sp-12. 再帰と繰り返しの回数

（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

12-1 繰り返し計算

12-2 パソコン演習

12-3 課題

本日の内容

- 再帰を使った，繰り返し計算
- 数学の「再帰的定義」と，Scheme プログラムの関係
- 繰り返し回数
  - 関数は「何回繰り返して」実行されるか

12-1 繰り返し計算

繰り返しの例

- 階乗
  - 「n - 1 の階乗」に n を足すことを繰り返す
- ユークリッドの互助法
  - m と n の最大公約数を求めるために，「割った余りを求めること」を，余りが 0 になるまで繰り返す

12- 2 パソコン演習

パソコン演習の進め方

- 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 の使用

- 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．　階乗

- 階乗を計算する関数！を作り，実行する
  - 次の方針でプログラムを作成する

    n > 0 のとき，n! = n × (n-1)!

例）　6! = 6×5!

階乗

- n = 0 のとき

$$n! = 1$$

- n > 0 のとき

$$n! = n \times (n-1)!$$

「例題１．階乗」の手順

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

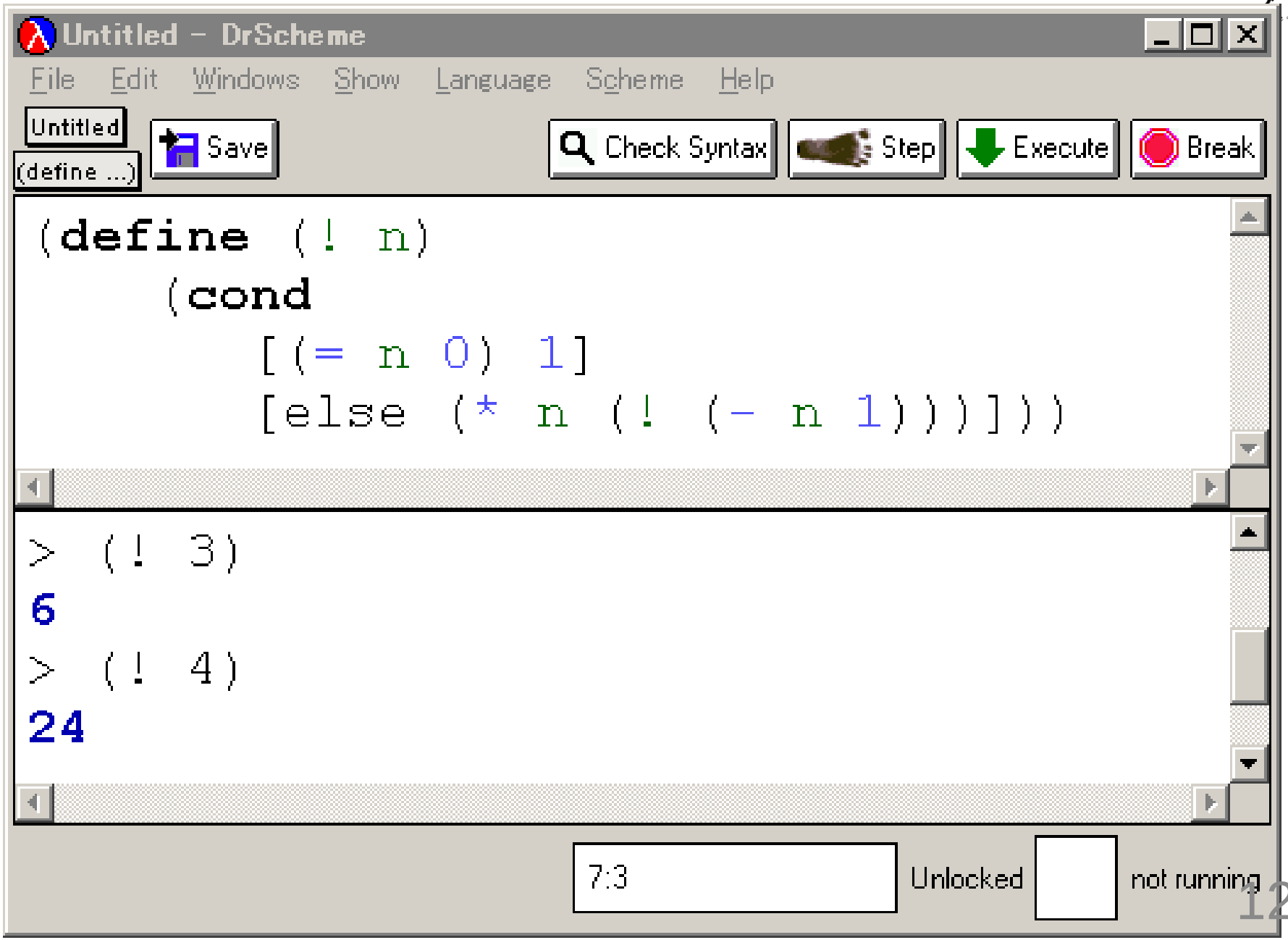
```
;;! : number -> number
;;to compute n*(n-1)*...*2*1
(define (! n)
   (cond
      [(= n 0) 1]
      [else (* n (! (- n 1)))]))
```

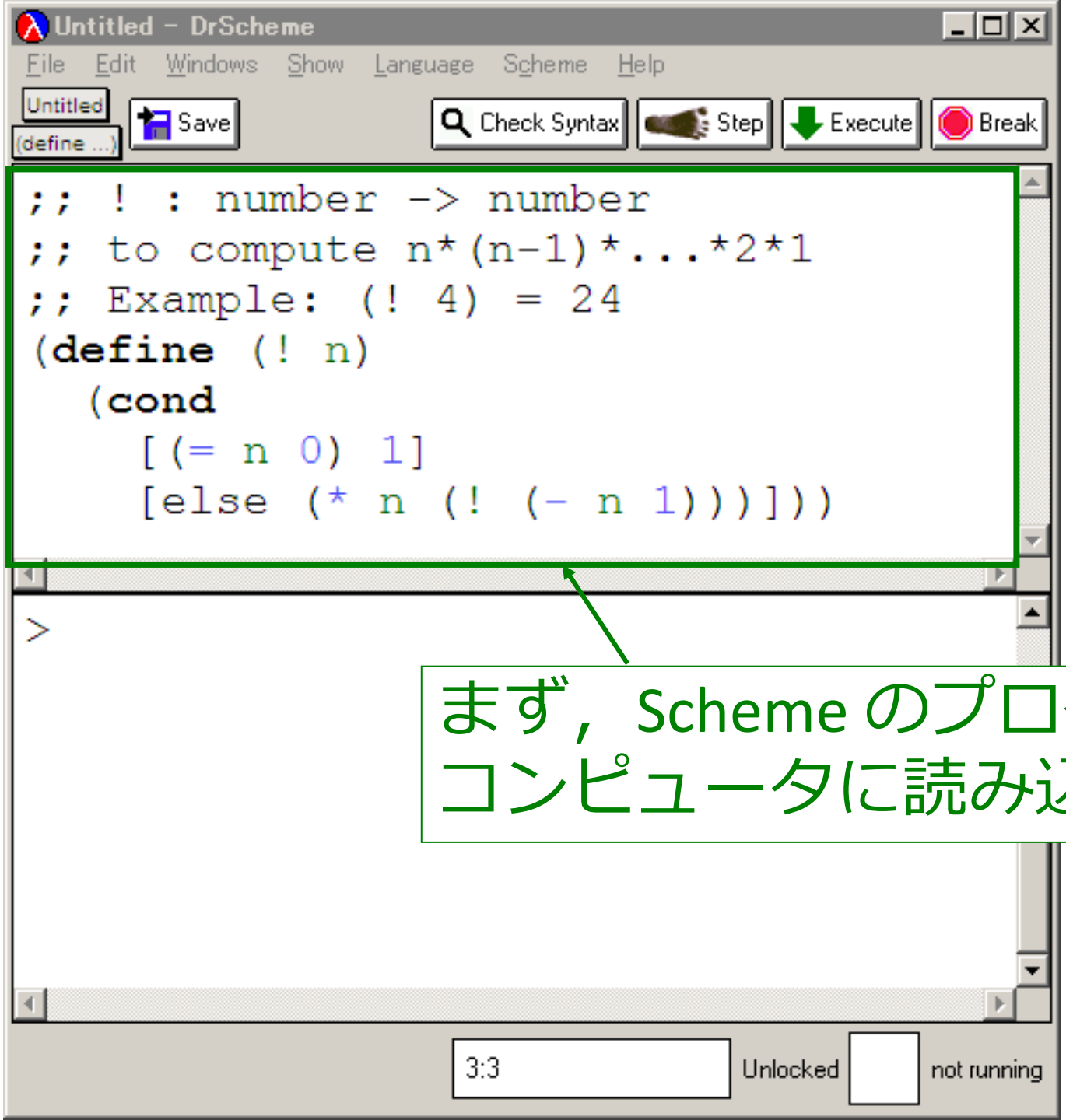
2. その後，次を「実行用ウインドウ」で実行しなさい

```
(! 3)
(! 4)
```

