6.001: Structure and Interpretation of Computer Programs

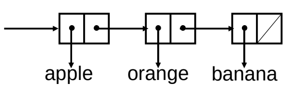
- Symbols
- Quotation
  - Relevant details of the reader
- Example of using symbols
  - Alists
  - Differentiation

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 (λ (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”

```
(quote alpha)
;Value: alpha

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

pi
;Value: 3.1415926535

(quote pi)
;Value: pi
```

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

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

(define fav (quote pi))

fav
;Value: 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
  - Only way to “make one” is to type it
    - (or via `string->symbol` from character strings, but shhhh…)
- selectors
  None
  - (except `symbol->string`)

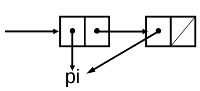
R5RS shows the full riches of Scheme

- operations:

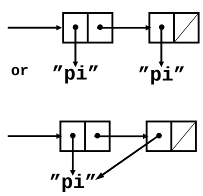
```
symbol?   ; type: anytype -> boolean
   (symbol? (quote alpha)) ==> #t

eq?         ; discuss in a minute
```


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

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

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

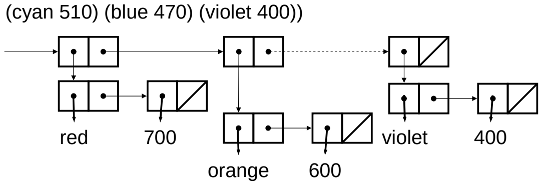
- For those who are interested:

```
; 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

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



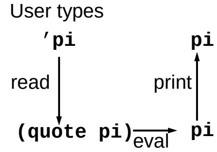
(list (list (quote red) 700) (list (quote orange) 600)
… (list (quote violet) 400))

More Syntactic Sugar

- To the reader,
  `’pi`
  is exactly the same as
  if you had typed
  `(quote pi)`
- Remember REPL

```
'pi
;Value: pi
```

User types



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)` → eval → `17` → print → `17`

```
'pi
;Value: pi

'17
;Value: 17

'"hi there"
;Value: "hi there"
```


More Syntactic Sugar

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

User types

`'(+ 3 4)` → read → `(quote (+ 3 4))` → eval → `(+ 3 4)` → print → `(+ 3 4)`

```
'pi
;Value: pi

'17
;Value: 17

'"hi there"
;Value: "hi there"

'(+ 3 4)
;Value: (+ 3 4)
```


More Syntactic Sugar

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

User types

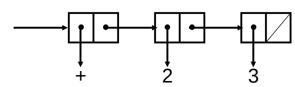
`''pi` → read → `(quote (quote pi))` → eval → `(quote pi)` → print → `(quote pi)`

```
'pi
;Value: pi

'17
;Value: 17

'"hi there"
;Value: "hi there"

'(+ 3 4)
;Value: (+ 3 4)

''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

(null? (cdddr '(+ 2 3)))
;Value: #t
```



Now we know that *expressions* are represented by *lists!*

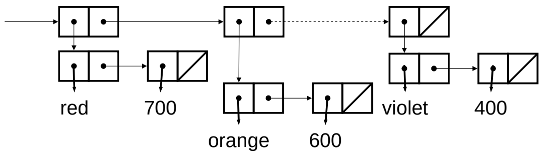
Your turn: what does evaluating these print out?

```
(define x 20)

(+ x 3)                ==>

'(+ x 3)               ==>

(list (quote +) x '3)  ==>

(list '+ x 3)          ==>

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


Grimson's Rule of Thumb for Quote

```
'(quote fred (quote quote) (+ 3 5))

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

???
```

Revisit making list structure with symbols



(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"?

