記 号 処 理 28－4
（1984. 6. 15）

第三回LISPコンテストおよび第一回PROLOGコンテストの課題案

奥乃 博

日本電信電話公社 武蔵野電気通信研究所 情報通信基礎研究部 第二研究室
〒180 東京都武蔵野市緑町3—9—11 ☎(0422)59—3323

1 はじめに

1974年に第一回LISPコンテストが行われ、1978年に第二回LISPコンテストが行われてから早6年経った。その間に、コンテストで用いられたベンチマーク・プログラムはLISPのみならず色々なプログラミング言語処理系の性能評価にも使われている。この6年の間には人工知能分野の応用領域が拡大して『知識工学』という言葉が定着するようになった。このように人工知能が産業界で認知されるようになるにしたがって、LISPの要望が高まり数多くのLISP処理系が作成されたり、著名なLISPシステムが日本でも使用できるようになってきた。また、一昨年より始まった第五世代コンピュータ・プロジェクトでは、その核言語のプロトタイプとしてPROLOGを採用したことから世界的にPROLOGへの関心も高まっている。このPROLOGは『LISPに追いつけ、LISPを追い越せ』を掛け声に開発されていることからも、今回6年振りにLISPコンテストを行うにあったては、LISPのみならずPROLOGも取り上げることとし今後の討論の場を提供することにした。

今回のコンテストでは処理系作成者だけでなく、ユーザにも前回にまして意味のあるものにしたいと考えている。すなわち、処理系作成者に対しては、処理系作成技術の評価尺度を与えること、新しい処理系作成技術の提案の場とすること、今後の拡張の方向を見定めることを、また、ユーザに対しては、自分の目的に合った処理系を選択するための指針を与えること、および、現在使用している処理系の癖を知り、より効率のよいプログラムを作成するための目安を与えることを狙っている。

LISPとPROLOGのコンテストを行うときに注意しなければならないのは、ベンチマークの結果の評価法である[Gabriel 82]。つまり、単純に速度だけで評価するのではなく処理系の性能、あるいは、処理系が作成されている計算機のアーキテクチャとの関連でベンチマークの結果を評価しなければならない。というのは、処理系というものは、それを実現すべき計算機アーキテクチャという制約の下でできるだけ豊富な機能をできるだけ高速な処理速度で提供しようといういわば妥協の産物に他ならないからである。LISPが人工知能の分野で20年以上にわたって使用されてきた一つの理由として会話性をあげることができよう。INTERLISPは種々のLISPのなかにあって最も会話性の優れた処理系として知られているが、その処理速度は決して高速ではない。単なる速度比較だけではINTERLISPが『知識工学』を産み出す母体になったことは説明ができないであろう。

LISPコンテストの課題については、LISP処理系がすでに数多く出回っていることから、小規模で細かいベンチマークは避け中規模以上の応用指向のものとした。一方、PROLOGの課題については、処理系作成技法が未だ確定していないことから、やや細かいかとも思ったがICOTの希望もあり、低レベルのベンチマークを中心にし、さらに応用指向のものも入れるようにした。

課題案を決めるにあったっては、記号処理研究会の連絡委員の方々、Nilsonを始めとするスェーデンの研究者たち、あるいは、その他の多くの人達から意見をいただいた。また、J. A. Robinson教授はPROLOG用ベンチマークとして地理データベースの使用を快く承諾してくださった。電総研の中島秀之博士はLISP用ベンチマークとしてPortable PROLOGの使用を承諾してくださった。直接あるいは間接的に御協力いただいたこれら多くの人達に感謝します。

以下、2章では、LISPコンテストの課題案を、3章では、PROLOGコンテストの課題案を提示する。実施要領は2章の最後で提案する。

参考文献

竹内郁雄、LISP処理系コンテストの結果、記号処理5—3、情報処理学会、8月、1978.

Gabriel, R.P. and Masinter, L. M., Performance of Lisp Systems, *Proc. of 1982 ACM symposium on Lisp and Functional Programming*, ACM, Aug. 1982.

Wilk, P.F., The Production and Evaluation of a set of PROLOG Benchmarks,.Dept. of Artificial Intelligence, Univ. of Edinburgh, 1984.

