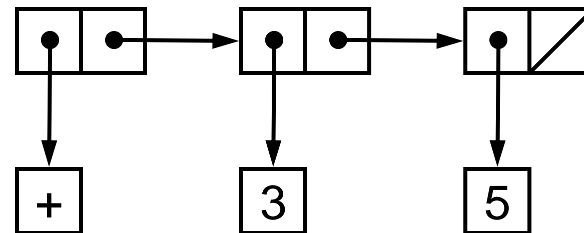
CS1 Recitation 6

Fall 2008

quote

- `quote` inhibits the evaluation of its operand
  - `(quote (+ 3 5))`
    → `(+ 3 5)`
- Abbreviate `quote` with `'`
  - `'(+ 3 5)` → `(+ 3 5)`



- Nested `quote`?
  - `(quote (quote 5))   ;; or ''5`
    → `(quote 5)          ;; or '5`
- Outermost quote also inhibits evaluation of inner quotes
  - `'(* x x (+ 3 x 'y))`
    → `(* x x (+ 3 x 'y))`

`quote` and Lambda Evaluation

- `quote` also interferes with substitution in lambda expressions!
  - "Quote inhibits the evaluation of its operand"
  - No substitution occurs for quoted variables
  - `((lambda (x) (list 'x x)) 3)`
    → `(x 3)`
- From homework:

```
(let ((x 2))
  (quote x))
```

  - Hint: `let` desugars into a `lambda` expression

Tagged Data

- Tag values with a symbol
  - e.g. indicate the units of a value

```
(meter . 35)
```

- Use cons-pairs for storing symbol and value

```
(define (make-tagval tag value) (cons tag value))
```

  - Assignment’s code doesn’t include this abstraction…
- Use abstractions to extract tag and value

```
(define (get-tag   m) (car m))
(define (get-value m) (cdr m))
```

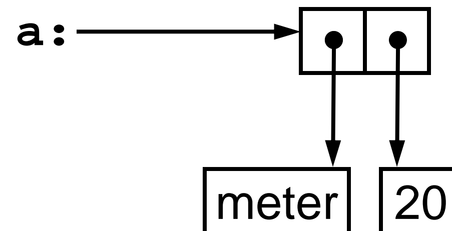
Lengths

- Make meters:

```
(define (make-meter length)
  (cons 'meter length))
```

- Can create meters now:

```
(define a (make-meter 20))
```



- Is it a meter?

```
(define (meter? m)
  (and (pair? m)
       (eq? (get-tag m) 'meter)))
```

  - **`pair?`** returns **`#t`** if the argument is a cons-pair, **`#f`** otherwise
- Do same thing for feet…

Adding Lengths

- Define addition for each unit type

```
(define (meter-add a b)
  (make-meter (+ (get-value a) (get-value b))))
(define (foot-add a b)
  (make-foot (+ (get-value a) (get-value b))))
```

- Note: Could make this code more bulletproof…

```
(define (meter-add a b)
  (if (and (meter? a) (meter? b))
      (make-meter (+ (get-value a) (get-value b)))
      (error "both arguments must be meters")))
```

  - Best to check for these kinds of issues and report an error
  - Second version is <u>fail-fast</u>: it reports error situations right away

Type Conversions

- Build in support to convert between types

```
(define meters-per-foot 0.3048)
(define (feet-to-meters a)
  (make-meter
    (* meters-per-foot (get-value a))))
```

  - Note **`meters-per-foot`** constant
- Avoid having <u>magic numbers</u> in your programs
  - Hard-coded constants whose meaning is not obvious from the context
  - The same number repeated all over the place
  - Giving it a name makes for easier maintenance

Type Conversions (2)

- Convert the other direction too

```
(define feet-per-meter (/ 1 meters-per-foot))
(define (meters-to-feet a)
  (make-foot
    (* feet-per-meter (get-value a))))
```

  - Should error-check here too…

Generic `length-add` Function

- Now it's easy to write a **`length-add`** function

```
(define (length-add a b)
  (cond ((and (meter? a) (meter? b))
         (meter-add a b))
        ((and (foot? a) (foot? b))
         (foot-add a b))
        ((and (foot? a) (meter? b))
         (foot-add a (meters-to-feet b)))
        ((and (meter? a) (foot? b))
         (meter-add  a (feet-to-meters b)))
        (else (error "length-add given bad units"))))
```

