Meta-Circular Evaluators

António Menezes Leitão

May 24, 2021

Meta-Circular Evaluators

Problem

- Programming languages are designed to facilitate the description of certain kinds of problems.
- Not all problems can be easily described in the current programming languages.

Solution: Linguistic Abstraction

- A new programming language is invented to facilitate the description of the problem.

Evaluator

Evaluator

An evaluator (or interpreter) of a language is a procedure that, when applied to an expression of that language, computes the operations described in that expression.

Fundamental Ideia

An evaluator is a program that computes the meaning of a program.

Meta-Circular Evaluator

An evaluator written in the language that it evaluates.

Evaluator

Abstract Syntax

Our evaluator will operate in terms of an *abstract syntax*: the syntax is recognized by predicates that abstract from the concrete syntax used.

Example: Concrete Syntax for Assignments

```
a := b

a = b

a ← b

(set! a b)
```

Example: Abstract Syntax for Assignments

```
(define (set? expression)
  (eq? (car expression) 'set!))
```

Evaluator

Scheme Evaluator

```
(define (eval exp)
```

Evaluator

Scheme Evaluator

```
(define (eval exp)
  (cond ((self-evaluating? exp) exp)
```

Evaluator

Scheme Evaluator

```
(define (eval exp)
  (cond ((self-evaluating? exp) exp)
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Evaluator

Scheme Evaluator

```
(define (eval exp)
  (cond ((self-evaluating? exp) exp)
        (else
         (error "Unknown expression type -- EVAL" exp))))

(define (self-evaluating? exp)
```

Evaluator

Scheme Evaluator

```
(define (eval exp)
  (cond ((self-evaluating? exp) exp)
        (else
         (error "Unknown expression type -- EVAL" exp))))

(define (self-evaluating? exp)
  (or (number? exp) (string? exp)))
```

Evaluator

Scheme Evaluator

```
(define (eval exp)
  (cond ((self-evaluating? exp) exp)
        (else
         (error "Unknown expression type -- EVAL" exp))))

(define (self-evaluating? exp)
  (or (number? exp) (string? exp)))

(define (repl)
```

Evaluator

Scheme Evaluator

```
(define (eval exp)
  (cond ((self-evaluating? exp) exp)
        (else
         (error "Unknown expression type -- EVAL" exp))))

(define (self-evaluating? exp)
  (or (number? exp) (string? exp)))

(define (repl)
  (prompt-for-input)
  (let ((input (read)))
    (let ((output (eval input)))
      (print output)))
  (repl))
```

Evaluator

Scheme Evaluator

```
(define (eval exp)
  (cond ((self-evaluating? exp) exp)
        (else
         (error "Unknown expression type -- EVAL" exp))))

(define (self-evaluating? exp)
  (or (number? exp) (string? exp)))

(define (repl)
  (prompt-for-input)
  (let ((input (read)))
    (let ((output (eval input)))
      (print output)))
  (repl))

(define (prompt-for-input)
  (display ">> "))
```

Evaluator

Scheme Evaluator

```
(define (eval exp)
  (cond ((self-evaluating? exp) exp)
        (else
         (error "Unknown expression type -- EVAL" exp))))

(define (self-evaluating? exp)
  (or (number? exp) (string? exp)))

(define (repl)
  (prompt-for-input)
  (let ((input (read)))
    (let ((output (eval input)))
      (print output)))
  (repl))

(define (prompt-for-input)
  (display ">> "))

(define (print object)
  (write object)
  (newline))
```

Evaluator

Example

```
>
```

Evaluator

Example

```
> (repl)
```

Evaluator

Example

```
> (repl)

>>
```

Evaluator

Example

```
> (repl)

>> 1
```

Evaluator

Example

```
> (repl)

>> 1
1
```

Evaluator

Example

```
> (repl)

>> 1
1
>> 2
```

Evaluator

Example

```
> (repl)

>> 1
1
>> 2
2
```

Evaluator

Example

```
> (repl)

>> 1
1
>> 2
2
>> "Hello"
```

Evaluator

Example

```
> (repl)

>> 1
1
>> 2
2
>> "Hello"
"Hello"
```

Evaluator

Example

```
> (repl)

>> 1
1
>> 2
2
>> "Hello"
"Hello"
>> (+ 1 2)
```

Evaluator

Example

```
> (repl)

>> 1
1
>> 2
2
>> "Hello"
"Hello"
>> (+ 1 2)
error in "Unknown expression type -- EVAL": (+ 1 2)

backtrace:
  0  (eval input)
  ..."eval.scm" line 14
  1  (repl)
  ..."/dev/stdin" line 1

>
```

Evaluator

Example

```
> (repl)

>> 1
1
>> 2
2
>> "Hello"
"Hello"
>> (+ 1 2)
error in "Unknown expression type -- EVAL": (+ 1 2)

backtrace:
  0  (eval input)
  ..."eval.scm" line 14
  1  (repl)
  ..."/dev/stdin" line 1

> (repl)
```

Evaluator

Example

```
> (repl)

>> 1
1
>> 2
2
>> "Hello"
"Hello"
>> (+ 1 2)
error in "Unknown expression type -- EVAL": (+ 1 2)

backtrace:
  0  (eval input)
  ..."eval.scm" line 14
  1  (repl)
  ..."/dev/stdin" line 1

> (repl)

>>
```

Evaluator

Scheme Evaluator

```
(define (eval exp)
  (cond ((self-evaluating? exp) exp)



        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Evaluator

Scheme Evaluator

```
(define (eval exp)
  (cond ((self-evaluating? exp) exp)
        ((addition? exp)


        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Evaluator

Scheme Evaluator

```
(define (eval exp)
  (cond ((self-evaluating? exp) exp)
        ((addition? exp)
         (+ (first-operand exp)
            (second-operand exp)))
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Evaluator

Scheme Evaluator

```
(define (eval exp)
  (cond ((self-evaluating? exp) exp)
        ((addition? exp)
         (+ (first-operand exp)
            (second-operand exp)))
        (else
         (error "Unknown expression type -- EVAL" exp))))

(define (addition? exp)
```

Evaluator

Scheme Evaluator

```
(define (eval exp)
  (cond ((self-evaluating? exp) exp)
        ((addition? exp)
         (+ (first-operand exp)
            (second-operand exp)))
        (else
         (error "Unknown expression type -- EVAL" exp))))

(define (addition? exp)
  (and (pair? exp)
       (eq? (car exp) 'plus)))
```

Evaluator

Scheme Evaluator

```scheme
(define (eval exp)
  (cond ((self-evaluating? exp) exp)
        ((addition? exp)
         (+ (first-operand exp)
            (second-operand exp)))
        (else
         (error "Unknown expression type -- EVAL" exp))))

(define (addition? exp)
  (and (pair? exp)
       (eq? (car exp) 'plus)))

(define (first-operand exp)
  (cadr exp))

(define (second-operand exp)
  (caddr exp))
```

Evaluator

Example

```
> (repl)
```

Evaluator

Example

```
> (repl)

>> (plus 1 2)
```

Evaluator

Example

```
> (repl)

>> (plus 1 2)
3
```

Evaluator

Example

```
> (repl)

>> (plus 1 2)
3
>> (plus (plus 1 2) (plus 3 4))
```

Evaluator

Example

```
> (repl)

>> (plus 1 2)
3
>> (plus (plus 1 2) (plus 3 4))
error in +: expected number, but got (plus 1 2), as argument 1

irritants:
  ((plus 1 2) (plus 3 4))

backtrace:
  0  (+ (first-operand exp) (second-operand exp))
  ..."eval.scm" line 31
  1  (eval input)
  ..."eval.scm" line 12
  2  (repl)
  ..."/dev/stdin" line 1
```

Evaluator

Scheme Evaluator

```
(define (eval exp)
  (cond ((self-evaluating? exp) exp)
        ((addition? exp)
         (+ (first-operand exp)
            (second-operand exp)))
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Evaluator

Scheme Evaluator

```
(define (eval exp)
  (cond ((self-evaluating? exp) exp)
        ((addition? exp)
         (+ (eval (first-operand exp))
            (eval (second-operand exp))))
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Evaluator

Scheme Evaluator

```
(define (eval exp)
  (cond ((self-evaluating? exp) exp)
        ((addition? exp)
         (+ (eval (first-operand exp))
            (eval (second-operand exp))))
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Example

```
>> (plus 1 2)
3
```

Evaluator

Scheme Evaluator

```
(define (eval exp)
  (cond ((self-evaluating? exp) exp)
        ((addition? exp)
         (+ (eval (first-operand exp))
            (eval (second-operand exp))))
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Example

```
>> (plus 1 2)
3
>> (plus (plus 1 2) (plus 3 4))
```

Evaluator

Scheme Evaluator

```
(define (eval exp)
  (cond ((self-evaluating? exp) exp)
        ((addition? exp)
         (+ (eval (first-operand exp))
            (eval (second-operand exp))))
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Example

```
>> (plus 1 2)
3
>> (plus (plus 1 2) (plus 3 4))
10
```

Evaluator

Scheme Evaluator

```
(define (eval exp)
  (cond ((self-evaluating? exp) exp)
        ((addition? exp)
         (+ (eval (first-operand exp))
            (eval (second-operand exp))))
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Example

```
>> (plus 1 2)
3
>> (plus (plus 1 2) (plus 3 4))
10
>> (times 2 3)
```

Evaluator

Scheme Evaluator

```
(define (eval exp)
  (cond ((self-evaluating? exp) exp)
        ((addition? exp)
         (+ (eval (first-operand exp))
            (eval (second-operand exp))))
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Example

```
>> (plus 1 2)
3
>> (plus (plus 1 2) (plus 3 4))
10
>> (times 2 3)
error in "Unknown expression type -- EVAL": (times 2 3)
```

Evaluator

Scheme Evaluator

```
(define (eval exp)
  (cond ((self-evaluating? exp) exp)
        ((addition? exp)
         (+ (eval (first-operand exp))
            (eval (second-operand exp))))



        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Evaluator

Scheme Evaluator

```
(define (eval exp)
  (cond ((self-evaluating? exp) exp)
        ((addition? exp)
         (+ (eval (first-operand exp))
            (eval (second-operand exp))))
        ((product? exp)


        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Evaluator

Scheme Evaluator

```
(define (eval exp)
  (cond ((self-evaluating? exp) exp)
        ((addition? exp)
         (+ (eval (first-operand exp))
            (eval (second-operand exp))))
        ((product? exp)
         (* (eval (first-operand exp))
            (eval (second-operand exp))))
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Evaluator

Scheme Evaluator

```
(define (eval exp)
  (cond ((self-evaluating? exp) exp)
        ((addition? exp)
         (+ (eval (first-operand exp))
            (eval (second-operand exp))))
        ((product? exp)
         (* (eval (first-operand exp))
            (eval (second-operand exp))))
        (else
         (error "Unknown expression type -- EVAL" exp))))

(define (product? exp)
  (and (pair? exp)
       (eq? (car exp) 'times)))
```

Evaluator

Example

```
>> (times 2 3)
6
```

Evaluator

Example

```
>> (times 2 3)
6
```

Improved Syntax

```
(define (addition? exp)
  (and (pair? exp)
       (eq? (car exp) '+)))

(define (product? exp)
  (and (pair? exp)
       (eq? (car exp) '*)))
```

Evaluator

Example

```
>> (times 2 3)
6
```

Improved Syntax

```
(define (addition? exp)
  (and (pair? exp)
       (eq? (car exp) '+)))

(define (product? exp)
  (and (pair? exp)
       (eq? (car exp) '*)))
```

Example

```
>> (+ 2 3)
5
>> (* (+ 2 3) (+ 4 5))
45
```

Evaluator

Naming

- Naming is the fundamental abstraction operator.
- A name abstracts a value and creates a scope where the name has a meaning.
- A name is not a first class value but denotes a first class value.
- Names are maintained in an *environment*: a collection of names.
- The environment is implemented as an *association list*:
  `((`$name_0$ `.` $value_0$`) (`$name_1$ `.` $value_1$`)` … `(`$name_n$ `.` $value_n$`))`

Example

```
>> (let ((pi 3.14159) (radius 5))
     (* 2 (* pi radius)))
31.4159
```

Evaluator

Scheme Evaluator

```
(define (eval exp)
  (cond ((self-evaluating? exp) exp)
        ((addition? exp)
         (+ (eval (first-operand exp))
            (eval (second-operand exp))))
        ((product? exp)
         (* (eval (first-operand exp))
            (eval (second-operand exp))))
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Evaluator

Scheme Evaluator

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((addition? exp)
         (+ (eval (first-operand exp) env)
            (eval (second-operand exp) env)))
        ((product? exp)
         (* (eval (first-operand exp) env)
            (eval (second-operand exp) env)))
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Evaluator

Scheme Evaluator

```scheme
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)




        ((addition? exp)
         (+ (eval (first-operand exp) env)
            (eval (second-operand exp) env)))
        ((product? exp)
         (* (eval (first-operand exp) env)
            (eval (second-operand exp) env)))
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Evaluator

Scheme Evaluator

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((name? exp)



        ((addition? exp)
         (+ (eval (first-operand exp) env)
            (eval (second-operand exp) env)))
        ((product? exp)
         (* (eval (first-operand exp) env)
            (eval (second-operand exp) env)))
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Evaluator

Scheme Evaluator

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((name? exp)
         (eval-name exp env))


        ((addition? exp)
         (+ (eval (first-operand exp) env)
            (eval (second-operand exp) env)))
        ((product? exp)
         (* (eval (first-operand exp) env)
            (eval (second-operand exp) env)))
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Evaluator

Scheme Evaluator

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((name? exp)
         (eval-name exp env))
        ((let? exp)

        ((addition? exp)
         (+ (eval (first-operand exp) env)
            (eval (second-operand exp) env)))
        ((product? exp)
         (* (eval (first-operand exp) env)
            (eval (second-operand exp) env)))
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Evaluator

Scheme Evaluator

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((name? exp)
         (eval-name exp env))
        ((let? exp)
         (eval-let exp env))
        ((addition? exp)
         (+ (eval (first-operand exp) env)
            (eval (second-operand exp) env)))
        ((product? exp)
         (* (eval (first-operand exp) env)
            (eval (second-operand exp) env)))
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Evaluator

Environment

```
;;Example:
;;(let ((pi 3.14159) (radius 5))
;;  (* 2 (* pi radius)))
```

Evaluator

Environment

```
;;Example:
;;(let ((pi 3.14159) (radius 5))
;;  (* 2 (* pi radius)))

(define (name? exp)
  (symbol? exp))
```

Evaluator

Environment

```
;;Example:
;;(let ((pi 3.14159) (radius 5))
;;  (* 2 (* pi radius)))

(define (name? exp)
  (symbol? exp))

(define (let? exp)
  (and (pair? exp)
       (eq? (car exp) 'let)))
```

Evaluator

Environment

```
;;Example:
;;(let ((pi 3.14159) (radius 5))
;;  (* 2 (* pi radius)))

(define (name? exp)
  (symbol? exp))

(define (let? exp)
  (and (pair? exp)
       (eq? (car exp) 'let)))

(define (let-names exp)
  (map car (cadr exp)))
```

Evaluator

Environment

```
;;Example:
;;(let ((pi 3.14159) (radius 5))
;;  (* 2 (* pi radius)))

(define (name? exp)
  (symbol? exp))

(define (let? exp)
  (and (pair? exp)
       (eq? (car exp) 'let)))

(define (let-names exp)
  (map car (cadr exp)))

(define (let-inits exp)
  (map cadr (cadr exp)))
```

Evaluator

Environment

```
;;Example:
;;(let ((pi 3.14159) (radius 5))
;;  (* 2 (* pi radius)))

(define (name? exp)
  (symbol? exp))

(define (let? exp)
  (and (pair? exp)
       (eq? (car exp) 'let)))

(define (let-names exp)
  (map car (cadr exp)))

(define (let-inits exp)
  (map cadr (cadr exp)))

(define (let-body exp)
  (caddr exp))
```

Evaluator

Environment

```
;;Environment implementation:
;;((name-0 . value-0) (name-1 . value-1) ... (name-n . value-n))
```

Evaluator

Environment

```
;;Environment implementation:
;;((name-0 . value-0) (name-1 . value-1) ... (name-n . value-n))

(define (eval-name name env)
  (cond ((null? env)
         (error "Unbound name -- EVAL-NAME" name))
        ((eq? name (caar env))
         (cdar env))
        (else
         (eval-name name (cdr env)))))
```

Evaluator

Environment

```
;;Environment implementation:
;;((name-0 . value-0) (name-1 . value-1) ... (name-n . value-n))

(define (eval-name name env)
  (cond ((null? env)
         (error "Unbound name -- EVAL-NAME" name))
        ((eq? name (caar env))
         (cdar env))
        (else
         (eval-name name (cdr env)))))

(define (eval-let exp env)
  (let ((values (eval-exprs (let-inits exp) env)))
    (let ((extended-environment
           (augment-environment (let-names exp)
                                values
                                env)))
      (eval (let-body exp) extended-environment))))
```

Evaluator

Environment

```
(define (eval-exprs exprs env)
  (if (null? exprs)
      (list)
      (cons (eval (car exprs) env)
            (eval-exprs (cdr exprs) env))))
```

Evaluator

Environment

```
(define (eval-exprs exprs env)
  (if (null? exprs)
      (list)
      (cons (eval (car exprs) env)
            (eval-exprs (cdr exprs) env))))

;;(let ((x 1) (y (+ 1 1))) ...)
;;=> (augment-environment '(x y) '(1 2) '((z . 3)))
```

Evaluator

Environment

```
(define (eval-exprs exprs env)
  (if (null? exprs)
      (list)
      (cons (eval (car exprs) env)
            (eval-exprs (cdr exprs) env))))

;;(let ((x 1) (y (+ 1 1))) ...)
;;=> (augment-environment '(x y) '(1 2) '((z . 3)))
;;<= ((x . 1) (y . 2) (z . 3))
```

Evaluator

Environment

```
(define (eval-exprs exprs env)
  (if (null? exprs)
      (list)
      (cons (eval (car exprs) env)
            (eval-exprs (cdr exprs) env))))

;;(let ((x 1) (y (+ 1 1))) ...)
;;=> (augment-environment '(x y) '(1 2) '((z . 3)))
;;<= ((x . 1) (y . 2) (z . 3))

(define (augment-environment names values env)
  (if (null? names)
      env
      (cons (cons (car names) (car values))
            (augment-environment (cdr names) (cdr values) env))))
```

Evaluator

Environment

```
(define (eval-exprs exprs env)
  (if (null? exprs)
      (list)
      (cons (eval (car exprs) env)
            (eval-exprs (cdr exprs) env))))

;;(let ((x 1) (y (+ 1 1))) ...)
;;=> (augment-environment '(x y) '(1 2) '((z . 3)))
;;<= ((x . 1) (y . 2) (z . 3))

(define (augment-environment names values env)
  (if (null? names)
      env
      (cons (cons (car names) (car values))
            (augment-environment (cdr names) (cdr values) env))))

(define empty-environment (list))
```

Evaluator

REPL

```
(define (repl)
  (prompt-for-input)
  (let ((input (read)))
    (let ((output (eval input)))
      (print output)))
  (repl))
```

Evaluator

REPL

```
(define (repl)
  (prompt-for-input)
  (let ((input (read)))
    (let ((output (eval input empty-environment)))
      (print output)))
  (repl))
```

Evaluator

REPL

```
(define (repl)
  (prompt-for-input)
  (let ((input (read)))
    (let ((output (eval input empty-environment)))
      (print output)))
  (repl))
```

Example

```
>> (let ((pi 3.14159) (radius 5))
     (* 2 (* pi radius)))
31.4159
```

Evaluator

Example

```
>> (let ((pi 3.14159) (radius 5))
     (* 2 (* pi radius)))
```

Evaluator

Example

```
>> (let ((pi 3.14159) (radius 5))
     (* 2 (* pi radius)))

=> (eval (let ((pi 3.14159) (radius 5)) (* 2 (* pi radius))) ())
```

Evaluator

Example

```
>> (let ((pi 3.14159) (radius 5))
     (* 2 (* pi radius)))

=> (eval (let ((pi 3.14159) (radius 5)) (* 2 (* pi radius))) ())
=>   (eval 3.14159 ())
```

Evaluator

Example

```
>> (let ((pi 3.14159) (radius 5))
     (* 2 (* pi radius)))

=> (eval (let ((pi 3.14159) (radius 5)) (* 2 (* pi radius))) ())
=>   (eval 3.14159 ())
<=   3.14159
```

Evaluator

Example

```
>> (let ((pi 3.14159) (radius 5))
     (* 2 (* pi radius)))

=> (eval (let ((pi 3.14159) (radius 5)) (* 2 (* pi radius))) ())
=>   (eval 3.14159 ())
<=   3.14159
=>   (eval 5 ())
<=   5
```

Evaluator

Example

```
>> (let ((pi 3.14159) (radius 5))
     (* 2 (* pi radius)))

=> (eval (let ((pi 3.14159) (radius 5)) (* 2 (* pi radius))) ())
=>   (eval 3.14159 ())
<=   3.14159
=>   (eval 5 ())
<=   5
=>   (eval (* 2 (* pi radius)) ((pi . 3.14159) (radius . 5)))
```

Evaluator

Example

```
>> (let ((pi 3.14159) (radius 5))
     (* 2 (* pi radius)))

=> (eval (let ((pi 3.14159) (radius 5)) (* 2 (* pi radius))) ())
=>   (eval 3.14159 ())
<=   3.14159
=>   (eval 5 ())
<=   5
=>   (eval (* 2 (* pi radius)) ((pi . 3.14159) (radius . 5)))
=>     (eval 2 ((pi . 3.14159) (radius . 5)))
<=     2
```

Evaluator

Example

```
>> (let ((pi 3.14159) (radius 5))
     (* 2 (* pi radius)))

=> (eval (let ((pi 3.14159) (radius 5)) (* 2 (* pi radius))) ())
=>   (eval 3.14159 ())
<=   3.14159
=>   (eval 5 ())
<=   5
=>   (eval (* 2 (* pi radius)) ((pi . 3.14159) (radius . 5)))
=>     (eval 2 ((pi . 3.14159) (radius . 5)))
<=     2
=>     (eval (* pi radius) ((pi . 3.14159) (radius . 5)))
```

Evaluator

Example

```
>> (let ((pi 3.14159) (radius 5))
     (* 2 (* pi radius)))

=> (eval (let ((pi 3.14159) (radius 5)) (* 2 (* pi radius))) ())
=>   (eval 3.14159 ())
<=   3.14159
=>   (eval 5 ())
<=   5
=>   (eval (* 2 (* pi radius)) ((pi . 3.14159) (radius . 5)))
=>     (eval 2 ((pi . 3.14159) (radius . 5)))
<=     2
=>     (eval (* pi radius) ((pi . 3.14159) (radius . 5)))
=>       (eval pi ((pi . 3.14159) (radius . 5)))
<=       3.14159
=>       (eval radius ((pi . 3.14159) (radius . 5)))
<=       5
```

Evaluator

Example

```
>> (let ((pi 3.14159) (radius 5))
     (* 2 (* pi radius)))

=> (eval (let ((pi 3.14159) (radius 5)) (* 2 (* pi radius))) ())
=>   (eval 3.14159 ())
<=   3.14159
=>   (eval 5 ())
<=   5
=>   (eval (* 2 (* pi radius)) ((pi . 3.14159) (radius . 5)))
=>     (eval 2 ((pi . 3.14159) (radius . 5)))
<=     2
=>     (eval (* pi radius) ((pi . 3.14159) (radius . 5)))
=>       (eval pi ((pi . 3.14159) (radius . 5)))
<=       3.14159
=>       (eval radius ((pi . 3.14159) (radius . 5)))
<=       5
<=     15.70795
```

Evaluator

Example

```
>> (let ((pi 3.14159) (radius 5))
     (* 2 (* pi radius)))

=> (eval (let ((pi 3.14159) (radius 5)) (* 2 (* pi radius))) ())
=>   (eval 3.14159 ())
<=   3.14159
=>   (eval 5 ())
<=   5
=>   (eval (* 2 (* pi radius)) ((pi . 3.14159) (radius . 5)))
=>     (eval 2 ((pi . 3.14159) (radius . 5)))
<=     2
=>     (eval (* pi radius) ((pi . 3.14159) (radius . 5)))
=>       (eval pi ((pi . 3.14159) (radius . 5)))
<=       3.14159
=>       (eval radius ((pi . 3.14159) (radius . 5)))
<=       5
<=     15.70795
<=   31.4159
```

Evaluator

Example

```
>> (let ((pi 3.14159) (radius 5))
     (* 2 (* pi radius)))

=> (eval (let ((pi 3.14159) (radius 5)) (* 2 (* pi radius))) ())
=>   (eval 3.14159 ())
<=   3.14159
=>   (eval 5 ())
<=   5
=>   (eval (* 2 (* pi radius)) ((pi . 3.14159) (radius . 5)))
=>     (eval 2 ((pi . 3.14159) (radius . 5)))
<=     2
=>     (eval (* pi radius) ((pi . 3.14159) (radius . 5)))
=>       (eval pi ((pi . 3.14159) (radius . 5)))
<=       3.14159
=>       (eval radius ((pi . 3.14159) (radius . 5)))
<=       5
<=     15.70795
<=   31.4159
<= 31.4159
```

Evaluator

Example

```
>> (let ((a 1))
     (let ((b (+ a 2)))
       (let ((a (+ a b)))
         (* 2 a))))
```

Evaluator

Example

```
>> (let ((a 1))
     (let ((b (+ a 2)))
       (let ((a (+ a b)))
         (* 2 a))))

=> (eval (let ((a 1)) (let ((b (+ a 2))) (let ((a (+ a b))) (* 2 a)))) ())
```

Evaluator

Example

```
>> (let ((a 1))
     (let ((b (+ a 2)))
       (let ((a (+ a b)))
         (* 2 a))))

=> (eval (let ((a 1)) (let ((b (+ a 2))) (let ((a (+ a b))) (* 2 a)))) ())
=>   (eval 1 ()) <= 1
```

Evaluator

Example

```
>> (let ((a 1))
     (let ((b (+ a 2)))
       (let ((a (+ a b)))
         (* 2 a))))

=> (eval (let ((a 1)) (let ((b (+ a 2))) (let ((a (+ a b))) (* 2 a)))) ())
=>   (eval 1 ()) <= 1
=>   (eval (let ((b (+ a 2))) (let ((a (+ a b))) (* 2 a))) ((a . 1)))
```

Evaluator

Example

```
>> (let ((a 1))
     (let ((b (+ a 2)))
       (let ((a (+ a b)))
         (* 2 a))))

=> (eval (let ((a 1)) (let ((b (+ a 2))) (let ((a (+ a b))) (* 2 a)))) ())
=>   (eval 1 ()) <= 1
=>   (eval (let ((b (+ a 2))) (let ((a (+ a b))) (* 2 a))) ((a . 1)))
=>     (eval (+ a 2) ((a . 1)))
```

Evaluator

Example

```
>> (let ((a 1))
     (let ((b (+ a 2)))
       (let ((a (+ a b)))
         (* 2 a))))

=> (eval (let ((a 1)) (let ((b (+ a 2))) (let ((a (+ a b))) (* 2 a)))) ())
=>   (eval 1 ()) <= 1
=>   (eval (let ((b (+ a 2))) (let ((a (+ a b))) (* 2 a))) ((a . 1)))
=>     (eval (+ a 2) ((a . 1)))
=>       (eval a ((a . 1))) <= 1
```

Evaluator

Example

```
>> (let ((a 1))
     (let ((b (+ a 2)))
       (let ((a (+ a b)))
         (* 2 a))))

=> (eval (let ((a 1)) (let ((b (+ a 2))) (let ((a (+ a b))) (* 2 a)))) ())
=>   (eval 1 ()) <= 1
=>   (eval (let ((b (+ a 2))) (let ((a (+ a b))) (* 2 a))) ((a . 1)))
=>     (eval (+ a 2) ((a . 1)))
=>       (eval a ((a . 1))) <= 1
=>       (eval 2 ((a . 1))) <= 2
```

Evaluator

Example

```
>> (let ((a 1))
     (let ((b (+ a 2)))
       (let ((a (+ a b)))
         (* 2 a))))

=> (eval (let ((a 1)) (let ((b (+ a 2))) (let ((a (+ a b))) (* 2 a)))) ())
=>   (eval 1 ()) <= 1
=>   (eval (let ((b (+ a 2))) (let ((a (+ a b))) (* 2 a))) ((a . 1)))
=>     (eval (+ a 2) ((a . 1)))
=>       (eval a ((a . 1))) <= 1
=>       (eval 2 ((a . 1))) <= 2
<=     3
```

Evaluator

Example

```
>> (let ((a 1))
     (let ((b (+ a 2)))
       (let ((a (+ a b)))
         (* 2 a))))

=> (eval (let ((a 1)) (let ((b (+ a 2))) (let ((a (+ a b))) (* 2 a)))) ())
=>   (eval 1 ()) <= 1
=>   (eval (let ((b (+ a 2))) (let ((a (+ a b))) (* 2 a))) ((a . 1)))
=>     (eval (+ a 2) ((a . 1)))
=>       (eval a ((a . 1))) <= 1
=>       (eval 2 ((a . 1))) <= 2
<=     3
=>     (eval (let ((a (+ a b))) (* 2 a)) ((b . 3) (a . 1)))
```

Evaluator

Example

```
>> (let ((a 1))
     (let ((b (+ a 2)))
       (let ((a (+ a b)))
         (* 2 a))))

=> (eval (let ((a 1)) (let ((b (+ a 2))) (let ((a (+ a b))) (* 2 a)))) ())
=>   (eval 1 ()) <= 1
=>   (eval (let ((b (+ a 2))) (let ((a (+ a b))) (* 2 a))) ((a . 1)))
=>     (eval (+ a 2) ((a . 1)))
=>       (eval a ((a . 1))) <= 1
=>       (eval 2 ((a . 1))) <= 2
<=     3
=>     (eval (let ((a (+ a b))) (* 2 a)) ((b . 3) (a . 1)))
=>       (eval (+ a b) ((b . 3) (a . 1)))
```

Evaluator

Example

```
>> (let ((a 1))
     (let ((b (+ a 2)))
       (let ((a (+ a b)))
         (* 2 a))))

=> (eval (let ((a 1)) (let ((b (+ a 2))) (let ((a (+ a b))) (* 2 a)))) ())
=>   (eval 1 ()) <= 1
=>   (eval (let ((b (+ a 2))) (let ((a (+ a b))) (* 2 a))) ((a . 1)))
=>     (eval (+ a 2) ((a . 1)))
=>       (eval a ((a . 1))) <= 1
=>       (eval 2 ((a . 1))) <= 2
<=     3
=>     (eval (let ((a (+ a b))) (* 2 a)) ((b . 3) (a . 1)))
=>       (eval (+ a b) ((b . 3) (a . 1)))
=>         (eval a ((b . 3) (a . 1))) <= 1
```

Evaluator

Example

```
>> (let ((a 1))
     (let ((b (+ a 2)))
       (let ((a (+ a b)))
         (* 2 a))))

=> (eval (let ((a 1)) (let ((b (+ a 2))) (let ((a (+ a b))) (* 2 a)))) ())
=>   (eval 1 ()) <= 1
=>   (eval (let ((b (+ a 2))) (let ((a (+ a b))) (* 2 a))) ((a . 1)))
=>     (eval (+ a 2) ((a . 1)))
=>       (eval a ((a . 1))) <= 1
=>       (eval 2 ((a . 1))) <= 2
<=     3
=>     (eval (let ((a (+ a b))) (* 2 a)) ((b . 3) (a . 1)))
=>       (eval (+ a b) ((b . 3) (a . 1)))
=>         (eval a ((b . 3) (a . 1))) <= 1
=>         (eval b ((b . 3) (a . 1))) <= 3
```

Evaluator

Example

```
>> (let ((a 1))
     (let ((b (+ a 2)))
       (let ((a (+ a b)))
         (* 2 a))))

=> (eval (let ((a 1)) (let ((b (+ a 2))) (let ((a (+ a b))) (* 2 a)))) ())
=>   (eval 1 ()) <= 1
=>   (eval (let ((b (+ a 2))) (let ((a (+ a b))) (* 2 a))) ((a . 1)))
=>     (eval (+ a 2) ((a . 1)))
=>       (eval a ((a . 1))) <= 1
=>       (eval 2 ((a . 1))) <= 2
<=     3
=>     (eval (let ((a (+ a b))) (* 2 a)) ((b . 3) (a . 1)))
=>       (eval (+ a b) ((b . 3) (a . 1)))
=>         (eval a ((b . 3) (a . 1))) <= 1
=>         (eval b ((b . 3) (a . 1))) <= 3
<=       4
```

Evaluator

Example

```
>> (let ((a 1))
     (let ((b (+ a 2)))
       (let ((a (+ a b)))
         (* 2 a))))

=> (eval (let ((a 1)) (let ((b (+ a 2))) (let ((a (+ a b))) (* 2 a)))) ())
=>   (eval 1 ()) <= 1
=>   (eval (let ((b (+ a 2))) (let ((a (+ a b))) (* 2 a))) ((a . 1)))
=>     (eval (+ a 2) ((a . 1)))
=>       (eval a ((a . 1))) <= 1
=>       (eval 2 ((a . 1))) <= 2
<=     3
=>     (eval (let ((a (+ a b))) (* 2 a)) ((b . 3) (a . 1)))
=>       (eval (+ a b) ((b . 3) (a . 1)))
=>         (eval a ((b . 3) (a . 1))) <= 1
=>         (eval b ((b . 3) (a . 1))) <= 3
<=       4
=>       (eval (* 2 a) ((a . 4) (b . 3) (a . 1)))
```

Evaluator

Example

```
>> (let ((a 1))
     (let ((b (+ a 2)))
       (let ((a (+ a b)))
         (* 2 a))))

=> (eval (let ((a 1)) (let ((b (+ a 2))) (let ((a (+ a b))) (* 2 a)))) ())
=>   (eval 1 ()) <= 1
=>   (eval (let ((b (+ a 2))) (let ((a (+ a b))) (* 2 a))) ((a . 1)))
=>     (eval (+ a 2) ((a . 1)))
=>       (eval a ((a . 1))) <= 1
=>       (eval 2 ((a . 1))) <= 2
<=     3
=>     (eval (let ((a (+ a b))) (* 2 a)) ((b . 3) (a . 1)))
=>       (eval (+ a b) ((b . 3) (a . 1)))
=>         (eval a ((b . 3) (a . 1))) <= 1
=>         (eval b ((b . 3) (a . 1))) <= 3
<=       4
=>       (eval (* 2 a) ((a . 4) (b . 3) (a . 1)))
=>         (eval 2 ((a . 4) (b . 3) (a . 1))) <= 2
```

Evaluator

Example

```
>> (let ((a 1))
     (let ((b (+ a 2)))
       (let ((a (+ a b)))
         (* 2 a))))

=> (eval (let ((a 1)) (let ((b (+ a 2))) (let ((a (+ a b))) (* 2 a)))) ())
=>   (eval 1 ()) <= 1
=>   (eval (let ((b (+ a 2))) (let ((a (+ a b))) (* 2 a))) ((a . 1)))
=>     (eval (+ a 2) ((a . 1)))
=>       (eval a ((a . 1))) <= 1
=>       (eval 2 ((a . 1))) <= 2
<=     3
=>     (eval (let ((a (+ a b))) (* 2 a)) ((b . 3) (a . 1)))
=>       (eval (+ a b) ((b . 3) (a . 1)))
=>         (eval a ((b . 3) (a . 1))) <= 1
=>         (eval b ((b . 3) (a . 1))) <= 3
<=       4
=>       (eval (* 2 a) ((a . 4) (b . 3) (a . 1)))
=>         (eval 2 ((a . 4) (b . 3) (a . 1))) <= 2
=>         (eval a ((a . 4) (b . 3) (a . 1))) <= 4
```

Evaluator

Example

```
>> (let ((a 1))
     (let ((b (+ a 2)))
       (let ((a (+ a b)))
         (* 2 a))))

=> (eval (let ((a 1)) (let ((b (+ a 2))) (let ((a (+ a b))) (* 2 a)))) ())
=>   (eval 1 ()) <= 1
=>   (eval (let ((b (+ a 2))) (let ((a (+ a b))) (* 2 a))) ((a . 1)))
=>     (eval (+ a 2) ((a . 1)))
=>       (eval a ((a . 1))) <= 1
=>       (eval 2 ((a . 1))) <= 2
<=     3
=>     (eval (let ((a (+ a b))) (* 2 a)) ((b . 3) (a . 1)))
=>       (eval (+ a b) ((b . 3) (a . 1)))
=>         (eval a ((b . 3) (a . 1))) <= 1
=>         (eval b ((b . 3) (a . 1))) <= 3
<=       4
=>       (eval (* 2 a) ((a . 4) (b . 3) (a . 1)))
=>         (eval 2 ((a . 4) (b . 3) (a . 1))) <= 2
=>         (eval a ((a . 4) (b . 3) (a . 1))) <= 4
<=       8
<=     8
<=   8
<= 8
```

Evaluator

Pre-defined Names

- The REPL starts in a clean state.
- Forcing us to introduce all needed names.
- Make it better: we can provide some pre-defined names.

Evaluator

Pre-defined Names

- The REPL starts in a clean state.
- Forcing us to introduce all needed names.
- Make it better: we can provide some pre-defined names.

Initial Bindings

```
(define initial-bindings
  (list (cons 'pi 3.14159)
        (cons 'e  2.71828)
        ...))
```

Evaluator

Pre-defined Names

- The REPL starts in a clean state.
- Forcing us to introduce all needed names.
- Make it better: we can provide some pre-defined names.

Initial Bindings

```
(define initial-bindings
  (list (cons 'pi 3.14159)
        (cons 'e  2.71828)
        ...))

(define initial-environment
  (augment-environment (map car initial-bindings)
                       (map cdr initial-bindings)
                       empty-environment))
```

Evaluator

REPL

```
(define (repl)
  (prompt-for-input)
  (let ((input (read)))
    (let ((output (eval input empty-environment)))
      (print output)))
  (repl))
```

Evaluator

REPL

```
(define (repl)
  (prompt-for-input)
  (let ((input (read)))
    (let ((output (eval input initial-environment)))
      (print output)))
  (repl))
```

Evaluator

REPL

```
(define (repl)
  (prompt-for-input)
  (let ((input (read)))
    (let ((output (eval input initial-environment)))
      (print output)))
  (repl))
```

Example

```
>> pi
```

Evaluator

REPL

```
(define (repl)
  (prompt-for-input)
  (let ((input (read)))
    (let ((output (eval input initial-environment)))
      (print output)))
  (repl))
```

Example

```
>> pi
3.14159
```

Evaluator

REPL

```
(define (repl)
  (prompt-for-input)
  (let ((input (read)))
    (let ((output (eval input initial-environment)))
      (print output)))
  (repl))
```

Example

```
>> pi
3.14159
>> (+ pi e)
5.85987
```

Evaluator

REPL

```
(define (repl)
  (prompt-for-input)
  (let ((input (read)))
    (let ((output (eval input initial-environment)))
      (print output)))
  (repl))
```

Example

```
>> pi
3.14159
>> (+ pi e)
5.85987
>> (let ((e 10))
     (* pi e))
```

Evaluator

REPL

```
(define (repl)
  (prompt-for-input)
  (let ((input (read)))
    (let ((output (eval input initial-environment)))
      (print output)))
  (repl))
```

Example

```
>> pi
3.14159
>> (+ pi e)
5.85987
>> (let ((e 10))
      (* pi e))
31.4159
```

Evaluator

REPL

```
(define (repl)
  (prompt-for-input)
  (let ((input (read)))
    (let ((output (eval input initial-environment)))
      (print output)))
  (repl))
```

Example

```
>> pi
3.14159
>> (+ pi e)
5.85987
>> (let ((e 10))
     (* pi e))
31.4159
>> (* (let ((e 10)) (* e e)) e)
```

Evaluator

REPL

```
(define (repl)
  (prompt-for-input)
  (let ((input (read)))
    (let ((output (eval input initial-environment)))
      (print output)))
  (repl))
```

Example

```
>> pi
3.14159
>> (+ pi e)
5.85987
>> (let ((e 10))
     (* pi e))
31.4159
>> (* (let ((e 10)) (* e e)) e)
271.828
```

Evaluator

Example: Add the areas of two circles with radius 10 and 100

```
>> (let ((radius-0 10))
     (let ((square-radius-0 (* radius-0 radius-0)))
       (let ((area-circle-0 (* pi square-radius-0)))
         (let ((radius-1 100))
           (let ((square-radius-1 (* radius-1 radius-1)))
             (let ((area-circle-1 (* pi square-radius-1)))
               (+ area-circle-0 area-circle-1)))))))
31730.058999999997
```

Evaluator

Example: Add the areas of two circles with radius 10 and 100

```
>> (let ((radius-0 10))
     (let ((square-radius-0 (* radius-0 radius-0)))
       (let ((area-circle-0 (* pi square-radius-0)))
         (let ((radius-1 100))
           (let ((square-radius-1 (* radius-1 radius-1)))
             (let ((area-circle-1 (* pi square-radius-1)))
               (+ area-circle-0 area-circle-1)))))))
31730.058999999997
```

Functions

- So far, we only have abstracted *values*.
- But it is also good to have abstracted *expressions*.
- Abstract expressions are known as *functions*.