2. LISPコンテスト課題案

```
; [1] function call/return
; **** Tarai ****
;      where number-manipulations may be replaced
;      by those restricted to integer if available

(defun tarai (x y z)
  (cond ((greaterp x y)
         (tarai (tarai (subl x) y z)
                (tarai (subl y) z x)
                (tarai (subl z) x y) ))
        (t y) ))

(defun tak (x y z)
  (cond ((not (lessp y x)) z)
        (t (tak (tak (subl x) y z)
                (tak (subl y) z x)
                (tak (subl z) x y) ))))

; Measure the following forms:
; [1-1:]
(tarai 8 4 0)          ; tarai is called 12605 times.
                       ;    with 9453 subl's and 9454 else parts.
; [1-2:]
(tarai 10 5 0)         ; tarai is called 343073 times.
                       ;    with 257304 subl's.
; [1-3:]
(tarai 12 6 0)         ; tarai is called 12604861 times.
                       ;    with 9453645 subl's.
; [1-4:]
(tak 18 12 6)          ; tak is called 63609 times.
                       ;    in honor of USA Lisp comunity

; [2] List manipulation  **** Tarai with list ****

(defun list-tarai (x y z)
  (cond ((lessp (car x) (car y))
         (list-tarai (list-tarai (copy (cdr x)) y z)
                     (list-tarai (copy (cdr y)) z x)
                     (list-tarai (copy (cdr z)) x y) ))
        (t y) ))

; [2-1:]
(list-tarai '(1 2 3 4 5 6 7 8 9) '(5 6 7 8 9) '(9))
                ; analogous to (tarai 8 4 0)
                ; list-tarai is called 12605 times.
                ;    with 9453 copy's and 25210 car's.

; **** Slow Reverse **** (c) Hideyuki Nakashima and Satoru Tomura
(defun srev (x)
  (cond ((null x) nil)
        (t (sapp (srev (cdr x)) (ncons (car x)))) ))

(defun sapp (x y)
  (cond ((null x) y)
        (t (sapp (srev (cdr (srev x)))
                 (cons (car (srev x)) y) ))))

; [2-2:]
(srev '(1 2 3 4 5))
; [2-3:]
(srev '(1 2 3 4 5 6))
```

```
; **** Quicksort ****

(defun qsort (l r)
  (cond ((null l) r)
        (t ((lambda (p)
              (qsort (car p)
                     (cons (car l) (qsort (cdr p) r) )))
            (partition (cdr l) (car l)) ))))

(defun partition (l x)
  (cond ((null l) (cons nil nil))
        (t ((lambda (p)
              (cond ((lessp (car l) x)
                     (cons (cons (car l) (car p)) (cdr p)) )
                    (t (cons (car p) (cons (car l) (cdr p))))) )
            (partition (cdr l) x) ))))

; ---- Common data definition ----

(setq list30 '( 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
               16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 ))

(setq list50
      '(27 74 17 33 94 18 46 83 65  2 32 53 28 85 99 47 28 82  6 11
        55 29 39 81 90 37 10  0 66 51  7 21 85 27 31 63 75  4 95 99
        11 28 61 74 18 92 40 53 59  8 ))

(setq list10 '(1 2 3 4 5 6 7 8 9 10))
(setq list100 (index 1 100 1))
(setq list1000 (index 1 1000 1))
(setq list10000 (index 1 10000 1))
(setq rlist10 '(10 9 8 7 6 5 4 3 2 1))
(setq rlist100 (index 100 1 -1))
(setq rlist1000 (index 1000 1 -1))
(setq rlist10000 (index 10000 1 -1))

; where index may be defined as follows:

(defun index (start end step)
  (do ((i start (+ i step)) result)
      ((or (and (minusp step) (< i end))
           (and (not (minusp step)) (< i end)) )
       (copy-list (nreverse result)) )   ; copy-list is for cdr-coding
      (setq result (cons i result)) ))

; [2-4:] Sort a list of 50 element.
;        Compare the performance of quicksort in Prolog.
(qsort list50 ())

; **** Naive reverse to compare with Prolog ****

(defun napp (x y)
  (cond ((pairp x) (cons (car x) (napp (cdr x) y)) )
        (t y) ))

(defun nrev (x)
  (cond ((pairp x) (napp (nrev (cdr x)) (list (car x))) )
        (t nil) ))

; [2-5:]
(nrev list30)
```

課題の配布方法

(1) ソースリスティング（各々約1500行）：請求して下さい。
(2) 磁気テープ：　磁気テープを送って下されば，課題を入れて返送致します。
　　フォーマット：TOPS-20 の Dumper フォーマット
　　VAX/VMS の種々のフォーマット
　　UNIX の tar フォーマット
　　[注] 御希望のフォーマットを必ず明記して下さい。
(3) フロッピーディスク：媒体をお送り下さい。課題を入れて返送致します。
　　フォーマット：8インチ，片面単密度，CP/M 形式のみ。

測定結果の集計方法

(1) コンピュータ・リーダブルな媒体（磁気テープかフロッピーディスク）にて結果をお送り下さい。なお，送られてきた媒体はお返しします。
(2) 使用した言語の概要，計算機システムの概要を記入すべき様式も配布する課題と共に入っております。また，日本語で記入できるようにハードコピーも添えておきますが，できるだけファイルで提供されるものに御記入下さい。

測定方法

(1) 組込みの時計以外のもの，例えば腕時計，ストップウォッチ等を使用された場合にはその旨を記入して下さい。
(2) ベンチマークの測定は次のものについて行って下さい。
　　[1] ベンチマークを文字通りに走らせたときの結果
　　[2] 各々の処理系で最高速となるように宣言を加えたり，関数の一部を置換したときの結果
　　[3][4] コンパイラがある場合にはインタプリタの [1] と [2] と同様について測定して下さい．それを各々 [3]，[4] とします。

　　[注] [2] と [4] については置換した関数，追加した宣言を明記して下さい。

(3) 短時間で終了するようなベンチマークについては，ループで何回か回して測定して下さい。その場合には，正確を期すために実行時間は本体が空のループとの差を取るようにして下さい（裏面参照）。
(4) 実行不能の場合には，その理由を記入して下さい。
　例えば，LISP の [6] のベンチマークは，数を属性として認めない処理系では動きません。
(5) PROLOG のベンチマークで数が大き過ぎて表現できない場合には桁数を減らして実行して下さい。

ベンチマークの表現方法

(1) LISP については，MacLisp，Interlisp，UCILisp，PSL で表現したものがありますので，最も近いものを御指定下さい。
(2) PROLOG については，DEC-10 Prolog，PROLOG/KR 風で表現したものがありますので，どちらがよいかを御指定下さい。

