### 6.001: Structure and Interpretation of Computer Programs
- Symbols
- Quotation
- Relevant details of the reader
- Example of using symbols
- Differentiation

### Data Types in Lisp/Scheme
- Conventional
  - Numbers (integer, real, rational, complex)
    - Interesting property in “real” Scheme: exactness
  - Booleans: #t, #f
  - Characters and strings: #\a, “Hello World!”
  - Vectors: #(0 “hi” 3.7)
- Lisp-specific
  - Procedures: value of +, result of evaluating (\( \lambda (x) x \))
  - Pairs and Lists: (3 . 7), (1 2 3 5 7 11 13 17)
  - Symbols: pi, +, MyGreatGrandMotherSue

### Symbols
- So far, we’ve seen them as the names of variables
- But, in Lisp, all data types are first class
- Therefore, we should be able to:
  - Pass symbols as arguments to procedures
  - Return them as values of procedures
  - Associate them as values of variables
  - Store them in data structures
  - E.g., (apple orange banana)

### How do we refer to Symbols?
- Substitution Model’s rule of evaluation:
  - Value of a symbol is the value it is associated with in the environment
  - We associate symbols with values using the special form define
    - `(define pi 3.1415926535)`
  - … but that doesn’t help us get at the symbol itself

### Referring to Symbols
- Say your favorite color
- Say “your favorite color”
- In the first case, we want the meaning associated with the expression, e.g.,
  - red
- In the second, we want the expression itself, e.g.,
  - your favorite color
- We use quotation to distinguish our intended meaning

### New Special Form: quote
- Need a way of telling interpreter: “I want the following object as whatever it is, not as an expression to be evaluated”

```scheme
(quote alpha)
; Value: alpha

(define pi 3.1415926535)
; Value: "pi --> 3.1415926535"

(pi)
; Value: 3.1415926535

(define fav (quote pi))

(define pi)
; Value: pi
```

```scheme
(quote pi)
; Value: pi
```

```scheme
(+ pi pi)
; Value: 6.283185307
```

```scheme
(+ pi (quote pi))
; The object pi, passed as the first argument to integer->flonum, is not the correct type.
```

```scheme
(define fav (quote pi))
```

```scheme
(define pi)
```

```scheme
(define pi)
```

```scheme
(define pi)
```

```scheme
(define pi)
```

```scheme
(define pi)
```
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>

Symbol: a primitive type

- **constructors:**
  
  None since really a primitive, not an object with parts

- **selectors:**

  None

- **operations:**

  (define symbol->string
     (lambda (symbol) stringify symbol))

  (define r5rs-shows-full-riches-of-scheme
     (lambda ()...))

What’s the difference between symbols and strings?

- **Symbol**
  
  Evaluates to the value associated with it by define

  Every time you type a particular symbol, you get the exact same one! Guaranteed.

  E.g. (list (quote pi) (quote pi))

- **String**
  
  Evaluates to itself

  Every time you type a particular string, it's up to the implementation whether you get the same one or different ones.

  E.g. (list "pi" "pi")

  or

  "pi" "pi"

  or

  "pi" "pi"

  "pi"

Making list structure with symbols

((red 700) (orange 600) (yellow 575) (green 550)
 (cyan 510) (blue 470) (violet 400))

(list (quote red) 700) (list (quote orange) 600)
...
(list (quote violet) 400))

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?

More Syntactic Sugar

- To the reader,

  ’pi

  is exactly the same as if you had typed

  (quote pi)

- Remember REPL

  User types

  ;Value: pi

  ’pi

  ;Value: pi

  read

  print

  (quote pi)

  pi
More Syntactic Sugar
- To the reader, ‘pi is exactly the same as if you had typed (quote pi)
- Remember REPL

User types

'17
read
(quote 17)
print

'17
;Value: 17

'pi
;Value: pi

User types

'(quote pi)
read
print

'pi
;Value: (quote pi)

But in Dr. Scheme, ‘pi

But wait... Clues about “guts” of Scheme
(pair? (quote (+ 2 3)))
;Value: #t

(pair? '(+ 2 3))
;Value: #t

car '(+ 2 3))
;Value: +

cadr '(+ 2 3))
;Value: 2

Now we know that expressions are represented by lists!

Your turn: what does evaluating these print out?

