CS 481 — AI —Project 2 GP Arithmetic Ideas

```
In office hours, I was asked how one might go about generating random
critters for the GP project.  There are many ways.  Here is one sample.
Note, when I say "pick" I mean randomly pick.

How to pick a simple list:
Algo A1:
1. Pick an operation
2. Pick a number of arguments (eg, from 1 to 4)
  NB, Lisp arithmetic ops usually take any number of arguments
  EX:  (+) ==> 0;  (*) ==> 1;  (-) ==> 0
3. Pick a random arg value for each arg from integers and var names
  If you want, you can ignore how many variable names are in the list
4. Build the list.
Now, you can build a random simple list.
EX: (+ x 3 2)

How to build a deep-ish list:  Ex: (+ x 3 2)
Algo A2:
Given a list:
1. Pick an argument to replace  Ex: nth 2
2. Pick a new simple list  Ex: (* -4 z)
3. Replace the arg with the new simple list  Ex: (+ x (* -4 z) 2)
3'. Alt: Build a new list from the old list parts and the new simple list.
Now, you can build a random 2-level list.

To build a slightly deeper list, in algo A2 step 2, you could pick a
deeper list, either recursively or iteratively.

Next, how to pick a cross-over point in a deep-ish list:
Given a deep-ish list:
Algo A3:
1. Pick an argument spot (in length of the list)
2. If it is the op (nth 0), then it must be swapped with another op
Else, its not the op, so:
3. If the picked spot's arg is not a list, you're done
Else, the arg spot is a list (a sublist)
4. Decide (eg, flip a coin) if you want to dive into the arg's list to
  find the spot.  If so, you can call yourself recursively.

Note: rplaca and rplacd are Lisp “surgery” operations.  Usually a “change” in a
list actually produces a new list leaving the prior list unchanged - and,
importantly, maybe sharing a tail part of the prior list.  Rplaca is a way of
replacing the car of a list without building a new list.  Ditto rplacd for
replacing the cdr.
```

```
;; Simple supt fcns.
;; TOC
cell-count (rt)
  "Return the number of nodes/cells in the tree.  Skip non-cells."
tree-nth (rnth rtree)
  "Return the DFS N-th subtree/elt in the given tree."
tree-nth-cell (rnth rtree)
  "Return the DFS N-th cell in the given tree: 1-based."
random-tree-cell (rtree)
  "Return random cell in the tree, but not the whole tree."
make-kid (rmom rtgt rnew)
  "Return kid: copy of mom with tgt cell replaced by given new cell, or nil."
test-make-kid (rtree)
  "Test make-kid with random tgt cell and fixed replacement list."

;; Illustrations of how some GP Framework things could be done.
;; TOC
get-front-upto-nth ( rn rlist )
  "Return list head from 0-th thru N-th elt."
get-score (rcritter)
  "Get score for critter.  Dummy fcn: just its return length."
score-pop ( rpop )
  "Create Pop-Scored pairs, like (Score Critter), given Pop list of critters."
safe-sort-scored-pop ( rscored-pop )
  "Return a sorted list of scored-critter pairs.  Don't change given list."
get-pop-from-scored (rscored-pop)
  "Return just the Pop of critters from the Scored Pop."

;; ------------------------------------------------------------ cell-count ----
(defun cell-count (rt)
  "Return the number of nodes/cells in the tree.  Skip non-cells."
  (cond
   ((null rt) 0)
   ((not (listp rt)) 0)
   (t (let ((cc (length rt)))
        (+ cc (apply #'+ (mapcar #'cell-count rt)))))))
;; Tests
;;(cell-count '(a b c))
;;3
;;(cell-count '(a (b) c))
;;4
;; (cell-count '(a))
;; 1

;; -------------------------------------------------------------- tree-nth ----
(defun tree-nth (rnth rtree)
  "Return the DFS N-th subtree/elt in the given tree."
  (let ((size (cell-count rtree)))
    ;; (print (list :dbga rnth size (car rtree)))
    (cond
     ((not (integerp rnth)) nil)
     ((not (listp rtree)) nil) ;; Not a tree?
```

```
      ((null rtree) nil) ;; No tree elts?
      ((= 1 rnth) (car rtree)) ;; 1st elt of list is its car subtree.
      ((>= 0 rnth) nil) ;; Nth 0 or negative?
      ((> rnth size) nil) ;; N-th beyond tree's end?
      ((= 1 size) (car rtree)) ;; 1st elt is Tree's car.
      (t ;; Elt is in car subtree or cdr "subtree".
       (setq rnth (1- rnth)) ;;Account: Elt isn't the current (car+cdr's) node.
       (let ((size1 (cell-count (car rtree))))
         ;; (print (list :dbgb rnth size1 (car rtree)))
         (cond
          ((>= 0 size1) (tree-nth ;; No car subtree.
                         rnth
                         (cdr rtree))) ;; Find elt in the cdr subtree.
          ((<= rnth size1) (tree-nth ;;  Elt is in car subtree.
                            rnth
                            (car rtree))) ;; Find elt in the car subtree.
