Project 1: Part 1

Project 1 will be to calculate orthogonal polynomials. It will have several parts.

Note: The scheme code in this writeup is available in the file `project1.scm`, available from the course web page.

Warmup: Solving quadratic equations

The quadratic formula says that the solutions of $ax^2 + bx + c = 0$ are given by

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

If $b^2 - 4ac < 0$ then $\sqrt{b^2 - 4ac}$ is imaginary, so there is no problem with round-off error.

If $b^2 - 4ac > 0$ then cancellation can occur in $-b + \sqrt{b^2 - 4ac}$ if $b > 0$ and in $-b - \sqrt{b^2 - 4ac}$ if $b < 0$. Thus, if $b > 0$ one would want to use the computations

$$x_1 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}, \quad x_2 = \frac{2c}{-b - \sqrt{b^2 - 4ac}}.$$

Similarly if $b < 0$.

Write a function `(quadratic-solver a b c)` that returns a list of the two roots of $ax^2 + bx + c = 0$ as accurately as possible. Test your code on the following problems:

```
euler-126% gsi
Gambit v4.1.2
> (sqrt -1)
+i
> (sqrt +i)
.7071067811865476+.7071067811865475i
> (load "quadratic-solver")
"/export/users/lucier/programs/615project/2007/project-1/quadratic-solver.scm"
> (quadratic-solver 1 2 5)
(-1+2i -1-2i)
> (quadratic-solver 1 -1 1)
(1/2+.8660254037844386i 1/2-.8660254037844386i)
> (quadratic-solver 1 2 -1)
(-2.414213562373095 .4142135623730951)
> (quadratic-solver 4 1 1)
(-1/8+.4841229182759271i -1/8-.4841229182759271i)
> (quadratic-solver 4 4 1)
(-1/2 -1/2)
> (quadratic-solver 4 0 1)
(+1/2i -1/2i)
> (quadratic-solver 0 0 1)
*** ERROR IN (console)@11.1 -- not a quadratic:  0 0 1
1>
> (quadratic-solver 4 0 -1)
(-1/2 1/2)
> (quadratic-solver 1 3138428376721 1)
(-3.138428376721e12 -3.186308177103568e-13)
>
*** EOF again to exit
```

Meroon

Standard Scheme (so-called R5RS Scheme, which Gambit implements) does not have an object system. We use an object system provided by the software package Meroon.

To use Gambit, you need to have `/pkgs/Gambit-C/current/bin/` in your path. The Gambit interpreter is called `gsi` and the Gambit compiler is called `gsc`.

To have Gambit load Meroon automatically, just call `gsi++` or `gsc++`.

Our system has two differences with standard Meroon:

(1) In standard Meroon, keywords begin with a colon; in our Meroon keywords end with a colon:

```
(define-class Polynomial Object
  ((= variable immutable:)
   (= terms    immutable:)))
```

(2) In standard Meroon, so-called setters begin with `set-` and end with `!`. In our Meroon, setters end with `-set!`:

```
euler-130% gsi++
[ Meroon V3 Paques2001+1 $Revision: 1.5 $ ]
Gambit v4.1.2
> (define-class Point Object (x y))
Point
> (define p (make-Point 0 1))
> (unveil p)
(a Point <------------- [Id: 1]
 x: 0
 y: 1 end Point)
#t
> (Point-x-set! p 1)
#<meroon #2>
> (unveil p)
(a Point <------------- [Id: 1]
 x: 1
 y: 1 end Point)
#t
>
```

Numerical Integration

This first part will be about numerical integration (quadrature rules).

The Gauss-Lobatto quadrature rules with n points have the form

$$\int_{-1}^{1} f(x)\,dx \approx \sum_{\nu=0}^{n-1} \gamma_{n\nu} f(x_{n\nu}).$$

Here $x_{n0} = -1$, $x_{n,n-1} = 1$, and $x_{n,\nu}$, $\nu = 1, \ldots, n-2$, are the zeros of the degree $n-2$ orthogonal polynomial over $[-1, 1]$ with the weight

$$w(x) = 1 - x^2.$$

If we define

$$\ell_{n\kappa}(x) = \prod_{\substack{\nu=0 \\ \nu \neq \kappa}}^{n-1} \frac{x - x_{n\nu}}{x_{n\kappa} - x_{n\nu}}$$

then $\ell_{n\kappa}$ has degree $n-1$ and satisfies

$$\ell_{n\kappa}(x_{n\nu}) = \begin{cases} 1, & \nu = \kappa, \\ 0, & \nu \neq \kappa. \end{cases}$$

