト

出力はソート済みの
リスト



;; sort: list-of-numbers -> list-of-numbers

```
(define (sort alon)
```

```
  (cond
```

```
    [(empty? alon) empty]
```

```
    [else (insert (first alon) (sort (rest alon)))]))
```

;; insert: number list-of-numbers(sorted) -> list-of-numbers(sorted)

```
(define (insert n alon)
```

```
  (cond
```

```
    [(empty? alon) (cons n empty)]
```

```
    [else (cond
```

```
      [(<= (first alon) n) (cons n alon)]
```

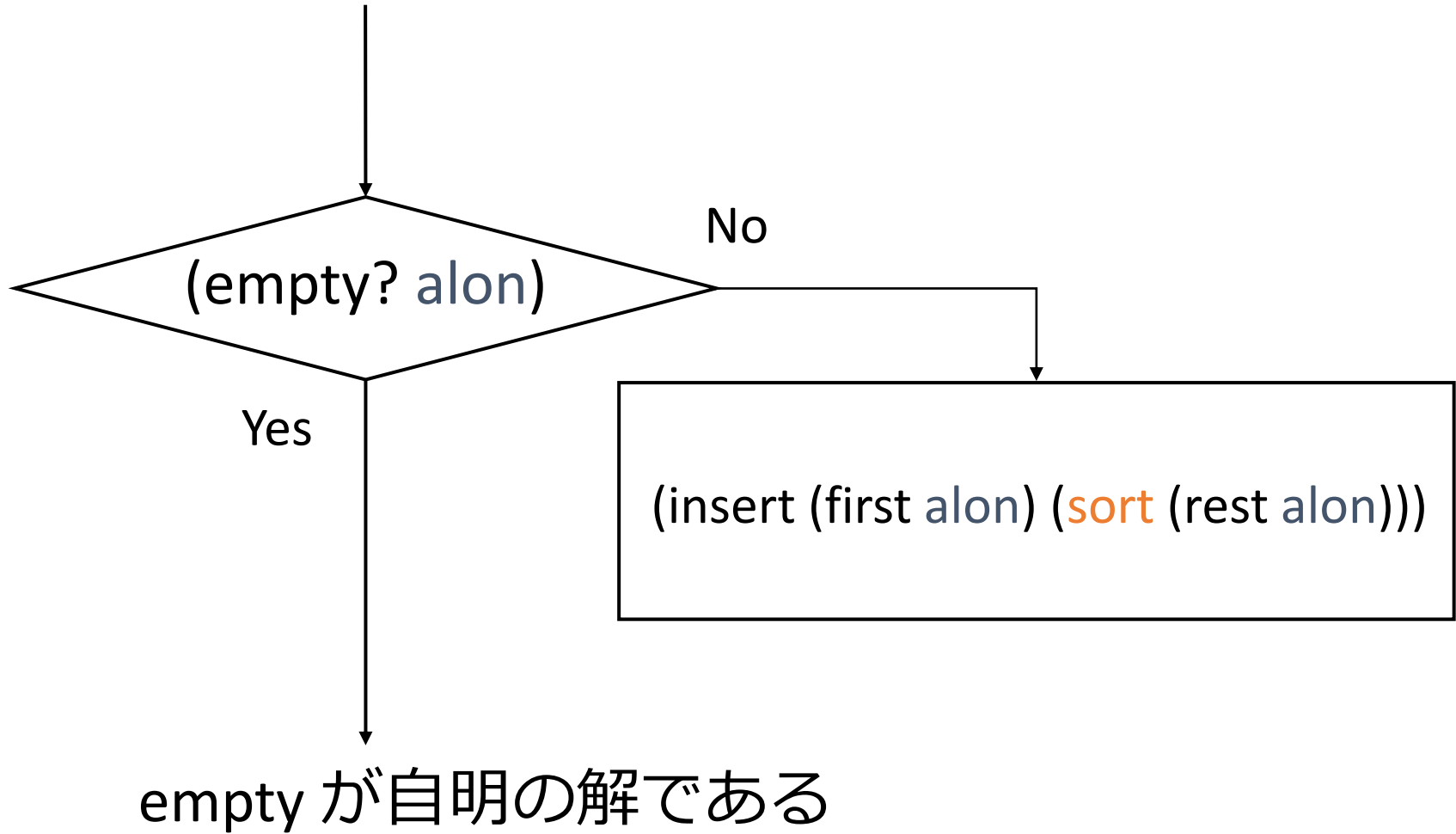
```
      [(> (first alon) n)
```

```
        (cons (first alon) (insert n (rest alon)))]))])
```

1. リストが空ならば： → 終了条件
empty → 自明な解

2. そうで無ければ：

「リストの rest をソートしたリスト
に対して、その先頭要素を挿入した
もの」が求める解



インサージョンソート



- sort の内部に sort が登場

```
(define (sort alon)
  (cond
    [(empty? alon) empty]
    [else (insert (first alon) (sort (rest alon)))]))
```

- sort の実行が繰り返される

例 : (sort (list 3 5 1 4))
= (insert 3 (sort (list 5 1 4)))



(sort (list 3 5 1 4)) から (list 5 4 3 1)) が得られる過程の概略 (1/2)

(sort (list 3 5 1 4))

= ...