課題の追加

```
======== Addition of benchmarks ========
******** for Lisp ********

; [12] **** Yet another theorem-prover ****
; Rewrite rule base theorem-proving programs
;         written by Bob Boyer.

******** for Prolog ********

% [17] **** Slow reverse ****

srev([],[]).
srev([A|X],Z) :- srev(X,Y),sapp(Y,[A],Z).

sapp([],X,X).
sapp(X,Y,Z) :- srev(X,[A|RX]),
               srev(X,Dummy),  % necessary fo fairness with Lisp
               srev(RX,RRX),
               sapp(RRX,[A|Y],Z).

% [17-1:] srev-4
srev4 :- srev([1,2,3,4],X).

% [17-2:] srev-5
srev5 :- srev([1,2,3,4,5],X).

% [17-3:] srev-6
srev6 :- srev([1,2,3,4,5,6],X).

================ How to measure the execution time ================
**** for Lisp ****                                    (測定方法)

(defmarco benchmark (n &rest body)
  `(let (time1 time2 time3)
     (setq time1 (runtime))
     (loop for i from 1 to ,n ,@body)        ; main loop
     (setq time2 (runtime))
     (loop for i form 1 to ,n)               ; dummy loop
     (setq time3 (runtime))
     (- (+ time2 time2) time1 time3)))

That is,

(benchmark 10 (tarai 8 4 0)) is expanded to

(let (time1 time2 time3)
     (setq time1 (runtime))
     (loop for i from 1 to 10 (tarai 8 4 0))  ; main loop
     (setq time2 (runtime))
     (loop for i form 1 to 10)                ; dummy loop
     (setq time3 (runtime))
     (- (+ time2 time2) time1 time3)))


******** for Prolog ********

bench_srev4(N) :-
     statistics(runtime,[_,_]),!,
     loop_srev4(0,N),                         % main loop
     statistics(runtime,[_,T1]),!,
     loop_dummy(0,N),                         % dummy loop
     statistics(runtime,[_,T2]),
     T3 is T1-T2,
     write(T3), nl.

loop_dummy(N,N) :- !.
loop_dummy(I,N) :-
     I1 is I+1, !, loop_dummy(I1,N).

loop_srev4(N,N) :- !.
loop_srev4(I,N) :-
     I1 is I+1, srev4, !, loop_srev4(I1,N).
```

```
; **** Built-in (Non-/Destructive) Reverse ****
; Use the built-in destructive reverse function.
; If it is not provided by the system, use the following function:

(defun nreverse (l)
  (cond ((null l) nil)
        (t (do ((current (cdr l)) (previous l) next)
               ((null current) (rplacd l nil) previous)
             (setq next (cdr current))
             (rplacd current previous)
             (setq previous current current next) ))))

; [2-6:]
(reverse list30)
; [2-7:]
(nreverse list30)
; [2-8:]
(reverse list10)
; [2-9:]
(reverse list100)
; [2-10:]
(reverse list1000)
; [2-11:]
(reverse list10000)
; [2-12:]
(nreverse list10)
; [2-13:]
(nreverse list100)
; [2-14:]
(nreverse list1000)
; [2-15:]
(nreverse list10000)

; [3] String manipulation     **** Tarai with string ****

(defun string-tarai (x y z)
  (cond ((string-lessp (shead x) (shead y))
         (string-tarai (string-tarai (stail x) y z)
                       (string-tarai (stail y) z x)
                       (string-tarai (stail z) x y) ))
        (t y) ))
              ; where shead and stail must create new string

; [3-1:]
(string-tarai "abcdefghi" "efghi" "i")
                ; where string-tarai is called 12605 times.
                ; stail is called 9453 times.

; [4] Number consing **** Tarai with number ****

; [4-1:] Flonum
(tarai 9.0 5.0 1.0)
     ; This complexity is equivalent to that of tarai-4.
     ; where number-manipulations may be replaced
     ; by those restricted to flonum if available

; [4-2:] Bignum
(tarai big-x big-y big-z)
     ; This complexity is equivalent to that of tarai-4.
     ; where big-z should be greater than the least positive
     ; bignum, big-x = big-z + 8, and big-y = big-z + 4.
```

```
; [5] Array manipulation  **** Bubble sort ****
                          (アルゴリズムを下記に示す)
(defun bubblesort (ls)
  (prog (top bound)
        (setq bound 49)
        (loop for i fixnum from 0 to bound
              do (store (data i) (pop ls)) )
   loop (setq top 0)
        (loop for j fixnum from 0 to (1- bound)
              for x fixnum = (data j)
              do (cond ((> (setq x (data j)) (data (1+ j)))
                        (store (data j) (data (1+ j)))
                        (store (data (1+ j)) x)
                        (setq top j) )))
        (cond ((zerop top)
               (return (loop for j fixnum from 0 to 49
                             collect (data j) into x
                             finally (return x) ))))
        (setq bound top)
        (go loop) ))

; [5-1:] Sort 50 elements by bubble sort.
(bubblesort list50)

; [6] Property list     **** Generate sequence ****

(defun sequence (n)
  (cond ((get 'seq-table n) (get 'seq-table n))
        (t (putprop 'seq-table
              (- (sequence (subl n)) (sequence (- n 2)))
              n )
           (get 'seq-table (get 'seq-table n)) )))

; ---- Initialization of property list of sequence ----
(setplist 'seq-table '(-1 1 0 1 1  1))

; [6-1:] Calculate 100th number
(sequence 100)

; [7] Mapping function and non-local variable access
; **** BITA ****

(defun bita (a)
  (cond ((null (cdr a)) a)
        ((null (cddr a)) (list (cons (car a) (cons '$ (cdr a)))))
        (t (bitl (cdr a) (list (car a)))) ))

(defun bitl (x j)
  (cond ((null x) nil)
        (t (nconc
            (mapcan
             (function
              (lambda (k)
                (mapcar
                 (function
                  (lambda (l) (list l '$ k) ))
