```
#lang racket
;Darius Hooks
;Nov 15, 2016
;Project #2
;
;
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;DEGREE OF POLYNOMIAL;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define degree
  (lambda (ply)
    (if(null? ply)
       0
       (- (length ply) 1)
     )
   )
 )
;
;
;
;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;DISPLAY TERM;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;
(define displayTerm
  (lambda (coef exp)
    (cond
       ;IF COEFFICIENT IS 1
       [(= coef 1) (begin (if (= exp 0) (begin(display coef)(newline)) (if (= exp 1) (begin(display "x")(newline)) (begin (display "x^")(display exp)))))]

       ;IF COEFFICIENT IS GREATER THAN 1
       [(> coef 1) (begin (display coef) (if (= exp 0) (newline) (if (= exp 1) (begin(display "x")(newline)) (begin (display "x^")(display exp)))))]

       ;IF COEFFICIENT IS 0
       [(= coef 0) (display "Not a polynomial")]
     );END COND

   )
 );END DISPLAY TERM
;
;
;
;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;DISPLAY POLYNOMIAL;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;
(define displayPoly
  (lambda (p)
    (cond
           );END BEGIN
         );END

        ;IF FRONT OF LIST NOT 0
        ((not (equal? (car p) 0))
          (begin
            (cond
              ((equal? (- (length p) 1) 1) (begin (if (equal? (car p) 1) (display "x")(begin(display (car p)) (display "x"))) (if (equal? (cadr p) 0) (displayPoly (cdr p)) (begin (display "  +  ") (displayPoly (cdr p))))))
              ((equal? (- (length p) 1) 0) (begin (display (car p)) (displayPoly (cdr p))))
              (else (if (equal? (car p) 1) (display "x^") (begin(display (car p)) (display "x^"))) (display (- (length p) 1)) (if (equal? (cadr p) 0) (displayPoly (cdr p)) (begin (display "  +  ") (displayPoly (cdr p)))))
             );END COND
           );END BEGIN
         );END

     );END COND
   )
 );END DISPLAY POLY
;
;
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;isPOLYNOMIAL;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define polynomial?
  (lambda (z)
    (let ([num 0])
      (counter z num)
     )
   )
 )
;HELPER FUNCTION FOR POLYNOMIAL
(define counter
  (lambda (lst i)
    (cond
         ((null? lst) #f)
         ((string? lst) #f)
         ((string? (car lst)) #f)
         ((list? (car lst)) #f)
         ((equal? (list-ref lst 0) 0) (if (= (length lst) 1) #t (if (equal? i 0) #f (counter (cdr lst)(+ i 1)))))
         ((= (length lst) 1) #t)
         (else (counter (cdr lst)(+ i 1)))
       )
   )
 )
```

```
    (let ((coef (car p)) (x v) (expn (- (length p) 1)))
     (+ (* coef (expt x expn)) (evalPoly (cdr p) v))
    )
   );END ELSE
  );END COND
 )
);END EVALUATE POLYNOMIAL
;
;
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;MULTIPLY POLYNOMIAL BY CONSTANT;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define multiplyPolyByConstant
 (lambda (p a)
  (cond
   ((equal? a 0) '(0))
   (else(map (lambda (x) (* x a)) p))
  )
 )
)
;
;
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;MULTIPLY POLYNOMIAL BY X;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define multiplyPolyByX
 (lambda (p)
  (foldl cons '(0) (reverse p))
 )
)
;
;
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;ADD POLYNOMIAL;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define addPoly
 (lambda (p1 p2)
  (cond
   ;LENGTH P1 > LENGTH P2
   ((> (length p1) (length p2))
    (begin
     (let [(p3 (append (make 0 (- (length p1) (length p2))) p2))]
      (let ([p (map (lambda (x y) (+ x y)) p1 p3)])
       (cond
        ((equal? (foldl + 0 p) 0) '(0))
        ((equal? (car p) 0) (begin (remove 0 p) p))
        (else p)
```

```
                ((equal? (car p) 0) (begin (remove 0 p) p))
                (else p)
               )
             )
           )
         )
       );END 2ND CONDITION

      ;LENGTH P1 = LENGTH P2
      ((= (length p1) (length p2))
       (let ([p (map (lambda (x y) (+ x y)) p1 p2)])
         (cond
           ((equal? (foldl + 0 p) 0) '(0))
           ((equal? (car p) 0) (begin (remove 0 p) p))
           (else p)
          )
        )
       )
      ;END 3RD CONDITION
     )
   )
 )
;
;
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;SUBTRACT POLYNOMIAL;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define subtractPoly
  (lambda (p1 p2)
    (cond
      ;LENGTH P1 > LENGTH P2
      ((> (length p1) (length p2))
        (begin
          (let [(p3 (append (make 0 (- (length p1) (length p2))) p2))]
            (let ([p (map (lambda (x y) (- x y)) p3 p1)])
              (cond
                ((equal? (foldl + 0 p) 0) '(0))
                ((equal? (car p) 0) (begin (remove 0 p) p))
                (else p)
               )
             )
           )
         )
       );END 1ST CONDITION

      ;LENGTH P2 > LENGTH P1
      ((< (length p1) (length p2))
        (begin
          (let [(p3 (append (make 0 (- (length p2) (length p1))) p1))]
       (let ([p (map (lambda (x y) (- x y)) p1 p2)])
         (cond
           ((equal? (foldl + 0 p) 0) '(0))
           ((equal? (car p) 0) (begin (remove 0 p) p))
           (else p)
          )
        )
       )
      ;END 3RD CONDITION
     )
   )
 )
;
;
;
;
;HELPER FUNCTION FOR ADD AND SUBTRACT
(define make
  (lambda (n i)
    (if (= i 0)
        '()
        (cons n (make n (- i 1)))
     )
   )
 )
;
;
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;MULTIPLY POLYNOMIAL;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define multiplyPoly
  (lambda (p1 p2)
    (let ([x "INCOMPLETE FUNCTION"])
      (display x)
      '()
     )
   )
 )
;
;
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;TEST CASES;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define testcases
  (lambda ()
    (let ( [t1 '( 2  6  0  4  3  2)]
           [t2 '(6   -3   0   -3   0 )]
           [t3 '( 1   -3   0   0   0   4   -12)] )

      (displayln (evalPoly t1  -2))
      (displayln ( evalPoly t2 1.50))
      (displayln (evalPoly t3 3))

      (displayln  (polynomial? t3))
      (displayln( polynomial?  "polynomial"))
      (displayln ( polynomial? '( 0 5 6 7) ) )
      (displayln (polynomial?  (subtractPoly t1  t1)))
 )))
```