MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Spring 2006

Recitation 11 — Evacuation Day (3/17/2006)
Tagged Data — Solutions

Tagging procedure:

```
(define (tagged-list? x tag)
  (and (pair? x) (eq? (car x) tag)))
```

Problems

1. Build a tagged abstraction for variables:

   ```
   (define *variable-tag* 'variable)
   ```

   (a) Write the constructor `make-variable`:

   ```
   (define (make-variable vname)
     (list *variable-tag* vname))
   ```

   (b) Write the type predicate `variable?`:

   ```
   (define (variable? x)
     (tagged-list? x *variable-tag*))
   ```

   (c) Write the selector `varname`:

   ```
   (define (varname var)
     (if (variable? var)
         (cadr var)
         (error "not a variable: " var)))
   ```

   (d) Write the equality predicate `variable=?`:

   ```
   (define (variable=? v1 v2)
     (eq? (varname v1) (varname v2)))
   ```

Tagged abstraction for constants:

```
(define *constant-tag* 'constant)

(define (make-constant c)
  (list *constant-tag* c))

(define (constant? x)
  (tagged-list? x *constant-tag*))

(define (constval c)
  (if (constant? x)
      (cadr x)
      (error "not a constant: " c)))
```

Tagged abstraction for polynomials:

```
(define *poly-tag* 'poly)

(define (make-polynomial var terms)
  (list *poly-tag* var terms))

(define (poly? x)
  (tagged-list? x *poly-tag*))

(define (poly-get-var poly)
  (if (poly? poly)
      (cadr poly)
      (error "not a polynomial:" poly)))

(define (poly-get-term i poly)
  (if (poly? poly)
      (list-ref (caddr poly) i)
      (error "not a polynomial:" poly)))

(define (poly-get-terms poly)
  (caddr poly))
```

2. Write `constant-add`:

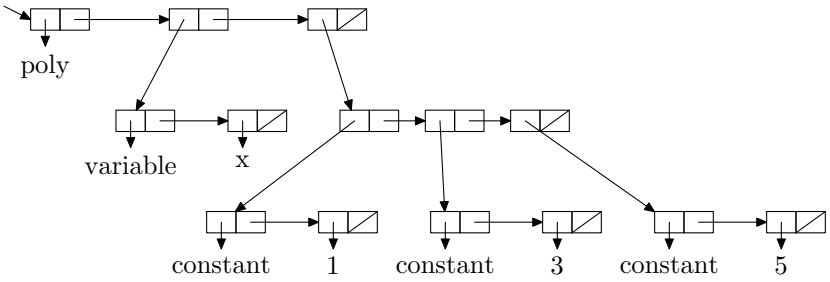
   ```
   (define (constant-add c1 c2)
     (make-constant (+ (constval c1) (constval v2))))
   ```

3. Write a basic `add`, which works only on constants and polynomials, assuming you have a procedure `poly-add` which adds two polynomials:

```
(define (add e1 e2)
  (cond ((and (constant? e1)
              (constant? e2))
         (constant-add e1 e2))
        ((and (poly? e1)
              (poly? e2))
         (poly-add e1 e2))
        (else (error "not constant or poly"))))
```

4. Draw a box-and-pointer diagram of the representation of $5x^2 + 3x + 1$.

```
(make-polynomial (make-var 'x)
                 (list (make-constant 1)
                       (make-constant 3)
                       (make-constant 5)))
```



5. Write `poly-add`, which adds two polynomials

   (a) First write `add-terms`, which takes two lists of terms and returns a new list of sum terms:

```
(define (add-terms t1 t2)
  (cond ((null? t1)
         t2)
        ((null? t2)
         t1)
        (else
         (cons (add (car t1) (car t2))
               (add-terms (cdr t1) (cdr t2))))))
```

   (b) Then write `poly-add` using `add-terms`:

```
(define (poly-add p1 p2)
   (cond ((and (poly? p1) (poly? p2))
          (if (variable=? (poly-get-var p1)
                          (poly-get-var p2))
              (make-poly
               (poly-get-var p1)
               (add-terms (poly-get-terms p1) (pol-get-terms p2)))
              (make-poly
```

```
                    (poly-get-var p1)
                    (cons (add (car (poly-get-terms p1))
                               p2)
                          (cdr (poly-get-terms p1)))))
              (else (error "not given two polys")))))
```

6. Write `var->poly`, which *promotes* a variable to a polynomial:

```
(define (var->poly var)
  (make-poly var (list 0 1)))
```

7. Write `const->poly`, which *promotes* a constant to a polynomial:

```
(define (const->poly var const)
  (make-poly var (list const)))
```

8. Write `->poly`, which converts it's input to a polynomial:

```
(define (->poly var exp)
  (cond ((constant? exp)
         (const->poly var exp))
        ((variable? exp)
         (var->poly exp))
        ((poly? exp)
         exp)
        (else
         (error "unknown exp" exp))))
```

9. Write a new version of `add` which uses promotion. Use the following procedure to guess what variable to use when promoting:

```
(define (find-var e1 e2)
  (cond ((poly? e1)
         (poly-get-var e1))
        ((poly? e2)
         (poly-get-var e2))
        ((variable? e1)
         e1)
        ((variable? e2)
         e2)
        (else
         (make-variable 'x))))

(define (add exp1 exp2)
  (if (and (constant? e1)
           (constant? e2))
      (constant-add e1 e2)
      (let ((var (find-var e1 e2)))
        (poly-add (->poly var e1)
                  (->poly var e2)))))
```