                 (bita j) )))
             (bita x))
            (bitl (cdr x) (append j (list (car x)))) ))))
```

```
; **** BITB ****

(defun bitb (a)                    ; a is non-local.
  (cond ((null a) nil)
        ((null (cdr a)) a)
        (t ((lambda (c)
              (setq a (list (car a) '$ (cadr a)))
              (mapcon
               (function (lambda (b) (g (car b))))
               (bitb (cdr c))) )
            a )
           )))

(defun g (b)
  (cond ((atom b) (list a))      ; a is defined in bitb
        (t (cons (list (car a) '$ b)
                 (mapcar
                  (function (lambda (a) (cons a (cdr b))))
                  (g (car b)) )))))

; [7-1:] BITA-5
(bita '(a b c d e))
; [7-2:] BITA-6
(bita '(a b c d e f))
; [7-3:] BITB-5
(bitb '(a b c d e))
; [7-4:] BITB-6
(bitb '(a b c d e f))

; [8] Sort

; **** Measure the performance of a built-in function sort ****

; [8-1:]
(sort list50 '<)
; [8-2:]
(sort list10 '<)
; [8-3:]
(sort list100 '<)
; [8-4:]
(sort list1000 '<)
; [8-5:]
(sort list10000 '<)
; [8-6:]
(sort rlist10 '<)
; [8-7:]
(sort rlist100 '<)
; [8-8:]
(sort rlist1000 '<)
; [8-9:]
(sort rlist10000 '<)

; [9] TPU
; *** TPU program ****
; Theorem Prover by Unity binary resulution
;         [See]  C-L. Chang  and  R.  C-T.  Lee,
;                Symbolic Logic  and  Mechanical Theorem Proving,
;                Academic Press, 1973.

; [9-1:] TPU-1
; [9-2:] TPU-2
; [9-3:] TPU-3
; [9-4:] TPU-4
; [9-5:] TPU-5
; [9-6:] TPU-6
; [9-7:] TPU-7
; [9-8:] TPU-8
; [9-9:] TPU-9

; [10] Portable Prolog

; Portable-PROLOG originally written by H. Nakashima
; transferred to ECL on   Monday, 9 November 1981

; This program may be modified in order to speed up its performance.

(declare (special file old-subst cue new-subst undo-list *subst*))

(defun prolog fexpr (file)
  (prog (result)
        (cond ((null file) (setq file t))
              (t (setq file (open file))))
        (cond ((eq file t)
               (print '(portable prolog on LISP))
               (terpri)))
   loop (cond ((eq file t) (print 'prl>)))
        (cond((eq (setq result (read-exec)) 'epilog)
              (return 'epilog))
             ((eq result 'syntax-error) nil)
             ((eq file t) (print (cond (result result)
                                       (t '*failure*)))))
        (go loop)))

(defun read-exec ()
  (prog (sign)
    (setq sign (read file))
    (return (cond ((eq sign '/+)(define-clause (read-rest)))
                  ((eq sign '/-)(refute-clause (read-rest)))
                  ((memq sign '(end ok stop epilog)) 'epilog)
                  (t (read-error sign))))))

(defun read-rest ()
  (prog (form rest)
    (setq form (read file))
    (return (cond ((atom form)(read-error form))
                  ((eq (setq rest (read-sign)) 'syntax-error)
                   'syntax-error)
                  (t (cons form rest))))))

(defun read-sign ()
  (prog (sign)
    (setq sign (read file))
    (return (cond((eq sign '/-)(read-rest))
                 ((eq sign ':) nil)
                 (t (read-error sign))))))

(defun read-error (obj)
  (print (list 'syntax 'error obj)) 'syntax-error)
```

```
(defun first macro (x) (cons 'car (cdr x)))
(defun second macro (x) (cons 'cadr (cdr x)))
(defun third macro (x) (cons 'caddr (cdr x)))

(defun define-clause (clause)
  (cond ((eq clause 'syntax-error) 'syntax-error)
        (t (prog (definition)
             (setq definition (get (caar clause) 'prolog))
             (cond ((null definition)
                    (putprop (caar clause)
                             (cons clause nil) 'prolog))
                   (t (nconc definition (cons clause nil))))
             (return 'defined)))))

(defun refute-clause (clause)
  (cond ((eq clause 'syntax-error) 'syntax-error)
        (t (refutes clause (cons nil nil) (cons nil nil) nil))))

(defun refutes (clause new-subst old-subst cue)
  (cond ((null clause)
         (cond ((null cue) '*proved*)
               (t (refutes (car (first cue)) (cons nil nil)
                           (cdr (first cue)) (cdr cue)))))
         (t (refute clause))))

(defun refute (clause)
  (prog (definition undo-list result)
    (setq definition (get (caar clause) 'prolog))
   refuteloop
    (or definition
        (return
         (cond ((and
                 (setq result (try-sys (car clause) old-subst))
                 (refutes (cdr clause) (cons nil nil) old-subst cue))
                result)
               (t (undo) nil))))
    (cond ((and (unify (car clause) old-subst
                       (caar definition) new-subst)
                (refutes (cdr (first definition)) (cons nil nil)
                         new-subst
                         (cons (cons (cdr clause) old-subst) cue)))
           (return '*proved*))
          (t (undo)))
    (setq definition (cdr definition))
    (go refuteloop)))

(defun undo()
  (prog()
   loop (cond ((null undo-list) (return nil))
              (t (rplacd (car undo-list) (cddar undo-list))
                 (setq undo-list (cdr undo-list))
                 (go loop)))))
```

```
(defun try-sys (form subst)
  (cond ((eq (car form) 'call)
         (apply (fetch-value (second form) subst)
                (mapcar '(lambda (x) (fetch-value x subst))
                        (cddr form)))
         '*proved*)
        ((eq (car form) 'eval)
         (cond ((unify
