代数式を比較する

和田 英一（IIJ 技術研究所）
wada@u-tokyo.ac.jp

■式の構文規則

教師を経験した人なら学生の書いたプログラムや数式の判読や理解がいかに面倒であるか，十分理解できよう．私が学生のころ，ある教授は試験にノートを持参させるのは答案に非常識な式を書いて欲しくないからと言われた．今回は2002年11月金沢大会の問題B, Equals are Equalsを話題にする．千差万別の答案の代数式の中から，教師の代数式と式の意味が一致するものを探すプログラムを書けというものだ．

コンパイラを書いたことがあれば，そう面倒な問題ではないが，最近コンパイラを書く機会もさほどないから，挑戦した諸君には難しかったかもしれない．Scheme報告書[1]風な拡張BNFによる，この問題での代数式の定義は以下の通り．

〈代数式〉→〈項〉〈後続項〉*
〈後続項〉→ +〈項〉| - 〈項〉
〈項〉→〈因子〉+
〈因子〉→〈非負整数〉|〈変数〉|〈変数〉 ^ 〈0以外の数字〉|（〈代数式〉）
〈非負整数〉→〈数字〉+
〈0以外の数字〉→ 1|2|3|4|5|6|7|8|9
〈数字〉→ 0|〈0以外の数字〉
〈変数〉→ a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z

構文記述で〈何か〉* は〈何か〉の0回以上の繰り返し，〈何か〉+は〈何か〉の1回以上の繰り返しを表す．

式の意味では，〈項〉を + や - でつなげたものは項の和，差を表し，〈因子〉の並びは〈因子〉の積を表す．〈変数〉 ^ 〈0以外の数字〉は〈変数〉の〈0以外の数字〉乗を表す．

+ や - は単項演算子としては使わず，必ず二項演算子とする．〈非負整数〉や ^ の後の〈0以外の数字〉に別の〈非負整数〉が続くときは，その間に1個以上の空白を置く．空白は〈非負整数〉の途中には置けないが，代数式を見やすくするため，式の途中に置くことはできる．またそれぞれの代数式は1行（80字）に収まっている．

例：
```
a+b+c
(a+b)+c
a-(b-c)+2
.
4ab
(a - b)(0-b+a) - 1a ^ 2 - b ^ 2
2 a 2 b
.
108 a
2 2 3 3 3 a
```

```
4 a^1 27
.
.
```

のように入力にはピリオドで区切られた代数式のグループがあり，グループ内の最初が教師の式，それに続くのが学生の式で，その各々に対して最初の式と同じならyesと，違えばnoと答える．例題の最初のグループは，各行が$a+b+c$, $a+b+c$, $a-b+c+2$ だから，yes, noと答える．次は$4ab$, $-2ab$, $4ab$でno, yes．最後はすべて$108a$ゆえyes, yesである．グループ内に式が1行もなければ入力は終わる．

■比較のための標準形

2つのものを比べるには，分数なら通分して分子を比較するように，何か標準形にしなければならない．ハッカー英語辞典[2]に「"canonical"をcanonicalな形で使った」と出てくるあのcanonical formである．中学校で習う式の展開と因数分解の別名はそれぞれ加法標準形，乗法標準形であるが，経験上因数分解は困難だから，当然加法標準形に持っていくことになる．つまり式の積は展開し，変数の冪乗は変数を指数の個数だけ掛けた形とし，最終的には係数と変数の積の形の項を足し合わせた形にする．並べる順番も重要である．項の中の変数の積は，変数のアルファベット順に並べ，項の並べ方は変数の並びの辞書式順とする．

この問題では変数は1文字だから，変数の積は文字を単に連結すればよい．その表現法として文字列と記号アトムの優劣を比較すると，記号アトムの方が見た目はきれいだが，しかし

- 定数項の扱いで，空の文字列はあるが，空の記号アトムはない．
- Schemeには文字列を扱う手続きはあるが，記号アトムは文字列に変換して操作しなければならない．
- 使用済み文字列はゴミとして回収されるが，記号アトムはoblistに登録され，ゴミとならない．

などの考察から，変数は1字の文字列，変数の積は文字列で表現するのがよいと考えた．

各項は係数と変数の積を表す文字列の2要素だから，点による対で表現しよう．式の全体はこのような項のリストにすればよいから，たとえば$a+b+1$の3乗の標準形は

```
((1 . "") (3 . "a") (3 . "aa") (1 . "aaa") (3 . "aab") (6 . "ab")
 (3 . "abb") (3 . "b") (3 . "bb") (1 . "bbb"))
```

