6.001 Recitation 11: Tagged Data

RI: Gerald Dalley, dalleyg@mit.edu, 16 Mar 2007
http://people.csail.mit.edu/dalleyg/6.001/SP2007/

Tagging Procedure

Today we'll be developing a new set of tagged data structures. Here's a handy little procedure that makes writing predicates for tagged data much easier:

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

Variable Abstraction

```
; Write the constructor for variables
(define (make-var vname)
```

```
; Write the type predicate var?
(define (var? x)
```

```
; Write the selector var/name
(define (var/name var)
```

```
; Write the equality predicate variable=?
(define (var=? v1 v2)
```

Constant and Polynomial Abstractions

```
; Tagged abstraction for constants:

(define *const-tag* 'const)

(define (make-const c)
  (if (number? c) (list *const-tag* c)
      (error "constants␣may␣only␣be␣made␣out␣of␣numbers:␣" c)))

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

(define (const/val c)
  (if (const? c)
      (cadr c)
      (error "not␣a␣const:␣" c)))

; Tagged abstraction for polynomials:

(define *poly-tag* 'poly)

(define (make-poly var coefs)
  (cond ((not (var? var)) (error "Not␣a␣var:␣" var))
        ((not (null? (filter (lambda (x) (not (or (poly? x) (const? x))))
                             coefs)))
         (error "coefs␣must␣be␣a␣list␣of␣consts"))
        (else
         (list *poly-tag* var coefs))))

(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-coef i poly)
  ; Returns the coefficient for x^i if x is the var for poly
  (if (poly? poly)
      (list-ref (caddr poly) i)
      (error "not␣a␣polynomial:" poly)))

(define (poly/get-coefs poly)
  (caddr poly))
```

Adding Addition

```
; Write a defensive procedure that adds two constants
(define (+const c1 c2)
```

```
; Write a basic add, which works only on constants and polynomials, assuming you have a
; procedure poly-add which adds two polynomials:
(define (add exp1 exp2)
```

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

```
; We want to write a procedure +poly to add two polynomials.  First +coeffs which takes two
; lists of coefficients and returns a new list of summed coefficients:
(define (+coefs coef1 coef2)
```

```
; Now, write +poly using +coeffs...
(define (+poly p1 p2)
```

Promotion

```
; Write var->poly, which _promotes_ a variable to a polynomial:
(define (var->poly var)
```

```
; Write const->poly, which promotes a constant to a polynomial:
(define (const->poly var const)
```

```
; Write ->poly, which converts its input to a polynomial:
(define (->poly var exp)
```

```
; 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))
        ((var? e1)   e1)
        ((var? e2)   e2)
        (else        (make-var    x ))))

; Now fill in the implementation for add:
(define (add exp1 exp2)
```

Asides

Why use symbols and not strings for tagged data?

The current implementation of DrScheme (version 360) gives us a good example of what can happen when insufficient defensive programming is used. We're used to having expressions like `(define x 3)`. In Wednesday's brainteaser, we suggested a modification of Scheme that would allow `define` to be a primitive procedure and not a special form if we called it as `(define 'x 3)`. If you type this expression into DrScheme, it turns out that it will actually be evaluated without any errors! To understand what's happening, first recognize that our expression will be desugared to `(define (quote x) 3)`. DrScheme interprets this expression as creating a new procedure called `quote` with a single parameter, `x`, which returns 3. In violation of §5.3 of the R5RS specification, this overrides the syntactic keyword `quote`. For a little more fun, take a look at the following code:

```
;(define my-lambda lambda) ; doesn't work (good): lambda is a special form
(define y      2) ; this is fine
(define lambda 3) ; we shouldn't be able to do this!
(define quote  4) ; ...or this
(define define 5) ; ...and definitely not this
(define x      6) ; => reference to undefined identifier: x
(define y      7) ; => procedure application: expected procedure, given: 5; arguments were: 2 7
(lambda (x) 8)    ; => reference to undefined identifier: x
```

Solutions

Tagging Procedure

Today we'll be developing a new set of tagged data structures. Here's a handy little procedure that makes writing predicates for tagged data much easier:

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

Variable Abstraction

In the solutions, a number of automatic assertion tests are included. Here are the definitions for the testing procedures that we used:

```
(define (assert-equal actual expected)
  (if (not (equal? actual expected))
      (error "Actual value: " actual
             " not equal? to the expected value: " expected)))

(define (assert-eq actual expected)
  (if (not (eq? actual expected))
      (error "Actual value: " actual
             " not eq? to the expected value: " expected)))

(define (assert-t actual)
  (if (not actual)
      (error "Failed assertion")))

(define (assert-f actual)
  (if actual
      (error "Failed assertion")))
```

```
; Write the constructor for variables
(define (make-var vname)
  (if (symbol? vname)
      (list *var-tag* vname)
      (error "Variable names must be symbols")))

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

```
; Write the type predicate var?
(define (var? x)
  (tagged-list? x *var-tag*))
```

```
; Write the selector var/name
(define (var/name var)
  (if (var? var)
      (second var)
      (error "not a var:" var)))
```

```
; Write the equality predicate variable=?
(define (var=? v1 v2)
  (eq? (var/name v1) (var/name v2)))

(assert-equal (make-var 'a) '(var a))
(assert-t     (var? (make-var 'a)))
(assert-f     (var? 'a))
(assert-eq    (var/name (make-var 'a)) 'a)
(assert-t     (var=? (make-var 'a) (make-var 'a)))
(assert-f     (var=? (make-var 'a) (make-var 'b)))
```

Constant and Polynomial Abstractions

```
; Tagged abstraction for constants:

(define *const-tag* 'const)

(define (make-const c)
  (if (number? c) (list *const-tag* c)
      (error "constants may only be made out of numbers: " c)))

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

(define (const/val c)
  (if (const? c)
      (cadr c)
      (error "not a const: " c)))

; Tagged abstraction for polynomials:

(define *poly-tag* 'poly)

(define (make-poly var coefs)
  (cond ((not (var? var)) (error "Not a var: " var))
        ((not (null? (filter (lambda (x) (not (or (poly? x) (const? x))))
                             coefs)))
         (error "coefs must be a list of consts"))
        (else
         (list *poly-tag* var coefs))))

(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-coef i poly)
  ; Returns the coefficient for x^i if x is the var for poly
  (if (poly? poly)
      (list-ref (caddr poly) i)
      (error "not a polynomial:" poly)))

(define (poly/get-coefs poly)
  (caddr poly))
```

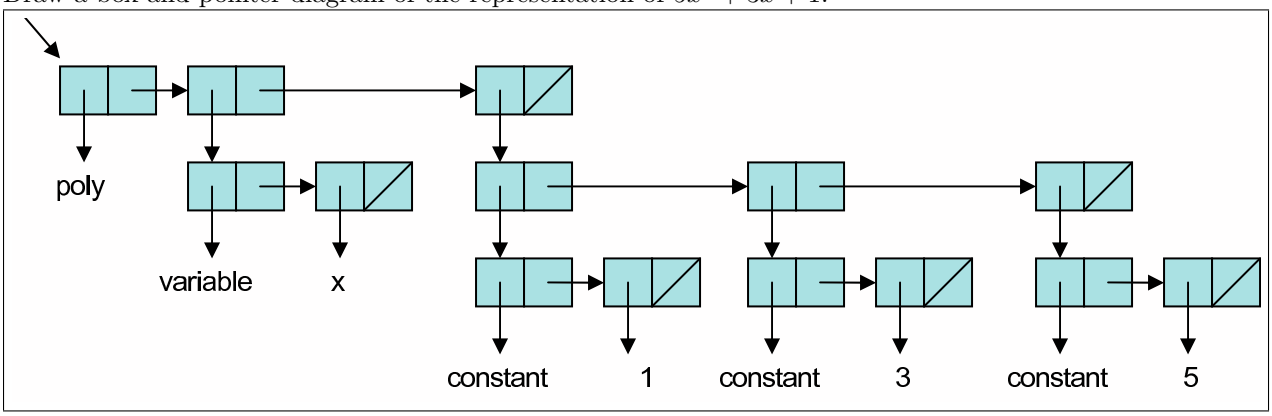
Adding Addition

```
; Write a defensive procedure that adds two constants
(define (+const c1 c2)
  (make-const (+ (const/val c1)
                 (const/val c2))))
; Why don't we need to do a const? check?
; A: because const/val will do it for us.

