CPSC 481 — AI —Project 2b GP Arithmetic Support 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."

;; ------------------------------------------------------------ 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))
```

CPSC 481 — AI —Project 2b GP Arithmetic Support Fcns

```
;; (: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)))
```