となる．

■式の読み込み

例によってLispの一方言のSchemeでプログラムしたい．代数式を読み込み，変数，演算子，整数のリストに変換できれば，あとはこちらのものだ．元の代数式が入れ子になっているから，入れ子の代数式はリストの入れ子に変換するのは常識であろう．

上の例題の各行を次のように変換して読み込むのも一案だ．

```
("a" + "b" + "c")
(("a" + "b") + "c")
("a" - ("b" - "c") + 2)
()
(4 "a" "b")
(("a" - "b") (0 - "b" + "a") - 1 "a" ^ 2 - "b" ^ 2)
(2 "a" 2 "b")
()
(108 "a")
(2 2 3 3 3 "a")
(4 "a" ^ 1 27)
()
()
```

基本方針は次の通り．〈代数式〉は〈項〉のリストとする．〈変数〉は英字1文字なので，そのまま1文字の文字列にする．読み込んだ〈非負整数〉と ^ の後の〈0以外の数字〉は整数のアトムにする．演算子には +, -, ^ があるが，そのまま記号アトムにする．ピリオドだけの行は，他との類推では (.) かもしれないが，Lispではピリオドには特別の意味があり，このようなS式は扱えないので，() で表現する．

```
(define (readline)              ;式の読み込み  "."のみのときはnilを返す
  (let ((char '()))
    (define (readch) (set! char (char->integer (read-char s))))
    (define (read-syl exp)      ;シラブルリード
      (define (next) (readch) (read-syl exp))
      (define (integer->symbol x)
        (string->symbol (string (integer->char x))))
      (define (readnum n)       ;整数部分の読み込み
        (readch)
        (cond ((<= 48 char 57) (readnum (+ (* n 10) char -48)))
              (else (set! exp (cons n exp)))))
      (cond ((<= 48 char 57) (readnum (- char 48)) (read-syl exp))
            ((<= 97 char 122)                    ;英小文字は1文字列に
             (set! exp (cons (string (integer->char char)) exp)) (next))
            ((= char 32) (next))                 ;空白
            ((= char 46) (next))                 ;ピリオド
            ((= char 41) (reverse exp))          ;かっこ閉じ
            ((= char 10) (reverse exp))          ;改行
            ((= char 40)                         ;かっこ開き
             (readch) (set! exp (cons (read-syl '()) exp)) (next))
            (else (set! exp (cons (integer->symbol char) exp)) (next))))
    (readch) (read-syl '())))
```

手続きreadlineは手続きreadchでcharに次の文字のアスキーコードの数値を代入し，結果を累積する引数expを空にして手続きread-sylを呼ぶ．read-sylはコードが48～57（数字），97～122（英小文字），32（空白），46（ピリオド），41（かっこ閉じ），10（改行），40（かっこ開き），else（それ以外）のいずれかにより，分岐する．

数字の場合は数字以外のものがくるまで10進2進変換を続け，数字以外がくれば，変換済みの数値をexpにconsし，read-sylを続ける．

英小文字は文字に戻し文字列に変換し，expにconsし，(next経由で) read-sylを続ける．

空白とピリオドは何もせず次の文字を読み込み，(next経由で) read-sylを続ける．

かっこ開きは新しくexpを空にし，次の文字を読み，read-sylへ入り，帰ってきたリストを古いexpにconsして続ける．

改行がきたらリストは逆順にできているので，expを逆転して戻る．かっこ閉じも同様．それ以外は演算子で，アスキーコードになっているのを記号アトムに戻す．

■式の変形

readlineはたとえば

```
(a - b)(0-b+a) - 1a ^ 2 - b ^ 2
```

のような入力の式を

```
(("a" - "b") (0 - "b" + "a") - 1 "a" ^ 2 - "b" ^ 2)
```

の形に変えてくれる．ここまでくれば，後はリスト処理を繰り返せばどうにでもなる．

まず冪乗を処理する．^の後は数字1文字なので，たかだかその数だけ直前の文字列を繰り返すことにする．つまり上の式は手続きev^により

```
(("a" - "b") (0 - "b" + "a") - 1 "aa" - "bb")
```

