リファクタリングにむけて

はじめに (2007-12-19)

Scheme 初心者まるだしの、恥ずかしいコードをゼッサン拡大中の UikiTeXi ですが、少しずつでもリファクタリングしていきたい。

コードが見苦しくなってしまう原因には色々あると思いますが、個人的には、言語仕様の理解があやふやで、冗長で無駄な記述をしてしまうケースが少なくないように思います。

どんどん機能を追加していくぜ〜というときに、「まともな」書き方がわからなくても手を止めずに、自分なりに安全だと思う[*1]書き方で書いてしまうことは、悪いことではないと思っています。

でも、そういうのは、少しずつ理解を深めていくことにより、減らしていきたいのです。

letrec なんてのがあるのね (2007-12-19)

let や let* は最近使えるようになったんだけど、再帰的な関数内関数と同居するにはどうしたらいいの症候群。

仕方がないので、現状の僕のコードは次のようになっています：

```
(define (count-ten)
  (define max-iteration 10)

  (define (lp-count n)
    (cond
     ((> n 0)
      (display n)
      (lp-count (- n 1)))))

  (lp-count max-iteration))
```

まあ、これくらいの規模だからいいようなものの、気持ち悪いです。実際、現状の UikiTeXi コード中では、かなり見苦しいことになっていると思います。

次のような書き方はたぶんだめだろうなぁと感じてのことだったのですが、やっぱりだめみたい。

```
(define (count-ten2)
  (let ((max-iteration 10)
        (lp-count
         (lambda (n)
                 (cond
                  ((> n 0)
                   (display n)
```

[*1] こう書いておけば、ともかく期待どおり動くだろうという意味で

```
                    (lp-count (- n 1))))))) ; *** ERROR: unbound variable: lp-count
```
になるよ

```
    (lp-count max-iteration)))
```

むーん、次のように書けということか？まあ、悪くはないですね。

```
(define (count-ten3)
  (let ((max-iteration 10))
    (define (lp-count n)
      (cond
       ((> n 0)
        (display n)
        (lp-count (- n 1)))))

    (lp-count max-iteration)))
```

で、さっき調べて知ったのが letrec です。こんな感じ。

```
(define (count-ten4)
  (letrec ((max-iteration 10)
           (lp-count
            (lambda (n)
              (cond
               ((> n 0)
                (display n)
                (lp-count (- n 1)))))))
    (lp-count max-iteration)))
```

let* のような参照はできないようだ。
むむ、名前付き let とはなんぞ？

#f 以外は真である件 (2007-12-19)

たとえば、x に何らかの文字列が入っているケースと、そうでなければ #f が入ってますという場合、なんとなく不安だった僕は

```
(cond
 ((string? x)
  ...))
```

みたいに書いてました。もしかするともしかするな〜とか思いつつ[*2]。

[*2] Ruby ユーザーですからっ

もしかしましたね。

```
(define (check x)
  (cond
   (x    (display "true\n"))
   (else (display "false\n"))))

(check #t) ; => true
(check #f) ; => false

(check 3.14)          ; =>?
(check "some string") ; =>?
```

下ふたつが不安だったのが原因だったのですが、いずれも true となります。

R6RS における boolean の説明[*3]では、

A boolean is a truth value, and can be either true or false. In Scheme, the object for “false” is written #f. The object for “true” is written #t. In most places where a truth value is expected, however, any object different from #f counts as true.

と書かれています[*4][*5]。

[*3] http://www.r6rs.org/final/html/r6rs/r6rs-Z-H-4.html#node_sec_Temp_6

[*4] R5RS にも同様の記述があるはずですが、まじめに探してない。

[*5] UikiTeXi が blockquote を未サポートなのは、見ての通りであるが、内緒だ。