= (insert 3 (sort (rest (list 3 5 1 4))))

= (insert 3 (sort (list 5 1 4)))

= ...

= (insert 3 (insert 5 (sort (rest (list 5 1 4)))))

= (insert 3 (insert 5 (sort (list 1 4))))

= ...

= (insert 3 (insert 5 (insert 1 (sort (rest (list 1 4))))))

= (insert 3 (insert 5 (insert 1 (sort (list 4)))))

= ...

= (insert 3 (insert 5 (insert 1 (insert 4 (sort (rest (list 4)))))))

= (insert 3 (insert 5 (insert 1 (insert 4 (sort empty)))))

= ...

= (insert 3 (insert 5 (insert 1 (insert 4 empty)))) [次ページへ](#) 31

(sort (list 3 5 1 4)) から (list 5 4 3 1)) が得られる過程の概略 (1/2)

```
(sort (list 3 5 1 4))  
= ...  
= (insert 3 (sort (rest (list 3 5 1 4))))  
= (insert 3 (sort (list 5 1 4)))
```

これは、

```
(define (sort alon)  
  (cond  
    [(empty? alon) empty]  
    [else (insert (first alon) (sort (rest alon)))]))
```

の alon を (list 3 5 1 4) で置き換えたもの

```
= (insert 3 (insert 5 (insert 1 (insert 4 (sort (rest (list 4)))))))  
= (insert 3 (insert 5 (insert 1 (insert 4 (sort empty))))))  
= ...  
= (insert 3 (insert 5 (insert 1 (insert 4 empty))))
```


(sort (list 3 5 1 4)) から (list 5 4 3 1) が得られる過程の概略 (2/2)

= (insert 3 (insert 5 (insert 1 (insert 4 empty))))
= ...
= (insert 3 (insert 5 (cons 4 (insert 1 (rest (list 4))))))
= (insert 3 (insert 5 (cons 4 (insert 1 empty))))
= ...
= (insert 3 (insert 5 (cons 4 (cons 1 empty))))
= ...
= (insert 3 (cons 5 (list 4 1)))
= ...
= (cons 5 (insert 3 (list 4 1)))
= ...
= (cons 5 (cons 4 (insert 3 (list 1))))
= ...
= (cons 5 (cons 4 (cons 3 (list 1))))
= (list 5 4 3 1)

ここまでのまとめ



- リストを出力とするような関数
 - 要素の挿入
 - インサージョンソート

どれも `cons` を使用.

例題 3. インサージョンソートでの繰り返し回数

- インサージョンソートについて, ステップ実行を行い, 繰り返し回数を数えてみる
 - sort の実行回数はいくらか
 - insert の実行回数はいくらか



- インサージョンソートでの sort 関数の実行回数

リストの要素数を n とすると

$n+1$ 回

例 $n=3$ のとき)

(sort (list 80 30 50))

= (insert 80 (sort (list 30 50)))

= (insert 80 (insert 30 (sort (list 50))))

= (insert 80 (insert 30 (insert 50 (sort empty))))

• インサージョンソートでの insert 関数の実行回数

リストの要素数を n とすると
最小 n 回, 最大 $n(n+1)/2$ 回

例 $n=3$ のとき)

sort □ □ □
→ insert □ sort □ □ 1,2,3回
→ insert □ sort □ 1,2回
→ insert □ sort 1回

insert では, 内部で insert が再帰的に呼び出される

- 1 つめの insert では, 0, 1, または 2回
 - 2 つめの insert では, 0, または 1回
 - 3 つめの insert では, 0回
- } 何回になるかは
データの並びで変わる

• インサクションソートでの insert 関数の実行回数 (平均)

リストの要素数を n とすると

$$1 + 1.5 + 2 + \dots + (n+1)/2 \text{ 回} = n^2/4 + 3n/4$$

例 $n=3$ のとき)

sort □ □ □

→ insert □ sort □ □ 1,2,3回

→ insert □ sort □ 1,2回

→ insert □ sort 1回

insert では、内部で insert が再帰的に呼び出される

- 1 つめの insert では、0, 1, または 2回 : 平均 1 回
- 2 つめの insert では、0, または 1回 : 平均 0.5 回
- 3 つめの insert では、0 回 : 平均 0 回

• インサクションソートでの sort 関数の実行回数

リストの要素数を n とすると

$n+1$ 回

例 $n=3$ のとき)

(sort (list 80 30 50))

= (insert 80 (sort (list 30 50)))

= (insert 80 (insert 30 (sort (list 50))))

= (insert 80 (insert 30 (insert 50 (sort empty))))

sort の実行回数 (平均)



