

6.001: Structure and Interpretation of Computer Programs

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

6.001 SICP

1/41

Data Types in Lisp/Scheme

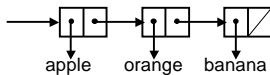
- Conventional
 - Numbers (integer, real, rational, complex)
 - Interesting property in “real” Scheme: exactness
 - Booleans: #t, #f
 - Characters and strings: #\a, “Hello World!”
 - Vectors: #(0 “hi” 3.7)
- Lisp-specific
 - Procedures: value of +, result of evaluating (λ (x) x)
 - Pairs and Lists: (3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

6.001 SICP

2/41

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)



6.001 SICP

3/41

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

6.001 SICP

4/41

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

6.001 SICP

5/41

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

(define pi 3.1415926535) (+ pi (quote pi))
;Value: "pi --> 3.1415926535" ;The object pi, passed as
                                the first argument to
                                integer->flonum, is not
                                the correct type.

pi
;Value: 3.1415926535

(define fav (quote pi))

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

6.001 SICP

6/41

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>
```

6.001 SICP

7/41

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`)
- operations:
 - `symbol?` ; type: anytype -> boolean
 - (`symbol? (quote alpha)`) ==> #t
 - `eq?` ; discuss in a minute

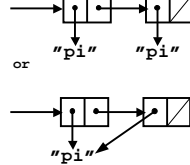
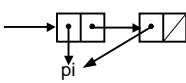
R5RS shows the full riches of Scheme

6.001 SICP

8/41

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")



6.001 SICP

9/41

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
```

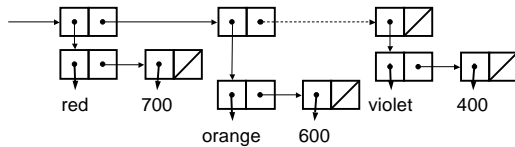
- For those who are interested:
 - `eq?`: EQtype, EQtype ==> boolean
 - `EQtype` = any type **except number or string**
- One should therefore use = for equality of numbers, not eq?

6.001 SICP

10/41

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))
```

6.001 SICP

11/41

More Syntactic Sugar

- To the reader,
 - `'pi` ;Value: pi
- is exactly the same as if you had typed

```
(quote pi)
```

- Remember REPL

User types

```
'pi
read
↓
(quote pi)
↑
eval
pi
print
↑
pi
```

6.001 SICP

12/41

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"
  
```

6.001 SICP 13/41

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)
  
```

6.001 SICP 14/41

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
  
```

6.001 SICP 15/41

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

Now we know that expressions are represented by lists!

6.001 SICP 16/41

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) ==>
  
```

6.001 SICP 17/41

Your turn: what does evaluating these print out?

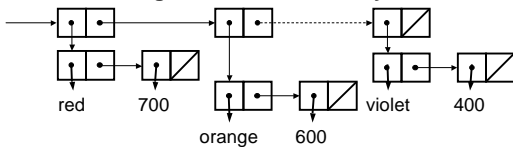
```

(define x 20)

(+ x 3) ==> 23
'(+ x 3) ==> (+ x 3)
(list (quote +) x '3) ==> (+ 20 3)
(list '+ x 3) ==> (+ 20 3)
(list + x 3) ==> ([procedure #...] 20 3)
  
```

6.001 SICP 18/41

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!

6.001 SICP

19/41

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

6.001 SICP

20/41

Symbolic differentiation

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

Algebraic expression	Representation
$x + 3$	<code>(+ x 3)</code>
x	<code>x</code>
$5y$	<code>(* 5 y)</code>
$x + y + 3$	<code>(+ x (+ y 3))</code>

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

6.001 SICP

21/41

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

6.001 SICP

22/41

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

6.001 SICP

23/41

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> >>
```

6.001 SICP

24/41

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))))
```

6.001 SICP

28/41

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

6.001 SICP

28/41

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>
      )
  )))
```

6.001 SICP

27/41

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!)
```

6.001 SICP

28/41

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

6.001 SICP

28/41

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))))
```

6.001 SICP

30/41

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)))
```

6.001 SICP

33/41

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))
  ))
```

6.001 SICP

32/41

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
```

6.001 SICP

33/41

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?

6.001 SICP

34/41

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))))
```

6.001 SICP

35/41

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))))
```

6.001 SICP

36/41

Simplifying sums

```
(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? aug))
            aug)
           ((and (number? aug)
                 (zero? add))
            add)
           (else (make-sum add aug))))
    ((eq? add aug)
     ;; adding same term twice
     (make-product 2 add)))
```

6.001 SICP 37/41

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)))
           (mld (simplify (multiplicand aug))))
       (if (and (number? mulr)
                (eq? add mld))
           (make-product (+ 1 mulr) add)
           ;; not special case: lose
           (make-sum add aug))))
    (else (make-sum add aug))))
```

6.001 SICP 38/41

Special cases in simplifying products

```
(define (simplify-product f1 f2)
  (cond ((and (number? f1) (number? f2))
        (* f1 f2))
        ((number? f1)
         (cond ((zero? f1) 0)
               ((= f1 1) f2)
               (else (make-product f1 f2))))
        ((number? f2)
         (cond ((zero? f2) 0)
               ((= f2 1) f1)
               (else (make-product f2 f1))))
        (else (make-product f1 f2))))
```

6.001 SICP 39/41

Simplified derivative looks better

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

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

- But, which is simpler?
 - $a^*(b+c)$
 - or
 - $a*b + a*c$
- Depends on context...

6.001 SICP

40/41

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)

6.001 SICP

41/41