Scheme Notes 02

Geoffrey Matthews

Department of Computer Science
Western Washington University

January 5, 2017

Mantra Review

- Every expression has a value (exceptions: errors, infinite loops and define)
- To find the value of a combination,
  - Find the values of all of the subexpressions, in any order
  - Apply the value of the first to the values of the rest
- The value of a lambda expression is a procedure

Values of combinations

- To find the value of a combination,
  - Find the values of all of the subexpressions, in any order
  - Apply the value of the first to the values of the rest

```
(+ 2 3)
```

Values of combinations

- To find the value of a combination,
  - Find the values of all of the subexpressions, in any order
  - Apply the value of the first to the values of the rest

```
(+ 2 3)
(+ 2 3)
```

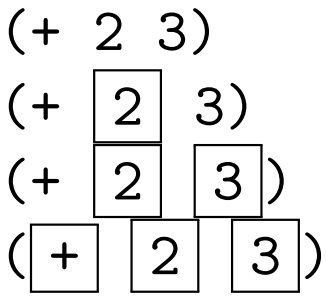
Values of combinations

- To find the value of a combination,
  - Find the values of all of the subexpressions, in any order
  - Apply the value of the first to the values of the rest

```
(+ 2 3)
(+ 2 3)
(+ 2 3)
```

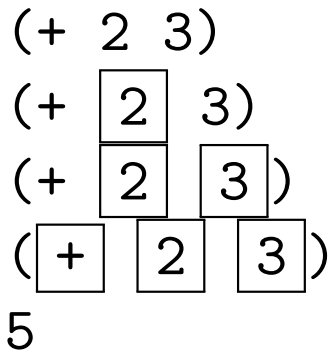
Values of combinations

- To find the value of a combination,
  - Find the values of all of the subexpressions, in any order
  - Apply the value of the first to the values of the rest



Values of combinations

- To find the value of a combination,
  - Find the values of all of the subexpressions, in any order
  - Apply the value of the first to the values of the rest



Values of combinations

- To find the value of a combination,
  - Find the values of all of the subexpressions, in any order
  - Apply the value of the first to the values of the rest

```
(+ 3 (- 8 2) (+ 2 3))
```

Values of combinations

- To find the value of a combination,
  - Find the values of all of the subexpressions, in any order
  - Apply the value of the first to the values of the rest

```
(+ 3 (- 8 2) (+ 2 3))
(+ 3 (- 8 2) (+ 2 3))
```

Values of combinations

- To find the value of a combination,
  - Find the values of all of the subexpressions, in any order
  - Apply the value of the first to the values of the rest

```
(+ 3 (- 8 2) (+ 2 3))
(+ [3] (- 8 2) (+ 2 3))
(+ [3] (- 8 2) [(+ 2 3)])
```

Values of combinations

- To find the value of a combination,
  - Find the values of all of the subexpressions, in any order
  - Apply the value of the first to the values of the rest

```
(+ 3 (- 8 2) (+ 2 3))
(+ [3] (- 8 2) (+ 2 3))
(+ [3] (- 8 2) [(+ 2 3)])
(+ [3] (- 8 2) [5])
```

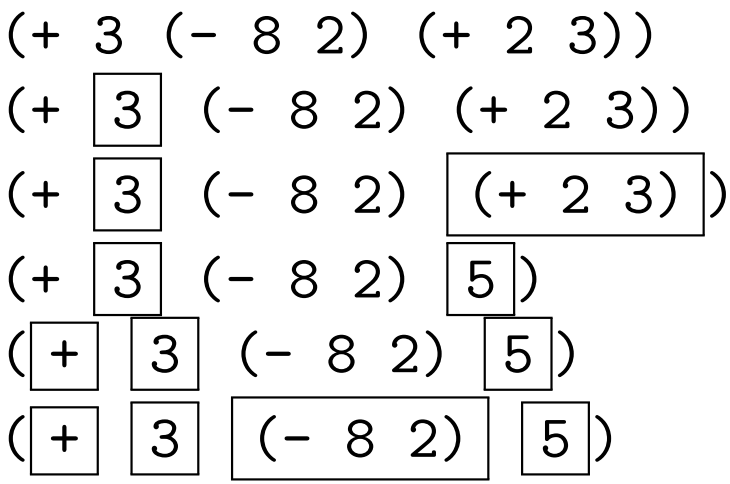
Values of combinations

- To find the value of a combination,
  - Find the values of all of the subexpressions, in any order
  - Apply the value of the first to the values of the rest

```
(+ 3 (- 8 2) (+ 2 3))
(+ [3] (- 8 2) (+ 2 3))
(+ [3] (- 8 2) [(+ 2 3)])
(+ [3] (- 8 2) [5])
([+] [3] (- 8 2) [5])
```