                 (apply (fetch-value (car (second form)) subst)
                        (mapcar '(lambda (x) (fetch-value x subst))
                                (cdr (second form))))
                 subst (third form) subst)
                '*proved*)
               (t nil)))
        ((eq (car form) 'end) 'epilog)
        ((eq (car form) 'delete)
         (mapcar '(lambda (x) (remprop (fetch-value x subst) 'prolog))
                 (cdr form))
         'deleted)
        (t nil)))

(defun unify (x x-subst y y-subst)
  (cond ((var? x)
         (cond ((assigned? x x-subst)
                (unify (fetch x x-subst) *subst* y y-subst))
               (t (link x x-subst y y-subst))))
        ((var? y) (unify y y-subst x x-subst))
        ((atom x) (eq x y))
        ((atom y) nil)
        ((unify (car x) x-subst (car y) y-subst)
         (unify (cdr x) x-subst (cdr y) y-subst))
        (t nil)))

(defun var? (x) (and (symbolp x) (eq (character x) '*)))

(defun assigned? (x subst) (assoc x (cdr subst)))

(defun fetch (x subst)
  (setq *subst* subst)
  (cond ((var? x)
         (prog (v)
           (setq v (assoc x (cdr subst)))
           (cond ((null v) (return x)))
           (setq *subst* (cddr v))
           (return (fetch (second v) (cddr v)))))
        (t x)))

(defun fetch-value (x subst)
  (cond ((var? x)
         (prog (v)
           (setq v (assoc x (cdr subst)))
           (cond ((null v) (return x))
                 (t (return (fetch-value (second v) (cddr v)))))))
        ((atom x) x)
        (t (cons (fetch-value (car x) subst)
                 (fetch-value (cdr x) subst)))))
```

```
(defun link (x x-subst y y-subst)
  (setq undo-list
        (cons (rplacd x-subst
                      (cons (cons x (cons (fetch y y-subst)
                                          *subst*))
                            (cdr x-subst)))
              undo-list)))

(defun character (x) (cond ((null x) nil)
                           (t (car (explode x)))))

; HOW TO USE Portable-PROLOG
;       (prolog {file-name})
;               (prolog)        input from terminal
;               (prolog {(directory)} file-name {extension})
;                       A file must be terminated by "stop".
;
; The Syntax of Portable-Prolog
;       +(fl {pl}...):                                  asserts a fact
;       +(fl {pl}...)-(f2 {p2}...) ... -(fn {pn}...):   asserts a theorem
;       -(fl {pl}...) ... -(fn {pn}...):                        query
;       -(end):, stop, end, epilog, or end              end of prolog session
;
; Convensions
;       *variable               represents a variable
;       -(call function parameter...)   apply a function to parameters
;                                       with a success of unification
;       -(eval form clause)     unifies the value of form with clause

; ---- Now Enter the Portable-Prolog ----

; Problem 1. Naive reverse

+(nreverse (*x . *l0) *l)-(nreverse *l0 *ll)-(concat *ll (*x) *l):
+(nreverse () ()):

+(concat (*a . *x) *y (*a . *z))-(concat *x *y *z):
+(concat () *y *y):

; [10-1:] Reverse 15 element.

+(rev15 *x)-(nreverse ( 1  2  3  4  5  6  7  8  9 10
                       11 12 13 14 15 ) *x):

; Problem 2. Quicksort.

+(qsort (*x . *l) *r *r0)
      -(partition *l *x *ll *l2)
      -(qsort *l2 *rl *r0)
      -(qsort *ll *r (*x . *rl)):
+(qsort () *r *r):

+(partition (*x . *l) *y (*x . *ll) *l2)
      -(eval (not (greaterp *x *y)) t)
      -(partition *l *y *ll *l2):
+(partition (*x . *l) *y *ll (*x . *l2))
      -(eval (greaterp *x *y) t)
      -(partition *l *y *ll *l2):
+(partition () *a () ()):

; [10-2:] Sort 20 element.

+(sort20 *x)-(qsort (27 74 17 33 94 18 46 83 65  2
                     32 53 28 85 99 47 28 82  6 11) *x ()):

; !!!!! How to know the time !!!!!
;
; It is necessary to declare two global variables, time1 and time2.

-(eval (time) *time1) <clauses to be measured>
       -(eval (time) *time2) -(call print (*time2 *time1)):

[11] Differentiation by data-driven programming

; Main routine

(defun diff (exp x)
  (cond ((eq exp x) 1)
        ((atom exp) 0)
        ((atom (car exp))
         ((lambda (fn)
            (cond ((and fn (= (length exp) 3))
                   (apply fn (list (cdr exp) x)) )
                  (t 'error) ))
          (get (car exp) 'diff-fn) ))))

; Define how to differentiate to each operator

(putprop '+ 'plus-diff 'diff-fn)
(putprop '- 'difference-diff 'diff-fn)
(putprop '* 'times-diff 'diff-fn)
(putprop '// 'quotient-diff 'diff-fn)
(putprop '** 'expt-diff 'diff-fn)

(defun plus-diff (sexp x)
  (simp-plus (diff (car sexp) x) (diff (cadr sexp) x)) )

(defun simp-plus (arg1 arg2)
  (cond ((and (numberp arg1) (numberp arg2))
         (plus arg1 arg2) )
        ((eq arg1 0) arg2)
        ((eq arg2 0) arg1)
        (t (list '+ arg1 arg2)) ))

(defun difference-diff (sexp x)
  (simp-difference (diff (car sexp) x) (diff (cadr sexp) x)) )

(defun simp-difference (arg1 arg2)
  (cond ((and (numberp arg1) (numberp arg2))
         (difference arg1 arg2) )
        ((eq arg1 0) (list '* arg2 -1))
        ((eq arg2 0) arg1)
        ((numberp arg2) (list '+ arg1 (times arg2 -1)))
        (t (list '- arg1 arg2)) ))

(defun times-diff (sexp x)
  (simp-plus (simp-times  (diff (car sexp) x) (cadr sexp))
             (simp-times (car sexp) (diff (cadr sexp) x)) ))
```