Evaluator

Example: Add the areas of two circles with radius 10 and 100

```
>> (let ((radius-0 10) (radius-1 100))
     (flet ((square (x) (* x x)))
       (flet ((area-circle (radius)
                (* pi (call square radius))))
         (+ (call area-circle radius-0)
            (call area-circle radius-1)))))
31730.058999999997
```

Functions

- Abstracting an expression allow us to give it a name and to reuse it in several different places just by naming it.
- We must distinguish between names that are bound in the surrounding context and names that are bound in the function call (e.g., `pi` *vs* `radius` inside `area-circle`).
- The distinction is made by providing a *parameter list*.

Evaluator

Functions

- New naming operator: `flet`
  - Augment the environment to contain associations between the names and the functions.
- New call operator: `call`
  - Obtain the function associated with the name.
  - Evaluate all argument expressions.
  - Augment the environment to contain the associations between function parameters and the evaluated arguments.
  - Evaluate the function body in the augmented environment.

Evaluator

Functions

```
;(call foo 1 (+ 2 3) 4)
```

Evaluator

Functions

```
;(call foo 1 (+ 2 3) 4)

(define (call? exp)
  (and (pair? exp)
       (eq? (car exp) 'call)))
```

Evaluator

Functions

```
;(call foo 1 (+ 2 3) 4)

(define (call? exp)
  (and (pair? exp)
       (eq? (car exp) 'call)))

(define (call-operator exp)
  (cadr exp))
```

Evaluator

Functions

```
;(call foo 1 (+ 2 3) 4)

(define (call? exp)
  (and (pair? exp)
       (eq? (car exp) 'call)))

(define (call-operator exp)
  (cadr exp))

(define (call-operands exp)
  (cddr exp))
```

Evaluator

Functions

```
;(call foo 1 (+ 2 3) 4)

(define (call? exp)
  (and (pair? exp)
       (eq? (car exp) 'call)))

(define (call-operator exp)
  (cadr exp))

(define (call-operands exp)
  (cddr exp))

;(flet ((square (x) (* x x))
;       (foo (a b c) (+ a (* b c))))
;  ...)
```

Evaluator

Functions

```
;(call foo 1 (+ 2 3) 4)

(define (call? exp)
  (and (pair? exp)
       (eq? (car exp) 'call)))

(define (call-operator exp)
  (cadr exp))

(define (call-operands exp)
  (cddr exp))

;(flet ((square (x) (* x x))
;        (foo (a b c) (+ a (* b c))))
;   ...)

(define (flet? exp)
  (and (pair? exp)
       (eq? (car exp) 'flet)))
```

Evaluator

Functions

```
;(flet ((square (x) (* x x))
;       (foo (a b c) (+ a (* b c))))
;   ...)
```

Evaluator

Functions

```
;(flet ((square (x) (* x x))
;       (foo (a b c) (+ a (* b c))))
;   ...)

(define (flet-names exp)
  (map car (cadr exp)))
```

Evaluator

Functions

```
;(flet ((square (x) (* x x))
;        (foo (a b c) (+ a (* b c))))
;   ...)

(define (flet-names exp)
  (map car (cadr exp)))

(define (flet-functions exp)
  (map (lambda (f)
         (make-function (cadr f) (cddr f)))
       (cadr exp)))
```

Evaluator

Functions

```
;(flet ((square (x) (* x x))
;       (foo (a b c) (+ a (* b c))))
;  ...)

(define (flet-names exp)
  (map car (cadr exp)))

(define (flet-functions exp)
  (map (lambda (f)
         (make-function (cadr f) (cddr f)))
       (cadr exp)))

(define (make-function parameters body)
  (cons 'function
        (cons parameters
              body)))
```

Evaluator

Functions

```
;(flet ((square (x) (* x x))
;       (foo (a b c) (+ a (* b c))))
;  ...)

(define (flet-names exp)
  (map car (cadr exp)))

(define (flet-functions exp)
  (map (lambda (f)
         (make-function (cadr f) (cddr f)))
       (cadr exp)))

(define (make-function parameters body)
  (cons 'function
        (cons parameters
              body)))

(define (flet-body exp)
  (caddr exp))
```

Evaluator

Flet Evaluation

```
;; Flet Evaluation
;; exp: (flet ((bar (x) (+ x 5)))
;;        (+ (call bar a) b))
;; env: ((a . 1) (b . 2))
;; =>
;; exp: (+ (call bar a) b)
;; env: ((bar . (function (x) (+ x 5))) ((a . 1) (b . 2)))
```

Evaluator

Flet Evaluation

```
;; Flet Evaluation
;; exp: (flet ((bar (x) (+ x 5)))
;;        (+ (call bar a) b))
;; env: ((a . 1) (b . 2))
;; =>
;; exp: (+ (call bar a) b)
;; env: ((bar . (function (x) (+ x 5))) ((a . 1) (b . 2)))

(define (eval-flet exp env)
  (let ((extended-environment
         (augment-environment (flet-names exp)
                              (flet-functions exp)
                              env)))
    (eval (flet-body exp) extended-environment)))
```

Evaluator

Call Evaluation

```
;; exp: (+ (call bar a) b)
;; env: ((bar . (function (x) (+ x 5))) ((a . 1) (b . 2)))

(define (function? obj)
  (and (pair? obj)
       (eq? (car obj) 'function)))
```

Evaluator

Call Evaluation

```
;; exp: (+ (call bar a) b)
;; env: ((bar . (function (x) (+ x 5))) ((a . 1) (b . 2)))

(define (function? obj)
  (and (pair? obj)
       (eq? (car obj) 'function)))

(define (function-parameters func)
  (cadr func))
```

Evaluator

Call Evaluation

```
;; exp: (+ (call bar a) b)
;; env: ((bar . (function (x) (+ x 5))) ((a . 1) (b . 2)))

(define (function? obj)
  (and (pair? obj)
       (eq? (car obj) 'function)))

(define (function-parameters func)
  (cadr func))

(define (function-body func)
  (caddr func))
```

Evaluator

Call Evaluation

```
;; exp: (+ (call bar a) b)
;; env: ((bar . (function (x) (+ x 5))) ((a . 1) (b . 2)))

(define (function? obj)
  (and (pair? obj)
       (eq? (car obj) 'function)))

(define (function-parameters func)
  (cadr func))

(define (function-body func)
  (caddr func))

(define (eval-call exp env)
  (let ((func (eval-name (call-operator exp) env))
        (args (eval-exprs (call-operands exp) env)))
    (let ((extended-environment
           (augment-environment (function-parameters func)
                                args
                                env)))
      (eval (function-body func) extended-environment))))
```

Evaluator

Evaluator

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((name? exp)
         (eval-name exp env))
        ((let? exp)
         (eval-let exp env))
        ((addition? exp)
         (+ (eval (first-operand exp) env)
            (eval (second-operand exp) env)))
        ((product? exp)
         (* (eval (first-operand exp) env)
            (eval (second-operand exp) env)))




        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Evaluator

Evaluator

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((name? exp)
         (eval-name exp env))
        ((let? exp)
         (eval-let exp env))
        ((addition? exp)
         (+ (eval (first-operand exp) env)
            (eval (second-operand exp) env)))
        ((product? exp)
         (* (eval (first-operand exp) env)
            (eval (second-operand exp) env)))
        ((flet? exp)



        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Evaluator

Evaluator

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((name? exp)
         (eval-name exp env))
        ((let? exp)
         (eval-let exp env))
        ((addition? exp)
         (+ (eval (first-operand exp) env)
            (eval (second-operand exp) env)))
        ((product? exp)
         (* (eval (first-operand exp) env)
            (eval (second-operand exp) env)))
        ((flet? exp)
         (eval-flet exp env))


        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Evaluator

Evaluator

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((name? exp)
         (eval-name exp env))
        ((let? exp)
         (eval-let exp env))
        ((addition? exp)
         (+ (eval (first-operand exp) env)
            (eval (second-operand exp) env)))
        ((product? exp)
         (* (eval (first-operand exp) env)
            (eval (second-operand exp) env)))
        ((flet? exp)
         (eval-flet exp env))
        ((call? exp)

        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Evaluator

Evaluator

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((name? exp)
         (eval-name exp env))
        ((let? exp)
         (eval-let exp env))
        ((addition? exp)
         (+ (eval (first-operand exp) env)
            (eval (second-operand exp) env)))
        ((product? exp)
         (* (eval (first-operand exp) env)
            (eval (second-operand exp) env)))
        ((flet? exp)
         (eval-flet exp env))
        ((call? exp)
         (eval-call exp env))
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Evaluator

A Striking Resemblance

```
(define (eval-let exp env)
  (let ((values (eval-exprs (let-inits exp) env)))
    (let ((extended-environment
           (augment-environment (let-names exp)
                                values
                                env)))
      (eval (let-body exp) extended-environment))))

(define (eval-call exp env)
  (let ((func (eval-name (call-operator exp) env))
        (args (eval-exprs (call-operands exp) env)))
    (let ((extended-environment
           (augment-environment (function-parameters func)
                                args
                                env)))
      (eval (function-body func) extended-environment))))
```

Evaluator

A Striking Resemblance

```
(define (eval-let exp env)
  (let ((values (eval-exprs (let-inits exp) env)))
    (let ((extended-environment
           (augment-environment (let-names exp)
                                values
                                env)))
      (eval (let-body exp) extended-environment))))

(define (eval-call exp env)
  (let ((func (eval-name (call-operator exp) env))
        (args (eval-exprs (call-operands exp) env)))
    (let ((extended-environment
           (augment-environment (function-parameters func)
                                args
                                env)))
      (eval (function-body func) extended-environment))))
```

Evaluator

A Striking Resemblance

```
(define (eval-let exp env)
  (let ((values (eval-exprs (let-inits exp) env)))
    (let ((extended-environment
           (augment-environment (let-names exp)
                                values
                                env)))
      (eval (let-body exp) extended-environment))))

(define (eval-call exp env)
  (let ((func (eval-name (call-operator exp) env))
        (args (eval-exprs (call-operands exp) env)))
    (let ((extended-environment
           (augment-environment (function-parameters func)
                                args
                                env)))
      (eval (function-body func) extended-environment))))
```

Evaluator

A Striking Resemblance

```
(define (eval-let exp env)
  (let ((values (eval-exprs (let-inits exp) env)))
    (let ((extended-environment
           (augment-environment (let-names exp)
                                values
                                env)))
      (eval (let-body exp) extended-environment))))

(define (eval-call exp env)
  (let ((func (eval-name (call-operator exp) env))
        (args (eval-exprs (call-operands exp) env)))
    (let ((extended-environment
           (augment-environment (function-parameters func)
                                args
                                env)))
      (eval (function-body func) extended-environment))))
```

Evaluator

A Striking Resemblance

- The evaluation of a `let` form is very similar to the evaluation of a function call.
- In fact, any `let` form can be "translated" into a function that is immediately called.
- We will return to this subject later.

Translation Example

```
>> (let ((a 1) (b 2))
     (+ a b))
```

Evaluator

A Striking Resemblance

- The evaluation of a `let` form is very similar to the evaluation of a function call.
- In fact, any `let` form can be "translated" into a function that is immediately called.
- We will return to this subject later.

Translation Example

```
>> (let ((a 1) (b 2))
     (+ a b))
```

Evaluator

A Striking Resemblance

- The evaluation of a `let` form is very similar to the evaluation of a function call.
- In fact, any `let` form can be "translated" into a function that is immediately called.
- We will return to this subject later.

Translation Example

```
>> (let ((a 1) (b 2))
     (+ a b))

>> (flet ((anonymous (a b)
            (+ a b)))
     (call anonymous 1 2))
```

Evaluator

Example: Add the areas of two circles with radius 10 and 100

```
>> (let ((radius-0 10) (radius-1 100))
     (flet ((square (x) (* x x)))
       (flet ((area-circle (radius)
                (* pi (call square radius))))
         (+ (call area-circle radius-0)
            (call area-circle radius-1)))))
```

Evaluator

Example: Add the areas of two circles with radius 10 and 100

```
>> (let ((radius-0 10) (radius-1 100))
     (flet ((square (x) (* x x)))
       (flet ((area-circle (radius)
                (* pi (call square radius))))
         (+ (call area-circle radius-0)
            (call area-circle radius-1)))))
31730.058999999997
```

Evaluator

Example: Add the areas of two circles with radius 10 and 100

```
>> (let ((radius-0 10) (radius-1 100))
     (flet ((square (x) (* x x)))
       (flet ((area-circle (radius)
                 (* pi (call square radius))))
         (+ (call area-circle radius-0)
            (call area-circle radius-1)))))
31730.058999999997
```

Problem

- There is a syntactic difference between the operators (e.g. `(* 10 10)`) and the functions (e.g. `(call square 10)`)
- It will look much better if we use the same syntax (e.g. `(square 10)`).
- We just have to redefine the concrete syntax of function calls.

Evaluator

Evaluator

```
;;(square radius)
```

Evaluator

Evaluator

```
;;(square radius)

(define (call? exp)
  (pair? exp))
```

Evaluator

Evaluator

```
;;(square radius)

(define (call? exp)
  (pair? exp))

(define (call-operator exp)
  (car exp))
```

Evaluator

Evaluator

```
;;(square radius)

(define (call? exp)
  (pair? exp))

(define (call-operator exp)
  (car exp))

(define (call-operands exp)
  (cdr exp))
```

Evaluator

Evaluator

```
;;(square radius)

(define (call? exp)
  (pair? exp))

(define (call-operator exp)
  (car exp))

(define (call-operands exp)
  (cdr exp))

;;Let's also add square as a pre-defined function:
```

Evaluator

Evaluator

```
;;(square radius)

(define (call? exp)
  (pair? exp))

(define (call-operator exp)
  (car exp))

(define (call-operands exp)
  (cdr exp))

;;Let's also add square as a pre-defined function:
(define initial-bindings
  (list (cons 'pi 3.14159)
        (cons 'e  2.71828)
        (cons 'square (make-function '(x) '((* x x))))))
```

Evaluator

Example

```
>> (square 10)
100
```

Evaluator

Example

```
>> (square 10)
100
>> (* (square (+ 1 2)) 3)
27
```

Evaluator

Example

```
>> (square 10)
100
>> (* (square (+ 1 2)) 3)
27
>> (flet ((area-circle (radius)
            (* pi (square radius))))
     (+ (area-circle 10)
        (area-circle 100)))
```

Evaluator

Example

```
>> (square 10)
100
>> (* (square (+ 1 2)) 3)
27
>> (flet ((area-circle (radius)
            (* pi (square radius))))
     (+ (area-circle 10)
        (area-circle 100)))
31730.058999999997
```

Evaluator

Example: Rebinding Names

```
>> (flet ((square (a) "I'm a square"))
     (square 10))
```

Evaluator

Example: Rebinding Names

```
>> (flet ((square (a) "I'm a square"))
     (square 10))
"I'm a square"
```

Evaluator

Example: Rebinding Names

```
>> (flet ((square (a) "I'm a square"))
     (square 10))
"I'm a square"
>> (flet ((+ (x y) "I'm an addition"))
     (+ 10 20))
```

Evaluator

Example: Rebinding Names

```
>> (flet ((square (a) "I'm a square"))
     (square 10))
"I'm a square"
>> (flet ((+ (x y) "I'm an addition"))
     (+ 10 20))
30
```

Evaluator

Example: Rebinding Names

```
>> (flet ((square (a) "I'm a square"))
     (square 10))
"I'm a square"
>> (flet ((+ (x y) "I'm an addition"))
     (+ 10 20))
30
```

Problem

- There is a semantic difference between operators (e.g. `(+ 10 20)`) and functions (e.g. `(square 10)`)
- Names bound to functions can be shadowed.
- Names bound to operators cannot be shadowed.
- The problem can be solved by treating operators as being bound to special *primitive* operations.

Evaluator

Primitives

```
(define (make-primitive f)
  (list 'primitive f))
```

Evaluator

Primitives

```
(define (make-primitive f)
  (list 'primitive f))

(define (primitive? obj)
  (and (pair? obj)
       (eq? (car obj) 'primitive)))
```

Evaluator

Primitives

```
(define (make-primitive f)
  (list 'primitive f))

(define (primitive? obj)
  (and (pair? obj)
       (eq? (car obj) 'primitive)))

(define (primitive-operation prim)
  (cadr prim))
```

Evaluator

Primitives

```
(define (make-primitive f)
  (list 'primitive f))

(define (primitive? obj)
  (and (pair? obj)
       (eq? (car obj) 'primitive)))

(define (primitive-operation prim)
  (cadr prim))

(define initial-bindings
  (list (cons 'pi 3.14159)
        (cons 'e  2.71828)
        (cons 'square (make-function '(x) '((* x x))))
        (cons '+ (make-primitive +))
        (cons '* (make-primitive *))))
```

Evaluator

Primitives

```
(define (apply-primitive-function prim args)
  (cond ((eq? (primive-operation prim) +)
          (+ (car args) (cadr args)))
        ((eq? (primive-operation prim) *)
          (* (car args) (cadr args)))
        ...))
```

Evaluator

Primitives

```
(define (apply-primitive-function prim args)
  (apply (primitive-operation prim) args))
```

Evaluator

Primitives

```
(define (apply-primitive-function prim args)
  (apply (primitive-operation prim) args))

(define (eval-call exp env)
  (let ((func (eval-name (call-operator exp) env))
        (args (eval-exprs (call-operands exp) env)))
    (if (primitive? func)
        (apply-primitive-function func args)
        (let ((extended-environment
               (augment-environment (function-parameters func)
                                    args
                                    env)))
          (eval (function-body func) extended-environment)))))
```

Evaluator

Evaluator

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((name? exp)
         (eval-name exp env))
        ((let? exp)
         (eval-let exp env))
        ((addition? exp)
         (+ (eval (first-operand exp) env)
            (eval (second-operand exp) env)))
        ((product? exp)
         (* (eval (first-operand exp) env)
            (eval (second-operand exp) env)))
        ((flet? exp)
         (eval-flet exp env))
        ((call? exp)
         (eval-call exp env))
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Evaluator

Evaluator

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((name? exp)
         (eval-name exp env))
        ((let? exp)
         (eval-let exp env))
        ((flet? exp)
         (eval-flet exp env))
        ((call? exp)
         (eval-call exp env))
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Evaluator

Example

```
>> (+ 3 4)
```

Evaluator

Example

```
>> (+ 3 4)
7
```

Evaluator

Example

```
>> (+ 3 4)
7
>> (flet ((+ (x y) "I'm an addition"))
     (+ 10 20))
```

Evaluator

Example

```
>> (+ 3 4)
7
>> (flet ((+ (x y) "I'm an addition"))
     (+ 10 20))
"I'm an addition"
```

Evaluator

Example

```
>> (+ 3 4)
7
>> (flet ((+ (x y) "I'm an addition"))
     (+ 10 20))
"I'm an addition"
>> (+ 3 (flet ((+ (x y) (* x y))) (+ 4 5)))
```

Evaluator

Example

```
>> (+ 3 4)
7
>> (flet ((+ (x y) "I'm an addition"))
     (+ 10 20))
"I'm an addition"
>> (+ 3 (flet ((+ (x y) (* x y))) (+ 4 5)))
23
```

Evaluator

Example

```
>> (+ 3 4)
7
>> (flet ((+ (x y) "I'm an addition"))
     (+ 10 20))
"I'm an addition"
>> (+ 3 (flet ((+ (x y) (* x y))) (+ 4 5)))
23
>> (+ 1 2 3 4 5 6)
```

Evaluator

Example

```
>> (+ 3 4)
7
>> (flet ((+ (x y) "I'm an addition"))
     (+ 10 20))
"I'm an addition"
>> (+ 3 (flet ((+ (x y) (* x y))) (+ 4 5)))
23
>> (+ 1 2 3 4 5 6)
21
```

Evaluator

Example

```
>> (+ 3 4)
7
>> (flet ((+ (x y) "I'm an addition"))
     (+ 10 20))
"I'm an addition"
>> (+ 3 (flet ((+ (x y) (* x y))) (+ 4 5)))
23
>> (+ 1 2 3 4 5 6)
21
>> (let ((* 10))
     (+ * *))
```

Evaluator

Example

```
>> (+ 3 4)
7
>> (flet ((+ (x y) "I'm an addition"))
     (+ 10 20))
"I'm an addition"
>> (+ 3 (flet ((+ (x y) (* x y))) (+ 4 5)))
23
>> (+ 1 2 3 4 5 6)
21
>> (let ((* 10))
     (+ * *))
20
```

Evaluator

Example

```
>> (+ 3 4)
7
>> (flet ((+ (x y) "I'm an addition"))
     (+ 10 20))
"I'm an addition"
>> (+ 3 (flet ((+ (x y) (* x y))) (+ 4 5)))
23
>> (+ 1 2 3 4 5 6)
21
>> (let ((* 10))
     (+ * *))
20
>> (let ((+ *)
         (* +))
     (+ (* 1 2) 3))
```

Evaluator

Example

```
>> (+ 3 4)
7
>> (flet ((+ (x y) "I'm an addition"))
     (+ 10 20))
"I'm an addition"
>> (+ 3 (flet ((+ (x y) (* x y))) (+ 4 5)))
23
>> (+ 1 2 3 4 5 6)
21
>> (let ((* 10))
     (+ * *))
20
>> (let ((+ *)
         (* +))
     (+ (* 1 2) 3))
9
```

Evaluator

Example

```
>> (+ 3 4)
7
>> (flet ((+ (x y) "I'm an addition"))
     (+ 10 20))
"I'm an addition"
>> (+ 3 (flet ((+ (x y) (* x y))) (+ 4 5)))
23
>> (+ 1 2 3 4 5 6)
21
>> (let ((* 10))
     (+ * *))
20
>> (let ((+ *)
         (* +))
     (+ (* 1 2) 3))
9
>> (flet ((+ (x y) (* x y))
          (* (x y) (+ x y)))
      (+ (* 1 2) 3))
```

Evaluator

Example

```
>> (+ 3 4)
7
>> (flet ((+ (x y) "I'm an addition"))
     (+ 10 20))
"I'm an addition"
>> (+ 3 (flet ((+ (x y) (* x y))) (+ 4 5)))
23
>> (+ 1 2 3 4 5 6)
21
>> (let ((* 10))
     (+ * *))
20
>> (let ((+ *)
         (* +))
     (+ (* 1 2) 3))
9
>> (flet ((+ (x y) (* x y))
          (* (x y) (+ x y)))
      (+ (* 1 2) 3))
...infinite loop...
```

Evaluator

Problem

- Naming is an important abstraction tool.
- But there is more than simply attaching a name to something.
- Naming a value is different from naming a function (or is it not?).
- Using a name that abstracts a value is different from using a name that abstracts a function (or is it not?)
- Scope must also be considered.
- Shadowing must also be considered.
- Can a name abstract more than one thing at the same time?
  - He said that that that that man said was false.
  - Buffalo buffalo buffalo buffalo buffalo.
- We will return to this subject later.

Evaluator

Primitives

```
(define initial-bindings
  (list (cons 'pi 3.14159)
        (cons 'e  2.71828)
        (cons 'square (make-function '(x) '((* x x))))
        (cons '+ (make-primitive +))
        (cons '* (make-primitive *))








        ...))
```

More Primitives

Evaluator

Primitives

```
(define initial-bindings
  (list (cons 'pi 3.14159)
        (cons 'e  2.71828)
        (cons 'square (make-function '(x) '((* x x))))
        (cons '+ (make-primitive +))
        (cons '* (make-primitive *))







        ...))
```

More Primitives

- We need more arithmetic operators ...

Evaluator

Primitives

```
(define initial-bindings
  (list (cons 'pi 3.14159)
        (cons 'e  2.71828)
        (cons 'square (make-function '(x) '((* x x))))
        (cons '+ (make-primitive +))
        (cons '* (make-primitive *))
        (cons '- (make-primitive -))
        (cons '/ (make-primitive /))





        ...))
```

More Primitives

- We need more arithmetic operators ...

Evaluator

Primitives

```
(define initial-bindings
  (list (cons 'pi 3.14159)
        (cons 'e  2.71828)
        (cons 'square (make-function '(x) '((* x x))))
        (cons '+ (make-primitive +))
        (cons '* (make-primitive *))
        (cons '- (make-primitive -))
        (cons '/ (make-primitive /))





        ...))
```

More Primitives

- We need more arithmetic operators ...
- ...and also relational operators.

Evaluator

Primitives

```
(define initial-bindings
  (list (cons 'pi 3.14159)
        (cons 'e  2.71828)
        (cons 'square (make-function '(x) '((* x x))))
        (cons '+ (make-primitive +))
        (cons '* (make-primitive *))
        (cons '- (make-primitive -))
        (cons '/ (make-primitive /))
        (cons '= (make-primitive =))
        (cons '< (make-primitive <))
        (cons '> (make-primitive >))
        (cons '<= (make-primitive <=))
        (cons '>= (make-primitive >=))
        ...))
```

More Primitives

- We need more arithmetic operators ...
- ...and also relational operators.

Evaluator

Relational Operators

```
>> (= (- 1 2) (- 3 4))
```

Evaluator

Relational Operators

```
>> (= (- 1 2) (- 3 4))
#t
```

Evaluator

Relational Operators

```
>> (= (- 1 2) (- 3 4))
#t
>> (>= 2 5)
```

Evaluator

Relational Operators

```
>> (= (- 1 2) (- 3 4))
#t
>> (>= 2 5)
#f
```

Evaluator

Relational Operators

```
>> (= (- 1 2) (- 3 4))
#t
>> (>= 2 5)
#f
>> #t
```

Evaluator

Relational Operators

```
>> (= (- 1 2) (- 3 4))
#t
>> (>= 2 5)
#f
>> #t
error in "Unknown expression type -- EVAL": #t
```

Booleans

- Relational operators compute *Boolean* values that we pass to the evaluated language...
- ...but these are not (yet) self-evaluating forms.

Evaluator

Relational Operators

```
>> (= (- 1 2) (- 3 4))
#t
>> (>= 2 5)
#f
>> #t
error in "Unknown expression type -- EVAL": #t
```

Booleans

- Relational operators compute *Boolean* values that we pass to the evaluated language...
- ...but these are not (yet) self-evaluating forms.

Self-Evaluating Forms

```
(define (self-evaluating? exp)
  (or (number? exp) (string? exp)))
```

Evaluator

Relational Operators

```
>> (= (- 1 2) (- 3 4))
#t
>> (>= 2 5)
#f
>> #t
error in "Unknown expression type -- EVAL": #t
```

Booleans

- Relational operators compute *Boolean* values that we pass to the evaluated language...
- ...but these are not (yet) self-evaluating forms.

Self-Evaluating Forms

```
(define (self-evaluating? exp)
  (or (number? exp) (string? exp) (boolean? exp)))
```

Evaluator

Booleans

- What are the uses of boolean values? What are the uses of a data type that only has two members?
- Booleans are good for expressing a choice between two alternatives.
- *Conditional Expressions* allow such choices. They were invented in 1959 (by John McCarthy) and are available in all modern programming languages.

Conditional Expression Example

```
(if (< x y)
  (+ x 5)
  (* y 3))
```

Evaluator

Conditional Expressions

```
;(if (< x y)
;  (+ x 5)
;  (* y 3))

(define (if? exp)
  (and (pair? exp) (eq? (car exp) 'if)))
```

Evaluator

Conditional Expressions

```
;(if (< x y)
;   (+ x 5)
;   (* y 3))

(define (if? exp)
  (and (pair? exp) (eq? (car exp) 'if)))

(define (if-condition exp)
  (cadr exp))
```

Evaluator

Conditional Expressions

```
;(if (< x y)
;  (+ x 5)
;  (* y 3))

(define (if? exp)
  (and (pair? exp) (eq? (car exp) 'if)))

(define (if-condition exp)
  (cadr exp))

(define (if-consequent exp)
  (caddr exp))
```

Evaluator

Conditional Expressions

```
;(if (< x y)
;  (+ x 5)
;  (* y 3))

(define (if? exp)
  (and (pair? exp) (eq? (car exp) 'if)))

(define (if-condition exp)
  (cadr exp))

(define (if-consequent exp)
  (caddr exp))

(define (if-alternative exp)
  (cadddr exp))
```

Evaluator

Conditional Expressions

```
;(if (< x y)
;  (+ x 5)
;  (* y 3))

(define (if? exp)
  (and (pair? exp) (eq? (car exp) 'if)))

(define (if-condition exp)
  (cadr exp))

(define (if-consequent exp)
  (caddr exp))

(define (if-alternative exp)
  (cadddr exp))

(define (eval-if exp env)
```

Evaluator

Conditional Expressions

```
;(if (< x y)
;  (+ x 5)
;  (* y 3))

(define (if? exp)
  (and (pair? exp) (eq? (car exp) 'if)))

(define (if-condition exp)
  (cadr exp))

(define (if-consequent exp)
  (caddr exp))

(define (if-alternative exp)
  (cadddr exp))

(define (eval-if exp env)
  (if (true? (eval (if-condition exp) env))
```

Evaluator

Conditional Expressions

```
;(if (< x y)
;  (+ x 5)
;  (* y 3))

(define (if? exp)
  (and (pair? exp) (eq? (car exp) 'if)))

(define (if-condition exp)
  (cadr exp))

(define (if-consequent exp)
  (caddr exp))

(define (if-alternative exp)
  (cadddr exp))

(define (eval-if exp env)
  (if (true? (eval (if-condition exp) env))
      (eval (if-consequent exp) env)
```

Evaluator

Conditional Expressions

```
;(if (< x y)
;  (+ x 5)
;  (* y 3))

(define (if? exp)
  (and (pair? exp) (eq? (car exp) 'if)))

(define (if-condition exp)
  (cadr exp))

(define (if-consequent exp)
  (caddr exp))

(define (if-alternative exp)
  (cadddr exp))

(define (eval-if exp env)
  (if (true? (eval (if-condition exp) env))
      (eval (if-consequent exp) env)
      (eval (if-alternative exp) env)))
```

Evaluator

Conditional Expression

- The semantics of conditional expressions are directly inherited from the semantics of the underlying Scheme.
- But we can be more specific regarding the possible values of the condition.
- There are several possibilities:
    - The booleans are the only acceptable values (as in Pascal and Java).
    - Any value is accepted: some of them are considered false and all the other values are considered true (as in Python, Javascript and Perl).
    - Any value is accepted: one specific value is false and all the other values are considered true (as in C and Scheme).

Evaluator

Conditional Expressions

```
(define (eval-if exp env)
  (if (true? (eval (if-condition exp) env))
      (eval (if-consequent exp) env)
      (eval (if-alternative exp) env)))
```

Evaluator

Conditional Expressions

```
(define (eval-if exp env)
  (if (true? (eval (if-condition exp) env))
      (eval (if-consequent exp) env)
      (eval (if-alternative exp) env)))

;;Java
(define (true? value)
  (cond ((eq? value #t) #t)
        ((eq? value #f) #f)
        (else
         (error "Incorrect boolean value" value))))
```

Evaluator

Conditional Expressions

```
(define (eval-if exp env)
  (if (true? (eval (if-condition exp) env))
      (eval (if-consequent exp) env)
      (eval (if-alternative exp) env)))

;;Java
(define (true? value)
  (cond ((eq? value #t) #t)
        ((eq? value #f) #f)
        (else
         (error "Incorrect boolean value" value))))

;;Javascript
(define (true? value)
  (not (or (eq? value #f)
           (eqv? value 0)
           (eq? value ""))))
```

Evaluator

Conditional Expressions

```
(define (eval-if exp env)
  (if (true? (eval (if-condition exp) env))
      (eval (if-consequent exp) env)
      (eval (if-alternative exp) env)))

;;Java
(define (true? value)
  (cond ((eq? value #t) #t)
        ((eq? value #f) #f)
        (else
         (error "Incorrect boolean value" value))))

;;Javascript
(define (true? value)
  (not (or (eq? value #f)
           (eqv? value 0)
           (eq? value ""))))

;;Scheme
(define (true? value)
  (not (eq? value #f)))
```

Evaluator

Evaluator

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((name? exp)
         (eval-name exp env))


        ((let? exp)
         (eval-let exp env))
        ((flet? exp)
         (eval-flet exp env))
        ((call? exp)
         (eval-call exp env))
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Evaluator

Evaluator

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((name? exp)
         (eval-name exp env))
        ((if? exp)

        ((let? exp)
         (eval-let exp env))
        ((flet? exp)
         (eval-flet exp env))
        ((call? exp)
         (eval-call exp env))
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Evaluator

Evaluator

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((name? exp)
         (eval-name exp env))
        ((if? exp)
         (eval-if exp env))
        ((let? exp)
         (eval-let exp env))
        ((flet? exp)
         (eval-flet exp env))
        ((call? exp)
         (eval-call exp env))
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Evaluator

Example

```
>> (flet ((abs (x)
            (if (< x 0)
              (- x)
              x)))
     (abs -5))
```

Evaluator

Example

```
>> (flet ((abs (x)
           (if (< x 0)
             (- x)
             x)))
     (abs -5))
5
```

Evaluator

Example

```
>> (flet ((abs (x)
            (if (< x 0)
              (- x)
              x)))
     (abs -5))
5
>> (flet ((fact (n)
            (if (= n 0)
              1
              (* n (fact (- n 1))))))
     (fact 3))
```

Evaluator

Example

```
>> (flet ((abs (x)
            (if (< x 0)
              (- x)
              x)))
     (abs -5))
5
>> (flet ((fact (n)
            (if (= n 0)
              1
              (* n (fact (- n 1))))))
     (fact 3))
6
```

Evaluator

Example

```
>> (flet ((abs (x)
            (if (< x 0)
              (- x)
              x)))
      (abs -5))
5
>> (flet ((fact (n)
            (if (= n 0)
              1
              (* n (fact (- n 1))))))
      (fact 3))
6
>> (flet ((fact (n)
            (if (= n 0)
              1
              (* n (fact (- n 1))))))
      (fact 50))
```

Evaluator

Example

```
>> (flet ((abs (x)
            (if (< x 0)
              (- x)
              x)))
     (abs -5))
5
>> (flet ((fact (n)
            (if (= n 0)
              1
              (* n (fact (- n 1))))))
     (fact 3))
6
>> (flet ((fact (n)
            (if (= n 0)
              1
              (* n (fact (- n 1))))))
     (fact 50))
30414093201713378043612608166064768844377641568960512000000000000
```

Evaluator

Example

```
=> (eval (flet ((fact (n) (if (= n 0) 1 (* n (fact (- n 1)))))) (fact 3)) (…))
```

Evaluator

Example

```
=> (eval (flet ((fact (n) (if (= n 0) 1 (* n (fact (- n 1)))))) (fact 3)) (…))
=>  (eval (fact 3) ((fact . (function (n) (if (= n 0) 1 (* n (fact (- n 1)))))) …))
```

Evaluator

Example

```
=> (eval (flet ((fact (n) (if (= n 0) 1 (* n (fact (- n 1)))))) (fact 3)) (…))
=>  (eval (fact 3) ((fact . (function (n) (if (= n 0) 1 (* n (fact (- n 1)))))) …))
=>   (eval 3 ((fact . (function (n) (if (= n 0) 1 (* n (fact (- n 1)))))) …))
<=   3
```

Evaluator

Example

```
=> (eval (flet ((fact (n) (if (= n 0) 1 (* n (fact (- n 1)))))) (fact 3)) (…))
=>  (eval (fact 3) ((fact . (function (n) (if (= n 0) 1 (* n (fact (- n 1)))))) …))
=>   (eval 3 ((fact . (function (n) (if (= n 0) 1 (* n (fact (- n 1)))))) …))
<=   3
=>   (eval (if (= n 0) 1 (* n (fact (- n 1)))) ((n . 3) (fact . λ) …))
```

Evaluator

Example

```
=> (eval (flet ((fact (n) (if (= n 0) 1 (* n (fact (- n 1)))))) (fact 3)) (…))
=>  (eval (fact 3) ((fact . (function (n) (if (= n 0) 1 (* n (fact (- n 1)))))) …))
=>   (eval 3 ((fact . (function (n) (if (= n 0) 1 (* n (fact (- n 1)))))) …))
<=   3
=>   (eval (if (= n 0) 1 (* n (fact (- n 1)))) ((n . 3) (fact . λ) …))
=>    (eval (= n 0) ((n . 3) (fact . λ) …))
```

Evaluator

Example

```
=> (eval (flet ((fact (n) (if (= n 0) 1 (* n (fact (- n 1)))))) (fact 3)) (…))
=>  (eval (fact 3) ((fact . (function (n) (if (= n 0) 1 (* n (fact (- n 1)))))) …))
=>   (eval 3 ((fact . (function (n) (if (= n 0) 1 (* n (fact (- n 1)))))) …))
<=   3
=>   (eval (if (= n 0) 1 (* n (fact (- n 1)))) ((n . 3) (fact . λ) …))
=>    (eval (= n 0) ((n . 3) (fact . λ) …))
=>     (eval n ((n . 3) (fact . λ) …))
<=     3
=>     (eval 0 ((n . 3) (fact . λ) …))
<=     0
<=    #f
```

Evaluator

Example

```
=> (eval (flet ((fact (n) (if (= n 0) 1 (* n (fact (- n 1)))))) (fact 3)) (…))
=>  (eval (fact 3) ((fact . (function (n) (if (= n 0) 1 (* n (fact (- n 1)))))) …))
=>   (eval 3 ((fact . (function (n) (if (= n 0) 1 (* n (fact (- n 1)))))) …))
<=   3
=>   (eval (if (= n 0) 1 (* n (fact (- n 1)))) ((n . 3) (fact . λ) …))
=>    (eval (= n 0) ((n . 3) (fact . λ) …))
=>     (eval n ((n . 3) (fact . λ) …))
<=     3
=>     (eval 0 ((n . 3) (fact . λ) …))
<=     0
<=    #f
=>    (eval (* n (fact (- n 1))) ((n . 3) (fact . λ) …))
=>     (eval n ((n . 3) (fact . λ) …))
<=     3
=>     (eval (fact (- n 1)) ((n . 3) (fact . λ) …))
=>      (eval (- n 1) ((n . 3) (fact . λ) …))
=>       (eval n ((n . 3) (fact . λ) …))
<=       3
=>       (eval 1 ((n . 3) (fact . λ) …))
<=       1
<=      2
```