(define x 20)
(+ x 3) ➞
'(+ x 3) ➞
(list (quote +) (quote x) '3) ➞
(list '(+ x 3)) ➞
(list (+ x 3)) ➞

Your turn: what does evaluating these print out?

(define x 20)
(+ x 3) ➞ 23
'(+ x 3) ➞ (+ x 3)
(list (quote +) (quote x) '3) ➞ (+ 20 3)
(list '(+ x 3)) ➞ (+ 20 3)
(list (+ x 3)) ➞ ([procedure #…] 20 3)
Revisit making list structure with symbols

```plaintext
red 700  orange 600  violet 400
```

(list (list (quote red) 700) (list (quote orange) 600))

(list 'red 700) (list 'orange 600) ... (list 'violet 400))

'((red 700) (orange 600) (yellow 575) (green 550) (cyan 510) (blue 470) (violet 400))

Because the reader knows how to turn parenthesized (for lists) and dotted (for pairs) expressions into list structure!

Aside: What all does the reader “know”?

- Recognizes and creates
  - Various kinds of numbers
    - 312 ==> integer
    - 3.12e17 ==> real, etc.
  - Strings enclosed by ‘’
  - Booleans #t and #f
  - Symbols
    - ‘… ==> (quote …)
    - (...) ==> pairs (and lists, which are made of pairs)
  - and a few other obscure things

Symbolic differentiation

```
(deriv \(<\text{expr}>\) <with-respect-to-var>) \implies \langle\text{new-expr}\rangle
```

Algebraic expression | Representation
--- | ---
\(x + 3\) | \(+\ x\ 3\)
\(x\) | \(x\)
\(5y\) | \("\ 5y\"
\(x + y + 3\) | \(+\ x\ (+\ y\ 3)\)

```
(deriv \('('\ x'\ 3) 'x') \implies 1
(deriv \('('\ (+\ (*\ x\ y) 4) '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

```plaintext
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

```plaintext
; 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

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

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

\[
\text{(define simple-expr? (lambda (e) (not (pair? e)))}
\]

Simple expressions

• One branch for each subpart of the type

\[
\text{(define deriv (lambda (expr var) (if (simple-expr? expr) (if (number? expr) 0 (if (eq? expr var) 1 0)) (if (eq? (car expr) '+) (list '+ (deriv (cadr expr) var) (deriv (caddr expr) var)) (deriv product expression)))))}
\]

• Implement each branch by looking at the math

Compound expressions

• One branch for each subpart of the type

\[
\text{(define deriv (lambda (expr var) (if (simple-expr? expr) (if (number? expr) 0 (if (eq? expr var) 1 0)) (if (eq? (car expr) '+) (list '+ (deriv (cadr expr) var) (deriv (caddr expr) var)) (deriv product expression)))))}
\]

Sum expressions

• To implement the sum branch, look at the math

\[
\text{(define deriv (lambda (expr var) (if (simple-expr? expr) (if (number? expr) 0 (if (eq? expr var) 1 0)) (if (eq? (car expr) '+) (list '+ (deriv (cadr expr) var) (deriv (caddr expr) var)) (deriv product expression)))))}
\]

\[
(deriv '(+ x y) 'x) \Rightarrow (+ 1 0)
\]

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:
    \[
    \text{(define sum-expr? (lambda (e) (and (pair? e) (eq? (car e) '+))) ; type: Expr \Rightarrow boolean}
    \]
  • do this for every branch:
    \[
    \text{(define variable? (lambda (e) (and (not (pair? e)) (symbol? e)))}
    \]

\[
(deriv '(+ x y) 'x) \Rightarrow (+ 1 0)
\]
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)))

(define augend (lambda (sum) (caddr 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 (e1 e2)
        (cond ((number? e1)
                (if (number? e2)
                    (+ e1 e2)
                    (list '+ e1 e2)))
            ((number? e2)
                (list '+ e2 e1))
            (else (list '+ e1 e2)))))
```

```scheme
(deriv '(+ x y) 'x) ==> 1
```

Modularity makes changes easier

- But conventional mathematics doesn’t use prefix notation like this:
  (+ 2 x) or (* (+ 3 x) (+ x y))
  (2 + x) or ((3 + x) * (x + y))
  What do we need to change?