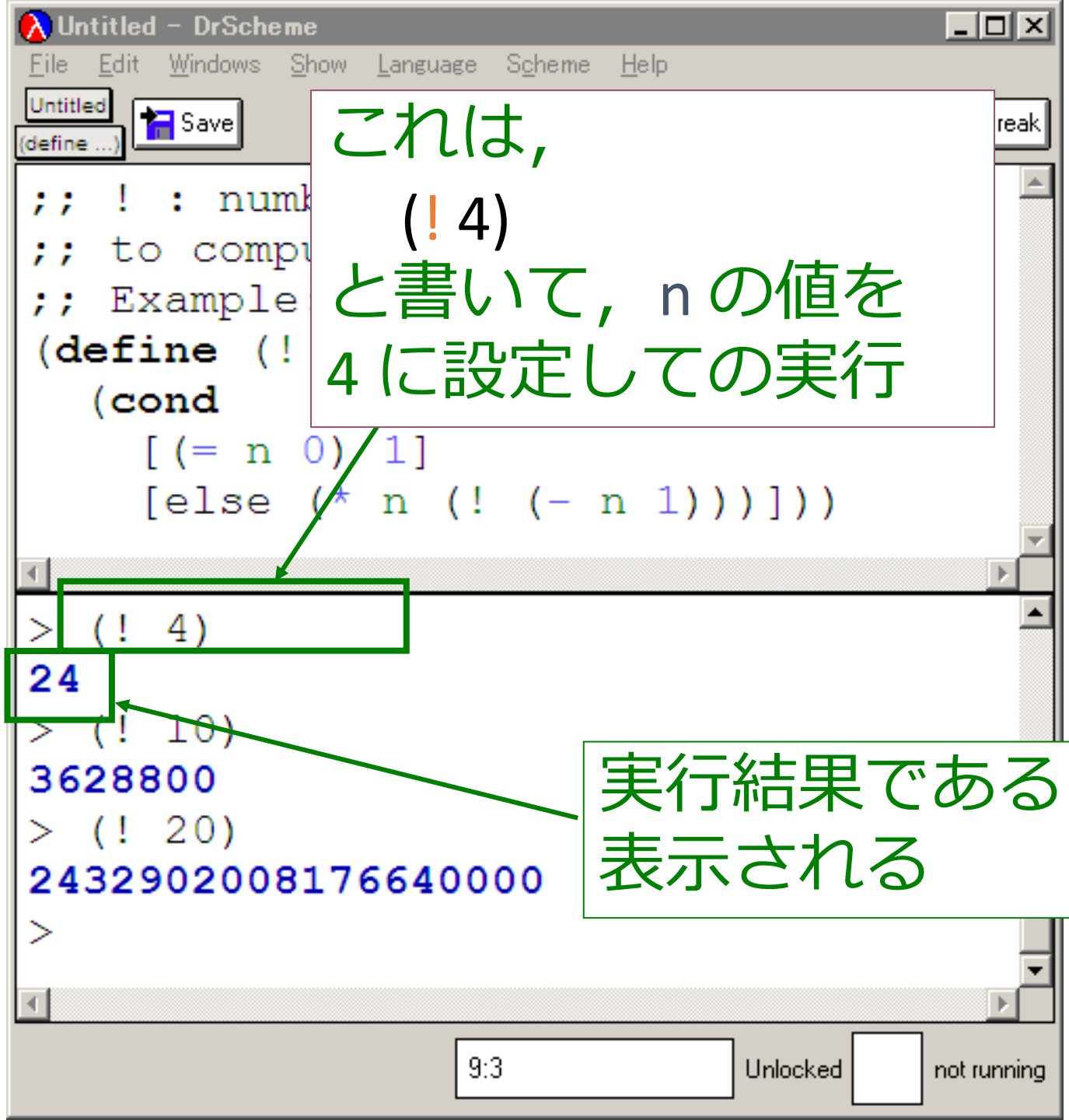
☆　次は，例題２に進んでください

「例題１．階乗」の実行結果


Untitled - DrScheme
File Edit Windows Show Language Scheme Help
Untitled
(define ...)
Save
Check Syntax
Step
Execute
Break
(define (! n)
(cond
[(= n 0) 1]
[else (* n (! (- n 1)))]))
> (! 3)
6
> (! 4)
24
7:3
Unlocked
not running



まず，Scheme のプログラムを
コンピュータに読み込ませている



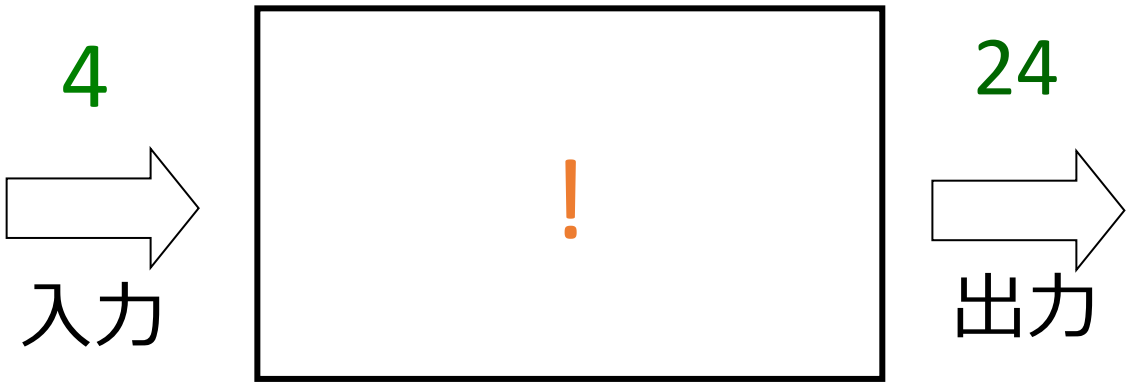
これは,

(! 4)

と書いて，n の値を
4 に設定しての実行

実行結果である「24」が
表示される

入力と出力



入力は数値

出力は数値

! 関数

```
;; ! : number -> number
;; to compute n*(n-1)*...*2*1
;; Example: (! 4) = 24
(define (! n)
    (cond
        [(= n 0) 1]
        [else (* n (! (- n 1)))]))
```

0! = 1 である

n! は (n-1)! を計算して，n を掛ける

階乗

1. n = 0 ならば：　→　終了条件

　　1　→　自明な解

2. そうで無ければ：

– 「n - 1 の階乗」に n をかける

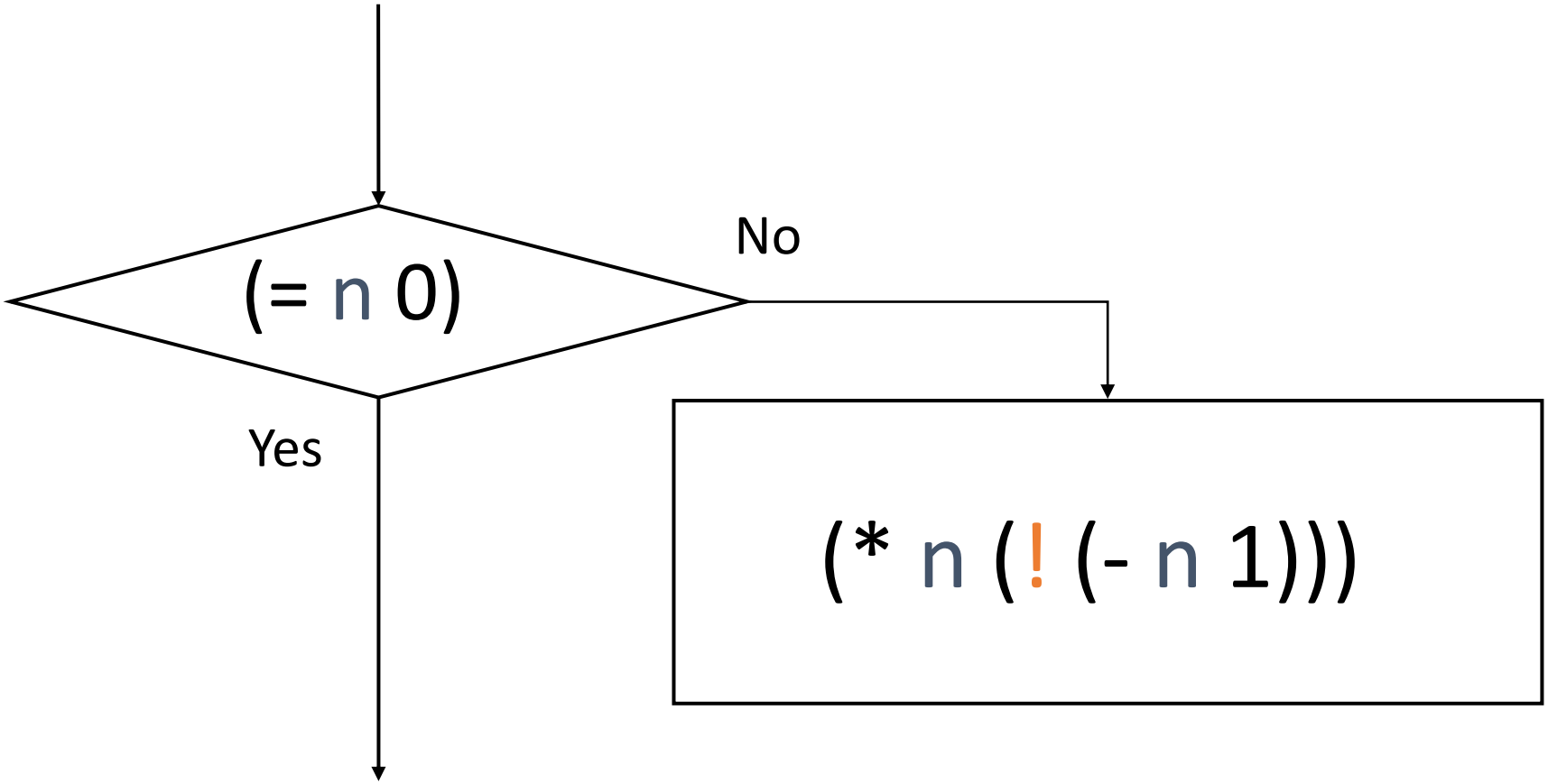
⇒ 結局，1 から n までのかけ算を繰り返す

階乗

```
;; ! : number -> number
;; to compute n*(n-1)*...*2*1
;; (! 4) = 24
(define (! n)
  (cond
    [(= n 0) 1]
    [else (* n (! (- n 1)))]))
```

終了条件　自明な解

終了条件



1 が自明の解

階乗

- ! の内部に ! が登場

```
(define (! n)
   (cond
      [(= n 0) 1]
      [else (* n (! (- n 1)))]))
```

- ! の実行が繰り返される

例： (! 5) = (* 5 (! 4))

(! 4) = (* 4 (! 3))

