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 ______)

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

A compound proc that squares its argument

(quote beta)

A useful property of the quote special form

(list (quote delta) (quote delta))

Two quote expressions with the same name return the same object

The operation eq? tests for the same object

(eq? (quote delta) (quote delta)) ==> #t
(eq? (quote delta) (quote delta)) ==> #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) (quote a) a a
2. (quote (a b)) (quote (a b))
3. (quote 1)

Shorthand: the single quote mark

'a is shorthand for (quote a)
'(1 2) (quote (1 2))
# Davis' Rule of Thumb for Quote

\[
\begin{align*}
\text{(list 'quote fred (quote quote) (+ 3 5)))} \\
\text{(list (quote (quote fred (quote quote) (+ 3 5)))}
\end{align*}
\]

???

# Symbolic differentiation

\[
\text{deriv} \quad <\text{expr}> \quad <\text{with-respect-to-var}> \quad \Rightarrow \quad <\text{new-expr}>
\]

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

\[
\begin{align*}
\text{(deriv } '(+ x 3) 'x) & \Rightarrow 1 \\
\text{(deriv } '(+ (* x y) 4) 'x) & \Rightarrow y \\
\text{(deriv } '(* x x) 'x) & \Rightarrow (+ \times x x)
\end{align*}
\]

# 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 \(x\)
- otherwise \(dx = 0\)
- 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
  - \(x\)
  - \((+ \times 2 x)\)
  - \((+ \times 2 x 3)\)
- illegal expressions
  - \((3 5 +)\)
  - \((3)\)
  - \((\times)\)

\[
\begin{align*}
\text{Expr} &= \text{SimpleExpr | CompoundExpr} \\
\text{SimpleExpr} &= \text{number | symbol} \\
\text{CompoundExpr} &= \text{a list of three elements where the first element is either + or \times}
\end{align*}
\]

\[
\begin{align*}
\text{define deriv} &= \text{lambda} \quad \text{expr var} \\
&\quad \text{if simple-expr? expr} \\
&\quad \quad \text{<handle simple expression>} \\
&\quad \quad \text{<handle compound expression>} \\
&\quad \} \\
\end{align*}
\]

- To implement simple-expr? look at the type
  - CompoundExpr is a pair
  - nothing inside SimpleExpr is a pair
  - therefore

\[
\begin{align*}
\text{define simple-expr?} &= \text{lambda} \quad \text{e} \\
&\quad \text{not (pair? e)})
\end{align*}
\]
Simple expressions

- One branch for each subpart of the type

\[
\text{(define deriv (lambda (expr var))}
\begin{align*}
&\text{(if (simple-expr? expr)} \\
&\quad \text{(if (number? expr) 0)} \\
&\quad \text{(handle number)} 0 \\
&\quad \text{(handle symbol)} (\text{if (eq? expr var)} \text{ 1 0}) \\
&\quad \text{(handle compound expression)} >) \\
\end{align*}
\]
- Implement each branch by looking at the math

```
(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))
        )
    )))
```

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? expr var) 1 0)
            (if (eq? (car expr) '+)
                (handle add expression)
                (handle product expression)
            )
        )
    )))
```

Sum expressions

- To implement the sum branch, look at the math

```
(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)
            )
        )
    )))
```

\[
\text{(deriv '(+ x y) 'x) ==> (+ 1 0) (a list!)}
\]

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:
    \[
    \text{(define sum-expr? (lambda (e) (and (not (pair? e)) (symbol? e))))
    \]
    \[
    \text{type: Expr -> boolean}
    \]
  - do this for every branch:
    \[
    \text{(define variable? (lambda e (and (not (pair? e)) (symbol? e)))))
    \]

Use data abstractions

- To eliminate dependence on the representation:

\[
\begin{align*}
&\text{(define make-sum (lambda (el e2))} \\
&\quad \text{(list '+ el e2))} \\
&\text{(define addend (lambda (sum) (cadr sum)))}
\end{align*}
\]
A better implementation

```
(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

```
(define make-sum
  (lambda (el e2)
    (cond ((number? e1)
          (if (number? e2)
            (+ e1 e2)
            (list '+ e1 e2))
          ((number? e2)
            (list '+ e2 e1))
          (else (list '+ e1 e2)))))

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

(define (simplify-sum expr)
  (cond ((and (number? (addend expres)))
          (if (number? (augend expres))
            (+ (addend expres) (augend expres))
            (++ (addend expres) (augend expres))
          (else expr)))
        (else expr)))
```

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?

Just change data abstraction

- Constructors
  (define (make-sum el e2) (list el '+ e2))
  
- Accessors
  (define (augend expres) (car expres))
  
- Predicates
  (define (sum-expr? Expr) (and (pair? Expr) (eq? '+ (cadr expres))))

Modularity helps in other ways

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

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

Separating out aspects of simplification

```
(define (simplify-sum expres)
  (cond ((and (number? (addend expres)) (number? (augend expres)))
          (+ (addend expres) (augend expres))
          (+ 2 x) (+ 2 x))
        ((eq? (addend expres) (augend expres))
          (make-product 2 (addend expres))
          (+ x x) (* 2 x))
        (else expres)))
```

```