6.001: Structure and Interpretation of Computer Programs
- Symbols
- Quotation
- Relevant details of the reader
- Example of using symbols
- Alists
- 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.,
- In the second, we want the expression itself, e.g.,
- 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"

```lisp
(quote alpha)
;Value: alpha
(+ pi pi)
;Value: 6.283185307
(define pi 3.1415926535)
;Value: "pi --> 3.1415926535"
(pi)
;Value: 3.1415926535
(quote pi)
;Value: pi
(define fav (quote pi))
(fav)
;Value: pi
```
Review: data abstraction

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

Symbol: a primitive type

- **constructors:**
  None since really a primitive, not an object with parts
- **operations:**
  ```scheme```
  ```
  (eq? (quote eps) (quote eps))   ==> #t
  (eq? (quote delta) (quote eps)) ==> #f
  ```
- **selectors**
  None
- **operations:**
  ```scheme```
  ```
  (symbol? (quote alpha)) => #t
  (eq? ; discuss in a minute
  ```

What's the difference between **symbols** and **strings**?

- **Symbol**
  - Evaluates to the value associated with it in define
  - Every time you type a particular symbol, you get the exact same one. Guaranteed.
  - Called **interning**
  - 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 or different ones.
  - E.g., `(list "pi" "pi")`

The operation `eq?` tests for the same object

- A primitive procedure
- Returns `#t` if its two arguments are the same object
- Very fast

```scheme```
```
(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?`

