ソフトウェア技法 ミニプロジェクト (前半) [中間レポートの課題]

亀山幸義 (筑波大学情報科学類)

[2017/7/14] 問題 (A-6) に 2 点追記しました.

1 多倍長演算; 非負の巨大整数 [必須課題]

整数型 (int 型) や浮動小数点型 (float 型) は、32bit あるいは 64bit といった固定長であるので、精度に限界がある。しかし、可変長であるリストを使えば、このような固定長データの限界を越えることができる。たとえば Lisp 言語の多くの方言では、bignum というデータ構造 (big number の意味) が用意されており、これは「(メモリが許す限り) いくらでも大きな整数」を表せるデータ構造である。

多倍長計算は、たくさんの固定長データを束ねて、可変長のデータを表す手法である。固定長データとして，0 から 9 までの整数データのみを使うことを考えよう．このような整数のリストとしては，いくらでも長いものを使えるとする．

すると、8141592653 という大きな整数を整数リストとして,

[3; 5; 6; 2; 9; 5; 1; 4; 1; 8]

と表すことができる。(下の桁 (けた) から並べたのは、数学的な理由はなく，ただ，後で処理するとき，下の桁からの処理が多くなるので，書きやすくなる，という理由である。)

上記の数を、141421356 と加えるには、まず，それぞれをリスト表現して,

[3; 5; 6; 2; 9; 5; 1; 4; 1; 8]
[6; 5; 3; 1; 2; 4; 1; 4; 1]

これらを、桁ごとに加えて

[9; 10; 9; 3; 11; 9; 2; 8; 2; 8]

を得て，最後に、繰り上げ処理，つまり，0 から 9 までの整数の範囲を越えた数の処理をすればよい。

[9; 0; 0; 4; 1; 0; 3; 8; 2; 8]

このようにすれば，加算が実現できることがわかるであろう．

さて，上記では、1 桁で表せる数を 0 から 9 の整数にしたが、1 桁は、もっと大きな数でよい。そこで 1 桁の範囲を 0 から R-1 とする．この場合，$x_0, x_1, \cdots, x_k$ が 0 から R-1 までの整数だとすると、

$[x_0; x_1; \cdots; x_k]$

という整数リストは、$x_0 + x_1R + x_2R^2 + \ldots x_kR^k$ という非負の整数をあらわす。ただし、最上位の桁である x_k は 0 でないものとする。(従って、整数の 0 は、多倍長整数として表すと [0] でなく空リスト [ ] となる。)

R としてどのくらい大きな整数が許されるかは、後でどのような計算をするかによる[*1]。ここでは，簡単のため，R=10000 (10 の 4 乗) とする．

演習問題 (A)

(A-1) 上記の多倍長整数と、整数 1 つを与えられたとき、それらの加算をおこなう関数 `bi_add1` を定義せよ。ただし、「0 から 9 の数字」ではなく、「0 から R-1 までの数」とせよ。(R は、プログラムの最初の方で `let r = 10000` などとして与えることにして、それ以降のプログラムでは、R の具体的な値に依存しないようにせよ。)
例 (R=10 のとき): `bi_add1 [9; 8; 5]  8 = [7; 9; 5]`
ヒント: この関数の定義は、以下のようにするとよい。ポイントは、加算における繰り上がりの処理である。

[*1] 乗算のことを考えると $(R+1)^2 < \text{max\_int}$ という制約がある。

```
(* 多倍長整数の基数（1桁が表す整数の最大値プラス1） *)
let r = 10000 ;;

(* bigint 型を int list 型であると定義する *)
type bigint = int list ;;

(* 多倍長整数と整数の加算 *)
let rec bi_add1 b1 n =
  if n = 0 then b1
  else
    match b1 with
    | [] →
          if n < r then [n]
          else (n mod r) :: (bi_add1 [] (n / r))
          (* n >= r である場合は、ちょっと面倒な処理が必要 *)
    | h::t →
          let v = h + n in
          if v >= r then
             ..... （ここを実装すること）
          else
             v :: t
;;
```

