

ソフトウェア技法: No.3 (自然数上の再帰関数)

亀山幸義 (筑波大学情報科学類)

1 再帰関数 (recursive function) とは

関数型プログラム言語では、繰返しを表現するために、再帰を使うことが多い。(一方、C言語などでは、for や while などのループ構造を用いることが多い。)再帰 (recursion) は、関数定義においては「定義の中で自分自身を呼びだすこと」である。関数の再帰的定義の他にも、型の再帰的定義 (この授業の後半で説明) や、論理式の再帰的定義などがある。ラムダ計算 (lambda calculus) の基礎理論では、ラムダ式をつかって再帰的関数を表すことができるという定理 (再帰定理) が極めて重要な役割を果たしている。

ここでは、自然数を対象とした再帰的な関数^{*1}を定義する方法を、例を通して学ぶことにしよう。

```
(* 再帰しない関数 *)
let f n = n + 1 ;;
let _ = f 5 ;;
let _ = f 10 ;;
(* ここで使った _ は、変数みたいなものだが、その値を、あとで使わない場 *)
(* 合に使うパターンである。underscore と呼ぶ。 *)

(* 次のものは再帰関数のつもりだったが、書き損なったもの *)
let f n =
  if n = 0 then 0
  else f(n-1) + n in
  f 5      ;;
let _ = f 5 ;; (* 期待した結果が得られない *)

(* 再帰関数は let rec で始める *)
let rec f n =
  if n = 0 then 0
  else f(n-1) + n ;;

let _ = f 5 ;; (* 1から5までの総和が得られる *)
```

1.1 第一歩

再帰で関数を書く、ということは、数学で言うところの「漸化式」を書くことに類似している。まず、これを理解しよう。

「1からnまでの総和」を求めたいとする。その一歩手前である「1からn-1までの総和」が計算できているとして、その計算結果を用いて、「1からnまでの総和」をどのように計算すればよいか考える。

そうすると、以下の式に思いあたるであろう。

$$\text{「1からnまでの総和」} = \text{「1からn-1までの総和」} + n$$

これはほぼ正しいが、そのままプログラム化すると、うまく行かない。

^{*1} 厳密に言えば、ここで言う「再帰関数」は、数学的には、関数とは限らず部分関数であるので、「再帰部分関数」というべきであるが、プログラム言語では、数学的な部分関数の意味で「関数」と呼んでしまうので、ここでもそれにならって再帰関数と呼ぶ。

```
let rec sum_buggy n =
  (sum_buggy (n-1)) + n ;;
```

この関数 `sum_buggy` は、どんな引数に対しても停止しない関数になってしまう。
漸化式の時と同様、 $n = 0$ のときを別扱いにすると良さそうだと気付く。

($n = 0$ のとき) 「1 から n までの総和」 = 0

($n > 0$ のとき) 「1 から n までの総和」 = 「1 から $n-1$ までの総和」 + n

これを、OCaml プログラムとして書くと以下のようになる。

```
let rec sum n =
  if n = 0 then 0
  else (sum (n-1)) + n ;;
```

これで OK である。試しに、`(sum 5)` の計算を、手動と OCaml とでやってみてほしい。

1.2 第二步

次に、「 m から n までの総和」を求める関数を書いてみよう。今度は、関数の引数が m と n の 2 つになるので、漸化式を作ろうとすると、 m と n のどちらから 1 を引いたケースを考えるかで、2 通りの選択肢がある。(場合によっては、 m と n の両方とも 1 を引いたケースを考えることもある。)

先ほどの問題と同様に考えれば、 n を 1 つ減らしたケース、つまり、「 m から $n-1$ までの総和」を使えばよさそうだと気付く。

($m > n$ のとき) 「 m から n までの総和」 = 0

($m \leq n$ のとき) 「 m から n までの総和」 = 「 m から $n-1$ までの総和」 + n

OCaml プログラムとして表現すると以下の通り。

```
let rec sum_fromto m n =
  if m > n then 0
  else (sum_fromto m (n-1)) + n ;;
```

これで、きちんと動作する。試しに、`(sum_fromto 2 5)` と `(sum_fromto 5 2)` を手動と OCaml プログラムの両方で計算してみなさい。