に変形する．

次は - を処理しよう．それには -1 を係数に組み込み，項は + でつなぐ．つまり上の式は手続き ev- により

```
(("a" - "b") (0 - "b" + "a") + -1 1 "aa" + -1 "bb")
```

になる（かっこ内の式は未処理なことに注意）．

最後に手続き ev+ を使い，

```
(("a" - "b") (0 - "b" + "a")) (-1 1 "aa") (-1 "bb"))
```

に変形する．つまり + で区切られた項のリストを項ごとにリストにし，+ を除く．これら一連のプログラムは

```
(define (ev^ l)                    ;冪乗の処理
  (define (rep a b c)
    (cons (make-string b (string-ref a 0)) c))   ;文字aをb個連結しcへcons
  (cond ((< (length l) 3) l)
        ((eq? (cadr l) '^) (rep (car l) (caddr l) (ev^ (cdddr l))))
        (else (cons (car l) (ev^ (cdr l))))))
(define (ev- l)                    ;負号の処理
  (if (null? l) l
    (let ((h (car l)) (r (ev- (cdr l))))
      (if (eq? h '-) (cons '+ (cons -1 r)) (cons h r)))))
(define (ev+ l)                    ;+を区切りとするリストの作成
  (define (ev++ l p)
    (if (null? l) (p '() '())
      (ev++ (cdr l)
        (lambda (x y)
          (if (eq? (car l) '+) (p '() (cons x y))
            (p (cons (car l) x) y))))))
  (ev++ l (lambda (x y) (cons x y))))
```

である．

このうち，ev^, ev- のプログラムは簡単だが，ev+ は多少ややこしい．

冪乗 ^ の処理ではリストの先頭から文字列，^, 整数と並ぶところがないか見ていく．リスト長が3より短くなればそういう候補はなく，リストをそのまま返す．そうでなければ先頭から2番目の要素が ^ かを見，そうなら先頭の文字列，3番目の整数，それ以降を ^ 処理したものをもって rep を呼ぶ．rep は第3引数の前に，文字列を整数個並べたものを（make-string を使い）構成する．いずれでもなければリストの次以降を ^ 処理したものの前に先頭の要素を付ける．

ev+ には ev++ という下請け手続きがあり，これは第2引数に2引数の手続き p をとる．p の第1引数(x)は現在処理中の項の要素を後方から順に接続したリスト，第2引数(y)はすでに処理した項のリストである．ev++ を見ると，第1引数(l)が空なら，どちらのリストも空にして p を呼ぶ．つまり両引数のリストも空にして，要素が最後の1個の時の ev++ から渡された p を呼ぶのである．第1引数(l)が空でなければ，第1引数を (cdr l) にして ev++ を再帰的に呼ぶが，そのとき渡す第2引数 p は，下請けで呼ばれたときに使う引数を (x y) とし，それに対して (car l) が + かどうかで，いかに処理し，貰ってきた p を呼ぶかが記述してある．この方式は手続き ev-term でも3引数にして利用する．

■項の処理

ev+ で項の並びになった式は各項を ev-term で処理する．それには項の中の要素を整数，変数，かっこ内の式の3種に分類する．分類は ev+ と同様な手法を使う．上の3種のそれぞれが，x, y, z に戻されると，x の全要素を掛け合わせ，y の全要素を連結（append）する．Scheme の乗算 * は任意個の引数を取るが，0個なら乗算の単位元1を返す．この両結果を cons してとりあえず項を作る．次にかっこ内の式が z にあれば，式を処理する手続き ev-exp をそれぞれの式に作用させ，項の列の形をした式の並びが得られるが，これを手続き muls で一斉に乗算する．muls は引数が m . l とあるので分かるように1個以上任意個の引