実　施　要　領

課題の配布方法　(1)ソースリスティング(約3K行)
(2)磁気テープ(TOPS-20 or VAX/VMSで利用可能な形式)　(3)フロッピーディスク(CP/M形式)

応募時の記入事項

(1) LISPかPROLOGシステムの概要
- 作成者名，所属　・データ型，関数の種類

(2) 測定した計算機の概要
- 汎用システムは機種名とメモリ量
- 専用システムはハードウェアの特徴

(3) ベンチマーク測定結果
- 実行不能のときはその理由も記入して下さい。
- ガーベッジコレクタの時間は含まれているか否かを明記して下さい。(両方のデータが望ましい)
- コンパイラが備っているときには，インタプリタとコンパイラと両方のデータを測定して下さい。
- コンパイルにいくつかのモードがあるときには，モードの説明と各モードでの測定をお願いします。

(4) 結果は計算機可読な形式で報告して下さい。
すなわち(1)磁気テープか(2)フロッピーディスクでお願いします。

集計締切　59年12月末

報告集　60年3月末に英文の詳細報告集を作成予定。概要報告は本研究会と本誌にて行う予定。

海外とのコンタクト　米国，欧州各国，韓国へ案内を出す。特に，米国についてはARPAネットを通じて宣伝を行いたい。

```
(defun simp-times (arg1 arg2)
  (cond ((and (numberp arg1) (numberp arg2))
         (times arg1 arg2) )
        ((or (eq arg1 0) (eq arg2 0)) 0)
        ((eq arg1 1) arg2)
        ((eq arg2 1) arg1)
        (t (list '* arg1 arg2)) ))

(defun quotient (sexp x)
  (diff (list '* (car sexp) (list '** (cadr sexp) -1)) x) )

(defun expt-diff (sexp x)
  (cond ((= (cadr sexp) 0) 0)
        ((= (cadr sexp) 1) (diff (car sexp) x))
        (t (diff (list '* (car sexp)
                       (simp-expt (car sexp) (subl (cadr sexp))) )
                 x ))))

(defun simp-expt (arg exp)
  (cond ((zerop exp) 1)
        ((= exp 1) arg)
        (t (list '** arg exp)) ))

; [11-1:]
(diff (diff '(+ (+ (+ (** x 3) (* 3 (** x 2))) (* 3 x)) 1) 'x) 'x)

; This must return (+ (+ (+ x x) (+ (+ x x) (* x 2))) 6) .

; [11-2:] d(6) (x - 1)**6/dx
(diff (diff (diff (diff (diff (diff
                        '(** (- x 1) 6) 'x) 'x) 'x) 'x) 'x) 'x)

; This must result in 720.

; ******** That's all ********
```

