**Symbols?**
- Say your favorite color
- Say "your favorite color"
- What is the difference?

**Creating and Referencing Symbols**
- How do I create a symbol?
  (define alpha 27)
- How do I reference a symbol's value?
  alpha
  ;Value: 27
- How do I reference the symbol itself?
  e.g.: How can I build this list: (27 alpha)
  (list alpha ??? )
  (27 alpha )

**Review: data abstraction**
- A data abstraction consists of:
  - constructors
    (define make-point
      (lambda (x y) (list x y)))
  - selectors
    (define x-coor
      (lambda (pt) (car pt)))
  - operations
    (define on-y-axis?
      (lambda (pt) (= (x-coor pt) 0)))
  - contract
    (x-coor (make-point <x> <y>)) = <x>

**Symbols?**
- Say your favorite color
- Say "your favorite color"
- What is the difference?
  - In one case, we want the meaning associated with the expression
  - In the other case, we want the actual words (or symbols) of the expression

**Quote**
- Need a way of telling interpreter: "I want the following object as a data structure, not as an expression to be evaluated"

(quote alpha)
;Value: alpha
Symbol: a primitive type
- constructors:
  None since really a primitive not an object with parts
- selectors
  None
- operations:
  symbol? ; type: anytype -> boolean
  (symbol? (quote alpha)) ==> #t
  eq? ; discuss in a minute

Symbol: printed representation

Symbol: printed representation

Symbols are ordinary values
(list 1 2)             ==> (1 2)
(list (quote delta) (quote gamma))
  ==> (delta gamma)

A useful property of the quote special form
(list (quote delta) (quote delta))
  ==> (delta delta)

The operation eq? tests for the same object
- a primitive procedure
- returns #t if its two arguments are the same object
- very fast

(eq? (quote eps) (quote eps)) ==> #t
(eq? (quote delta) (quote eps)) ==> #f

For those who are interested:
; eq?: EQtype, EQtype ==> boolean
; EQtype = any type except number or string

One should therefore use = for equality of numbers, not eq?

Generalization: quoting other expressions

Expression: Reader converts to: Prints out as:
1. (quote a)                 a
2. (quote (a b))             (a b)
3. (quote 1)
Shorthand: the single quote mark

'\(a\) is shorthand for \((\text{quote } a)\)

'(1 2) is shorthand for \((\text{quote } (1 2))\)

Your turn: what does evaluating these print out?

(define x 20)
(+ x 3) \implies\ (\text{quote } (\text{quote } (+ x 3)))

'(\text{quote } + x 3) \implies\ (\text{quote } (\text{quote } (+ x 3)))

(list (quote +) x '3) \implies\ (\text{quote } ((\text{quote } +) x '3))

(list '+ x 3) \implies\ (\text{quote } ((\text{quote } +) x '3))

(list + x 3) \implies\ (\text{quote } ((\text{quote } +) x '3))

Davis' Rule of Thumb for \text{Quote}

(list (quote fred (quote quote) (+ 3 5)))

(list (quote (quote fred (quote quote) (+ 3 5))))

???

What's the value of the quoted expression?

WHATEVER IS UNDER YOUR THUMB!

Symbolic differentiation

(deriv \langle expr \rangle \langle with-respect-to-var \rangle) \implies \langle new-expr \rangle

<table>
<thead>
<tr>
<th>Algebraic expression</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x + 3)</td>
<td>(+ x 3)</td>
</tr>
<tr>
<td>(x)</td>
<td>(x)</td>
</tr>
<tr>
<td>(5y)</td>
<td>*(5 y)</td>
</tr>
<tr>
<td>(x + y + 3)</td>
<td>(+ x (+ y 3))</td>
</tr>
</tbody>
</table>

(deriv '(+ x 3) 'x) \implies\ 1
(deriv '(* x y) 'x) \implies\ y
(deriv '(* x x) 'x) \implies\ (+ x x)
Building a system for differentiation
Example of:
• Lists of lists
• How to use the symbol type
• Symbolic manipulation

1. how to get started
2. a direct implementation
3. a better implementation

1. How to get started
• Analyze the problem precisely
  deriv constant dx = 0
  deriv variable dx = 1 if variable is the same as x
  = 0 otherwise
  deriv (e1+e2) dx = deriv e1 dx + deriv e2 dx
  deriv (e1*e2) dx = e1 (deriv e2 dx) + e2 (deriv e1 dx)
• Observe:
  • e1 and e2 might be complex subexpressions
  • derivative of (e1+e2) formed from deriv e1 and deriv e2
  • a tree problem

Type of the data will guide implementation
• legal expressions
  x (+ x y)
  2 (* 2 x) (+ (* x y) 3)
• illegal expressions
  * (3 5 +) (+ x y z)
  () (3) (* x)

; Expr = SimpleExpr | CompoundExpr
; SimpleExpr = number | symbol
; CompoundExpr = a list of three elements where the first
  element is either + or *
; = pair< (+|*), pair<Expr, pair<Expr,null> >>