Making list structure with symbols

```scheme```
```
(list (list (quote red) 700) (list (quote orange) 600) … (list (quote violet) 400))
```
More Syntactic Sugar

• To the reader, '
pi
is exactly the same as if you had typed
(quote pi)
• Remember REPL

User types


(pi) ;Value: pi

'17 ;Value: 17

"hi there" ;Value: "hi there"

But in Dr. Scheme,

'pi ;Value: (quote pi)

But in Dr. Scheme,

'pi ;Value: (quote pi)

But in Dr. Scheme,

'pi ;Value: (quote pi)

But in Dr. Scheme,

'pi ;Value: (quote pi)

But in Dr. Scheme,

'pi ;Value: (quote pi)

But in Dr. Scheme,

'pi ;Value: (quote pi)

But in Dr. Scheme,

'pi ;Value: (quote pi)

But in Dr. Scheme,

'pi ;Value: (quote pi)

But in Dr. Scheme,

'pi ;Value: (quote pi)

But in Dr. Scheme,

'pi ;Value: (quote pi)

But in Dr. Scheme,

'pi ;Value: (quote pi)

But in Dr. Scheme,

'pi ;Value: (quote pi)

But in Dr. Scheme,

'pi ;Value: (quote pi)

But in Dr. Scheme,

'pi ;Value: (quote pi)

But in Dr. Scheme,

'pi ;Value: (quote pi)

But in Dr. Scheme,

'pi ;Value: (quote pi)

But in Dr. Scheme,

'pi ;Value: (quote pi)

But in Dr. Scheme,

'pi ;Value: (quote pi)

But in Dr. Scheme,

'pi ;Value: (quote pi)

But in Dr. Scheme,

'pi ;Value: (quote pi)

But in Dr. Scheme,

'pi ;Value: (quote pi)

But in Dr. Scheme,

'pi ;Value: (quote pi)

But in Dr. Scheme,

'pi ;Value: (quote pi)

But in Dr. Scheme,

'pi ;Value: (quote pi)

But in Dr. Scheme,

'pi ;Value: (quote pi)

But in Dr. Scheme,

'pi ;Value: (quote pi)

But in Dr. Scheme,

'pi ;Value: (quote pi)

But in Dr. Scheme,

'pi ;Value: (quote pi)

But in Dr. Scheme,

'pi ;Value: (quote pi)

But in Dr. Scheme,

'pi ;Value: (quote pi)

But in Dr. Scheme,

'pi ;Value: (quote pi)

But in Dr. Scheme,

'pi ;Value: (quote pi)

But in Dr. Scheme,

'pi ;Value: (quote pi)

But in Dr. Scheme,

'pi ;Value: (quote pi)

But in Dr. Scheme,

'pi ;Value: (quote pi)

But in Dr. Scheme,

'pi ;Value: (quote pi)

But in Dr. Scheme,

'pi ;Value: (quote pi)

But in Dr. Scheme,

'pi ;Value: (quote pi)

But in Dr. Scheme,

'pi ;Value: (quote pi)

But in Dr. Scheme,

'pi ;Value: (quote pi)

But in Dr. Scheme,

'pi ;Value: (quote pi)

But in Dr. Scheme,

'pi ;Value: (quote pi)

But in Dr. Scheme,

'pi ;Value: (quote pi)

But in Dr. Scheme,

'pi ;Value: (quote pi)

But in Dr. Scheme,

'pi ;Value: (quote pi)

But in Dr. Scheme,

'pi ;Value: (quote pi)

But in Dr. Scheme,

'pi ;Value: (quote pi)

But in Dr. Scheme,

'pi ;Value: (quote pi)

But in Dr. Scheme,

'pi ;Value: (quote pi)

But in Dr. Scheme,

'pi ;Value: (quote pi)

But in Dr. Scheme,

'pi ;Value: (quote pi)

But in Dr. Scheme,

'pi ;Value: (quote pi)

But in Dr. Scheme,

'pi ;Value: (quote pi)

But in Dr. Scheme,

'pi ;Value: (quote pi)

But in Dr. Scheme,

'pi ;Value: (quote pi)

But in Dr. Scheme,

'pi ;Value: (quote pi)

But in Dr. Scheme,

'pi ;Value: (quote pi)

But in Dr. Scheme,

'pi ;Value: (quote pi)

But in Dr. Scheme,

'pi ;Value: (quote pi)

But in Dr. Scheme,

'pi ;Value: (quote pi)

But in Dr. Scheme,

'pi ;Value: (quote pi)

But in Dr. Scheme,

'pi ;Value: (quote pi)

But in Dr. Scheme,

'pi ;Value: (quote pi)

But in Dr. Scheme,

'pi ;Value: (quote pi)

But in Dr. Scheme,

'pi ;Value: (quote pi)

But in Dr. Scheme,

'pi ;Value: (quote pi)

But in Dr. Scheme,

'pi ;Value: (quote pi)

But in Dr. Scheme,

'pi ;Value: (quote pi)

But in Dr. Scheme,

'pi ;Value: (quote pi)

But in Dr. Scheme,

'pi ;Value: (quote pi)

But in Dr. Scheme,

'pi ;Value: (quote pi)

But in Dr. Scheme,

'pi ;Value: (quote pi)

But in Dr. Scheme,

'pi ;Value: (quote pi)

But in Dr. Scheme,

'pi ;Value: (quote pi)

But in Dr. Scheme,

'pi ;Value: (quote pi)

But in Dr. Scheme,
Revisit making list structure with symbols

- red 700
- orange 600
- violet 400

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

(list (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

Traditional LISP structure: association list

- A list where each element is a list of the key and value.
- Represent the table

<table>
<thead>
<tr>
<th>x</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>20</td>
</tr>
</tbody>
</table>

as the alist: ((x 15) (y 20))

Alist operation: find-assoc

\[
\text{define (find-assoc key alist)} \rightarrow \begin{cases}
\text{null? alist} \\ #f
\text{(equal? key (caar alist)) (cadar alist)}
\text{(else (find-assoc key (cdr alist)))}
\end{cases}
\]

\[
\text{define a1 '((x 15) (y 20))}
\]

(find-assoc 'y a1) \rightarrow 20

An aside on testing equality

- = tests equality of numbers
- Eq? Tests equality of symbols
- Equal? Tests equality of symbols, numbers or lists of symbols and/or numbers that print the same

Alist operation: add-assoc

\[
\text{define (add-assoc key val alist)} \rightarrow \text{cons (list key val) alist)}
\]

\[
\text{define a2 (add-assoc 'y 10 a1)}
\]

a2 \rightarrow ((y 10) (x 15) (y 20))

(find-assoc 'y a2) \rightarrow 10

We say that the new binding for y "shadows" the previous one
Alists are not an abstract data type

- Missing a constructor:
  - Used quote or list to construct
    ```lisp
    (define al '((x 15) (y 20)))
    ```
- There is no abstraction barrier: the implementation is exposed.
- User may operate on alists using standard list operations.
  ```lisp
  (filter (lambda (a) (< (cadr a) 16)) al))
  ==> ((x 15))
  ```

Why do we care that Alists are not an ADT?

- Modularity is essential for software engineering
  - Build a program by sticking modules together
  - Can change one module without affecting the rest
- Alists have poor modularity
  - Programs may use list ops like `filter` and `map` on alists
  - These ops will fail if the implementation of alists change
  - Must change whole program if you want a different table
- To achieve modularity, hide information
  - Hide the fact that the table is implemented as a list
  - Do not allow rest of program to use list operations
  - ADT techniques exist in order to do this

Symbolic differentiation

- **Derivative**
  - ```lisp
    (deriv <expr> <with-respect-to-var>) ==> <new-expr>
    ```
  - **Algebraic expression**
    - ```lisp
      x + 3
      x
      5y
      x + y + 3
      ```
    - **Representation**
      - ```lisp
        (+ x 3)
        x
        (* 5 y)
        (+ x (+ y 3))
        ```
  - **Example**
    - ```lisp
      (deriv '(+ x 3) 'x)       ==> 1
      (deriv '(* x y) 'x)       ==> y
      (deriv '(* x x) 'x)       ==> (+ 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**
  - ```lisp
    x (+ x y) 2 (+ 2 x) (+ (* x y) 3)
    ```
- **Illegal expressions**
  - ```lisp
    * (3 5) (+ x y z) (3) (* x)
    ```

```scheme
; 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)))
    ```

---

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:
    ```
    (define sum-expr? (lambda (e)
      (and (pair? e) (eq? (car e) '+))
      (list '+
        (deriv (cadadr expr) var)
        (deriv (cadexp expr) var))
    ))
    ```

- do this for every branch:
  ```
  (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)))
(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
(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))))

