関数型言語（OCaml 演習）(13)

逐次実行

浅井 健一

お茶の水女子大学

副作用を持つ関数

出力を計算する以外のことも行う関数。例：

`print_string` *`str`*

文字列 *`str`* を受け取ったら、
それを画面に出力した上で () を返す。

実行された時点で（結果に関わらず）表示される。

副作用を持つ関数

出力を計算する以外のことも行う関数。例：

`print_string` *`str`*

文字列 *`str`* を受け取ったら、
それを画面に出力した上で () を返す。

実行された時点で（結果に関わらず）表示される。

```
# print_string "こんにちは" ;;
こんにちは- : unit = ()
```

副作用を持つ関数

出力を計算する以外のことも行う関数。例：

`print_string` *`str`*

文字列 *`str`* を受け取ったら、
それを画面に出力した上で () を返す。

実行された時点で（結果に関わらず）表示される。

```
# print_string "こんにちは" ;;
こんにちは- : unit = ()

# "新" ^ "横浜" ;;                    ← 純粋な関数の場合
- : string = "新横浜"
# String.length ("新" ^ "横浜") ;;
- : int = 6
```

unit 型

() : unit

unit 型は「値に意味がない」ときに使う型。その型の値は () のみで、これも「ユニット」と読む。

```
# () ;;
- : unit = ()
```

unit 型

() : unit

unit 型は「値に意味がない」ときに使う型。その型の値は () のみで、これも「ユニット」と読む。

```
# () ;;
- : unit = ()
# print_string ;;
- : string -> unit = <fun>
```

print_string の返す値には意味がないことを示す。

unit 型

() : unit

unit 型は「値に意味がない」ときに使う型。その型の値は () のみで、これも「ユニット」と読む。

```
# () ;;
- : unit = ()
# print_string ;;
- : string -> unit = <fun>
# print_newline ;;
- : unit -> unit = <fun>
# print_newline () ;;
```

改行を出力する関数

← 改行が出力されている

```
- : unit = ()
```

逐次実行

(式$_1$; 式$_2$; ...; 式$_n$; 式)

式$_1$ を実行し、その結果を捨てて、式$_2$ を実行し、その結果を捨てて、と繰り返し、最後に一番最後の式を実行し、その結果を全体の結果とする。

逐次実行

(式$_1$; 式$_2$; ...; 式$_n$; 式)

式$_1$ を実行し、その結果を捨てて、式$_2$ を実行し、その結果を捨てて、と繰り返し、最後に一番最後の式を実行し、その結果を全体の結果とする。

```
# (print_string "最短距離は ";
   print_float 3.2;
   print_string "km です。";
   print_newline ();
   0) ;;
最短距離は 3.2km です。
- : int = 0
```

式 $_n$ までは、副作用を持つ式でないと意味がない。

実行中の変数の表示

```
(* 目的：自然数 m, n の最大公約数を求める *)
(* gcd : int -> int -> int *)
let rec gcd m n =


  if n = 0 then m
           else gcd n (m mod n)
```

実行中の変数の表示

```
(* 目的：自然数 m, n の最大公約数を求める *)
(* gcd : int -> int -> int *)
let rec gcd m n =
 (print_string "m = ";
  print_int m;                (* m の値を表示 *)
  print_string ", n = ";
  print_int n;                (* n の値を表示 *)
  print_newline ();           (* 改行         *)
  if n = 0 then m
           else gcd n (m mod n))
```

逐次実行文で、本来の処理の前に表示関数を挿入する。

実行中の変数の表示

```
# gcd 216 126 ;;
m = 216, n = 126
m = 126, n = 90
m = 90, n = 36
m = 36, n = 18
m = 18, n = 0
- : int = 18
#
```

実行の順序

式の依存関係から自然に定まる。

`gcd 216 126 + gcd 36 24` なら

- `gcd 216 126`
- `gcd 36 24`

の計算を + の前に行う必要がある。

実行の順序

式の依存関係から自然に定まる。

`gcd 216 126 + gcd 36 24` なら

- `gcd 216 126`
- `gcd 36 24`

の計算を + の前に行う必要がある。

純粋な関数なら、それ以外の実行順序は任意。

ふたつの `gcd` の計算は、どちらを先に行っても良い。

実行の順序

式の依存関係から自然に定まる。

`gcd 216 126 + gcd 36 24` なら

- `gcd 216 126`
- `gcd 36 24`

の計算を `+` の前に行う必要がある。

副作用を持つ関数だと実行順序によって挙動が変わる。

どちらの `gcd` を先に計算するかによって、表示される `m, n` の値が変わってしまう。

OCaml では、式を「右から」実行する。

まとめ

- `unit` 型とその値 `()`。
- 副作用のある関数群：

```
print_string   : string -> unit
print_int      : int -> unit
print_float    : float -> unit
print_newline  : unit -> unit
```

- 逐次実行文。(式$_1$; 式$_2$; ...; 式$_n$; 式)
  式$_1$ から式$_n$ までは、普通は `unit` 型の式。

副作用のある関数は、実行順序によって挙動が変わる。

(OCaml では式を右から実行するが、それに依存するプログラムを書くのは避けるべき。)