例題２．ステップ実行

- 関数 ! （例題１）について，実行結果に至る過程を見る
  - (! 3) から 6 に至る過程を見る
  - DrScheme の stepper を使用する

```
(define (! n)
   (cond
      [(= n 0) 1]
      [else (* n (! (- n 1)))]))
```

```
(! 3)
=    ...
=    (* 3 (! 2))
=    ...
=    (* 3 (* 2 (! 1)))
=    ...
=    (* 3 (* 2 (* 1 (! 0))))
=    ...
=    (* 3 (* 2 (* 1 1)))
=    (* 3 (* 2 1))
=    (* 3 2)
=    6
```

「例題２．ステップ実行」の手順

1. 次を「定義用ウインドウ」で，実行しなさい
   - Intermediate Student で実行すること
   - 入力した後に，Execute ボタンを押す

```
;;! : number -> number
;;to compute n*(n-1)*...*2*1
(define (! n)
   (cond
      [(= n 0) 1]
      [else (* n (! (- n 1)))]))
(! 3)
```

← 例題１と同じ

2. DrScheme を使って，ステップ実行の様子を確認しなさい　（Step ボタン，Next ボタンを使用）
   - 理解しながら進むこと

☆　次は，例題３に進んでください

Stepper

File Edit Windows Help

Home << Previous Next >>

```
(define (! n)
  (cond
   ((= n 0) 1)
   (else (* n (! (- n 1))))))
```

```
(! 3)
```

→

```
(cond
 ((= 3 0) 1)
 (else
  (*
   3
    (! (- 3 1)))))
```

!の「n」は「3」で置き換わる

Stepper

File Edit Windows Help

Home << Previous Next >>

```
(define (! n)
  (cond
   ((= n 0) 1)
   (else (* n (! (- n 1))))))
```

```
(cond
 ((= 3 0) 1)
 (else
  (*
   3
   (! (- 3 1)))))
```

```
(cond
 (false 1)
 (else
  (*
   3
   (! (- 3 1)))))
```

「(= 3 0)」は「false」で置き換わる

Stepper

File Edit Windows Help

Home << Previous Next >>

```
(define (! n)
  (cond
   ((= n 0) 1)
   (else (* n (! (- n 1))))))
```

```
(cond
 (false 1)
 (else
  (*
   3
   (! (- 3 1)))))
```

→

```
(* 3 (! (- 3 1)))
```

「(cond [false 式X] [else 式Y])」は
「式Y」で置き換わる

Stepper

File Edit Windows Help

Home << Previous Next >>

```
(define (! n)
  (cond
   ((= n 0) 1)
   (else (* n (! (- n 1))))))
```

```
(* 3 (! (- 3 1)))  ➡ (* 3 (! 2))
```

「(- 3 1)」は，「2」で置き換わる

Stepper

File Edit Windows Help

Home << Previous Next >>

```
(define (! n)
  (cond
   ((= n 0) 1)
   (else (* n (! (- n 1))))))
```

```
(* 3 (! 2))
```

```
(*
 3
 (cond
  ((= 2 0) 1)
  (else
   (*
    2
    (! (- 2 1))))))
```

!の「n」は「2」で置き換わる

```
(define (! n)
  (cond
   ((= n 0) 1)
   (else (* n (! (- n 1))))))
```

```
(*
 3
 (cond
  ((= 2 0) 1)
  (else
   (*
    2
    (! (- 2 1))))))
```

```
(*
 3
 (cond
  (false 1)
  (else
   (*
    2
    (! (- 2 1))))))
```

「(= 2 0)」は「false」で置き換わる

```
(define (! n)
  (cond
   ((= n 0) 1)
   (else (* n (! (- n 1))))))
```

```
(*
 3
 (cond
  (false 1)
  (else
   (*
    2
    (! (- 2 1))))))
```

```
(*
 3
 (* 2 (! (- 2 1))))
```

「(cond [false 式X] [else 式Y])」は
「式Y」で置き換わる

```
(define (! n)
  (cond
   ((= n 0) 1)
   (else (* n (! (- n 1))))))
```

```
(*                              (* 3 (* 2 (! 1)))
 3
 (* 2 (! (- 2 1))))
```

「(- 2 1)」は，「1」で置き換わる

```
(define (! n)
  (cond
   ((= n 0) 1)
   (else (* n (! (- n 1))))))
```

```
(* 3 (* 2 (! 1)))
```

```
(*
 3
 (*
  2
  (cond
   ((= 1 0) 1)
   (else
    (*
     1
     (! (- 1 1)))))))
```

!の「n」は「1」で置き換わる

```
(define (! n)
  (cond
   ((= n 0) 1)
   (else (* n (! (- n 1))))))
```

```
(*
 3
 (*
  2
  (cond
   ((= 1 0) 1)
   (else
    (*
     1
     (! (- 1 1)))))))
```

```
(*
 3
 (*
  2
  (cond
   (false 1)
   (else
    (*
     1
     (! (- 1 1)))))))
```

「(= 1 0)」は「false」で置き換わる

```
(define (! n)
  (cond
   ((= n 0) 1)
   (else (* n (! (- n 1))))))
```

```
(*
 3
 (*
  2
  (cond
   (false 1)
   (else
    (*
     1
     (! (- 1 1)))))))
```

```
(*
 3
 (*
  2
  (* 1 (! (- 1 1))))
```

「(cond [false 式X] [else 式Y])」は
「式Y」で置き換わる

```
(define (! n)
  (cond
   ((= n 0) 1)
   (else (* n (! (- n 1))))))
```

```
(*                          (*
 3                           3
 (*                    ➡     (* 2 (* 1 (! 0))))
  2
  (* 1 (! (- 1 1))))
```

「(- 1 1)」は，「0」で置き換わる

```
(define (! n)
  (cond
   ((= n 0) 1)
   (else (* n (! (- n 1))))))
```

```
(*
 3
 (* 2 (* 1 (! 0))))
```

```
(*
 3
 (*
  2
  (*
   1
   (cond
    ((= 0 0) 1)
    (else
     (*
      0
      (!
       (- 0 1))))))))
```

!の「n」は「1」で置き換わる

```
(define (! n)
  (cond
   ((= n 0) 1)
   (else (* n (! (- n 1))))))
```

```
(*                                  (*
 3                                   3
 (*                                  (*
  2                                   2
  (*                                  (*
   1                                   1
   (cond                    ➡          (cond
    ((= 0 0) 1)                         (true 1)
    (else                               (else
     (*                                  (*
      0                                   0
      (!                                  (!
       (- 0 1                                    ))))))
```

「(= 0 0)」は「true」で置き換わる

```
(define (! n)
  (cond
   ((= n 0) 1)
   (else (* n (! (- n 1))))))
```

```
(*                                  (* 3 (* 2 (* 1 1)))
 3
 (*
  2
  (*
   1
   (cond
    (true 1)
    (else
     (*
      0
      (!
       (- 0 1)))))))
```

「(cond [true 式X] [else 式Y])」は「式X」で置き換わる

```
(define (! n)
  (cond
   ((= n 0) 1)
   (else (* n (! (- n 1))))))
```

```
(* 3 (* 2 (* 1 1)))   ➡ (* 3 (* 2 1))
```

「(* 1 1)」は，「1」で置き換わる

```
(define (! n)
  (cond
   ((= n 0) 1)
   (else (* n (! (- n 1))))))
```

```
(* 3 (* 2 1))          ➡ (* 3 2)
```

「(* 2 1)」は，「2」で置き換わる

```
(define (! n)
  (cond
   ((= n 0) 1)
   (else (* n (! (- n 1))))))
```

```
(* 3 2)
```

➡ 6

「(* 3 2)」は，「6」で置き換わる

(! 3) から 6 が得られる過程の概略

(! 3) 最初の式

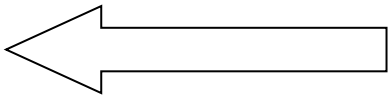
```
=   ...
=   (* 3 (! 2))
= ...
= (* 3 (* 2 (! 1)))
=   ...
=   (* 3 (* 2 (* 1 (! 0))))
=   ...
=   (* 3 (* 2 (* 1 1)))
=   (* 3 (* 2 1))
=   (* 3 2)
```

コンピュータ内部での計算

= 6 実行結果

(! 3) から 6 が得られる過程の概略

```
(! 3)
=    ...
=    (* 3 (! 2))
```



この部分は

```
(! 3)
= (cond
      [(= 3 0) 1]
      [else (* 3 (! (- 3 1)))])
= (cond
      [false 1]
      [else (* 3 (! (- 3 1)))])
= (* 3 (! (- 3 1)))
= (* 3 (! 2))
```

```
= ...
= (* 3 (* 2 (! 1)))
=    ...
=    (* 3 (* 2 (* 1 (! 0))))
=    ...
=    (* 3 (* 2 (* 1 1)))
=    (* 3 (* 2 1))
=    (* 3 2)
=  6
```

(! 3) から 6 が得られる過程の概略

(! 3)
= ...
= (* 3 (! 2))

この部分は

(! 3)
= (cond
[(= 3 0) 1]
[else (* 3 (! (- 3 1)))])
= (cond
[false 1]

= ...
= (* 3 (* 2 (! 1)))
=
=
=
=
=
=
=
= 6

これは,

```
(define (! n)
  (cond
    [(= n 0) 1]
    [else (* n (! (- n 1)))]))
```

の n を 3 で置き換えたもの

(! 3) から 6 に至る過程

```
(! 3)
=    …
=    (* 3 (! 2))
=    …
=    (* 3 (* 2 (! 1)))
=    …
=    (* 3 (* 2 (* 1 (! 0))))
=    …
=    (* 3 (* 2 (* 1 1)))
=    (* 3 (* 2 1))
=    (* 3 2)
=    6
```

基本的な計算式への展開

「(! 3)」 が膨張して
「(* 3 (* 2 (* 1 (! 0))))」
になる

演算の実行

「(* 3 (* 2 (* 1 (! 0))))」
が収縮して 6 になる

線形再帰的プロセス

- 基本的な計算式へ展開

例　(! 3)　⇒　(* 3 (* 2 (* 1 (! 0))))

