ソフトウェア技法 後半 (2016/2/6, 16:05 最終更新)

亀山 幸義

注意追加: 多倍長計算では，$R = 2^{31}$ でやってください．$R = 2^{32}$ にしてしまうと，OCaml の int の乗算ではあふれる可能性があります.

全体注意追加: 解答プログラムの概形で `let rec ...` と書いてあっても、再帰関数でなくて構いません。

1 OCaml プログラミングの基礎

前半の Garrigue 先生の授業では、言語としての OCaml の基礎的なこと、特徴的なこと、便利な仕組みを幅広くカバーしていた。

後半の授業では、一般的な「関数プログラミングの基本」の考えかたを学ぶことにしよう。具体的には、「再帰関数」と「代数的データ型 (Garrigue 先生の授業では「再帰データ型」と呼んだもの)」、の話題に絞って、もう少し深く、例題を見てみることにする。「関数プログラミングの心」ともいえる高階関数や多相型といった機能は、今回の授業では (ちらちらと出てくるが) 深入りはできない。これは、独学するか、将来の別の授業で学んでほしい。

授業資料は以下の 2 つの OCaml ファイル (コメントいり) である。

- art1.ml: 再帰関数
- art2.ml: 代数的データ型 (再帰データ型)

2 OCaml ミニプロジェクト

- 数式処理: 微積分 (必須課題)
- 多倍長計算 (必須課題)
- スタック機械: 逆ポーランド記法 (発展課題)
- 論理式: 命題論理の証明 (発展課題)
- 自分で設定する課題 (発展課題)

2.1 数式処理: 微分 (必須課題)

数式の中でも「多項式」とよばれる式は、$a_0 + a_1x + a_2x^2 + \cdots + a_nx^n$ の形をしている。このような多項式を `[a0; a1; a2; ...; an]` と表したとき、微分および積分をあらわす関数を定義せよ。(注意: 以前のこのファイルでは、「積分結果が多項式にならないときは、failwith でエラーを発生させよ」と書いたが、そのような場合はないので、この部分は削除した。) また、「積分」の結果は、定数 C の分だけ不定であるので、これは 0 であるとせよ。(つまり、不定積分ではなく、定積分 $\int_0^x f(x)dx$ を計算してほしい。)

```
type polynomial = float list
let rec differentiation (p1:polynomial) : polynomial =
   ...
let rec integral (p1:polynomial) : polynomial =
   ...
```

```
differential [1.0; 2.0; 3.0] = [2.0; 6.0]
integral     [1.0; 2.0; 3.0] = [0.0; 1.0; 1.0; 1.0]
```

なお、let rec と書いてあるが、integral などの関数自身は再帰関数でなくてもよい (入れ子の関数としてもよい、というか入れ子、あるいは、別関数で再帰するように書くことになる)。

(発展課題) 余力がある人は、複数の変数を含む多項式の微分や積分などにも挑戦せよ。

2.2 多倍長演算 (必須課題)

いろいろなプログラミング言語がもっているデータ型である浮動小数点型 (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。(OCaml では 64bit) 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精度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」をあらわすことのできるデータ構造である。

多倍長計算は、固定長データを可変個たばねて、可変長のデータを表す手法である。その原理は極めて簡単である。仮に、1 から 10 までの整数しか表せないが、リストを使えるプログラム言語があったとする。すると、3141592653 という大きな整数を、

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

と表すことができる。(ここで逆順に並べたのは、その方がプログラミングしやすいという理由である。)

上記の数を、141421356 と加えるには、

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

を、桁ごとに加えて

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

次に、繰り上げ処理をすればよい。

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

あまりにも簡単なことに驚くであろう。効率よく実装するためには、いろいろなことを考えなければいけないが、今回の授業では、「正しく実装する」ことを主眼におき、効率はあまり考えないことにする。

上記では、「1 けた分」を 1 から 10 の整数にしたが、OCaml の整数として表せる数なら (OCaml で加減乗除ができる限りは) 「1 けた分」は、もっと広い範囲の数でよい。

ここで、x_i が 0〜R-1 の整数だとすると、

[x0; x1; x2; ...; xk]

という整数リストは、$x_0 + x_1R + x_2R^2 + \ldots x_kR^k$ という整数をあらわす。ただし、最上位の桁である x_k は 0 でないものとする。したがって、整数の 0 の多倍長整数としての表現は、[0] でなく、空リスト [] であることに注意せよ。(簡単のため、負の数や、小数は考えていない。)

演習問題 (必須課題):

- 上記の多倍長整数に対する、加算をおこなう関数 add を定義せよ。ただし、リストはいくらでも長くできるものとし、「リストが長過ぎるためのエラー」が起きることはないものと仮定してよい。

  なお、「固定長データ」としては、「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 = 10` あるいは `let r = power 2 31` として与え、プログラム中では、R の具体的な値には依存しないようにせよ。)