Evaluator

Example

```
=> (eval (flet ((fact (n) (if (= n 0) 1 (* n (fact (- n 1)))))) (fact 3)) (…))
=>  (eval (fact 3) ((fact . (function (n) (if (= n 0) 1 (* n (fact (- n 1)))))) …))
=>   (eval 3 ((fact . (function (n) (if (= n 0) 1 (* n (fact (- n 1)))))) …))
<=   3
=>   (eval (if (= n 0) 1 (* n (fact (- n 1)))) ((n . 3) (fact . λ) …))
=>    (eval (= n 0) ((n . 3) (fact . λ) …))
=>     (eval n ((n . 3) (fact . λ) …))
<=     3
=>     (eval 0 ((n . 3) (fact . λ) …))
<=     0
<=    #f
=>    (eval (* n (fact (- n 1))) ((n . 3) (fact . λ) …))
=>     (eval n ((n . 3) (fact . λ) …))
<=     3
=>     (eval (fact (- n 1)) ((n . 3) (fact . λ) …))
=>      (eval (- n 1) ((n . 3) (fact . λ) …))
=>       (eval n ((n . 3) (fact . λ) …))
<=       3
=>       (eval 1 ((n . 3) (fact . λ) …))
<=       1
<=      2
=>      (eval (if (= n 0) 1 (* n (fact (- n 1)))) ((n . 2) (n . 3) (fact . λ) …))
=>       (eval (= n 0) ((n . 2) (n . 3) (fact . λ) …))
=>        (eval n ((n . 2) (n . 3) (fact . λ) …))
<=        2
=>        (eval 0 ((n . 2) (n . 3) (fact . λ) …))
<=        0
<=       #f
=>       (eval (* n (fact (- n 1))) ((n . 2) (n . 3) (fact . λ) …))
=>        (eval n ((n . 2) (n . 3) (fact . λ) …))
<=        2
=>        (eval (fact (- n 1)) ((n . 2) (n . 3) (fact . λ) …))
=>         (eval (- n 1) ((n . 2) (n . 3) (fact . λ) …))
```

Evaluator

Example

```
=>       (eval n ((n . 2) (n . 3) (fact . λ) …))
<=       2
=>       (eval 1 ((n . 2) (n . 3) (fact . λ) …))
<=       1
<=      1
```

Evaluator

Example

```
=>        (eval n ((n . 2) (n . 3) (fact . λ) …))
<=        2
=>        (eval 1 ((n . 2) (n . 3) (fact . λ) …))
<=        1
<=       1
=>       (eval (if (= n 0) 1 (* n (fact (- n 1)))) ((n . 1) (n . 2) (n . 3) (fact . λ) …))
=>        (eval (= n 0) ((n . 1) (n . 2) (n . 3) (fact . λ) …))
=>         (eval n ((n . 1) (n . 2) (n . 3) (fact . λ) …))
<=         1
=>         (eval 0 ((n . 1) (n . 2) (n . 3) (fact . λ) …))
<=         0
<=        #f
=>        (eval (* n (fact (- n 1))) ((n . 1) (n . 2) (n . 3) (fact . λ) …))
=>         (eval n ((n . 1) (n . 2) (n . 3) (fact . λ) …))
<=         1
=>         (eval (fact (- n 1)) ((n . 1) (n . 2) (n . 3) (fact . λ) …))
=>          (eval (- n 1) ((n . 1) (n . 2) (n . 3) (fact . λ) …))
=>           (eval n ((n . 1) (n . 2) (n . 3) (fact . λ) …))
<=           1
=>           (eval 1 ((n . 1) (n . 2) (n . 3) (fact . λ) …))
<=           1
<=          0
```

Evaluator

Example

```
=>      (eval n ((n . 2) (n . 3) (fact . λ) …))
<=      2
=>      (eval 1 ((n . 2) (n . 3) (fact . λ) …))
<=      1
<=     1
=>     (eval (if (= n 0) 1 (* n (fact (- n 1)))) ((n . 1) (n . 2) (n . 3) (fact . λ) …))
=>      (eval (= n 0) ((n . 1) (n . 2) (n . 3) (fact . λ) …))
=>       (eval n ((n . 1) (n . 2) (n . 3) (fact . λ) …))
<=       1
=>       (eval 0 ((n . 1) (n . 2) (n . 3) (fact . λ) …))
<=       0
<=      #f
=>      (eval (* n (fact (- n 1))) ((n . 1) (n . 2) (n . 3) (fact . λ) …))
=>       (eval n ((n . 1) (n . 2) (n . 3) (fact . λ) …))
<=       1
=>       (eval (fact (- n 1)) ((n . 1) (n . 2) (n . 3) (fact . λ) …))
=>        (eval (- n 1) ((n . 1) (n . 2) (n . 3) (fact . λ) …))
=>         (eval n ((n . 1) (n . 2) (n . 3) (fact . λ) …))
<=         1
=>         (eval 1 ((n . 1) (n . 2) (n . 3) (fact . λ) …))
<=         1
<=        0
=>        (eval (if (= n 0) 1 (* n (fact (- n 1)))) ((n . 0) (n . 1) (n . 2) (n . 3) (fact . λ) …))
```

Evaluator

Example

```
=>        (eval n ((n . 2) (n . 3) (fact . λ) …))
<=        2
=>        (eval 1 ((n . 2) (n . 3) (fact . λ) …))
<=        1
<=       1
=>       (eval (if (= n 0) 1 (* n (fact (- n 1)))) ((n . 1) (n . 2) (n . 3) (fact . λ) …))
=>        (eval (= n 0) ((n . 1) (n . 2) (n . 3) (fact . λ) …))
=>         (eval n ((n . 1) (n . 2) (n . 3) (fact . λ) …))
<=         1
=>         (eval 0 ((n . 1) (n . 2) (n . 3) (fact . λ) …))
<=         0
<=        #f
=>        (eval (* n (fact (- n 1))) ((n . 1) (n . 2) (n . 3) (fact . λ) …))
=>         (eval n ((n . 1) (n . 2) (n . 3) (fact . λ) …))
<=         1
=>         (eval (fact (- n 1)) ((n . 1) (n . 2) (n . 3) (fact . λ) …))
=>          (eval (- n 1) ((n . 1) (n . 2) (n . 3) (fact . λ) …))
=>           (eval n ((n . 1) (n . 2) (n . 3) (fact . λ) …))
<=           1
=>           (eval 1 ((n . 1) (n . 2) (n . 3) (fact . λ) …))
<=           1
<=          0
=>          (eval (if (= n 0) 1 (* n (fact (- n 1)))) ((n . 0) (n . 1) (n . 2) (n . 3) (fact . λ) …))
=>           (eval (= n 0) ((n . 0) (n . 1) (n . 2) (n . 3) (fact . λ) …))
```

Evaluator

Example

```
=>        (eval n ((n . 2) (n . 3) (fact . λ) …))
<=        2
=>        (eval 1 ((n . 2) (n . 3) (fact . λ) …))
<=        1
<=       1
=>       (eval (if (= n 0) 1 (* n (fact (- n 1)))) ((n . 1) (n . 2) (n . 3) (fact . λ) …))
=>        (eval (= n 0) ((n . 1) (n . 2) (n . 3) (fact . λ) …))
=>         (eval n ((n . 1) (n . 2) (n . 3) (fact . λ) …))
<=         1
=>         (eval 0 ((n . 1) (n . 2) (n . 3) (fact . λ) …))
<=         0
<=        #f
=>        (eval (* n (fact (- n 1))) ((n . 1) (n . 2) (n . 3) (fact . λ) …))
=>         (eval n ((n . 1) (n . 2) (n . 3) (fact . λ) …))
<=         1
=>         (eval (fact (- n 1)) ((n . 1) (n . 2) (n . 3) (fact . λ) …))
=>          (eval (- n 1) ((n . 1) (n . 2) (n . 3) (fact . λ) …))
=>           (eval n ((n . 1) (n . 2) (n . 3) (fact . λ) …))
<=           1
=>           (eval 1 ((n . 1) (n . 2) (n . 3) (fact . λ) …))
<=           1
<=          0
=>          (eval (if (= n 0) 1 (* n (fact (- n 1)))) ((n . 0) (n . 1) (n . 2) (n . 3) (fact . λ) …))
=>           (eval (= n 0) ((n . 0) (n . 1) (n . 2) (n . 3) (fact . λ) …))
=>            (eval n ((n . 0) (n . 1) (n . 2) (n . 3) (fact . λ) …))
<=            0
=>            (eval 0 ((n . 0) (n . 1) (n . 2) (n . 3) (fact . λ) …))
<=            0
<=           #t
```

Evaluator

Example

```
=>      (eval n ((n . 2) (n . 3) (fact . λ) …))
<=      2
=>      (eval 1 ((n . 2) (n . 3) (fact . λ) …))
<=      1
<=     1
=>     (eval (if (= n 0) 1 (* n (fact (- n 1)))) ((n . 1) (n . 2) (n . 3) (fact . λ) …))
=>      (eval (= n 0) ((n . 1) (n . 2) (n . 3) (fact . λ) …))
=>       (eval n ((n . 1) (n . 2) (n . 3) (fact . λ) …))
<=       1
=>       (eval 0 ((n . 1) (n . 2) (n . 3) (fact . λ) …))
<=       0
<=      #f
=>      (eval (* n (fact (- n 1))) ((n . 1) (n . 2) (n . 3) (fact . λ) …))
=>       (eval n ((n . 1) (n . 2) (n . 3) (fact . λ) …))
<=       1
=>       (eval (fact (- n 1)) ((n . 1) (n . 2) (n . 3) (fact . λ) …))
=>        (eval (- n 1) ((n . 1) (n . 2) (n . 3) (fact . λ) …))
=>         (eval n ((n . 1) (n . 2) (n . 3) (fact . λ) …))
<=         1
=>         (eval 1 ((n . 1) (n . 2) (n . 3) (fact . λ) …))
<=         1
<=        0
=>        (eval (if (= n 0) 1 (* n (fact (- n 1)))) ((n . 0) (n . 1) (n . 2) (n . 3) (fact . λ) …))
=>         (eval (= n 0) ((n . 0) (n . 1) (n . 2) (n . 3) (fact . λ) …))
=>          (eval n ((n . 0) (n . 1) (n . 2) (n . 3) (fact . λ) …))
<=          0
=>          (eval 0 ((n . 0) (n . 1) (n . 2) (n . 3) (fact . λ) …))
<=          0
<=         #t
=>         (eval 1 ((n . 0) (n . 1) (n . 2) (n . 3) (fact . λ) …))
<=         1
```

Evaluator

Example

```
=>      (eval n ((n . 2) (n . 3) (fact . λ) …))
<=      2
=>      (eval 1 ((n . 2) (n . 3) (fact . λ) …))
<=      1
<=     1
=>     (eval (if (= n 0) 1 (* n (fact (- n 1)))) ((n . 1) (n . 2) (n . 3) (fact . λ) …))
=>      (eval (= n 0) ((n . 1) (n . 2) (n . 3) (fact . λ) …))
=>       (eval n ((n . 1) (n . 2) (n . 3) (fact . λ) …))
<=       1
=>       (eval 0 ((n . 1) (n . 2) (n . 3) (fact . λ) …))
<=       0
<=      #f
=>      (eval (* n (fact (- n 1))) ((n . 1) (n . 2) (n . 3) (fact . λ) …))
=>       (eval n ((n . 1) (n . 2) (n . 3) (fact . λ) …))
<=       1
=>       (eval (fact (- n 1)) ((n . 1) (n . 2) (n . 3) (fact . λ) …))
=>        (eval (- n 1) ((n . 1) (n . 2) (n . 3) (fact . λ) …))
=>         (eval n ((n . 1) (n . 2) (n . 3) (fact . λ) …))
<=         1
=>         (eval 1 ((n . 1) (n . 2) (n . 3) (fact . λ) …))
<=         1
<=        0
=>        (eval (if (= n 0) 1 (* n (fact (- n 1)))) ((n . 0) (n . 1) (n . 2) (n . 3) (fact . λ) …))
=>         (eval (= n 0) ((n . 0) (n . 1) (n . 2) (n . 3) (fact . λ) …))
=>          (eval n ((n . 0) (n . 1) (n . 2) (n . 3) (fact . λ) …))
<=          0
=>          (eval 0 ((n . 0) (n . 1) (n . 2) (n . 3) (fact . λ) …))
<=          0
<=         #t
=>         (eval 1 ((n . 0) (n . 1) (n . 2) (n . 3) (fact . λ) …))
<=         1
<=    …
<= 6
```

Evaluator

Definitions

- So far, our environments are *functional*: they are created, used and discarded.
- This means that the REPL has no memory of previous bindings.
- It would be good to have an *incremental* scope for names.

Example

Evaluator

Definitions

- So far, our environments are *functional*: they are created, used and discarded.
- This means that the REPL has no memory of previous bindings.
- It would be good to have an *incremental* scope for names.

Example

```
>> (def x (+ 1 2))
```

Evaluator

Definitions

- So far, our environments are *functional*: they are created, used and discarded.
- This means that the REPL has no memory of previous bindings.
- It would be good to have an *incremental* scope for names.

Example

```
>> (def x (+ 1 2))
3
```

Evaluator

Definitions

- So far, our environments are *functional*: they are created, used and discarded.
- This means that the REPL has no memory of previous bindings.
- It would be good to have an *incremental* scope for names.

Example

```
>> (def x (+ 1 2))
3
>> (+ x 2)
```

Evaluator

Definitions

- So far, our environments are *functional*: they are created, used and discarded.
- This means that the REPL has no memory of previous bindings.
- It would be good to have an *incremental* scope for names.

Example

```
>> (def x (+ 1 2))
3
>> (+ x 2)
5
```

Evaluator

Definitions

- So far, our environments are *functional*: they are created, used and discarded.
- This means that the REPL has no memory of previous bindings.
- It would be good to have an *incremental* scope for names.

Example

```
>> (def x (+ 1 2))
3
>> (+ x 2)
5
>> (fdef triple (a) (+ a a a))
```

Evaluator

Definitions

- So far, our environments are *functional*: they are created, used and discarded.
- This means that the REPL has no memory of previous bindings.
- It would be good to have an *incremental* scope for names.

Example

```
>> (def x (+ 1 2))
3
>> (+ x 2)
5
>> (fdef triple (a) (+ a a a))
(function (a) (+ a a a))
```

Evaluator

Definitions

- So far, our environments are *functional*: they are created, used and discarded.
- This means that the REPL has no memory of previous bindings.
- It would be good to have an *incremental* scope for names.

Example

```
>> (def x (+ 1 2))
3
>> (+ x 2)
5
>> (fdef triple (a) (+ a a a))
(function (a) (+ a a a))
>> (triple (+ x 3))
```

Evaluator

Definitions

- So far, our environments are *functional*: they are created, used and discarded.
- This means that the REPL has no memory of previous bindings.
- It would be good to have an *incremental* scope for names.

Example

```
>> (def x (+ 1 2))
3
>> (+ x 2)
5
>> (fdef triple (a) (+ a a a))
(function (a) (+ a a a))
>> (triple (+ x 3))
18
```

Evaluator

Definitions

```
;(def x (+ 1 2))
```

Evaluator

Definitions

```
;(def x (+ 1 2))

(define (def? exp)
  (and (pair? exp) (eq? (car exp) 'def)))
```

Evaluator

Definitions

```
;(def x (+ 1 2))

(define (def? exp)
  (and (pair? exp) (eq? (car exp) 'def)))

(define (def-name exp)
  (cadr exp))
```

Evaluator

Definitions

```
;(def x (+ 1 2))

(define (def? exp)
  (and (pair? exp) (eq? (car exp) 'def)))

(define (def-name exp)
  (cadr exp))

(define (def-init exp)
  (caddr exp))
```

Evaluator

Definitions

```
;(def x (+ 1 2))

(define (def? exp)
  (and (pair? exp) (eq? (car exp) 'def)))

(define (def-name exp)
  (cadr exp))

(define (def-init exp)
  (caddr exp))

(define (eval-def exp env)
  (let ((value (eval (def-init exp) env)))
    (define-name! (def-name exp) value env)
    value))
```

Evaluator

Definitions

```
;(def x (+ 1 2))

(define (def? exp)
  (and (pair? exp) (eq? (car exp) 'def)))

(define (def-name exp)
  (cadr exp))

(define (def-init exp)
  (caddr exp))

(define (eval-def exp env)
  (let ((value (eval (def-init exp) env)))
    (define-name! (def-name exp) value env)
    value))

(define (define-name! name value env)
  (let ((binding (cons name value))
        (new-pair (cons (car env) (cdr env))))
    (set-car! env binding)
    (set-cdr! env new-pair)))
```

Evaluator

Definitions

```
;(fdef triple (a) (+ a a a))
```

Evaluator

Definitions

```
;(fdef triple (a) (+ a a a))

(define (fdef? exp)
  (and (pair? exp) (eq? (car exp) 'fdef)))
```

Evaluator

Definitions

```
;(fdef triple (a) (+ a a a))

(define (fdef? exp)
  (and (pair? exp) (eq? (car exp) 'fdef)))

(define (fdef-name exp)
  (cadr exp))
```

Evaluator

Definitions

```
;(fdef triple (a) (+ a a a))

(define (fdef? exp)
  (and (pair? exp) (eq? (car exp) 'fdef)))

(define (fdef-name exp)
  (cadr exp))

(define (fdef-parameters exp)
  (caddr exp))
```

Evaluator

Definitions

```
;(fdef triple (a) (+ a a a))

(define (fdef? exp)
  (and (pair? exp) (eq? (car exp) 'fdef)))

(define (fdef-name exp)
  (cadr exp))

(define (fdef-parameters exp)
  (caddr exp))

(define (fdef-body exp)
  (cdddr exp))
```

Evaluator

Definitions

```
;(fdef triple (a) (+ a a a))

(define (fdef? exp)
  (and (pair? exp) (eq? (car exp) 'fdef)))

(define (fdef-name exp)
  (cadr exp))

(define (fdef-parameters exp)
  (caddr exp))

(define (fdef-body exp)
  (cdddr exp))

(define (eval-fdef exp env)
  (let ((value
         (make-function (fdef-parameters exp)
                        (fdef-body exp))))
    (define-name! (fdef-name exp) value env)
    value))
```

Evaluator

Evaluator

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((name? exp)
         (eval-name exp env))
        ((if? exp)
         (eval-if exp env))
        ((let? exp)
         (eval-let exp env))
        ((flet? exp)
         (eval-flet exp env))




        ((call? exp)
         (eval-call exp env))
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Evaluator

Evaluator

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((name? exp)
         (eval-name exp env))
        ((if? exp)
         (eval-if exp env))
        ((let? exp)
         (eval-let exp env))
        ((flet? exp)
         (eval-flet exp env))
        ((def? exp)



        ((call? exp)
         (eval-call exp env))
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Evaluator

Evaluator

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((name? exp)
         (eval-name exp env))
        ((if? exp)
         (eval-if exp env))
        ((let? exp)
         (eval-let exp env))
        ((flet? exp)
         (eval-flet exp env))
        ((def? exp)
         (eval-def exp env))


        ((call? exp)
         (eval-call exp env))
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Evaluator

Evaluator

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((name? exp)
         (eval-name exp env))
        ((if? exp)
         (eval-if exp env))
        ((let? exp)
         (eval-let exp env))
        ((flet? exp)
         (eval-flet exp env))
        ((def? exp)
         (eval-def exp env))
        ((fdef? exp)

        ((call? exp)
         (eval-call exp env))
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Evaluator

Evaluator

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((name? exp)
         (eval-name exp env))
        ((if? exp)
         (eval-if exp env))
        ((let? exp)
         (eval-let exp env))
        ((flet? exp)
         (eval-flet exp env))
        ((def? exp)
         (eval-def exp env))
        ((fdef? exp)
         (eval-fdef exp env))
        ((call? exp)
         (eval-call exp env))
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Evaluator

Example

```
>> (def foo 1)
```

Evaluator

Example

```
>> (def foo 1)
1
```

Evaluator

Example

```
>> (def foo 1)
1
>> (+ foo (def foo 2))
```

Evaluator

Example

```
>> (def foo 1)
1
>> (+ foo (def foo 2))
3   ;;But it could have been 4
```

Evaluator

Example

```
>> (def foo 1)
1
>> (+ foo (def foo 2))
3  ;;But it could have been 4
>> (def bar 1)
```

Evaluator

Example

```
>> (def foo 1)
1
>> (+ foo (def foo 2))
3  ;;But it could have been 4
>> (def bar 1)
1
```

Evaluator

Example

```
>> (def foo 1)
1
>> (+ foo (def foo 2))
3  ;;But it could have been 4
>> (def bar 1)
1
>> (+ (def bar 2) bar)
```

Evaluator

Example

```
>> (def foo 1)
1
>> (+ foo (def foo 2))
3  ;;But it could have been 4
>> (def bar 1)
1
>> (+ (def bar 2) bar)
4  ;;But it could have been 3
```

Definitions

- The evaluation of a definition *modifies* the current environment.
- The modification occurs as a *side-effect*.
- Understanding side-effects requires an *evaluation order*:
  - Letf-to-right in Java and Common Lisp.
  - Right-to-left in APL and J.
  - Unspecified in Scheme, C, Pascal, and many others.

Evaluator

Example (with left-to-right evaluation order)

```
>> (def baz 3)
3
```

Evaluator

Example (with left-to-right evaluation order)

```
>> (def baz 3)
3
>> (+ (let ((x 0))
        (def baz 5))
      baz)
```

Evaluator

Example (with left-to-right evaluation order)

```
>> (def baz 3)
3
>> (+ (let ((x 0))
        (def baz 5))
      baz)
8
```

Evaluator

Example (with left-to-right evaluation order)

```
>> (def baz 3)
3
>> (+ (let ((x 0))
        (def baz 5))
      baz)
8
>> (+ (let ()
        (def baz 6))
      baz)
```

Evaluator

Example (with left-to-right evaluation order)

```
>> (def baz 3)
3
>> (+ (let ((x 0))
        (def baz 5))
      baz)
8
>> (+ (let ()
        (def baz 6))
      baz)
12   ;;Huuh? Shouldn't it be 9?
```

Problem

- An “empty” scope should not be different from a “normal” scope.

Evaluator

Environments

- In our current implementation, the environment is implemented as a continuous sequence of bindings without separation between different scopes.
- To add such separation, we need to partition that sequence into *frames*.
- Each time we augment the environment (i.e., for every `let`, `flet` and function call), we create a new frame.
- Each frame contains an association list for the bindings.
- This change require us to redefine:
  - the process of finding bindinds: `eval-name`
  - the process of augmenting the environment: `augment-environment`
  - the process of updating the environment: `define-name!`

Evaluator

Environments as Sequences of Frames

```
(define (augment-environment names values env)
  (if (null? names)
      env
      (cons (cons (car names) (car values))
            (augment-environment (cdr names) (cdr values) env))))
```

Evaluator

Environments as Sequences of Frames

```
(define (augment-environment names values env)
  (cons (map cons names values) env))
```

Evaluator

Environments as Sequences of Frames

```
(define (augment-environment names values env)
  (cons (map cons names values) env))

(define (eval-name name env)
  (cond ((null? env)
         (error "Unbound name -- EVAL-NAME" name))
        ((eq? name (caar env))
         (cdar env))
        (else
         (eval-name name (cdr env)))))
```

Evaluator

Environments as Sequences of Frames

```
(define (augment-environment names values env)
  (cons (map cons names values) env))

(define (eval-name name env)
  (define (lookup-in-frame frame)
    (cond ((null? frame)
           (eval-name name (cdr env)))
          ((eq? name (caar frame))
           (cdar frame))
          (else
           (lookup-in-frame (cdr frame)))))
  (if (null? env)
      (error "Unbound name -- EVAL-NAME" name)
      (lookup-in-frame (car env))))
```

Evaluator

Environments as Sequences of Frames

```
(define (augment-environment names values env)
  (cons (map cons names values) env))

(define (eval-name name env)
  (define (lookup-in-frame frame)
    (cond ((null? frame)
           (eval-name name (cdr env)))
          ((eq? name (caar frame))
           (cdar frame))
          (else
           (lookup-in-frame (cdr frame)))))
  (if (null? env)
      (error "Unbound name -- EVAL-NAME" name)
      (lookup-in-frame (car env))))

(define (define-name! name value env)
  (let ((binding (cons name value))
        (new-pair (cons (car env) (cdr env))))
    (set-car! env binding)
    (set-cdr! env new-pair)))
```

Evaluator

Environments as Sequences of Frames

```
(define (augment-environment names values env)
  (cons (map cons names values) env))

(define (eval-name name env)
  (define (lookup-in-frame frame)
    (cond ((null? frame)
           (eval-name name (cdr env)))
          ((eq? name (caar frame))
           (cdar frame))
          (else
           (lookup-in-frame (cdr frame)))))
  (if (null? env)
      (error "Unbound name -- EVAL-NAME" name)
      (lookup-in-frame (car env))))

(define (define-name! name value env)
  (let ((binding (cons name value)))
    (set-car! env (cons binding (car env)))))
```

Evaluator

Example

```
>> (def baz 3)
3
```

Evaluator

Example

```
>> (def baz 3)
3
>> (+ (let ((x 0))
         (def baz 5))
      baz)
```

Evaluator

Example

```
>> (def baz 3)
3
>> (+ (let ((x 0))
        (def baz 5))
      baz)
8
```

Evaluator

Example

```
>> (def baz 3)
3
>> (+ (let ((x 0))
        (def baz 5))
      baz)
8
>> (+ (let ()
        (def baz 6))
      baz)
```

Evaluator

Example

```
>> (def baz 3)
3
>> (+ (let ((x 0))
        (def baz 5))
      baz)
8
>> (+ (let ()
        (def baz 6))
      baz)
9    ;;Good, it's fine now!
```

Evaluator

Example

```
>> (def baz 3)
3
>> (+ (let ((x 0))
        (def baz 5))
      baz)
8
>> (+ (let ()
        (def baz 6))
      baz)
9   ;;Good, it's fine now!
>> (def counter 0)
0
```

Evaluator

Example

```
>> (def baz 3)
3
>> (+ (let ((x 0))
        (def baz 5))
      baz)
8
>> (+ (let ()
        (def baz 6))
      baz)
9   ;;Good, it's fine now!
>> (def counter 0)
0
>> (fdef incr ()
     (def counter (+ counter 1)))
```

Evaluator

Example

```
>> (def baz 3)
3
>> (+ (let ((x 0))
        (def baz 5))
      baz)
8
>> (+ (let ()
        (def baz 6))
      baz)
9   ;;Good, it's fine now!
>> (def counter 0)
0
>> (fdef incr ()
     (def counter (+ counter 1)))
(function () (def counter (+ counter 1)))
```

Evaluator

Example

```
>> (def baz 3)
3
>> (+ (let ((x 0))
        (def baz 5))
      baz)
8
>> (+ (let ()
        (def baz 6))
      baz)
9   ;;Good, it's fine now!
>> (def counter 0)
0
>> (fdef incr ()
     (def counter (+ counter 1)))
(function () (def counter (+ counter 1)))
>> (incr)
```

Evaluator

Example

```
>> (def baz 3)
3
>> (+ (let ((x 0))
        (def baz 5))
      baz)
8
>> (+ (let ()
        (def baz 6))
      baz)
9   ;;Good, it's fine now!
>> (def counter 0)
0
>> (fdef incr ()
     (def counter (+ counter 1)))
(function () (def counter (+ counter 1)))
>> (incr)
1
```

Evaluator

Example

```
>> (def baz 3)
3
>> (+ (let ((x 0))
        (def baz 5))
      baz)
8
>> (+ (let ()
        (def baz 6))
      baz)
9   ;;Good, it's fine now!
>> (def counter 0)
0
>> (fdef incr ()
     (def counter (+ counter 1)))
(function () (def counter (+ counter 1)))
>> (incr)
1
>> (incr)
```

Evaluator

Example

```
>> (def baz 3)
3
>> (+ (let ((x 0))
        (def baz 5))
      baz)
8
>> (+ (let ()
        (def baz 6))
      baz)
9   ;;Good, it's fine now!
>> (def counter 0)
0
>> (fdef incr ()
     (def counter (+ counter 1)))
(function () (def counter (+ counter 1)))
>> (incr)
1
>> (incr)
1 ;;What?
```

Evaluator

Assignments

- A scope protects names: all internally defined names are invisible to the outside.
- But sometimes we want to change a name defined in an outer scope.
- This requires the concept of *assignment*.
- Instead of defining a new binding, an assignment updates an existing binding.
- Concrete syntax: `(set!` *name expression*`)`

Evaluator

Assignments

```
;; (set! counter (+ counter 1))
```

Evaluator

Assignments

```
;; (set! counter (+ counter 1))

(define (set? exp)
  (and (pair? exp) (eq? (car exp) 'set!)))
```

Evaluator

Assignments

```
;; (set! counter (+ counter 1))

(define (set? exp)
  (and (pair? exp) (eq? (car exp) 'set!)))

(define (assignment-name exp)
  (cadr exp))
```

Evaluator

Assignments

```
;; (set! counter (+ counter 1))

(define (set? exp)
  (and (pair? exp) (eq? (car exp) 'set!)))

(define (assignment-name exp)
  (cadr exp))

(define (assignment-expression exp)
  (caddr exp))
```

Evaluator

Assignments

```
;; (set! counter (+ counter 1))

(define (set? exp)
  (and (pair? exp) (eq? (car exp) 'set!)))

(define (assignment-name exp)
  (cadr exp))

(define (assignment-expression exp)
  (caddr exp))

(define (eval-set exp env)
  (let ((value (eval (assignment-expression exp) env)))
    (update-name! (assignment-name exp) value env)
    value))
```

Analogy

- To update a name we must locate its binding but that's what the evaluation of the name does.

Evaluator

Evaluating a Name

```
(define (eval-name name env)
  (define (lookup-in-frame frame)
    (cond ((null? frame)
           (eval-name name (cdr env)))
          ((eq? name (caar frame))
           (cdar frame))
          (else
           (lookup-in-frame (cdr frame)))))
  (if (null? env)
      (error "Unbound name -- EVAL-NAME" name)
      (lookup-in-frame (car env))))
```

Evaluator

Updating a Name

```
(define (update-name! name value env)
  (define (update-in-frame frame)
    (cond ((null? frame)
           (update-name! name value (cdr env)))
          ((eq? name (caar frame))
           (set-cdr! (car frame) value))
          (else
           (update-in-frame (cdr frame)))))
  (if (null? env)
      (error "Unbound name -- EVAL-SET" name)
      (update-in-frame (car env))))
```

Evaluator

Evaluator

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((name? exp)
         (eval-name exp env))
        ((if? exp)
         (eval-if exp env))
        ((let? exp)
         (eval-let exp env))
        ((flet? exp)
         (eval-flet exp env))
        ((def? exp)
         (eval-def exp env))


        ((fdef? exp)
         (eval-fdef exp env))
        ((call? exp)
         (eval-call exp env))
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Evaluator

Evaluator

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((name? exp)
         (eval-name exp env))
        ((if? exp)
         (eval-if exp env))
        ((let? exp)
         (eval-let exp env))
        ((flet? exp)
         (eval-flet exp env))
        ((def? exp)
         (eval-def exp env))
        ((set? exp)

        ((fdef? exp)
         (eval-fdef exp env))
        ((call? exp)
         (eval-call exp env))
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Evaluator

Evaluator

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((name? exp)
         (eval-name exp env))
        ((if? exp)
         (eval-if exp env))
        ((let? exp)
         (eval-let exp env))
        ((flet? exp)
         (eval-flet exp env))
        ((def? exp)
         (eval-def exp env))
        ((set? exp)
         (eval-set exp env))
        ((fdef? exp)
         (eval-fdef exp env))
        ((call? exp)
         (eval-call exp env))
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Evaluator

Example

```
>> (def counter 0)
0
>> (fdef incr ()
     (set! counter (+ counter 1)))
(function () (set! counter (+ counter 1)))
```

Evaluator

Example

```
>> (def counter 0)
0
>> (fdef incr ()
     (set! counter (+ counter 1)))
(function () (set! counter (+ counter 1)))
>> (incr)
```

Evaluator

Example

```
>> (def counter 0)
0
>> (fdef incr ()
     (set! counter (+ counter 1)))
(function () (set! counter (+ counter 1)))
>> (incr)
1
```

Evaluator

Example

```
>> (def counter 0)
0
>> (fdef incr ()
     (set! counter (+ counter 1)))
(function () (set! counter (+ counter 1)))
>> (incr)
1
>> (incr)
```

Evaluator

Example

```
>> (def counter 0)
0
>> (fdef incr ()
     (set! counter (+ counter 1)))
(function () (set! counter (+ counter 1)))
>> (incr)
1
>> (incr)
2
```

Evaluator

Example

```
>> (def counter 0)
0
>> (fdef incr ()
     (set! counter (+ counter 1)))
(function () (set! counter (+ counter 1)))
>> (incr)
1
>> (incr)
2
>> (incr)
```

Evaluator

Example

```
>> (def counter 0)
0
>> (fdef incr ()
     (set! counter (+ counter 1)))
(function () (set! counter (+ counter 1)))
>> (incr)
1
>> (incr)
2
>> (incr)
3
```

Evaluator

Input/Output

- Besides assignments, there is another important source of side-effects: input/output.
- We need to include input-output operations in our evaluator.
- They must be implemented using operating system primitives.

Input/Output

```
(define initial-bindings
  (list (cons 'pi 3.14159)
        ...
        (cons '<= (make-primitive <=))
        (cons '>= (make-primitive >=))
        (cons 'display (make-primitive display))
        (cons 'newline (make-primitive newline))
        (cons 'read (make-primitive read))))
```

Evaluator

Example

```
(fdef print-value (text value)

    (display text)
    (display " = ")
    (display value)
    (newline)
    value))
```

Evaluator

Example

```
(fdef print-value (text value)
  (evaluate-all-and-return-the-last-one
    (display text)
    (display " = ")
    (display value)
    (newline)
    value))
```

Evaluator

Example

```
(fdef print-value (text value)
  (evaluate-all-and-return-the-last-one
    (display text)
    (display " = ")
    (display value)
    (newline)
    value))

(fdef evaluate-all-and-return-the-last-one (v1 v2 v3 v4 v5)
```

Evaluator

Example

```
(fdef print-value (text value)
  (evaluate-all-and-return-the-last-one
    (display text)
    (display " = ")
    (display value)
    (newline)
    value))

(fdef evaluate-all-and-return-the-last-one (v1 v2 v3 v4 v5)
  v5)
```

Evaluator

Example

```
(fdef print-value (text value)
  (evaluate-all-and-return-the-last-one
    (display text)
    (display " = ")
    (display value)
    (newline)
    value))

(fdef evaluate-all-and-return-the-last-one (v1 v2 v3 v4 v5)
  v5)

>> (print-value "fact(5)" (fact 5))
```

Evaluator

Example

```
(fdef print-value (text value)
  (evaluate-all-and-return-the-last-one
    (display text)
    (display " = ")
    (display value)
    (newline)
    value))

(fdef evaluate-all-and-return-the-last-one (v1 v2 v3 v4 v5)
  v5)

>> (print-value "fact(5)" (fact 5))
 = 120 fact(5) ;; Huh?
```

Evaluator

Example

```
(fdef print-value (text value)
  (evaluate-all-and-return-the-last-one
    (display text)
    (display " = ")
    (display value)
    (newline)
    value))
```

Does it work?

- Nothing guarantees that the arguments to the function `evaluate-all-and-return-the-last-one` are evaluated from left to right.
- We need a specific control structure to ensure left-to-right evaluation.
- Following Scheme tradition, we will use `begin`.

Evaluator

Sequential Evaluation

```
;;(begin expr0 expr1 ... exprn)
```

Evaluator

Sequential Evaluation

```
;;(begin expr0 expr1 ... exprn)

(define (begin? exp)
  (and (pair? exp) (eq? (car exp) 'begin)))
```

Evaluator

Sequential Evaluation

```scheme
;;(begin expr0 expr1 ... exprn)

(define (begin? exp)
  (and (pair? exp) (eq? (car exp) 'begin)))

(define (begin-expressions exp)
  (cdr exp))
```

Evaluator

Sequential Evaluation

```
;;(begin expr0 expr1 ... exprn)

(define (begin? exp)
  (and (pair? exp) (eq? (car exp) 'begin)))

(define (begin-expressions exp)
  (cdr exp))

(define (eval-begin exp env)
  (define (eval-sequence expressions env)
    (if (null? (cdr expressions))
        (eval (car expressions) env)
        (begin
          (eval (car expressions) env)
          (eval-sequence (cdr expressions) env))))
  (eval-sequence (begin-expressions exp) env))
```

Evaluator

Evaluator

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((name? exp)
         (eval-name exp env))
        ((if? exp)
         (eval-if exp env))
        ...
        ((def? exp)
         (eval-def exp env))
        ((set? exp)
         (eval-set exp env))
        ((fdef? exp)
         (eval-fdef exp env))


        ((call? exp)
         (eval-call exp env))
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Evaluator

Evaluator

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((name? exp)
         (eval-name exp env))
        ((if? exp)
         (eval-if exp env))
        ...
        ((def? exp)
         (eval-def exp env))
        ((set? exp)
         (eval-set exp env))
        ((fdef? exp)
         (eval-fdef exp env))
        ((begin? exp)

        ((call? exp)
         (eval-call exp env))
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Evaluator

Evaluator

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((name? exp)
         (eval-name exp env))
        ((if? exp)
         (eval-if exp env))
        ...
        ((def? exp)
         (eval-def exp env))
        ((set? exp)
         (eval-set exp env))
        ((fdef? exp)
         (eval-fdef exp env))
        ((begin? exp)
         (eval-begin exp env))
        ((call? exp)
         (eval-call exp env))
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Evaluator

Example

```
(fdef print-value (text value)
  (evaluate-all-and-return-the-last-one
    (display text)
    (display " = ")
    (display value)
    (newline)
    value))
```

Evaluator

Example

```
(fdef print-value (text value)
  (begin
    (display text)
    (display " = ")
    (display value)
    (newline)
    value))
```

Evaluator

Example

```
(fdef print-value (text value)
  (begin
    (display text)
    (display " = ")
    (display value)
    (newline)
    value))

>> (print-value "fact(5)" (fact 5))
```

Evaluator

Example

```
(fdef print-value (text value)
  (begin
    (display text)
    (display " = ")
    (display value)
    (newline)
    value))

>> (print-value "fact(5)" (fact 5))
fact(5) = 120
```

Evaluator

Example

```
(fdef print-value (text value)
  (begin
    (display text)
    (display " = ")
    (display value)
    (newline)
    value))

>> (print-value "fact(5)" (fact 5))
fact(5) = 120
120
```

Evaluator

Example

```
(fdef print-value (text value)
  (begin
    (display text)
    (display " = ")
    (display value)
    (newline)
    value))

>> (print-value "fact(5)" (fact 5))
fact(5) = 120
120
>> (print-value "35^3" (* 35 35 35))
```

Evaluator

Example

```
(fdef print-value (text value)
  (begin
    (display text)
    (display " = ")
    (display value)
    (newline)
    value))

>> (print-value "fact(5)" (fact 5))
fact(5) = 120
120
>> (print-value "35^3" (* 35 35 35))
35^3 = 42875
42875
```

Implicit `begin`

- It is usual to consider that every `function` body, every `let` body, and every `flet` body has an *implicit* `begin` form.