The weights $\gamma_{n\nu}$ satisfy

$$\gamma_{n\nu} = \int_{-1}^{1} \ell_{n,\nu}(x)\,dx.$$

So, the first part of the project is to write code to manipulate polynomials. We'll keep the code at
`http://mitpress.mit.edu/sicp/full-text/sicp/book/node49.html`
in the back of our heads as a model, while writing new code that uses Meroon's framework of classes/objects and generics/methods. We'll refer to this web page as "the SICP web page."

A good thing to keep in the back of your mind is that, generally speaking, code should not depend on the implementation technique of terms or termlists. The code in SICP uses lists for both terms and termlists of polynomials, but that can be confusing, especially for debugging.

So we'll define a polynomial class:

```
(define-class Polynomial Object
  ((= variable immutable:)        ; a symbol
   (= terms    immutable:)))      ; a list of terms
```

and a way to check whether two Polynomial variables are the same:

```
(define (Polynomial-variable= var1 var2)
  (eq? var1 var2))
```

The terms of a polynomial is just a list of nonzero terms, in decreasing order by degree (unfortunately called "order" at the SICP web page), so we need some code to manipulate terms and lists of terms:

```
;;; a term is a pair (order coeff) (order should really be degree, but ...)
;;; We're going to use a Meroon class for terms to aid debugging.
(define-class term Object
  ((= order)
   (= coeff)))
```

The generic function `initialize!` is called on all Meroon objects after their construction, and for polynomials we use `initialize!` to check that the invariants we assume about polynomials are satisfied whenever we create one:

```
(define-method (initialize! (p Polynomial))
  (let ((terms (Polynomial-terms p)))
    ;; check that
    ;; (a) termlists are in decreasing order
    (let loop ((terms terms))
      (cond ((or (empty-termlist? terms)
                 (empty-termlist? (rest-terms terms)))
             ;; We're done checking the orders of terms
             )
            ((> (term-order (first-term terms))
                (term-order (first-term (rest-terms terms))))
             (loop (rest-terms terms)))
            (else
             (error "initialize! for Polynomials: terms are not in decreasing order" p))))
    ;; (b) check that coeffs are not zero
    (let loop ((terms terms))
      (cond ((empty-termlist? terms)
             ;; We're done checking that terms are nonzero
             )
            ((not (=zero? (term-coeff (first-term terms))))
             (loop (rest-terms terms)))
            (else
             (error "initialize! for Polynomials: some terms have zero coefficients" p))))
    (call-next-method)))
```

You are asked to define the generic function `=zero?` in Problem 1.

It is not good form to use `initialize!` to enforce the invariants by, e.g., sorting the terms in decreasing order or removing all terms with zero coefficients. Having `initialize!` check these invariants, and ensuring all code that manipulates Polynomials preserves these invariants helps in debugging.

We will need some operations on lists. The functions `map` and `for-each` are built into Scheme (learn them!) but we will need some more:

```
;;; See the Haskell Wiki page
;;; http://www.haskell.org/haskellwiki/Fold
;;; for a good explanation, together with pictures, for how
;;; fold-left and fold-right work.
(define (fold-left operator initial-value list)
  (if (null? list)
      initial-value
      (fold-left operator
                 (operator initial-value (car list))
                 (cdr list))))
(define (fold-right operator initial-value list)
  (if (null? list)
      initial-value
      (operator (car list)
                (fold-right operator initial-value (cdr list)))))
;;; map is a builtin function that works on lists, but it could
;;; be defined as follows:
(define (my-map f list)
  (fold-right (lambda (v l)
                (cons (f v) l))
              '()
              list))
```

And one can use the usual functions `car`, `cdr`, `cadr`, `null?` and the usual empty list `'()`, but we need a better way to add a term to a list of terms, since we don't want any zero terms in our termlists:

```
(define (adjoin-term term term-list)
  (if (=zero? (term-coeff term))
      term-list
      (cons term term-list)))
(define (map-termlist f list)
  (fold-right (lambda (v l)
                (adjoin-term (f v) l))
              '()
              list)
```

In `map-termlist`, the function `f` must return a `term` and we let the `initialize!` method for `Polynomials` check that the order of the terms is decreasing.

