

6.001: Structure and Interpretation of Computer Programs

- Symbols
- Example of using symbols
 - Differentiation

6.001 SICP

1/31

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

2/31

Symbols?

- Say your favorite color
- Say "your favorite color"
- What is the difference?

6.001 SICP

3/31

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 _____)
```

6.001 SICP

5/31

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
```

6.001 SICP

6/31

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
```

6.001 SICP

7/31

Symbol: printed representation

```
(lambda (x) (* x x))
```

eval
lambda-rule

#[compound-...]

print

A compound proc that squares its argument

```
(quote beta)
```

eval
quote-rule

beta

print

symbol
beta

6.001 SICP 8/31

Symbols are ordinary values

```
(list 1 2) ==> (1 2)
```

```
(list (quote delta) (quote gamma)) ==> (delta gamma)
```

6.001 SICP 9/31

A useful property of the quote special form

```
(list (quote delta) (quote delta))
```

Two quote expressions with the same name return the same object

6.001 SICP 10/31

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 11/31

Generalization: quoting other expressions

Expression:	Reader converts to:	Prints out as:
1. (quote a)		
2. (quote (a b))		
3. (quote 1)		

6.001 SICP 12/31

Shorthand: the single quote mark

'a is shorthand for (quote a)

'(1 2) (quote (1 2))

6.001 SICP 13/31

Davis' Rule of Thumb for Quote

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

6.001 SICP

16/31

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)
```

6.001 SICP

18/31

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

19/31

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

20/31

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

21/31

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

22/31

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

23/31

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

24/31

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

25/31

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

26/31

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

27/31

Use data abstractions

- To eliminate dependence on the representation:

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

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

6.001 SICP

28/31

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

29/31

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

30/31

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?

6.001 SICP

31/31

Just change data abstraction

- Constructors

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

- Accessors

```
(define (augend expr)
  (car expr))
```

- Predicates

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

6.001 SICP

32/31

Modularity helps in other ways

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

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

(simplify (deriv '(+ x y) 'x))
```

6.001 SICP

33/31

Separating out aspects of simplification

```
(define (simplify-sum expr)
  (cond ((and (number? (addend expr)) (number? (augend expr)))
        (+ (addend expr) (augend expr))) (+ 2 3) → 5
        ((or (number? (addend expr)) (number? (augend expr)))
         expr) (+ 2 x) → (+ 2 x)
        ((eq? (addend expr) (augend expr))
         (make-product 2 (addend expr))) (+ x x) → (* 2 x)
        ((product-expr? (augend expr))
         (if (and (number? (multiplier (augend expr)))
                  (eq? (addend expr) (+ x (* 3 x)) → (* 4 x)
                       (multiplicand (augend expr))))
             (make-product (+ 1 (multiplier (augend expr)))
                             (augend expr))
             (make-product
              (simplify (multiplier (augend expr)))
              (simplify (multiplicand (augend expr))))))
        (else expr)))
```

6.001 SICP

34/31