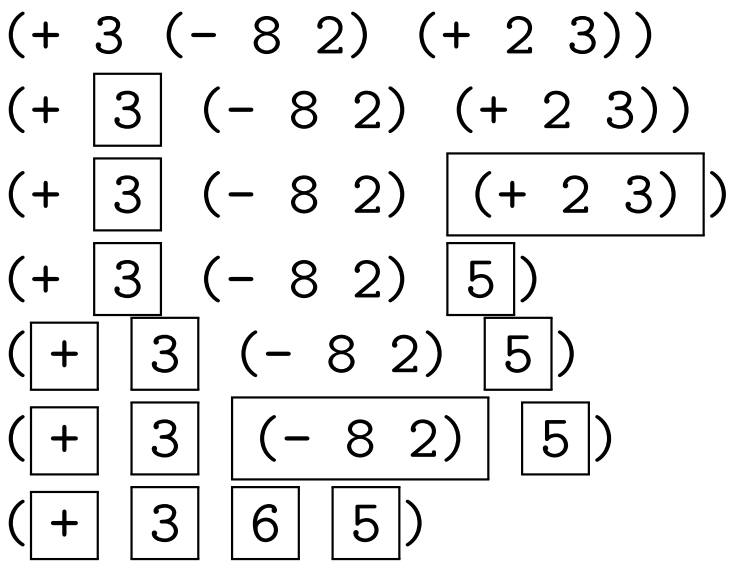
Values of combinations

- To find the value of a combination,
  - Find the values of all of the subexpressions, in any order
  - Apply the value of the first to the values of the rest



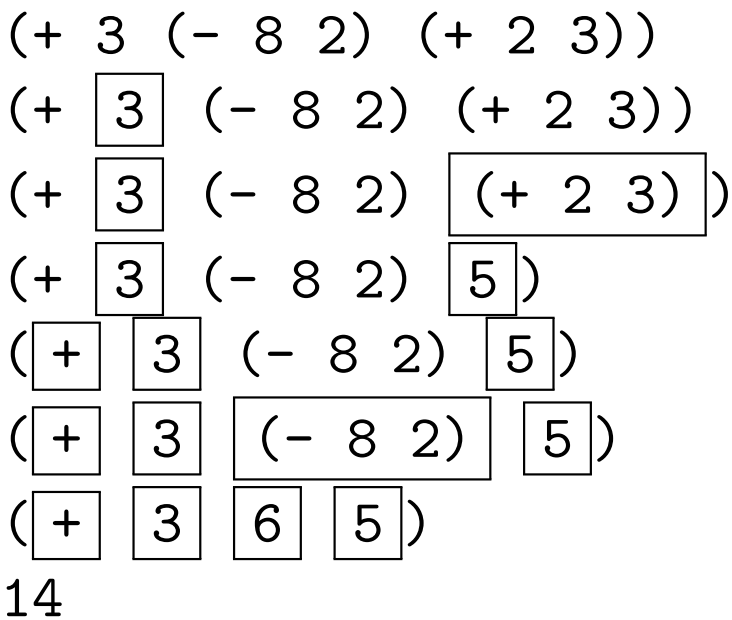
Values of combinations

- To find the value of a combination,
  - Find the values of all of the subexpressions, in any order
  - Apply the value of the first to the values of the rest



Values of combinations

- To find the value of a combination,
  - Find the values of all of the subexpressions, in any order
  - Apply the value of the first to the values of the rest



Compound procedures

- `(lambda (x) (* x x))`
- What is/are the parameters?

Compound procedures

- `(lambda (x) (* x x))`
- What is/are the parameters?
- `x` is the only parameter in the parameter list `(x)`

Compound procedures

- `(lambda (x) (* x x))`
- What is/are the parameters?
- `x` is the only parameter in the parameter list `(x)`
- What is the body?

Compound procedures

- `(lambda (x) (* x x))`
- What is/are the parameters?
- `x` is the only parameter in the parameter list `(x)`
- What is the body?
- `(* x x)` is the body of the lambda expression.