The SICP web page has code for adding two polynomials. Putting it into our terms we define a generic function `add` that should work for everything, and we start with it working with numbers:

```
(define-generic (add (x) y)
  (if (number? x)
      (if (number? y)
          ;; We know how to add two numbers.
          (+ x y)
          ;; We don't know how to add a number to an arbitrary object,
          ;; but perhaps there is a method to add the object y to
          ;; a number, so we swap arguments and try again.
          (add y x))
      ;; If x isn't a number, we don't have a method for it.
```

```
      (error "add: This generic is not defined on these objects: " x y)))
```

and then we define a method for adding Polynomials:

```
(define-method (add (p_1 Polynomial) p_2)
  (cond ((number? p_2)
         (add p_1 (number->Polynomial p_2 (Polynomial-variable p_1))))
        ((and (Polynomial? p_2)
              (Polynomial-variable= (Polynomial-variable p_1)
                                    (Polynomial-variable p_2)))
         (instantiate Polynomial
                      variable: (Polynomial-variable p_1)
                      terms:    (add-terms (Polynomial-terms p_1)
                                           (Polynomial-terms p_2))))
        (else
         (error "add: p_2 is neither a number nor a polynomial with the same variable as
p_1 " p_1 p_2))))
```

This method is called only when `p_1` is a polynomial; if `p_2` is a number, it converts `p_2` to a Polynomial with the same variable as `p_1` and calls `add` again with both arguments now a Polynomial. So you need to define `number->Polynomial`, which takes two arguments.

The SICP web page has code for `add-terms`:

```
(define (add-terms l1 l2)
  (cond ((null? l1) l2)
        ((null? l2) l1)
        (else
         (let ((t1 (car l1))
               (t2 (car l2)))
           (cond ((> (term-order t1)
                     (term-order t2))
                  (adjoin-term t1
                               (add-terms (cdr l1) l2)))
                 ((< (term-order t1)
                     (term-order t2))
                  (adjoin-term t2
                               (add-terms l1 (cdr l2))))
                 (else
                  (adjoin-term
                   (make-term (term-order t1)
                              (add (term-coeff t1)
                                   (term-coeff t2)))
                   (add-terms (cdr l1)
                              (cdr l2)))))))))
```

You need to define a `multiply` generic that works with numbers by default, and a method for `multiply` that works on Polynomials; follow ths same pattern as for `add`. The SICP web page has the guts of the code:

```
(define (multiply-terms l1 l2)
  (if (null? l1)
      l1
      (add-terms (multiply-term-by-all-terms (car l1) l2)
                 (multiply-terms (cdr l1) l2))))
(define (multiply-term-by-all-terms t1 L)
  (if (null? L)
      L
      (let ((t2 (car L)))
        (adjoin-term
```

```
          (make-term (+ (term-order t1)
                        (term-order t2))
                     (multiply (term-coeff t1)
                               (term-coeff t2)))
          (multiply-term-by-all-terms t1 (cdr L))))))
```

So that's pretty much the code that comes on the web page. Meroon defines a generic function `show` that we can specialize for Polynomials as such:

```
(define-method (show (p Polynomial) . stream)
  (let ((port (if (null? stream)
                  (current-output-port)
                  (car stream))))
    (if (=zero? p)
        (display 0)
        (show-terms (Polynomial-variable p)
                    (Polynomial-terms p)
                    port))
    (newline port)))
(define (show-terms variable terms port)
  (show-first-term variable (car terms) port)
  (for-each (lambda (term)
              (show-term variable term port))
            (cdr terms)))
(define (show-first-term variable term port)
  (let ((coeff (term-coeff term))
        (order (term-order term)))
    (print port: port
           (list (if (and (= coeff 1)
                          (positive? order))
                     '()
                     coeff)
                 (cond ((zero? order) '())
                       ((= order 1) variable)
                       (else
                        (list variable "^" order)))))))
(define (show-term variable term port)
  (let ((coeff (term-coeff term))
        (order (term-order term)))
    (print port: port
           (list (if (negative? coeff)
                     "-"
                     "+")
                 (let ((abs-coeff (abs coeff)))
                   (if (and (eq? coeff 1)
                            (< 0 order))
                       '()
                       (abs coeff)))
                 (cond ((zero? order) '())
                       ((= order 1) variable)
                       (else
                        (list variable "^" order)))))))
```

It will probably help your debugging.