再帰の呼び出し回数（＝ステップ数ともいう）に比例して成長する
⇒「線形再帰」の名前の由来

! が繰り返される回数

(! 3) ①
= ...
= (* 3 (! 2)) ②
= ...
= (* 3 (* 2 (! 1))) ③
= ...
= (* 3 (* 2 (* 1 (! 0)))) ④
= ...
= (* 3 (* 2 (* 1 1)))
= (* 3 (* 2 1))
= (* 3 2)
= 6

n=3 のとき,
4回繰り返して実行される

! が繰り返される回数

(! 4) ①
= …
= (* 4 (! 3)) ②
= …
= (* 4 (* 3 (! 2))) ③
= …
= (* 4 (* 3 (* 2 (! 1)))) ④
= …
= (* 4 (* 3 (* 2 (* 1 (! 0))))) ⑤
= …
= (* 4 (* 3 (* 2 (* 1 1))))
= (* 4 (* 3 (* 2 1)))
= (* 4 (* 3 2))
= (* 4 6)
= 24

n=4 のとき,
5回繰り返して実行される

例題３． 反復的プロセスでの階乗

- 階乗を計算する関数！を作り，実行する
  - 次の方針でプログラムを作成する

    n > 0 のとき，１から開始して，１×２×・・・×n を計算する

例） (! 6) = １×２×３×４×５×６

反復的プロセスでの階乗

- n! の計算
  1. まず，1 に 2 を掛ける
  2. 次に，3 を掛ける
  3. ・・・
  4. n に達するまで続ける

「例題３．反復的プロセスでの階乗」の手順

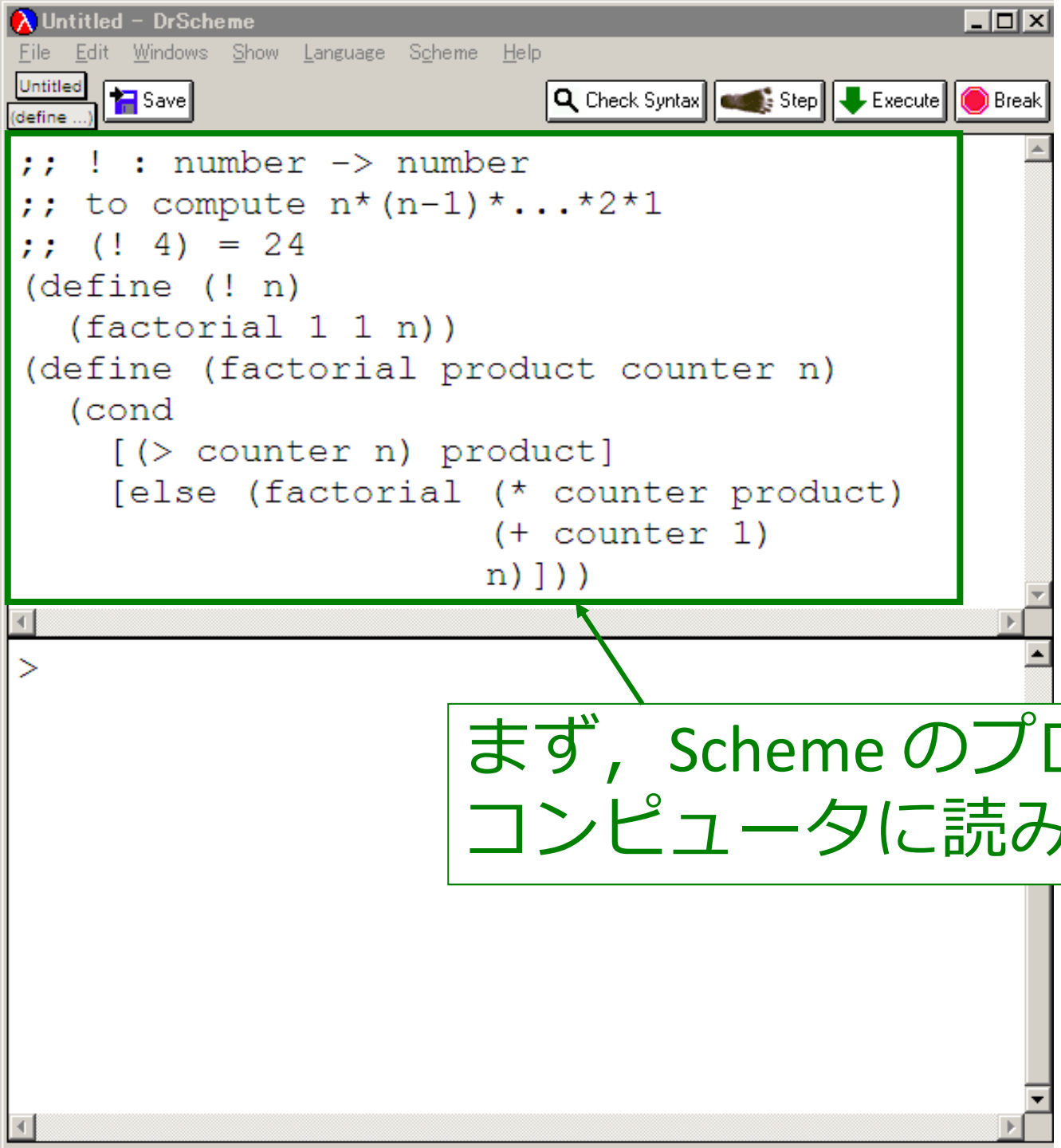
1. 次を「定義用ウインドウ」で，実行しなさい
   - 入力した後に，Execute ボタンを押す

```
;; ! : number -> number
;; to compute n*(n-1)*...*2*1
;; (! 4) = 24
(define (! n)
   (factorial 1 1 n))
(define (factorial product counter max)
   (cond
      [(> counter max) product]
      [else (factorial (* counter product)
                       (+ counter 1)
                       max)]))
```

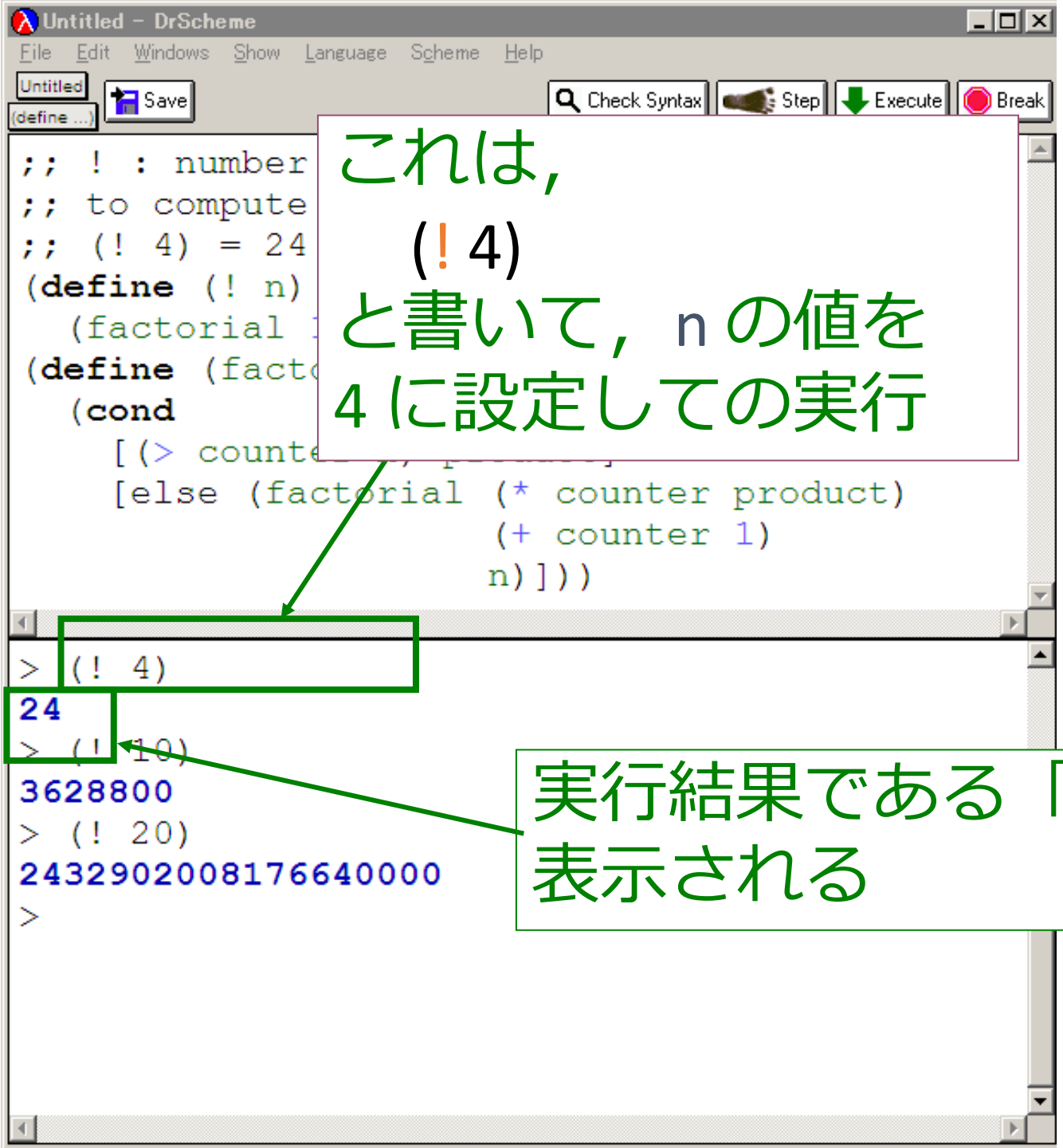
2. その後，次を「実行用ウインドウ」で実行しなさい

```
(! 4)
(! 10)
(! 20)
```