Substitution Model

To apply a compound procedure to arguments, evaluate the body of the procedure with each formal parameter replaced by the corresponding argument.

```
(define absolute-value (lambda (n) (if (< n 0) (- n) n)

(absolute-value (+ 3 -8))
```

Substitution Model

To apply a compound procedure to arguments, evaluate the body of the procedure with each formal parameter replaced by the corresponding argument.

```
(define absolute-value (lambda (n) (if (< n 0) (- n) n)

(absolute-value (+ 3 -8))
(absolute-value (+ 3 -8))
```

Substitution Model

To apply a compound procedure to arguments, evaluate the body of the procedure with each formal parameter replaced by the corresponding argument.

```
(define absolute-value (lambda (n) (if (< n 0) (- n) n)

(absolute-value (+ 3 -8))
(absolute-value (+ 3 -8))
(absolute-value -5)
```

Substitution Model

To apply a compound procedure to arguments, evaluate the body of the procedure with each formal parameter replaced by the corresponding argument.

```
(define absolute-value (lambda (n) (if (< n 0) (- n) n)

(absolute-value (+ 3 -8))
(absolute-value (+ 3 -8) )
(absolute-value -5 )
(absolute-value -5 )
```

Substitution Model

To apply a compound procedure to arguments, evaluate the body of the procedure with each formal parameter replaced by the corresponding argument.

```
(define absolute-value (lambda (n) (if (< n 0) (- n) n)

(absolute-value (+ 3 -8))
(absolute-value (+ 3 -8) )
(absolute-value -5 )
( absolute-value  -5 )
( (lambda (n) (if (< n 0) (- n) n))  -5 )
```

Substitution Model

To apply a compound procedure to arguments, evaluate the body of the procedure with each formal parameter replaced by the corresponding argument.

```
(define absolute-value (lambda (n) (if (< n 0) (- n) n)

(absolute-value (+ 3 -8))
(absolute-value (+ 3 -8) )
(absolute-value -5 )
( absolute-value  -5 )
( (lambda (n) (if (< n 0) (- n) n))  -5 )
(if (< -5 0) (- -5 ) -5 ))
```

Substitution Model

To apply a compound procedure to arguments, evaluate the body of the procedure with each formal parameter replaced by the corresponding argument.

```
(define absolute-value (lambda (n) (if (< n 0) (- n) n)

(absolute-value (+ 3 -8))
(absolute-value (+ 3 -8) )
(absolute-value -5 )
( absolute-value  -5 )
( (lambda (n) (if (< n 0) (- n) n))  -5 )
(if (< -5 0) (- -5 ) -5 ))
(if  (< -5 0)  (- -5 ) -5 ))
```

Substitution Model

To apply a compound procedure to arguments, evaluate the body of the procedure with each formal parameter replaced by the corresponding argument.

```
(define absolute-value (lambda (n) (if (< n 0) (- n) n)

(absolute-value (+ 3 -8))
(absolute-value (+ 3 -8) )
(absolute-value -5 )
( absolute-value  -5 )
( (lambda (n) (if (< n 0) (- n) n))  -5 )
(if (< -5 0) (- -5 ) -5 ))
(if  (< -5 0)  (- -5 ) -5 ))
(if  #t  (- -5 ) -5 ))
```

Substitution Model

To apply a compound procedure to arguments, evaluate the body of the procedure with each formal parameter replaced by the corresponding argument.

```
(define absolute-value (lambda (n) (if (< n 0) (- n) n)

(absolute-value (+ 3 -8))
(absolute-value (+ 3 -8))
(absolute-value -5)
(absolute-value -5)
((lambda (n) (if (< n 0) (- n) n)) -5)
(if (< -5 0) (- -5) -5))
(if (< -5 0) (- -5) -5))
(if #t (- -5) -5))
(- -5)
```

Substitution Model

To apply a compound procedure to arguments, evaluate the body of the procedure with each formal parameter replaced by the corresponding argument.

```
(define absolute-value (lambda (n) (if (< n 0) (- n) n)

(absolute-value (+ 3 -8))
(absolute-value (+ 3 -8) )
(absolute-value -5 )
( absolute-value  -5 )
( (lambda (n) (if (< n 0) (- n) n))  -5 )
(if (< -5 0) (- -5 ) -5 ))
(if  (< -5 0)  (- -5 ) -5 ))
(if  #t  (- -5 ) -5 ))
(- -5 )
5
```

Computing the Euclidean distance between two points

```
(define square
   (lambda (x) (* x x)))

(define sum-squares
   (lambda (x y) (+ (square x) (square y))))

(define dist-between-pts
   (lambda (x1 y1 x2 y2)
      (sqrt (sum-squares (- x1 x2) (- y1 y2)))))
```

Use the substitution model to evaluate (dist-between-pts 1 1 4 5):

Applicative and Normal Order

- Applicative Order: Evaluate the arguments and then apply the value of the first to the value of the rest.
  - Everything is evaluated, whether or not we use it.
  - This is the method Scheme uses and is the reason that we need special forms.
- Normal Order: Fully expand, then reduce. Don't evaluate operands until they are needed.