(A-2) 上記の多倍長整数を 2 つもらい、それらの加算をおこなう関数 `bi_add` を定義せよ。

例 (R=10 のとき): `bi_add [3; 4; 5] [2; 3; 4; 2] = [5; 7; 9; 2]`

例 (R=10 のとき): `bi_add [9; 8; 5] [2; 3; 4; 2] = [1; 2; 0; 3]`

ヒント: 実装は以下のようになるだろう。ここで補助関数 `bi_add_carry` の最後の引数 carry は「下の桁からの繰り上がり (キャリー)」を意味する。

```
(* 補助関数（繰り上がり対処つき）*)
let rec bi_add_carry b1 b2 carry =
  match b1 with
  | [] → bi_add1 b2 carry   (* 前問の関数を利用 *)
  | h1::t1 →
     begin
        match b2 with
        | [] → bi_add1 b1 carry
        | h2::t2 → ... （ここを実装すること）
     end
;;

(* bigint 同士の加算 *)
let bi_add b1 b2 =
   bi_add_carry b1 b2 0 ;;

(* 例題 *)
let bigint1 = [3154;  519; 6101;   21; 9485;   51; 1000; 4401;  101; 8810] ;;
let bigint2 = [6184; 5103; 3181;   11; 1032; 4919; 9999; 4910; 1018] ;;
let bigint3 = bi_add bigint1 bigint2 ;;
```

補足 (上記の begin ...end という構文について): OCaml の match 式は「match 式終わり」を示すキーワードがないため、構文上の曖昧さが生じることがある。これを避けるためには、丸括弧を使ってもよいが、見た目が悪くなるので、丸括弧のかわりに begin ... end を使うことが多い。

たとえば、以下のようなネストした match 式では、h1::t1 のケースが match b2 with ... の中のケースと見なされてしまう (match b1 with のケースとならない) ため、意図したプログラムとならない。このような場合、内側の match 式の前後を begin ... end で囲むと良い。

```
  match b1 with
  | [] →
        match b2 with
        | [] → ...
        | h2::t2 → ...
  | h1::t1 → ...
```

(A-3) 上記の多倍長整数に対する、減算をおこなう関数 `bi_sub` を定義せよ。`bi_sub` においては、「最上位の桁が 0 でない」という条件を満たすように作成せよ。なお、0 が最上位にいくつかあらわれる場合、すべて除去する操作が必要である。また、答えが負の数になる場合は、failwith でエラーにせよ。

例 (R=10 のとき): `bi_sub [2; 3; 4; 2] [9; 8; 5]= [3; 4; 8; 1]`

例 (R=10 のとき): `bi_sub [9; 8; 5]  [2; 3; 4; 2] =` (異常終了)

(A-4) 上記の多倍長整数に対して, int 型の整数 N を乗算した結果を返す関数 `bi_mul1` を定義せよ。ただし、ここでは $0 \leq N < R$ と仮定してよい。

例 (R=10 のとき): `bi_mul1 [2; 3; 4; 2] 5 = [0; 6; 1; 2; 1]`

ヒント: 実装は以下のようになるだろう。(補助関数における carry という引数は、加算のときと同様である。)

```
(* bigint と int の乗算 *)
(* ここでは、 0 <= n < r と仮定する *)
let rec bi_mul1_carry b1 n carry =
  match b1 with
  | [] →
     if carry = 0 then [] else [carry]
  | h1::t1 →
     ...
;;

let bi_mul1 b1 n =
   bi_mul1_carry b1 n 0
;;
```

(A-5) 上記の多倍長整数に対する、乗算をおこなう関数 `bi_mul` を定義せよ。(ひとつ前の `bi_mul1` を利用するとよい。)
例 (R=10 のとき): `bi_mul [2; 3; 4; 2] [5; 4] = [0; 4; 4; 9; 0; 1]`

(A-6) 通常の int 型の整数 N が与えられたとき、N! (N の階乗) の計算を行う関数 `bi_fact` を定義せよ。1000 の階乗が何秒で計算できるか、など、限界まで挑戦せよ。
[2017/7/14 追加] 実行時間を計測できる人は,「1 秒以内で計算できる N! の N はいくつか」という問題設定にチャレンジしてみてほしい.
参考: 実行時間を測定するため、OCaml の Unix モジュールに含まれる time 関数が使える。time 関数の使い方は「TA のページ (ウェブページ参照)」で紹介されているので興味がある人はそちらを見てほしい。
例 (R=10000 のとき): `bi_fact 10` = [8800; 362]
例 (R=10000 のとき): `bi_fact 15` = [8000; 7436; 3076; 1]

[2017/7/14 追加] この問題は，乗算 `bi_mul` が実装できたことを前提としているが，乗算の実装が完了しなかった人は，階乗のかわりに，Fibonacci 数列の第 N 項を計算するのでもよい．(fib(1)=fib(2)=1 で fib(n+2)=fib(n)+fib(n+1) となる関数のことである．これは足し算さえ実装できていれば，計算できる．)

2 多倍長演算; 負の数の表現 [発展課題]