Evaluator

Example

```
(fdef print-value (text value)
  (begin
    (display text)
    (display " = ")
    (display value)
    (newline)
    value))
```

Evaluator

Example

```
(fdef print-value (text value)
  (display text)
  (display " = ")
  (display value)
  (newline)
  value)
```

Evaluator

Example

```
(fdef print-value (text value)
  (display text)
  (display " = ")
  (display value)
  (newline)
  value)
```

Implicit begin

- By wrapping the list of forms in a `begin` form we automatically transform the list into a single expression.
- This expression will be evaluated sequentially.
- To implement the implicit `begin`, we have to modify the selectors that return the body of a `let`, the body of a `flet` and the body of a `function`.

Evaluator

Body Selectors

```
;; (let (...)
;;    ...)
```

Evaluator

Body Selectors

```
;; (let (...)
;;   ...)

(define (let-body exp)
  (caddr exp))
```

Evaluator

Body Selectors

```
;; (let (...)
;;   ...)

(define (let-body exp)
  (caddr exp))

;; (flet (...)
;;   ...)
```

Evaluator

Body Selectors

```
;; (let (...)
;;   ...)

(define (let-body exp)
  (caddr exp))

;; (flet (...)
;;   ...)

(define (flet-body exp)
  (caddr exp))
```

Evaluator

Body Selectors

```
;; (let (...)
;;   ...)

(define (let-body exp)
  (caddr exp))

;; (flet (...)
;;   ...)

(define (flet-body exp)
  (caddr exp))

;; (function (...)
;;   ...)
```

Evaluator

Body Selectors

```
;; (let (...)
;;   ...)

(define (let-body exp)
  (caddr exp))

;; (flet (...)
;;   ...)

(define (flet-body exp)
  (caddr exp))

;; (function (...)
;;   ...)

(define (function-body func)
  (caddr func))
```

Evaluator

Body Selectors

```
;; (let (...)
;;   ...)

(define (let-body exp)
  (caddr exp))

;; (flet (...)
;;   ...)

(define (flet-body exp)
  (caddr exp))

;; (function (...)
;;   ...)

(define (function-body func)
  (caddr func))

(define (make-begin expressions)
  (cons 'begin expressions))
```

Evaluator

Body Selectors

```
;; (let (...)
;;   ... ... ...)

(define (let-body exp)
  (make-begin (cddr exp)))

;; (flet (...)
;;   ... ... ...)

(define (flet-body exp)
  (make-begin (cddr exp)))

;; (function (...)
;;   ... ... ...)

(define (function-body func)
  (make-begin (cddr func)))

(define (make-begin expressions)
  (cons 'begin expressions))
```

Evaluator

Higher-Order Functions

```
>> (fdef reflexive-addition (x)
     (+ x x))
```

Evaluator

Higher-Order Functions

```
>> (fdef reflexive-addition (x)
     (+ x x))
(function (x) (+ x x))
```

Evaluator

Higher-Order Functions

```
>> (fdef reflexive-addition (x)
     (+ x x))
(function (x) (+ x x))
>> (reflexive-addition 3)
```

Evaluator

Higher-Order Functions

```
>> (fdef reflexive-addition (x)
     (+ x x))
(function (x) (+ x x))
>> (reflexive-addition 3)
6
```

Evaluator

Higher-Order Functions

```
>> (fdef reflexive-addition (x)
     (+ x x))
(function (x) (+ x x))
>> (reflexive-addition 3)
6
>> (fdef reflexive-product (x)
     (* x x))
```

Evaluator

Higher-Order Functions

```
>> (fdef reflexive-addition (x)
     (+ x x))
(function (x) (+ x x))
>> (reflexive-addition 3)
6
>> (fdef reflexive-product (x)
     (* x x))
(function (x) (* x x))
```

Evaluator

Higher-Order Functions

```
>> (fdef reflexive-addition (x)
     (+ x x))
(function (x) (+ x x))
>> (reflexive-addition 3)
6
>> (fdef reflexive-product (x)
     (* x x))
(function (x) (* x x))
>> (reflexive-product 3)
```

Evaluator

Higher-Order Functions

```
>> (fdef reflexive-addition (x)
     (+ x x))
(function (x) (+ x x))
>> (reflexive-addition 3)
6
>> (fdef reflexive-product (x)
     (* x x))
(function (x) (* x x))
>> (reflexive-product 3)
9
```

Evaluator

Higher-Order Functions

```
>> (fdef reflexive-addition (x)
     (+ x x))
(function (x) (+ x x))
>> (reflexive-addition 3)
6
>> (fdef reflexive-product (x)
     (* x x))
(function (x) (* x x))
>> (reflexive-product 3)
9
>> (fdef reflexive-operation (f x)
     (f x x))
```

Evaluator

Higher-Order Functions

```
>> (fdef reflexive-addition (x)
     (+ x x))
(function (x) (+ x x))
>> (reflexive-addition 3)
6
>> (fdef reflexive-product (x)
     (* x x))
(function (x) (* x x))
>> (reflexive-product 3)
9
>> (fdef reflexive-operation (f x)
     (f x x))
(function (f x) (f x x))
```

Evaluator

Higher-Order Functions

```
>> (fdef reflexive-addition (x)
     (+ x x))
(function (x) (+ x x))
>> (reflexive-addition 3)
6
>> (fdef reflexive-product (x)
     (* x x))
(function (x) (* x x))
>> (reflexive-product 3)
9
>> (fdef reflexive-operation (f x)
     (f x x))
(function (f x) (f x x))
>> (reflexive-operation + 3)
```

Evaluator

Higher-Order Functions

```
>> (fdef reflexive-addition (x)
     (+ x x))
(function (x) (+ x x))
>> (reflexive-addition 3)
6
>> (fdef reflexive-product (x)
     (* x x))
(function (x) (* x x))
>> (reflexive-product 3)
9
>> (fdef reflexive-operation (f x)
     (f x x))
(function (f x) (f x x))
>> (reflexive-operation + 3)
6
```

Evaluator

Higher-Order Functions

```
>> (fdef reflexive-addition (x)
     (+ x x))
(function (x) (+ x x))
>> (reflexive-addition 3)
6
>> (fdef reflexive-product (x)
     (* x x))
(function (x) (* x x))
>> (reflexive-product 3)
9
>> (fdef reflexive-operation (f x)
     (f x x))
(function (f x) (f x x))
>> (reflexive-operation + 3)
6
>> (reflexive-operation * 3)
```

Evaluator

Higher-Order Functions

```
>> (fdef reflexive-addition (x)
     (+ x x))
(function (x) (+ x x))
>> (reflexive-addition 3)
6
>> (fdef reflexive-product (x)
     (* x x))
(function (x) (* x x))
>> (reflexive-product 3)
9
>> (fdef reflexive-operation (f x)
     (f x x))
(function (f x) (f x x))
>> (reflexive-operation + 3)
6
>> (reflexive-operation * 3)
9
```

Evaluator

Higher-Order Functions

```
>> (fdef reflexive-addition (x)
     (+ x x))
(function (x) (+ x x))
>> (reflexive-addition 3)
6
>> (fdef reflexive-product (x)
     (* x x))
(function (x) (* x x))
>> (reflexive-product 3)
9
>> (fdef reflexive-operation (f x)
     (f x x))
(function (f x) (f x x))
>> (reflexive-operation + 3)
6
>> (reflexive-operation * 3)
9
>> (reflexive-operation / 3)
```

Evaluator

Higher-Order Functions

```
>> (fdef reflexive-addition (x)
     (+ x x))
(function (x) (+ x x))
>> (reflexive-addition 3)
6
>> (fdef reflexive-product (x)
     (* x x))
(function (x) (* x x))
>> (reflexive-product 3)
9
>> (fdef reflexive-operation (f x)
     (f x x))
(function (f x) (f x x))
>> (reflexive-operation + 3)
6
>> (reflexive-operation * 3)
9
>> (reflexive-operation / 3)
1
```

Evaluator

Higher-Order Functions

```
>> (fdef reflexive-addition (x)
     (+ x x))
(function (x) (+ x x))
>> (reflexive-addition 3)
6
>> (fdef reflexive-product (x)
     (* x x))
(function (x) (* x x))
>> (reflexive-product 3)
9
>> (fdef reflexive-operation (f x)
     (f x x))
(function (f x) (f x x))
>> (reflexive-operation + 3)
6
>> (reflexive-operation * 3)
9
>> (reflexive-operation / 3)
1
>> (reflexive-operation - 3)
```

Evaluator

Higher-Order Functions

```
>> (fdef reflexive-addition (x)
     (+ x x))
(function (x) (+ x x))
>> (reflexive-addition 3)
6
>> (fdef reflexive-product (x)
     (* x x))
(function (x) (* x x))
>> (reflexive-product 3)
9
>> (fdef reflexive-operation (f x)
     (f x x))
(function (f x) (f x x))
>> (reflexive-operation + 3)
6
>> (reflexive-operation * 3)
9
>> (reflexive-operation / 3)
1
>> (reflexive-operation - 3)
0
```

Evaluator

Higher-Order Functions

- A *higher-order function* is a function that accepts other functions as arguments or that computes functions as results.
- Most modern programming languages support higher-order functions and provide libraries containing higher-order functions (e.g., `qsort` in C, `map` in Scheme).
- In what concerns higher-order functions, our evaluator has the same expressive power as the C programming language:
  - We can pass functions as arguments.
  - We can apply functions contained in parameters.
  - But we cannot create anonymous functions (yet).
  - And we have other problems that C avoids by forbidding internal function definitions (we will discuss them later).

Evaluator

Lies, Damned Lies

```
>> (fdef lie (truth)
    (display "it is not true that ")
    (display truth)
    (newline))
```

Evaluator

Lies, Damned Lies

```
>> (fdef lie (truth)
    (display "it is not true that ")
    (display truth)
    (newline))
(function (truth) ...)
```

Evaluator

Lies, Damned Lies

```
>> (fdef lie (truth)
     (display "it is not true that ")
     (display truth)
     (newline))
(function (truth) ...)
>> (lie "1+1=2")
```

Evaluator

Lies, Damned Lies

```
>> (fdef lie (truth)
    (display "it is not true that ")
    (display truth)
    (newline))
(function (truth) ...)
>> (lie "1+1=2")
it is not true that 1+1=2
```

Evaluator

Lies, Damned Lies

```
>> (fdef lie (truth)
    (display "it is not true that ")
    (display truth)
    (newline))
(function (truth) ...)
>> (lie "1+1=2")
it is not true that 1+1=2
>> (lie "it is not true that 1+1=2")
```

Evaluator

Lies, Damned Lies

```
>> (fdef lie (truth)
    (display "it is not true that ")
    (display truth)
    (newline))
(function (truth) ...)
>> (lie "1+1=2")
it is not true that 1+1=2
>> (lie "it is not true that 1+1=2")
it is not true that it is not true that 1+1=2
```

Evaluator

Lies, Damned Lies

```
>> (fdef lie (truth)
    (display "it is not true that ")
    (display truth)
    (newline))
(function (truth) ...)
>> (lie "1+1=2")
it is not true that 1+1=2
>> (lie "it is not true that 1+1=2")
it is not true that it is not true that 1+1=2
>> (fdef make-it-true (lie)
     (lie lie))
```

Evaluator

Lies, Damned Lies

```
>> (fdef lie (truth)
    (display "it is not true that ")
    (display truth)
    (newline))
(function (truth) ...)
>> (lie "1+1=2")
it is not true that 1+1=2
>> (lie "it is not true that 1+1=2")
it is not true that it is not true that 1+1=2
>> (fdef make-it-true (lie)
     (lie lie))
(function (lie) (lie lie))
```

Evaluator

Lies, Damned Lies

```
>> (fdef lie (truth)
    (display "it is not true that ")
    (display truth)
    (newline))
(function (truth) ...)
>> (lie "1+1=2")
it is not true that 1+1=2
>> (lie "it is not true that 1+1=2")
it is not true that it is not true that 1+1=2
>> (fdef make-it-true (lie)
     (lie lie))
(function (lie) (lie lie))
>> (make-it-true "1+1=3")
```

Evaluator

Lies, Damned Lies

```
>> (fdef lie (truth)
    (display "it is not true that ")
    (display truth)
    (newline))
(function (truth) ...)
>> (lie "1+1=2")
it is not true that 1+1=2
>> (lie "it is not true that 1+1=2")
it is not true that it is not true that 1+1=2
>> (fdef make-it-true (lie)
     (lie lie))
(function (lie) (lie lie))
>> (make-it-true "1+1=3")
error in cadr: expected appropriate list structure, but got "1+1=3"

backtrace:
  0  (cadr func)
  1  (function-parameters func)
  2  (eval input initial-environment)
  3  (repl)
```

Evaluator

Naming Problem

- Natural languages use many Homonyms:
  1. The bandage was wound around the wound.
  2. Don't object to the object.
  3. We polish the Polish furniture.
  4. The present is the best time to present the present.
- Programming languages tend to immitate natural languages and allow homonyms as long as they can be distinguished.
- In general, different *namespaces* are created to distinguish the different meanings (types, variables, functions, etc).
- Some languages do not allow more than one *namespace* and prefer to use *name conventions*.

Evaluator

Homonyms in Java

```
public class foo {

    int foo;

    foo() {
        foo = 0;
    }

    foo(foo foo) {
        this.foo = foo.foo + 1;
    }

    void foo(foo foo) {
        System.err.println(foo.foo);
    }

    public static void main(String args[]) {
        foo foo = new foo();
        foo:foo.foo(new foo(foo));
    }
}
```

Evaluator

Homonyms in Java

```
public class foo {

    int foo;

    foo() {
        foo = 0;
    }

    foo(foo foo) {
        this.foo = foo.foo + 1;
    }

    void foo(foo foo) {
        System.err.println(foo.foo);
    }

    public static void main(String args[]) {
        foo foo = new foo();
        foo:foo.foo(new foo(foo));
    }
}
```

Evaluator

Namespaces

- Java has namespaces for types, for constructors, for methods, for fields and variables, for labels, etc.
- C has a different namespace for *structs*, but a single namespace for functions and variables.
- Dylan has a single namespace and uses name conventions for distinguishing different meanings: `list` is a function, `<list>` is a type and `$list` is a global variable.
- Scheme has a single namespace: each name has just one meaning in each context (but recent name conventions are being proposed, e.g., exceptions must start with the letter `&`).
- Common Lisp has multiple namespaces: the name `list` designates a type, a function, a variable, a label, etc.

Evaluator

Lisp-1 vs Lisp-2

- A Lisp-2 is a Lisp that has one namespace for functions and another namespace for variables.
- A Lisp-1 is a Lisp that has a single namespace where a name either represents a function or a variable but not both.
- Note that Common Lisp is more than a Lisp-2: it is a Lisp-n
- Let's modify our evaluator to implement Lisp-2 semantics:
  1. Create a different namespace for functions.
  2. Each function definition is placed in the function namespace.
  3. Each function call accesses the function namespace.
- Our namespace for functions will be implemented using *hidden* naming conventions, but we could use more sophisticated (and efficient) approaches.

Evaluator

Function Namespace

```
(define (in-function-namespace name)
  (string->symbol
    (string-append
      "%function-"
      (symbol->string name))))
```

Evaluator

Function Namespace

```
(define (in-function-namespace name)
  (string->symbol
    (string-append
      "%function-"
      (symbol->string name))))
```

flet

```
;(flet ((square (x) (* x x))
;       (foo (a b c) (+ a (* b c))))
;  ...)
```

Evaluator

Function Namespace

```
(define (in-function-namespace name)
  (string->symbol
    (string-append
      "%function-"
      (symbol->string name))))
```

flet

```
;(flet ((square (x) (* x x))
;       (foo (a b c) (+ a (* b c))))
;  ...)

(define (flet-names exp)
  (map car (cadr exp)))
```

Evaluator

Function Namespace

```
(define (in-function-namespace name)
  (string->symbol
    (string-append
      "%function-"
      (symbol->string name))))
```

flet

```
;(flet ((square (x) (* x x))
;       (foo (a b c) (+ a (* b c))))
;   ...)

(define (flet-names exp)
  (map car (cadr exp)))

(define (flet-names exp)
  (map in-function-namespace
       (map car (cadr exp))))
```

Evaluator

fdef

```
;(fdef triple (a) (+ a a a))
```

Evaluator

fdef

```
;(fdef triple (a) (+ a a a))

(define (fdef-name exp)
  (cadr exp))
```

Evaluator

fdef

```
;(fdef triple (a) (+ a a a))

(define (fdef-name exp)
  (cadr exp))

(define (fdef-name exp)
  (in-function-namespace (cadr exp)))
```

Evaluator

fdef

```
;(fdef triple (a) (+ a a a))

(define (fdef-name exp)
  (cadr exp))

(define (fdef-name exp)
  (in-function-namespace (cadr exp)))
```

Function Call

```
;;(square radius)
```

Evaluator

fdef

```
;(fdef triple (a) (+ a a a))

(define (fdef-name exp)
  (cadr exp))

(define (fdef-name exp)
  (in-function-namespace (cadr exp)))
```

Function Call

```
;;(square radius)

(define (call-operator exp)
  (car exp))
```

Evaluator

fdef

```
;(fdef triple (a) (+ a a a))

(define (fdef-name exp)
  (cadr exp))

(define (fdef-name exp)
  (in-function-namespace (cadr exp)))
```

Function Call

```
;;(square radius)

(define (call-operator exp)
  (car exp))

(define (call-operator exp)
  (in-function-namespace (car exp)))
```

Evaluator

Initial Bindings

```
(define initial-bindings
  (list (cons 'pi 3.14159)
        ...
        (cons '+ (make-primitive +))
        (cons '* (make-primitive *))
        (cons '- (make-primitive -))
        (cons '/ (make-primitive /))
        (cons '= (make-primitive =))
        (cons '< (make-primitive <))
        (cons '> (make-primitive >))
        (cons '<= (make-primitive <=))
        (cons '>= (make-primitive >=))
        ...
        (cons 'read (make-primitive read))))
```

Evaluator

Initial Bindings

```
(define initial-bindings
  (list (cons 'pi 3.14159)
        ...
        (cons (in-function-namespace '+) (make-primitive +))
        (cons (in-function-namespace '*) (make-primitive *))
        (cons (in-function-namespace '-) (make-primitive -))
        (cons (in-function-namespace '/) (make-primitive /))
        (cons (in-function-namespace '=) (make-primitive =))
        (cons (in-function-namespace '<) (make-primitive <))
        (cons (in-function-namespace '>) (make-primitive >))
        (cons (in-function-namespace '<=) (make-primitive <=))
        (cons (in-function-namespace '>=) (make-primitive >=))
        ...
        (cons (in-function-namespace 'read) (make-primitive read))))
```

Evaluator

Lies, Damned Lies

```
>> (fdef lie (truth)
    (display "it is not true that ")
    (display truth)
    (newline))
```

Evaluator

Lies, Damned Lies

```
>> (fdef lie (truth)
    (display "it is not true that ")
    (display truth)
    (newline))
(function (truth) ...)
```

Evaluator

Lies, Damned Lies

```
>> (fdef lie (truth)
    (display "it is not true that ")
    (display truth)
    (newline))
(function (truth) ...)
>> (lie "1+1=2")
```

Evaluator

Lies, Damned Lies

```
>> (fdef lie (truth)
    (display "it is not true that ")
    (display truth)
    (newline))
(function (truth) ...)
>> (lie "1+1=2")
it is not true that 1+1=2
```

Evaluator

Lies, Damned Lies

```
>> (fdef lie (truth)
    (display "it is not true that ")
    (display truth)
    (newline))
(function (truth) ...)
>> (lie "1+1=2")
it is not true that 1+1=2
>> (fdef make-it-true (lie)
     (lie lie))
```

Evaluator

Lies, Damned Lies

```
>> (fdef lie (truth)
    (display "it is not true that ")
    (display truth)
    (newline))
(function (truth) ...)
>> (lie "1+1=2")
it is not true that 1+1=2
>> (fdef make-it-true (lie)
     (lie lie))
(function (lie) (lie lie))
```

Evaluator

Lies, Damned Lies

```
>> (fdef lie (truth)
    (display "it is not true that ")
    (display truth)
    (newline))
(function (truth) ...)
>> (lie "1+1=2")
it is not true that 1+1=2
>> (fdef make-it-true (lie)
     (lie lie))
(function (lie) (lie lie))
>> (make-it-true "1+1=3")
```

Evaluator

Lies, Damned Lies

```
>> (fdef lie (truth)
    (display "it is not true that ")
    (display truth)
    (newline))
(function (truth) ...)
>> (lie "1+1=2")
it is not true that 1+1=2
>> (fdef make-it-true (lie)
     (lie lie))
(function (lie) (lie lie))
>> (make-it-true "1+1=3")
it is not true that 1+1=3
```

Evaluator

Puzzle

```
>> (def foo 1)
```

Evaluator

Puzzle

```
>> (def foo 1)
1
```

Evaluator

Puzzle

```
>> (def foo 1)
1
>> (fdef foo () foo)
```

Evaluator

Puzzle

```
>> (def foo 1)
1
>> (fdef foo () foo)
(function () foo)
```

Evaluator

Puzzle

```
>> (def foo 1)
1
>> (fdef foo () foo)
(function () foo)
>> foo
```

Evaluator

Puzzle

```
>> (def foo 1)
1
>> (fdef foo () foo)
(function () foo)
>> foo
1
```

Evaluator

Puzzle

```
>> (def foo 1)
1
>> (fdef foo () foo)
(function () foo)
>> foo
1
>> (foo)
```

Evaluator

Puzzle

```
>> (def foo 1)
1
>> (fdef foo () foo)
(function () foo)
>> foo
1
>> (foo)
1
```

Evaluator

Puzzle

```
>> (def foo 1)
1
>> (fdef foo () foo)
(function () foo)
>> foo
1
>> (foo)
1
>> (let ((foo (+ (foo) 1)))
     (flet ((foo (foo) (+ foo 1)))
       (foo (+ foo 1))))
```

Evaluator

Puzzle

```
>> (def foo 1)
1
>> (fdef foo () foo)
(function () foo)
>> foo
1
>> (foo)
1
>> (let ((foo (+ (foo) 1)))
     (flet ((foo (foo) (+ foo 1)))
       (foo (+ foo 1))))
4
```

Evaluator

Puzzle

```
>> (def foo 1)
1
>> (fdef foo () foo)
(function () foo)
>> foo
1
>> (foo)
1
>> (let ((foo (+ (foo) 1)))
     (flet ((foo (foo) (+ foo 1)))
       (foo (+ foo 1))))
4
>> (+ foo
      (foo)
      (let ((foo 2))
        (foo))
      (flet ((foo () foo))
        (foo)))
```

Evaluator

Puzzle

```
>> (def foo 1)
1
>> (fdef foo () foo)
(function () foo)
>> foo
1
>> (foo)
1
>> (let ((foo (+ (foo) 1)))
     (flet ((foo (foo) (+ foo 1)))
       (foo (+ foo 1))))
4
>> (+ foo
      (foo)
      (let ((foo 2))
        (foo))
      (flet ((foo () foo))
        (foo)))
5
```

Evaluator

Lisp-2 and Higher-Order Functions

```
>> (fdef reflexive-operation (f x)
     (f x x))
```

Evaluator

Lisp-2 and Higher-Order Functions

```
>> (fdef reflexive-operation (f x)
     (f x x))
(function (f x) (f x x))
```

Evaluator

Lisp-2 and Higher-Order Functions

```
>> (fdef reflexive-operation (f x)
     (f x x))
(function (f x) (f x x))
>> (reflexive-operation + 3)
```

Evaluator

Lisp-2 and Higher-Order Functions

```
>> (fdef reflexive-operation (f x)
     (f x x))
(function (f x) (f x x))
>> (reflexive-operation + 3)
error in "Unbound name -- EVAL-NAME": +
```

Problem

- In a Lisp-2, a name that is *not* in function call position is always treated as *not* belonging to the function namespace.
- We need extra syntax to allow a name to be treated as belonging to the function namespace.
- We will use the form `(function +)`.
- If we allow changing the reader, we might even treat `#'`*foo* as an abbreviation for `(function` *foo*`)`.

Evaluator

Evaluator

```
;;(function foo)
```

Evaluator

Evaluator

```
;;(function foo)

(define (function-reference? exp)
  (and (pair? exp) (eq? (car exp) 'function)))
```

Evaluator

Evaluator

```
;;(function foo)

(define (function-reference? exp)
  (and (pair? exp) (eq? (car exp) 'function)))

(define (function-reference-name exp)
  (cadr exp))
```

Evaluator

Evaluator

```
;;(function foo)

(define (function-reference? exp)
  (and (pair? exp) (eq? (car exp) 'function)))

(define (function-reference-name exp)
  (cadr exp))

(define (eval-function-reference exp env)
  (eval (in-function-namespace (function-reference-name exp))
        env))
```

Evaluator

Evaluator

```
;;(function foo)

(define (function-reference? exp)
  (and (pair? exp) (eq? (car exp) 'function)))

(define (function-reference-name exp)
  (cadr exp))

(define (eval-function-reference exp env)
  (eval (in-function-namespace (function-reference-name exp))
        env))

(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((name? exp)
         (eval-name exp env))


        ...
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Evaluator

Evaluator

```
;;(function foo)

(define (function-reference? exp)
  (and (pair? exp) (eq? (car exp) 'function)))

(define (function-reference-name exp)
  (cadr exp))

(define (eval-function-reference exp env)
  (eval (in-function-namespace (function-reference-name exp))
        env))

(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((name? exp)
         (eval-name exp env))
        ((function-reference? exp)

        ...
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Evaluator

Evaluator

```
;;(function foo)

(define (function-reference? exp)
  (and (pair? exp) (eq? (car exp) 'function)))

(define (function-reference-name exp)
  (cadr exp))

(define (eval-function-reference exp env)
  (eval (in-function-namespace (function-reference-name exp))
        env))

(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((name? exp)
         (eval-name exp env))
        ((function-reference? exp)
         (eval-function-reference exp env))
        ...
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Evaluator

Lisp-2 and Higher-Order Functions

```
>> (fdef reflexive-operation (f x)
     (f x x))
```

Evaluator

Lisp-2 and Higher-Order Functions

```
>> (fdef reflexive-operation (f x)
     (f x x))
(function (f x) (f x x))
```

Evaluator

Lisp-2 and Higher-Order Functions

```
>> (fdef reflexive-operation (f x)
     (f x x))
(function (f x) (f x x))
>> (reflexive-operation (function +) 3)
```

Evaluator

Lisp-2 and Higher-Order Functions

```
>> (fdef reflexive-operation (f x)
     (f x x))
(function (f x) (f x x))
>> (reflexive-operation (function +) 3)
error in "Unbound name -- EVAL-NAME": %function-f
```

Problem

- In a Lisp-2, a name that is in function call position is always treated as belonging to the function namespace.
- We need extra syntax to allow a name to be treated as *not* belonging to the function namespace.
- We will use the form `(funcall f ...)`.

Evaluator

Evaluator

```
;;(funcall f ...)
```

Evaluator

Evaluator

```
;;(funcall f ...)

(define (funcall? exp)
  (and (pair? exp) (eq? (car exp) 'funcall)))
```

Evaluator

Evaluator

```
;;(funcall f ...)

(define (funcall? exp)
  (and (pair? exp) (eq? (car exp) 'funcall)))

(define (funcall-operator exp)
  (cadr exp))
```

Evaluator

Evaluator

```
;;(funcall f ...)

(define (funcall? exp)
  (and (pair? exp) (eq? (car exp) 'funcall)))

(define (funcall-operator exp)
  (cadr exp))

(define (funcall-operands exp)
  (cddr exp))
```

Evaluator

Evaluator

```
;;(funcall f ...)

(define (funcall? exp)
  (and (pair? exp) (eq? (car exp) 'funcall)))

(define (funcall-operator exp)
  (cadr exp))

(define (funcall-operands exp)
  (cddr exp))

(define (eval-funcall exp env)
  (let ((func (eval-name (funcall-operator exp) env))
        (args (eval-exprs (funcall-operands exp) env)))
    (if (primitive? func)
        (apply-primitive-function func args)
        (let ((extended-environment
               (augment-environment (function-parameters func)
                                    args
                                    env)))
          (eval (function-body func) extended-environment)))))
```

Evaluator

Evaluator

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((name? exp)
         (eval-name exp env))
        ((function-reference? exp)
         (eval-function-reference exp env))


        ((if? exp)
         (eval-if exp env))
        ((let? exp)
         (eval-let exp env))
        ((flet? exp)
         (eval-flet exp env))
        ...
        ((call? exp)
         (eval-call exp env))
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Evaluator

Evaluator

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((name? exp)
         (eval-name exp env))
        ((function-reference? exp)
         (eval-function-reference exp env))
        ((funcall? exp)

        ((if? exp)
         (eval-if exp env))
        ((let? exp)
         (eval-let exp env))
        ((flet? exp)
         (eval-flet exp env))
        ...
        ((call? exp)
         (eval-call exp env))
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Evaluator

Evaluator

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((name? exp)
         (eval-name exp env))
        ((function-reference? exp)
         (eval-function-reference exp env))
        ((funcall? exp)
         (eval-funcall exp env))
        ((if? exp)
         (eval-if exp env))
        ((let? exp)
         (eval-let exp env))
        ((flet? exp)
         (eval-flet exp env))
        ...
        ((call? exp)
         (eval-call exp env))
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Evaluator

Lisp-2 and Higher-Order Functions

```
>> (fdef reflexive-operation (f x)
     (funcall f x x))
```

Evaluator

Lisp-2 and Higher-Order Functions

```
>> (fdef reflexive-operation (f x)
     (funcall f x x))
(function (f x) (funcall f x x))
```

Evaluator

Lisp-2 and Higher-Order Functions

```
>> (fdef reflexive-operation (f x)
     (funcall f x x))
(function (f x) (funcall f x x))
>> (reflexive-operation (function +) 3)
```

Evaluator

Lisp-2 and Higher-Order Functions

```
>> (fdef reflexive-operation (f x)
     (funcall f x x))
(function (f x) (funcall f x x))
>> (reflexive-operation (function +) 3)
6
```

Evaluator

Lisp-2 and Higher-Order Functions

```
>> (fdef reflexive-operation (f x)
     (funcall f x x))
(function (f x) (funcall f x x))
>> (reflexive-operation (function +) 3)
6
>> (fdef sum (f a b)
     (if (> a b)
       0
       (+ (funcall f a)
          (sum f (+ a 1) b))))
```

Evaluator

Lisp-2 and Higher-Order Functions

```
>> (fdef reflexive-operation (f x)
     (funcall f x x))
(function (f x) (funcall f x x))
>> (reflexive-operation (function +) 3)
6
>> (fdef sum (f a b)
     (if (> a b)
       0
       (+ (funcall f a)
          (sum f (+ a 1) b))))
(function (f a b) ...)
```

Evaluator

Lisp-2 and Higher-Order Functions

```
>> (fdef reflexive-operation (f x)
     (funcall f x x))
(function (f x) (funcall f x x))
>> (reflexive-operation (function +) 3)
6
>> (fdef sum (f a b)
     (if (> a b)
       0
       (+ (funcall f a)
          (sum f (+ a 1) b))))
(function (f a b) ...)
>> (fdef cube (x) (* x x x))
```

Evaluator

Lisp-2 and Higher-Order Functions

```
>> (fdef reflexive-operation (f x)
     (funcall f x x))
(function (f x) (funcall f x x))
>> (reflexive-operation (function +) 3)
6
>> (fdef sum (f a b)
     (if (> a b)
       0
       (+ (funcall f a)
          (sum f (+ a 1) b))))
(function (f a b) ...)
>> (fdef cube (x) (* x x x))
(function (x) ...)
```

Evaluator

Lisp-2 and Higher-Order Functions

```
>> (fdef reflexive-operation (f x)
     (funcall f x x))
(function (f x) (funcall f x x))
>> (reflexive-operation (function +) 3)
6
>> (fdef sum (f a b)
     (if (> a b)
       0
       (+ (funcall f a)
          (sum f (+ a 1) b))))
(function (f a b) ...)
>> (fdef cube (x) (* x x x))
(function (x) ...)
>> (sum (function cube) 1 10)
```

Evaluator

Lisp-2 and Higher-Order Functions

```
>> (fdef reflexive-operation (f x)
     (funcall f x x))
(function (f x) (funcall f x x))
>> (reflexive-operation (function +) 3)
6
>> (fdef sum (f a b)
     (if (> a b)
       0
       (+ (funcall f a)
          (sum f (+ a 1) b))))
(function (f a b) ...)
>> (fdef cube (x) (* x x x))
(function (x) ...)
>> (sum (function cube) 1 10)
3025
```

Evaluator

Lisp-2 and Higher-Order Functions

```
>> (fdef reflexive-operation (f x)
     (funcall f x x))
(function (f x) (funcall f x x))
>> (reflexive-operation (function +) 3)
6
>> (fdef sum (f a b)
     (if (> a b)
       0
       (+ (funcall f a)
          (sum f (+ a 1) b))))
(function (f a b) ...)
>> (fdef cube (x) (* x x x))
(function (x) ...)
>> (sum (function cube) 1 10)
3025
>> (fdef identity (x) x)
```

Evaluator

Lisp-2 and Higher-Order Functions

```
>> (fdef reflexive-operation (f x)
     (funcall f x x))
(function (f x) (funcall f x x))
>> (reflexive-operation (function +) 3)
6
>> (fdef sum (f a b)
     (if (> a b)
       0
       (+ (funcall f a)
          (sum f (+ a 1) b))))
(function (f a b) ...)
>> (fdef cube (x) (* x x x))
(function (x) ...)
>> (sum (function cube) 1 10)
3025
>> (fdef identity (x) x)
(function (x) x)
```

Evaluator

Lisp-2 and Higher-Order Functions

```
>> (fdef reflexive-operation (f x)
     (funcall f x x))
(function (f x) (funcall f x x))
>> (reflexive-operation (function +) 3)
6
>> (fdef sum (f a b)
     (if (> a b)
       0
       (+ (funcall f a)
          (sum f (+ a 1) b))))
(function (f a b) ...)
>> (fdef cube (x) (* x x x))
(function (x) ...)
>> (sum (function cube) 1 10)
3025
>> (fdef identity (x) x)
(function (x) x)
>> (sum (function identity) 1 100)
```

Evaluator

Lisp-2 and Higher-Order Functions

```
>> (fdef reflexive-operation (f x)
     (funcall f x x))
(function (f x) (funcall f x x))
>> (reflexive-operation (function +) 3)
6
>> (fdef sum (f a b)
     (if (> a b)
       0
       (+ (funcall f a)
          (sum f (+ a 1) b))))
(function (f a b) ...)
>> (fdef cube (x) (* x x x))
(function (x) ...)
>> (sum (function cube) 1 10)
3025
>> (fdef identity (x) x)
(function (x) x)
>> (sum (function identity) 1 100)
5050
```

Evaluator

Approximating π

$$\pi = 4 \sum_{i=0}^{\infty} \frac{(-1)^i}{2i+1}$$

Approximating π

```
>> (fdef expt (x n)
     (if (= n 0) 1 (* x (expt x (- n 1)))))
```

Evaluator

Approximating π

$$\pi = 4\sum_{i=0}^{\infty}\frac{(-1)^i}{2i+1}$$

Approximating π

```
>> (fdef expt (x n)
     (if (= n 0) 1 (* x (expt x (- n 1)))))
(function (x n) ...)
```

Evaluator

Approximating π

$$\pi = 4\sum_{i=0}^{\infty}\frac{(-1)^i}{2i+1}$$

Approximating π

```
>> (fdef expt (x n)
     (if (= n 0) 1 (* x (expt x (- n 1)))))
(function (x n) ...)
>> (fdef approx-pi (n)
     (flet ((term (i) (/ (expt -1.0 i) (+ (* 2 i) 1))))
       (* 4 (sum (function term) 0 n))))
```

Evaluator

Approximating π

$$\pi = 4\sum_{i=0}^{\infty}\frac{(-1)^i}{2i+1}$$

Approximating π

```
>> (fdef expt (x n)
     (if (= n 0) 1 (* x (expt x (- n 1)))))
(function (x n) ...)
>> (fdef approx-pi (n)
     (flet ((term (i) (/ (expt -1.0 i) (+ (* 2 i) 1))))
       (* 4 (sum (function term) 0 n))))
(function (n) ...)
```

Evaluator

Approximating π

$$\pi = 4\sum_{i=0}^{\infty}\frac{(-1)^i}{2i+1}$$

Approximating π

```
>> (fdef expt (x n)
     (if (= n 0) 1 (* x (expt x (- n 1)))))
(function (x n) ...)
>> (fdef approx-pi (n)
     (flet ((term (i) (/ (expt -1.0 i) (+ (* 2 i) 1))))
       (* 4 (sum (function term) 0 n))))
(function (n) ...)
>> (approx-pi 10)
```

Evaluator

Approximating π

$$\pi = 4\sum_{i=0}^{\infty}\frac{(-1)^i}{2i+1}$$

Approximating π

```
>> (fdef expt (x n)
     (if (= n 0) 1 (* x (expt x (- n 1)))))
(function (x n) ...)
>> (fdef approx-pi (n)
     (flet ((term (i) (/ (expt -1.0 i) (+ (* 2 i) 1))))
       (* 4 (sum (function term) 0 n))))
(function (n) ...)
>> (approx-pi 10)
3.232315809405593
```

Evaluator

Approximating π

$$\pi = 4\sum_{i=0}^{\infty}\frac{(-1)^i}{2i+1}$$

Approximating π

```
>> (fdef expt (x n)
     (if (= n 0) 1 (* x (expt x (- n 1)))))
(function (x n) ...)
>> (fdef approx-pi (n)
     (flet ((term (i) (/ (expt -1.0 i) (+ (* 2 i) 1))))
       (* 4 (sum (function term) 0 n))))
(function (n) ...)
>> (approx-pi 10)
3.232315809405593
>> (approx-pi 100)
```

Evaluator

Approximating π

$$\pi = 4\sum_{i=0}^{\infty}\frac{(-1)^i}{2i+1}$$

Approximating π

```
>> (fdef expt (x n)
     (if (= n 0) 1 (* x (expt x (- n 1)))))
(function (x n) ...)
>> (fdef approx-pi (n)
     (flet ((term (i) (/ (expt -1.0 i) (+ (* 2 i) 1))))
       (* 4 (sum (function term) 0 n))))
(function (n) ...)
>> (approx-pi 10)
3.232315809405593
>> (approx-pi 100)
3.15149340107091
```

Evaluator

Approximating π

$$\pi = 4\sum_{i=0}^{\infty}\frac{(-1)^i}{2i+1}$$

Approximating π

```
>> (fdef expt (x n)
     (if (= n 0) 1 (* x (expt x (- n 1)))))
(function (x n) ...)
>> (fdef approx-pi (n)
     (flet ((term (i) (/ (expt -1.0 i) (+ (* 2 i) 1))))
       (* 4 (sum (function term) 0 n))))
(function (n) ...)
>> (approx-pi 10)
3.232315809405593
>> (approx-pi 100)
3.151493401070991
>> (approx-pi 1000)
```

Evaluator

Approximating π

$$\pi = 4\sum_{i=0}^{\infty}\frac{(-1)^i}{2i+1}$$

Approximating π

```
>> (fdef expt (x n)
     (if (= n 0) 1 (* x (expt x (- n 1)))))
(function (x n) ...)
>> (fdef approx-pi (n)
     (flet ((term (i) (/ (expt -1.0 i) (+ (* 2 i) 1))))
       (* 4 (sum (function term) 0 n))))
(function (n) ...)
>> (approx-pi 10)
3.232315809405593
>> (approx-pi 100)
3.151493401070991
>> (approx-pi 1000)
3.142591654339543
```

Evaluator

Factorial

$$n! = 1 \times 2 \times 3 \times \ldots \times n = \prod_{i=1}^{n} i$$

Factorial

