Jacques Garrigue, 2007 年 10 月 24 日

再帰関数プログラミング 1

6 反復と再帰

再帰関数の定義

自明でないプログラムを書くにはループ (反復) が必要である．しかし，for ループを使うときには，可変な変数が必要になり，プログラムの (理論的な) 解釈が複雑になる．再帰関数はもう一つのループの書き方を与えてくれる．

```
# let rec fact n =                                    (* let rec を使う *)
    if n = 0 then 1 else n * fact (n-1) ;;
val fact : int -> int = <fun>
# fact 5;;
- : int = 120
```

let rec というキーワードは再帰的な定義を表している．定義の中で定義しようとする関数を使ってもいい．分岐の中に再帰呼び出しを入れれば，実行の回数が制御できるわけだ．

#trace というコマンドで実際の呼び出しを見ることができる．

```
# #trace fact;;
fact is now traced.
# fact 3;;
fact <-- 3
fact <-- 2
fact <-- 1
fact <-- 0
fact --> 1
fact --> 1
fact --> 2
fact --> 6
- : int = 6
# #untrace fact;;
fact is no longer traced.
```

ちなみに，定義しようとする関数が見えているので，**and** を使うと相互再帰になる．

```
# let rec even n = if n = 0 then true else odd (n-1)
  and odd n = if n = 0 then false else even (n-1) ;;
val even : int -> bool = <fun>
val odd : int -> bool = <fun>
# even 3;;
- : bool = false
# odd (-1);;                                  (* 無限ループに陥いる *)
^CInterrupted.                                (* キーボードで Ctrl-C *)
```

練習問題 6.1 以下のものを計算する再帰関数を定義せよ

1. 1 から n までの合計 $(\sum_{i=1}^{n} i)$

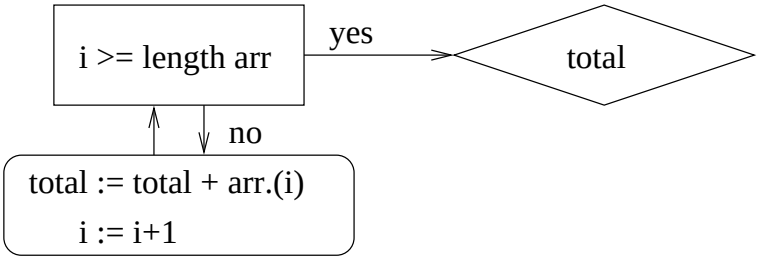
2. 高速乗算法を行う関数．x と n が与えられたとき，もしも n が偶数だったら，$x^{2n} = x^n \cdot x^n$，奇数だったら $x^{2n+1} = x^n \cdot x^n \cdot x$．プログラムでは x は `float` で，n は `int` にする．

反復の再帰への翻訳

さて，今まで for ループで定義していたものは再帰でも定義できるか？最初は前に見た配列の合計からやって見よう．

```
# let sum4 (arr : int array) =
    let rec loop i total =                          (* 元の sum のループの直訳 *)
      if i >= Array.length arr then total
      else loop (i+1) (total + arr.(i))
    in loop 0 0 ;;
val sum4 : int array -> int = <fun>
# sum4 [|1;2;3|];;
- : int = 6
```

内部で変更する全ての値を引数とする再帰関数 `loop` を定義すればいい．その値を変えてループを続けるときは `loop` を新しい値で呼び，終るときは最終的に返したい値だけを返せばいい．以下の図ではその処理を示す．



for ループや while ループで表現されるのは図の左の部分だけですが，再帰関数では結果を返すところまで含まれる．

さらに，再帰関数だと，戻りながらも計算できる．

```
# let sum5 (arr : int array) =
    let rec loop i =                                (* 引数が i のみ *)
      if i < 0 then 0 else loop (i-1) + arr.(i)
    in loop (Array.length arr - 1) ;;
val sum5 : int array -> int = <fun>
# sum5 [|1;2;3|];;
- : int = 6
```

こちらのプログラムを for ループで表現することが難しい．`loop i` の結果は配列の `i` 番目までの合計だが，処理として一旦 `i` を `Array.length arr - 1` から`-1` まで下げて行って，今度は戻りながら配列の要素を結果に足していく構造だ．