          (t (tree-nth ;; Elt is in cdr subtree.
              (- rnth size1) ;; Account for skipping car subtree.
              (cdr rtree)))))))) ;; Skip car subtree.
;; Tests
;; (tree-nth 1.3 '(a))
;; nil
;; (tree-nth 1 nil)
;; nil
;; (tree-nth 1 'a)
;; nil
;; (tree-nth 1 '(a))
;; a
;; (tree-nth 1 '(a b c))
;; a
;; (tree-nth 1 '((a) b c))
;; (a)
;; (tree-nth 2 '(a b c))
;; b
;; (tree-nth 2 '(a (b) c))
;; (b)
;; (tree-nth 3 '(a b c))
;; c
;; (tree-nth 4 '(a b c))
;; nil
;; (tree-nth 3 '(a (b) c))
;; b
;; (tree-nth 3 '((a f) b c))
;; f
;; (tree-nth 4 '((a f) b c))
;; b
;; (tree-nth 5 '((a f) b c))
;; c
;; (tree-nth 6 '((a f) b c))
;; nil
;; (tree-nth 6 '((a f) b c))
;; nil
;; (tree-nth 6 '((a f) ((b)) c))
;; b
;; (tree-nth 7 '((a f) ((b)) c))
;; c

;; ------------------------------------------------------------ tree-nth-cell ----
(defun tree-nth-cell (rnth rtree)
  "Return the DFS N-th cell in the given tree: 1-based."
  (let ((size (cell-count rtree)))
    ;;(print (list :dbga rnth size (car rtree)))
    (cond
     ((not (integerp rnth)) nil)
     ((not (listp rtree)) nil) ;; Not a tree?
     ((null rtree) nil) ;; No tree elts?
     ((= 1 rnth) rtree) ;; 1st elt of list is the tree, itself.
     ((>= 0 rnth) nil) ;; Nth 0 or negative?
     ((> rnth size) nil) ;; N-th beyond tree's end?
     (t ;; Elt is in car subtree or cdr "subtree".
      (setq rnth (1- rnth)) ;;Account: Elt isn't the current (car+cdr's) node.
      (let ((size1 (cell-count (car rtree))))
        ;;(print (list :dbgb rnth size1 (car rtree)))
        (cond
         ((>= 0 size1) (tree-nth-cell ;; No car subtree.
                        rnth
                        (cdr rtree))) ;; Find elt in the cdr subtree.
         ((<= rnth size1) (tree-nth-cell ;;  Elt is in car subtree.
                           rnth
                           (car rtree))) ;; Find elt in the car subtree.
         (t (tree-nth-cell ;; Elt is in cdr subtree.
             (- rnth size1) ;; Account for skipping car subtree.
             (cdr rtree)))))))) ;; Skip car subtree.
;; Tests
;; (tree-nth-cell 1 nil)
;; nil
;; (tree-nth-cell 1 '(a b c))
;; (a b c)
;; (tree-nth-cell 2 '(a b c))
;; (b c)
;; (tree-nth-cell 3 '(a b c))
;; (c)
;; (tree-nth-cell 1 '((a b) c))
;; ((a b) c)
;; (tree-nth-cell 2 '((a b) c))
;; (a b)
;; (tree-nth-cell 3 '((a b) c))
;; (b)
;; (tree-nth-cell 4 '((a b) c))
;; (c)
;; (tree-nth-cell 5 '((a b) c))
;; nil
;; (tree-nth-cell 2 '((a f) ((b)) c))
;; (a f)
;; (tree-nth-cell 3 '((a f) ((b)) c))
;; (f)
;; (tree-nth-cell 4 '((a f) ((b)) c))
;; (((b)) c)
;; (tree-nth-cell 5 '((a f) ((b)) c))
;; ((b))
;; (tree-nth-cell 6 '((a f) ((b)) c))
;; (b)
;; (tree-nth-cell 7 '((a f) ((b)) c))
;; (c)
;; (tree-nth-cell 8 '((a f) ((b)) c))
;; nil


;; -------------------------------------------------- random-tree-cell ----
(defun random-tree-cell (rtree)
  "Return random cell in the tree, but not the whole tree."
  (let* ((size (cell-count rtree))
         (rx (1+ (random (1- size)))) ;; Avoid 1st cell (the whole tree).
         (nth (1+ rx)) ;; Incr cuz our fcn is 1-based, not 0-based.
         (spot (tree-nth-cell nth rtree)))
    ;; (print (list :dbg size nth spot))
    spot))
;; Tests
;; (random-tree-cell '(+ (* 5 a b) (* c (- d 6))))
;; ((* c (- d 6))) ;; No op
;; (- d 6) ;; Has op
;; ((* 5 a b) (* c (- d 6))) ;; No op
;; (* 5 a b) ;; Has op
;; (+ (* 5 a b) (* c (- d 6))) ;; Has op
;; (* c (- d 6)) ;; Has op

;; ---------------------------------------------------------- make-kid ----
(defun make-kid (rmom rtgt rnew)
  "Return kid: copy of mom with tgt cell replaced by given new cell, or nil."