$n \rightarrow \infty$ では

$$n^2/4 + 3n/4 \rightarrow n^2/4$$

- 計算に要する時間は, n^2 に比例する
- $3n/4$ の項は無視できる
- $n^2/4$ のうち「/4」の部分が意味があるのは
 - insert の実行時間が, 実際に何秒であるかが分かる場合に限る

3n/4 の項は無視できる



n	$n^2/4$	$3n/4$
1	0.25	0.75
2	1	1.5
5	6.25	3.75
10	25	7.5
100	2500	75
1000	250000	750
10000	25000000	7500

例題 4 . append



- リストをつなげる関数 **append** を使ってみる
 - append は Scheme が備えている関数

「例題 4 . append」の手順



1. 次の式を「実行用ウィンドウ」で，実行しな

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

```
(append (list 1 2) (list 3 4) (list 5 6))
```

```
(append (list 1 2) 3 (list 4 5))
```

☆ 次は，例題 5 に進んでください



2つのリストを併合

3つのリストを併合

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

```
> (append (list 1 2) (list 3 4) (list 5 6))  
(list 1 2 3 4 5 6)
```

```
> (append (list 1 2) 3 (list 4 5))  
append: expects argument of type <proper list>; given 3
```

リストでないものは併合できない

例題 5. 大きな要素の選択



- 数値のリスト `alon` と, 数 `threshold` を入力として, `alon` から `threshold` 以上の数を選んで, リストを出力する関数 `larger-items` を作り, 実行する
 - リストの要素を1つずつ調べる

「例題 5. 大きな要素の選択」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 (larger-items alon threshold)
  (cond
    [(empty? alon) empty]
    [else
     (cond
       [(>= (first alon) threshold)
        (cons (first alon)
              (larger-items (rest alon) threshold))]
       [else (larger-items (rest alon) threshold)]))]))
```

2. その後、次を「実行用ウィンドウ」で実行しなさい

```
(larger-items (list 1 2 3 10 11 12 4 5 6) 6)
```

Untitled - DrScheme

File Edit Windows Show Language Scheme Help

Untitled Save

(define ...) Check Syntax Step Execute Break

```
(define (larger-items alon threshold)
  (cond
    [(empty? alon) empty]
    [else
     (cond
       [(>= (first alon) threshold)
        (cons (first alon)
              (larger-items (rest alon) threshold))]
       [else (larger-items (rest alon)
                             threshold)])])])
```

Welcome to [DrScheme](#), version 103p1.
Language: **Intermediate Student**

```
> (larger-items (list 1 2 3 10 11 12 4 5 6) 6)
(list 10 11 12 6)
```

実行結果

5:3 not running

larger-iterms の入力と出力

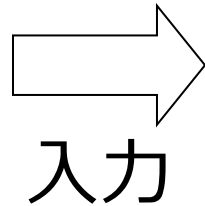


alon の値 :

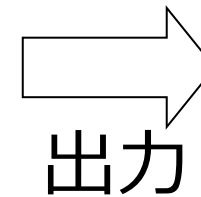
(list 1 2 3 10 11 12 4 5 6)

threshold の値:

6



(list 10 11 12 6)



入力はリストと数値

出力はリスト

;;larger-items: list-of-numbers number -> list-of-numbers

;; alon から threshold 以上の数を選びリストを作る