  ここで、x_i が 0〜R-1 の整数だとすると、

```
type bigint = int list
let rec bi_add (b1 : bigint) (b2 : bigint) : bigint =
   ...
let rec bi_sub (b1 : bigint) (b2 : bigint) : bigint =
   ...
let rec bi_mult (b1 : bigint) (b2 : bigint) : bigint =
   ...
let rec bi_print (b1 : bigint) : string =
   ...
```

- 上記の多倍長整数に対する、減算をおこなう関数 sub を定義せよ。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。(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。)

  [補足] ここで、10 -20 のように、答えが負の数になる場合はどうするか、が、問題文に書かれていない。そのような場合は、failwith でエラーにしてもよいし、あるいは、0 (つまり、空リスト) にしてもよい。

- 上記の多倍長整数に対する、乗算をおこなう関数を定義せよ。(OCaml の整数は 64bit なので、$R = 2^{31}$ にしておくことにより、0 から R-1 までの数 2 つの積は、OCaml の整数に対する乗算をそのまま使ってよいことになる。なお、OCaml では、乗算等で overflow してもエラーにならない。)

- 多倍長整数として、N! (N の階乗) の計算を行いなさい。実行時間 10 秒以内で何番目の階乗まで求まるか、限界まで挑戦せよ。(「実行時間 10 秒」を測定するためには、時間を計測する関数が必要であり、Unix モジュールの time 関数などが使える。ただし、この演習ではそんなところまでやらなくてよく、シェルコマンドなどで「おおよそ」の時間を計測すればよい。)

- 多倍長整数として、Fibonacci 数列の計算をおこない、実行時間 10 秒以内で何番目の Fibonacci 数まで求まるか、限界まで挑戦せよ。

- (発展課題) 多倍長整数を、10 進表現の文字列に変換する関数を定義せよ。たとえば、R=10 のとき、[1;2;3] を “321” という文字列に変換してほしい。(この文字列をプリンタで印刷すると、紙を大量に消費してしまうかもしれないので、やらないこと。テストは画面上でのみ行なうこと。)

- (発展課題) 上記を参考に、多倍長の「小数」演算を実装せよ。ここでは、ややこしさを避けて、[x0; x1; x2; ...; xk] という整数リストは、$x_0 + x_1R^{-1} + x_2R^{-2} + \ldots x_kR^{-k}$ という小数をあらわすこととする。(したがって、R より大きな数はあらわせない。) 注意点は、今度は「上のけた」である x_0 からリストにいれていることである。

- (発展課題) 多倍長「小数」に対して、除算 (割り算) を行なう関数を定義せよ。ただし、割られる数は多倍長小数であるが、割る数は 1 以上 R-1 以下の整数と仮定せよ。

- (発展課題) 多倍長「小数」を使って、「自然対数の底 (e)」のなるべく多くの桁を正しく計算せよ。実行時間 1 秒以内とする。

- (発展課題) 多倍長「小数」を使って、円周率 π のなるべく多くの桁を正しく計算せよ。

e の計算では、$e = 1 + 1/1! + 1/2! + 1/3! + \cdots + 1/n! + \cdots$ という公式を使うとよい。

π の計算としては、$\pi/4 = 4atan(1/5) - atan(1/239)$ を用いなさい。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。(プラスとマイナスが交互に来ることに注意せよ。)

2.3 スタック機械: 逆ポーランド記法 (お勧めの発展課題)

逆ポーランド記法は、数式の記述方法の一種であり、「後置記法」とも呼ばれる。通常の記法で、(1+2)*3+4 となる数式は、逆ポーランド記法では、12+3*4+ となる。このような記法は、「かっこ」を使わなくても曖昧さがないため、かっこは使わない。

本問では、簡単のため、「1 けたの数字」(0 から 9 まで) と、加算と乗算のみを持つ小さな言語を考え、この言語の式を入力すると、その値を計算するプログラム (評価器) を作る。

ちょっと考えるとわかるように、このような式の処理には、スタックを 1 本もっているとよいことがわかる。このスタックには、「読みこんだが、まだ計算していない部分 (式の一部)」を積んでおく。

たとえば、「12+3*4+」という式で、「1,2」まで読み終わったときは、スタックには「1」と「2」がこの順番に積まれる。次に「+」を読むと、演算子なので、スタックのトップ (一番最後に積まれたもの) と、トップの次の要素である「2」と「1」が取り出され、足し算をやったあと、結果の「3」が、またスタックにつまれる。この瞬間のスタックは「3」だけが積まれ、まだ読んでいない式は「3*4+」である。以下順次読みこみを進めて、数が読まれたらそれをスタックに積み、演算子が読まれたら、スタックからデータを取り出して計算をして結果をスタックに積む、という操作を繰返す。

