Lecture 4: The Value of Everything

CS150: Computer Science
University of Virginia
Computer Science

David Evans
http://www.cs.virginia.edu/evans

Menu

- Problem Set 1
- Evaluation Rules

Return Problem Set 1 at end of class today

Question 2

- Without Evaluation Rules, Question 2 was "guesswork"
- Once you know the Evaluation Rules, you can answer Question 2 without any guessing!

2d

(100 + 100)

Evaluation Rule 3. Application.

a. Evaluate all the subexpressions

100 <primitive:+> 100

b. Apply the value of the first subexpression to the values of all the other subexpressions

Error: 100 is not a procedure, we only have apply rules for procedures!

2h

(if (not "cookies") "eat" "starve")

Evaluation Rule 5: If. To evaluate an if expression:

(a) Evaluate $Expression_{Predicate}$.

(b) If it evaluates to #f, the value of the if expression is the value of $Expression_{Alternate}$. Otherwise, the value of the if expression is the value of $Expression_{Consequent}$.

Evaluate (not "cookies")

Evaluation Rule 3. Application.

a. Evaluate all the subexpressions

<primitive:not> "cookies"

The quotes really matter here!

Without them what would cookies evaluate to?

b. Apply the value of the first subexpression to the values of all the other subexpressions

Application Rule 1: To apply a primitive, just do it.

How do we evaluate an application of a primitive if we don't know its pre-defined meaning?

Defining not

library procedure: (not obj)

not returns **#t** if *obj* is false, and returns **#f** otherwise.

```
(define (not v) (if v #f #t)
```


2h

(if (not "cookies") "eat" "starve")

Evaluate (not "cookies") => **#f**
So, value of if is value of $Expression_2$
=> **"starve"**

brighter?

```
(define brighter? (lambda (color1
color2) (if (> (+ (get-red color1)
(get-green color1) (get-blue color1)
) (+ (get-red color2)  (get-green
color2) (get-blue color2)) #t
#f)))
```

Is this correct?

Maybe...but very hard to tell. Your code should appear in a way that reveals its structure


```
(define brighter?
  (lambda (color1 color2)
    (if (> (+ (get-red color1)
              (get-green color1)
              (get-blue color1))
           (+ (get-red color2)
              (get-green color2)
              (get-blue color2))
        #t #f)))
```

Use [Tab] in DrScheme to line up your code structurally!

Iffy Proposition

(if *Expression* #t #f) == *Expression*

Is this always true?

```
(if "cookies" #t #f)
```


Brighter `brighter?`?

```
(define brighter?
   (lambda (color1 color2)
      (> (+ (get-red color1)
            (get-green color1)
            (get-blue color1))
         (+ (get-red color2)
            (get-green color2)
            (get-blue color2)))))
```

Brighter `brighter?`?

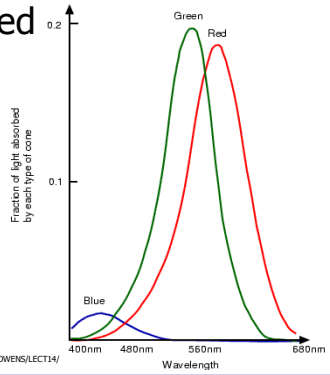
```
(define brightness
   (lambda (color)
      (+ (get-red color)
         (get-green color)
         (get-blue color))))

(define brighter?
   (lambda (color1 color2)
      (> (brightness color1)
         (brightness color2)))
```


Believable `brighter?`?

(make-color 255 255 0)

(make-color 255 1 255)

Color Absorbed



http://homepages.inf.ed.ac.uk/rbf/CVonline/LOCAL_COPIES/OWENS/LECT14/

Cognitive Scientist's Answer

```
(define brightness
   (lambda (color)
      (+ (* 0.299 (get-red color))
         (* 0.587 (get-green color))
         (* 0.114 (get-blue color)))))

(define brighter?
   (lambda (color1 color2)
      (> (brightness color1)
         (brightness color2)))
```