So, here are some problems.

(1) The above code uses a function `=zero?`. Define a generic function `=zero?` that handles numbers. Define a method that works with Polynomials. Why does the `initialize!` method for polynomials use `=zero?` to check whether coefficients are zero rather than the built-in Scheme function `zero?`.

(2) Define a generic function `(negate (x))` that handles numbers by default. Define a method for `negate` that works with Polynomials. Use the generic `negate` to define a regular function `(subtract x y)`. (Remember that a polynomial in `x` may have coefficients that are polynomials in `y`, so write `negate` so it works with these types of polynomials.)

(3) Define a function `(exponentiate x n)` that uses `multiply` to exponentiate anything that `multiply` can multiply. Use the discussion of exponentiation on page

`http://mitpress.mit.edu/sicp/full-text/book/book-Z-H-11.html#%_sec_1.2`

as your model. You should be able to exponentiate a polynomial.

(4) Define a function `(variable->Polynomial x)` that takes a symbol `x` and returns a Polynomial that represents the polynomial x, i.e., a single term with coefficient 1 and order 1. Define a function `(number->Polynomial n x)` where `n` is a number and `x` is a symbol. (Hint: Use `make-term`, `adjoin-term`, and `the-empty-termlist` to calculate the termlist in `number->Polynomial`.)

(5) Define a generic function `(evaluate f x)` that evaluates the function `f` at `x`. If `f` is a number, assume that it means a function that constantly returns `f` (so the generic is supposed to work with both numbers `v` and Scheme functions `f`). Define a method for Polynomials. If `p` is a polynomial, then you should be able to say `(evaluate p p)`, i.e., evaluate a polynomial with another polynomial as an argument.

If you've done the exercises until now, something like the following should work.

```
;;;  evaluation
(define-generic (evaluate (f) x)
  (cond ((number? f) f)
        ((procedure? f) (f x))
        (else (error "evaluate: unknown argument types " f x))))
(define-method (evaluate (p Polynomial) x)
  (evaluate-terms (Polynomial-terms p) x))
(define (evaluate-terms terms x)
  (if (null? terms)
      0
      (add (evaluate-term (car terms) x)
           (evaluate-terms (cdr terms) x))))
(define (evaluate-term term x)
  (multiply (exponentiate x (term-order term))
            (term-coeff term)))
```

Can you write a method that uses Horner's rule for evaluating Polynomials in our representation?

Changes make 2014/02/14

(1) Add a hint about how to write `number->Polynomial`.

Changes made 2012/02/27

(1) The project will just be about orthogonal polynomials.
(2) Corrected the formula for $\gamma_{n\nu}$.
(3) Changed the definition of `add` so it can add polynomials to numbers in either order.

I added some comments to the problems:

(1) Added `map-termlist` as an exercise.
(2) Made more explicit my expectations for `negate`, `exponentiate`, and `evaluate`.
(3) Redid the definition of the `evaluate` generic to match its specification.

Changes made 2012/03/07

(1) Make a `term` a Meroon object for easier debugging.

(2) `(Polynomial-terms p)` was a regular list, so I'm just going to use the regular list operations, except for `adjoin-term` and `map-termlist`, which check that the term is nonzero before adding it to the list. Got rid of `first-term`, `rest-terms`, `empty-termlist?`, etc., and changed their uses to regular list operations.
(3) Since `map-termlist` is already defined, don't have it as an exercise.
(4) Defined `fold-left` and `fold-right` earlier, so I could use them in `map-termlist`.

Changes made 2012/05/17

(1) Change the definition of the interpolation points to handle -1 and 1 in the same way as the other interpolation points. Fix the upper limit of the product defining $\ell_{n\kappa}$.
(2) Move the definition of `initialize!` for polynomials from the web page to here. Explain its use.
(3) Generally clean up the discussion.