Applicative and normal order

```
(define square
   (lambda (x) (* x x)))

(define sum-squares
   (lambda (x y) (+ (square x) (square y))))

(define dist-between-pts
   (lambda (x1 y1 x2 y2)
      (sqrt (sum-squares (- x1 x2) (- y1 y2)))))

(sum-squares (+ 5 1) (* 6 2))
```

Applicative and normal order

```
(define square
   (lambda (x) (* x x)))

(define sum-squares
   (lambda (x y) (+ (square x) (square y))))

(define dist-between-pts
   (lambda (x1 y1 x2 y2)
      (sqrt (sum-squares (- x1 x2) (- y1 y2)))))

(sum-squares (+ 5 1) (* 6 2))
```

- Normal order is not as efficient as applicative order. We needed to evaluate (+ 5 1) and (* 6 2) twice each using normal order, instead of once each with applicative order.

Applicative and normal order

```
(define square
   (lambda (x) (* x x)))

(define sum-squares
   (lambda (x y) (+ (square x) (square y))))

(define dist-between-pts
   (lambda (x1 y1 x2 y2)
      (sqrt (sum-squares (- x1 x2) (- y1 y2)))))

(sum-squares (+ 5 1) (* 6 2))
```

- Normal order is not as efficient as applicative order. We needed to evaluate (+ 5 1) and (* 6 2) twice each using normal order, instead of once each with applicative order.
- How can we test if Scheme is applicative or normal order?

Applicative and normal order

```
(define p (lambda () (p)))
```

- What does this function do?

Applicative and normal order

```
(define p (lambda () (p)))
```

- What does this function do?
- It calls itself repeatedly, causing an infinite loop.

Applicative and normal order

```
(define p (lambda () (p)))
```

- What does this function do?
- It calls itself repeatedly, causing an infinite loop.

```
(define test
   (lambda (x y)
      (if (= x 0)
          0
          y)))

(test 0 (p))
```

- What happens with applicative order?

Applicative and normal order

```
(define p (lambda () (p)))
```

- What does this function do?
- It calls itself repeatedly, causing an infinite loop.

```
(define test
   (lambda (x y)
      (if (= x 0)
          0
          y)))

(test 0 (p))
```

- What happens with applicative order?
- We get an infinite loop.

Applicative and normal order

```
(define p (lambda () (p)))
```

- What does this function do?
- It calls itself repeatedly, causing an infinite loop.

```
(define test
   (lambda (x y)
      (if (= x 0)
          0
          y)))

(test 0 (p))
```

- What happens with applicative order?
- We get an infinite loop.
- What happens with normal order?

Applicative and normal order

```
(define p (lambda () (p)))
```

- What does this function do?
- It calls itself repeatedly, causing an infinite loop.

```
(define test
   (lambda (x y)
      (if (= x 0)
          0
          y)))

(test 0 (p))
```

- What happens with applicative order?
- We get an infinite loop.
- What happens with normal order?
- It returns 0.

Writing our own `if`

What if `if` were not a special form?

```
(define new-if
   (lambda (predicate consequent alternative)
      (cond (predicate consequent)
            (else alternative))))
```

What happens when we evaluate `(new-if (> 3 2) 0 2)`?

```
(new-if (> 3 2) 0 2)
(new-if (> 3 2) 0 2)
(new-if (> 3 2) 0 2)
(new-if (> 3 2) 0 2)
(new-if #t 0 2)
(cond (#t 0) (else 2))
0
```

No real problems so far.

Writing our own `if`

```
(define fact
   (lambda (n)
      (if (= n 0)
          1
          (* n (fact (- n 1))))))
```

- What if we use new-if instead of the special form if?

Writing our own `if`

```
(define fact
   (lambda (n)
      (if (= n 0)
          1
          (* n (fact (- n 1))))))
```

- What if we use new-if instead of the special form if?
- We'll get an infinite loop, since (* n (fact (- n 1))) will be evaluated every time, even if we have hit the base case.