☆　次は，例題４に進んでください



まず，Scheme のプログラムを
コンピュータに読み込ませている



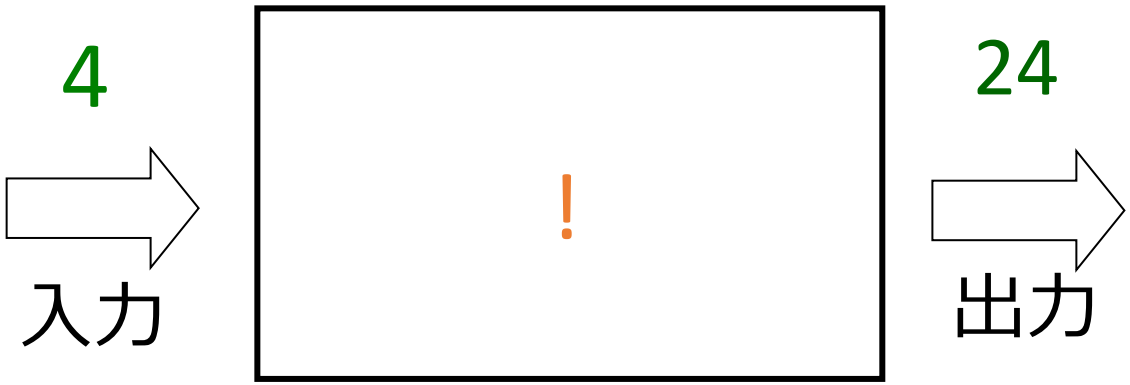
これは,

(! 4)

と書いて，n の値を
4 に設定しての実行

実行結果である「24」が
表示される

入力と出力



入力は数値

出力は数値

！関数

```
;; ! : number -> number
;; to compute n*(n-1)*...*2*1
;; (! 4) = 24
(define (! n)
   (factorial 1 1 n))
(define (factorial product counter n)
   (cond
      [(> counter n) product]
      [else (factorial (* counter product)
                       (+ counter 1)
                       n)]))
```

product ← counter ・ product

counter ← counter + 1

反復的プロセスでの階乗

counter:　1 から n まで数えるカウンタ
product: 部分積（計算の途中結果）
とする.

product ← counter・product
counter ← counter + 1

を繰り返す．counter が n に達すると
n!が求まる

反復的プロセスでの階乗

1. counter > n ならば：→　終了条件

   product　→　自明な解

2. そうで無ければ：

   次を実行する

   product ← counter・product

   counter ← counter + 1

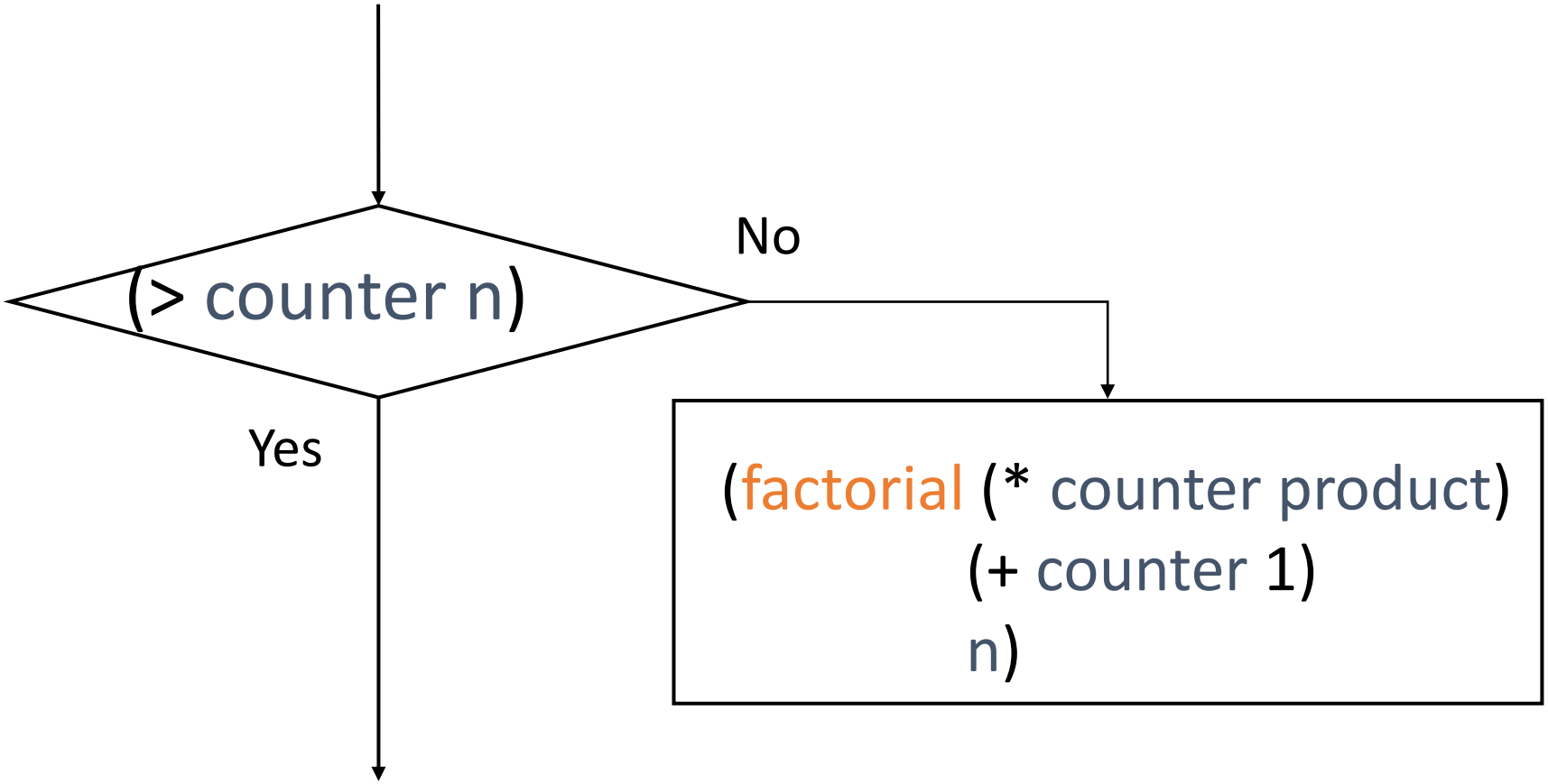
反復的プロセスでの階乗

```
;; ! : number -> number
;; to compute n*(n-1)*...*2*1
;; (! 4) = 24
(define (! n)
   (factorial 1 1 n))
(define (factorial product counter n)
   (cond
      [(> counter n) product]
      [else (factorial (* counter product)
                       (+ counter 1)
                       n)]))
```

終了条件

自明な解

終了条件



product が自明の解

例題４．ステップ実行

- factorial の内部に factorial が登場

```
(define (factorial product counter n)
   (cond
      [(> counter n) product]
      [else (factorial (* counter product)
                       (+ counter 1)
                       n)]))
```

- factorial の実行が繰り返される

  例：(factorial 6 4 10) = (factorial 24 5 10)

  (factorial 24 5 10) = (factorial 120 6 10)

例題４．ステップ実行

- 関数！（例題３）について，実行結果に至る過程を見る
  - (! 4) から 24 に至る過程を見る
  - DrScheme の stepper を使用する

```
(define (factorial product counter n)
   (cond
      [(> counter n) product]
      [else (factorial (* counter product)
                       (+ counter 1)
                       n)]))
```

```
(! 4)
=   (factorial 1  1  4)
=   ...
=   (factorial 1  2  4)
=   ...
=   (factorial 2  3  4)
=   ...
=   (factorial 6  4  4)
=   ...
=   (factorial 24  5  4)
=   ...
=   24
```

「例題４．ステップ実行」の手順

1. 次を「定義用ウインドウ」で，実行しなさい
   - Intermediate Student で実行すること
   - 入力した後に，Execute ボタンを押す

```
(define (! n)
   (factorial 1 1 n))
(define (factorial product counter max)
   (cond
      [(> counter max) product]
      [else (factorial (* counter product)
                       (+ counter 1)
                       max)]))
(! 4)
```

← 例題３と同じ

2. DrScheme を使って，ステップ実行の様子を確認しなさい　（Step ボタン，Next ボタンを使用）
   - 理解しながら進むこと

☆　次は，例題５に進んでください

(! 4) から 24 が得られる過程の概略

(! 4) 最初の式

```
=   (factorial 1  1  4)
=   ...
=   (factorial 1  2  4)
=   ...
=   (factorial 2  3  4)
=   ...
=   (factorial 6  4  4)
=   ...
=   (factorial 24  5  4)
=   ...                    コンピュータ内部での計算
```

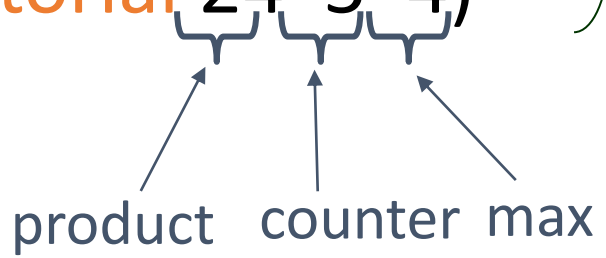
= 24 実行結果

(! 4) から 24 が得られる過程の概略

```
(! 4)
=   (factorial 1  1  4)
=   …
=   (factorial 1  2  4)
=   …
=   (factorial 2  3  4)
=   …
=   (factorial 6  4  4)
=   …
=   (factorial 24  5  4)
=   …
=   24
```

product, counter の
値が変化する

- counter
  → 繰り返し回数
- product
  → 部分積



反復的プロセスの特徴

- 線形再帰的プロセスのような伸び縮みは無い
- 関数を再帰的に呼び出す各ステップで計算が実行される

例　(! 3)　= (factorial 1 1 3)
= (factorial 1 2 3)
= (factorial 2 3 3)
= (factorial 6 4 3)

各ステップで，product と counter に関する計算が実行される

伸び縮み無し

(factorial 1 1 4) から (factorial 1 2 4) が得られる過程

(! 4)
= (factorial 1 1 4)
= ...
= (factorial 1 2 4)
= ...
= (factorial 2 3 4)
= ...
= (factorial 6 4 4)
= ...
= (factorial 24 5 4)
= ...
= 24

この部分は

```
(factorial 1 1 4)
= (cond
      [(> 1 4) 1]
      [else (factorial (* 1 1)
                       (+ 1 1)
                       4)])
= (cond
      [false 1]
      [else (factorial (* 1 1)
                       (+ 1 1)
                       4)])
= (factorial (* 1 1) (+ 1 1) 4)
= (factorial 1 (+ 1 1) 4)
= (factorial 1 2 4)
```