スタッに積むのは整数だけなので、スタックは「整数リスト」とするとよい。そして、以下のスタック操作関数を用意する。

```
type stack = int list
let push (n : int) (s : stack) : stack = n :: s
let pop  (s : stack) : int * stack =
   match s with
   | [] -> failwith "pop for empty stack"
   | h::s2 -> (h,s2)
let isEmpty (s : stack) : bool = (s = [])
```

ここで push/pop/isEmpty は、それぞれ、スタックに対するプッシュ操作 (データを 1 つ積む)、ポップ操作 (トップ要素をスタックから取りだす)、空スタックかどうかの判定操作をあらわす。

演習問題 (発展課題):

- 「12+3*4+」という式は、`"12+3*4+"`という string として与えられるとする。(なお、数はすべて一桁と仮定する。) この文字列を s とすると、i 番目の文字は、`s.[i]` で得られる。i が 0 からはじまることに注意せよ。このような文字列が与えられたとき、逆ポーランド記法の式として適切に処理せよ。

[補足] この処理は、一種のループ構造で処理するので、例によって「入れ子関数」で書けそうであるが、注意点は、(1) 内側の関数 (ループに対応) では、入力された文字列を 1 つずつ消費していくだけでなく、スタックを変

形していく (スタックに整数を積んだり、スタックから整数を取りだしたりする) ので、スタックを引数に取らないといけないし、返り値にもスタックを含めないといけないこと、(2) スタックの初期値は空であること、(3) 入力として間違ったものがあり得ること (たとえば、"+123" という式は、不正な式であるが、そのようなものを入力してしまうことがあること)、である。

```
let eval (str : string) : int =
  let rec walk (stk : stack) (i : int) : stack =
   if i < String.length str then
     match str.[i] with
     | '+' ->  let stk2 = ... in ... (walk stk2 (i+1)) ...
     | '*' ->  let stk2 = ... in ... (walk stk2 (i+1)) ...
     | '0' ->  ... ...
     | '1' ->  ... ...
     | ...
     | '9' ->  ... ...
     | _   ->  failwith ("illegal input: " ^ str.[i])
   else stk
  in
  let (stack_top, stk) = pop (walk [] 0) in
    if isEmpty stk then
      stack_top
    else
      failwith "illegal input"
```

なお、for ループや while ループで書いてもよいが、そうすると、「現在のスタック」を保存しておく、書換え可能変数 (破壊的変数、参照型の変数) を用意する必要がある。

```
  let stk = ref [] in
   for i = 0 to (String.length str - 1) do
     match str.[i] with
     | '+' ->  stk := ..(! stk) ...
     | ...
   done;
   let (stack_top, stk2) = pop (! stk) in
    if isEmpty stk2 then
      stack_top
    else
      failwith "illegal input"
```

2.4 論理式: 命題論理式の変形 (発展課題)

ファイル"art2.ml" の 演習問題 12.3 を解きなさい。

2.5 論理式: 命題論理の証明 (発展課題)

命題論理の論理式を formula データ型であらわす方法については、すでに演習でおこなった。ここでは、一歩すすめて、与えられた命題論理の論理式が証明可能かどうかを自動的に判定するプログラムを書いてみよう。これは「prover (証明器)」とよばれるプログラムとなる。

命題論理の証明器を書く方法はいろいろある。1) 真理値表を作って、すべておケースを調べる。2) 演習でおこなった方法で、論理積標準形に変形して、恒真かどうかを検査する。3) Wang のアルゴリズムを使う。

ここでは詳細は述べないが、発展課題としてこれにチャレンジしたい人は、教員に質問してみてほしい。また、インターネットには様々なヒントがのっている。

2.6 自分で設定する課題 (発展課題)

浅井先生の OCaml 入門書では、東京のメトロネットワークにもとづいて、「駅 A から駅 B への最短経路を求める問題」がとりあげられている。アルゴリズムとしては、Dijkstra 法として知られる古典的な課題であるが、OCaml でグラフなどのデータ構造を扱う演習としては、格好の問題である。発展課題としてこれにチャレンジしたい人は、是非やってみてほしい。

3 付録: 授業資料、レポート提出方法

- Garrigue 先生の授業資料: `http://www.math.nagoya-u.ac.jp/~garrigue/lecture/tsukuba16/`
- 亀山の授業資料: `http://logic.tsukuba.ac.jp/~kam/theart/`
- レポート提出は、manaba から (`https://manaba.tsukuba.ac.jp`)
- 2015 年度 (2016 年 1-2 月) の登録キーは **8603183**) です。
- 問合せ: `ocaml@logic.cs.tsukuba.ac.jp` (亀山および TA 薄井，山口)

レポート締切 (締切を過ぎると提出できません):

- Garrigue 先生: 2016/2/7(Sun) 23:55
- 亀山: 2016/2/15(Mon) 9:00 (午前)