```
(define (larger-items alon threshold)
  (cond
    [(empty? alon) empty]
    [else
     (cond
       [(>= (first alon) threshold)
        (cons (first alon)
              (larger-items (rest alon) threshold))]
       [else (larger-items (rest alon) threshold)])])])
```

1. リストが空ならば : → 終了条件

empty → 自明な解

2. そうで無ければ :

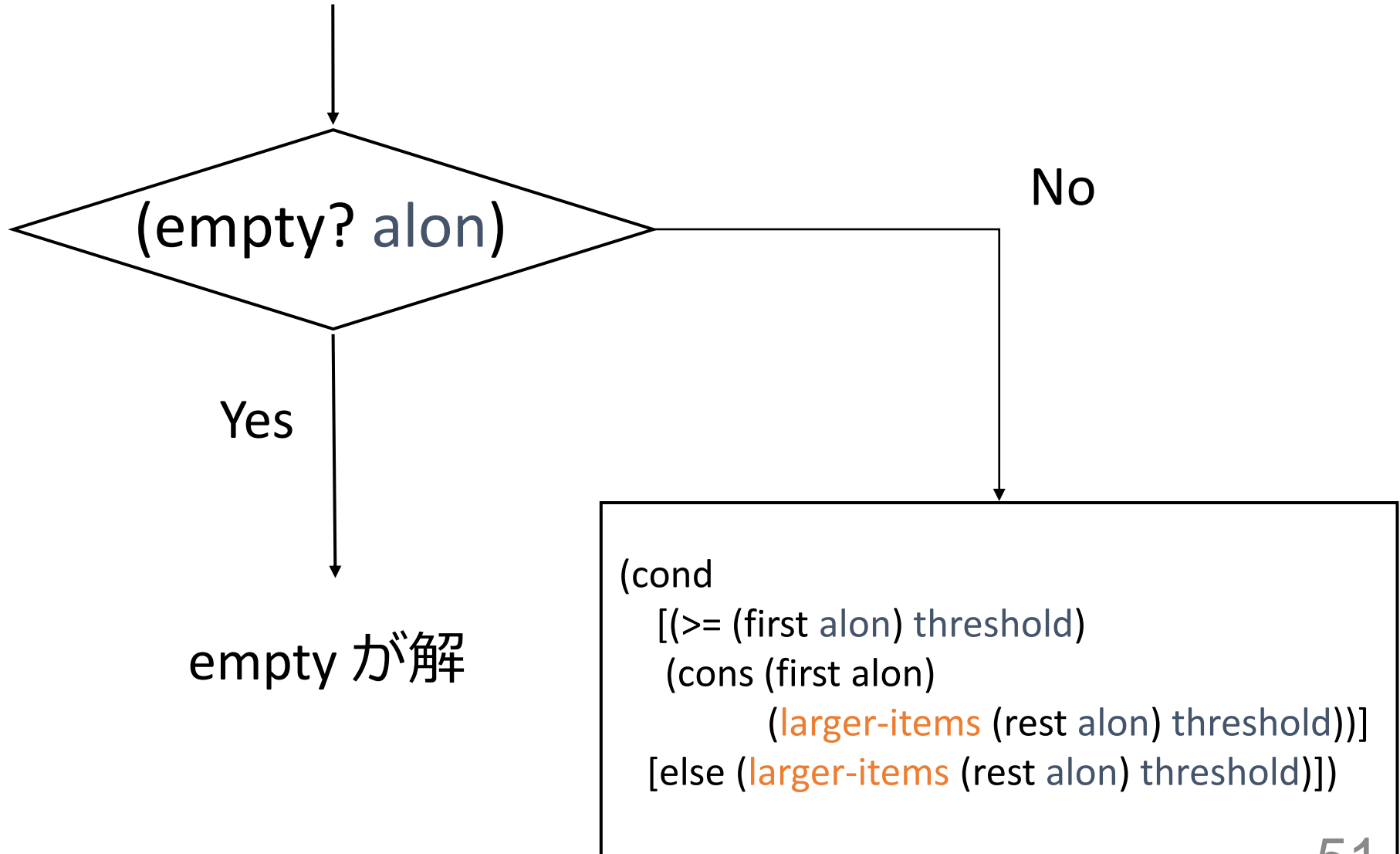
a. リストの $\text{first} \geq \text{threshold}$

「リストの rest に対する **larger-items** の結果 (リスト) の先頭に, リストの first (数値) をつなげたもの」 が求める解

b. リストの $\text{first} < \text{threshold}$

「リストの rest に対する **larger-items** の結果」 が求める解

繰り返し処理



繰り返し処理



- `larger-items` の内部に `larger-items` が登場

終了
条件

```
(define (larger-items alon threshold)
  (cond
    [(empty? alon) empty]
    [else
     (cond
       [(>= (first alon) threshold)
        (cons (first alon)
              (larger-items (rest alon) threshold))]
       [else (larger-items (rest alon) threshold)]))]))
```

- `larger-items` の実行が繰り返される

例 : `(larger-items (list 6 4 2) 3)`
= `(cons 6 (larger-items (list 4 2) 3))`

(larger-items (list 6 2 4) 3) から (list 6 4) が得られる過程の概略

(larger-items (list 6 2 4) 3)

= ...

= (cons 6 (larger-items (list 2 4) 3))

= ...

= (cons 6 (larger-items (list 4) 3))

= ...

= (cons 6 (cons 4 (larger-items empty 3)))

= ...

= (cons 6 (cons 4 empty))

= (list 6 4)

(larger-items (list 6 2 4) 3) から (list 6 4) が得られる過程の概略

(larger-items (list 6 2 4) 3)

= ...

= (cons 6 (larger-items (list 2 4) 3))

= ...

これは,

```
(define (larger-items alon threshold)
```

```
(cond
```

```
  [(empty? alon) empty]
```

```
  [else
```

```
    (cond
```

```
      [(>= (first alon) threshold)
```

```
        (cons (first alon)
```

```
              (larger-items (rest alon) threshold))]]
```

```
      [else (larger-items (rest alon) threshold)]])])])])
```

の alon を (list 6 2 4) で, threshold を 3 で置き換えたもの

例題 6 . 小さな要素の選択



- 数値のリスト `alon` と, 数 `threshold` を入力として, `alon` から `threshold` より小さな数を選んで, リストを出力するプログラム `smaller-items` を作り, 実行する
 - リストの要素を1つずつ調べる

「例題 6 . 小さな要素の選択」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 (smaller-items alon threshold)
  (cond
    [(empty? alon) empty]
    [else
     (cond
       [(< (first alon) threshold)
        (cons (first alon)
              (smaller-items (rest alon) threshold))]
       [else (smaller-items (rest alon) threshold)]))]))
```

2. その後、次を「実行用ウィンドウ」で実行しなさい

```
(smaller-items (list 1 2 3 10 11 12 4 5 6) 6)
```


実行結果の例



The screenshot shows the DrScheme IDE window titled "Untitled - DrScheme". The menu bar includes "File", "Edit", "Show", "Language", "Scheme", "Windows", and "Help". The toolbar contains buttons for "Save", "Check Syntax", "Execute", and "Break".

