Slide 1

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>
```

Slide 2

Symbols?

- Say your favorite color
- Say “your favorite color”
- What is the difference?
  - In one case, we want the meaning associated with the expression
  - In the other case, we want the actual words (or symbols) of the expression

Slide 3

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?

???

Slide 4

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
```

Slide 5

Symbol: a primitive type

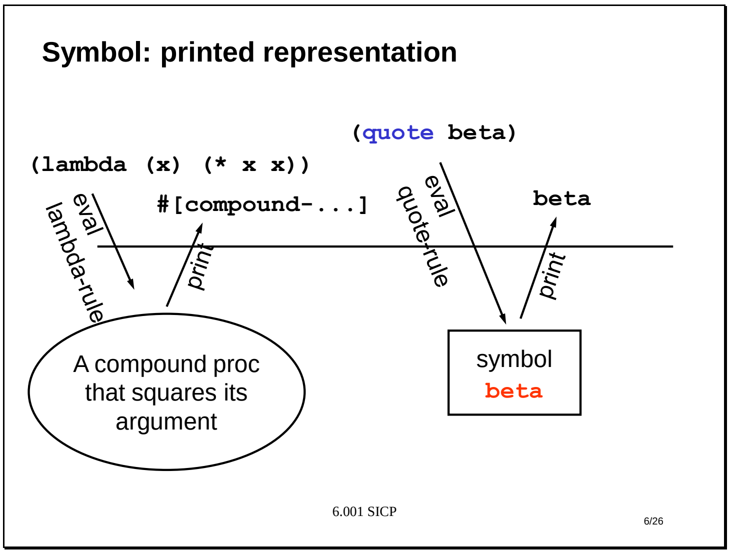
- constructors:
  None since really a primitive not an object with parts

- selectors
  None

- operations:

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

eq?          ; discuss in a minute
```

Slide 6



Symbol: printed representation
(quote beta)
(lambda (x) (* x x))
eval
lambda-rule
#[compound-...]
print
eval
quote-rule
beta
print
A compound proc that squares its argument
symbol
beta
6.001 SICP
6/26

___

___

___

___

___

___

___

___

___

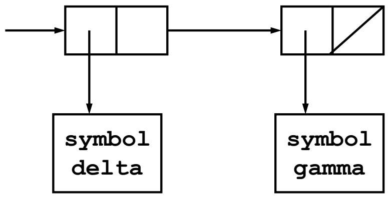
___

___

Slide 7

Symbols are ordinary values

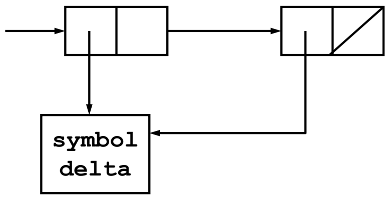
```
(list 1 2)              ==> (1 2)

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



Slide 8

A useful property of the quote special form

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



Two quote expressions with the same name return the same object

Slide 9

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)) ==>
```

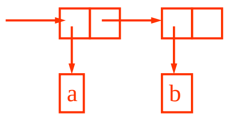
- For those who are interested:

```
; eq?:  EQtype, EQtype ==> boolean
; EQtype = any type except number or string
```

- One should therefore use = for equality of numbers, not eq?

Slide 10

Generalization: quoting other expressions

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

In general, **(quote DATUM)** is converted to **DATUM**

Slide 11

Shorthand: the single quote mark

`'a`	is shorthand for	`(quote a)`
`'(1 2)`		`(quote (1 2))`

Slide 12

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)            ==>
```

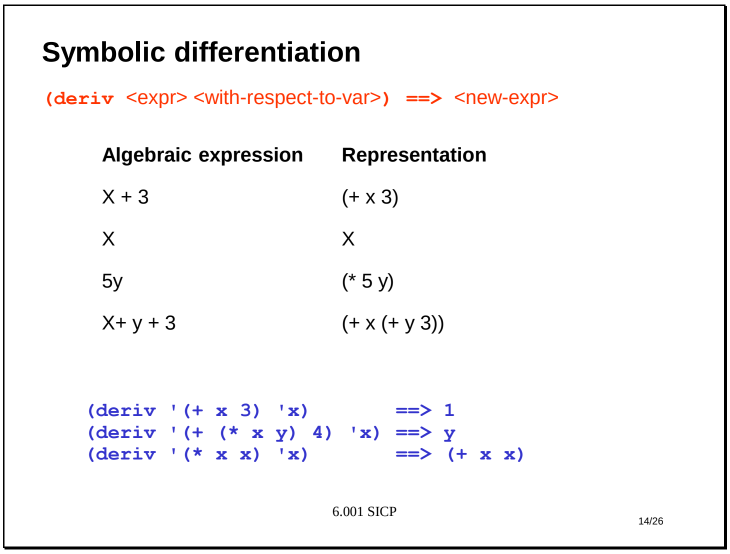
Slide 13

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)

Slide 14



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
14/26

Slide 15

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

Slide 16

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

Slide 17

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> >>
```

Slide 18

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))))
```

Slide 19

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>
      )
  )))
```

Slide 21

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!)

Slide 22

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

____

____

____

____

____

____

____

____

____

____

____

Slide 23

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))))
```

Slide 24

Use data abstractions

- To eliminate dependence on the representation:

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

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

Slide 25

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))
  ))
```

Slide 26

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
```