```
>> (fdef product (f a b)
     (if (> a b)
       1
       (* (funcall f a)
          (product f (+ a 1) b))))
```

Evaluator

Factorial

$$n! = 1 \times 2 \times 3 \times \ldots \times n = \prod_{i=1}^{n} i$$

Factorial

```
>> (fdef product (f a b)
     (if (> a b)
       1
       (* (funcall f a)
          (product f (+ a 1) b))))
(function (f a b) ...)
```

Evaluator

Factorial

$$n! = 1 \times 2 \times 3 \times \ldots \times n = \prod_{i=1}^{n} i$$

Factorial

```
>> (fdef product (f a b)
     (if (> a b)
       1
       (* (funcall f a)
          (product f (+ a 1) b))))
(function (f a b) ...)
>> (fdef fact (n)
```

Evaluator

Factorial

$$n! = 1 \times 2 \times 3 \times \ldots \times n = \prod_{i=1}^{n} i$$

Factorial

```
>> (fdef product (f a b)
     (if (> a b)
       1
       (* (funcall f a)
          (product f (+ a 1) b))))
(function (f a b) ...)
>> (fdef fact (n)
     (product (function identity) 1 n))
```

Evaluator

Factorial

$$n! = 1 \times 2 \times 3 \times \ldots \times n = \prod_{i=1}^{n} i$$

Factorial

```
>> (fdef product (f a b)
     (if (> a b)
       1
       (* (funcall f a)
          (product f (+ a 1) b))))
(function (f a b) ...)
>> (fdef fact (n)
     (product (function identity) 1 n))
(function (n) ...)
```

Evaluator

Factorial

$$n! = 1 \times 2 \times 3 \times \ldots \times n = \prod_{i=1}^{n} i$$

Factorial

```
>> (fdef product (f a b)
     (if (> a b)
       1
       (* (funcall f a)
          (product f (+ a 1) b))))
(function (f a b) ...)
>> (fdef fact (n)
     (product (function identity) 1 n))
(function (n) ...)
>> (fact 0)
```

Evaluator

Factorial

$$n! = 1 \times 2 \times 3 \times \ldots \times n = \prod_{i=1}^{n} i$$

Factorial

```
>> (fdef product (f a b)
     (if (> a b)
       1
       (* (funcall f a)
          (product f (+ a 1) b))))
(function (f a b) ...)
>> (fdef fact (n)
     (product (function identity) 1 n))
(function (n) ...)
>> (fact 0)
1
```

Evaluator

Factorial

$$n! = 1 \times 2 \times 3 \times \ldots \times n = \prod_{i=1}^{n} i$$

Factorial

```
>> (fdef product (f a b)
     (if (> a b)
       1
       (* (funcall f a)
          (product f (+ a 1) b))))
(function (f a b) ...)
>> (fdef fact (n)
     (product (function identity) 1 n))
(function (n) ...)
>> (fact 0)
1
>> (fact 10)
```

Evaluator

Factorial

$$n! = 1 \times 2 \times 3 \times \ldots \times n = \prod_{i=1}^{n} i$$

Factorial

```
>> (fdef product (f a b)
     (if (> a b)
       1
       (* (funcall f a)
          (product f (+ a 1) b))))
(function (f a b) ...)
>> (fdef fact (n)
     (product (function identity) 1 n))
(function (n) ...)
>> (fact 0)
1
>> (fact 10)
3628800
```

Evaluator

Sum

```
>> (fdef sum (f a b)
     (if (> a b)
       0
       (+ (funcall f a)
          (sum f (+ a 1) b))))
```

Product

```
>> (fdef product (f a b)
     (if (> a b)
       1
       (* (funcall f a)
          (product f (+ a 1) b))))
```

Evaluator

Sum

```
>> (fdef sum (f a b)
     (if (> a b)
       0
       (+ (funcall f a)
          (sum f (+ a 1) b))))
```

Product

```
>> (fdef product (f a b)
     (if (> a b)
       1
       (* (funcall f a)
          (product f (+ a 1) b))))
```

Evaluator

Sum

```
>> (fdef sum (f a b)
     (if (> a b)
       0
       (+ (funcall f a)
          (sum f (+ a 1) b))))
```

Product

```
>> (fdef product (f a b)
     (if (> a b)
       1
       (* (funcall f a)
          (product f (+ a 1) b))))
```

Evaluator

Sum

```
>> (fdef sum (f a b)
     (if (> a b)
       0
       (+ (funcall f a)
          (sum f (+ a 1) b))))
```

Product

```
>> (fdef product (f a b)
     (if (> a b)
       1
       (* (funcall f a)
          (product f (+ a 1) b))))
```

Abstraction

- Sums and products are so identical that it is possible to abstract them using additional (functional) parameters.

Evaluator

Abstraction: accumulation (in a Lisp-2)

```
>> (fdef accumulate (combiner neutral f a b)
     (if (> a b)
       neutral
       (funcall combiner
                (funcall f a)
                (accumulate combiner neutral f (+ a 1) b))))
```

Evaluator

Abstraction: accumulation (in a Lisp-2)

```
>> (fdef accumulate (combiner neutral f a b)
     (if (> a b)
       neutral
       (funcall combiner
                (funcall f a)
                (accumulate combiner neutral f (+ a 1) b))))
(function (combiner neutral f a b) ...)
```

Evaluator

Abstraction: accumulation (in a Lisp-2)

```
>> (fdef accumulate (combiner neutral f a b)
     (if (> a b)
       neutral
       (funcall combiner
                (funcall f a)
                (accumulate combiner neutral f (+ a 1) b))))
(function (combiner neutral f a b) ...)
>> (fdef fact (n)
```

Evaluator

Abstraction: accumulation (in a Lisp-2)

```
>> (fdef accumulate (combiner neutral f a b)
     (if (> a b)
       neutral
       (funcall combiner
                (funcall f a)
                (accumulate combiner neutral f (+ a 1) b))))
(function (combiner neutral f a b) ...)
>> (fdef fact (n)
     (accumulate (function *) 1 (function identity) 1 n))
```

Evaluator

Abstraction: accumulation (in a Lisp-2)

```
>> (fdef accumulate (combiner neutral f a b)
     (if (> a b)
       neutral
       (funcall combiner
                (funcall f a)
                (accumulate combiner neutral f (+ a 1) b))))
(function (combiner neutral f a b) ...)
>> (fdef fact (n)
     (accumulate (function *) 1 (function identity) 1 n))
(function (n) ...)
```

Evaluator

Abstraction: accumulation (in a Lisp-2)

```
>> (fdef accumulate (combiner neutral f a b)
     (if (> a b)
       neutral
       (funcall combiner
                (funcall f a)
                (accumulate combiner neutral f (+ a 1) b))))
(function (combiner neutral f a b) ...)
>> (fdef fact (n)
     (accumulate (function *) 1 (function identity) 1 n))
(function (n) ...)
```

Lisp-2 and Functional Programming

- Functional programming in a Lisp-2 tends to spread `funcall` and `function` all over the code.
- This makes the code hard to read.

Evaluator

Abstraction: accumulation (in a Lisp-1)

```
>> (fdef accumulate (combiner neutral f a b)
     (if (> a b)
       neutral
       (combiner
         (f a)
         (accumulate combiner neutral f (+ a 1) b))))
(function (combiner neutral f a b) ...)
>> (fdef fact (n)
     (accumulate * 1 identity 1 n))
(function (n) ...)
```

Lisp-1 and Functional Programming

- Functional programming in a Lisp-1 looks much better (at the price of loosing homonyms).
- Let's change our evaluator to become a Lisp-1.

Evaluator

Lisp-2

```
(define (in-function-namespace name)
  (string->symbol
   (string-append
    "%function-"
    (symbol->string name))))
```

Evaluator

Lisp-1

```
(define (in-function-namespace name)
  name)
```

Evaluator

Lisp-1

```
(define (in-function-namespace name)
  name)
```

Other simplifications

- Drop `function` (and `function-reference?`, `function-reference-name`, `eval-function-reference`).
- Drop `funcall` (and `funcall?`, `funcall-operator`, `funcall-operands`, `eval-funcall`).
- Drop `in-function-namespace`.

Evaluator

Next Integer (away from origin)

```
>> (fdef next (i)
     (if (< i 0)
       (+ i -1)
       (+ i +1)))
```

Evaluator

Next Integer (away from origin)

```
>> (fdef next (i)
     (if (< i 0)
       (+ i -1)
       (+ i +1)))
(function (i) ...)
```

Evaluator

Next Integer (away from origin)

```
>> (fdef next (i)
     (if (< i 0)
       (+ i -1)
       (+ i +1)))
(function (i) ...)
>> (next 3)
```

Evaluator

Next Integer (away from origin)

```
>> (fdef next (i)
     (if (< i 0)
       (+ i -1)
       (+ i +1)))
(function (i) ...)
>> (next 3)
4
```

Evaluator

Next Integer (away from origin)

```
>> (fdef next (i)
     (if (< i 0)
       (+ i -1)
       (+ i +1)))
(function (i) ...)
>> (next 3)
4
>> (next -5)
```

Evaluator

Next Integer (away from origin)

```
>> (fdef next (i)
     (if (< i 0)
       (+ i -1)
       (+ i +1)))
(function (i) ...)
>> (next 3)
4
>> (next -5)
-6
```

Evaluator

Next Integer (away from origin)

```
>> (fdef next (i)
     (if (< i 0)
       (+ i -1)
       (+ i +1)))
(function (i) ...)
>> (next 3)
4
>> (next -5)
-6
```

Conditional Expression

- There are repeated parts that could be factorized...
- ... by moving the conditional expression inside the addition.

Evaluator

Next Integer (away from origin)

```
>> (fdef next (i)
     (if (< i 0)
       (+ i -1)
       (+ i +1)))
(function (i) ...)
>> (next 3)
4
>> (next -5)
-6
>> (fdef next (i)
     (+ i (if (< i 0) -1 +1)))
(function (i) ...)
```

Conditional Expression

- There are repeated parts that could be factorized...
- ... by moving the conditional expression inside the addition.

Evaluator

Next Integer (away from origin) – Alternative Implementation

```
>> (fdef next (i)
     (if (< i 0)
       (- i 1)
       (+ i 1)))
```

Evaluator

Next Integer (away from origin) – Alternative Implementation

```
>> (fdef next (i)
     (if (< i 0)
       (- i 1)
       (+ i 1)))
(function (i) ...)
```

Evaluator

Next Integer (away from origin) – Alternative Implementation

```
>> (fdef next (i)
     (if (< i 0)
       (- i 1)
       (+ i 1)))
(function (i) ...)
>> (fdef next (i)
     ((if (< i 0) - +) i 1))
(function (i) ...)
```

Evaluator

Next Integer (away from origin) – Alternative Implementation

```
>> (fdef next (i)
     (if (< i 0)
       (- i 1)
       (+ i 1)))
(function (i) ...)
>> (fdef next (i)
     ((if (< i 0) - +) i 1))
(function (i) ...)
>> (next 3)
```

Evaluator

Next Integer (away from origin) – Alternative Implementation

```
>> (fdef next (i)
     (if (< i 0)
       (- i 1)
       (+ i 1)))
(function (i) ...)
>> (fdef next (i)
     ((if (< i 0) - +) i 1))
(function (i) ...)
>> (next 3)
error in "Unbound name -- EVAL-NAME": (if (< i 0) - +)
```

Conditional Expression in Operator Position

- We were expecting to see a *name* in the operator position, but we now have an expression there.
- Let's generalize the treatment of the operator.

Evaluator

Function Calls

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((name? exp)
         (eval-name exp env))
        ...
        ((call? exp)
         (eval-call exp env))
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Evaluator

Function Calls

```scheme
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((name? exp)
         (eval-name exp env))
        ...
        ((call? exp)
         (eval-call exp env))
        (else
         (error "Unknown expression type -- EVAL" exp))))

(define (eval-call exp env)
  (let ((func (eval-name (call-operator exp) env))
        (args (eval-exprs (call-operands exp) env)))
    (if (primitive? func)
        (apply-primitive-function func args)
        (let ((extended-environment
               (augment-environment (function-parameters func)
                                    args
                                    env)))
          (eval (function-body func) extended-environment)))))
```

Evaluator

Function Calls

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((name? exp)
         (eval-name exp env))
        ...
        ((call? exp)
         (eval-call exp env))
        (else
         (error "Unknown expression type -- EVAL" exp))))

(define (eval-call exp env)
  (let ((func (eval-name (call-operator exp) env))
        (args (eval-exprs (call-operands exp) env)))
    (if (primitive? func)
        (apply-primitive-function func args)
        (let ((extended-environment
               (augment-environment (function-parameters func)
                                    args
                                    env)))
          (eval (function-body func) extended-environment)))))
```

Evaluator

Function Calls

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((name? exp)
         (eval-name exp env))
        ...
        ((call? exp)
         (eval-call exp env))
        (else
         (error "Unknown expression type -- EVAL" exp))))

(define (eval-call exp env)
  (let ((func (eval (call-operator exp) env))
        (args (eval-exprs (call-operands exp) env)))
    (if (primitive? func)
        (apply-primitive-function func args)
        (let ((extended-environment
               (augment-environment (function-parameters func)
                                    args
                                    env)))
          (eval (function-body func) extended-environment)))))
```

Evaluator

Expressions as Operators

```
(fdef next (i)
  ((if (< i 0) - +) i 1))
```

Lisp-1 and Functional Programming

- In a Lisp-1 it is possible to generalize the operator in a function call:
    - The operator can be an expression that evaluates to a function.
    - Using a name as operator is just a special case of the use of an expression.
- In a Lisp-2 it is also possible to generalize the operator in a function call but semantics become more complex.

Evaluator

Approximating π in a Lisp-1

```
>> (fdef sum (f a b)
     (if (> a b)
       0
       (+ (f a)
          (sum f (+ a 1) b))))
```

Evaluator

Approximating π in a Lisp-1

```
>> (fdef sum (f a b)
     (if (> a b)
       0
       (+ (f a)
          (sum f (+ a 1) b))))
(function (f a b) ...)
```

Evaluator

Approximating π in a Lisp-1

```
>> (fdef sum (f a b)
     (if (> a b)
       0
       (+ (f a)
          (sum f (+ a 1) b))))
(function (f a b) ...)
>> (fdef approx-pi (n)
     (flet ((term (i) (/ (expt -1.0 i) (+ (* 2 i) 1))))
       (* 4 (sum term 0 n))))
```

Evaluator

Approximating π in a Lisp-1

```
>> (fdef sum (f a b)
     (if (> a b)
       0
       (+ (f a)
          (sum f (+ a 1) b))))
(function (f a b) ...)
>> (fdef approx-pi (n)
     (flet ((term (i) (/ (expt -1.0 i) (+ (* 2 i) 1))))
       (* 4 (sum term 0 n))))
(function (n) ...)
```

Evaluator

Approximating π in a Lisp-1

```
>> (fdef sum (f a b)
     (if (> a b)
       0
       (+ (f a)
          (sum f (+ a 1) b))))
(function (f a b) ...)
>> (fdef approx-pi (n)
     (flet ((term (i) (/ (expt -1.0 i) (+ (* 2 i) 1))))
       (* 4 (sum term 0 n))))
(function (n) ...)
>> (approx-pi 100)
```

Evaluator

Approximating π in a Lisp-1

```
>> (fdef sum (f a b)
     (if (> a b)
       0
       (+ (f a)
          (sum f (+ a 1) b))))
(function (f a b) ...)
>> (fdef approx-pi (n)
     (flet ((term (i) (/ (expt -1.0 i) (+ (* 2 i) 1))))
       (* 4 (sum term 0 n))))
(function (n) ...)
>> (approx-pi 100)
3.151493401070991
```

Anonymous Functions

- The function `term` is not useful for other purposes.
- It seems a waste to be forced to give it a name.
- Let's introduce the concept of anonymous function.

Evaluator

Named Functions

```
(fdef approx-pi (n)
  (flet ((term (i) (/ (expt -1.0 i) (+ (* 2 i) 1))))
    (* 4 (sum term
              0 n))))
```

Anonymous Functions

```
(fdef approx-pi (n)
  (* 4 (sum (lambda (i) (/ (expt -1.0 i) (+ (* 2 i) 1)))
            0 n)))
```

Evaluator

Named Functions

```
(fdef approx-pi (n)
  (flet ((term (i) (/ (expt -1.0 i) (+ (* 2 i) 1))))
    (* 4 (sum term
              0 n))))
```

Anonymous Functions

```
(fdef approx-pi (n)
  (* 4 (sum (lambda (i) (/ (expt -1.0 i) (+ (* 2 i) 1)))
            0 n)))
```

Evaluator

Named Functions

```
(fdef approx-pi (n)
  (flet ((term (i) (/ (expt -1.0 i) (+ (* 2 i) 1))))
    (* 4 (sum term
              0 n))))
```

Anonymous Functions

```
(fdef approx-pi (n)
  (* 4 (sum (lambda (i) (/ (expt -1.0 i) (+ (* 2 i) 1)))
            0 n)))
```

Lambda Forms

- A *lambda form* evaluates to a function without requiring an explicit name for the function.
- The function is *anonymous*.

Evaluator

Evaluator

```
;;(lambda (a b)
;;  (+ a b))
```

Evaluator

Evaluator

```
;;(lambda (a b)
;;  (+ a b))

(define (lambda? exp)
  (and (pair? exp)
       (eq? (car exp) 'lambda)))
```

Evaluator

Evaluator

```
;;(lambda (a b)
;;  (+ a b))

(define (lambda? exp)
  (and (pair? exp)
       (eq? (car exp) 'lambda)))

(define (lambda-parameters exp)
  (cadr exp))
```

Evaluator

Evaluator

```
;;(lambda (a b)
;;  (+ a b))

(define (lambda? exp)
  (and (pair? exp)
       (eq? (car exp) 'lambda)))

(define (lambda-parameters exp)
  (cadr exp))

(define (lambda-body exp)
  (cddr exp))
```

Evaluator

Evaluator

```
;;(lambda (a b)
;;  (+ a b))

(define (lambda? exp)
  (and (pair? exp)
       (eq? (car exp) 'lambda)))

(define (lambda-parameters exp)
  (cadr exp))

(define (lambda-body exp)
  (cddr exp))

(define (eval-lambda exp env)
  (make-function (lambda-parameters exp)
                 (lambda-body exp)))
```

Evaluator

Evaluator

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((name? exp)
         (eval-name exp env))


        ((if? exp)
         (eval-if exp env))
        ((let? exp)
         (eval-let exp env))
        ((flet? exp)
         (eval-flet exp env))
        ((def? exp)
         (eval-def exp env))
        ((set? exp)
         (eval-set exp env))
        ((fdef? exp)
         (eval-fdef exp env))
        ...
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Evaluator

Evaluator

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((name? exp)
         (eval-name exp env))
        ((lambda? exp)

        ((if? exp)
         (eval-if exp env))
        ((let? exp)
         (eval-let exp env))
        ((flet? exp)
         (eval-flet exp env))
        ((def? exp)
         (eval-def exp env))
        ((set? exp)
         (eval-set exp env))
        ((fdef? exp)
         (eval-fdef exp env))
        ...
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Evaluator

Evaluator

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((name? exp)
         (eval-name exp env))
        ((lambda? exp)
         (eval-lambda exp env))
        ((if? exp)
         (eval-if exp env))
        ((let? exp)
         (eval-let exp env))
        ((flet? exp)
         (eval-flet exp env))
        ((def? exp)
         (eval-def exp env))
        ((set? exp)
         (eval-set exp env))
        ((fdef? exp)
         (eval-fdef exp env))
        ...
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Evaluator

Approximating e

$$e = \sum_{i=0}^{\infty} \frac{1}{i!}$$

Approximating e

```
>> (fdef fact (n)
     (accumulate * 1 (lambda (i) i) 1 n))
```

Evaluator

Approximating e

$$e = \sum_{i=0}^{\infty} \frac{1}{i!}$$

Approximating e

```
>> (fdef fact (n)
     (accumulate * 1 (lambda (i) i) 1 n))
(function (n) ...)
```

Evaluator

Approximating e

$$e = \sum_{i=0}^{\infty} \frac{1}{i!}$$

Approximating e

```
>> (fdef fact (n)
     (accumulate * 1 (lambda (i) i) 1 n))
(function (n) ...)
>> (fdef approx-e (n)
     (sum (lambda (i) (/ 1.0 (fact i))) 0 n))
```

Evaluator

Approximating e

$$e = \sum_{i=0}^{\infty} \frac{1}{i!}$$

Approximating e

```
>> (fdef fact (n)
     (accumulate * 1 (lambda (i) i) 1 n))
(function (n) ...)
>> (fdef approx-e (n)
     (sum (lambda (i) (/ 1.0 (fact i))) 0 n))
(function (n) ...)
```

Evaluator

Approximating e

$$e = \sum_{i=0}^{\infty} \frac{1}{i!}$$

Approximating e

```
>> (fdef fact (n)
     (accumulate * 1 (lambda (i) i) 1 n))
(function (n) ...)
>> (fdef approx-e (n)
     (sum (lambda (i) (/ 1.0 (fact i))) 0 n))
(function (n) ...)
>> (approx-e 10)
```

Evaluator

Approximating *e*

$$e = \sum_{i=0}^{\infty} \frac{1}{i!}$$

Approximating *e*

```
>> (fdef fact (n)
     (accumulate * 1 (lambda (i) i) 1 n))
(function (n) ...)
>> (fdef approx-e (n)
     (sum (lambda (i) (/ 1.0 (fact i))) 0 n))
(function (n) ...)
>> (approx-e 10)
2.7182818011463845
```

Evaluator

Approximating *e*

$$e = \sum_{i=0}^{\infty} \frac{1}{i!}$$

Approximating *e*

```
>> (fdef fact (n)
     (accumulate * 1 (lambda (i) i) 1 n))
(function (n) ...)
>> (fdef approx-e (n)
     (sum (lambda (i) (/ 1.0 (fact i))) 0 n))
(function (n) ...)
>> (approx-e 10)
2.7182818011463845
>> (approx-e 100)
```

Evaluator

Approximating *e*

$$e = \sum_{i=0}^{\infty} \frac{1}{i!}$$

Approximating *e*

```
>> (fdef fact (n)
     (accumulate * 1 (lambda (i) i) 1 n))
(function (n) ...)
>> (fdef approx-e (n)
     (sum (lambda (i) (/ 1.0 (fact i))) 0 n))
(function (n) ...)
>> (approx-e 10)
2.7182818011463845
>> (approx-e 100)
2.718281828459045
```

Evaluator

Approximating e

$$e = \sum_{i=0}^{\infty} \frac{1}{i!}$$

Approximating e

```
>> (fdef fact (n)
     (accumulate * 1 (lambda (i) i) 1 n))
(function (n) ...)
>> (fdef approx-e (n)
     (sum (lambda (i) (/ 1.0 (fact i))) 0 n))
(function (n) ...)
>> (approx-e 10)
2.718281801146385
>> (approx-e 100)
2.71828182845904
>> (approx-e 1000)
```

Evaluator

Approximating *e*

$$e = \sum_{i=0}^{\infty} \frac{1}{i!}$$

Approximating *e*

```
>> (fdef fact (n)
     (accumulate * 1 (lambda (i) i) 1 n))
(function (n) ...)
>> (fdef approx-e (n)
     (sum (lambda (i) (/ 1.0 (fact i))) 0 n))
(function (n) ...)
>> (approx-e 10)
2.718281801146385
>> (approx-e 100)
2.71828182845904
>> (approx-e 1000)
2.71828182845904
```

Evaluator

A Striking Resemblance

- A `def` form associates a name to the evaluation of a form, e.g., `(def foo (* 2 2))`.
- A `fdef` form associates a name to a function, e.g., `(fdef square (x) (* x x))`.
- A `lambda` form evaluates to a function, e.g., `(lambda (x) (* x x))`
- Thus, we have the following equivalence: `(fdef square (x) (* x x))` = `(def square (lambda (x) (* x x)))`
- Or, in more abstract terms: `fdef` = `def` + `lambda`
- The same equivalence occurs with `flet` = `let` + `lambda`.

Evaluator

fdef

fdef = def + lambda

From

```
(fdef square (x)
  (* x x))
```

To

```
(def square
  (lambda (x)
    (* x x)))
```

Evaluator

fdef

fdef = def + lambda

From

```
(fdef square (x)
  (* x x))
```

To

```
(def square
  (lambda (x)
    (* x x)))
```

Transformation (without *backquote*)

```
(define (eval-fdef exp env)
  (eval (list 'def
              (cadr exp)
              (cons 'lambda (cddr exp)))
        env))
```

Evaluator

fdef

fdef = def + lambda

From

```
(fdef square (x)
  (* x x))
```

To

```
(def square
  (lambda (x)
    (* x x)))
```

Transformation (with *backquote*)

```
(define (eval-fdef exp env)
  (eval `(def ,(cadr exp)
              (lambda ,@(cddr exp)))
        env))
```

Evaluator

flet

flet = let + lambda

From

```
(flet ((foo (x)
         (+ x 1))
       (bar (a b)
         (* a b)))
  (foo (bar 2 3)))
```

To

```
(let ((foo (lambda (x)
             (+ x 1)))
      (bar (lambda (a b)
             (* a b))))
  (foo (bar 2 3)))
```

Evaluator

flet

flet = let + lambda

From

```
(flet ((foo (x)
         (+ x 1))
       (bar (a b)
         (* a b)))
   (foo (bar 2 3)))
```

To

```
(let ((foo (lambda (x)
             (+ x 1)))
      (bar (lambda (a b)
             (* a b))))
   (foo (bar 2 3)))
```

Transformation

```
(define (eval-flet exp env)
  (eval `(let ,(map (lambda (form)
                      `(,(car form) (lambda ,@(cdr form))))
                    (cadr exp))
           ,@(cddr exp))
        env))
```

Evaluator

let

let = lambda + function call

From

```
(let ((x (+ 1 2))
      (y (* 2 3)))
  (- x y))
```

To

```
((lambda (x y)
   (- x y))
 (+ 1 2)
 (* 2 3))
```

Evaluator

let

let = lambda + function call

From

```
(let ((x (+ 1 2))
      (y (* 2 3)))
  (- x y))
```

To

```
((lambda (x y)
   (- x y))
 (+ 1 2)
 (* 2 3))
```

Transformation

```
(define (eval-let exp env)
  (eval `((lambda ,(map car (cadr exp))
            ,@(cddr exp))
          ,@(map cadr (cadr exp)))
        env))
```

Evaluator

Source to Source Transformations

- Some forms of the language can be implemented as combinations of more primitive forms.
- This implementation can be done as *source-to-source* transformations.
- Lambda expressions (function abstraction) and function application are two of the most fundamental forms:
  - In theory, everything can be implemented on top of these two forms (Church numerals, Church booleans, Church pairs, etc).
  - In practice, all primitive data types are implemented using low-level capabilities.
- Source-to-source transformations might also be useful for the programmer. We will return to this later.

Evaluator

Second Order Sum

$$\sum_{i=i_0}^{i_1} f(i) \qquad \text{where} \qquad f(x) = ax^2 + bx + c$$

Second Order Sum

Evaluator

Second Order Sum

$$\sum_{i=i_0}^{i_1} f(i) \qquad \text{where} \qquad f(x) = ax^2 + bx + c$$

Second Order Sum

```
>> (fdef sum (f a b)
     (if (> a b)
       0
       (+ (f a)
          (sum f (+ a 1) b))))
```

Evaluator

Second Order Sum

$$\sum_{i=i_0}^{i_1} f(i) \qquad \text{where} \qquad f(x) = ax^2 + bx + c$$

Second Order Sum

```
>> (fdef sum (f a b)
     (if (> a b)
       0
       (+ (f a)
          (sum f (+ a 1) b))))
(function (f a b) ...)
```

Evaluator

Second Order Sum

$$\sum_{i=i_0}^{i_1} f(i) \qquad \text{where} \qquad f(x) = ax^2 + bx + c$$

Second Order Sum

```
>> (fdef sum (f a b)
     (if (> a b)
       0
       (+ (f a)
          (sum f (+ a 1) b))))
(function (f a b) ...)
>> (fdef second-order-sum (a b c i0 i1)
     (sum (lambda (x)
            (+ (* a x x) (* b x) c))
          i0 i1))
```

Evaluator

Second Order Sum

$$\sum_{i=i_0}^{i_1} f(i) \qquad \text{where} \qquad f(x) = ax^2 + bx + c$$

Second Order Sum

```
>> (fdef sum (f a b)
     (if (> a b)
       0
       (+ (f a)
          (sum f (+ a 1) b))))
(function (f a b) ...)
>> (fdef second-order-sum (a b c i0 i1)
     (sum (lambda (x)
            (+ (* a x x) (* b x) c))
          i0 i1))
(function (a b c i0 i1) ...)
```

Evaluator

Second Order Sum

$$\sum_{x=-1}^{1} 10x^2 + 5x = (10 - 5) + 0 + (10 + 5) = 20$$

Second Order Sum

```
>> (fdef sum (f a b)
     (if (> a b)
       0
       (+ (f a)
          (sum f (+ a 1) b))))
(function (f a b) ...)
>> (fdef second-order-sum (a b c i0 i1)
     (sum (lambda (x)
            (+ (* a x x) (* b x) c))
          i0 i1))
(function (a b c i0 i1) ...)
```

Evaluator

Second Order Sum

$$\sum_{x=-1}^{1} 10x^2 + 5x = (10 - 5) + 0 + (10 + 5) = 20$$

Second Order Sum

```
>> (fdef sum (f a b)
     (if (> a b)
       0
       (+ (f a)
          (sum f (+ a 1) b))))
(function (f a b) ...)
>> (fdef second-order-sum (a b c i0 i1)
     (sum (lambda (x)
            (+ (* a x x) (* b x) c))
          i0 i1))
(function (a b c i0 i1) ...)
>> (second-order-sum 10 5 0 -1 +1)
```

Evaluator

Second Order Sum

$$\sum_{x=-1}^{1} 10x^2 + 5x = (10-5) + 0 + (10+5) = 20$$

Second Order Sum

```
>> (fdef sum (f a b)
     (if (> a b)
       0
       (+ (f a)
          (sum f (+ a 1) b))))
(function (f a b) ...)
>> (fdef second-order-sum (a b c i0 i1)
     (sum (lambda (x)
            (+ (* a x x) (* b x) c))
          i0 i1))
(function (a b c i0 i1) ...)
>> (second-order-sum 10 5 0 -1 +1)
0 ;;What???
```

Evaluator

Dynamic Scope

- So far, we didn't think carefully about the *scope* of our binding constructs:
    - A `def` form creates a binding in the *current* environment.
    - A function call creates a set of bindings (for the parameters) that extends the *current* environment.
    - At the end of the call, the extended environment is discarded.
    - A name is resolved by searching the *current* environment.
- This scoping discipline is called *dynamic scope*.
- In dynamic scope, the environment grows and shrinks in parallel with the call stack.
- Dynamic scope is used in many languages such as EmacsLisp, AutoLisp, Logo, TEX, Perl (`local` variables) and Common Lisp (`special` variables).

Evaluator

Downwards Funarg Problem

- Passing a function with free variables as argument causes the *downwards funarg problem*...
- ... where the function is called in an environment where the free variables might be shadowed by other variables of the calling context.

Solving the Downwards Funarg Problem

- Functions must know the correct environment for resolving the free variables.
- One potential solution might be to use name-mangling techniques to avoid name clashes.

Evaluator

Higher-order Functions in a Dynamically Scoped Language

```
>> (fdef sum (f a b)
     (if (> a b)
       0
       (+ (f a)
          (sum f (+ a 1) b))))
(function (f a b) ...)
>> (fdef second-order-sum (a b c i0 i1)
     (sum (lambda (x)
            (+ (* a x x) (* b x) c))
          i0 i1))
(function (a b c i0 i1) ...)
>> (second-order-sum 10 5 0 -1 +1)
0 ;;What???
```

Evaluator

Higher-order Functions in a Dynamically Scoped Language

```
>> (fdef sum (%*f*% %*a*% %*b*%)
     (if (> %*a*% %*b*%)
       0
       (+ (%*f*% %*a*%)
          (sum %*f*% (+ %*a*% 1) %*b*%))))
(function (%*f*% %*a*% %*b*%) ...)
>> (fdef second-order-sum (a b c i0 i1)
     (sum (lambda (x)
            (+ (* a x x) (* b x) c))
          i0 i1))
(function (a b c i0 i1) ...)
>> (second-order-sum 10 5 0 -1 +1)
20
```

Name Mangling Problems

- It makes the code hard to read.
- It doesn't really solve the problem.

Evaluator

Downwards Funarg Problem

- The correct solution is to associate the downward function to the scope where the function was created:
  - That scope is still active when the passed function is called...
  - ...unless it is possible to store the function somewhere and call it later.
- Some languages solve this last problem by restricting the language:
  - Pascal forbids storing functions in variables.
  - C forbids inner functions.
- Other languages (e.g., Scheme, Haskel, Common Lisp) implement environments with indefinite extent.
- But there is a more difficult problem.

Evaluator

Higher-order Functions in a Dynamically Scoped Language

```
>> (fdef compose (f g)
     (lambda (x)
       (f (g x))))
```

Evaluator

Higher-order Functions in a Dynamically Scoped Language

```
>> (fdef compose (f g)
     (lambda (x)
       (f (g x))))
(function (f g) ...)
```

Evaluator

Higher-order Functions in a Dynamically Scoped Language

```
>> (fdef compose (f g)
     (lambda (x)
       (f (g x))))
(function (f g) ...)
>> (fdef foo (x) (+ x 5))
```

Evaluator

Higher-order Functions in a Dynamically Scoped Language

```
>> (fdef compose (f g)
     (lambda (x)
       (f (g x))))
(function (f g) ...)
>> (fdef foo (x) (+ x 5))
(function (x) (+ x 5))
```

Evaluator

Higher-order Functions in a Dynamically Scoped Language

```
>> (fdef compose (f g)
     (lambda (x)
       (f (g x))))
(function (f g) ...)
>> (fdef foo (x) (+ x 5))
(function (x) (+ x 5))
>> (fdef bar (x) (* x 3))
```

Evaluator

Higher-order Functions in a Dynamically Scoped Language

```
>> (fdef compose (f g)
     (lambda (x)
       (f (g x))))
(function (f g) ...)
>> (fdef foo (x) (+ x 5))
(function (x) (+ x 5))
>> (fdef bar (x) (* x 3))
(function (x) (* x 3))
```

Evaluator

Higher-order Functions in a Dynamically Scoped Language

```
>> (fdef compose (f g)
     (lambda (x)
       (f (g x))))
(function (f g) ...)
>> (fdef foo (x) (+ x 5))
(function (x) (+ x 5))
>> (fdef bar (x) (* x 3))
(function (x) (* x 3))
>> (def foobar (compose foo bar))
```

Evaluator

Higher-order Functions in a Dynamically Scoped Language

```
>> (fdef compose (f g)
     (lambda (x)
       (f (g x))))
(function (f g) ...)
>> (fdef foo (x) (+ x 5))
(function (x) (+ x 5))
>> (fdef bar (x) (* x 3))
(function (x) (* x 3))
>> (def foobar (compose foo bar))
(function (x) (f (g x)))
```

Evaluator

Higher-order Functions in a Dynamically Scoped Language

```
>> (fdef compose (f g)
     (lambda (x)
       (f (g x))))
(function (f g) ...)
>> (fdef foo (x) (+ x 5))
(function (x) (+ x 5))
>> (fdef bar (x) (* x 3))
(function (x) (* x 3))
>> (def foobar (compose foo bar))
(function (x) (f (g x)))
>> (foobar 2)
```

Evaluator

Higher-order Functions in a Dynamically Scoped Language

```
>> (fdef compose (f g)
     (lambda (x)
       (f (g x))))
(function (f g) ...)
>> (fdef foo (x) (+ x 5))
(function (x) (+ x 5))
>> (fdef bar (x) (* x 3))
(function (x) (* x 3))
>> (def foobar (compose foo bar))
(function (x) (f (g x)))
>> (foobar 2)
error in "Unbound name -- EVAL-NAME": f
```

Evaluator

Upwards Funarg Problem

- Returning a function with free variables as result causes the *upwards funarg problem*...
- ... where the function is called in an environment where the free variables are not longer bound (or are bound to different values).

Solving the Upwards Funarg Problem

- Some languages avoid the upwards funarg problem by restricting the language:
    - Pascal forbids functional returns.
    - C forbids inner functions.
    - Java forbids non-`final` free variables in anonymous inner classes.
- Other languages (e.g., Scheme, Haskel, Common Lisp) implement environments with indefinite extent.

Evaluator

Evaluator

```
(define (make-function parameters body)
  (cons 'function
        (cons parameters
              body)))


(define (function? obj)
  (and (pair? obj)
       (eq? (car obj) 'function)))

(define (function-parameters func)
  (cadr func))

(define (function-body func)
  (make-begin (cddr func)))
```

Evaluator

Evaluator

```
(define (make-function parameters body env)
  (cons 'function
        (cons parameters
              (cons env
                    body))))

(define (function? obj)
  (and (pair? obj)
       (eq? (car obj) 'function)))

(define (function-parameters func)
  (cadr func))

(define (function-body func)
  (make-begin (cdddr func)))
```

Evaluator

Evaluator

```
(define (make-function parameters body env)
  (cons 'function
        (cons parameters
              (cons env
                    body))))

(define (function? obj)
  (and (pair? obj)
       (eq? (car obj) 'function)))

(define (function-parameters func)
  (cadr func))

(define (function-body func)
  (make-begin (cdddr func)))

(define (function-environment func)
  (caddr func))
```

Evaluator

Evaluator

```
(define (eval-lambda exp env)
  (make-function (lambda-parameters exp)
                 (lambda-body exp)))


(define (eval-call exp env)
  (let ((func (eval (call-operator exp) env))
        (args (eval-exprs (call-operands exp) env)))
    (if (primitive? func)
        (apply-primitive-function func args)
        (let ((extended-environment
               (augment-environment (function-parameters func)
                                    args
                                    env)))
          (eval (function-body func) extended-environment)))))
```

Evaluator

Evaluator

```
(define (eval-lambda exp env)
  (make-function (lambda-parameters exp)
                 (lambda-body exp)
                 env))

(define (eval-call exp env)
  (let ((func (eval (call-operator exp) env))
        (args (eval-exprs (call-operands exp) env)))
    (if (primitive? func)
        (apply-primitive-function func args)
        (let ((extended-environment
               (augment-environment (function-parameters func)
                                    args
                                    (function-environment func))))
          (eval (function-body func) extended-environment)))))
```

Evaluator

Higher-order Functions in a Lexically Scoped Language

```
>> (fdef sum (f a b)
     (if (> a b)
       0
       (+ (f a)
          (sum f (+ a 1) b))))
(function (f a b) ...)
>> (fdef second-order-sum (a b c i0 i1)
     (sum (lambda (x)
            (+ (* a x x) (* b x) c))
          i0 i1))
(function (a b c i0 i1) ...)
>> (second-order-sum 10 5 0 -1 +1)
20 ;;As expected
```

Evaluator

Higher-order Functions in a Lexically Scoped Language

```
>> (fdef compose (f g)
     (lambda (x)
       (f (g x))))
(function (f g) ...)
>> (fdef foo (x) (+ x 5))
(function (x) (+ x 5))
>> (fdef bar (x) (* x 3))
(function (x) (* x 3))
>> (def foobar (compose foo bar))
(function (x) (f (g x)))
>> (foobar 2)
11 ;;As expected
```

Evaluator

Local Recursive Functions in a Lexically Scoped Language