(factorial 1 1 4) から (factorial 1 2 4) が得られる過程

(! 4)
= (factorial 1 1 4)
=
=
=
=
=
=
=
=
=
= 24

(factorial 1 1 4)
= (cond
[(> 1 4) 1]
[else (factorial (* 1 1)
(+ 1 1)
4)])
=(cond

この部分は

これは,

```
(define (factorial product counter n)
  (cond
    [(> counter n) product]
    [else (factorial (* counter product)
                     (+ counter 1)
                     n)]))
```

の counter を 1 で，product を 1 で，n を 4 で置き換えたもの

(! 4) から (* 4 (! 3)) が得られる過程


```
(! 4)
=    ...
=    (* 4 (! 3))
=  ...
=  (* 4 (* 3 (! 2)))
=
=
=
=
=
=
=
=  (* 4 6)
=   6
```

この部分は

```
(! 4)
= (cond
    [(= 4 0) 1]
    [else (* 4 (! (- 4 1)))])
= (cond
    [false 1]
```

これは,

```
(define (! n)
  (cond
    [(= n 0) 1]
    [else (* n (! (- n 1)))]))
```

の n を 4 で置き換えたもの

factorial が繰り返される回数

(! 4)
= (factorial 1 1 4) ①
= …
= (factorial 1 2 4) ②
= …
= (factorial 2 3 4) ③
= …
= (factorial 6 4 4) ④
= …
= (factorial 24 5 4) ⑤
= …
= 24

n=4 のとき，
5回繰り返して実行される

例題５．繰り返し回数

- 次のプログラムでは，square は何回実行されるか

```
(define (square x)
    (* x x))
(define (total-square x)
    (cond
        [(empty? x) 0]
        [else (+ (square (first x)) (total-square (rest x)))]))
```

繰り返し回数

実行結果の例

```
(total-square (list 10 20 30))

1400
```

- square は，リストの要素数だけ実行される

例題６．最大公約数の計算

- ユークリッドの互助法を使って，２つの整数 m, n から，最大公約数を求めるプログラム my-gcd を作り，実行する

  例）　180, 32 のとき：　4

  - ユークリッドの互助法を用いる

ユークリッドの互助法

- 2 つの整数 m, n の最大公約数:

  （m, n は正または 0 ）

  - n = 0 なら

    最大公約数は m

  - n ≠ 0 なら

    最大公約数は, 「m を n で割った余り」 と n の最大公約数に等しい

「例題６．最大公約数の計算」の手順

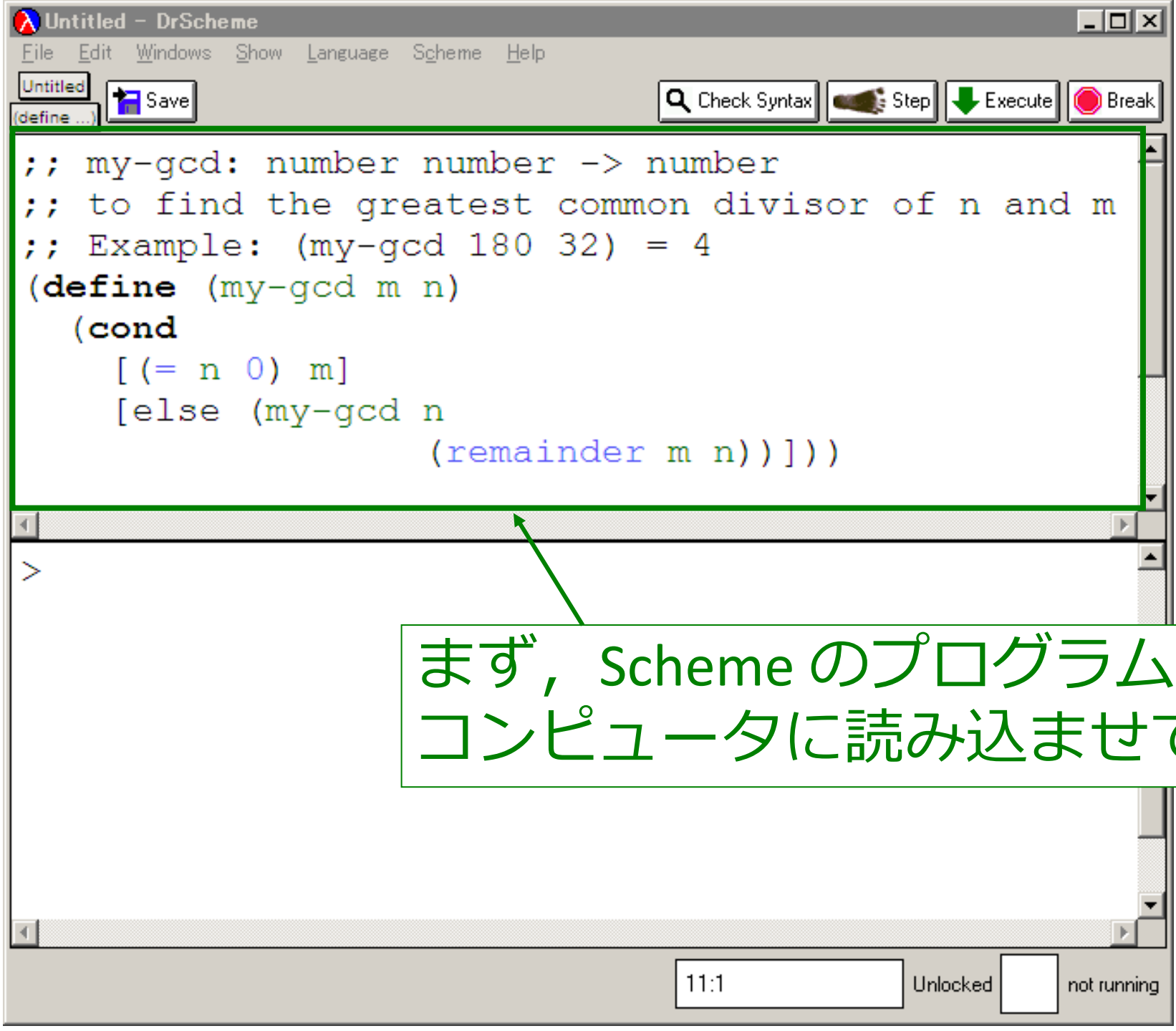
1. 次を「定義用ウインドウ」で，実行しなさい
   - 入力した後に，Execute ボタンを押す

```
;; my-gcd: number number -> number
;; to find the greatest common divisor of n and m
;; Example: (my-gcd 180 32) = 4
(define (my-gcd m n)
   (cond
      [(= n 0) m]
      [else (my-gcd n
                    (remainder m n))]))
```

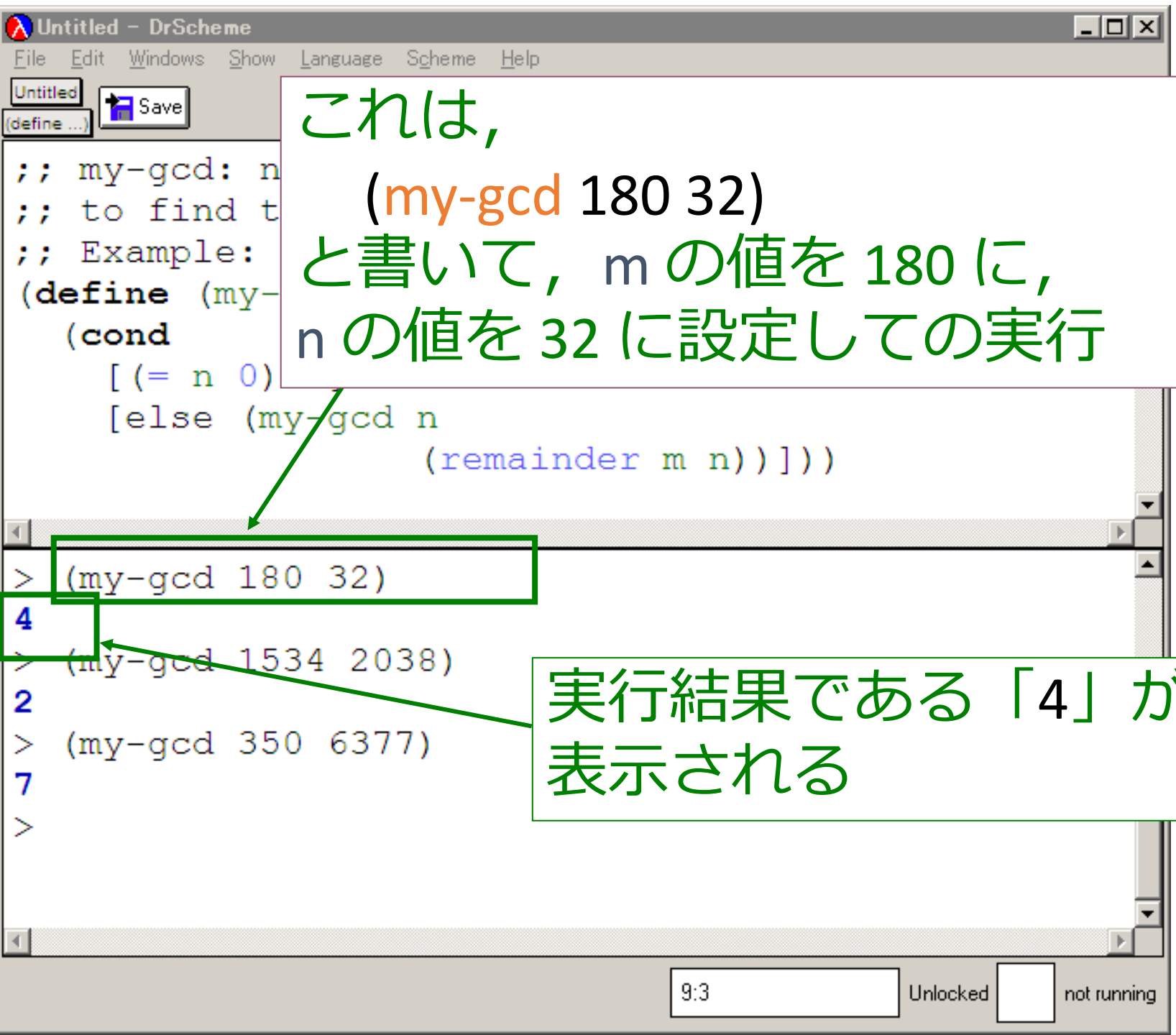
2. その後，次を「実行用ウインドウ」で実行しなさい

```
(mygcd 180 32)
```

☆　次は，例題７に進んでください



まず，Scheme のプログラムを
コンピュータに読み込ませている



これは,

(my-gcd 180 32)

と書いて，m の値を 180 に，
n の値を 32 に設定しての実行

実行結果である「4」が
表示される

入力と出力

入力は，２つの数値　　出力は数値

my-gcd 関数


```
;; my-gcd: number number -> number
;; to find the greatest common divisor of n and m
;; Example: (my-gcd 180 32) = 4
(define (my-gcd m n)
    (cond
        [(= n 0) m]
        [else (my-gcd n
                      (remainder m n))]))
```

最大公約数の計算

1. n = 0 ならば： → 終了条件

   m → 自明な解

2. そうで無ければ：

   (1) n

   (2) m を n で割った余り

   の２数の最大公約数を求める.