1.3 第三步

今度は、 n が素数であるかどうかを判定する関数を実装しよう。判定する関数というのは (1) n が素数なら `true` を返し、(2) n が素数でないなら `false` を返す、という振舞いをする関数を意味する。なお、C 言語等では、`true/false` は `1/0` であらわすが、OCaml では `true/false` と `1/0` は別なので、注意されたい。

さて、「 n が素数であることを判定」する関数の漸化式を書けるだろうか？ まず、考えつくのは、「 $n-1$ が素数であることの判定結果」を使って、「 n が素数であることの判定結果」を作ろうとする方法である。

「 n が素数である」 = 「 $n-1$ が素数である」が真のとき …。

これは、できそうもない。 $n-1$ が素数かどうかわかったところで、 n が素数かどうかはわからない。($n-1$ が素数でないとき、 n が素数かどうかはまったくわからない。どちらも十分あり得る。)

そこで、問題を少し変形して、「 n が、 $2,3,4,\dots,k$ のどの数でも割り切れない」ことを判定してみよう。仮に、この判定問題を計算する関数があったとすると、 $k=n-1$ の場合の判定結果が、 n が素数であるかどうかの判定結果と一致することがわかる。

漸化式の考え方でいえば、「 n が、 $2,3,4,\dots,k-1$ のどの数でも割り切れない」という判定結果を使って、「 n が、 $2,3,4,\dots,k$ のどの数でも割り切れない」ことを判定できればよい。これは簡単である。

「 n が、 $2,3,4,\dots,k$ のすべてで割り切れない」 = 「 n が、 k で割り切れない」かつ「 n が、 $2,3,4,\dots,k-1$ のすべてで割り切れない」

うまく漸化式で書くことができた。

ただし、このままだと、また、停止しない関数になってしまうので、一番簡単なケース (base case) だけ特別扱いにしよう。この場合、 k がどんどん小さくなっていくので、一番簡単なケースは、 $k=1$ である。このとき、 n は「 n が、 $2,3,4,\dots,k$ のすべてで割り切れない」というのは、true である。(0 個の論理式の論理積は真である。)

($k=1$ のとき) 「 n が $2,3,4,\dots,k$ のすべてで割り切れない」 = true

($k>1$ のとき) 「 n が $2,3,4,\dots,k$ のすべてで割り切れない」 = 「 n が k で割り切れない」かつ「 n が $2,3,4,\dots,k-1$ のすべてで割り切れない」

そこで、この漸化式に基づいて、OCaml 関数を書いてみよう。この判定を行う関数の名前を `is_prime_like`(素数みたいなもの) とする。

```
let rec is_prime_like n k =
  if k < 2 then true
  else
    (n mod k <> 0) && (is_prime_like n (k-1)) ;;
```

ここで `&&` は、論理記号の「かつ」を表す。また、 $n \bmod k \neq 0$ は「 n を k で割ったときの余りが 0 でないこと、つまり、 n が k で割り切れないこと」を意味する。

関数 `is_prime_like` が定義できると、素数であるかどうかの判定は簡単にできる。すなわち、 n が素数であるとは、 n が $2,3,\dots,n-1$ で割り切れないことである。

```
let rec isprime n =
  is_prime_like n (n-1) ;;
```

上記の関数 `isprime` そのものは再帰呼び出しを使っていないので (定義の右辺に `isprime` が出てこない)、`let rec` でなくて `let` で定義して構わない。その方が「再帰関数でない」ことが明示できるので、普通はそのように書く。

```
let isprime n =
  is_prime_like n (n-1) ;;
```

1.4 第四步: パラメータを増やしていくパターン

今度は、「 n 以上の最小の素数を求める」関数を書いてみよう。まず、漸化式で書けるだろうか？

「 n 以上の最小の素数」 = 「 $n-1$ 以上の最小の素数」が m のとき、...