```
>> (flet ((fact (n)
            (if (= n 0)
              1
              (* n (fact (- n 1))))))
     (fact 3))
```

Evaluator

Local Recursive Functions in a Lexically Scoped Language

```
>> (flet ((fact (n)
            (if (= n 0)
              1
              (* n (fact (- n 1))))))
     (fact 3))
error in "Unbound name -- EVAL-NAME": fact
```

Problem

- `flet` creates a function and establishes a scope where a name is bound to that function.
- But the function is created outside that scope.
- As a result, the function cannot refer itself.

Evaluator

Local Recursive Functions in a Lexically Scoped Language

```
>> (let ((fact #f))
     (set! fact (lambda (n)
                  (if (= n 0)
                    1
                    (* n (fact (- n 1))))))
     (fact 3))
```

Evaluator

Local Recursive Functions in a Lexically Scoped Language

```
>> (let ((fact #f))
     (set! fact (lambda (n)
                  (if (= n 0)
                    1
                    (* n (fact (- n 1))))))
     (fact 3))
6
```

Solution (the Common Lisp way – `labels`)

1. Creates the scope for the function name (temporarily binding it to some value).
2. Creates the function inside that scope.
3. Modifies the binding of the function name to point to the function.

Evaluator

Local Recursive Functions in a Lexically Scoped Language

```
>> (let ()
     (fdef fact (n)
       (if (= n 0)
         1
         (* n (fact (- n 1)))))
     (fact 3))
6
```

Solution (the Scheme way – `define`)

1. Creates an empty scope.
2. Creates the function inside that scope.
3. Modifies the scope, adding a new binding from the function name to the function.

Evaluator

Example: Pairs

```
>> (fdef kons (a b)
     (lambda (f)
       (f a b)))
```

Evaluator

Example: Pairs

```
>> (fdef kons (a b)
     (lambda (f)
       (f a b)))
(function (a b) ...)
```

Evaluator

Example: Pairs

```
>> (fdef kons (a b)
     (lambda (f)
       (f a b)))
(function (a b) ...)
>> (fdef kar (k)
     (k (lambda (x y) x)))
```

Evaluator

Example: Pairs

```
>> (fdef kons (a b)
     (lambda (f)
       (f a b)))
(function (a b) ...)
>> (fdef kar (k)
     (k (lambda (x y) x)))
(function (k) ...)
```

Evaluator

Example: Pairs

```
>> (fdef kons (a b)
     (lambda (f)
       (f a b)))
(function (a b) ...)
>> (fdef kar (k)
     (k (lambda (x y) x)))
(function (k) ...)
>> (fdef kdr (k)
     (k (lambda (x y) y)))
```

Evaluator

Example: Pairs

```
>> (fdef kons (a b)
     (lambda (f)
       (f a b)))
(function (a b) ...)
>> (fdef kar (k)
     (k (lambda (x y) x)))
(function (k) ...)
>> (fdef kdr (k)
     (k (lambda (x y) y)))
(function (k) ...)
```

Evaluator

Example: Pairs

```
>> (fdef kons (a b)
     (lambda (f)
       (f a b)))
(function (a b) ...)
>> (fdef kar (k)
     (k (lambda (x y) x)))
(function (k) ...)
>> (fdef kdr (k)
     (k (lambda (x y) y)))
(function (k) ...)
>> (def number-pair (kons 1 2))
```

Evaluator

Example: Pairs

```
>> (fdef kons (a b)
     (lambda (f)
       (f a b)))
(function (a b) ...)
>> (fdef kar (k)
     (k (lambda (x y) x)))
(function (k) ...)
>> (fdef kdr (k)
     (k (lambda (x y) y)))
(function (k) ...)
>> (def number-pair (kons 1 2))
(function (f) ...)
```

Evaluator

Example: Pairs

```
>> (fdef kons (a b)
     (lambda (f)
       (f a b)))
(function (a b) ...)
>> (fdef kar (k)
     (k (lambda (x y) x)))
(function (k) ...)
>> (fdef kdr (k)
     (k (lambda (x y) y)))
(function (k) ...)
>> (def number-pair (kons 1 2))
(function (f) ...)
>> (kar number-pair)
```

Evaluator

Example: Pairs

```
>> (fdef kons (a b)
     (lambda (f)
       (f a b)))
(function (a b) ...)
>> (fdef kar (k)
     (k (lambda (x y) x)))
(function (k) ...)
>> (fdef kdr (k)
     (k (lambda (x y) y)))
(function (k) ...)
>> (def number-pair (kons 1 2))
(function (f) ...)
>> (kar number-pair)
1
```

Evaluator

Example: Pairs

```
>> (fdef kons (a b)
     (lambda (f)
       (f a b)))
(function (a b) ...)
>> (fdef kar (k)
     (k (lambda (x y) x)))
(function (k) ...)
>> (fdef kdr (k)
     (k (lambda (x y) y)))
(function (k) ...)
>> (def number-pair (kons 1 2))
(function (f) ...)
>> (kar number-pair)
1
>> (kdr number-pair)
```

Evaluator

Example: Pairs

```
>> (fdef kons (a b)
     (lambda (f)
       (f a b)))
(function (a b) ...)
>> (fdef kar (k)
     (k (lambda (x y) x)))
(function (k) ...)
>> (fdef kdr (k)
     (k (lambda (x y) y)))
(function (k) ...)
>> (def number-pair (kons 1 2))
(function (f) ...)
>> (kar number-pair)
1
>> (kdr number-pair)
2
```

Evaluator

Example: Pairs

```
>> (def extra-number-pairs
     (kons (kons 1 2)
           (kons 3 4)))
```

Evaluator

Example: Pairs

```
>> (def extra-number-pairs
     (kons (kons 1 2)
           (kons 3 4)))
(function (f) ...)
```

Evaluator

Example: Pairs

```
>> (def extra-number-pairs
     (kons (kons 1 2)
           (kons 3 4)))
(function (f) ...)
>> (kar (kdr extra-number-pairs))
```

Evaluator

Example: Pairs

```
>> (def extra-number-pairs
     (kons (kons 1 2)
           (kons 3 4)))
(function (f) ...)
>> (kar (kdr extra-number-pairs))
3
```

Evaluator

Example: Pairs

```
>> (def extra-number-pairs
     (kons (kons 1 2)
           (kons 3 4)))
(function (f) ...)
>> (kar (kdr extra-number-pairs))
3
>> (def kadr (compose kar kdr))
(function (x) ...)
```

Evaluator

Example: Pairs

```
>> (def extra-number-pairs
     (kons (kons 1 2)
           (kons 3 4)))
(function (f) ...)
>> (kar (kdr extra-number-pairs))
3
>> (def kadr (compose kar kdr))
(function (x) ...)
>> (kadr extra-number-pairs)
```

Evaluator

Example: Pairs

```
>> (def extra-number-pairs
     (kons (kons 1 2)
           (kons 3 4)))
(function (f) ...)
>> (kar (kdr extra-number-pairs))
3
>> (def kadr (compose kar kdr))
(function (x) ...)
>> (kadr extra-number-pairs)
3
```

Evaluator

Example: Pairs

```
>> (def extra-number-pairs
     (kons (kons 1 2)
           (kons 3 4)))
(function (f) ...)
>> (kar (kdr extra-number-pairs))
3
>> (def kadr (compose kar kdr))
(function (x) ...)
>> (kadr extra-number-pairs)
3
>> (def kdar (compose kdr kar))
(function (x) ...)
```

Evaluator

Example: Pairs

```
>> (def extra-number-pairs
     (kons (kons 1 2)
           (kons 3 4)))
(function (f) ...)
>> (kar (kdr extra-number-pairs))
3
>> (def kadr (compose kar kdr))
(function (x) ...)
>> (kadr extra-number-pairs)
3
>> (def kdar (compose kdr kar))
(function (x) ...)
>> (kdar extra-number-pairs)
```

Evaluator

Example: Pairs

```
>> (def extra-number-pairs
     (kons (kons 1 2)
           (kons 3 4)))
(function (f) ...)
>> (kar (kdr extra-number-pairs))
3
>> (def kadr (compose kar kdr))
(function (x) ...)
>> (kadr extra-number-pairs)
3
>> (def kdar (compose kdr kar))
(function (x) ...)
>> (kdar extra-number-pairs)
2
```

Evaluator

Example: Lists

```
>> (fdef empty-list ()
     #f)
```

Evaluator

Example: Lists

```
>> (fdef empty-list ()
     #f)
(function () ...)
```

Evaluator

Example: Lists

```
>> (fdef empty-list ()
     #f)
(function () ...)
>> (fdef empty-list? (l)
     (not l))
```

Evaluator

Example: Lists

```
>> (fdef empty-list ()
     #f)
(function () ...)
>> (fdef empty-list? (l)
     (not l))
(function (l) ...)
```

Evaluator

Example: Lists

```
>> (fdef empty-list ()
     #f)
(function () ...)
>> (fdef empty-list? (l)
     (not l))
(function (l) ...)
>> (def join kons)
```

Evaluator

Example: Lists

```
>> (fdef empty-list ()
     #f)
(function () ...)
>> (fdef empty-list? (l)
     (not l))
(function (l) ...)
>> (def join kons)
(function (a b) ...)
```

Evaluator

Example: Lists

```
>> (fdef empty-list ()
     #f)
(function () ...)
>> (fdef empty-list? (l)
     (not l))
(function (l) ...)
>> (def join kons)
(function (a b) ...)
>> (def first kar)
```

Evaluator

Example: Lists

```
>> (fdef empty-list ()
     #f)
(function () ...)
>> (fdef empty-list? (l)
     (not l))
(function (l) ...)
>> (def join kons)
(function (a b) ...)
>> (def first kar)
(function (k) ...)
```

Evaluator

Example: Lists

```
>> (fdef empty-list ()
     #f)
(function () ...)
>> (fdef empty-list? (l)
     (not l))
(function (l) ...)
>> (def join kons)
(function (a b) ...)
>> (def first kar)
(function (k) ...)
>> (def rest kdr)
```

Evaluator

Example: Lists

```
>> (fdef empty-list ()
     #f)
(function () ...)
>> (fdef empty-list? (l)
     (not l))
(function (l) ...)
>> (def join kons)
(function (a b) ...)
>> (def first kar)
(function (k) ...)
>> (def rest kdr)
(function (k) ...)
```

Evaluator

Example: Lists

```
>> (fdef empty-list ()
     #f)
(function () ...)
>> (fdef empty-list? (l)
     (not l))
(function (l) ...)
>> (def join kons)
(function (a b) ...)
>> (def first kar)
(function (k) ...)
>> (def rest kdr)
(function (k) ...)
>> (def numbers (join 1 (join 2 (join 3 (empty-list)))))
```

Evaluator

Example: Lists

```
>> (fdef empty-list ()
     #f)
(function () ...)
>> (fdef empty-list? (l)
     (not l))
(function (l) ...)
>> (def join kons)
(function (a b) ...)
>> (def first kar)
(function (k) ...)
>> (def rest kdr)
(function (k) ...)
>> (def numbers (join 1 (join 2 (join 3 (empty-list)))))
(function (f) ...)
```

Evaluator

Example: Lists

```
>> (fdef empty-list ()
     #f)
(function () ...)
>> (fdef empty-list? (l)
     (not l))
(function (l) ...)
>> (def join kons)
(function (a b) ...)
>> (def first kar)
(function (k) ...)
>> (def rest kdr)
(function (k) ...)
>> (def numbers (join 1 (join 2 (join 3 (empty-list)))))
(function (f) ...)
>> (first numbers)
```

Evaluator

Example: Lists

```
>> (fdef empty-list ()
     #f)
(function () ...)
>> (fdef empty-list? (l)
     (not l))
(function (l) ...)
>> (def join kons)
(function (a b) ...)
>> (def first kar)
(function (k) ...)
>> (def rest kdr)
(function (k) ...)
>> (def numbers (join 1 (join 2 (join 3 (empty-list)))))
(function (f) ...)
>> (first numbers)
1
```

Evaluator

Example: Lists

```
>> (fdef empty-list ()
     #f)
(function () ...)
>> (fdef empty-list? (l)
     (not l))
(function (l) ...)
>> (def join kons)
(function (a b) ...)
>> (def first kar)
(function (k) ...)
>> (def rest kdr)
(function (k) ...)
>> (def numbers (join 1 (join 2 (join 3 (empty-list)))))
(function (f) ...)
>> (first numbers)
1
>> (first (rest numbers))
```

Evaluator

Example: Lists

```
>> (fdef empty-list ()
     #f)
(function () ...)
>> (fdef empty-list? (l)
     (not l))
(function (l) ...)
>> (def join kons)
(function (a b) ...)
>> (def first kar)
(function (k) ...)
>> (def rest kdr)
(function (k) ...)
>> (def numbers (join 1 (join 2 (join 3 (empty-list)))))
(function (f) ...)
>> (first numbers)
1
>> (first (rest numbers))
2
```

Evaluator

Example: Lists

```
>> (fdef print-list (l)
     (display "(")
     (print-list-elements l)
     (display ")")
     (newline))
```

Evaluator

Example: Lists

```
>> (fdef print-list (l)
     (display "(")
     (print-list-elements l)
     (display ")")
     (newline))
(function (l) ...)
```

Evaluator

Example: Lists

```
>> (fdef print-list (l)
     (display "(")
     (print-list-elements l)
     (display ")")
     (newline))
(function (l) ...)
>> (fdef print-list-elements (l)
     (if (empty-list? l)
       #f
       (begin
         (display (first l))
         (if (empty-list? (rest l))
           #f
           (begin
             (display " ")
             (print-list-elements (rest l)))))))
```

Evaluator

Example: Lists

```
>> (fdef print-list (l)
     (display "(")
     (print-list-elements l)
     (display ")")
     (newline))
(function (l) ...)
>> (fdef print-list-elements (l)
     (if (empty-list? l)
       #f
       (begin
         (display (first l))
         (if (empty-list? (rest l))
           #f
           (begin
             (display " ")
             (print-list-elements (rest l)))))))
(function (l) ...)
```

Evaluator

Example: Lists

```
>> (fdef print-list (l)
     (display "(")
     (print-list-elements l)
     (display ")")
     (newline))
(function (l) ...)
>> (fdef print-list-elements (l)
     (if (empty-list? l)
       #f
       (begin
         (display (first l))
         (if (empty-list? (rest l))
           #f
           (begin
             (display " ")
             (print-list-elements (rest l)))))))
(function (l) ...)
>> (print-list numbers)
```

Evaluator

Example: Lists

```
>> (fdef print-list (l)
     (display "(")
     (print-list-elements l)
     (display ")")
     (newline))
(function (l) ...)
>> (fdef print-list-elements (l)
     (if (empty-list? l)
       #f
       (begin
         (display (first l))
         (if (empty-list? (rest l))
           #f
           (begin
             (display " ")
             (print-list-elements (rest l)))))))
(function (l) ...)
>> (print-list numbers)
(1 2 3)
```

Evaluator

Data Structures

- So far, our language does not have any data structuring mechanisms.
- Most programming languages provide one or more such mechanisms:
  - C: structs, arrays
  - Pascal: records, arrays
  - Java: classes and arrays
  - Common Lisp: structures, classes, arrays, pairs
  - Scheme (pre R6RS): vectors, pairs
  - Scheme (pos R6RS): records, vectors, pairs
- We will implement just the fundamental data structuring mechanism: pairs.

Evaluator

Pairs

```
(define initial-bindings
  (list (cons 'pi 3.14159)
        ...
        (cons '> (make-primitive >))
        (cons '<= (make-primitive <=))
        (cons '>= (make-primitive >=))
        (cons 'display (make-primitive display))
        (cons 'newline (make-primitive newline))
        (cons 'read (make-primitive read))
```

Evaluator

Pairs

```
(define initial-bindings
  (list (cons 'pi 3.14159)
        ...
        (cons '> (make-primitive >))
        (cons '<= (make-primitive <=))
        (cons '>= (make-primitive >=))
        (cons 'display (make-primitive display))
        (cons 'newline (make-primitive newline))
        (cons 'read (make-primitive read))
        (cons 'list (make-primitive list))
        (cons 'cons (make-primitive cons))
```

Evaluator

Pairs

```
(define initial-bindings
  (list (cons 'pi 3.14159)
        ...
        (cons '> (make-primitive >))
        (cons '<= (make-primitive <=))
        (cons '>= (make-primitive >=))
        (cons 'display (make-primitive display))
        (cons 'newline (make-primitive newline))
        (cons 'read (make-primitive read))
        (cons 'list (make-primitive list))
        (cons 'cons (make-primitive cons))
        (cons 'car (make-primitive car))
        (cons 'cdr (make-primitive cdr))
```

Evaluator

Pairs

```
(define initial-bindings
  (list (cons 'pi 3.14159)
        ...
        (cons '> (make-primitive >))
        (cons '<= (make-primitive <=))
        (cons '>= (make-primitive >=))
        (cons 'display (make-primitive display))
        (cons 'newline (make-primitive newline))
        (cons 'read (make-primitive read))
        (cons 'list (make-primitive list))
        (cons 'cons (make-primitive cons))
        (cons 'car (make-primitive car))
        (cons 'cdr (make-primitive cdr))
        (cons 'pair? (make-primitive pair?))
        (cons 'null? (make-primitive null?))))
```

Evaluator

Pairs

```
>> (fdef caar (p) (car (car p)))
```

Evaluator

Pairs

```
>> (fdef caar (p) (car (car p)))
(function (p) ...)
```

Evaluator

Pairs

```
>> (fdef caar (p) (car (car p)))
(function (p) ...)
>> (fdef cadr (p) (car (cdr p)))
```

Evaluator

Pairs

```
>> (fdef caar (p) (car (car p)))
(function (p) ...)
>> (fdef cadr (p) (car (cdr p)))
(function (p) ...)
```

Evaluator

Pairs

```
>> (fdef caar (p) (car (car p)))
(function (p) ...)
>> (fdef cadr (p) (car (cdr p)))
(function (p) ...)
>> (fdef map (f l)
     (if (null? l)
       (list)
       (cons (f (car l))
             (map f (cdr l)))))
```

Evaluator

Pairs

```
>> (fdef caar (p) (car (car p)))
(function (p) ...)
>> (fdef cadr (p) (car (cdr p)))
(function (p) ...)
>> (fdef map (f l)
     (if (null? l)
       (list)
       (cons (f (car l))
             (map f (cdr l)))))
(function (f l) ...)
```

Evaluator

Pairs

```
>> (fdef caar (p) (car (car p)))
(function (p) ...)
>> (fdef cadr (p) (car (cdr p)))
(function (p) ...)
>> (fdef map (f l)
     (if (null? l)
       (list)
       (cons (f (car l))
             (map f (cdr l)))))
(function (f l) ...)
>> (map (lambda (x) (* x x x)) (list 1 2 3 4 5 6))
```

Evaluator

Pairs

```
>> (fdef caar (p) (car (car p)))
(function (p) ...)
>> (fdef cadr (p) (car (cdr p)))
(function (p) ...)
>> (fdef map (f l)
     (if (null? l)
       (list)
       (cons (f (car l))
             (map f (cdr l)))))
(function (f l) ...)
>> (map (lambda (x) (* x x x)) (list 1 2 3 4 5 6))
(1 8 27 64 125 216)
```

Evaluator

Pairs

```
>> (fdef caar (p) (car (car p)))
(function (p) ...)
>> (fdef cadr (p) (car (cdr p)))
(function (p) ...)
>> (fdef map (f l)
     (if (null? l)
       (list)
       (cons (f (car l))
             (map f (cdr l)))))
(function (f l) ...)
>> (map (lambda (x) (* x x x)) (list 1 2 3 4 5 6))
(1 8 27 64 125 216)
>> (fdef append (l0 l1)
     (if (null? l0)
       l1
       (cons (car l0)
             (append (cdr l0) l1))))
```

Evaluator

Pairs

```
>> (fdef caar (p) (car (car p)))
(function (p) ...)
>> (fdef cadr (p) (car (cdr p)))
(function (p) ...)
>> (fdef map (f l)
     (if (null? l)
       (list)
       (cons (f (car l))
             (map f (cdr l)))))
(function (f l) ...)
>> (map (lambda (x) (* x x x)) (list 1 2 3 4 5 6))
(1 8 27 64 125 216)
>> (fdef append (l0 l1)
     (if (null? l0)
       l1
       (cons (car l0)
             (append (cdr l0) l1))))
(function (l0 l1) ...)
```

Evaluator

Pairs

```
>> (fdef caar (p) (car (car p)))
(function (p) ...)
>> (fdef cadr (p) (car (cdr p)))
(function (p) ...)
>> (fdef map (f l)
     (if (null? l)
       (list)
       (cons (f (car l))
             (map f (cdr l)))))
(function (f l) ...)
>> (map (lambda (x) (* x x x)) (list 1 2 3 4 5 6))
(1 8 27 64 125 216)
>> (fdef append (l0 l1)
     (if (null? l0)
       l1
       (cons (car l0)
             (append (cdr l0) l1))))
(function (l0 l1) ...)
>> (append (list 1 2 3) (list 4 5 6))
```

Evaluator

Pairs

```
>> (fdef caar (p) (car (car p)))
(function (p) ...)
>> (fdef cadr (p) (car (cdr p)))
(function (p) ...)
>> (fdef map (f l)
     (if (null? l)
       (list)
       (cons (f (car l))
             (map f (cdr l)))))
(function (f l) ...)
>> (map (lambda (x) (* x x x)) (list 1 2 3 4 5 6))
(1 8 27 64 125 216)
>> (fdef append (l0 l1)
     (if (null? l0)
       l1
       (cons (car l0)
             (append (cdr l0) l1))))
(function (l0 l1) ...)
>> (append (list 1 2 3) (list 4 5 6))
(1 2 3 4 5 6)
```

Evaluator

Symbols

- Lisp was intended for *symbolic* processing.
- A *symbol* is an abstract entity that represents something, e.g., `moon`, `john`, `square`.
- Symbols can be used with lists to represent more complex concepts, e.g., `(believes John (loves Mary Peter))`
    - Two symbols are identical if they have the same name.
    - Symbols can be compared using eq?.

Symbols

```
(define initial-bindings
  (list (cons 'pi 3.14159)
        ...
        (cons 'null? (make-primitive null?))
```

Evaluator

Symbols

- Lisp was intended for *symbolic* processing.
- A *symbol* is an abstract entity that represents something, e.g., `moon`, `john`, `square`.
- Symbols can be used with lists to represent more complex concepts, e.g., `(believes John (loves Mary Peter))`
  - Two symbols are identical if they have the same name.
  - Symbols can be compared using eq?.

Symbols

```
(define initial-bindings
  (list (cons 'pi 3.14159)
        ...
        (cons 'null? (make-primitive null?))
        (cons 'eq? (make-primitive eq?))))
```

Evaluator

Symbols, Lists, and Quotation

- Symbols and Lists can be used as data structures...
- ...but they are also used to describe programs.
- How is it possible to distinguish between the two different purposes?
  - The evaluation rule for symbols retrieves the value associated with a symbol in the current environment.
  - We need a different evaluation rule for symbols that are to be taken literaly, i.e., that should not be evaluated.
  - Let's introduce the form `quote` for that purpose.
- `(quote foo)` evaluates to `foo`.
- `(quote foo)` can be abbreviated as `'foo`.

Evaluator

Evaluator

```
;;(quote foo)
```

Evaluator

Evaluator

```
;;(quote foo)

(define (quote? exp)
  (and (pair? exp)
       (eq? (car exp) 'quote)))
```

Evaluator

Evaluator

```
;;(quote foo)

(define (quote? exp)
  (and (pair? exp)
       (eq? (car exp) 'quote)))

(define (quoted-form exp)
  (cadr exp))
```

Evaluator

Evaluator

```
;;(quote foo)

(define (quote? exp)
  (and (pair? exp)
       (eq? (car exp) 'quote)))

(define (quoted-form exp)
  (cadr exp))

(define (eval-quote exp env)
  (quoted-form exp))
```

Evaluator

Evaluator

```
;;(quote foo)

(define (quote? exp)
  (and (pair? exp)
       (eq? (car exp) 'quote)))

(define (quoted-form exp)
  (cadr exp))

(define (eval-quote exp env)
  (quoted-form exp))

(define (eval exp env)
  (cond ((self-evaluating? exp) exp)


        ((name? exp)
         (eval-name exp env))
        ...
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Evaluator

Evaluator

```scheme
;;(quote foo)

(define (quote? exp)
  (and (pair? exp)
       (eq? (car exp) 'quote)))

(define (quoted-form exp)
  (cadr exp))

(define (eval-quote exp env)
  (quoted-form exp))

(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((quote? exp)

        ((name? exp)
         (eval-name exp env))
        ...
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Evaluator

Evaluator

```
;;(quote foo)

(define (quote? exp)
  (and (pair? exp)
       (eq? (car exp) 'quote)))

(define (quoted-form exp)
  (cadr exp))

(define (eval-quote exp env)
  (quoted-form exp))

(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((quote? exp)
         (eval-quote exp env))
        ((name? exp)
         (eval-name exp env))
        ...
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Evaluator

Examples

```
>> (quote 1)
```

Evaluator

Examples

```
>> (quote 1)
1
```

Evaluator

Examples

```
>> (quote 1)
1
>> (quote John)
```

Evaluator

Examples

```
>> (quote 1)
1
>> (quote John)
John
```

Evaluator

Examples

```
>> (quote 1)
1
>> (quote John)
John
>> (list (quote believes)
         (quote John)
         (list (quote loves) (quote Mary) (quote Peter)))
```

Evaluator

Examples

```
>> (quote 1)
1
>> (quote John)
John
>> (list (quote believes)
         (quote John)
         (list (quote loves) (quote Mary) (quote Peter)))
(believes John (loves Mary Peter))
```

Evaluator

Examples

```
>> (quote 1)
1
>> (quote John)
John
>> (list (quote believes)
         (quote John)
         (list (quote loves) (quote Mary) (quote Peter)))
(believes John (loves Mary Peter))
>> (list 'believes
         'John
         (list 'loves 'Mary 'Peter))
```

Evaluator

Examples

```
>> (quote 1)
1
>> (quote John)
John
>> (list (quote believes)
         (quote John)
         (list (quote loves) (quote Mary) (quote Peter)))
(believes John (loves Mary Peter))
>> (list 'believes
         'John
         (list 'loves 'Mary 'Peter))
(believes John (loves Mary Peter))
```

Evaluator

Examples

```
>> (quote 1)
1
>> (quote John)
John
>> (list (quote believes)
         (quote John)
         (list (quote loves) (quote Mary) (quote Peter)))
(believes John (loves Mary Peter))
>> (list 'believes
         'John
         (list 'loves 'Mary 'Peter))
(believes John (loves Mary Peter))
>> '(believes John (loves Mary Peter))
```

Evaluator

Examples

```
>> (quote 1)
1
>> (quote John)
John
>> (list (quote believes)
         (quote John)
         (list (quote loves) (quote Mary) (quote Peter)))
(believes John (loves Mary Peter))
>> (list 'believes
         'John
         (list 'loves 'Mary 'Peter))
(believes John (loves Mary Peter))
>> '(believes John (loves Mary Peter))
(believes John (loves Mary Peter))
```

Evaluator

Examples

```
>> (quote 1)
1
>> (quote John)
John
>> (list (quote believes)
         (quote John)
         (list (quote loves) (quote Mary) (quote Peter)))
(believes John (loves Mary Peter))
>> (list 'believes
         'John
         (list 'loves 'Mary 'Peter))
(believes John (loves Mary Peter))
>> '(believes John (loves Mary Peter))
(believes John (loves Mary Peter))
>> (list pi 'pi)
```

Evaluator

Examples

```
>> (quote 1)
1
>> (quote John)
John
>> (list (quote believes)
         (quote John)
         (list (quote loves) (quote Mary) (quote Peter)))
(believes John (loves Mary Peter))
>> (list 'believes
         'John
         (list 'loves 'Mary 'Peter))
(believes John (loves Mary Peter))
>> '(believes John (loves Mary Peter))
(believes John (loves Mary Peter))
>> (list pi 'pi)
(3.14159 pi)
```

Evaluator

Examples

```
>> (quote 1)
1
>> (quote John)
John
>> (list (quote believes)
         (quote John)
         (list (quote loves) (quote Mary) (quote Peter)))
(believes John (loves Mary Peter))
>> (list 'believes
         'John
         (list 'loves 'Mary 'Peter))
(believes John (loves Mary Peter))
>> '(believes John (loves Mary Peter))
(believes John (loves Mary Peter))
>> (list pi 'pi)
(3.14159 pi)
>> (list 'fdef 'square (list 'x) (list '* 'x 'x))
```

Evaluator

Examples

```
>> (quote 1)
1
>> (quote John)
John
>> (list (quote believes)
         (quote John)
         (list (quote loves) (quote Mary) (quote Peter)))
(believes John (loves Mary Peter))
>> (list 'believes
         'John
         (list 'loves 'Mary 'Peter))
(believes John (loves Mary Peter))
>> '(believes John (loves Mary Peter))
(believes John (loves Mary Peter))
>> (list pi 'pi)
(3.14159 pi)
>> (list 'fdef 'square (list 'x) (list '* 'x 'x))
(fdef square (x) (* x x))
```

Evaluator

Logical Operators: and

```
>> (fdef and (b0 b1)
     (if b0
       b1
       #f))
```

Evaluator

Logical Operators: and

```
>> (fdef and (b0 b1)
     (if b0
       b1
       #f))
(function (b0 b1) ...)
```

Evaluator

Logical Operators: and

```
>> (fdef and (b0 b1)
     (if b0
       b1
       #f))
(function (b0 b1) ...)
>> (fdef quotient-or-false (a b)
```

Evaluator

Logical Operators: and

```
>> (fdef and (b0 b1)
     (if b0
       b1
       #f))
(function (b0 b1) ...)
>> (fdef quotient-or-false (a b)
     (and (not (= b 0))
```

Evaluator

Logical Operators: and

```
>> (fdef and (b0 b1)
     (if b0
       b1
       #f))
(function (b0 b1) ...)
>> (fdef quotient-or-false (a b)
     (and (not (= b 0))
          (/ a b)))
```

Evaluator

Logical Operators: and

```
>> (fdef and (b0 b1)
     (if b0
       b1
       #f))
(function (b0 b1) ...)
>> (fdef quotient-or-false (a b)
     (and (not (= b 0))
          (/ a b)))
(function (a b) ...)
```

Evaluator

Logical Operators: and

```
>> (fdef and (b0 b1)
     (if b0
       b1
       #f))
(function (b0 b1) ...)
>> (fdef quotient-or-false (a b)
     (and (not (= b 0))
          (/ a b)))
(function (a b) ...)
>> (quotient-or-false 6 2)
```

Evaluator

Logical Operators: and

```
>> (fdef and (b0 b1)
     (if b0
       b1
       #f))
(function (b0 b1) ...)
>> (fdef quotient-or-false (a b)
     (and (not (= b 0))
          (/ a b)))
(function (a b) ...)
>> (quotient-or-false 6 2)
3
```

Evaluator

Logical Operators: and

```
>> (fdef and (b0 b1)
     (if b0
       b1
       #f))
(function (b0 b1) ...)
>> (fdef quotient-or-false (a b)
     (and (not (= b 0))
          (/ a b)))
(function (a b) ...)
>> (quotient-or-false 6 2)
3
>> (quotient-or-false 6 0)
```

Evaluator

Logical Operators: and

```
>> (fdef and (b0 b1)
     (if b0
       b1
       #f))
(function (b0 b1) ...)
>> (fdef quotient-or-false (a b)
     (and (not (= b 0))
          (/ a b)))
(function (a b) ...)
>> (quotient-or-false 6 2)
3
>> (quotient-or-false 6 0)
error in /: undefined for 0
```

Problem

- Function calls evaluate all their arguments...
- ...but and requires *short-circuit* evaluation.

Evaluator

Short-Circuit Evaluation

- We can implement special evaluation rules in the evaluator...
- ...but that's a job for the compiler writer.
- We want to allow the programmer to define code transformations:

From

```
(and (not (= b 0))
     (/ a b))
```

To

```
(if (not (= b 0))
  (/ a b)
  #f)
```

Evaluator

Short-Circuit Evaluation

- We can implement special evaluation rules in the evaluator...
- ...but that's a job for the compiler writer.
- We want to allow the programmer to define code transformations:

From

```
(and (not (= b 0))
     (/ a b))
```

To

```
(if (not (= b 0))
  (/ a b)
  #f)
```

Transformation

```
(mdef and (b0 b1)
  (list 'if b0 b1 '#f))
```

Evaluator

Macros

- A macro call is a form that:
  1. Receives syntactical forms as arguments.
  2. Computes a syntactical form as result (the *macro expansion*).
  3. The computed form is evaluated in place of the macro call.

Function Calls *vs* Macro Calls

- A function call evaluates the arguments and computes a result that is not evaluated.
- A macro call does not evaluate the arguments and computes a result that is evaluated.

Evaluator

Functions *vs* Macros

```
>> (fdef f-foo (x)
     (display x)
     (newline)
     x)
```

Evaluator

Functions *vs* Macros

```
>> (fdef f-foo (x)
     (display x)
     (newline)
     x)
(function (x) ...)
```

Evaluator

Functions *vs* Macros

```
>> (fdef f-foo (x)
     (display x)
     (newline)
     x)
(function (x) ...)
>> (mdef m-foo (x)
     (display x)
     (newline)
     x)
```

Evaluator

Functions *vs* Macros

```
>> (fdef f-foo (x)
     (display x)
     (newline)
     x)
(function (x) ...)
>> (mdef m-foo (x)
     (display x)
     (newline)
     x)
(function (x) ...)
```

Evaluator

Functions *vs* Macros

```
>> (fdef f-foo (x)
     (display x)
     (newline)
     x)
(function (x) ...)
>> (mdef m-foo (x)
     (display x)
     (newline)
     x)
(function (x) ...)
>> (f-foo (+ 1 2))
```

Evaluator

Functions *vs* Macros

```
>> (fdef f-foo (x)
     (display x)
     (newline)
     x)
(function (x) ...)
>> (mdef m-foo (x)
     (display x)
     (newline)
     x)
(function (x) ...)
>> (f-foo (+ 1 2))
3
3
```

Evaluator

Functions *vs* Macros

```
>> (fdef f-foo (x)
     (display x)
     (newline)
     x)
(function (x) ...)
>> (mdef m-foo (x)
     (display x)
     (newline)
     x)
(function (x) ...)
>> (f-foo (+ 1 2))
3
3
>> (m-foo (+ 1 2))
```

Evaluator

Functions *vs* Macros

```
>> (fdef f-foo (x)
     (display x)
     (newline)
     x)
(function (x) ...)
>> (mdef m-foo (x)
     (display x)
     (newline)
     x)
(function (x) ...)
>> (f-foo (+ 1 2))
3
3
>> (m-foo (+ 1 2))
(+ 1 2)
3
```

Evaluator

Functions *vs* Macros

```
>> (fdef f-bar ()
     (list '+
           '1
           '2))
```

Evaluator

Functions *vs* Macros

```
>> (fdef f-bar ()
     (list '+
           '1
           '2))
(function () ...)
```

Evaluator

Functions *vs* Macros

```
>> (fdef f-bar ()
     (list '+
           '1
           '2))
(function () ...)
>> (mdef m-bar ()
     (list '+
           '1
           '2))
```

Evaluator

Functions *vs* Macros

```
>> (fdef f-bar ()
     (list '+
           '1
           '2))
(function () ...)
>> (mdef m-bar ()
     (list '+
           '1
           '2))
(function () ...)
```

Evaluator

Functions *vs* Macros

```
>> (fdef f-bar ()
     (list '+
           '1
           '2))
(function () ...)
>> (mdef m-bar ()
     (list '+
           '1
           '2))
(function () ...)
>> (f-bar)
```

Evaluator

Functions *vs* Macros

```
>> (fdef f-bar ()
     (list '+
           '1
           '2))
(function () ...)
>> (mdef m-bar ()
     (list '+
           '1
           '2))
(function () ...)
>> (f-bar)
(+ 1 2)
```

Evaluator

Functions *vs* Macros

```
>> (fdef f-bar ()
     (list '+
           '1
           '2))
(function () ...)
>> (mdef m-bar ()
     (list '+
           '1
           '2))
(function () ...)
>> (f-bar)
(+ 1 2)
>> (m-bar)
```

Evaluator

Functions *vs* Macros

```
>> (fdef f-bar ()
     (list '+
           '1
           '2))
(function () ...)
>> (mdef m-bar ()
     (list '+
           '1
           '2))
(function () ...)
>> (f-bar)
(+ 1 2)
>> (m-bar)
3
```

Evaluator

Macro Implementation

- A macro *is* a function but is called differently.
- We will implement macros simply as *tagged* functions.
- In a call, we first check whether the function contains the tag.

Evaluator

```
;;(mdef quux (x) (list '+ x '1))
```

Evaluator

Macro Implementation

- A macro *is* a function but is called differently.
- We will implement macros simply as *tagged* functions.
- In a call, we first check whether the function contains the tag.

Evaluator

```
;;(mdef quux (x) (list '+ x '1))

(define (mdef? exp)
  (and (pair? exp)
       (eq? (car exp) 'mdef)))
```

Evaluator

Macro Implementation

- A macro *is* a function but is called differently.
- We will implement macros simply as *tagged* functions.
- In a call, we first check whether the function contains the tag.

Evaluator

```scheme
;;(mdef quux (x) (list '+ x '1))

(define (mdef? exp)
  (and (pair? exp)
       (eq? (car exp) 'mdef)))

(define (eval-mdef exp env)
  (eval `(def ,(cadr exp)
              (lambda ,(caddr exp) "macro" ,@(cdddr exp)))
        env))
```

Evaluator

Macro Implementation

- A macro *is* a function but is called differently.
- We will implement macros simply as *tagged* functions.
- In a call, we first check whether the function contains the tag.

Evaluator

```
;;(mdef quux (x) (list '+ x '1))

(define (mdef? exp)
  (and (pair? exp)
       (eq? (car exp) 'mdef)))

(define (eval-mdef exp env)
  (eval `(def ,(cadr exp)
              (lambda ,(caddr exp) "macro" ,@(cdddr exp)))
        env))
```

Evaluator

Macro Implementation

- A macro *is* a function but is called differently.
- We will implement macros simply as *tagged* functions.
- In a call, we first check whether the function contains the tag.

Evaluator

```
;;(mdef quux (x) (list '+ x '1))

(define (mdef? exp)
  (and (pair? exp)
       (eq? (car exp) 'mdef)))

(define (eval-mdef exp env)
  (eval `(def ,(cadr exp)
              (lambda ,(caddr exp) "macro" ,@(cdddr exp)))
        env))
```

Evaluator

Macro Implementation

- A macro *is* a function but is called differently.
- We will implement macros simply as *tagged* functions.
- In a call, we first check whether the function contains the tag.

Evaluator

```
;;(mdef quux (x) (list '+ x '1))

(define (mdef? exp)
  (and (pair? exp)
       (eq? (car exp) 'mdef)))

(define (eval-mdef exp env)
  (eval `(def ,(cadr exp)
              (lambda ,(caddr exp) "macro" ,@(cdddr exp)))
       env))
```

Evaluator

Macro Implementation

- A macro *is* a function but is called differently.
- We will implement macros simply as *tagged* functions.
- In a call, we first check whether the function contains the tag.

