`interact` のすすめ

それでも関数的に書きたいあなたに

2025-06-14

関数型まつり2025

山下伸夫 @ SampouOrg

interact

```
interact :: (String -> String) -> IO ()
            ------------------    -----
                 関　数           命 令
```

「関数」から「命令」への関数

関数的（functional）

> **Functional**(also called **right-unique** or **univalent**): for all $x \in X$ and all $y, z \in Y$, if xRy and xRz then $y = z$.
>
> WikiPedia: Binary relation

「関数的」のお気持

	関数的	命令的
思考	トップダウン	ボトムアップ
時間	静的	動的
状態	明示的	暗黙的
型付	静的	動的
計算	関数適用	命令実行
区切	関数合成演算子 `.`	命令区切り子 `;`

フォン・ノイマン型計算機を前提としない空想の世界

関数型（function type）

`a -> b` ： `a` 型の値に適用すると `b` 型の値になる「関数」の型

> σ および τ が型ならば $\sigma \to \tau$ は型

命令型（instruction type）

`IO a` ： `a` 型の値を得る「命令」の型

N.B. 「命令型」は、この資料独自の（オレオレ）用語法

Haskellプログラミング ≠ 関数プログラミング

Haskellは命令も書ける

```
main :: IO ()
main = putStrLn "こんにちは世界"
```

逐次実行構文

```
main :: IO ()
main = do
    { inp <- getContents                -- ^ 標準入力の内容取得 inp :: String
    ; let xs = words inp                -- ^ words :: String -> [String]
    ; ys <- return $ map (read @Int) xs -- ^ map (read @Int) :: [String] -> [Int]
    ; ys <- return $ scanl1 (+) ys      -- ^ 累積和列
    ; forM_ ys print                    -- ^ forM_ :: [Int] -> (Int -> IO ()) -> IO ()
    }
```

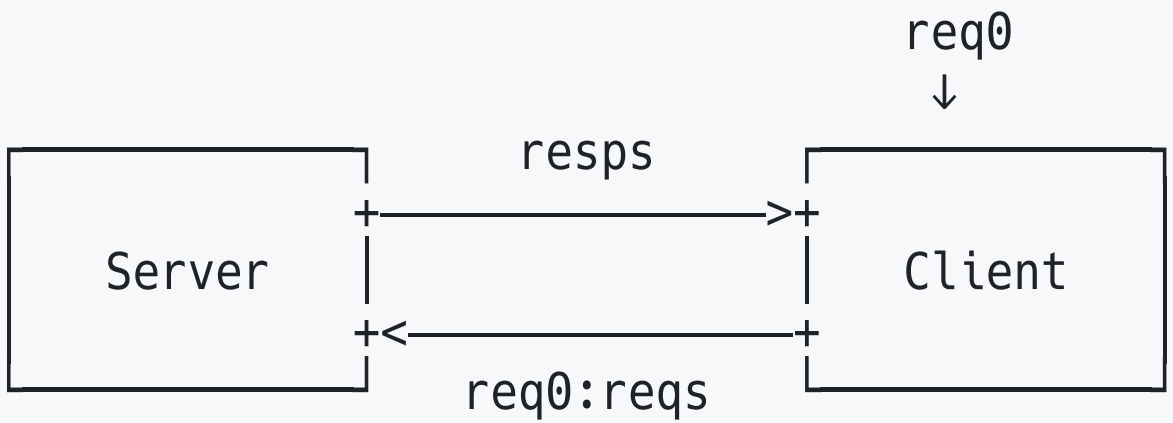
対話（dialogue）

対話：メッセージのやりとりの繰り返し

クライアント/サーバー

- サーバー： 要求列 → 応答列
- クライアント：応答列 → 要求列

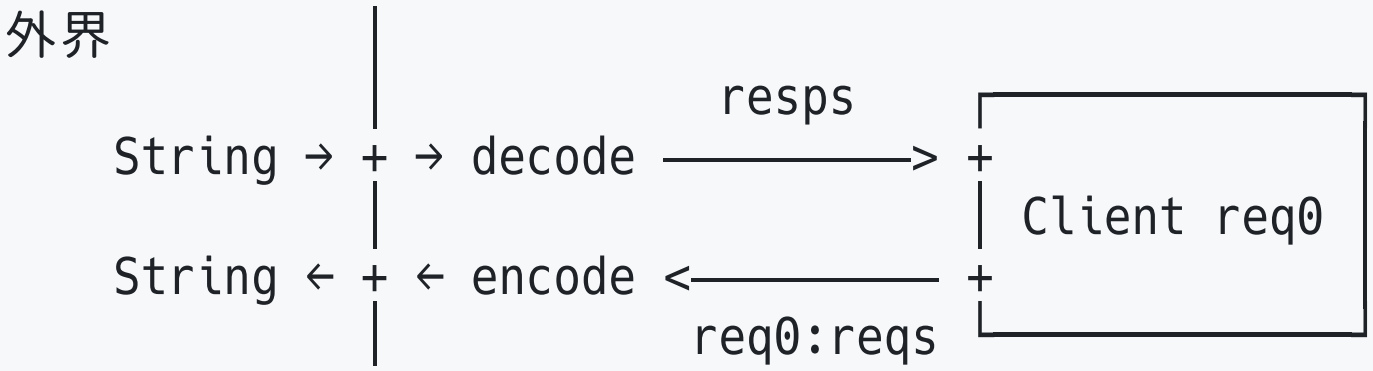
```haskell
type Server = [Request] -> [Response]
type Client = [Response] -> [Request]

server :: Server
server = map process

client :: Request -> Client
client req0 = (req0 :) . map next
```



```haskell
type Request = Double; type Response = Double
type Rate    = Double; type Amount   = Int   ; type Year = Int

process :: Request -> Response
process = ((1 + rate) *)

rate :: Rate
rate = 0.01

next :: Response -> Request
next = id

amount :: Rate -> Amount -> (Year -> Amount)
amount r a = floor . (reqs !!)
    where
        reqs = client req0 (server process reqs)
        req0 = fromIntegral a
{- ^
>>> amount 0.01 10000 20
12201
-}
```

外界との対話

```haskell
main :: IO ()
main = interact (encode . client req0 . decode)
```



```haskell
decode :: String -> [Responses]
decode = map dec . lines
{- lines :: String -> [String] -}
{- dec   :: String -> Response -}

encode :: [Request] -> String
encode = unlines . map enc
{- enc     :: Request  -> String -}
{- unlines :: [String] -> String -}
```

関数型の拡張

- `a -> b` を対話を含むように拡張

```
(a, [Response]) -> (b, [Request],[Response])
```

- カリー化

```
a -> ([Response] -> (b, [Request],[Response])
```

- 対話部分を抽象 `type IO_ b = [Response] -> (b,[Request],[Response])`

```
a -> IO_ b
```

- `IO_ b` の解釈： 対話から得られる `b` 型の値

値の対話への埋め込み

```
unitIO_ :: a -> IO_ a
unitIO v ps = (v, [], ps)
```

関数合成の拡張

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
(f . g) x = f (g x)

(<--) :: (b -> IO_ c) -> (a -> IO_ b) -> (a -> IO_ c)
(f <-- g) x ps = case g x ps of
    (y,qs,ps')        -> case f y ps' of
        (z,qs',ps'')      -> (a,qs ++ qs', ps'')
```

対話の起動

```
run :: IO_ a -> ([Response] -> [Request])
run action rs = case action rs of
    (_, qs, _) -> qs
```