お　願　い

① 処理系作成者のみならず，ユーザ自身が
② 出来るだけ多くのLISPとPROLOGについて
③ 種々な計算機で測定しましょう。

3. PROLOG コンテスト課題案

```
% **** Prolog Instruction Level Benchmar
% [1] Unification of atoms

p11(a).
p12(a,a,a,a,a).
p13(a,a,a,a,a,a,a,a,a,a).

% [1-1:] Arity of one
% [1-2:] Arity of five
% [1-3:] Arity of ten
% [1-4:] Arity of fifteen

q11 :- p11(a).
q12 :- p12(a,a,a,a,a).
q13 :- p13(a,a,a,a,a,a,a,a,a,a).

% [2] Unification of variables

p21(X).
p22(X,X,X,X,X).
p23(X,X,X,X,X,X,X,X,X,X).

% [2-1:] Arity of one
% [2-2:] Arity of five
% [2-3:] Arity of ten
% [2-4:] Arity of fifteen

q21 :- p21(X).
q22 :- p22(X,X,X,X,X).
q23 :- p23(X,X,X,X,X,X,X,X,X,X).

% [3] Unification of constant structure

p31(f(a)).
p32(f(a),f(a),f(a),f(a),f(a)).

% [3-1:] Arity of one
% [3-2:] Arity of five
% [3-3:] Arity of ten
% [3-4:] Arity of fifteen

q31 :- p31(f(a)).
q32 :- p32(f(a),f(a),f(a),f(a),f(a)).

% [4] Unification of structures with va

p41(f(X)).
p42(f(X),f(X),f(X),f(X),f(X)).

% [4-1:] Arity of one
% [4-2:] Arity of five
% [4-3:] Arity of ten
% [4-4:] Arity of fifteen

q41 :- p41(f(X)).
q42 :- p42(f(X),f(X),f(X),f(X),f(X)).

% [5] Unification of variables with str

p51(f(X)).
p52(f(X),f(X),f(X),f(X),f(X)).

% [5-1:] Arity of one
% [5-2:] Arity of five
% [5-3:] Arity of ten

q51 :- p51(X).
q52 :- p52(X,X,X,X,X).

% [6] Unification of variables and str

p61(X).
p62(X,X,X,X,X).
p63(X,X,X,X,X,X,X,X,X,X).

% [6-1:] Arity of one
% [6-2:] Arity of five
% [6-3:] Arity of ten

q61 :- p61(f(X)).
q62 :- p62(f(X),f(X),f(X),f(X),f(X)).

% [7] **** Clause Call/Return ****

p71(X) :- q71(X).
q71(X).

p72(X) :- q72(X).
q72(X) :- fail.
p72(X).

p73(X) :- fail.
p73(X) :- fail.
p73(X) :- fail.
p73(X) :- fail.
p73(X) :- fail.
p73(X) :- fail.
p73(X) :- fail.
p73(X) :- fail.
p73(X) :- fail.
p73(X) :- fail.
p73(X).

p74(X) :- q74(X),r74(X).
p74(X) :- a74(X),b74(X).
q74(X) :- s74(X),r74(X).
q74(X) :- s74(X),r74(X).
s74(X) :- r74(X).
s74(X) :- r74(X).
r74(X) :- fail.
r74(X) :- fail.
a74(X) :- s74(X),r74(X).
a74(X) :- b74(X),b74(X).
b74(X) :- fail.
b74(X).

% [7-1:] Deterministic simple call
% [7-2:] Nondeterministic simple call
% [7-3] Sallow backtracking
% [7-4] Deep backtracking

ck71 :- p71(X).
ck72 :- p72(X).
ck73 :- p73(X).
ck74 :- p74(X).

% [8] **** Clause Indexing ****
/*
 The following clauses are part of places database.
   country(moscow,ussr). ===>220 clauses. */

% [8-1:] Get the first clause with primary key.
% [8-2:] Get the first clause.
% [8-3:] Get the last clause with primary key.
% [8-4:] Get the last clause.
% [8-5:] Get the middle clause with primary key.
% [8-6:] Get the middle clause.

q81 :- country(moscow,X).
q82 :- country(X,ussr).
q83 :- country(gangtok,X).
q84 :- country(X,sikkim).
q85 :- country(manila,X).
q86 :- country(X,philippines).

% [9] **** Naive Reverse ****

nreverse([X|L0],L) :- nreverse(L0,L1), concatenate(L1,[X],L).
nreverse([],[]).

concatenate([X|L1],L2,[X|L3]) :- concatenate(L1,L2,L3).
concatenate([],L,L).

list30( [ 1, 2, 3, 4, 5, 6, 7, 8, 9,10,
         11,12,13,14,15,16,17,18,19,20,
         21,22,23,24,25,26,27,28,29,30 ]).

% [9-1:] Reverse a list of 30 elements.
%        This comptation involves 496 LI (Logical Inference).
q91(X) :- list30(L), nreverse(L,X).

% [10] **** Quick sort ****

qsort([X|L],R,R0) :- partition(L,X,L1,L2),
        qsort(L2,R1,R0), qsort(L1,R,[X|R1]).
qsort([],R,R).

partition([X|L],Y,[X|L1],L2) :- X =< Y, !, partition(L,Y,L1,L2).
partition([X|L],Y,L1,[X|L2]) :-  partition(L,Y,L1,L2).
partition([],_,[],[]).

list50( [27,74,17,33,94,18,46,83,65, 2,
         32,53,28,85,99,47,28,82, 6,11,
         55,29,39,81,90,37,10, 0,66,51,
          7,21,85,27,31,63,75, 4,95,99,
         11,28,61,74,18,92,40,53,59, 8 ]).

% [10-1:] Sort 50 elements.
%         The complexity of this computation is 609 LI.
q101(X) :- list50(L), qsort(L,X,[]).
```

```
% [11] **** Tree traversing ****

/* Bigapp : A non-determinate computation involving structure
  accessing and lots of backtracking.  Written by Paul F. Wilk. */

conslist(0 ,[]).
conslist(N,[1,2,3,4,5,6,7,8,9,0,1,2,3,4,5,6,7,8,9,0,1,2,3,4,5|L])
     :-   N > 0, N1 is N - 25, conslist(N1,L).

concat([],L,L).
concat([X|L1],L2,[X|L3]) :- concat(L1,L2,L3).

% [11-1:] Measure the time of concat in q111.
q111 :- conslist(1000,L), !, concat(L1,L2,L), fail.

% [12] ******** Lisp interpreter in Prolog ********
%                      Micro Lisp in Prolog

apply(S,car,[[X|_]|_],X) :- !.
apply(S,cdr,[[_|Y]|_],Y) :- !.
apply(S,atom,[[_|_]|_],[]) :- !.
apply(S,atom,[X|_],t) :- atom(X).
apply(S,cons,[X,Y|_],[X|Y]) :- !.
apply(S,eq,[X,X|_],t) :- !.
apply(S,eq,[X,Y|_],[]) :- neq(X,Y).
apply(S,null,[[]|_],t) :- !.
apply(S,subl,[X|_],Y) :- !, Y is X-1.
apply(S,addl,[X|_],Y) :- !, Y is X+1.
apply(S,plus,[X,Y|_],V) :- !, V is X+Y.
apply(S,difference,[X,Y|_],V) :- !, V is X-Y.
apply(S,times,[X,Y|_],V) :- !, V is X*Y.
apply(S,greaterp,[X,Y|_],t) :- X>Y, !.
apply(S,greaterp,[_,_|_],[]).
apply(S,zerop,[0|_],t) :- !.
apply(S,zerop,[_|_],[]).
apply(S,[lambda,X,Y|_],R,V) :- !, bind(S,X,R,S1), eval(S1,Y,V).
apply(S,[label,X,Y|_],R,V) :- !, bind(S,[X],[Y],S1), apply(S1,Y,R,V).
apply(S,A,R,V) :- value(S,A,W), apply(S,W,R,V), !.

eval(S,[quote,X|_],X) :- !.
eval(S,[cond|X],V) :- !, evcon(S,X,V).
eval(S,[X|Y],V) :- !, evlis(S,Y,W), apply(S,X,W,V).
eval(S,A,V) :- value(S,A,V), !.

evcon(S,[[X,Y|_]|_],V) :- eval(S,X,W), neq(W,[]), !, eval(S,Y,V).
evcon(S,[_|U],V) :- evcon(S,U,V), !.

evlis(S,[],[]) :- !.
evlis(S,[X|Y],[U|V]) :- !, eval(S,X,U), evlis(S,Y,V).

bind(S,[],[],S) :- !.
bind(S,[A|X],[V|Y],S1) :- bind([[A|V]|S],X,Y,S1), !.

