MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Fall Semester, 1996

Lecture Notes, October 1 – Quote and Symbolic Data

Quote

A **symbol** is a new primitive data type, where we distinguish between the symbol and the value of the symbol. We create symbols with "quote":

```
(define x 23)
x           ==> 23

(quote x)   ==> x
   'x       ==> x
```

Symbols

What does each of the following evaluate to? Print as?

```
(define z 'y)
z


(+ x 3)


(list + x 3)


(list '+ 'x '3)


'(2 a)


'(2 (b 3))
```

Numerical Computation

```
(define (numerical-derivative f)
  (define epsilon 0.0001)
  (lambda (x)
    (/ (- (f (+ x epsilon))
          (f x))
       epsilon)))
```

Symbolic Computation

```
(define (deriv exp var)
  (cond ((constant? exp) (make-constant 0))
        ((variable? exp)
         (if (same-variable? exp var)
             (make-constant 1)
             (make-constant 0)))
        ((sum? exp)
         (make-sum (deriv (addend exp) var)
                   (deriv (augend exp) var)))
        ((product? exp)
         (make-sum
           (make-product
             (multiplier exp)
             (deriv (multiplicand exp) var))
           (make-product
             (multiplicand exp)
             (deriv (multiplier exp) var))))))
```

Math Expression Abstraction

```
(make-constant <c>)           Construct constant <c>
(constant? <e>)               Is <e> a constant?

(make-variable <v>)           Construct a variable <v>
(variable? <e>)               Is <e> a variable?
(same-variable? <v1> <v2>)    Are <v1> and <v2> same?

(make-sum <addend> <augend>)  Construct sum
(sum? <e>)                    Is <e> a sum?
(addend <e>)                  Addend of sum <e>
(augend <e>)                  Augend of sum <e>

(make-product <multiplier> <multiplicand>)
(product? <e>)                Is <e> a product?
(multiplier <e>)              Multiplier of product <e>
(multiplicand <e>)            Multiplicand of product
```

Math Expression Implementation

```
(define (make-constant x) x)
(define (constant? x) (number? x))

(define (make-variable x) x)
(define (variable? x) (symbol? x))
(define (same-variable? v1 v2)
  (and (variable? v1)
       (variable? v2)
       (eq? v1 v2)))

(define (make-sum a1 a2) (list '+ a1 a2))
(define (sum? x)
  (and (pair? x) (eq? (car x) '+)))
(define (addend s) (cadr s))
(define (augend s) (caddr s))

(define (make-product m1 m2) (list '* m1 m2))
(define (product? x)
  (and (pair? x) (eq? (car x) '*)))
(define (multiplier m) (cadr m))
(define (multiplicand m) (caddr m))
```

A Reducing Math Expression Implementation

```
(define (make-sum a1 a2)
  (cond ((and (constant? a1) (constant? a2))
         (make-constant (+ a1 a2)))
        ((constant? a1)
         (if (= a1 0) a2 (list '+ a1 a2)))
        ((constant? a2)
         (if (= a2 0) a1 (list '+ a1 a2)))
        (else (list '+ a1 a2))))

(define (make-product m1 m2)
  (cond ((and (constant? m1) (constant? m2))
         (make-constant (* m1 m2)))
        ((constant? m1)
         (cond ((= m1 0) (make-constant 0))
               ((= m1 1) m2)
               (else (list '* m1 m2))))
        ((constant? m2)
         (cond ((= m2 0) (make-constant 0))
               ((= m2 1) m1)
               (else (list '* m1 m2))))
        (else (list '* m1 m2))))
```

Adding Exponential Expressions to Deriv

```
(define (deriv exp var)
  (cond
        ...
        ((exponential? exp)
         (make-product
           (make-product (exponent exp)
                         (make-exponential
                           (base exp)
                           (- (exponent exp) 1)))
           (deriv (base exp) var)))))

(define (make-exponential b e)
  (cond ((= e 0) (make-constant 1))
        ((= e 1) b)
        (else (list '** b e))))

(define (exponential? exp)
  (and (pair? exp) (eq? (car exp) '**)))

(define (base exp) (cadr exp))
(define (exponent exp) (caddr exp))
```

More Scheme syntax: dotted tail notation

```
(define (f x . y)
  <body>)

(f 1 2 3 4)
  in <body>   x bound to 1
              y bound to (2 3 4)
```

Math Expression Implementation - Variable # Terms

```
(define (make-sum a1 . a2)
  (cons '+ (cons a1 a2)))

(define (augend s)
  (if (null? (cdddr s))
      (caddr s)
      (cons '+ (cddr s))))

(define (multiplicand p)
  (if (null? (cdddr p))
      (caddr p)
      (cons '* (cddr p))))
```