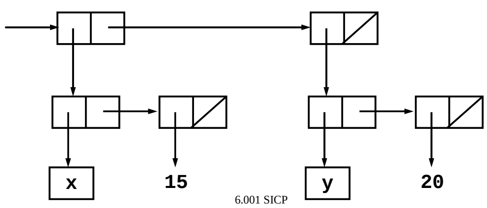
- Recognizes and creates
  - Various kinds of numbers
    - 312 ==> integer
    - 3.12e17 ==> real, etc.
  - Strings enclosed by "…"
  - Booleans #t and #f
  - Symbols
  - '… ==> (quote …)
  - (…) ==> pairs (and lists, which are made of pairs)
  - and a few other obscure things

Traditional LISP structure: association list

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

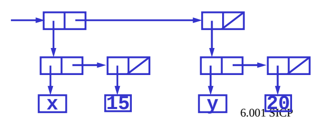
```
x: 15
y: 20
```

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



Alist operation: find-assoc

```
(define (find-assoc key alist)
  (cond
    ((null? alist) #f)
    ((equal? key (caar alist)) (cadar alist))
    (else (find-assoc key (cdr alist)))))

(define a1 '((x 15) (y 20)))
(find-assoc 'y a1)  ==> 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

```
(define (add-assoc key val alist)
   (cons (list key val) alist))

(define a2 (add-assoc 'y 10 a1))

a2                   ==> ((y 10) (x 15) (y 20))

(find-assoc 'y a2) ==> 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

```
(define a1 '((x 15) (y 20)))
```

- There is no abstraction barrier: the implementation is exposed.
- User may operate on alists using standard list operations.

```
(filter (lambda (a) (< (cadr a) 16)) a1))
           ==> ((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

(deriv <expr> <with-respect-to-var>**) ==>** <new-expr>

Algebraic expression	Representation
x + 3	(+ x 3)
x	x
5y	(* 5 y)
x + y + 3	(+ x (+ y 3))

```
(deriv '(+ x 3) 'x)       ==> 1
(deriv '(+ (* x y) 4) '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

```
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)
          <handle number>   0
          <handle symbol>  (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? 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>
      )
  )))
```

`(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:

```
(define sum-expr? (lambda (e)
    (and (pair? e) (eq? (car e) '+))))
; type: Expr -> boolean
```

- do this for every branch:

```
(define variable? (lambda (e)
    (and (not (pair? e)) (symbol? e))))
```

Use data abstractions

- To eliminate dependence on the representation:

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

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


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

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

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

```
(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))

- 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?

Just change data abstraction

- Constructors

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

- Accessors

```
(define (augend expr)
   (caddr expr))
```

- Predicates

```
(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

```
(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

```
(define (simplify-sum add aug)
  (cond
   ((and (number? add) (number? aug))
    ;; both terms are numbers: add them
    (+ add aug))                          (+ 2 3) → 5
   ((or (number? add)
        (number? aug))
    ;; one term only is number
    (cond ((and (number? add)
                (zero? add))
           aug)                           (+ 0 x) → x
          ((and (number? aug)
                (zero? aug))              (+ x 0) → x
           add)
          (else (make-sum add aug))))     (+ 2 x) → (+ 2 x)
   ((eq? add aug)
    ;; adding same term twice
    (make-product 2 add))                 (+ x x) → (* 2 x)
   …
```


More special cases in simplification

```
(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))            (+ x (* 3 x)) → (* 4 x)
          (make-product (+ 1 mulr) add)
          ;; not special case: lose
          (make-sum add aug))))
   (else (make-sum add aug))))
```


Special cases in simplifying products

```
(define (simplify-product f1 f2)
  (cond ((and (number? f1) (number? f2))
         (* f1 f2))                       (* 3 5) → 15
        ((number? f1)
         (cond ((zero? f1) 0)             (* 0 (+ x 1)) → 0
               ((= f1 1) f2)              (* 1 (+ x 1)) → (+ x 1)
               (else (make-product f1 f2))))
        ((number? f2)
         (cond ((zero? f2) 0)
               ((= f2 1) f1)              (* (+ 3 x) 2) → (* 2 (+ 3 x))
               (else (make-product f2 f1))))
        (else (make-product f1 f2))))
```


Simplified derivative looks better

(deriv '(+ x 3) 'x) ;Value: (+ 1 0)	(simplify (deriv '(+ x 3) 'x)) ;Value: 1
(deriv '(+ x (* x y)) 'x) ;Value: (+ 1 (+ (* x 0) (* 1 y)))	(simplify (deriv '(+ x (* x y)) 'x)) ;Value: (+ 1 y)

- But, which is simpler?
  - a*(b+c)

  or
  - a*b + a*c
- 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)