Chapter 3 (NUPSC 2018 講習資料)

§3.1. ループ

演習2.4cでは我々は range、 reduce などで階乗を計算するプログラムに挑んでいました。 例えばこのようなプログラムが階乗を計算することができます。

```
(defn factorial [n]
  (reduce * (filter #(> % 0) (range (+ n 1)))))
```

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

この解法は自然数を枚挙してから掛けあわせています。 そして自然数の枚挙は range という関数を用いていました。 実際これよりももっと直接な解法が存在しています。 このプログラムをみてみよう。

```
(defn factorial [n]
  (loop [m    1
         prod 1]
    (if (⩽ m n)
      (recur (+ m 1) (* prod m))
      prod)))
```

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

このコードブロックも階乗の関数を定義しています。 このプログラムを理解するにはまず loop/recur の説明をしないといけません。 loop は単語的に元々「輪」の意味ではありますが、 計算機科学では「繰返し」という意味でしばしば使っています。 日本語では「ループ」ともしばしば呼ばてれいます。 そして recur は単語的な意味はまさに「繰返し」になっています。 ClojureScript では loop と recure はお仲間で、いつでも一緒に現れます。 厳密にいいますと、 recur は loop で始まる式の中にいつも現れます。

ClojureScript では loop は「繰返しの基点」を決めて、 recur が呼ばれた時点、 その loop で決められた基点に戻るという決った動きになります。 そして loop の直後に続く大括弧の中身は、変数の宣言です。 例えば上の例では、 m と prod という二つの変数を宣言しています。 なぜ m と prod を変数と呼ぶかといいますと、 recur を呼んだ時点、 m と prod はそれぞれ違う数となって計算が繰返されるからです。 上の例では、繰返しが行われる度(つまり recur が呼ばれる度)、 m は (+ m 1) になり、 prod は (* prod m) になります。 このような繰返しはプログラムにあるif文の条件式で指定されているように、 n が m を越える時点で止まり、 loop から始まる式の全体の結果は、 その回の繰返しにおいての prod の値になります。 一つ説明が漏れていたのは

loop の直後に現れる変数の宣言では、 初回の繰返しの時 m と prod それぞれの値が指定されていることです。 上の例では m も prod も1に指定されていることが分かります。

例えば上のコードブロックで定義されている factorial を (factorial 4) で呼んでみると、毎回の繰返しにおける変数の値はこのように変化していくことが分かるでしょう。

```
[回数]  (n) m         prod
  0      4  1          1
  1      4  1+1 = 2    1*1 = 1
  2      4  2+1 = 3    1*2 = 2
  3      4  3+1 = 4    2*3 = 6
  4      4  4+1 = 5    6*4 = 24
    (<= m n)はfalseになりましたので、繰返しは終了します。
```

loop の使いかたはこうです

```
(loop [変数1 初期値1
       変数2 初期値2
       変数3 初期値3
       ...]
  繰返し計算される式)
```

その繰返し計算される式をボディーと呼びましょう。 但しボディーには recur が使われていて、 (recur v1 v2 v3 ...) のような式がありましたら 変数1、 変数2、 変数3 などは それぞれ v1、 v2、 v3 に更新され、 ボディーがもう一回実行されます。

演習3.1a

以下のプログラムを修正し、 reduce を使わず loop/recur を用いて引数nまでの偶数の全体の和を計算する関数を定義せよ。

```
(defn sum-even [n]
  (loop [m   0
         sum 0]
    (if (> m n)
      sum
      (recur (+ m 1)
             (if (= (mod m 2) 0)
               (+ sum m)
               sum)))))
```

```
#'cljs.user/sum-even
```

演習3.1b

以下のプログラムを修正し、 reduce を使わず loop/recur を用いて数のシーケンスで

ある引数vの要素全体を足し合せた和を計算する関数を定義せよ。 但し `(first [3 2 1 4])` は `3` となり、 `(rest [3 2 1 4])` は `(2 1 4)` となっているように、 `first` と `rest` でシーケンスの先頭要素とそれを除外したシーケンスが取れます。 又 `(empty? [])` は `true` となるように `empty?` を用いればシーケンスは空であるかを判断することができます。

```
(defn sum [s]
  (loop [l   s
         sum 0]
    (if (empty? l)
      sum
      (recur (rest l) (+ sum (first l))))))
```

```
#'cljs.user/sum
```

演習3.1c

以下のプログラムを修正し、 `reduce` を使わず `loop/recur` を用いて数のシーケンスである引数vの要素全体を掛け合せた積を計算する関数を定義せよ。

```
(defn prod [s]
  (loop [l s
         prod 1]
    (if (empty? l)
      prod
      (recur (rest l) (* prod (first l))))))
```

```
#'cljs.user/prod
```