Just change data abstraction

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

- Accessors
  ```scheme
  (define (augend expr)
      (caddr expr))
  ```

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

Separating simplification from differentiation

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

```scheme
(define (simplify expr)
    (cond ((or (number? expr) (variable? expr))
            expr)
          ((sum-expr? expr)
            (simplify-sum
              (simplify (addend expr))
              (simplify (augend expr))))
          ((product-expr? expr)
            (simplify-product
              (simplify (multiplier expr))
              (simplify (multiplicand expr))))
          (else (error "unknown expr type" expr))))
```
Simplifying sums

\[
\text{(define (simplify-sum add aug)}
\begin{cases}
\text{((and (number? add) (number? aug))} & \text{both terms are numbers: add them} \\
\text{(+ add aug))} & \text{(+ 2 3) } \rightarrow \text{ 5} \\
\text{((or (number? add) (number? aug))} & \text{one term only is number} \\
\text{((and (number? add) (zero? add))} & \text{(+ 0 x) } \rightarrow \text{ x} \\
\text{(zero? aug))} & \text{(+ x 0) } \rightarrow \text{ x} \\
\text{aug)} & \text{(else (make-sum add aug)))} & \text{(+ 2 x) } \rightarrow \text{ (+ 2 x)} \\
\text{(eq? add aug))} & \text{(+ x x) } \rightarrow \text{ (* 2 x)} \\
\text{else (make-sum add aug)))} & \text{(* 4 x)} \\
\end{cases}
\]

More special cases in simplification

\[
\text{(define (simplify-sum add aug)}
\begin{cases}
\text{((product-expr? aug))} & \text{check for special case of (+ x (* 3 x))} \\
\text{;; i.e., adding something to a multiple of itself} \\
\text{((and (mul (simplify (multiplier aug)))} & \text{(mul (simplify (multiplicand aug)))} \\
\text{((if (and (number? mulr) (eq? add muld))} & \text{(+ x (* 3 x)) } \rightarrow \text{ (+ 4 x)} \\
\text{(make-product (+ 1 mulr) add))} & \text{not special case: lose} \\
\text{((else (make-sum add aug)))})} & \text{(make-sum add aug))} \\
\end{cases}
\]

Special cases in simplifying products

\[
\text{(define (simplify-product f1 f2))}
\begin{cases}
\text{((and (number? f1) (number? f2))} & \text{(* 3 5) } \rightarrow \text{ 15} \\
\text{((number? f1) (number? f2))} & \text{(* 0 (+ x 1)) } \rightarrow \text{ 0} \\
\text{((= f1 1) f2)} & \text{(* 1 (+ x 1)) } \rightarrow \text{ (+ x 1)} \\
\text{((else (make-product f1 f2)))} & \text{(+ 2 x)} \\
\text{(number? f2))} & \text{(* 2 x)} \\
\text{((zero? f2) 0)} & \text{(+ (* x y) (* x y)))} & \text{Value: (+ 1 (+ (* x 0) (* 1 y)))} \\
\text{((else (make-product f1 f2)))} & \text{(+ x 1)} \\
\text{if (and (number? f1) (eq? add f2))} & \text{(* a b c)} \\
\text{else (make-product f1 f2)))} & \text{or} \\
\end{cases}
\]

Simplified derivative looks better

\[
\text{(deriv '(+ 3 'x) 'x)} & \text{Value: (+ 1 0)} \\
\text{(simplify (deriv '(+ 3 'x) 'x))} & \text{Value: 1} \\
\text{(deriv '(+ x (* x y)) 'x) & Value: (+ 1 (+ (* x 0) (* 1 y)))} \\
\text{(simplify (deriv '(' x y)) 'x)} & \text{Value: (+ 1 y)} \\
\text{• But, which is simpler?} \\
\text{• a b c) } & \text{or} \\
\text{• a b + a c) } & \text{Depends on context...}
\]

Recap

• Symbols
  • Are first class objects
  • Allow us to represent names
• Quotation (and the reader’s syntactic sugar for ‘) 
  • Let us evaluate (quote ...) to get ... as the value 
    – I.e., “prevents one evaluation” 
    – Not really, but informally, has that effect.
• Lisp expressions are represented as lists
  • Encourages writing programs that manipulate programs 
    – Much more, later
• Symbolic differentiation (introduction)