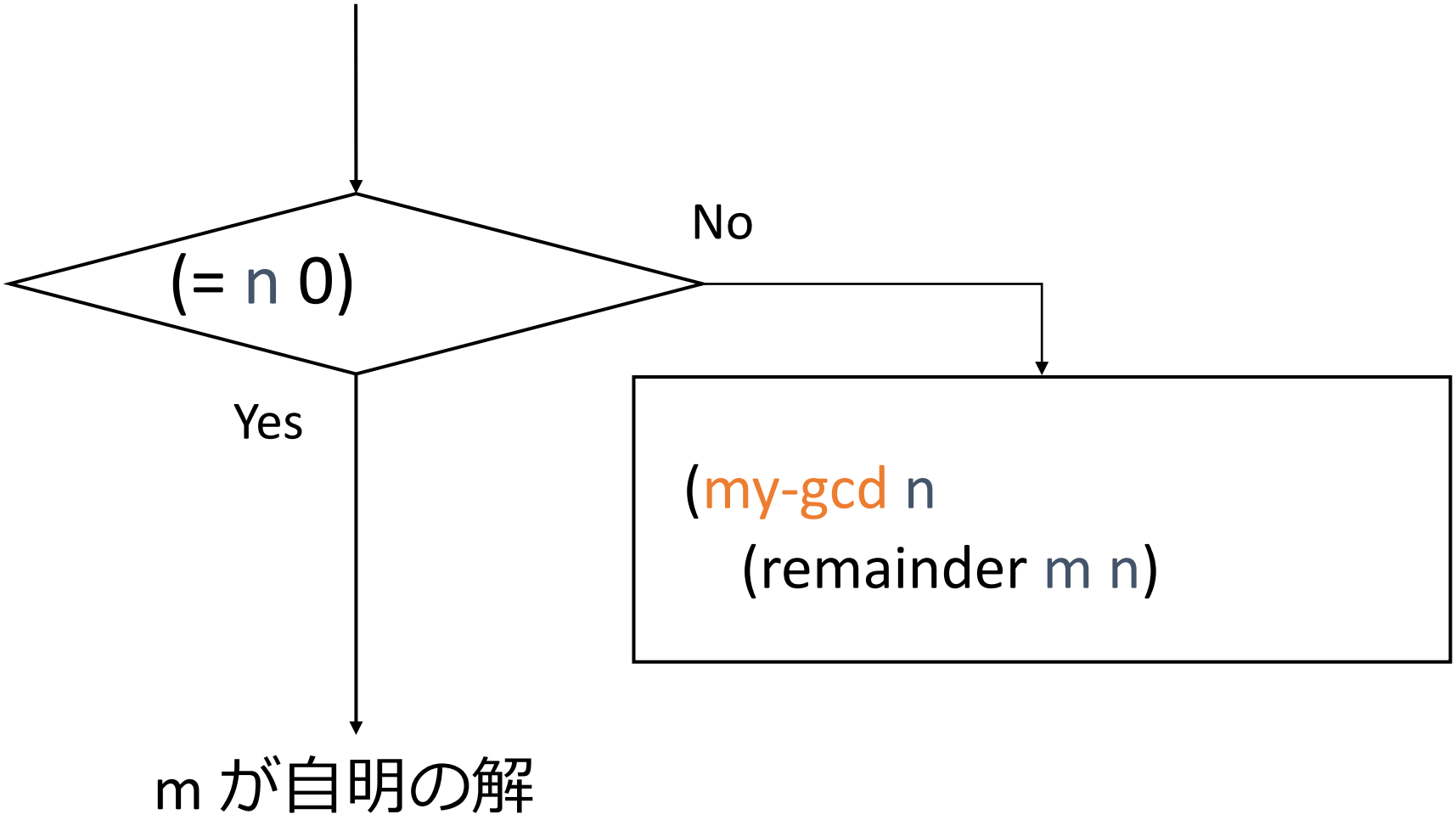
以上のことを，n が 0 に達するまで繰り返す.

```
;; my-gcd: number number -> number

;; to find the greatest common divisor of n and m

;; Example: (my-gcd 180 32) = 4

(define (my-gcd m n)

    (cond

      [(= n 0) m]

        [else (my-gcd n

                      (remainder m n))]))
```

終了条件

自明な解

終了条件



最大公約数の計算

- my-gcd の内部に my-gcd が登場

```
(define (my-gcd m n)
   (cond
      [(= n 0) m]
      [else (my-gcd n
                    (remainder m n))]))
```

- my-gcd の実行が繰り返される

例： (my-gcd 180 32)

= (my-gcd 32 20)

例題７．ステップ実行

- 関数 my-gcd （例題６）について，実行結果に至る過程を見る
  - (my-gcd 180 32) から 4 に至る過程を見る
  - DrScheme の stepper を使用する

```
(define (my-gcd m n)
  (cond
    [(= n 0) m]
    [else (my-gcd n
                  (remainder m n))]))
```

```
(my-gcd 180 32)
= ...
= (my-gcd 32 20)
= ...
= (my-gcd 20 12)
= ...
= (my-gcd 12 8)
= ...
= (my-gcd 8 4)
= ...
= (my-gcd 4 0)
= ...
= 4
```

「例題７．ステップ実行」の手順

1. 次を「定義用ウインドウ」で，実行しなさい
   - Intermediate Student で実行すること
   - 入力した後に，Execute ボタンを押す

```
;; my-gcd: number number -> number
;; to find the greatest common divisor of n and m
;; Example: (my-gcd 180 32) = 4
(define (my-gcd m n)
   (cond
      [(= n 0) m]
      [else (my-gcd n
                    (remainder m n))]))
(my-gcd 180 32)
```

例題６と同じ

2. DrScheme を使って，ステップ実行の様子を確認しなさい　（Step ボタン，Next ボタンを使用）
   - 理解しながら進むこと

☆　次は，課題に進んでください

(my-gcd 180 32) から 4 が得られる過程の概略

(my-gcd 180 32) 最初の式

= …
= (my-gcd 32 20)
= …
= (my-gcd 20 12)
= …
= (my-gcd 12 8)
= …
= (my-gcd 8 4)
= …
= (my-gcd 4 0)
= …
= 4

コンピュータ内部での計算

実行結果

(my-gcd 180 32)から (my-gcd 32 20) が得られる過程

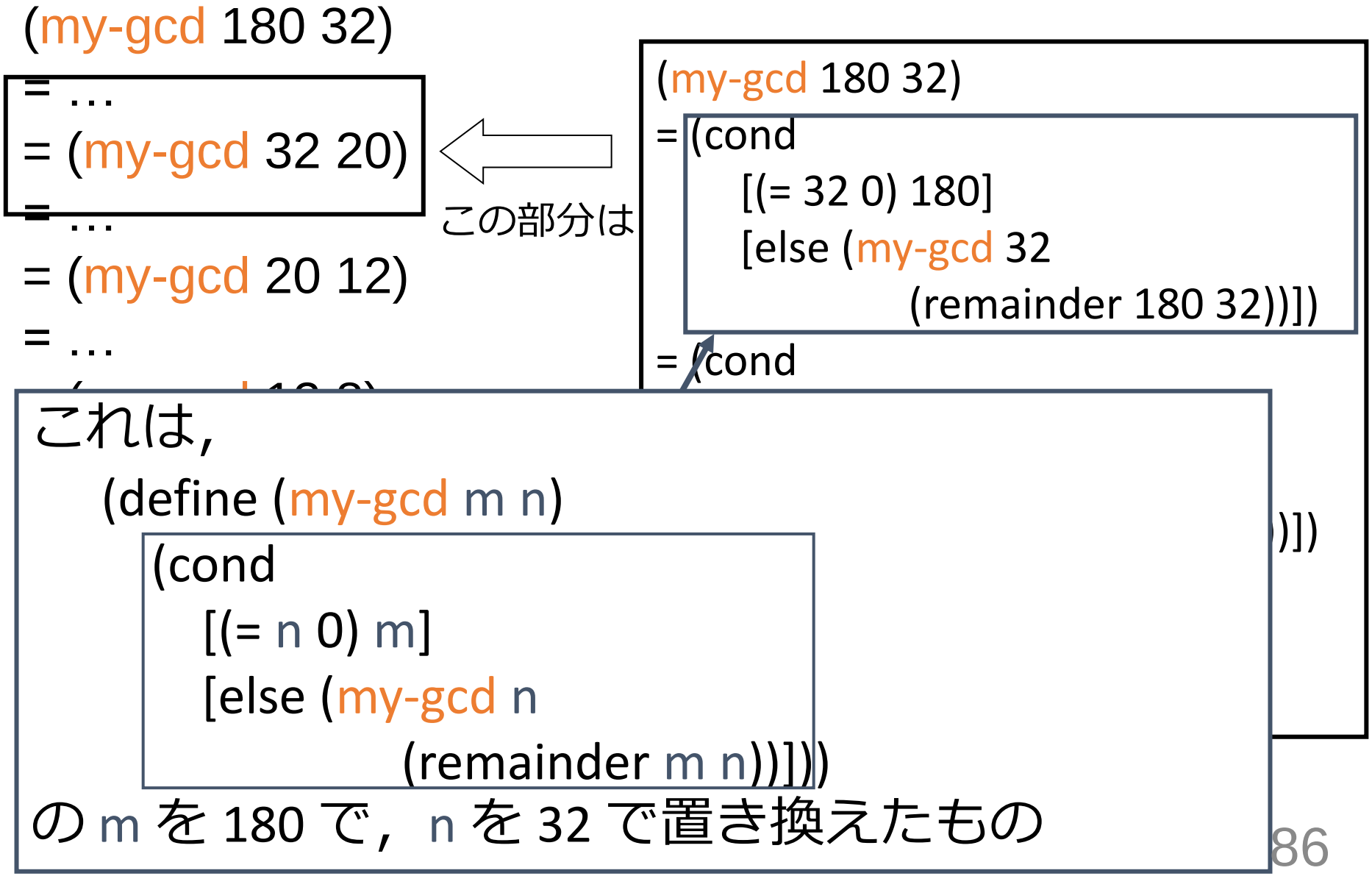
```
(my-gcd 180 32)
= …
= (my-gcd 32 20)
= …
= (my-gcd 20 12)
= …
= (my-gcd 12 8)
= …
= (my-gcd 8 4)
= …
= (my-gcd 4 0)
= …
= 4
```