数を取るようにできている．しかし実は2個ずつmulを使って乗算している．乗算は2段のmapで両方の式からすべての項の組合せをe0とe1に貰い，係数は掛け合わせ，変数は連結する．外側のmapの結果はappendしなければ1段の並びは得られない．これにxとyから一応作っておいた項を掛けると全体の項ができ上がる．

```
(define (ev-term l)              ;項の処理
  (define (term l p)
    (if (null? l) (p '() '() ())
        (term (cdr l)
          (lambda (x y z)        ;x:係数の列，y:文字列の列，z:かっこの式の列
            (let ((h (car l)))
              (cond ((number? h) (p (cons h x) y z))
                    ((string? h) (p x (cons h y) z))
                    (else (p x y (cons h z)))))))))
  (term l (lambda (x y z)
    (let ((x (cons (apply * x) (apply string-append y))))
      (if z (map                 ;かっこの式があれば処理する
              (lambda (y)
                (cons (* (car x) (car y))
                      (string-append (cdr x) (cdr y))))
              (apply muls (map ev-exp z)))
            (list x))))))
(define (muls m . l)
  (define (mul es)               ;2項の乗算
    (apply append
      (map (lambda (e0)
        (map (lambda (e1) (cons (* (car e0) (car e1))    ;係数は掛ける
          (string-append (cdr e0) (cdr e1))))            ;文字列はappend
            es)) m)))
  (if (null? l) m (mul (apply muls l))))
(define (ev-exp exp)             ;式の処理
  (norm (apply append (map ev-term (ev+ (ev- (ev^ exp)))))))
```

手続きmulは2項の乗算だが，第1項はmuls m . lのmを利用している．この辺はSchemeのスコープルールを援用している．点による対を利用しているので，変数の積を取り出すにもcadrではなく，cdrが利用でき，簡単だ．

■正規化

標準形に直す正規化手続きnormは以下の通り．

```
(define (norm e)                 ;正規化
  (define (sortstr s)            ;文字列内のソート
    (list->string (sort (string->list s) char<?)))
  (define (norm1 e)
    (cond ((null? e) e)
          ((= (caar e) 0) (norm1 (cdr e)))       ;係数が0なら削除
          ((and (> (length e) 1) (string=? (cdar e) (cdadr e))) ;同類項を統合
           (norm1 (cons (cons (+ (caar e) (caadr e)) (cdar e)) (cddr e))))
          (else (cons (car e) (norm1 (cdr e))))))        ;後続の項の正規化
  (norm1 (sort
   (map (lambda (x) (cons (car x) (sortstr (cdr x)))) e)
     (lambda (x y) (string<? (cdr x) (cdr y))))))
```

変数の積の文字列をソートするのが手続きsortstrだ．手続きstring->listはSchemeの標準手続きで文字列を文字のリストに展開する．たとえば(string->list "foo") ==> (#\f #\o #\o)これをchar<?でソートし，手続きlist->stringで元に戻せば，文字列内ソートができる．たとえば

(sortstr "foobar") ==> "abfoor".

この準備のあと，下から2行目で各項をsortstrし，それをstring<?で辞書式順にソートし，反復的正規化手続きnorm1に送る．norm1は空なら終わり．係数が0ならその項を削除して，後続の項の正規化へ．項が2個以上あって，先頭の2つが同類項なら，係数を加えて1個の項にし，さらにこの項から正規化を行う．それ以外なら，後続の項の正規化したものに，先頭の項をconsする．

■駆動ループ

式の処理は以上で終わりだが，最後に駆動ループがいる．これもScheme流に再帰によるループで書いてある．

```
(define (eqeq)                    ;駆動ループ
  (let ((exp (readline)) (ans '()))
    (define (subeq)               ;ブロック内のループ(末尾再帰でループする)
      (let ((exp (readline)) (tmp '()))
        (if exp
          (begin ;(show exp)      ;読み込んだ学生の式を表示
            (set! tmp (ev-exp exp)) ;(show tmp) ;学生の式の展開形
            (if (equal? tmp ans) (show 'yes) (show 'no)) (subeq))
          (show '.))))
    (if exp
      (begin ;(show exp)          ;読み込んだ教師の式を表示
        (set! ans (ev-exp exp)) ;(show ans)      ;教師の式の展開形を表示
        (subeq) (eqeq))           ;末尾再帰によるループ
      (newline))))

(define (show x) (newline) (write x))    ;虫取り用改行つき印字ルーチン
(define s (open-input-file "icpc.data"));データ読み込みのファイル名
(eqeq)                            ;駆動ループを起動
```

手続きshowは改行つきの出力ルーチンである．displayでなく，writeを使ったのは，文字列の2重引用符も出力したかったからだ．次の手続きopen-input-fileで，入力ファイルを指定する．最後に駆動手続き(eqeq)でプログラムを開始する．