The code editor contains the following Scheme code:

```
[(< (first alon) threshold)
  (cons (first alon) (smaller-items (rest
alon) threshold)))]
[else (smaller-items (rest alon) threshold))]]))
```

The interactive console shows the following input and output:

```
> (smaller-items (list 1 2 3 10 11 12 4 5 6) 6)
(list 1 2 3 4 5)
> (smaller-items (list 1 2 3 10 11 12 4 5 6) 2)
(list 1)
>
```

The status bar at the bottom indicates the cursor is at line 8:3, the file is Read/Write, and the program is not running.



;; smaller-items: list-of-numbers number -> list-of-numbers

;; alon から threshold より小さな数を選びリストを作る

```
(define (smaller-items alon threshold)
```

```
  (cond
```

```
    [(empty? alon) empty]
```

```
    [else
```

```
      (cond
```

```
        [(< (first alon) threshold)
```

```
          (cons (first alon)
```

```
                (smaller-items (rest alon) threshold)))]
```

```
        [else (smaller-items (rest alon) threshold)]])])])
```

例題 7. クイックソート



- 数値のリスト `alon` をソートするための、クイックソートのプログラム `quick-sort` を作り、実行する
 - 例題 5 の関数 `larger-items` と、例題 6 の関数 `smaller-items` を使う
 - クイックソートの `pivot` (基準値) としては、リストの先頭要素を使う
 - 2つのリストと `pivot` をつなげて、全体として1つのリストを作るために `append` を使う

「例題7. クイックソート」の手順 (1/2)

1. 次に「定義用ウィンドウ」で、実行し

なさい

```
(define (larger-items alon threshold)
  (cond
    [(empty? alon) empty]
    [else
     (cond
       [(>= (first alon) threshold)
        (cons (first alon)
              (larger-items (rest alon) threshold))]
       [else (larger-items (rest alon) threshold)]))])
```

例題4
と同じ

```
(define (smaller-items alon threshold)
  (cond
    [(empty? alon) empty]
    [else
     (cond
       [(< (first alon) threshold)
        (cons (first alon)
              (smaller-items (rest alon) threshold))]
       [else (smaller-items (rest alon) threshold)]))])
```

例題5
と同じ

```
;; quick-sort : list-of-numbers -> list-of-numbers
(define (quick-sort alon)
  (cond
    [(empty? alon) empty]
    [else
     (append
      (quick-sort (smaller-items (rest alon) (first alon)))
      (list (first alon))
      (quick-sort (larger-items (rest alon) (first alon))))]))
```

「例題7. クイックソート」の手順 (2/2)



2. その後, 次を「実行用ウィンドウ」で実行しなさい

```
(quick-sort (list 4 6 2))
```

```
(quick-sort (list 8 10 6 3 5))
```

☆ 次は, 課題に進んでください



Untitled - DrScheme

File Edit Windows Show Language Scheme Help

Untitled Save

Check Syntax Step Execute Break

```
(define (quick-sort alon)
  (cond
    [(empty? alon) empty]
    [else
     (append
      (quick-sort (smaller-items (rest alon) (first alon)))
      (list (first alon))
      (quick-sort (larger-items (rest alon) (first alon))))]))
```

Welcome to [DrScheme](#), version 103p1.
Language: **Intermediate Student**.

```
> (quick-sort (list 4 6 2))
(list 2 4 6)
> (quick-sort (list 8 10 6 3 5))
(list 3 5 6 8 10)
>
```

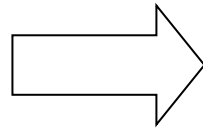
7:3 Unlocked not running

quick-sort の入力と出力

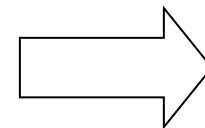


alon の値 :

(list 4 6 2)



入力



出力

(list 2 4 6)

入力はリスト

出力はリスト

クイックソートのプログラム



```
:: quick-sort : list-of-numbers -> list-of-numbers
```

```
(define (quick-sort alon)
```

```
  (cond
```

```
    [(empty? alon) empty]
```

```
    [else
```

