Chapter 2 (NUPSC 2018 講習資料)

§2.1. 無名関数

前章では我々は `defn` を使い関数をいくつか定義してきました。 例えばこのように「引数を'+'1」を意味する関数が定義できます。

```
(defn inc [x] (+ x 1))
```

```
#'cljs.user/inc
```

又 `defn` を説明した時 `defn` を含む式は一つの関数を結果として持つことも説明しました。 それはつまりこのような使いかたもできます。

```
((defn inc [x] (+ x 1))
 10)
```

```
11
```

このコードブロックでは我々は割と面白いことをしています。 それはつまり関数を値として *そのまま* 使っていることです。 C言語 などの一部の言語と違い、ClojureScript では関数は数と同じように値として扱い、 加工したり他の関数に渡したりすることができます。

実は ClojureScript では名前を与えずに関数を作ることもできます。 例えば上のソースと同じ結果になるプログラムは以下のように書けます。

```
((fn [x] (+ x 1))
 10)
```

```
11
```

そう、このコードブロックでみえたように `defn` の代わりに `fn` を使えば名前を付けずに関数を作ることができます。 このような関数は我々通常「無名関数」、 或いは「ラムダ関数」、「lambda関数」、「λ関数」と呼びます。

無名関数の面白い使いかたとして、以下のコードブロックをみてください。

```
(defn plus [x]
  (fn [y] (+ x y)))
((plus 7) 9)
```

```
16
```

少しややこしいプログラムになっていそうですが、 引数の部分適用としも理解でき

る使いかたなのでどこかで使い道があるだと納得できるのでしょう。

無名関数はよく作られていますので ClojureScript では無名関数を記述するための便利な文法があります。 例えば上の二つのコードブロックがしていたことをその文法で記述しますとこうなります。

```
(#(+ % 1) 10)
(defn plus [x] #(+ x %))
((plus 7) 9)
```

```
16
```

`#(+ % 1)`の部分を注目してみましょう。 この式は `#` で始まり、その直後に括弧式が続いています。 そしてその括弧式のなかには `%` が含んでいます。 ClojureScript ではこのような式は無名関数ショートハンド (anonymous function shorthand) としばしば呼ばれています。 `#` はこの文法の発動を意味していて `%` はホール、 つまりこの関数が取る引数を表します。

勿論、この文法を使って複数の引数を取るような関数も作れます。例えば

```
(#(/ (+ % %2) (- % %2)) 4 2)
```

```
3
```

のような関数も作れます。

最後に `defn` の仲間、`def` を紹介しておきましょう。 `defn` は前章で説明したように1.関数を作る 2.関数に名前を付ける といった二つのことをしていますが、 `fn` はその1.ができ、`def` はその2.ができます。 つまり `(defn square [x] (* x x))` は実は `(def square (fn [x] (* x x)))` とほぼ同じ意味を持ちます。

勿論 `def` は関数のほかいろんな値にも名前を付けることができます。 例えば以下のコードブロックでは10に `x`、20に `y` という名前を付けて計算を行いました。

```
(def x 10)
(def y 20)
(+ x y)
```

```
30
```

演習2.1a.

以下のプログラムにある `times` という関数の計算式を修正し、 `((times x) y)` のような式は正しく「x*y」を計算できるようにせよ。

```
(defn times [x]
  #(* x %))
```

```
#'cljs.user/times
```

演習2.1b.

以下のプログラムにある `curry` という関数の計算式を修正し、 任意の二つの引数を取る関数 `f` に対し `(curry f)` は `(= (f x y) (((curry f) x) y))` をなりたつようにせよ。

修正された `curry` を用いれば上に定義されていた `plus` 及び `times` はそれぞれ `(curry +)` と `(curry *)` で得られます。

但し無名関数ショートハンドの計算式の中では無名関数ショートは使用できないことに注意せよ。

```
(defn curry [f]
  (fn [x]
    (fn [y] (f x y))))
```

```
#'cljs.user/curry
```

§2.2. ベクター

今までみて来た ClojureScript の機能のみを使えば我々は一つの値しか表現できませんでした。 例えば演習1.3.では二次方程式の根は一般的に二つあるにも関わらずその内の一つを返すような関数でしか表現できなかった。 勿論コンピュータは複数の値を一気に扱うことができます。 ClojureScript では複数の値を一気に扱うに最もよく使われているものは「ベクター (vector)」と呼ばれているものです。 注意してほしいのは、ClojureScript でのこのベクターは数学や物理に出てくるベクターは違う概念であって、 いくつかの値の順番を考慮した集まりです。 実際「リスト (list)」と呼んだほうが適切かも知れませんが、 「リスト」という名前は別のものに取られてしまいましたので、 ベクターという名前になってしまっています。

