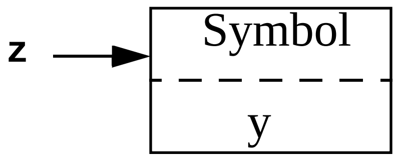
Using Symbols in Expressions (1)

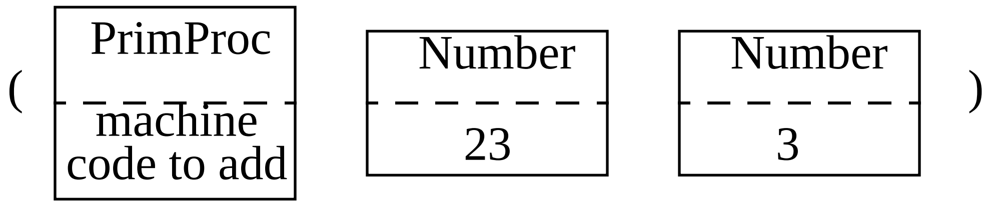
```
(define z 'y)
```



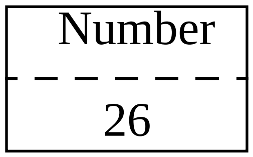
`z ==> y`
prints as

```
(+ x 3)
```

evaluate sub-expressions...



apply...

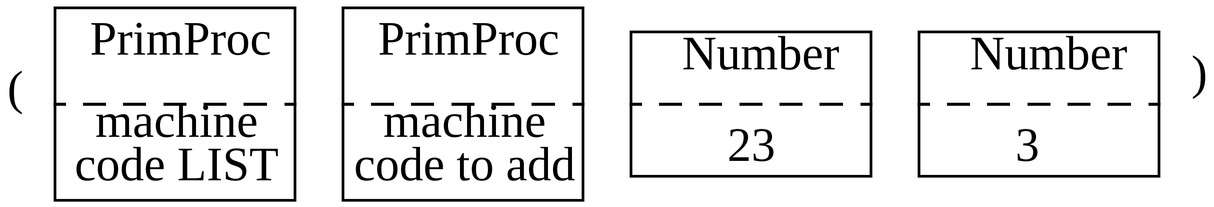


`==> 26`
prints as

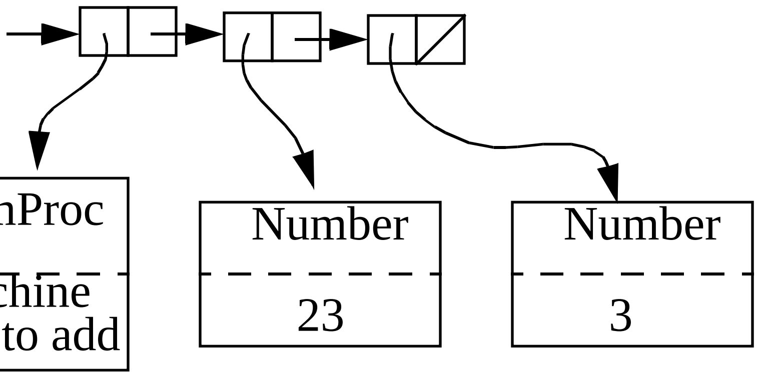
Using Symbols in Expressions (2)

```
(list + x 3)
```

evaluate sub-expressions...



apply...



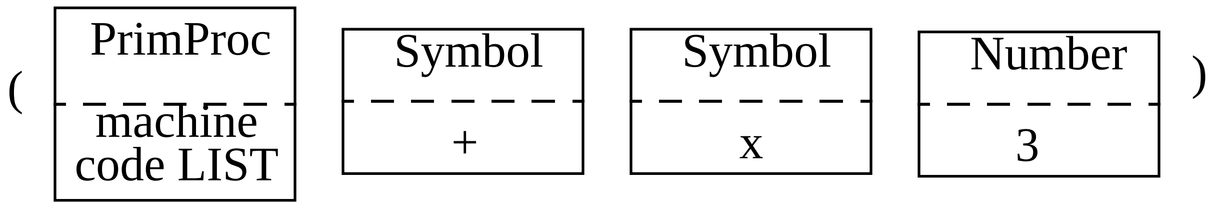
```
==>   ([#prim-proc 7] 23 3)
```

prints as

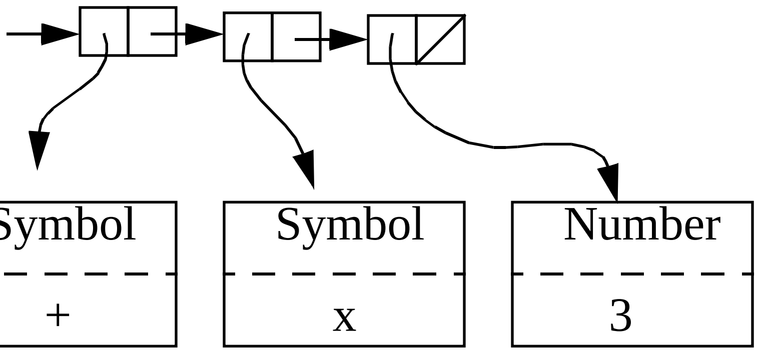
Using Symbols in Expressions (3)

```
(list '+ 'x '3)
```

evaluate sub-expressions...



apply...