前節では、非負の整数のみを表現する方式として、

[3; 5; 6; 2; 9; 5; 1; 4; 1; 8]

といった整数リストを用いた。これだけでは負の数が表現できず、`bi_sub` においては、負の数になるとエラーにするしかないなど、不都合があった。そこで、負の数も表せるように拡張しよう。拡張のしかたはいろいろがあるが、ここでは、(プログラミングの練習を兼ねて) 組 (tuple、タプル) を使うことにする。

まず、組のデータ型を使ってみよう。

```
(* 組（タプル） *)
(* int * float 型のデータ *)
let pair1  = (10, 3.14) ;;

(* 組を使うためのパターンマッチ *)
let f x =
  match x with
  | (p,q) → (float p) *. q ;;

let _ = f pair1 ;;

let _ = f (20, 2.8) ;;

(* パターンは 関数の引数でも書くことができる *)
(* 以下の関数 f2 は上記の f と同じ *)
let f2 (p,q) =
   (float p) *. q ;;

let _ = f2 pair1 ;;
```

2 つ組だけでなく、3 つ以上の組も同様に定義できる。

```
(* 3つ組 *)
(* int * float * string 型のデータ *)
let triple1  = (10, 3.14, "abc") ;;

(* 組を使うためのパターンマッチ *)
let g x =
  match x with
  | (p,q,_) → (float p) *. q ;;

let _ = g triple1 ;;

(* 2つ組と3つ組は型が違うので、以下のものはエラー *)
let _ = f triple1 ;;

(* 2つ組と3つ組は型が違うので、以下のものはエラー *)
let _ = g pair1 ;;
```

上記の「組」の型を用いて、負の整数も表せるような多倍長整数の表現を作ろう。ここでは、正か負かを表すために真理値型の true/false を用い、それと、前節の bigint 型と同様の整数リストを組にしよう。

```
(* 符号つき巨大整数の型 *)
type signed_bigint = bool * bigint ;;

(* 正の巨大整数の例 *)
let ex1 = (true, [3; 5; 6; 2; 9; 5; 1; 4; 1; 8]) ;;

(* 負の巨大整数の例 *)
let ex2 = (false, [3; 5; 6; 2; 9; 5; 1; 4; 1; 8]) ;;
```

ただし、0 は、2 つの表現を持つ。

```
(* 以下のつは、どちらも 0 を表す。2*)

let ex3 = (true, []) ;;
let ex4 = (false, []) ;;
```

この表現のもとで、巨大整数の加減乗除を計算してみよう。

演習問題 (B) すべて発展課題

(B-1) 符号つき巨大整数に対する加算 `sbi_add` と減算 `sbi_sub` を定義せよ。(sbi というのは signed big integer のイニシャルである。)

```
(* 符号つき巨大整数の加算。ここでは r = 10 と仮定 *)

let test1 =
  sbi_add  (true, [3; 7; 6]) (false, [5; 2; 9]) ;;
==> (false, [2; 5; 2])

let test2 =
  sbi_sub (true, [3; 7; 6]) (false, [5; 2; 9]) ;;
==> (true, [8; 9; 5; 1])
```

(B-2) 符号つき巨大整数に対する乗算 `sbi_mul` を定義せよ。
ヒント: 乗算の場合は、むしろ簡単であり、符号なし巨大整数に対する乗算 `bi_mul` をほとんどそのまま利用できる。(符号の部分の計算のみ追加でプログラムを書く必要があるが、本体の bigint 同士の乗算には `bi_mul` が使える。)

3 多倍長演算; 小数の表現 [発展課題]

前節と同様の考え方により「多倍長の小数」も表すことができる。ここでは、「整数部分が 0 以上 R-1 以下の非負の小数」だけを表すことにする。たとえば、円周率 π を小数点以下 9 桁で切り捨てた値を、

[3; 1; 4; 1; 5; 9; 2; 6; 5; 3]

と表そう。今度は「上位の桁」が前に来ていることに注意せよ。(従って、加算の定義も、もう 1 度完全にやり直す必要がある。)

演習問題 (C) すべて発展課題

(C-1) $e = 1 + 1/1! + 1/2! + 1/3! + \cdots + 1/n! + \cdots$ という公式を使って、e の値をなるべく精度よく計算せよ。

(C-2) $\pi/4 = 4atan(1/5) - atan(1/239)$ という公式を使って、π の値をなるべく精度よく計算せよ。

ここで atan は arc tan(tan の逆関数) であり、展開すると、$atan(x) = x - x^3/3 + x^5/5 - x^7/7 + x^9/9 - \cdots$ となる.