これは明らかにうまくいかない。「 $n-1$ 以上の最小の素数」をもとめるために、「 n 以上の最小の素数」を使うことはあっても、逆はないからである。もっと端的に言えば、「 $n-1$ 以上の最小の素数」が $n-1$ そのものだったとき、まったく役に立たない情報である。

(そんな計算をしても「 n 以上の最小の素数」はまったくわからない。)

そこで、人間がどうやって「 n 以上の最小の素数」を計算するか考えてみると、まず、 n が素数かどうか試して、それが素数でなければ、次に $n+1$ を試すであろう。つまり、 $n-1$ でなく $n+1$ を見にいけば良い。これは、普通の意味での「漸化式」ではないが、漸化式風だと思って書くと以下ようになる。

「 n 以上の最小の素数」 = if (n が素数) then n else 「 $n+1$ 以上の最小の素数」

これなら、OCamlプログラムにできそうである。

```
let rec least_prime n =
  if (isprime n) then n
  else least_prime (n+1) ;;
```

これでうまく。

このプログラムは、再帰呼び出しをするごとに、引数 n の値がどんどん大きくなっていくので、(プログラムの定義だけを見ると)止まらないことがあるように思える関数である。

しかし、我々は、「いくらでも大きい素数がある(素数は無限にある)」ということを知っているので、その事実に照らすと、上記の`least_prime`関数に、どんな n を与えても、いつか必ず停止して n 以上の素数を返すことがわかる。

このように、再帰呼び出しは、数学的に漸化式であらわされるもの以外(引数が減っていくとは限らない場合)も記述できるパワーをもっている。

1.5 第五歩

最後に、「 n の約数のうち、 n 自身以外で最大のもの求める」関数を書いてみよう。漸化式で書けるだろうか？

「 n の約数のうち、 n 自身以外で最大のもの」
= 「 $n-1$ の約数のうち、 $n-1$ 自身以外で最大のもの」。。。。

うまく書けない。 n の約数であることと $n-1$ の約数であることはほぼ無関係なので、 $n-1$ の約数の計算ができたところで、それを使って n の約数を計算することはできそうもない。

人間だったら、どう計算するか考えてみると、一番素朴な方法は、(1) n を $n-1$ で割ってみる (2) n を $n-2$ で割ってみる、(3) n を $n-3$ で割ってみる、...を繰り返して最初に割り切れた数を見つけることである。つまり、 n を固定した上で、それ以外に、 $n-1, n-2, n-3, \dots, 1$ という範囲を動く変数を持たせると良さそうである。つまり、この部分処理する関数は、 n 以外にもう1つ引数があるとよい。

このように、「最終的に作りたい関数」が1引数なのに、途中の段階では2個以上の引数をもつ関数を使いたい場合は、まず、途中段階の処理をする関数を定義し、次に、最終的に作りたい関数を定義するとよい。このような「途中段階の処理をする関数」を補助関数と言う。

まず、補助関数を定義しよう。 n 以外にもう1つ引数を持つ。

「 n の約数のうち m 以下で最大のもの」
= if 「 n が m で割り切れる」 then m
else 「 n の約数のうち $m-1$ 以下で最大のもの」

この関数も、計算が止まるかどうか、ちょっと心配だが、 m が1まで減ったら、(n は必ず1で割り切れるので)そこで停止するので問題ない。

OCamlではこう書ける。

```

let rec largest_factor_upto n m =
  if (n mod m = 0) then m
  else largest_factor_upto n (m - 1) ;;

```

ここで mod は、整数上の割り算における「余り」を計算する関数である。たとえば、 $13 \bmod 5$ は 3 である。

上記の補助関数を使って、最初の「 n の約数のうち、 n 自身以外で最大のものを求める」関数は以下のように書ける。

```

let largest_factor n =
  largest_factor_upto n (n-1) ;;

```

今回も再帰関数ではないので、rec をつけていない。また、よく考えると、 n の約数で n 自身以外のものは、 $n/2$ 以下であるので、上記の計算をはじめるのは $n-1$ からでなく、 $n/2$ からで十分である。

```

let largest_factor n =
  largest_factor_upto n (n/2) ;;

```

1.6 再帰関数いろいろ

いろいろな関数の定義を見てみよう。それぞれの関数の定義を読み、また、実際に走らせて、再帰関数への理解を深めてほしい。

```

(* 1 から n までの積、つまり n の階乗 *)
let rec factorial n =
  if n = 0 then 1
  else factorial (n-1) * n in
  factorial 5 ;;

```

```

(* 1 から n までの二乗の和 *)
let rec square_sum n =
  if n = 0 then 0
  else square_sum(n-1) + n * n in
  square_sum 5 ;;