Message-Passing Implementations

- When you hear "…message passing implementation of foo"…
- Think "make-foo creates a lambda expression that handles the different operation tags."

```
(define (make-foo x y z)  ;; initial values for new foo
  (lambda (op . args)
    (cond ((eq? op 'mesg1) ...)    ;; handle mesg1
          ((eq? op 'mesg2) ...)    ;; handle mesg2
          ...
          (else (error "unrecognized operation" op)))))
```

Message-Passing Impls. (2)

- What is the contract of **`make-foo`** ?

```
(define (make-foo x y z)  ;; initial values for new foo
  (lambda (op . args)
    (cond ((eq? op 'mesg1) ...)    ;; handle mesg1
          ((eq? op 'mesg2) ...)    ;; handle mesg2
          ...
          (else (error "unrecognized operation" op)))))
```

- **`make-foo`** technically returns a procedure, but think of it as a new data-type

```
;; make-foo: number number number -> foo
(define (make-foo x y z) ...
```

Using Message-Passing Objects

- Create new "objects"…

```
(define f1 (make-foo 3 5.6 2.4))
(define f2 (make-foo 0.4 1.9 -5))
```

  - **`f1`** and **`f2`** are actually procedures that can handle messages
- Pass messages to the objects

```
(f1 'print)
(define f3 (f1 'combine f2))
```

- Message consists of operation tag, and any necessary arguments
  - Dotted-tail notation: **`(lambda (op . args) ... )`**
  - Operation is required, arguments are optional.

Message-Passing Meters

- Implement a message-passing version of meter
  - **`'get-meter`** returns length in meters
  - **`'unit-type`** returns 'meter
  - **`'add`** adds two meters, creates a new meter
- Form of implementation:

```
(define (make-meter x)
  (lambda (op . args)
    (cond ((eq? op 'get-meter) ... )
          ((eq? op 'unit-type) ... )
          ((eq? op 'add) ... )
          (else (error "unknown operation" op)))))
```

Message-Passing Meters (2)

```
(define (make-meter x)
  (lambda (op . args)
    (cond ((eq? op 'get-meter) x)
          ((eq? op 'unit-type) 'meter)
          ((eq? op 'add)
           (if (meter? (car args))
               (make-meter
                 (+ x ((car args) 'get-meter)))
               (error "not a meter!")))
          (else (error "unknown operation" op)))))
```

- Need new definition of **meter?** now…

```
(define (meter? m)
  (eq? (m 'unit-type) 'meter))
```

Message-Passing Meters (3)

- Repeated `(car args)` everywhere can get ugly…
- Can use `let`-expression to reduce the mess

```
(define (make-meter x)
  (lambda (op . args)
    (cond ((eq? op 'get-meter) x)
          ((eq? op 'unit-type) 'meter)
          ((eq? op 'add)
           (let ((m (car args)))   ;; Pull out arg 1
             (if (meter? m)
                 (make-meter (+ x (m 'get-meter)))
                 (error "not a meter!"))))
          (else (error "unknown operation" op)))))
```

Initialization Values

- Initial values are "stored" in procedure body
  - *…somehow…*
- Operation:

```
(define m1 (make-meter 10))
```

- Definition:

```
(define (make-meter x)
  (lambda (op . args)
    (cond ((eq? op 'get-meter) x)
          ... )))
```

  - How is x = 10 remembered?