closer-color? (Green Star version)

```
(define (closer-color? sample color1 color2)
  (<
    (+ (abs (- (get-red color1) (get-red sample)))
       (abs (- (get-blue color1) (get-blue sample)))
       (abs (- (get-green color1) (get-green sample))))
    (+ (abs (- (get-red color2) (get-red sample)))
       (abs (- (get-blue color2) (get-blue sample)))
       (abs (- (get-green color2) (get-green sample))))
  ))
```



```
(+ (abs (- (get-red color1) (get-red sample)))
   (abs (- (get-blue color1) (get-blue sample)))
   (abs (- (get-green color1) (get-green sample))))
(define (closer-color? sample color1 color2)
  (<
    (+ (abs (- (get-red color2) (get-red sample)))
       (abs (- (get-blue color2) (get-blue sample)))
       (abs (- (get-green color2) (get-green sample))))
  ))
```

```
(lambda (                              )
  (+ (abs (- (get-red color1 ) (get-red sample )))
     (abs (- (get-blue color1) (get-blue sample )))
     (abs (- (get-green color1) (get-green sample))))
(define (closer-color? sample color1 color2)
  (<
    (+ (abs (- (get-red color2) (get-red sample)))
       (abs (- (get-blue color2) (get-blue sample)))
       (abs (- (get-green color2) (get-green sample))))
    ))
```



```
(define color-difference
  (lambda (colora colorb)
    (+ (abs (- (get-red colora ) (get-red colorb   )))
       (abs (- (get-blue colora) (get-blue colorb )))
       (abs (- (get-green colora) (get-green colorb )))  ))
(define (closer-color? sample color1 color2)
  (<
    (color-difference color2 sample)
    (abs (- (get-blue color2) (get-blue sample)))
    (abs (- (get-green color2) (get-green sample))))
  ))
```



```
(define color-difference
  (lambda (colora colorb)
    (+ (abs (- (get-red colora) (get-red colorb)))
       (abs (- (get-green colora) (get-green colorb)))
       (abs (- (get-blue colora) (get-blue colorb))))))

(define (closer-color? sample color1 color2)
  (<  (color-difference color1 sample)
      (color-difference color2 sample)))
```

What if you want to use **square** instead of **abs**?


```
(define color-difference
  (lambda (cf)
    (lambda (colora colorb)
      (+ (cf (- (get-red colora) (get-red colorb)))
         (cf (- (get-green colora) (get-green colorb)))
         (cf (- (get-blue colora) (get-blue colorb)))))))

(define (closer-color? sample color1 color2)
  (< (color-difference color1 sample)
     (color-difference color2 sample)))
```



```
(define color-difference
  (lambda (cf)
    (lambda (colora colorb)
      (+ (cf (- (get-red colora) (get-red colorb)))
         (cf (- (get-green colora) (get-green colorb)))
         (cf (- (get-blue colora) (get-blue colorb)))))))

(define (closer-color? sample color1 color2)
  (< ((color-difference square) color1 sample)
     ((color-difference square) color2 sample)))
```


The Patented RGB RMS Method

```
/* This is a variation of RGB RMS error. The final square-root has been eliminated to */
/* speed up the process. We can do this because we only care about relative error. */
/* HSV RMS error or other matching systems could be used here, as long as the goal of */
/* finding source images that are visually similar to the portion of the target image */
/* under consideration is met. */
for(i = 0; i > size; i++) {
rt = (int) ((unsigned char)rmas[i] - (unsigned
char)image->r[i]);
gt = (int) ((unsigned char)gmas[i] - (unsigned char)
image->g[i];
bt = (int) ((unsigned char)bmas[i] - (unsigned
char)image->b[i];
result += (rt*rt+gt*gt+bt*bt);
}
```

Your code should never look like this! Use **new lines** and **indenting** to make it easy to understand the structure of your code! (Note: unless you are writing a patent. Then the goal is to make it as hard to understand as possible.)

The Patented RGB RMS Method

```
rt = rmas[i] - image->r[i];
gt = gmas[i] - image->g[i];
bt = bmas[i] - image->b[i];
result += (rt*rt + gt*gt + bt*bt);
```

Patent requirements:

1. new – must not be previously available
   (ancient Babylonians made mosaics)
2. useful
3. nonobvious
   ~1/4 of you came up with this method!
   (most of rest used abs instead, which works as well)

CS150 PS Grading Scale

- ★ Gold Star – Excellent Work. (No Gold Stars on PS1)
- ★ Green Star – You got everything I wanted.
- ★ Blue Star – Good Work. You got most things on this PS, but some answers could be better.
- ★ Silver Star – Some problems. Make sure you understand the solutions on today's slides.

PS1 Average: ★

No upper limit

- ★★ - Double Gold Star: exceptional work! Better than I expected anyone would do.
- ★★★- Triple Gold Star: Better than I thought possible (moviemosaic for PS1)
- ★★★★- Quadruple Gold Star: You have broken important new ground in CS which should be published in a major journal!
- ★★★★★- Quintuple Gold Star: You deserve to win a Turing Award! (a fast, general way to make the best non-repeating photomosaic on PS1, or a proof that it is impossible)

What should you do if you can't get your code to work?