2. A direct implementation
• Overall plan: one branch for each subpart of the type
  (define deriv (lambda (expr var)
  (if (simple-expr? expr)
    <handle simple expression>
    <handle compound expression>))
• To implement simple-expr? look at the type
  • CompoundExpr is a pair
  • nothing inside SimpleExpr is a pair
  • therefore
    (define simple-expr? (lambda (e)
      (not (pair? e))))

Simple expressions
• One branch for each subpart of the type
  (define deriv (lambda (expr var)
    (if (simple-expr? expr)
      (if (number? expr)
        0
        <handle number>
        (if (eq? expr var)
          1 0)
       <handle compound expression>
    )))
• Implement each branch by looking at the math

Compound expressions
• One branch for each subpart of the type
  (define deriv (lambda (expr var)
    (if (simple-expr? expr)
      (if (number? expr) 0
        (if (eq? (car expr) '+)
          (if (eq? (car expr) 'l 0)
            <handle add expression>
            <handle product expression>
          )))
)))
Sum expressions

- To implement the sum branch, look at the math

```scheme
(define deriv (lambda (expr var)
  (if (simple-expr? expr)
      (if (number? expr) 0
          (if (eq? expr var) 1 0))
      (if (eq? (car expr) '+)
          (list '+
                  (deriv (cadr expr) var)
                  (deriv (caddr expr) var))
          <handle product expression>))
))
```

(deriv '(+ x y) 'x) ==> (+ 1 0) (a list!)

The direct implementation works, but...

- Programs always change after initial design
- Hard to read
- Hard to extend safely to new operators or simple exprs
- Can’t change representation of expressions
- Source of the problems:
  - nested if expressions
  - explicit access to and construction of lists
  - few useful names within the function to guide reader

3. A better implementation

1. Use `cond` instead of nested `if` expressions
2. Use data abstraction

- To use `cond`:
  - write a predicate that collects all tests to get to a branch:
    ```scheme
    (define sum-expr? (lambda (e)
      (and (pair? e) (eq? (car e) '+))))
    ; type: Expr -> boolean
    
    - do this for every branch:
      ```scheme
      (define variable? (lambda (e)
        (and (not (pair? e)) (symbol? e))))
      ```

Use data abstractions

- To eliminate dependence on the representation:

  ```scheme
  (define make-sum (lambda (e1 e2)
    (list '+ e1 e2))
  )
  (define addend (lambda (sum) (cadr sum)))
  ```

A better implementation

```scheme
(define deriv (lambda (expr var)
  (cond
     ((number? expr) 0)
     ((variable? expr) (if (eq? expr var) 1 0))
     ((sum-expr? expr)
      (make-sum (deriv (addend expr) var)
                (deriv (augend expr) var)))
     ((product-expr? expr)
      <handle product expression>)
     (else
      (error "unknown expression type" expr))
  ))
```

Isolating changes to improve performance

```scheme
(deriv '(+ x y) 'x) ==> (+ 1 0) (a list!)
```

```scheme
(define make-sum (lambda (el e2)
  (list '+ el e2)))
```

```scheme
(define make-sum
  (lambda (el e2)
    (cond ((number? el)
           (if (number? e2)
               (+ el e2)
               (list '+ el e2)))
           ((number? e2)
               (list '+ e2 el))
           (else (list '+ el e2))))
)
```

(deriv '(+ x y) 'x) ==> 1
Modularity makes changes easier

- So it seems like a bit of a pain to be using expressions like: (+ 2 x) or (* (+ 3 x) (+ x y))
- It would be cleaner somehow to use more algebraic expressions, like: (2 * x) or ((3 + x) *(x + y))
- What do we need to change?

Modularity helps in other ways

- Rather than changing the code to handle simplifications of expressions, write a separate simplifier:

```lisp
(define (simplify expr)
  (cond ((sum-expr? expr)
             (simplify-sum expr))
        ((product-expr? expr)
             (simplify-product expr))
        (else expr)))
```

Just change data abstraction

- Constructors

  ```lisp
  (define (make-sum el e2)
    (list el '+ e2))
  ```

- Accessors

  ```lisp
  (define (augend expr)
    (car expr))
  ```

- Predicates

  ```lisp
  (define (sum-expr? Expr)
    (and (pair? Expr) (eq? '+ (cadr expr))))
  ```

Separating out aspects of simplification

```lisp
(define (simplify-sum expr)
  (cond ((and (number? (addend expr)) (number? (augend expr)))
             (+ (addend expr) (augend expr))
        ((or (number? (addend expr)) (number? (augend expr)))
             expr)
        ((eq? (addend expr) (augend expr))
             (* 2 x))
        (else expr))
```

```lisp
(define (simplify-product expr)
  (cond ((and (number? (multiplier (augend expr)))
             (eq? (addend expr) (multiplicand (augend expr))))
             (* (+ 1 (multiplier (augend expr)))
                (addend expr))
        (else expr))
```

```lisp
(simplify (deriv '(+ x y) 'x))
```