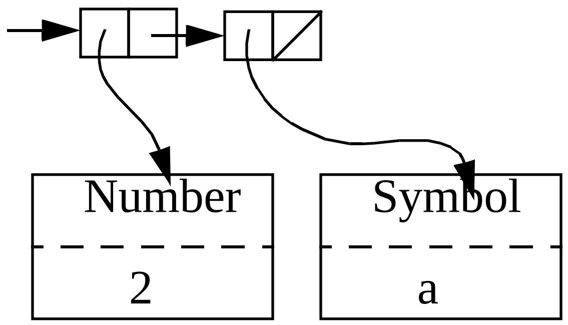


```
==>   (+ x 3)
```

prints as

Using Symbols in Expressions (4)

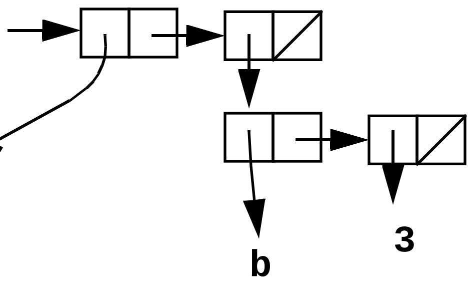
```
'(2 a)
```



```
==>    (2 a)
```

prints as

```
'(2 (b 3))
```



```
==>    (2 (b 3))
```

prints as

Example: Substituting Symbols

```
(define (substitute new old lst)
  (if (null? lst)
      ’()
      (let ((rest (substitute new old (cdr lst))))
        (if (eq? old (car lst))
            (cons new rest)
            (cons (car lst) rest)))))


(substitute ’shemp ’curley
            ’(moe pounded curley on the head))
==> (moe pounded shemp on the head)
```

Numerical Computation

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

Symbolic Computation

$$\frac{dc}{dx} = 0 \qquad \frac{dx}{dx} = 1 \qquad \frac{dy}{dx} = 0$$

$$\frac{d}{dx}(u+v) = \frac{du}{dx} + \frac{dv}{dx}$$

$$\frac{d}{dx}(uv) = u\left(\frac{dv}{dx}\right) + v\left(\frac{du}{dx}\right)$$

```
(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

Constructors, Selectors, and Predicates:

`(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))
```

Math Expressions

4x+2

```
(define math
  (make-sum
    (make-product (make-constant 4)
                  (make-variable ’x))
    (make-constant 2)))

==> (+ (* 4 x) 2)
```

Symbolic Derivative - Spaghetti Code

```
(define (deriv exp var)
  (cond ((number? exp) 0)
        ((symbol? exp)
         (if (and (symbol? var) (eq? exp var))
             1
             0))
        ((and (pair? exp) (eq? (car exp) ’+))
         (list ’+
               (deriv (cadr exp) var)
               (deriv (caddr exp) var)))
        ((and (pair? exp) (eq? (car exp) ’*))
         (list ’+
               (list ’* (cadr exp)
                        (deriv (caddr exp) var))
               (list ’* (deriv (caddr exp) var)
                        (caddr exp))))))
```

Math Expression Implementation (Alternative)

```
(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 'SUM a1 a2))
(define (sum? x)
  (and (pair? x) (eq? (car x) 'SUM)))
(define (addend s) (cadr s))
(define (augend s) (caddr s))

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

Math Expressions (Alternative)

4x+2

```
(define math
  (make-sum
    (make-product (make-constant 4)
                  (make-variable ’x))
    (make-constant 2)))

==> (SUM (PROD 4 x) 2)
```

Deriv Example

```
math
==> (+ (* 4 x) 2)
```

Follow substitution model through:

```
(deriv math ’x)

(make-sum (deriv ’(* 4 x) ’x)
          (deriv 2 ’x))

(make-sum (deriv ’(* 4 x) ’x)
          0)

(make-sum (make-sum (make-product 4 (deriv ’x ’x))
                    (make-product (deriv 4 ’x) ’x))
          0)

(+ (+ (* 4 1)
      (* 0 x))
   0)
```

Deriv - Reduction Problem

```
(deriv ’(+ x 3) ’x)
==> (+ 1 0)

(deriv ’(* x y) ’x)
==> (+ (* x 0) (* 1 y))

(derive ’(* (* x y) (+ x 3)) ’x)
==> (+ (* (* x y) (+ 1 0))
       (* (+ (* x 0) (* 1 y))
          (+ x 3)))
```

(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))))

 (list ‘* m1 m2))
```

Variable # Arguments in Procedures

In Scheme:

```
(+ (* x 3) 10 (+ 2 x))
```

Would like to be able to build similar products and sums with arbitrary number of arguments.

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))))
```

Adding Exponential Expressions to Deriv

$$\frac{du^n}{dx} = nu^{n-1}\frac{du}{dx}$$

```
(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))
```