Evaluator

```
;;(mdef quux (x) (list '+ x '1))

(define (mdef? exp)
  (and (pair? exp)
       (eq? (car exp) 'mdef)))

(define (eval-mdef exp env)
  (eval `(def ,(cadr exp)
              (lambda ,(caddr exp) "macro" ,@(cdddr exp)))
        env))
```

Evaluator

Evaluator

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ...
        ((fdef? exp)
         (eval-fdef exp env))


        ((begin? exp)
         (eval-begin exp env))
        ((call? exp)
         (eval-call exp env))
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Evaluator

Evaluator

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
         ...
        ((fdef? exp)
         (eval-fdef exp env))
        ((mdef? exp)

        ((begin? exp)
         (eval-begin exp env))
        ((call? exp)
         (eval-call exp env))
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Evaluator

Evaluator

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
          ...
        ((fdef? exp)
         (eval-fdef exp env))
        ((mdef? exp)
         (eval-mdef exp env))
        ((begin? exp)
         (eval-begin exp env))
        ((call? exp)
         (eval-call exp env))
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Evaluator

Evaluator

```
(define (eval-call exp env)
  (let ((func (eval (call-operator exp) env))
        (args (eval-exprs (call-operands exp) env)))
    (if (primitive? func)
        (apply-primitive-function func args)
        (let ((extended-environment
               (augment-environment (function-parameters func)
                                    args
                                    (function-environment func))))
          (eval (function-body func) extended-environment)))))
```

Evaluator

Evaluator

```
(define (eval-call exp env)
  (let ((func (eval (call-operator exp) env)))
    (apply-function func (eval-exprs (call-operands exp) env))))

(define (apply-function func args)
  (if (primitive? func)
      (apply-primitive-function func args)
      (let ((extended-environment
             (augment-environment (function-parameters func)
                                  args
                                  (function-environment func))))
        (eval (function-body func) extended-environment))))
```

Evaluator

Evaluator

```
(define (eval-call exp env)
  (let ((func (eval (call-operator exp) env)))
    (if (macro? func)
        (let ((expansion (apply-function func (call-operands exp))))
          (eval expansion env))
        (apply-function func (eval-exprs (call-operands exp) env)))))
```

Evaluator

Evaluator

```
(define (eval-call exp env)
  (let ((func (eval (call-operator exp) env)))
    (if (macro? func)
        (let ((expansion (apply-function func (call-operands exp))))
          (eval expansion env))
        (apply-function func (eval-exprs (call-operands exp) env)))))

(define (macro? obj)
  (and (function? obj)
       (equal? (cadr (function-body obj)) "macro")))
```

Macro Calls

- Don't evaluate the operands.
- Evaluate the returned value (in the current environment).

Evaluator

Evaluator

```
(define (eval-call exp env)
  (let ((func (eval (call-operator exp) env)))
    (if (macro? func)
        (let ((expansion (apply-function func (call-operands exp))))
          (eval expansion env))
        (apply-function func (eval-exprs (call-operands exp) env)))))

(define (macro? obj)
  (and (function? obj)
       (equal? (cadr (function-body obj)) "macro")))
```

Macro Calls

- Don't evaluate the operands.
- Evaluate the returned value (in the current environment).

Evaluator

Evaluator

```
(define (eval-call exp env)
  (let ((func (eval (call-operator exp) env)))
    (if (macro? func)
        (let ((expansion (apply-function func (call-operands exp))))
          (eval expansion env))
        (apply-function func (eval-exprs (call-operands exp) env)))))

(define (macro? obj)
  (and (function? obj)
       (equal? (cadr (function-body obj)) "macro")))
```

Macro Calls

- Don't evaluate the operands.
- Evaluate the returned value (in the current environment).

Evaluator

Evaluator

```
(define (eval-call exp env)
  (let ((func (eval (call-operator exp) env)))
    (if (macro? func)
        (let ((expansion (apply-function func (call-operands exp))))
          (eval expansion env))
        (apply-function func (eval-exprs (call-operands exp) env)))))

(define (macro? obj)
  (and (function? obj)
       (equal? (cadr (function-body obj)) "macro")))
```

Macro Calls

- Don't evaluate the operands.
- Evaluate the returned value (in the current environment).

Evaluator

From

```
(and (not (= b 0))
     (/ a b))
```

To

Evaluator

From

```
(and (not (= b 0))
     (/ a b))
```

To

```
(if (not (= b 0))
  (/ a b)
  #f)
```

Evaluator

From

```
(and (not (= b 0))
     (/ a b))
```

To

```
(if (not (= b 0))
  (/ a b)
  #f)
```

Transformation

```
(mdef and (b0 b1)
  (list 'if
    b0
    b1
    '#f))
```

Evaluator

From

```
(and (not (= b 0))
     (/ a b))
```

To

```
(if (not (= b 0))
  (/ a b)
  #f)
```

Transformation

```
(mdef and (b0 b1)
  (list 'if
    b0
    b1
    '#f))
```

Evaluator

From

```
(and (not (= b 0))
     (/ a b))
```

To

```
(if (not (= b 0))
  (/ a b)
  #f)
```

Transformation

```
(mdef and (b0 b1)
  (list 'if
    b0
    b1
    '#f))
```

Evaluator

From

```
(for i 5 10
  (display i))
```

To

Evaluator

From

```
(for i 5 10
  (display i))
```

To

```
(let ()
  (fdef loop (i)
    (if (< i 10)
      (begin
        (display i)
        (loop (+ i 1)))
      #f))
  (loop 5))
```

Evaluator

From

```
(for i 5 10
  (display i))
```

To

```
(let ()
  (fdef loop (i)
    (if (< i 10)
      (begin
        (display i)
        (loop (+ i 1)))
      #f))
  (loop 5))
```

Transformation

```
(mdef for (var inf sup form)
  (list 'let (list)
        (list 'fdef 'loop (list var)
          (list 'if (list '< var sup)
                (list 'begin
                      form
                      (list 'loop (list '+ var 1)))
                '#f))
        (list 'loop inf)))
```

Evaluator

From

```
(for i 5 10
  (display i))
```

To

```
(let ()
  (fdef loop (i)
    (if (< i 10)
      (begin
        (display i)
        (loop (+ i 1)))
      #f))
  (loop 5))
```

Transformation

```
(mdef for (var inf sup form)
  (list 'let (list)
        (list 'fdef 'loop (list var)
          (list 'if (list '< var sup)
                (list 'begin
                      form
                      (list 'loop (list '+ var 1)))
                '#f))
        (list 'loop inf)))
```

Evaluator

From

```
(for i 5 10
  (display i))
```

To

```
(let ()
  (fdef loop (i)
    (if (< i 10)
      (begin
        (display i)
        (loop (+ i 1)))
      #f))
  (loop 5))
```

Transformation

```
(mdef for (var inf sup form)
  (list 'let (list)
        (list 'fdef 'loop (list var)
          (list 'if (list '< var sup)
                (list 'begin
                      form
                      (list 'loop (list '+ var 1)))
                '#f))
        (list 'loop inf)))
```

Evaluator

From

```
(for i 5 10
  (display i))
```

To

```
(let ()
  (fdef loop (i)
    (if (< i 10)
      (begin
        (display i)
        (loop (+ i 1)))
      #f))
  (loop 5))
```

Transformation

```
(mdef for (var inf sup form)
  (list 'let (list)
        (list 'fdef 'loop (list var)
          (list 'if (list '< var sup)
                (list 'begin
                      form
                      (list 'loop (list '+ var 1)))
                '#f))
        (list 'loop inf)))
```

Evaluator

Example: or

```
>> (mdef or (b0 b1)
     (list 'if
       b0
       b0
       b1))
```

Evaluator

Example: or

```
>> (mdef or (b0 b1)
     (list 'if
       b0
       b0
       b1))
(function (b0 b1) ...)
```

Evaluator

Example: or

```
>> (mdef or (b0 b1)
     (list 'if
       b0
       b0
       b1))
(function (b0 b1) ...)
>> (fdef in-interval (val inf sup)
    (or (and (< val inf)
             inf)
        (or (and (> val sup)
                 sup)
            val)))
```

Evaluator

Example: or

```
>> (mdef or (b0 b1)
     (list 'if
       b0
       b0
       b1))
(function (b0 b1) ...)
>> (fdef in-interval (val inf sup)
    (or (and (< val inf)
             inf)
        (or (and (> val sup)
                 sup)
            val)))
(function (val inf sup) ...)
```

Evaluator

Example: or

```
>> (mdef or (b0 b1)
     (list 'if
       b0
       b0
       b1))
(function (b0 b1) ...)
>> (fdef in-interval (val inf sup)
    (or (and (< val inf)
             inf)
        (or (and (> val sup)
                 sup)
            val)))
(function (val inf sup) ...)
>> (in-interval 1 3 7)
3
```

Evaluator

Example: or

```
>> (mdef or (b0 b1)
     (list 'if
       b0
       b0
       b1))
(function (b0 b1) ...)
>> (fdef in-interval (val inf sup)
    (or (and (< val inf)
             inf)
        (or (and (> val sup)
                 sup)
            val)))
(function (val inf sup) ...)
>> (in-interval 1 3 7)
3
>> (in-interval 9 3 7)
```

Evaluator

Example: or

```
>> (mdef or (b0 b1)
     (list 'if
       b0
       b0
       b1))
(function (b0 b1) ...)
>> (fdef in-interval (val inf sup)
    (or (and (< val inf)
             inf)
        (or (and (> val sup)
                 sup)
            val)))
(function (val inf sup) ...)
>> (in-interval 1 3 7)
3
>> (in-interval 9 3 7)
7
```

Evaluator

Example: or

```
>> (mdef or (b0 b1)
      (list 'if
        b0
        b0
        b1))
(function (b0 b1) ...)
>> (fdef in-interval (val inf sup)
     (or (and (< val inf)
              inf)
         (or (and (> val sup)
                  sup)
             val)))
(function (val inf sup) ...)
>> (in-interval 1 3 7)
3
>> (in-interval 9 3 7)
7
>> (in-interval 4 3 7)
```

Evaluator

Example: or

```
>> (mdef or (b0 b1)
     (list 'if
       b0
       b0
       b1))
(function (b0 b1) ...)
>> (fdef in-interval (val inf sup)
    (or (and (< val inf)
             inf)
        (or (and (> val sup)
                 sup)
            val)))
(function (val inf sup) ...)
>> (in-interval 1 3 7)
3
>> (in-interval 9 3 7)
7
>> (in-interval 4 3 7)
4
```

Evaluator

Example: or

```
>> (fdef quotient-or-zero (a b)
     (or (and (not (= b 0))
              (/ a b))
         0))
```

Evaluator

Example: or

```
>> (fdef quotient-or-zero (a b)
     (or (and (not (= b 0))
              (/ a b))
         0))
(function (a b) ...)
```

Evaluator

Example: or

```
>> (fdef quotient-or-zero (a b)
     (or (and (not (= b 0))
              (/ a b))
         0))
(function (a b) ...)
>> (quotient-or-zero 6 2)
```

Evaluator

Example: or

```
>> (fdef quotient-or-zero (a b)
     (or (and (not (= b 0))
              (/ a b))
         0))
(function (a b) ...)
>> (quotient-or-zero 6 2)
3
```

Evaluator

Example: or

```
>> (fdef quotient-or-zero (a b)
     (or (and (not (= b 0))
              (/ a b))
         0))
(function (a b) ...)
>> (quotient-or-zero 6 2)
3
>> (quotient-or-zero 6 0)
```

Evaluator

Example: or

```
>> (fdef quotient-or-zero (a b)
     (or (and (not (= b 0))
              (/ a b))
         0))
(function (a b) ...)
>> (quotient-or-zero 6 2)
3
>> (quotient-or-zero 6 0)
0
```

Evaluator

Example: or

```
>> (fdef quotient-or-zero (a b)
     (or (and (not (= b 0))
              (/ a b))
         0))
(function (a b) ...)
>> (quotient-or-zero 6 2)
3
>> (quotient-or-zero 6 0)
0
>> (fdef quotient-or-zero (a b)
     (or (begin
           (display "Division:")
           (and (not (= b 0))
                (/ a b)))
         0))
```

Evaluator

Example: or

```
>> (fdef quotient-or-zero (a b)
     (or (and (not (= b 0))
              (/ a b))
         0))
(function (a b) ...)
>> (quotient-or-zero 6 2)
3
>> (quotient-or-zero 6 0)
0
>> (fdef quotient-or-zero (a b)
     (or (begin
           (display "Division:")
           (and (not (= b 0))
                (/ a b)))
         0))
(function (a b) ...)
```

Evaluator

Example: or

```
>> (fdef quotient-or-zero (a b)
     (or (and (not (= b 0))
              (/ a b))
         0))
(function (a b) ...)
>> (quotient-or-zero 6 2)
3
>> (quotient-or-zero 6 0)
0
>> (fdef quotient-or-zero (a b)
     (or (begin
           (display "Division:")
           (and (not (= b 0))
                (/ a b)))
         0))
(function (a b) ...)
>> (quotient-or-zero 6 0)
```

Evaluator

Example: or

```
>> (fdef quotient-or-zero (a b)
     (or (and (not (= b 0))
              (/ a b))
         0))
(function (a b) ...)
>> (quotient-or-zero 6 2)
3
>> (quotient-or-zero 6 0)
0
>> (fdef quotient-or-zero (a b)
     (or (begin
           (display "Division:")
           (and (not (= b 0))
                (/ a b)))
         0))
(function (a b) ...)
>> (quotient-or-zero 6 0)
Division:0
```

Evaluator

Example: or

```
>> (fdef quotient-or-zero (a b)
     (or (and (not (= b 0))
              (/ a b))
         0))
(function (a b) ...)
>> (quotient-or-zero 6 2)
3
>> (quotient-or-zero 6 0)
0
>> (fdef quotient-or-zero (a b)
     (or (begin
           (display "Division:")
           (and (not (= b 0))
                (/ a b)))
         0))
(function (a b) ...)
>> (quotient-or-zero 6 0)
Division:0
>> (quotient-or-zero 6 2)
```

Evaluator

Example: or

```
>> (fdef quotient-or-zero (a b)
     (or (and (not (= b 0))
              (/ a b))
         0))
(function (a b) ...)
>> (quotient-or-zero 6 2)
3
>> (quotient-or-zero 6 0)
0
>> (fdef quotient-or-zero (a b)
     (or (begin
           (display "Division:")
           (and (not (= b 0))
                (/ a b)))
         0))
(function (a b) ...)
>> (quotient-or-zero 6 0)
Division:0
>> (quotient-or-zero 6 2)
Division:Division:3  ;;What?
```

Evaluator

From

```
(or (begin
      (display "Division:")
      (and (not (= b 0))
           (/ a b)))
    0)
```

Transformation

```
(mdef or (b0 b1)
  (list 'if
    b0
    b0
    b1))
```

Evaluator

From

```
(or (begin
      (display "Division:")
      (and (not (= b 0))
           (/ a b)))
    0)
```

To

```
(if (begin
      (display "Division:")
      (and (not (= b 0))
           (/ a b)))
  (begin
      (display "Division:")
      (and (not (= b 0))
           (/ a b)))
  0)
```

Transformation

```
(mdef or (b0 b1)
  (list 'if
    b0
    b0
    b1))
```

Evaluator

Macro Problems

- Macro calls do not evaluate their arguments.
- Macro expansions might have multiple occurrences of the same parameter.
- Thus, macro expansions might have multiple occurrences of a form that was passed as argument to the macro call.

Solution

- Never repeat a parameter in a macro expansion.
- If necessary, expand into a binding form that binds a local variable to that parameter and reuse the local variable in the expanded code.

Evaluator

From

```
(or (begin
      (display "Division:")
      (and (not (= b 0))
           (/ a b)))
    0)
```

Transformation

```
(mdef or (b0 b1)
  (list 'let (list (list 'val b0))
    (list 'if
      'val
      'val
      b1)))
```

Evaluator

From

```
(or (begin
      (display "Division:")
      (and (not (= b 0))
           (/ a b)))
    0)
```

To

```
(let ((val
        (begin
          (display "Division:")
          (and (not (= b 0))
               (/ a b)))))
  (if val
    val
    0))
```

Transformation

```
(mdef or (b0 b1)
  (list 'let (list (list 'val b0))
    (list 'if
      'val
      'val
      b1)))
```

Evaluator

Example: or

```
>> (fdef quotient-or-zero (a b)
     (or (begin
           (display "Division:")
           (and (not (= b 0)) (/ a b)))
         0))
```

Evaluator

Example: or

```
>> (fdef quotient-or-zero (a b)
     (or (begin
           (display "Division:")
           (and (not (= b 0)) (/ a b)))
         0))
(function (a b) ...)
```

Evaluator

Example: or

```
>> (fdef quotient-or-zero (a b)
     (or (begin
           (display "Division:")
           (and (not (= b 0)) (/ a b)))
         0))
(function (a b) ...)
>> (quotient-or-zero 6 0)
```

Evaluator

Example: or

```
>> (fdef quotient-or-zero (a b)
     (or (begin
           (display "Division:")
           (and (not (= b 0)) (/ a b)))
         0))
(function (a b) ...)
>> (quotient-or-zero 6 0)
Division:0
```

Evaluator

Example: or

```
>> (fdef quotient-or-zero (a b)
     (or (begin
           (display "Division:")
           (and (not (= b 0)) (/ a b)))
         0))
(function (a b) ...)
>> (quotient-or-zero 6 0)
Division:0
>> (quotient-or-zero 6 2)
```

Evaluator

Example: or

```
>> (fdef quotient-or-zero (a b)
     (or (begin
           (display "Division:")
           (and (not (= b 0)) (/ a b)))
         0))
(function (a b) ...)
>> (quotient-or-zero 6 0)
Division:0
>> (quotient-or-zero 6 2)
Division:3  ;;Now it's OK
```

Evaluator

Example: or

```
>> (fdef quotient-or-zero (a b)
     (or (begin
           (display "Division:")
           (and (not (= b 0)) (/ a b)))
         0))
(function (a b) ...)
>> (quotient-or-zero 6 0)
Division:0
>> (quotient-or-zero 6 2)
Division:3  ;;Now it's OK
>> (in-interval 1 3 7)
```

Evaluator

Example: or

```
>> (fdef quotient-or-zero (a b)
     (or (begin
           (display "Division:")
           (and (not (= b 0)) (/ a b)))
         0))
(function (a b) ...)
>> (quotient-or-zero 6 0)
Division:0
>> (quotient-or-zero 6 2)
Division:3  ;;Now it's OK
>> (in-interval 1 3 7)
3
```

Evaluator

Example: or

```
>> (fdef quotient-or-zero (a b)
     (or (begin
           (display "Division:")
           (and (not (= b 0)) (/ a b)))
         0))
(function (a b) ...)
>> (quotient-or-zero 6 0)
Division:0
>> (quotient-or-zero 6 2)
Division:3  ;;Now it's OK
>> (in-interval 1 3 7)
3
>> (in-interval 9 3 7)
```

Evaluator

Example: or

```
>> (fdef quotient-or-zero (a b)
     (or (begin
           (display "Division:")
           (and (not (= b 0)) (/ a b)))
         0))
(function (a b) ...)
>> (quotient-or-zero 6 0)
Division:0
>> (quotient-or-zero 6 2)
Division:3  ;;Now it's OK
>> (in-interval 1 3 7)
3
>> (in-interval 9 3 7)
error in >: expected real, but got #f, as argument 1
```

Evaluator

From

```
(fdef in-interval (val inf sup)
  (or (and (< val inf)
           inf)
      (or (and (> val sup)
               sup)
          val)))
```

To

```
(fdef in-interval (val inf sup)
  (let ((val (and (< val inf)
                  inf)))
    (if val
      val
      (let ((val (and (> val sup)
                      sup)))
        (if val
          val
          val)))))
```

Transformation

```
(mdef or (b0 b1)
  (list 'let (list (list 'val b0))
    (list 'if
      'val
      'val
      b1)))
```

Evaluator

Macro Problems

- Macro calls accept forms as arguments.
- Those forms might (will) have free variables.
- When the macro expands into binding forms, the created scope will shadow some of those free variables.
- This is an even more serious problem in a Lisp-1.

Solution

- Use name mangling techniques.
- Use `gensyms`.
- Use *hygienic macros*.

Evaluator

From

```
(fdef in-interval (val inf sup)
  (or (and (< val inf)
           inf)
      (or (and (> val sup)
               sup)
          val)))
```

To

```
(fdef in-interval (val inf sup)
  (let ((val (and (< val inf)
                  inf)))
    (if val
      val
      (let ((val (and (> val sup)
                      sup)))
        (if val
          val
          val)))))
```

Transformation (without name mangling)

```
(mdef or (b0 b1)
  (list 'let (list (list 'val b0))
    (list 'if
      'val
      'val
      b1)))
```

Evaluator

From

```
(fdef in-interval (val inf sup)
  (or (and (< val inf)
           inf)
      (or (and (> val sup)
               sup)
          val)))
```

To

```
(fdef in-interval (val inf sup)
  (let ((%v% (and (< val inf)
                  inf)))
    (if %v%
      %v%
      (let ((%v% (and (> val sup)
                      sup)))
        (if %v%
          %v%
          val))))))
```

Transformation (with name mangling)

```
(mdef or (b0 b1)
  (list 'let (list (list '%v% b0))
    (list 'if
      '%v%
      '%v%
      b1)))
```

Evaluator

Macro Problems

- Name mangling is the same as creating a different namespace for the bindings that occur in the macro-expansion.
- But it doesn't completely solve the problem because different macros might use the same (mangled) symbols.

Solution

- gensym returns a symbol that is guaranted to be unique.

gensym

```
(define initial-bindings
  (list ...
        (cons 'eq? (make-primitive eq?))
```

Evaluator

Macro Problems

- Name mangling is the same as creating a different namespace for the bindings that occur in the macro-expansion.
- But it doesn't completely solve the problem because different macros might use the same (mangled) symbols.

Solution

- `gensym` returns a symbol that is guaranted to be unique.

`gensym`

```
(define initial-bindings
  (list ...
        (cons 'eq? (make-primitive eq?))
        (cons 'gensym (make-primitive gensym))))
```

Evaluator

From

```
(fdef in-interval (val inf sup)
  (or (and (< val inf)
           inf)
      (or (and (> val sup)
               sup)
          val)))
```

To

```
(fdef in-interval (val inf sup)
  (let ((val (and (< val inf)
                  inf)))
    (if val
      val
      (let ((val (and (> val sup)
                      sup)))
        (if val
          val
          val)))))
```

Transformation (without gensym)

```
(mdef or (b0 b1)
  (list 'let (list (list 'val b0))
    (list 'if
      'val
      'val
      b1)))
```

Evaluator

From

```
(fdef in-interval (val inf sup)
  (or (and (< val inf)
           inf)
      (or (and (> val sup)
               sup)
          val)))
```

To

```
(fdef in-interval (val inf sup)
  (let ((g23 (and (< val inf)
                  inf)))
    (if g23
      g23
      (let ((g24 (and (> val sup)
                      sup)))
        (if g24
          g24
          val)))))
```

Transformation (with gensym)

```
(mdef or (b0 b1)
  (let ((val (gensym)))
    (list 'let (list (list val b0))
      (list 'if
        val
        val
        b1))))
```

Evaluator

Macro Problems

- Gensyms solve the problem of macro-expansions that include binding forms (e.g., `lets`).
- But they do nothing to solve the problem of macro-expansions that have free variables.
- This is the reverse of the problem of higher-order functions in dynamically scoped languages.

Solution

- In a lexically scoped language, functions capture the surrounding environment.

Evaluator

From

```
(fdef in-interval (val inf sup)
  (or (and (< val inf)
           inf)
      (or (and (> val sup)
               sup)
          val)))
```

To

```
(fdef in-interval (val inf sup)
  (or-function
    (and (< val inf)
         inf)
    (lambda ()
      (or-function
        (and (> val sup)
             sup)
        (lambda ()
          val)))))
```

Evaluator

From

```
(fdef in-interval (val inf sup)
  (or (and (< val inf)
           inf)
      (or (and (> val sup)
               sup)
          val)))
```

To

```
(fdef in-interval (val inf sup)
  (or-function
    (and (< val inf)
         inf)
    (lambda ()
      (or-function
        (and (> val sup)
             sup)
        (lambda ()
          val)))))
```

Transformation (with `lambda`)

```
(mdef or (b0 b1)
     (list 'or-function
       b0
       (list 'lambda (list) b1)))
```

Evaluator

Example

```
(fdef or-function (b0 f1)
  (if b0
    b0
    (f1)))
```

Hygiene

- The macro call expands into a function call that protects the "expanded" code from being evaluated in the scope of the macro call.
- The parameters are wrapped in `lambdas` that protect the macro arguments from being evaluated in any scope created by the macro expansion.
- Hygienic macros in Scheme operate differently but with the same net result.

Evaluator

Templates

- The macro body is an expression that, when evaluated, computes an expression that, when evaluated, computes the result of the macro call.
- In most cases, the macro expansion can be described by a *template* of code.
- The template, when evaluated, computes an expression that, when evaluated, computes the result of the macro call.
- To be flexible, a template must allow for some parts of it to be replaced with values computed at expansion time.

Evaluator

Backquote/Quasiquote

- Common Lisp provides the *backquote* form.
- Scheme provides the *quasiquote* form:
  - `(quasiquote` *foo*`)` means "don't evaluate *foo* except for `unquote` and `unquote-splicing` subforms."
  - `(unquote` *foo*`)` means "evaluate *foo*."
  - `(unquote-splicing` *foo*`)` means "evaluate *foo* and spread the resulting list."
- `(quasiquote foo)` can be abbreviated as `` `foo ``.
- `(unquote foo)` can be abbreviated as `,foo`.
- `(unquote-splicing foo)` can be abbreviated as `,@foo`.
- `` `(x ,x) `` = `(list 'x x)`
- `` `(x ,x ,@x) `` = `(append (list 'x) (list x) x)`

Evaluator

From

```
(for i 5 10
  (display i))
```

To

```
(let ()
  (fdef loop (i)
    (if (< i 10)
      (begin
        (display i)
        (loop (+ i 1)))
      #f))
  (loop 5))
```

Transformation (without quasiquote)

```
(mdef for (var inf sup form)
  (list 'let (list)
        (list 'fdef 'loop (list var)
          (list 'if (list '< var sup)
                (list 'begin
                      form
                      (list 'loop (list '+ var 1)))
                '#f))
        (list 'loop inf)))
```

Evaluator

From

```
(for i 5 10
  (display i))
```

To

```
(let ()
  (fdef loop (i)
    (if (< i 10)
      (begin
        (display i)
        (loop (+ i 1)))
      #f))
  (loop 5))
```

Transformation (with quasiquote)

```
(mdef for (var inf sup form)
  `(let ()
     (fdef loop (,var)
       (if (< ,var ,sup)
         (begin
           ,form
           (loop (+ ,var 1)))
         #f))
     (loop ,inf)))
```

Evaluator

Evaluator

```
;; (quasiquote foo)
```

Evaluator

Evaluator

```
;; (quasiquote foo)

(define (quasiquote? exp)
  (and (pair? exp)
       (eq? (car exp) 'quasiquote)))
```

Evaluator

Evaluator

```
;; (quasiquote foo)

(define (quasiquote? exp)
  (and (pair? exp)
       (eq? (car exp) 'quasiquote)))

(define (quasiquoted-form exp)
  (cadr exp))
```

Evaluator

Evaluator

```
;; (quasiquote foo)

(define (quasiquote? exp)
  (and (pair? exp)
       (eq? (car exp) 'quasiquote)))

(define (quasiquoted-form exp)
  (cadr exp))

(define (eval exp env)
  (cond ((self-evaluating? exp)
         exp)
        ((quote? exp)
         (eval-quote exp env))


        ...
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Evaluator

Evaluator

```
;; (quasiquote foo)

(define (quasiquote? exp)
  (and (pair? exp)
       (eq? (car exp) 'quasiquote)))

(define (quasiquoted-form exp)
  (cadr exp))

(define (eval exp env)
  (cond ((self-evaluating? exp)
         exp)
        ((quote? exp)
         (eval-quote exp env))
        ((quasiquote? exp)

        ...
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Evaluator

Evaluator

```
;; (quasiquote foo)

(define (quasiquote? exp)
  (and (pair? exp)
       (eq? (car exp) 'quasiquote)))

(define (quasiquoted-form exp)
  (cadr exp))

(define (eval exp env)
  (cond ((self-evaluating? exp)
         exp)
        ((quote? exp)
         (eval-quote exp env))
        ((quasiquote? exp)
         (eval-quasiquote exp env))
        ...
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Evaluator

Evaluator

```
(define (eval-quasiquote exp env)
  (eval (expand (quasiquoted-form exp)) env))
```

Evaluator

Evaluator

```
(define (eval-quasiquote exp env)
  (eval (expand (quasiquoted-form exp)) env))

(define (expand form)
  (cond ((not (pair? form))
         (list 'quote form))
        ((eq? (car form) 'unquote)
         (cadr form))
        ((eq? (car form) 'quasiquote)
         (expand (cadr form)))
        (else
         (expand-list form))))
```

Evaluator

Evaluator

```
(define (splicing? form)
  (and (pair? form)
       (or (and (pair? (car form))
                (eq? (caar form) 'unquote-splicing))
           (splicing? (cdr form)))))
```

Evaluator

Evaluator

```
(define (splicing? form)
  (and (pair? form)
       (or (and (pair? (car form))
                (eq? (caar form) 'unquote-splicing))
           (splicing? (cdr form)))))

(define (expand-list form)
  (if (splicing? form)
      (cons 'append
            (map (lambda (subform)
                   (if (and (pair? subform)
                            (eq? (car subform) 'unquote-splicing))
                       (cadr subform)
                       (list 'list (expand subform))))
                 form))
      (cons 'list (map expand form))))
```

Evaluator

Evaluator

```
(mdef and (b0 b1)
  (list 'if
        b0
        b1
        '#f))
```

```
(mdef or (b0 b1)
  (list 'or-function
        b0
        (list 'lambda (list) b1)))
```

Evaluator

Evaluator

```
(mdef and (b0 b1)
  (list 'if
        b0
        b1
        '#f))

(mdef and (b0 b1)
  `(if ,b0
     ,b1
     #f))

(mdef or (b0 b1)
  (list 'or-function
        b0
        (list 'lambda (list) b1)))
```

Evaluator

Evaluator

```
(mdef and (b0 b1)
  (list 'if
        b0
        b1
        '#f))

(mdef and (b0 b1)
  `(if ,b0
     ,b1
     #f))

(mdef or (b0 b1)
  (list 'or-function
        b0
        (list 'lambda (list) b1)))

(mdef or (b0 b1)
  `(or-function
     ,b0
     (lambda () ,b1)))
```

Evaluator

Evaluator

```
(define (eval exp env)
  (cond ((self-evaluating? exp)
         exp)
        ((quote? exp)
         (eval-quote exp env))
        ((quasiquote? exp)
         (eval-quasiquote exp env))
        ((name? exp)
         (eval-name exp env))
        ((lambda? exp)
         (eval-lambda exp env))
        ((if? exp)
         (eval-if exp env))
        ((let? exp) (eval-let exp env))
        ((flet? exp) (eval-flet exp env))
        ((def? exp)
         (eval-def exp env))
        ((set? exp)
         (eval-set exp env))
        ((fdef? exp) (eval-fdef exp env))
        ((mdef? exp) (eval-mdef exp env))
        ((begin? exp)
         (eval-begin exp env))
        ((call? exp)
         (eval-call exp env))
        (else
          (error "Unknown expression type -- EVAL" exp))))
```

Evaluator

Evaluator

```
(define (eval exp env)
  (cond ((self-evaluating? exp)
         exp)
        ((quote? exp)
         (eval-quote exp env))
        ((quasiquote? exp)
         (eval-quasiquote exp env))
        ((name? exp)
         (eval-name exp env))
        ((lambda? exp)
         (eval-lambda exp env))
        ((if? exp)
         (eval-if exp env))
        ((let? exp) (eval-let exp env))
        ((flet? exp) (eval-flet exp env))
        ((def? exp)
         (eval-def exp env))
        ((set? exp)
         (eval-set exp env))
        ((fdef? exp) (eval-fdef exp env))
        ((mdef? exp) (eval-mdef exp env))
        ((begin? exp)
         (eval-begin exp env))
        ((call? exp)
         (eval-call exp env))
        (else
          (error "Unknown expression type -- EVAL" exp))))
```

Evaluator

Simplifying the Evaluator

- We defined `and` and `or` as macros that expand into other forms.
- This means that the evaluator no longer needs to specially identify `and`s or `or`s and to provide specific semantics for their evaluation.
- The same can be said for other "special forms."

Evaluator

Macros

```
(mdef fdef (name parameters body)
  `(def ,name (lambda ,parameters ,body)))
```

Evaluator

Macros

```
(mdef fdef (name parameters body)
  `(def ,name (lambda ,parameters ,body)))

(mdef flet (binds body)
  `(let ,(map (lambda (form)
                `(,(car form)
                  (lambda ,@(cdr form))))
              binds)
     ,body))
```

Evaluator

Macros

```
(mdef fdef (name parameters body)
  `(def ,name (lambda ,parameters ,body)))

(mdef flet (binds body)
  `(let ,(map (lambda (form)
                `(,(car form)
                  (lambda ,@(cdr form))))
              binds)
     ,body))

(mdef let (binds body)
 `((lambda ,(map car binds)
     ,body)
   ,@(map cadr binds)))
```

Evaluator

Evaluator

```
(define (eval exp env)
  (cond ((self-evaluating? exp)
         exp)
        ((quote? exp)
         (eval-quote exp env))
        ((quasiquote? exp)
         (eval-quasiquote exp env))
        ((name? exp)
         (eval-name exp env))
        ((lambda? exp)
         (eval-lambda exp env))
        ((if? exp)
         (eval-if exp env))
        ((let? exp) (eval-let exp env))
        ((flet? exp) (eval-flet exp env))
        ((def? exp)
         (eval-def exp env))
        ((set? exp)
         (eval-set exp env))
        ((fdef? exp) (eval-fdef exp env))
        ((mdef? exp) (eval-mdef exp env))
        ((begin? exp)
         (eval-begin exp env))
        ((call? exp)
         (eval-call exp env))
        (else
          (error "Unknown expression type -- EVAL" exp))))
```

Evaluator

Evaluator

```
(define (eval exp env)
  (cond ((self-evaluating? exp)
         exp)
        ((quote? exp)
         (eval-quote exp env))
        ((quasiquote? exp)
         (eval-quasiquote exp env))
        ((name? exp)
         (eval-name exp env))
        ((lambda? exp)
         (eval-lambda exp env))
        ((if? exp)
         (eval-if exp env))
        ((let? exp) (eval-let exp env))
        ((flet? exp) (eval-flet exp env))
        ((def? exp)
         (eval-def exp env))
        ((set? exp)
         (eval-set exp env))
        ((fdef? exp) (eval-fdef exp env))
        ((mdef? exp) (eval-mdef exp env))
        ((begin? exp)
         (eval-begin exp env))
        ((call? exp)
         (eval-call exp env))
        (else
          (error "Unknown expression type -- EVAL" exp))))
```

Evaluator

Evaluator

```
(define (eval exp env)
  (cond ((self-evaluating? exp)
         exp)
        ((quote? exp)
         (eval-quote exp env))
        ((quasiquote? exp)
         (eval-quasiquote exp env))
        ((name? exp)
         (eval-name exp env))
        ((lambda? exp)
         (eval-lambda exp env))
        ((if? exp)
         (eval-if exp env))
        ((def? exp)
         (eval-def exp env))
        ((set? exp)
         (eval-set exp env))
        ((mdef? exp) (eval-mdef exp env))
        ((begin? exp)
         (eval-begin exp env))
        ((call? exp)
         (eval-call exp env))
        (else
          (error "Unknown expression type -- EVAL" exp))))
```

Evaluator

Bootstrapping Macros

```
(def mdef
  (lambda (name parameters body)
    "macro"
    `(def ,name
       (lambda ,parameters
         "macro"
         ,body))))
```

Macro-defining Macro

- A macro is just a tagged function.
- An `mdef` call is a macro call that expands into a macro definition.
- We no longer need to specially recognize macro definitions in the evaluator.

Evaluator

Evaluator

```
(define (eval exp env)
  (cond ((self-evaluating? exp)
         exp)
        ((quote? exp)
         (eval-quote exp env))
        ((quasiquote? exp)
         (eval-quasiquote exp env))
        ((name? exp)
         (eval-name exp env))
        ((lambda? exp)
         (eval-lambda exp env))
        ((if? exp)
         (eval-if exp env))
        ((def? exp)
         (eval-def exp env))
        ((set? exp)
         (eval-set exp env))
        ((mdef? exp) (eval-mdef exp env))
        ((begin? exp)
         (eval-begin exp env))
        ((call? exp)
         (eval-call exp env))
        (else
          (error "Unknown expression type -- EVAL" exp))))
```

Evaluator

Evaluator

```
(define (eval exp env)
  (cond ((self-evaluating? exp)
         exp)
        ((quote? exp)
         (eval-quote exp env))
        ((quasiquote? exp)
         (eval-quasiquote exp env))
        ((name? exp)
         (eval-name exp env))
        ((lambda? exp)
         (eval-lambda exp env))
        ((if? exp)
         (eval-if exp env))
        ((def? exp)
         (eval-def exp env))
        ((set? exp)
         (eval-set exp env))
        ((mdef? exp) (eval-mdef exp env))
        ((begin? exp)
         (eval-begin exp env))
        ((call? exp)
         (eval-call exp env))
        (else
          (error "Unknown expression type -- EVAL" exp))))
```

Evaluator

Evaluator

```
(define (eval exp env)
  (cond ((self-evaluating? exp)
         exp)
        ((quote? exp)
         (eval-quote exp env))
        ((quasiquote? exp)
         (eval-quasiquote exp env))
        ((name? exp)
         (eval-name exp env))
        ((lambda? exp)
         (eval-lambda exp env))
        ((if? exp)
         (eval-if exp env))
        ((def? exp)
         (eval-def exp env))
        ((set? exp)
         (eval-set exp env))
        ((begin? exp)
         (eval-begin exp env))
        ((call? exp)
         (eval-call exp env))
        (else
          (error "Unknown expression type -- EVAL" exp))))
```

Evaluator

Bootstrapping the Evaluator

```
(eval
  '(begin
     (def mdef (lambda (name parameters body) "macro"
                 `(def ,name (lambda ,parameters "macro" ,body))))
     (mdef fdef (name parameters body)
           `(def ,name (lambda ,parameters ,body)))
     (mdef and (b0 b1) `(if ,b0 ,b1 #f))
     (mdef or (b0 b1) `(or-function ,b0 (lambda () ,b1)))
     (fdef or-function (b0 f1) (if b0 b0 (f1)))
     (fdef not (b) (if b #f #t))
     (fdef map (f l)
           (if (null? l) (list) (cons (f (car l)) (map f (cdr l)))))
     (fdef append (l0 l1)
           (if (null? l0) l1 (cons (car l0) (append (cdr l0) l1))))
     (fdef caar (p)
           (car (car p)))
     (fdef cadr (p)
           (car (cdr p)))
     (mdef flet (binds body)
           `(let ,(map (lambda (form)
                         `(,(car form)
                           (lambda ,@(cdr form))))
                       binds)
              ,body))
     (mdef let (binds body)
           `((lambda ,(map car binds)
               ,body)
             ,@(map cadr binds))))
  initial-environment)
```

Evaluator

Bootstrapping the Evaluator

```
>> (mdef define (name form)
     (if (pair? name)
       `(def ,(car name)
          (lambda ,(cdr name)
            ,form))
       `(def ,name ,form)))
(function (name form) ...)
```

Evaluator

Bootstrapping the Evaluator

```
>> (mdef define (name form)
     (if (pair? name)
       `(def ,(car name)
          (lambda ,(cdr name)
            ,form))
       `(def ,name ,form)))
(function (name form) ...)
>> (define pi 3.14159)
```

Evaluator

Bootstrapping the Evaluator