末尾再帰

`sum4` が一対一でループを使ったプログラムに対応しているのに，`sum5` はそういう対応ができない．こういうときは，`sum4` の `loop` 関数を末尾再帰といい，`sum5` のは末尾再帰でないという．区別する方法は，再帰呼び出しの後に計算が残っているかどうかだ．計算が残らない場合は OCaml が再帰呼び出しを末尾呼び出しと認識し，ジャンプとしてコンパイルされるので，結果的に for ループと全く同じコードが生成される．効率の面で末尾再帰が有利なときが多いが，そのためにプログラムが分かりにくくなったり，他のところで非効率になることもありうるので，常に末尾再帰にこだわる必要がない．

反復の抽象化

iter は配列の各要素にある関数を適用することで反復を抽象化しているが，iter を使うときは可変な変数を使わなければならない．下記のように，可変な変数を全く使わない汎関数も定義できる．

```
# let fold f init arr =
    let len = Array.length arr in
    let rec loop i r =
      if i >= len then r else loop (i+1) (f r arr.(i))
    in loop 0 init ;;
val fold : ('a -> 'b -> 'a) -> 'a -> 'b array -> 'a = <fun>
# let sum6 = fold (+) 0 ;;
val sum6 : int array -> int = <fun>
# sum6 [|1;2;3|];;
- : int = 6
```

練習問題 6.2 *1.* fold を使って，配列の中の最大値を求めよ．

2. fold と^を使って文字列の配列を空白で区切られた文字列に変えよ．

```
concat [|"Little"; "brown"; "fox"|] --> "Little brown fox"
```

再帰関数と帰納法

最大公約数の計算を見る．

```
# let rec gcd m n =                                    (* let rec は再帰的な定義 *)
    if n = 0 then m else gcd n (m mod n) ;;
val gcd : int -> int -> int = <fun>
# gcd 15 70;;
- : int = 5
```

もしも同じプログラムを while ループで書いたら，大分長くなる．

```
# let gcd2 m n =
    let m = ref m and n = ref n in
    while !n <> 0 do                                        (* while ループ *)
      let n' = !m mod !n in                               (* 名前は n' でもいい! *)
      m := !n; n := n'                                    (* 同時代入ができない *)
    done;
    !m ;;
val gcd2 : int -> int -> int = <fun>
```

しかし，再帰の最大のメリットは短かさではなく，証明しやすさである．再帰関数は帰納法と同じ構造をしているので，帰納法により証明が簡単にできる．gcd の場合では ($m, n \geq 0$ と仮定して)

- $n = 0$ ならば，0 はどんな自然数でも割れるが，m を割る最大の自然数は m 自身である．ゆえに m と n の最大公約数は 0 である．
- $n > 0$ ならば，任意の $k < n$ と任意の m に対して，gcd m k が m と k の最大公約数であると仮定する．
  m と n の最大公約数は m **mod** n を割らなければならない．逆に，q が n と m **mod** n を割っていれば，m も q で割れる．ゆえに m と n 最大公約数は n と m **mod** n の最大公約数でもある．$0 \leq m$ **mod** $n < n$ がなりたつので，帰納法の仮定が適用できて，(m と n の最大公約数) = (n と m **mod** n の最大公約数) = gcd n (m **mod** n) = gcd m n.

練習問題 6.3 sum5 が正しく配列の中身の合計を計算していることを証明せよ.

線形でない再帰

再帰では，単純なループで表現できない計算の構造も表現できる.

```
# let rec fib n = if n < 2 then 1 else fib (n-1) + fib (n-2) ;;
val fib : int -> int = <fun>
# fib 5;;
- : int = 8
```

しかし，こういうものは必ずしも効率よくない.

```
# #trace fib;;                                          (* 呼び出しを表示させる *)
fib is now traced.
# fib 4;;
fib <-- 4
fib <-- 2
fib <-- 0
fib --> 1
fib <-- 1                                               (* fib 1 が呼ばれる *)
fib --> 1
fib --> 2
fib <-- 3
fib <-- 1                                                 (* ここも *)
fib --> 1
fib <-- 2
fib <-- 0
fib --> 1
fib <-- 1                                                 (* ここも *)
fib --> 1
fib --> 2
fib --> 3
fib --> 5
- : int = 5
```

練習問題 6.4 fib のもっと賢い書き方があるはず．補助定義を使ってもいい.
ヒント：補助関数が fib(n) と fib($n-1$) の組みを返せばいい.