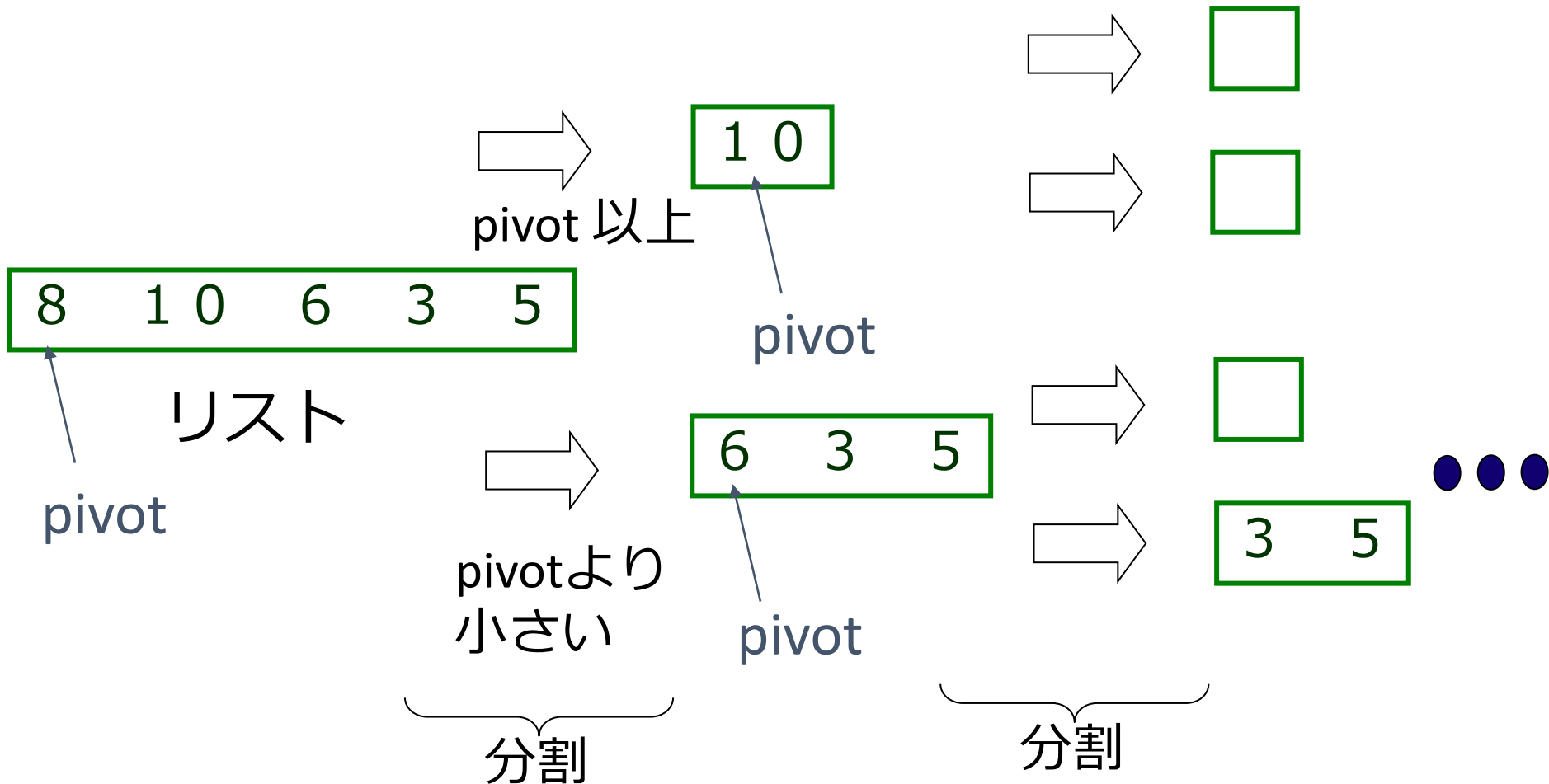
```
      (append
```

```
        (quick-sort (smaller-items (rest alon) (first alon)))
```

```
        (list (first alon))
```

```
        (quick-sort (larger-items (rest alon) (first alon))))]))
```