手続きeqeqは手続きreadlineで1つの式を読み，expにおく．次に下請けルーチンsubeqを内部手続きとして定義し，下の方のifへいく．expが空でなければ，入力は終わりではないから，ev-expで標準形にしてansに入れておき，subeqを呼ぶ．これはローカルなexpに学生の式を読み込む．これが空でなければ標準形にしてtmpにおき，これと先ほどのansとの等価を調べ，答を出力して，subeqを再帰呼出しする．入力のグループの最後のピリオドを読むと，expは空になり，下請けルーチンから脱出する．そこにはeqeqの再帰呼び出しが待っていて，次のグループの処理に移る．入力最後のピリオドはeqeqのreadlineで読まれ，if文はフェイルし，プログラムは終了する．

審判団が持っていたテストデータ(1000行ほど)を実行するのに20秒弱かかった．

■数値実験による判定

式を標準形に変形して比較せずとも，変数に適当な値を代入して式の値を計算すればよいとも考えられる．しかしa=2, b=3とすれば，3 a = 2 bになってしまい，もともと等しくない式も等しくなる．そういうFermatテストのような一抹の不安はあれど，やってみた．冪乗a ^ 2は(expt a 2)の演算を行う．それぞれの変数にどういう値を代入するかは任意だが，Gödel Numberingにならい，

a	b	c	d	e	...	x	y	z
3	5	7	11	13	...	97	101	103

としてみた．これがよいという特段の理由はない．

前のプログラムと同様だが，readlineで英小文字を読んだときにassocを使って上の値に置き換える．ev^で冪乗を実行する．ev+で項の乗算と項同士の加算を実行するなどが違っている．

前と同じ例: (a - b)(0-b+a) - 1a ^ 2 - b ^ 2をreadline, ev^, ev-, ev+の各処理が終わったところで示せば，それぞれ

```
((3 - 5) (0 - 5 + 3) - 1 3 ^ 2 - 5 ^ 2)
((3 - 5) (0 - 5 + 3) - 1 9 - 25)
((3 - 5) (0 - 5 + 3) + -1 1 9 + -1 25)
-30
```

となる．プログラムは以下の通り．

```
(define (ev^ l)                  ;冪乗の演算実行
  (cond ((< (length l) 3) l)
        ((eq? (cadr l) '^)
         (cons (expt (car l) (caddr l)) (ev^ (cdddr l))))       ;冪乗実行
        (else (cons (car l) (ev^ (cdr l))))))
(define (ev- l)                  ;負号の処理 前のプログラムと同じ
  (if (null? l) l
    (let ((h (car l)) (r (ev- (cdr l))))
      (if (eq? h '-) (cons '+ (cons -1 r)) (cons h r)))))
(define (ev+ l)                  ;因子間の乗算,  項間の加算実行
  (define (ev++ l p)
    (if (null? l) (p 1 0)        ;pの引数1と0は乗算と加算の単位元
      (ev++ (cdr l)              ;前のプログラムの引数()はappendの単位元
        (lambda (x y)
          (cond ((eq? (car l) '+) (p 1 (+ x y)))         ;項を足す
                ((pair? (car l)) (p (* (ev-exp (car l)) x) y))  ;因子を掛ける
                (else (p (* (car l) x) y)))))))          ;因子を掛ける
  (ev++ l (lambda (x y) (+ x y))))       ;最後の項を足す
(define (ev-exp l)
  (ev+ (ev- (ev^ l))))
(define alist '                  ;assoc用の対応表
  ((a . 3) (b . 5) (c . 7) (d . 11) (e . 13) (f . 17) (g . 19) (h . 23) (i . 29)
   (j . 31) (k . 37) (l . 41) (m . 43) (n . 47) (o . 53) (p . 59) (q . 61)
   (r . 67) (s . 71) (t . 73) (u . 79) (v . 83) (w . 89) (x . 97) (y . 101)
   (z . 103)))
```

readlineの英小文字の処理は

```
((<= 97 char 122)
 (set! exp (cons (cdr (assoc (integer->symbol char) alist)) exp))
 (next))
```

のように変更した．これだと1000行のデータでも，7秒くらいで完了する．心配だったら別の組合せでもう一度実行することも可能である．zの9乗は60ビットを超えるが，bignumを装備しているLispなら恐くない．

参考文献

1）Kelsey, R., Clinger, W. and Ress, J.(ed.): Revised[5] Report on the Algorithmic Language Scheme, http://swissnet.ai.mit.edu/ftpdir/scheme-reports/r5rs.ps.

2）Raymond, E.(ed.): The New Hacker's Dictionary, MIT Press.

（平成15年6月24日受付）