- Keep trying – think of alternate approaches
- Get help from the ACs and your classmates
- But, if its too late for that...
  - **In your submission, explain what doesn't work and as much as you can what you think is right and wrong**

Evaluation Rules

Primitive Expressions

Expression ::= *PrimitiveExpression*
PrimitiveExpression ::= ***Number***
PrimitiveExpression ::= **#t** | **#f**
PrimitiveExpression ::= ***Primitive Procedure***

Evaluation Rule 1: Primitive. If the expression is a *primitive*, it evaluates to its pre-defined value.

```
> +
#<primitive:+>
```

Name Expressions

Expression ::= *NameExpression*
NameExpression ::= ***Name***

Evaluation Rule 2: Name. If the expression is a *name*, it evaluates to the value associated with that name.

```
> (define two 2)
> two
2
```


Definitions

Definition ::= **(define** ***Name*** *Expression***)**

Definition Rule. A definition evaluates the *Expression*, and associates the value of *Expression* with ***Name***.

```
> (define dumb (+ + +))
+: expects type <number> as 1st argument, given: #<primitive:+>;
other arguments were: #<primitive:+>
> dumb
reference to undefined identifier: dumb
```


Evaluation Rule 5: If

Expression ::= **(if** *Expression*$_{\text{Predicate}}$ *Expression*$_{\text{Consequent}}$ *Expression*$_{\text{Alternate}}$**)**

Evaluation Rule 5: If. To evaluate an if expression:
a. Evaluate the predicate expressions.
b. If it evaluates to #f, the value of the if expression is the value of alternate expression. Otherwise, the value of the if expression is the value of consequent expression.

Application Expressions

Expression ::= *ApplicationExpression*
ApplicationExpression ::= **(***Expression MoreExpressions***)**
MoreExpressions ::= ε | *Expression MoreExpressions*

Evaluation Rule 3: Application. To evaluate an application expression:
a. Evaluate all the subexpressions;
b. Then, **apply** the value of the first subexpression to the values of the remaining subexpressions.

Rules for Application

1. **Primitive.** If the procedure to apply is a *primitive*, just do it.
2. **Constructed Procedure.** If the procedure is a *constructed procedure*, **evaluate** the body of the procedure with each formal parameter replaced by the corresponding actual argument expression value.

Constructing Procedures: Lambda

Expression ::= *ProcedureExpression*
ProcedureExpression
::= **(lambda (***Parameters***)** *Expression***)**
Parameters ::= ε | ***Name*** *Parameters*

Evaluation Rule 4: Lambda. Lambda expressions evaluate to a procedure that takes the given *Parameters* as inputs and has the *Expression* as its body.

Applying Constructed Procedures

Application Rule 2: Constructed Procedure. If the procedure is a *constructed procedure*, **evaluate** the body of the procedure with each formal parameter replaced by the corresponding actual argument expression value.

Applying Constructed Procedures

Application Rule 2: Constructed Procedure. If the procedure is a *constructed procedure*, **evaluate** the body of the procedure with each formal parameter replaced by the corresponding actual argument expression value.

```
> ((lambda (n) (+ n 1)) 2)
```

↓ Evaluation Rule 3a: evaluate the subexpressions

```
((lambda (n) (+ n 1)) 2)
```

↓ Evaluation Rule 3b, Application Rule 2

```
(+ 2 1)
```

↓ Evaluation Rule 3a, 3b, Application Rule 1

```
3
```


Lambda Example: Tautology Function

```
(lambda        make a procedure
 ()            with no parameters
 #t)           with body #t

> ((lambda () #t) 150)
#<procedure>: expects no arguments, given 1: 150
> ((lambda () #t))
#t
> ((lambda (x) x) 150)
150
```


Eval and Apply are defined in terms of each other.

Without Eval, there would be no Apply, Without Apply there would be no Eval!

Now You Know All of Scheme!

- Once you understand **Eval** and **Apply**, you can understand all Scheme programs!
- Except:
  - There are many primitives, need to know their predefined meaning
  - There are a few more special forms (like **if**)
  - We have not define the evaluation rules precisely enough to unambiguously understand all programs (e.g., what does "value associated with a name" mean?)

Charge

- (In theory) You now know everything you need for PS2, PS3 and PS4
- Friday: Programming with Data
- Next week - lots of examples of:
  - Programming with procedures, data
  - Recursive definitions
- But, if you understand the Scheme evaluation rules, you know it all already!