§3.2. 状態と遷移

前節では我々は `loop/recur` についてみてきました。 つまり、繰返しについてみてきました。 繰返しを行う上、「変数」という概念も導入しました。 「変数」という言葉の意味は「変動する数」の意味であって、 「数」というのは数学用語の名残で、実際プログラミングでは値を意味していまして、 では「変動」というのはどういう状況でしょうか。 `loop/recur` では変動の概念を理解するにはそう難しくはない。 何故ならば繰返しには「何回目」という概念があるからです。 つまり次の「回」は前の「回」と違うものが変数となる訳であって、 その変ったことは「変動」となります。

この変動という概念は実際繰返しとは又独立しているものであって、 *何か*は時間などに従って*変化*することがしばしばあります。 例えば時計の秒針は一秒ごとひと単位進んでいます。 このように変動するものは計算科学ではしばしば「状態」と呼んでいます。 そして状態が変わることは「状態遷移」と呼んでいます。

状態とその遷移の仕方を表すには関数がしばしば使われています。 例えば秒針の例、0-59の自然数のいずれを秒針の状態としますと、 その状態遷移は以下の関数で表わすことができます。 又このような状態遷移の仕方を表わしている関数は「状態遷移関数」と呼ばれていて、 可能な全ての状態は「状態空間」だと呼ばれています。

```
(defn fsec [prev]
  (if (< prev 59)
    (+ prev 1)
    0))
```

```
#'cljs.user/fsec
```

この関数は遷移の前の状態(prev、previous の略)を受け取り、 遷移後の状態を返しています。

初期状態を指定して遷移関数を使ってみましょう。

```
(defn fsec [prev]
  (if (< prev 59)
    (+ prev 1)
    0))
(def init1 3)
(def init2 58)
[[(fsec init1)
  (fsec (fsec init1))
  (fsec (fsec (fsec (fsec init1))))]

 [(fsec init2)
  (fsec (fsec init2))
  (fsec (fsec (fsec (fsec init2))))]]
```

```
[[4 5 7] [59 0 2]]
```

このように状態遷移関数を複数回適用することで、 秒針の進行をシミュレートすることができます。

秒針の進行のシミュレーションはあっけないですが、 最近は天気予報などのため、地球にまとう大気全体をたくさんの枠に分けて、 それぞれの枠にある大気の性質(例えば気圧、気温、湿度など)のすべてのバリエーションを状態空間として大気の動きをシミュレートしています。 スケールが全く違うのですが、上の秒針の進行シミュレーションの例とは全く同じ原理的に基いています。 大気のような大規模のシミュレーションを行うには莫大な計算を行う必要があります。 それは人の手でどうしても間に合わないものであって、 コンピューター技術があってからできるものです。 そしてそのようなことはまさにプログラミングを習得しないとできないことであり、状態遷移の考えかたも、プログラミングをしてみないとなかなか身にしみらないものではないでしょうか。

演習3.2a

上では例えば4回秒針進行の遷移関数を適用するには `(fsec (fsec (fsec (fsec init))))` を書いていましたが、 以下のプログラムを修正して、 `((n-times 4 fsec) init)` で同じことができるような `n-times` という関数を定義せよ。

```
(defn n-times [n f]
  (fn [prev]
    (loop [m 0
           acc prev]
      (if (≥ m n)
        acc
        (recur (+ m 1) (f acc))))))
```

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

演習3.2b

以下のプログラムを修正し、 `(until pred f)` は「引数として受け取る状態が pred という述語が満せるまで状態遷移 f を適用しつつける関数」を意味するように関数 `until` を定義せよ。例えば `((until #(= (mod % 7) 2) fsec) 32)` は37になる。

```
(defn until [pred f]
  (fn [prev]
    (loop [acc prev]
      (if (pred acc)
        acc
        (recur (f acc))))))
```

```
#'cljs.user/until
```

演習3.2c

コラッツの予想 という数学の未解決問題があります。 以下のコードブロックにある `fcollatz` をコラッツの予想における数にたいする操作と一致した状態遷移関数に修正せよ。 又 `(until #(= % 1) fcollatz)` を用いてコラッツの予想の正しさを引数nまで検証する関数 `check-collatz` を補完せよ。

```clojure
(defn fcollatz [prev]
  (if (= (mod prev 2) 0)
    (/ prev 2)
    (+ (* prev 3) 1)))
(defn check-collatz [n]
  (loop [m 1]
    (if (≥ m n)
      true
      (if (= ((until #(= % 1) fcollatz) m) 1)
        (recur (+ m 1))
        false))))
```

```
#'cljs.user/check-collatz
```