クイックソートの考え方



リストが空になるまで, pivot の選択と, pivot による要素の分割を続ける

「クイックソートのプログラム」 の理解のポイント



- 繰り返しの終了条件
 - 入力が空 (empty) である
- 入力は1つ
 - ソートすべきリスト: `alon`

クイックソートの繰り返し処理



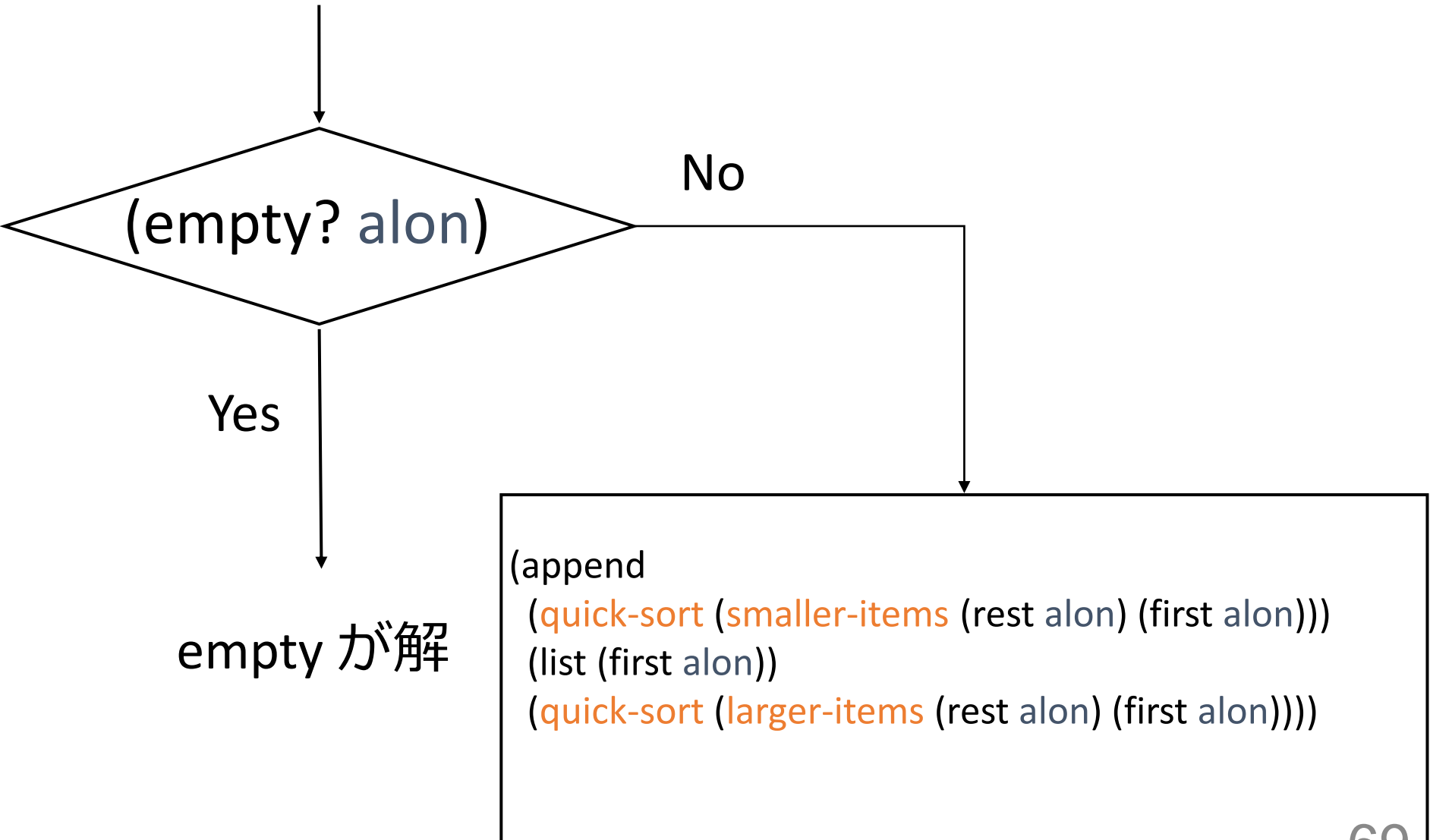
1. `empty` ならば : → 終了条件
 `empty` → 自明な解
2. そうで無ければ :
 - リストの分割を行う
 - 結局, 「リストが空になる」まで繰り返す

クイックソートの終了条件



- pivot による要素の分割で、**smaller-items**, **larger-items** は、入力（リスト）よりも小さなリストを生成する
- **smaller-items**, **larger-items** 内で呼び出される **quick-sort** は、元の入力よりも小さなリストを扱う
- 最終的に、**quick-sort** は空リスト（empty）を受け取る

繰り返し処理



繰り返し処理



- quick-sort の内部に quick-sort が登場

```
(define (quick-sort alon)
```

```
(cond
```

終了
条件

```
[(empty? alon) empty]
```

自明な解

```
[else
```

```
(append
```

```
(quick-sort (smaller-items (rest alon) (first alon)))
```

```
(list (first alon))
```

```
(quick-sort (larger-items (rest alon) (first alon))))))
```

- quick-sort の実行が繰り返される

(quick-sort (list 6 2 4)) からの過程



(quick-sort (list 6 2 4))

= ...

= (append

```
(quick-sort (smaller-items (rest (list 6 2 4)) (first (list 6 2 4))))  
(list (first (list 6 2 4)))  
(quick-sort (larger-items (rest (list 6 2 4)) (first (list 6 2 4)))))]])
```

要するに、3つのリスト

(quick-sort (smaller-items (list 2 4) 6) → 6 以上

(list 6) → (list 6)

(quick-sort (larger-items (list 2 4) 6) → 6 未満

に分割されている

部分問題の例



- 休暇旅行の旅程（家から旅先のホテルまで）
 - 家→空港
 - 空港→旅先の空港
 - 旅先の空港→ホテル

クイックソートの部分問題



1. 「基準値より大きい要素」のソート
2. 「基準値より小さい要素」のソート

分割統治法 (divide and conquer)