```

```

(* 1 から n までの奇数の和 *)
let rec odd_sum n =
  if n = 0 then 0
  else if n mod 2 = 0 then odd_sum (n-1)
  else odd_sum (n-1) + n in
  odd_sum 5 ;;

```

```
(* もっと効率よく書ける *)
let rec odd_sum n =
  if n = 0 then 0
  else if n = 1 then 1
  else if n mod 2 = 0 then odd_sum (n-1)
  else odd_sum (n-2) + n in (* 1つ前の偶数は飛ばす*)
  odd_sum 5 ;;
```

```
(* フィボナッチ関数 *)
let rec fib x =
  if x <= 2 then 1
  else (fib (x - 2)) + (fib (x - 1)) ;;
```

```
(* 最大公約数を求めるユークリッドの互除法 *)
let rec gcd x y =
  if x = 0 then y
  else if y = 0 then x
  else if x > y then gcd (x-y) y
  else gcd (y-x) x ;;
```

2 演習問題

以下の問題のうち、3問以上に解答せよ。

- 正整数 n の「整数上の平方根」を求める関数 $\text{isqrt } n$ を (整数上の関数のみを使って) 定義せよ。
例: $\text{isqrt } 16 = 4$, $\text{isqrt } 30 = 5$
- 正整数 n と k が与えられたとき、 n の k 乗を求める関数 $\text{power } n \ k$ を定義せよ。
例: $\text{power } 2 \ 5 = 32$, $\text{power } 3 \ 4 = 81$
- 正整数 n と 2 以上の整数 k が与えられたとき、 n が k で何回割り切れるかを求める関数 $\text{factor } n \ k$ を定義せよ。
例: $\text{factor } 80 \ 2 = 4$, $\text{factor } 80 \ 3 = 0$, $\text{factor } 80 \ 5 = 1$
- 与えられた正の整数 n の相異なる素因数の個数を求める関数 $\text{prime_factor } n$ を定義せよ。(素因数とは、 n の約数のうち、素数となっているもののこと。)
例: $\text{prime_factor } 20 = 2$ (20 の素因数は 2 と 5 なので)
- 与えられた正の整数 n が、2 進数であらわすと何 bit になるか求める。(log の関数を使うとすぐ計算できてしまうので、ここでは、それを使わず、整数上の演算だけで求めること。)
- 竹内関数 (tarai 関数) は、以下のように定義される関数である。

$$\text{tarai}(x, y, z) = \begin{cases} y & \text{if } x \leq y \\ \text{tarai}(\text{tarai}(x-1, y, z), \text{tarai}(y-1, z, x), \text{tarai}(z-1, x, y)) & \text{if } x > y \end{cases}$$

これをそのまま実装するのはあまりに簡単であるので省略し、ここでは実行中に起きる関数呼び出しの回数を計算する関数

を作りなさい。たとえば、 $\text{tarai}(3,2,1)$ は

$$\begin{aligned}\text{tarai}(3, 2, 1) &= \text{tarai}(\text{tarai}(2, 2, 1), \text{tarai}(1, 1, 3), \text{tarai}(0, 3, 2)) \\ &= \text{tarai}(2, 1, 3) = \text{tarai}(\text{tarai}(1, 1, 3), \text{tarai}(0, 3, 2), \text{tarai}(2, 2, 1)) \\ &= \text{tarai}(1, 3, 2) = 3\end{aligned}$$

となり、9回 tarai 関数が呼ばれる (最初の1回も含めている)。

発展問題:

- 近似値の計算を再帰を使ってあらわしてみよう。
自然対数の底 e は、 $e = 1 + 1/1! + 1/2! + 1/3! + \dots + 1/n! + \dots$ という公式で計算できる。そこで、自然数 n を引数として、この式の第 n 項までを (浮動小数点数として) 求める関数を再帰的に定義しなさい。
- 前問と同様のことを、円周率 π に対してやってみよう π の計算としては、 $\pi/4 = 4\text{atan}(1/5) - \text{atan}(1/239)$ が有名な公式である。ここで atan は $\text{arc tan}(\text{tan}$ の逆関数) であり、展開すると、 $\text{atan}(x) = x - x^3/3 + x^5/5 - x^7/7 + x^9/9 - \dots$ となる。(プラスとマイナスが交互に来ることに注意せよ。)