この部分は

```
(my-gcd 180 32)
= (cond
      [(= 32 0) 180]
      [else (my-gcd 32
                    (remainder 180 32))])
= (cond
      [false 180]
      [else (my-gcd 32
                    (remainder 180 32))])
= (my-gcd 32
       (remainder 180 32))
= (my-gcd 32 20)
```

180 を 32 で割った余りは 20

(my-gcd 180 32) から (my-gcd 32 20) が得られる過程



my-gcd が繰り返される回数

(my-gcd 180 32) ①
= …
= (my-gcd 32 20) ②
= …
= (my-gcd 20 12) ③
= …
= (my-gcd 12 8) ④
= …
= (my-gcd 8 4) ⑤
= …
= (my-gcd 4 0) ⑥
= …
= 4

m=180, n=32 のとき，
6回繰り返して実行される

12-3 課題

課題 1

- 関数 my-gcd （授業の例題６）に関する問題
  - (my-gcd 210 66) から 6 が得られる過程の概略を数行程度で説明しなさい
  - DrScheme の stepper を使うと，すぐに分かる

課題 2

- バックの中のコインの合計を求める関数 sum-coin についての問題
  - 下記の空欄を埋めて，関数 sum-coins の定義を終えなさい．実行結果も報告しなさい
  - sum-coins は，コインの個数のリストと，コインの金額のリストの２つのリストを入力とする

```
;; Contract: sum-coins : a-list-of-numbers a-list-of-numbers -> number
;; Example: (sum-coins (list 23 0 5 7) (list 1 5 10 25)) = 248
(define (sum-coins a b)
    (cond   [(                    )                              ]
  [else     (+  (*   (            ) (            ))
             (         (                ) (                ))))]))
```

課題２のヒント

- ここにあるのは「間違い」の例です．同じ間違いをしないこと

１．(= a empty) は間違い

⇒　a がリストのとき、(= a empty) はエラーです．
「=」は数値の比較には使えるが，リスト同士の比較には使えないものと考えて下さい
正しくは，(empty? a) です

課題３．繰り返し回数

- 関数 my-gcd （授業の例題６）に関する問題
  - m, n の最大公約数を，ユークリッドの互助法で求めた場合と，次ページに示すような方法で求めた場合とで，関数の繰り返し回数を比較し，自分なりの考察を加えて報告しなさい

```
(define (first-divisor n m i)
   (cond
      [(= i 1) 1]
      [else (cond
         [(and (= (remainder n i) 0)
               (= (remainder m i) 0)) i]
         [else (first-divisor n m (- i 1))])]))
(define (gcd-structural n m)
     (first-divisor n m (min n m)))
```

「i で割り切れるかを調べながら，i を 1 減らす」ことを，割り切れるまで繰り返す

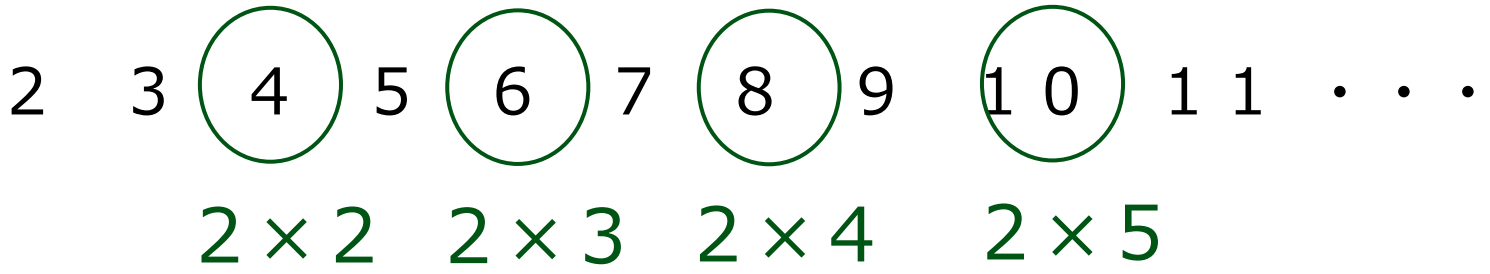
1. まず，n と m の小さい方を変数 i に入れる.
2. i が n と m の両方を割れれば i の値を返し，終了.
3. i の値を1小さくして2へ.

   → n と m は変わらない．i が変化

課題４．エラトステネスのふるい

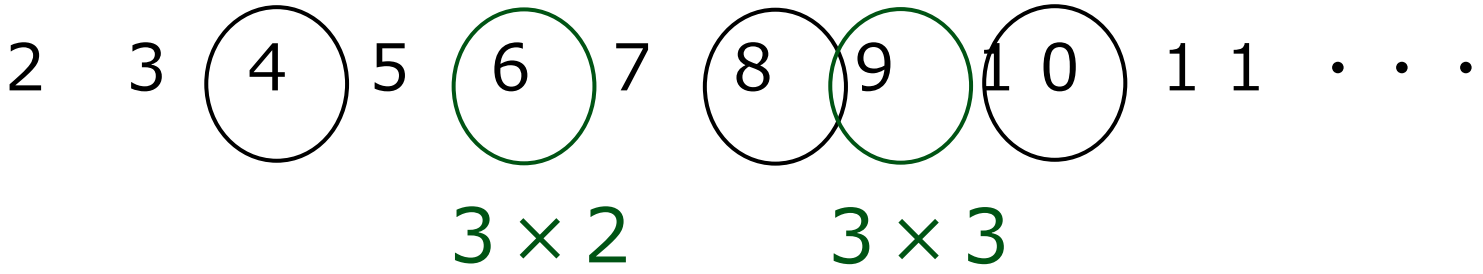
- 「エラトステネスのふるい」の原理に基づいて100以下の素数を求めるプログラムを作りなさい

エラトステネスのふるい (1/4)



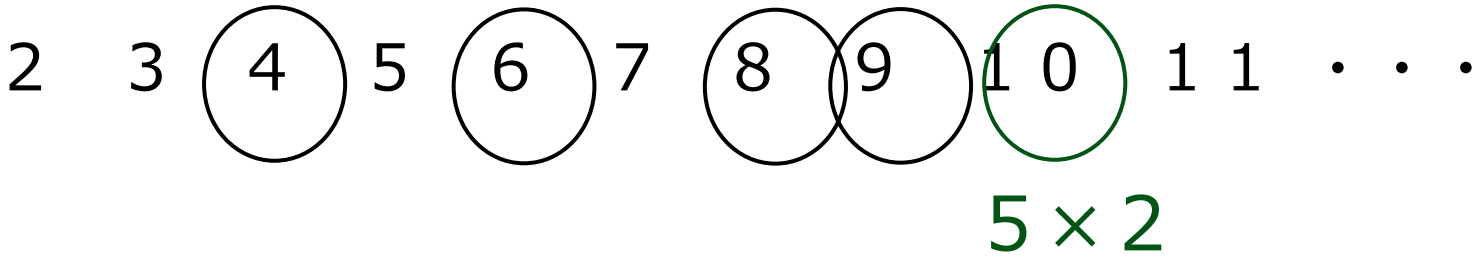
まず，２の倍数を消す

エラトステネスのふるい (2/4)



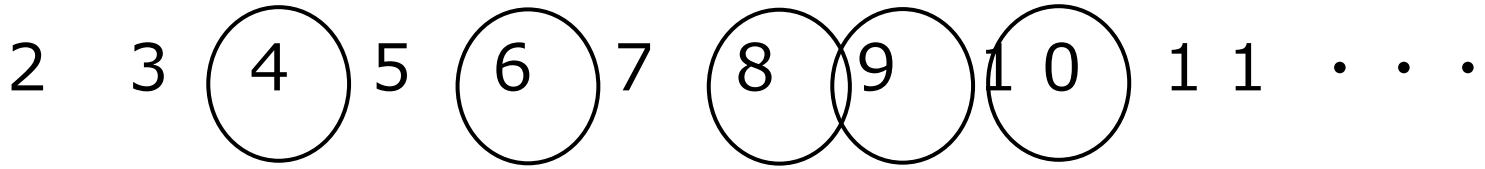
次に，３の倍数を消す

エラトステネスのふるい (3/4)



次に，５の倍数を消す
（「４の倍数」は考えない．
それは，「４」がすでに消えているから）

エラトステネスのふるい (4/4)



以上のように，２，３，５・・・の倍数を消す.

１０（これは１００の平方根）を超えたら，この操作を止める
（１００以下で，１１，１３・・・の倍数はすでに消えている）