(assert-equal
 (const/val (+const (make-const 1) (make-const 3)))
 (+ 1 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 exp1 exp2)
  (cond ((and (const? exp1) (const? exp2))
         (+const exp1 exp2))
        ((and (poly? exp1) (poly? exp2))
         (+poly exp1 exp2))
        (else (error "can only add two consts or two polynomials"))))
```

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




```
; We want to write a procedure +poly to add two polynomials.  First +coeffs which takes two
; lists of coefficients and returns a new list of summed coefficients:
(define (+coefs coef1 coef2)
  (cond ((null? coef1) coef2)
        ((null? coef2) coef1)
        (else
         (cons (add    (car coef1) (car coef2))
               (+coefs (cdr coef1) (cdr coef2))))))

(define one   (make-const 1))
(define two   (make-const 2))
(define three (make-const 3))
(define four  (make-const 4))
(define five  (make-const 5))
(define six   (make-const 6))
(assert-equal (map const/val (+coefs (list one)           (list two)))   '(3))
(assert-equal (map const/val (+coefs (list one two three) (list four))) '(5 2 3))
```

```
; Now, write +poly using +coeffs...
(define (+poly p1 p2)
  (let ((var1  (poly/get-var   p1))
        (var2  (poly/get-var   p2))
        (coef1 (poly/get-coefs p1))
        (coef2 (poly/get-coefs p2)))
    (if (var=? var1 var2)
        (make-poly var1 (+coefs coef1 coef2))
        (make-poly var1 (cons (add (car coef1) p2)
                              (cdr coef1))))))

(define xvar           (make-var 'x))
(define yvar           (make-var 'y))
(define poly_5x^2+3x+1 (make-poly xvar (list one three five)))
(define poly_2x+1      (make-poly xvar (list one two)))
(define poly_5x^2+5x+1 (+poly poly_5x^2+3x+1 poly_2x+1))
(assert-t     (var=? xvar (poly/get-var poly_5x^2+5x+1)))
(assert-equal (map const/val (poly/get-coefs poly_5x^2+5x+1)) '(2 5 5))
```

Promotion

```
; Write var->poly, which _promotes_ a variable to a polynomial:
(define (var->poly var)
  (if (var? var)
      (make-poly var '(0 1))
      (error "Must be a var:" var)))
```

```
; Write const->poly, which promotes a constant to a polynomial:
(define (const->poly var const)
  (make-poly var (list const)))
```

```
; Write ->poly, which converts its input to a polynomial:
(define (->poly var exp)
  (cond ((const? exp)  (const->poly var exp))
        ((var?   exp)  (var->poly       exp))
        ((poly?  exp)  exp)
        (else "unrecognized expression type:" exp)))
```

```
; 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))
        ((var? e1)   e1)
        ((var? e2)   e2)
        (else        (make-var    x ))))

; Now fill in the implementation for add:
(define (add exp1 exp2)
  (if (and (const? exp1) (const? exp2))
      (+const exp1 exp2)
      (let ((var (find-var exp1 exp2)))
        (+poly (->poly var exp1)
               (->poly var exp2)))))

(assert-equal (add one poly_5x^2+3x+1)
              '(poly (var x) ((const 2) (const 3) (const 5))))

(define p1 (make-poly xvar (list one three five)))
(define p2 (make-poly yvar '((const 10) (const 30))))
(assert-equal (+poly (+poly p1 p2) p2)
              '(poly (var x)
                     ((poly (var y) ((const 21) (const 60)))
                      (const 3)
                      (const 5))))
```

Asides

Why use symbols and not strings for tagged data?

Strings would be really slow. Doing all these defensive tag checks has a small cost for symbols. The cost would be much higher for strings because `string=?` is $\Theta(n)$ and `eq?` is $\Theta(1)$.

The current implementation of DrScheme (version 360) gives us a good example of what can happen when insufficient defensive programming is used. We're used to having expressions like `(define x 3)`. In Wednesday's brainteaser, we suggested a modification of Scheme that would allow `define` to be a primitive procedure and not a special form if we called it as `(define 'x 3)`. If you type this expression into DrScheme, it turns out that it will actually be evaluated without any errors! To understand what's happening, first recognize that our expression will be desugared to `(define (quote x) 3)`. DrScheme interprets this expression as creating a new procedure called `quote` with a single parameter, `x`, which returns 3. In violation of §5.3 of the R5RS specification, this overrides the syntactic keyword `quote`. For a little more fun, take a look at the following code:

```
;(define my-lambda lambda) ; doesn't work (good): lambda is a special form
(define y      2) ; this is fine
(define lambda 3) ; we shouldn't be able to do this!
(define quote  4) ; ...or this
(define define 5) ; ...and definitely not this
(define x      6) ; => reference to undefined identifier: x
(define y      7) ; => procedure application: expected procedure, given: 5; arguments were: 2 7
(lambda (x) 8)    ; => reference to undefined identifier: x
```