ともあれベクターの例を見てみましょう。

```
[1 2 3]
```

```
[1 2 3]
```

これは三つの要素、1と2と3の順番で並んで構成されているベクターです。
ClojureScript では大括弧を使えばベクターをそのまま書けます。

ではベクターに対して我々は何ができるでしょう。 まず最も基本的なのはベクター

の長さが取れます。 そしてそれぞれの要素を場所を指定することで取りだすこともできます。

```
(def v [1 2 3 4])
[(count v) (get v 0) (get v 2) (get v 1)]
```

```
[4 1 3 2]
```

ここで一つ注意したいのは多くのプログラミング言語と同じように、 ClojureScript では場所の数えかたは0から始まっていることです。 又このように自然数で場所を表すようなものは「インデックス (index)」としばしば呼ばれています。

演習2.2.

以下のプログラムを後ろから二番目の要素を返す関数に修正せよ。

```
(defn rsecond [v]
  (get v (- (count v) 2)))
```

```
#'cljs.user/rsecond
```

演習2.3.

以下のプログラムを修正し、 二次方程式 $ax^2 + bx + c = 0$を解く関数を定義せよ。 但し実数根が存在しない場合空のベクター、 実数根が存在する場合長さ1か2のベクターで結果を返せ。

```
(defn delta [a b c]
  (- (* b b)
     (* 4 a c)))
(defn qsolve [a b c]
  (if (< (delta a b c) 0)
    []
    [(/ (+ (- b) (Math/sqrt (delta a b c)))
        (* 2 a))
     (/ (- (- b) (Math/sqrt (delta a b c)))
        (* 2 a))]))
```

```
#'cljs.user/qsolve
```

§2.3. ベクターに関する操作

ベクターのように複数の要素を同時に扱うためのものは一般的に「コンテナ (container)」、 また ClojureScript では「シーケンス (sequence)」と呼ばれています。

シーケンスは複数の要素を同時に扱うものなので、 `get` のようなもので要素を一つずつ処理することは多くありません。

要素を同時に扱う手法としては `filter`、 `map`、 `reduce` といった関数がよく使われています。 これらの使いかたを演習を通して覚えておきましょう。

演習2.4a

`map` 関数はシーケンスの要素一個ずつ変更する関数です。 例えば

```
(map #(+ % 1) [1 2 3])
```

```
(2 3 4)
```

以下のプログラムを修正し、平方数を枚挙する関数を作れ。 但し `(range n)` はn未満の自然数を枚挙する関数です。

```
(defn range-squares [n]
  (map #(* % %) (range n)))
```

```
#'cljs.user/range-squares
```

演習2.4b

`filter` 関数はシーケンスの要素を述語によって選別する関数です。 例えば

```
(filter #(> % 0) [-1 2 3 -4 7])
```

```
(2 3 7)
```

以下のプログラムを修正し、偶数である平方数を枚挙する関数を作れ。 但し `(mod 9 4)` のように「9割る4の余り」を計算することができます。

```
(defn range-even-squares [n]
  (map #(* % %)
       (filter #(= (mod % 2) 0)
               (range n))))
```

```
#'cljs.user/range-even-squares
```

演習2.4c

`reduce` 関数はシーケンスの要素を一個ずつ「潰す」関数です。 例えば `(reduce + [1 2 3 4])` は `(+ (+ (+ 1 2) 3) 4)` と同じ意味になります。 又 `(reduce + 1000 [1 2 3 4]` は `(+ (+ (+ (+ 1000 1) 2) 3) 4)` と同じ意味となるように最初の値を指定することもできます。

以下のプログラムを修正し、階乗を求める関数を作れ。

```
(defn factorial [n]
  (reduce * (map #(+ % 1) (range n))))
```

```
#'cljs.user/factorial
```

演習2.4d

以下のプログラムを修正し、シーケンスにある最大値を返す関数を作れ。

```
(defn findmax [s]
  (reduce #(if (> %1 %2) %1 %2) s))
```

```
#'cljs.user/findmax
```

演習2.4e

以下のプログラムを修正し、`reduce` を用いてシーケンスの要素数を求める関数を作れ。

```
(defn mycount [s]
  (reduce #(+ % 1) 0 s))
```

```
#'cljs.user/mycount
```