- サイズ N の問題を解くのに、サイズが約 $N/2$ の部分問題2つに分けて、それぞれを再帰的に解き、その後でその2つの解を合わせて目的の解を得る

例題 8 . クイックソート



- AddressNote 構造体のリストをソートするための、クイックソートのプログラム **quick-sort** を作り、実行する
 - 名前の順でソートする
 - クイックソートの pivot (基準値) としては、リストの先頭要素を使う
 - 2つのリストと pivot をつなげて、全体として1つのリストを作るために **append** を使う



;;larger-items: list of address-note, number -> listof numbers

;; a-list から threshold 以上の数を選びリストを作る

```
(define (larger-items a-list threshold)
```

```
  (cond
```

```
    [(empty? a-list) empty]
```

```
    [else
```

```
      (cond
```

```
        [(string>=? (address-record-name (first a-list))
```

```
threshold)
```

```
          (cons (first a-list)
```

```
                (larger-items (rest a-list) threshold)))]
```

```
        [else (larger-items (rest a-list) threshold)]))])
```

;; smaller-items: list of data, number -> list of numbers

;; a-list から threshold より小さな数を選びリストを作る

```
(define (smaller-items a-list threshold)
```

```
  (cond
```

```
    [(empty? a-list) empty]
```

```
    [else
```

```
      (cond
```

```
        [(string<? (address-record-name (first a-list))
```

```
threshold)
```

```
          (cons (first a-list)
```

```
                (smaller-items (rest a-list) threshold)))]
```

```
        [else (smaller-items (rest a-list) threshold))]))])
```

クイックソートのプログラム



```
:: quick-sort : list of data -> list of numbers
```

```
(define (quick-sort a-list)
```

```
  (cond
```

```
    [(empty? a-list) empty]
```

```
    [else
```

```
      (append
```

```
        (quick-sort (smaller-items (rest a-list)
```

```
          (address-record-name (first a-list))))
```

```
        (list (first a-list))
```

```
        (quick-sort (larger-items (rest a-list)
```

```
          (address-record-name (first a-
```

```
list)))))))]))
```

(quick-sort book) からの過程の概略



(quick-sort book)

= (quick-sort (list

(make-address-record "Mike" 10 "Fukuoka")

(make-address-record "Bill" 20 "Saga")

(make-address-record "Ken" 30 "Nagasaki"))))

= ...

= (append

(quick-sort (smaller-items

(list

(make-address-record "Bill" 20 "Saga")

(make-address-record "Ken" 30 "Nagasaki"))

"Mike"))

(list (make-address-record "Mike" 10 "Fukuoka"))

(quick-sort (larger-items

(list

(make-address-record "Bill" 20 "Saga")

(make-address-record "Ken" 30 "Nagasaki"))

"Mike"))

15-3 課題

課題 1



- 関数 **quick-sort** (授業の例題 5) についての問題
 - (quick-sort (list 8 10 6 3 5)) から (list 3 5 6 8 10) が得られる過程の概略を数行程度で説明しなさい

```
;; quick-sort : list-of-numbers -> list-of-numbers
(define (quick-sort alon)
  (cond
    [(empty? alon) empty]
    [else
     (append
      (quick-sort (smaller-items (rest alon) (first alon)))
      (list (first alon))
      (quick-sort (larger-items (rest alon) (first alon))))]))
```

課題 2 . 住所録構造体のクイックソート



- 次ページ以降にある「住所録構造体のクイックソート」についての問題
 - (**quick-sort** book) の実行結果を報告しなさい

住所録構造体のクイックソート (1/2)



```
(define-struct address-record
  (name age address))
(define book (list
  (make-address-record "Mike" 10 "Fukuoka")
  (make-address-record "Bill" 20 "Saga")
  (make-address-record "Ken" 30 "Nagasaki")))
(define (larger-items a-list threshold)
  (cond
    [(empty? a-list) empty]
    [else
     (cond
       [(string>=? (address-record-name (first a-list)) threshold)
        (cons (first a-list)
              (larger-items (rest a-list) threshold))]
       [else (larger-items (rest a-list) threshold)]))]))
```

住所録構造体のクイックソート (2/2)



```
(define (smaller-items a-list threshold)
  (cond
    [(empty? a-list) empty]
    [else
     (cond
       [(string<? (address-record-name (first a-list)) threshold)
        (cons (first a-list)
              (smaller-items (rest a-list) threshold))]
       [else (smaller-items (rest a-list) threshold)]))])

(define (quick-sort a-list)
  (cond
    [(empty? a-list) empty]
    [else
     (append
      (quick-sort (smaller-items (rest a-list)
                                (address-record-name (first a-list))))
      (list (first a-list))
      (quick-sort (larger-items (rest a-list)
                                (address-record-name (first a-list))))))])
```

- リストからの検索プログラムの作成.
 1. ある数 n が, 数のリスト `a-list` に存在するかどうかを検索する関数 `search` を作成し, 実行結果を報告しなさい
 - 存在するときに限り `true` を返す
 2. ある数 n が, 数のソート済みのリスト `a-list` に存在するかどうかを検索するプログラム `search-sorted` を作れ
 - 存在するときに限り `true` を返す
 - `search-sorted` はリストがソート済みという事実を利用しなくてはならない