`(m1 'get-meter)` → 10

Initialization Values (2)

- Desugared **`make-meter`**:

```
(define make-meter
  (lambda (x)
    (lambda (op . args)
      (cond ((eq? op 'get-meter) x)
            ... ))))
```

- "Apply **`make-meter`** to 10…"
  - Substitute x = 10 into body of **`(lambda (x) ... )`**
  - Result of substitution:

```
(lambda (op . args)
  (cond ((eq? op 'get-meter) 10)
        ... )))
```

  - A good enough answer, for now. ☺

Symbolic Differentiator

- Parts B and C involve a symbolic differentiator
  - Good news: most of the code is provided for you
  - Bad news: you will have to figure the code out
- Part B:
  - Differentiator handles Scheme-like math expressions
    - …but only expressions involving addition and multiplication.
    - `(deriv '(* x y) 'x)` → `y`
  - In base impl, + and * can only take two arguments
    - e.g. this won't work: `(deriv '(* 3 x y) 'x)`
  - Need to extend existing code in various ways

Differentiator: The Details

- A significant source of complexity is simplification
  - Want to simplify expressions like **`(+ x 0)`** to just **`x`**
- Without simplification:

```
(define (make-sum a1 a2) (list '+ a1 a2))
```

- With simplification:

```
(define (make-sum a1 a2)
  (cond ((=number? a1 0) a2)
        ((=number? a2 0) a1)
        ((and (number? a1) (number? a2)) (+ a1 a2))
        (else (list '+ a1 a2))))
```

  - All extra code is for simplifying expressions

New Predicates

- Differentiator code includes some new predicates
- For variables in an expression, e.g. `'x` in `'(* x 3)`

  ```
  (define (variable? x) (symbol? x))
  ```

  - `symbol?` is a built-in function
    - Returns `#t` if argument is a single symbol
    - `(symbol? 'x)` → `#t`
    - `(symbol? '(x))` → `#f`
  - `number?` is also a built-in function
    - Returns `#t` if argument is a number
    - `(number? 5)` → `#t`
    - `(number? '5)` → `#t`

Extending Existing Code

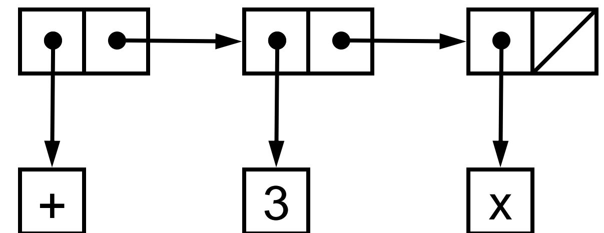
- When extending an existing program:
- Step 1:
  - Understand what the program currently does!
  - (This is probably the majority of time on this lab.)
- Step 2:
  - Find a way to “fit” the new functionality into the way the program currently works.
  - See if you can make the extended functionality “look like” what the program already does.

Part B, Exercise 2.56

- Extend differentiator to handle exponentiation
  `(deriv '(** x 3) 'x)` → `(* 3 (** x 2))`
- Differentiator has set of functions for each expr. type
  - Sums: **`make-sum`**, **`sum?`**, **`addend`**, **`augend`**
  - Products: **`make-product`**, **`product?`**, **`multiplier`**, **`multiplicand`**
- Implement a new set of functions for exponentiation
  - **`make-exponentiation`**, **`exponentiation?`**, **`base`**, **`exponent`**
  - …then update **`deriv`** function to differentiate exponents
  - Use existing implementations for guidance!

Part B, Exercise 2.57

- Extend existing differentiator to handle arbitrary number of arguments to + and *
  - Shouldn't change implementation of **`deriv`** itself
- Change way that sums and products themselves are represented
  - For sums, tinker with **`make-sum`**, **`sum?`**, **`addend`**, **`augend`** functions
  - Same with products…
- Differentiator depends on structure of expressions like **`'(+ 3 x)`**
  - Think about how the existing code changes when moving to multiple args



Interfaces!

- Different expression types must provide the same set of operations
  - "Make a new expression of a given type."
  - "Does an expression have a particular type?"
  - "Break an expression down into its component parts."
- Idea of Part C:
  - Combine expressions and their associated operations into message-passing objects
  - All expression objects will have the same interface; i.e. they will handle the same set of messages

Final Debugging Hint

- When working on a large problem, don't test it only at the main entry-point!
  - e.g. for Part B, don't just try your code out by invoking **`deriv`** !
  - Try lower-level functions on simple expressions to make sure they work properly, in isolation
- Debugging a program is largely about identifying where the problem is happening
  - Want to eliminate pieces of the program as quickly as possible
  - Can't do this when you only run the whole thing…