value([[A|V]|S],A,V) :- !.
value([X|S],A,V) :- value(S,A,V), !.

atom([X|Y]) :- !, fail.
atom(X).

neq(X,X) :- !, fail.
neq(X,Y).

lisp(X,V) :-
   eval([[[],[t|t],[f],
         [reverse|[lambda,[l],[rev,l,[]]]],
         [rev|[lambda,[l,m],
                 [cond,[[atom,l],m],
                       [t,[rev,[cdr,l],[cons,[car,l],m]] ]]]
        ],X,V).

% ---- Lisp programs ----

% [12-1:] TARAI-3
tarai3(V) :-
 lisp([[label,tarai,
         [lambda,[x,y,z],
             [cond,[[greaterp,x,y],
                    [tarai,[tarai,[subl,x],y,z],
                           [tarai,[subl,y],z,x],
                           [tarai,[subl,z],x,y]]],
                   [t,y] ]]],
       [quote,6],[quote,3],[quote,0] ], V).

% [12-2:] Fibonacci number for 10
fib10(V) :-
 lisp([[label,fib,
         [lambda,[n],
            [cond,[[eq,n,[quote,1]],[quote,1]],
                  [[eq,n,[quote,2]],[quote,1]],
                  [t,[plus,[fib,[subl,n]],[fib,[difference,n,[quote,2]]]]]
            ]]],
       [quote,10]], V).

% [12-3:] Reverse a list of 30 elements
reverse30(V) :-
 lisp([reverse,[quote,[1,2,3,4,5,6,7,8,9,0,1,2,3,4,5,6,7,8,9,0,
                       1,2,3,4,5,6,7,8,9,0]]], V).

% [13] **** Eight Queens ****

queen(N,L) :- generate(N,L1), try(N,L1,[],L,[],[]).

generate(0,[]) :- !.
generate(N,[N|L]) :- N1 is N-1, generate(N1,L).

try(_,[],L,L,_,_) :- !.
try(M,S,L1,L,C,D) :-
    select(S,A,S1), C1 is M+A, notmem(C1,C), D1 is M-A,
    notmem(D1,D), M1 is M-1, try(M1,S1,[A|L1],L,[C1|C],[D1|D]).

select([A|L],A,L).
select([A|L],X,[A|L1]) :- select(L,X,L1).

notmem(_,[]) :- !.
notmem(A,[B|L]) :- A=\=B, notmem(A,L).

% [13-1:] Get the first solution.
q131 :- queen(8,L).

% [13-2:] Get all the solution.
q132 :- queen(8,X), fail.
```

```
% [14] Differentiation
%          This program is the same as that written in Lisp.

:-op(300,xfy,^).

d(U+V,X,DY) :- !, d(U,X,DU), d(V,X,DV), simp_plus(DU,DV,DY).
d(U-V,X,DY) :- !, d(U,X,DU), d(V,X,DV), simp_diff(DU,DV,DY).
d(U*V,X,DY) :- !, d(U,X,DU), d(V,X,DV),
   simp_time(DU,V,D1), simp_time(U,DV,D2), simp_plus(D1,D2,DY).
d(U^0,X,0) :- !.
d(U^1,X,DY) :- !, d(U,X,DY).
d(U^N,X,DY) :- !, N1 is N-1, simp_expt(U,N1,Y), d(U*Y,X,DY).
d(X,X,1) :-!.
d(C,X,0) :- atomic(C), C \== 0.

simp_plus(X,Y,Z) :- integer(X), integer(Y), !, Z is X+Y.
simp_plus(0,Y,Y) :- !.
simp_plus(X,0,X) :- !.
simp_plus(X,Y,X+Y).

simp_diff(X,Y,Z) :- integer(X), integer(Y), !, Z is X-Y.
simp_diff(0,Y,-1*Y) :- !.
simp_diff(X,0,X) :- !.
simp_diff(X,Y,X+V) :- integer(Y), !, V is -Y.
simp_diff(X,Y,X-Y).

simp_time(X,Y,Z) :- integer(X), integer(Y), !, Z is X*Y.
simp_time(_,0,0) :- !.
simp_time(0,_,0) :- !.
simp_time(X,1,X) :- !.
simp_time(1,Y,Y) :- !.
simp_time(X,Y,X*Y).

simp_expt(_,0,1) :- !.
simp_expt(X,1,X) :- !.
simp_expt(X,N,X^N).

% [14-1:]
ql41(DF) :- d(x^3+3*x^2+3*x+1,x,DG),d(DG,x,DF).
% [14-2:]
ql42(DF) :-
   d((x-1)^5,x,D1),d(D1,x,D2),d(D2,x,D3),d(D3,x,D4),d(D4,x,DF).

% [15] **** Reversible functions ****

sum(X,0,X).
sum(X,s(Y),s(Z)) :- sum(X,Y,Z).
prod(X,0,0).
prod(X,s(Y),Z) :- prod(X,Y,W), sum(W,X,Z).
fact(0,s(0)).
fact(s(X),Z) :- fact(X,Y), prod(s(X),Y,Z).

fib(0,s(0)).
fib(s(0),s(0)).
fib(s(s(N)),V) :- fib(s(N),X), fib(N,Y), sum(X,Y,V).

% [15-1:]
solve(Y) :- fib(X,s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(0)))))))))))))))),
            fact(X,Y).
```

```
% [16] **** Database Manipulations ****
%              This database is created by J. A. Robinson et al.

% [16-1:] Which country's capital is Tokyo? and its GNP ?
db1(X,G) :- country(tokyo,X).

% [16-2:] What are the cities in Japan?
db2(S) :- setof(C, country(C,japan), S).

% [16-3:] Which far-east countres is landlocked?
db3(C) :- region(C,far_east), iscountry(C), landlocked(C).
% [16-4:]
db4(S) :- setof(C, db3(C), S).

% [16-5:] Which countries border two seas?
db5(C) :- iscountry(C), setof(X, border_sea(C,X), S), size(S,2).
border_sea(C,X) :- borders(C,X), open_water(X).
size([_,_],2).
% [16-6:]
db6(T) :- setof(C, db5(C),T).

% [16-7:] Which country bordering the Mediterranean borders a country
% that is bordered by a country whose population exceeds the population
% of the United State?
db7(C) :-
       population(india,Y), borders(C,mediterranean_sea), iscountry(C),
       borders(C,C1), iscountry(C1), borders(C1,C2),
       population(C2,X), X>Y.
% [16-8:]
db8(S) :- setof(C, db7(C), S).

% [16-9:] What are the oil production figures for the non-Arab OPEC
% countries in the year 1975?
db9(S) :- setof([C,P], oil_production(C,P), S).
oil_production(C,P) :-
  belongs(C,opec), \+belongs(C,arab-league), produces(C,oil,P,1975).

% [16-10:] Which is the ocean that borders African countries and that
% borders European countries?
db10(X) :-
  open water(X), borders(X,C), region(C,africa), iscountry(C),
  borders(X,C1), region(C1,europe), iscountry(C1).

% ---- Definition of the database ----
/* If numerical values cannot be represented in your system,
   chang the format.  For example, DEC-10 Prolog cannot
   express more than 2^18 or floating numbers. */

% population(afghanistan,20900000). ===> 161 clauses.
% adjoins(canada,usa). ===> 313 clauses.
% open_water(atlantic_ocean). ===> 40 clauses.
% country(moscow,ussr). ===>220 clauses.
% region(canada,north_america). ===> 162 clauses.
% produces(ussr,oil,491,1975). ===> 252 clauses.
% belongs(canada,nato). ===> 177 clauses.

iscountry(X):-country(Y,X).
landlocked(X):- iscountry(X), \+(landlocked_test(X)).
landlocked_test(X):-borders(X,Z),open_water(Z).
borders(X,Y):-adjoins(X,Y).
borders(X,Y):-adjoins(Y,X).
% ******** That's all ********
```