(define addend (lambda (sum) (cadr sum)))
```

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

Modularity makes changes easier
• But conventional mathematics doesn’t use prefix notation like this:
  (+ 2 x) or (* (+ 3 x) (+ x y))
• Could we change our program somehow to use more algebraic expressions, still fully parenthesized, like:
  (2 + x) or ((3 + x) * (x + y))
• What do we need to change?

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

(define augend (lambda (expr) (caddr expr)))

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

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

\begin{verbatim}
(define (simplify-sum add aug)
  (cond ((and (number? add) (number? aug)) ;; both terms are numbers: add them
         (+ add aug))
        (or (number? add) (number? aug)) ;; one term only is number
         (cond ((and (number? add) (zero? add)) add) ;; adding same term twice
                ((and (number? aug) (zero? aug)) aug)
                (else (make-sum add aug))))
        (eq? add aug) ;; adding same term twice
        (make-product 2 add)))
\end{verbatim}

More special cases in simplification

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

Special cases in simplifying products

\begin{verbatim}
(define (simplify-product f1 f2)
  (cond ((and (number? f1) (number? f2)) (* f1 f2)) ;; (* 3 5) Æ 15
        ((number? f1) (cond ((zero? f1) 0) ;; (* 0 (+ x 1)) Æ 0
                              ((= f1 1) f2) ;; (* 1 (+ x 1)) Æ (+ x 1)
                              (else (make-product f1 f2))))
        ((number? f2) (cond ((zero? f2) 0) ;; (* 2 0) Æ 0
                              ((= f2 1) f1) ;; (+ x 0) Æ (+ x 1)
                              (else (make-product f2 f1))))
        (else (make-product f1 f2))))
\end{verbatim}

Simplified derivative looks better

\begin{verbatim}
(deriv '(+ x 3) 'x) ;;Value: (+ 1 0)
(deriv '(+ x (* x y)) 'x) ;;Value: (+ 1 (+ (* x 0) (* 1 y)))
\end{verbatim}

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)