  (if (not (and rmom rtgt rnew
                (listp rmom)
                (listp rtgt)
                (listp rnew)))
      rmom
    (if (eq rmom rtgt)
        rnew
      (cons (make-kid (car rmom) rtgt rnew)
            (make-kid (cdr rmom) rtgt rnew)))))

;; ----------------------------------------------------- test-make-kid ----
(defun test-make-kid (rtree)
  "Test make-kid with random tgt cell and fixed replacement list."
  (let* ((tgt (random-tree-cell rtree))
         (newop '(/ 2 3)) ;; New has op.
         (newnop '(8 9)) ;; New has no op.
         (ops '(+ - * /)))
    (print (list :dbg :tgt tgt))
    (make-kid rtree
              tgt
              (if (member (car tgt) ops) ;; Tgt also has an op?
                  newop
                newnop))))
;; Tests
;; (cell-count '(+ (* 5 a b) (* c (- d 6))))
;; 13
;; (test-make-kid '(+ (* 5 a b) (* c (- d 6))))
;; (:dbg :tgt (c (- d 6)))
;; (+ (* 5 a b) (* 8 9))
;; (:dbg :tgt ((- d 6)))
;; (+ (* 5 a b) (* c 8 9))
;; (:dbg :tgt ((- d 6)))
;; (+ (* 5 a b) (* c 8 9))
;; (:dbg :tgt (d 6))
;; (+ (* 5 a b) (* c (- 8 9)))
;; (:dbg :tgt (5 a b))
;; (+ (* 8 9) (* c (- d 6)))
;; (:dbg :tgt (a b))
;; (+ (* 5 8 9) (* c (- d 6)))
;; (:dbg :tgt ((* 5 a b) (* c (- d 6))))
;; (+ 8 9)
;; (:dbg :tgt ((- d 6)))
;; (+ (* 5 a b) (* c 8 9))
;; (:dbg :tgt (6))
;; (+ (* 5 a b) (* c (- d 8 9)))
```

CS 481 — AI —Project 2 GP Arithmetic Ideas

```
;; Illustrations of how some GP Framework things could be done.


;; ----------------------------------------------------- get-front-upto-nth ----
(defun get-front-upto-nth ( rn rlist )
  "Return list head from 0-th thru N-th elt.  Assumes elt-n is unique."
  (let ((elt-n (nth rn rlist)))
    (reverse (member elt-n (reverse rlist)))))
;; Tests
;; (setq my-list '((1 a) (2 b) (3 c) (4 d) (5 e) (6 f) (7 g)))
;; (get-front-from-nth 4 my-list)
;; ((1 a) (2 b) (3 c) (4 d) (5 e))
;; (get-front-from-nth 2 my-list)
;; ((1 a) (2 b) (3 c))

;; ------------------------------------------------------------- get-score ----
(defun get-score (rcritter)
  "Get score for critter.  Dummy fcn: just return its length."
  (length rcritter))
;; Tests
;; (get-score '(+ 3 4))
;; 3

;; ------------------------------------------------------------- score-pop ----
(defun score-pop ( rpop ) ;; Pop is a population.
  "Create Pop-Scored pairs (Score Critter) given Pop list of critters."
  (mapcar #'(lambda (critter)
              (let ((score (get-score critter)))
                (list score critter)))
          rpop))
;; Tests
;; (setq my-pop '((a b c)
;;                (a)
;;                (e f g)
;;                (a d)))
;; ((a b c) (a) (e f g) (a d))
;; (setq my-pop-scored (score-pop my-pop))
;; ((3 (a b c)) (1 (a)) (3 (e f g)) (2 (a d)))
;; -------------------------------------------------- safe-sort-scored-pop ----
(defun safe-sort-scored-pop ( rscored-pop )
  "Return a sorted list of scored-critter elts.  Don't change given list.
   NB, your Lisp's built-in sort fcn may damage the incoming list."
  (let ((sacrifice-list (copy-list rscored-pop)))
    (sort sacrifice-list
          #'(lambda (scored-critter-1 scored-critter-2)
               (< (car scored-critter-1) (car scored-critter-2))))))
;; Tests
;; my-pop-scored
;; ((3 (a b c)) (1 (a)) (3 (e f g)) (2 (a d)))
;; (safe-sort-scored-pop my-pop-scored)
;; ((1 (a)) (2 (a d)) (3 (a b c)) (3 (e f g)))
;; my-pop-scored
;; ((3 (a b c)) (1 (a)) (3 (e f g)) (2 (a d)))

;; --------------------------------------------------- get-pop-from-scored ----
(defun get-pop-from-scored (rscored-pop)
  "Return just the Pop of critters from the Scored Pop."
  ;;Alt: (mapcar #'(lambda (elt) (nth 1 elt)) rscored-pop)
  (mapcar #'cadr rscored-pop))
;; Tests
;; my-pop-scored
;; ((3 (a b c)) (1 (a)) (3 (e f g)) (2 (a d)))
;; (get-pop-from-scored my-pop-scored)
;; ((a b c) (a) (e f g) (a d))
```