```
>> (mdef define (name form)
     (if (pair? name)
       `(def ,(car name)
          (lambda ,(cdr name)
            ,form))
       `(def ,name ,form)))
(function (name form) ...)
>> (define pi 3.14159)
3.14159
```

Evaluator

Bootstrapping the Evaluator

```
>> (mdef define (name form)
     (if (pair? name)
       `(def ,(car name)
          (lambda ,(cdr name)
            ,form))
       `(def ,name ,form)))
(function (name form) ...)
>> (define pi 3.14159)
3.14159
>> (define (fact n)
     (if (= n 0)
       1
       (* n (fact (- n 1)))))
```

Evaluator

Bootstrapping the Evaluator

```
>> (mdef define (name form)
     (if (pair? name)
       `(def ,(car name)
          (lambda ,(cdr name)
            ,form))
       `(def ,name ,form)))
(function (name form) ...)
>> (define pi 3.14159)
3.14159
>> (define (fact n)
     (if (= n 0)
       1
       (* n (fact (- n 1)))))
(function (n) ...)
```

Evaluator

Bootstrapping the Evaluator

```
>> (mdef define (name form)
     (if (pair? name)
       `(def ,(car name)
          (lambda ,(cdr name)
            ,form))
       `(def ,name ,form)))
(function (name form) ...)
>> (define pi 3.14159)
3.14159
>> (define (fact n)
     (if (= n 0)
       1
       (* n (fact (- n 1)))))
(function (n) ...)
>> (fact 10)
```

Evaluator

Bootstrapping the Evaluator

```
>> (mdef define (name form)
     (if (pair? name)
       `(def ,(car name)
          (lambda ,(cdr name)
            ,form))
       `(def ,name ,form)))
(function (name form) ...)
>> (define pi 3.14159)
3.14159
>> (define (fact n)
     (if (= n 0)
       1
       (* n (fact (- n 1)))))
(function (n) ...)
>> (fact 10)
3628800
```

Evaluator

MetaCircular Evaluator

- We described an *operational* model of the language.
- The advantage of operational models is that they are executable, meaning that we can input a program and output its evaluation according to the rules of the language.
- This is useful to understand the semantics of a language but also to experiment with different semantics.
- Another important result: programs are data for the evaluator:
    - The evaluator introspects the program.
    - The evaluator manipulates the program.
    - The evaluator creates a program.
    - The evaluator interprets the program.

Evaluator

MetaCircular Evaluator

- But the programs do not have the same capability:
  - Although it is possible to write a program that computes, introspects or manipulates another program, we cannot interpret the generated program.
  - Macros only solve part of the problem: a macro call *statically* computes a new program fragment.
- To provide our programs with evaluation capabilities, we need to build an evaluator.
- Simplest solution: the language of our programs is the same as the language of the evaluator so we can simply input the evaluator to the evaluator and create another level of interpretation.
- Best solution: we can provide the evaluator as a primitive operation (but we also need to reify environments).

Evaluator

Evaluator

```
;;(current-environment)
```

Evaluator

Evaluator

```
;;(current-environment)

(define (current-environment? exp)
  (and (pair? exp)
       (eq? (car exp) 'current-environment)))
```

Evaluator

Evaluator

```
;;(current-environment)

(define (current-environment? exp)
  (and (pair? exp)
       (eq? (car exp) 'current-environment)))

(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ...
        ((current-environment? exp)
         env)
        ((call? exp)
         (eval-call exp env))
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Evaluator

Evaluator

```
;;(current-environment)

(define (current-environment? exp)
  (and (pair? exp)
       (eq? (car exp) 'current-environment)))

(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ...
        ((current-environment? exp)
         env)
        ((call? exp)
         (eval-call exp env))
        (else
         (error "Unknown expression type -- EVAL" exp))))

(define initial-bindings
  (list (cons '+ (make-primitive +))
        ...
```

Evaluator

Evaluator

```
;;(current-environment)

(define (current-environment? exp)
  (and (pair? exp)
       (eq? (car exp) 'current-environment)))

(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ...
        ((current-environment? exp)
         env)
        ((call? exp)
         (eval-call exp env))
        (else
         (error "Unknown expression type -- EVAL" exp))))

(define initial-bindings
  (list (cons '+ (make-primitive +))
        ...
        (cons 'eval (make-primitive eval))))
```

Evaluator

Example

```
>> (eval (list '+ '1 '2) (current-environment))
```

Evaluator

Example

```
>> (eval (list '+ '1 '2) (current-environment))
3
```

Evaluator

Example

```
>> (eval (list '+ '1 '2) (current-environment))
3
>> (define my-env
     (let ((x 1) (y 2))
       (current-environment)))
```

Evaluator

Example

```
>> (eval (list '+ '1 '2) (current-environment))
3
>> (define my-env
     (let ((x 1) (y 2))
       (current-environment)))
...
```

Evaluator

Example

```
>> (eval (list '+ '1 '2) (current-environment))
3
>> (define my-env
     (let ((x 1) (y 2))
       (current-environment)))
...
>> (+ (eval 'x my-env) (eval 'y my-env))
```

Evaluator

Example

```
>> (eval (list '+ '1 '2) (current-environment))
3
>> (define my-env
     (let ((x 1) (y 2))
       (current-environment)))
...
>> (+ (eval 'x my-env) (eval 'y my-env))
3
```

Evaluator

Example

```
>> (eval (list '+ '1 '2) (current-environment))
3
>> (define my-env
     (let ((x 1) (y 2))
       (current-environment)))
...
>> (+ (eval 'x my-env) (eval 'y my-env))
3
>> (define (kons kar kdr)
     (current-environment))
```

Evaluator

Example

```
>> (eval (list '+ '1 '2) (current-environment))
3
>> (define my-env
     (let ((x 1) (y 2))
       (current-environment)))
...
>> (+ (eval 'x my-env) (eval 'y my-env))
3
>> (define (kons kar kdr)
     (current-environment))
(function (kar kdr) ...)
```

Evaluator

Example

```
>> (eval (list '+ '1 '2) (current-environment))
3
>> (define my-env
     (let ((x 1) (y 2))
       (current-environment)))
...
>> (+ (eval 'x my-env) (eval 'y my-env))
3
>> (define (kons kar kdr)
     (current-environment))
(function (kar kdr) ...)
>> (define (kar kons)
     (eval 'kar kons))
```

Evaluator

Example

```
>> (eval (list '+ '1 '2) (current-environment))
3
>> (define my-env
     (let ((x 1) (y 2))
       (current-environment)))
...
>> (+ (eval 'x my-env) (eval 'y my-env))
3
>> (define (kons kar kdr)
     (current-environment))
(function (kar kdr) ...)
>> (define (kar kons)
     (eval 'kar kons))
(function (kons) ...)
```

Evaluator

Example

```
>> (eval (list '+ '1 '2) (current-environment))
3
>> (define my-env
     (let ((x 1) (y 2))
       (current-environment)))
...
>> (+ (eval 'x my-env) (eval 'y my-env))
3
>> (define (kons kar kdr)
     (current-environment))
(function (kar kdr) ...)
>> (define (kar kons)
     (eval 'kar kons))
(function (kons) ...)
>> (define (kdr kons)
     (eval 'kdr kons))
```

Evaluator

Example

```
>> (eval (list '+ '1 '2) (current-environment))
3
>> (define my-env
     (let ((x 1) (y 2))
       (current-environment)))
...
>> (+ (eval 'x my-env) (eval 'y my-env))
3
>> (define (kons kar kdr)
     (current-environment))
(function (kar kdr) ...)
>> (define (kar kons)
     (eval 'kar kons))
(function (kons) ...)
>> (define (kdr kons)
     (eval 'kdr kons))
(function (kons) ...)
```

Evaluator

Example

```
>> (eval (list '+ '1 '2) (current-environment))
3
>> (define my-env
     (let ((x 1) (y 2))
       (current-environment)))
...
>> (+ (eval 'x my-env) (eval 'y my-env))
3
>> (define (kons kar kdr)
     (current-environment))
(function (kar kdr) ...)
>> (define (kar kons)
     (eval 'kar kons))
(function (kons) ...)
>> (define (kdr kons)
     (eval 'kdr kons))
(function (kons) ...)
>> (kar (kons 1 2))
```

Evaluator

Example

```
>> (eval (list '+ '1 '2) (current-environment))
3
>> (define my-env
     (let ((x 1) (y 2))
       (current-environment)))
...
>> (+ (eval 'x my-env) (eval 'y my-env))
3
>> (define (kons kar kdr)
     (current-environment))
(function (kar kdr) ...)
>> (define (kar kons)
     (eval 'kar kons))
(function (kons) ...)
>> (define (kdr kons)
     (eval 'kdr kons))
(function (kons) ...)
>> (kar (kons 1 2))
1
```

Evaluator

Example

```
>> (eval (list '+ '1 '2) (current-environment))
3
>> (define my-env
     (let ((x 1) (y 2))
       (current-environment)))
...
>> (+ (eval 'x my-env) (eval 'y my-env))
3
>> (define (kons kar kdr)
     (current-environment))
(function (kar kdr) ...)
>> (define (kar kons)
     (eval 'kar kons))
(function (kons) ...)
>> (define (kdr kons)
     (eval 'kdr kons))
(function (kons) ...)
>> (kar (kons 1 2))
1
>> (kdr (kons 1 2))
```

Evaluator

Example

```
>> (eval (list '+ '1 '2) (current-environment))
3
>> (define my-env
     (let ((x 1) (y 2))
       (current-environment)))
...
>> (+ (eval 'x my-env) (eval 'y my-env))
3
>> (define (kons kar kdr)
     (current-environment))
(function (kar kdr) ...)
>> (define (kar kons)
     (eval 'kar kons))
(function (kons) ...)
>> (define (kdr kons)
     (eval 'kdr kons))
(function (kons) ...)
>> (kar (kons 1 2))
1
>> (kdr (kons 1 2))
2
```

Evaluator

Example

```
>> (mdef defclass (name slots)
     `(fdef ,name ,slots
        (current-environment)))
```

Evaluator

Example

```
>> (mdef defclass (name slots)
     `(fdef ,name ,slots
        (current-environment)))
(function (name slots) ...)
```

Evaluator

Example

```
>> (mdef defclass (name slots)
     `(fdef ,name ,slots
        (current-environment)))
(function (name slots) ...)
>> (fdef slot-value (instance slot)
     (eval slot instance))
```

Evaluator

Example

```
>> (mdef defclass (name slots)
     `(fdef ,name ,slots
        (current-environment)))
(function (name slots) ...)
>> (fdef slot-value (instance slot)
     (eval slot instance))
(function (instance slot) ...)
```

Evaluator

Example

```
>> (mdef defclass (name slots)
     `(fdef ,name ,slots
        (current-environment)))
(function (name slots) ...)
>> (fdef slot-value (instance slot)
     (eval slot instance))
(function (instance slot) ...)
>> (defclass person
     (name age sex))
```

Evaluator

Example

```
>> (mdef defclass (name slots)
     `(fdef ,name ,slots
        (current-environment)))
(function (name slots) ...)
>> (fdef slot-value (instance slot)
     (eval slot instance))
(function (instance slot) ...)
>> (defclass person
     (name age sex))
(function (name age sex) ...)
```

Evaluator

Example

```
>> (mdef defclass (name slots)
     `(fdef ,name ,slots
        (current-environment)))
(function (name slots) ...)
>> (fdef slot-value (instance slot)
     (eval slot instance))
(function (instance slot) ...)
>> (defclass person
     (name age sex))
(function (name age sex) ...)
>> (def john (person "John Adams" 42 "male"))
```

Evaluator

Example

```
>> (mdef defclass (name slots)
     `(fdef ,name ,slots
        (current-environment)))
(function (name slots) ...)
>> (fdef slot-value (instance slot)
     (eval slot instance))
(function (instance slot) ...)
>> (defclass person
     (name age sex))
(function (name age sex) ...)
>> (def john (person "John Adams" 42 "male"))
...
```

Evaluator

Example

```
>> (mdef defclass (name slots)
     `(fdef ,name ,slots
        (current-environment)))
(function (name slots) ...)
>> (fdef slot-value (instance slot)
     (eval slot instance))
(function (instance slot) ...)
>> (defclass person
     (name age sex))
(function (name age sex) ...)
>> (def john (person "John Adams" 42 "male"))
...
>> (def sally (person "Sally Field" 64 "female"))
```

Evaluator

Example

```
>> (mdef defclass (name slots)
     `(fdef ,name ,slots
        (current-environment)))
(function (name slots) ...)
>> (fdef slot-value (instance slot)
     (eval slot instance))
(function (instance slot) ...)
>> (defclass person
     (name age sex))
(function (name age sex) ...)
>> (def john (person "John Adams" 42 "male"))
...
>> (def sally (person "Sally Field" 64 "female"))
...
```

Evaluator

Example

```
>> (mdef defclass (name slots)
     `(fdef ,name ,slots
        (current-environment)))
(function (name slots) ...)
>> (fdef slot-value (instance slot)
     (eval slot instance))
(function (instance slot) ...)
>> (defclass person
     (name age sex))
(function (name age sex) ...)
>> (def john (person "John Adams" 42 "male"))
...
>> (def sally (person "Sally Field" 64 "female"))
...
>> (slot-value john 'age)
```

Evaluator

Example

```
>> (mdef defclass (name slots)
     `(fdef ,name ,slots
        (current-environment)))
(function (name slots) ...)
>> (fdef slot-value (instance slot)
     (eval slot instance))
(function (instance slot) ...)
>> (defclass person
     (name age sex))
(function (name age sex) ...)
>> (def john (person "John Adams" 42 "male"))
...
>> (def sally (person "Sally Field" 64 "female"))
...
>> (slot-value john 'age)
42
```

Evaluator

Example

```
>> (mdef defclass (name slots)
     `(fdef ,name ,slots
        (current-environment)))
(function (name slots) ...)
>> (fdef slot-value (instance slot)
     (eval slot instance))
(function (instance slot) ...)
>> (defclass person
     (name age sex))
(function (name age sex) ...)
>> (def john (person "John Adams" 42 "male"))
...
>> (def sally (person "Sally Field" 64 "female"))
...
>> (slot-value john 'age)
42
>> (slot-value sally 'name)
```

Evaluator

Example

```
>> (mdef defclass (name slots)
     `(fdef ,name ,slots
        (current-environment)))
(function (name slots) ...)
>> (fdef slot-value (instance slot)
     (eval slot instance))
(function (instance slot) ...)
>> (defclass person
     (name age sex))
(function (name age sex) ...)
>> (def john (person "John Adams" 42 "male"))
...
>> (def sally (person "Sally Field" 64 "female"))
...
>> (slot-value john 'age)
42
>> (slot-value sally 'name)
"Sally Field"
```

Evaluator

Example

```
>> (mdef send (obj message)
     `(eval ',message ,obj))
```

Evaluator

Example

```
>> (mdef send (obj message)
     `(eval ',message ,obj))
(function (obj message) ...)
```

Evaluator

Example

```
>> (mdef send (obj message)
     `(eval ',message ,obj))
(function (obj message) ...)
>> (send john (fdef fact (n)
                (if (= n 0)
                  1
                  (* n (fact (- n 1))))))
```

Evaluator

Example

```
>> (mdef send (obj message)
     `(eval ',message ,obj))
(function (obj message) ...)
>> (send john (fdef fact (n)
                (if (= n 0)
                  1
                  (* n (fact (- n 1))))))
(function (n) ...)
```

Evaluator

Example

```
>> (mdef send (obj message)
     `(eval ',message ,obj))
(function (obj message) ...)
>> (send john (fdef fact (n)
                (if (= n 0)
                  1
                  (* n (fact (- n 1))))))
(function (n) ...)
>> (send sally (fdef fact (what)
                 (begin
                   (display "It's a fact that ")
                   (display what)
                   (newline))))
```

Evaluator

Example

```
>> (mdef send (obj message)
     `(eval ',message ,obj))
(function (obj message) ...)
>> (send john (fdef fact (n)
                (if (= n 0)
                  1
                  (* n (fact (- n 1))))))
(function (n) ...)
>> (send sally (fdef fact (what)
                 (begin
                   (display "It's a fact that ")
                   (display what)
                   (newline))))
(function (what) ...)
```

Evaluator

Example

```
>> (mdef send (obj message)
     `(eval ',message ,obj))
(function (obj message) ...)
>> (send john (fdef fact (n)
                (if (= n 0)
                  1
                  (* n (fact (- n 1))))))
(function (n) ...)
>> (send sally (fdef fact (what)
                 (begin
                   (display "It's a fact that ")
                   (display what)
                   (newline))))
(function (what) ...)
>> (send john (fact 10))
```

Evaluator

Example

```
>> (mdef send (obj message)
     `(eval ',message ,obj))
(function (obj message) ...)
>> (send john (fdef fact (n)
                (if (= n 0)
                  1
                  (* n (fact (- n 1))))))
(function (n) ...)
>> (send sally (fdef fact (what)
                 (begin
                   (display "It's a fact that ")
                   (display what)
                   (newline))))
(function (what) ...)
>> (send john (fact 10))
3628800
```

Evaluator

Example

```
>> (mdef send (obj message)
     `(eval ',message ,obj))
(function (obj message) ...)
>> (send john (fdef fact (n)
                (if (= n 0)
                  1
                  (* n (fact (- n 1))))))
(function (n) ...)
>> (send sally (fdef fact (what)
                 (begin
                   (display "It's a fact that ")
                   (display what)
                   (newline))))
(function (what) ...)
>> (send john (fact 10))
3628800
>> (send sally (fact "Lisp is great!"))
```

Evaluator

Example

```
>> (mdef send (obj message)
     `(eval ',message ,obj))
(function (obj message) ...)
>> (send john (fdef fact (n)
                (if (= n 0)
                  1
                  (* n (fact (- n 1))))))
(function (n) ...)
>> (send sally (fdef fact (what)
                 (begin
                   (display "It's a fact that ")
                   (display what)
                   (newline))))
(function (what) ...)
>> (send john (fact 10))
3628800
>> (send sally (fact "Lisp is great!"))
It's a fact that Lisp is great!
```

Evaluator

Continuations

- There are operations whose semantics is difficult to express in the "usual" computational model:
  - `error`
  - `throw`
  - `yield`
  - `suspend`
- These operations seem to *violate* the semantics of function calls because either they don't return to the caller (`error`, `throw`) or they allow a function to resume its computation after returning (`yield`, `suspend`).
- We need a different computational model that allows these operations.

Evaluator

Problem

```
>> (define (mystery a b c d e)
     (+ a (* b (/ (- c d) e))))
...
```

Evaluator

Problem

```
>> (define (mystery a b c d e)
     (+ a (* b (/ (- c d) e))))
...
>> (mystery 5 4 3 2 1)
```

Evaluator

Problem

```
>> (define (mystery a b c d e)
     (+ a (* b (/ (- c d) e))))
...
>> (mystery 5 4 3 2 1)
9
```

Evaluator

Problem

```
>> (define (mystery a b c d e)
     (+ a (* b (/ (- c d) e))))
...
>> (mystery 5 4 3 2 1)
9
>> (mystery 4 3 2 1 0)
```

Evaluator

Problem

```
>> (define (mystery a b c d e)
     (+ a (* b (/ (- c d) e))))
...
>> (mystery 5 4 3 2 1)
9
>> (mystery 4 3 2 1 0)
ERROR ;;and the evaluator stops!!!!
```

Is it solvable?

Evaluator

Problem

```
>> (define (mystery a b c d e)
     (+ a (* b (/ (- c d) e))))
...
>> (mystery 5 4 3 2 1)
9
>> (mystery 4 3 2 1 0)
ERROR ;;and the evaluator stops!!!!
```

Is it solvable?

```
(define (mystery a b c d e)
  (+ a (* b (safe-/ (- c d) e))))
```

Evaluator

Problem

```
>> (define (mystery a b c d e)
     (+ a (* b (/ (- c d) e))))
...
>> (mystery 5 4 3 2 1)
9
>> (mystery 4 3 2 1 0)
ERROR ;;and the evaluator stops!!!!
```

Is it solvable?

```
(define (mystery a b c d e)
  (+ a (* b (safe-/ (- c d) e))))

(define (safe-/ x y)
  (if (= y 0)
      (error "Can't divide " (list x y))
      (/ x y)))
```

Evaluator

Problem

```
>> (define (mystery a b c d e)
     (+ a (* b (/ (- c d) e))))
...
>> (mystery 5 4 3 2 1)
9
>> (mystery 4 3 2 1 0)
ERROR ;;and the evaluator stops!!!!
```

Is it solvable?

```
(define (mystery a b c d e)
  (+ a (* b (safe-/ (- c d) e))))

(define (safe-/ x y)
  (if (= y 0)
      (error "Can't divide " (list x y))
      (/ x y)))

(define (error msg args)
```

Evaluator

Problem

```
>> (define (mystery a b c d e)
      (+ a (* b (/ (- c d) e))))
...
>> (mystery 5 4 3 2 1)
9
>> (mystery 4 3 2 1 0)
ERROR ;;and the evaluator stops!!!!
```

Is it solvable?

```
(define (mystery a b c d e)
  (+ a (* b (safe-/ (- c d) e))))

(define (safe-/ x y)
  (if (= y 0)
      (error "Can't divide " (list x y))
      (/ x y)))

(define (error msg args)
  ???)
```

Evaluator

Errors

```
char *ptr = malloc(2000000000UL);

if (ptr == NULL) {
  /* allocation failed, but what can I do? */
} else {
  ...
}
```

Signaling Errors

- Some languages (e.g., C) use unusual values as errors (a negative number when a positive number is expected, `NULL` when a pointer is expected, etc).
- Each caller in the call chain that caused the error must check the return of the called function.
- This is not practical.

Evaluator

Errors

```
char *ptr = malloc(2000000000UL);

if (ptr == NULL) {
  throw "Allocation Error"
} else {
  ...
}
```

Signaling Errors

- Some languages (e.g., C++) allow non-local transfers of control.
- The throw statement transfers control to the most recent `try-catch` statement that handles it.
- This is practical.
- Note: Java's Checked Exceptions are *not* practical.

Evaluator

Handling Errors

- It is generally difficult to handle exceptional situations in the callee.
- Each callee can be called from many different places.
- The callee rarely knows how to handle exceptional situations in a way that pleases all callers.
- Non-local transfers of control allow the callee to *signal* that an exceptional situation ocurred...
- ... and the control is transfered to the most recent handler established in the call chain.
- This is impossible to do with function calls...
- ... isn't it?

Evaluator

Continuations

- During a function call, the caller must wait for the callee to finish its computation and return a value.
- After the return of the callee, the caller proceeds with its computation.
- This computation is the *continuation* of the call.
- The *continuation* of the call is whatever remains to be done after the return from the call.
- In most high-level languages, the *continuation* is implicitly stored in a *stack*.
- In languages with first-class (and higher-order) functions, the *continuation* can be explicitly represented with functions.

Evaluator

Using let

```
(define (mystery a b c d e)
  (+ a (* b (/ (- c d) e))))
```

Evaluator

Using let

```
(define (mystery a b c d e)
  (+ a (* b (/ (- c d) e))))

(define (mystery a b c d e)
  (let ((r0 (- c d)))
    (+ a (* b (/ r0 e)))))
```

Evaluator

Using let

```
(define (mystery a b c d e)
  (+ a (* b (/ (- c d) e))))

(define (mystery a b c d e)
  (let ((r0 (- c d)))
    (+ a (* b (/ r0 e)))))
```

Evaluator

Using let

```
(define (mystery a b c d e)
  (+ a (* b (/ (- c d) e))))

(define (mystery a b c d e)
  (let ((r0 (- c d)))
    (+ a (* b (/ r0 e)))))

(define (mystery a b c d e)
  (let ((r0 (- c d)))
    (let ((r1 (/ r0 e)))
      (+ a (* b r1)))))
```

Evaluator

Using let

```
(define (mystery a b c d e)
  (+ a (* b (/ (- c d) e))))

(define (mystery a b c d e)
  (let ((r0 (- c d)))
    (+ a (* b (/ r0 e)))))

(define (mystery a b c d e)
  (let ((r0 (- c d)))
    (let ((r1 (/ r0 e)))
      (+ a (* b r1)))))
```

Evaluator

Using let

```
(define (mystery a b c d e)
  (+ a (* b (/ (- c d) e))))

(define (mystery a b c d e)
  (let ((r0 (- c d)))
    (+ a (* b (/ r0 e)))))

(define (mystery a b c d e)
  (let ((r0 (- c d)))
    (let ((r1 (/ r0 e)))
      (+ a (* b r1)))))

(define (mystery a b c d e)
  (let ((r0 (- c d)))
    (let ((r1 (/ r0 e)))
      (let ((r2 (* b r1)))
        (+ a r2)))))
```

Evaluator

Using let

```
(define (mystery a b c d e)
  (+ a (* b (/ (- c d) e))))

(define (mystery a b c d e)
  (let ((r0 (- c d)))
    (+ a (* b (/ r0 e)))))

(define (mystery a b c d e)
  (let ((r0 (- c d)))
    (let ((r1 (/ r0 e)))
      (+ a (* b r1)))))

(define (mystery a b c d e)
  (let ((r0 (- c d)))
    (let ((r1 (/ r0 e)))
      (let ((r2 (* b r1)))
        (+ a r2)))))
```

Evaluator

Using let

```
(define (mystery a b c d e)
  (+ a (* b (/ (- c d) e))))

(define (mystery a b c d e)
  (let ((r0 (- c d)))
    (+ a (* b (/ r0 e)))))

(define (mystery a b c d e)
  (let ((r0 (- c d)))
    (let ((r1 (/ r0 e)))
      (+ a (* b r1)))))

(define (mystery a b c d e)
  (let ((r0 (- c d)))
    (let ((r1 (/ r0 e)))
      (let ((r2 (* b r1)))
        (+ a r2)))))

(define (mystery a b c d e)
  (let ((r0 (- c d)))
    (let ((r1 (/ r0 e)))
      (let ((r2 (* b r1)))
        (let ((r3 (+ a r2)))
          r3)))))
```

Evaluator

Using the let-lambda equivalence

```
(define (mystery a b c d e)
  (+ a (* b (/ (- c d) e))))

(define (mystery a b c d e)
  ((λ (r0) (+ a (* b (/ r0 e))))
   (- c d)))

(define (mystery a b c d e)
  (let ((r0 (- c d)))
    (let ((r1 (/ r0 e)))
      (+ a (* b r1)))))

(define (mystery a b c d e)
  (let ((r0 (- c d)))
    (let ((r1 (/ r0 e)))
      (let ((r2 (* b r1)))
        (+ a r2)))))

(define (mystery a b c d e)
  (let ((r0 (- c d)))
    (let ((r1 (/ r0 e)))
      (let ((r2 (* b r1)))
        (let ((r3 (+ a r2)))
          r3)))))
```

Evaluator

Using the let-lambda equivalence

```
(define (mystery a b c d e)
  (+ a (* b (/ (- c d) e))))

(define (mystery a b c d e)
  ((λ (r0) (+ a (* b (/ r0 e))))
   (- c d)))

(define (mystery a b c d e)
  ((λ (r0) ((λ (r1) (+ a (* b r1)))
            (/ r0 e)))
   (- c d)))

(define (mystery a b c d e)
  (let ((r0 (- c d)))
    (let ((r1 (/ r0 e)))
      (let ((r2 (* b r1)))
        (+ a r2)))))

(define (mystery a b c d e)
  (let ((r0 (- c d)))
    (let ((r1 (/ r0 e)))
      (let ((r2 (* b r1)))
        (let ((r3 (+ a r2)))
          r3)))))
```

Evaluator

Using the let-lambda equivalence

```
(define (mystery a b c d e)
  (+ a (* b (/ (- c d) e))))

(define (mystery a b c d e)
  ((λ (r0) (+ a (* b (/ r0 e))))
   (- c d)))

(define (mystery a b c d e)
  ((λ (r0) ((λ (r1) (+ a (* b r1)))
            (/ r0 e)))
   (- c d)))

(define (mystery a b c d e)
  ((λ (r0) ((λ (r1) ((λ (r2) (+ a r2))
                     (* b r1)))
            (/ r0 e)))
   (- c d)))

(define (mystery a b c d e)
  (let ((r0 (- c d)))
    (let ((r1 (/ r0 e)))
      (let ((r2 (* b r1)))
        (let ((r3 (+ a r2)))
          r3)))))
```

Evaluator

Using the let-lambda equivalence

```
(define (mystery a b c d e)
  (+ a (* b (/ (- c d) e))))

(define (mystery a b c d e)
  ((λ (r0) (+ a (* b (/ r0 e))))
   (- c d)))

(define (mystery a b c d e)
  ((λ (r0) ((λ (r1) (+ a (* b r1)))
            (/ r0 e)))
   (- c d)))

(define (mystery a b c d e)
  ((λ (r0) ((λ (r1) ((λ (r2) (+ a r2))
                     (* b r1)))
            (/ r0 e)))
   (- c d)))

(define (mystery a b c d e)
  ((λ (r0) ((λ (r1) ((λ (r2) ((λ (r3) r3)
                              (+ a r2)))
                     (* b r1)))
            (/ r0 e)))
   (- c d)))
```

Evaluator

Simplification

- In order to simplify the program, we will define functions that call the continuation with the value of the numeric operation.

Functions Accepting Explicit Continuations

```
(define (+c x y cont)
  (cont (+ x y)))

(define (-c x y cont)
  (cont (- x y)))

(define (*c x y cont)
  (cont (* x y)))

(define (/c x y cont)
  (cont (/ x y)))
```

Evaluator

Explicit Continuation

```
(define (mystery a b c d e)
  ((λ (r0) ((λ (r1) ((λ (r2) ((λ (r3) r3)
                               (+ a r2)))
                     (* b r1)))
            (/ r0 e)))
   (- c d)))
```

Evaluator

Explicit Continuation

```
(define (mystery a b c d e)
  ((λ (r0) ((λ (r1) ((λ (r2) ((λ (r3) r3)
                              (+ a r2)))
                     (* b r1)))
            (/ r0 e)))
   (- c d)))

(define (mystery a b c d e)
  (-c c d
      (λ (r0) ((λ (r1) ((λ (r2) ((λ (r3) r3)
                                 (+ a r2)))
                        (* b r1)))
               (/ r0 e)))))
```

Evaluator

Explicit Continuation

```
(define (mystery a b c d e)
  ((λ (r0) ((λ (r1) ((λ (r2) ((λ (r3) r3)
                              (+ a r2)))
                     (* b r1)))
            (/ r0 e)))
   (- c d)))

(define (mystery a b c d e)
  (-c c d
      (λ (r0) ((λ (r1) ((λ (r2) ((λ (r3) r3)
                                  (+ a r2)))
                         (* b r1)))
                (/ r0 e))))))

(define (mystery a b c d e)
  (-c c d
      (λ (r0) (/c r0 e
                  (λ (r1) ((λ (r2) ((λ (r3) r3)
                                     (+ a r2)))
                            (* b r1)))))))
```

Evaluator

Explicit Continuation

```
(define (mystery a b c d e)
  (-c c d
      (λ (r0) (/c r0 e
                  (λ (r1) ((λ (r2) ((λ (r3) r3)
                                    (+ a r2)))
                           (* b r1)))))))
```

Evaluator

Explicit Continuation

```
(define (mystery a b c d e)
  (-c c d
      (λ (r0) (/c r0 e
                  (λ (r1) ((λ (r2) ((λ (r3) r3)
                                    (+ a r2)))
                           (* b r1)))))))

(define (mystery a b c d e)
  (-c c d
      (λ (r0) (/c r0 e
                  (λ (r1) (*c b r1
                              (λ (r2) ((λ (r3) r3)
                                       (+ a r2)))))))))
```

Evaluator

Explicit Continuation

```
(define (mystery a b c d e)
  (-c c d
      (λ (r0) (/c r0 e
                  (λ (r1) ((λ (r2) ((λ (r3) r3)
                                    (+ a r2)))
                           (* b r1)))))))

(define (mystery a b c d e)
  (-c c d
      (λ (r0) (/c r0 e
                  (λ (r1) (*c b r1
                               (λ (r2) ((λ (r3) r3)
                                        (+ a r2)))))))))

(define (mystery a b c d e)
  (-c c d
      (λ (r0) (/c r0 e
                  (λ (r1) (*c b r1
                              (λ (r2) (+c a r2
                                          (λ (r3) r3)))))))))
```

Evaluator

Function mystery

```
(define (mystery a b c d e)
  (+ a (* b (/ (- c d) e))))
```

Function mystery

```
(define (mystery a b c d e)
  (-c c d
      (λ (r0) (/c r0 e
                  (λ (r1) (*c b r1
                              (λ (r2) (+c a r2
                                          (λ (r3) r3)))))))))
```

Continuations

- Each function now accepts an explicit continuation.
- In order to “return”, the function calls the continuation.

Evaluator

Function mystery

```
(define (mystery a b c d e)
  (+ a (* b (/ (- c d) e))))
```

Function mystery

```
(define (mystery a b c d e)
  (-c c d
      (λ (r0) (/c r0 e
                  (λ (r1) (*c b r1
                              (λ (r2) (+c a r2
                                          (λ (r3) r3)))))))))
```

Continuations

- Each function now accepts an explicit continuation.
- In order to "return", the function calls the continuation.

Evaluator

Function mystery

```
(define (mystery a b c d e)
  (+ a (* b (/ (- c d) e))))
```

Function mystery

```
(define (mystery a b c d e)
  (-c c d
      (λ (r0) (/c r0 e
                  (λ (r1) (*c b r1
                              (λ (r2) (+c a r2
                                          (λ (r3) r3)))))))))
```

Continuations

- Each function now accepts an explicit continuation.
- In order to “return”, the function calls the continuation.

Evaluator

Function mystery

```
(define (mystery a b c d e)
  (+ a (* b (/ (- c d) e))))
```

Function mystery

```
(define (mystery a b c d e)
  (-c c d
      (λ (r0) (/c r0 e
                    (λ (r1) (*c b r1
                                  (λ (r2) (+c a r2
                                                (λ (r3) r3)))))))))
```

Continuations

- Each function now accepts an explicit continuation.
- In order to “return”, the function calls the continuation.

Evaluator

Function mystery

```
(define (mystery a b c d e)
  (+ a (* b (/ (- c d) e))))
```

Function mystery

```
(define (mystery a b c d e)
  (-c c d
      (λ (r0) (/c r0 e
                  (λ (r1) (*c b r1
                              (λ (r2) (+c a r2
                                          (λ (r3) r3)))))))))
```

Continuations

- Each function now accepts an explicit continuation.
- In order to “return”, the function calls the continuation.

Evaluator

Signalling Errors

- A “normal” function call always returns to its caller...
- ... because the continuation is implicit.
- When the continuation is explicit, it is possible to *not* call it, i.e., it is possible to *not* return to its caller.

Function /c

```
(define (/c x y cont)
  (cont (/ x y)))
```

Function /c that Signals an Error

```
(define (/c x y cont)
  (if (= y 0)
    (error "Can't divide " (list x y))
    (cont (/ x y))))
```

Evaluator

Example

```
(define (mystery a b c d e)
  (-c c d
      (λ (r0) (/c r0 e
                  (λ (r1) (*c b r1
                              (λ (r2) (+c a r2
                                          (λ (r3) r3)))))))))

(define (/c x y cont)
  (if (= y 0)
    (error "Can't divide " (list x y))
    (cont (/ x y))))

(define (error msg args)
  (display msg)
  (display args))
```

Evaluator

Example

```
(define (mystery a b c d e)
  (-c c d
      (λ (r0) (/c r0 e
                  (λ (r1) (*c b r1
                              (λ (r2) (+c a r2
                                          (λ (r3) r3)))))))))

(define (/c x y cont)
  (if (= y 0)
    (error "Can't divide " (list x y))
    (cont (/ x y))))

(define (error msg args)
  (display msg)
  (display args))

>> (mystery 5 4 3 2 1)
```

Evaluator

Example

```
(define (mystery a b c d e)
  (-c c d
      (λ (r0) (/c r0 e
                  (λ (r1) (*c b r1
                              (λ (r2) (+c a r2
                                          (λ (r3) r3)))))))))

(define (/c x y cont)
  (if (= y 0)
    (error "Can't divide " (list x y))
    (cont (/ x y))))

(define (error msg args)
  (display msg)
  (display args))

>> (mystery 5 4 3 2 1)
9
```

Evaluator

Example

```
(define (mystery a b c d e)
  (-c c d
      (λ (r0) (/c r0 e
                  (λ (r1) (*c b r1
                              (λ (r2) (+c a r2
                                          (λ (r3) r3)))))))))

(define (/c x y cont)
  (if (= y 0)
    (error "Can't divide " (list x y))
    (cont (/ x y))))

(define (error msg args)
  (display msg)
  (display args))

>> (mystery 5 4 3 2 1)
9
>> (mystery 4 3 2 1 0)
```

Evaluator

Example

```
(define (mystery a b c d e)
  (-c c d
      (λ (r0) (/c r0 e
                  (λ (r1) (*c b r1
                              (λ (r2) (+c a r2
                                          (λ (r3) r3)))))))))

(define (/c x y cont)
  (if (= y 0)
    (error "Can't divide " (list x y))
    (cont (/ x y))))

(define (error msg args)
  (display msg)
  (display args))

>> (mystery 5 4 3 2 1)
9
>> (mystery 4 3 2 1 0)
Can't divide (1 0)
```

Evaluator

Are we done?

```
>> (+ 1 (mystery 5 4 3 2 1))
```

Evaluator

Are we done?

```
>> (+ 1 (mystery 5 4 3 2 1))
10
```

Evaluator

Are we done?

```
>> (+ 1 (mystery 5 4 3 2 1))
10
>> (+ 1 (mystery 4 3 2 1 0))
```

Evaluator

Are we done?

```
>> (+ 1 (mystery 5 4 3 2 1))
10
>> (+ 1 (mystery 4 3 2 1 0))
ERROR ;;and the evaluator stops!!!!
```

Errors

- We converted the primitive operations to use explicit continuations.
- But we didn't convert the `mystery` function to use an explicit continuation.
- When the function `/c` calls `error`, the returned value of the `error` function is also returned from `mystery`.
- The expression `(+ 1 (mystery ...))` then tries to add `1` to that returned value.

Evaluator

Example

```
(define (mystery a b c d e)
  (-c c d
      (λ (r0) (/c r0 e
                  (λ (r1) (*c b r1
                              (λ (r2) (+c a r2
                                          (λ (r3) r3)))))))))
```

Evaluator

Example

```
(define (mystery-c a b c d e cont)
  (-c c d
      (λ (r0) (/c r0 e
                  (λ (r1) (*c b r1
                              (λ (r2) (+c a r2
                                          (λ (r3) (cont r3))))))))))
```

Including a continuation

- The function must accept a continuation.
- The function "transfers" control to (i.e., calls) the continuation with the final result.

Evaluator

Example

```
(define (mystery-c a b c d e cont)
  (-c c d
      (λ (r0) (/c r0 e
                  (λ (r1) (*c b r1
                              (λ (r2) (+c a r2
                                          (λ (r3) (cont r3))))))))))
```

Including a continuation

- The function must accept a continuation.
- The function “transfers” control to (i.e., calls) the continuation with the final result.

Evaluator

Example

```
(define (mystery-c a b c d e cont)
  (-c c d
      (λ (r0) (/c r0 e
                  (λ (r1) (*c b r1
                              (λ (r2) (+c a r2
                                          cont))))))))
```

Including a continuation

- `(lambda (x) (f x))` = `f`
- This is η-reduction.

Evaluator

Example

```
>> (+ 1 (mystery 5 4 3 2 1))
```

Example

```
>> (mystery-c 5 4 3 2 1 (lambda (r) (+ 1 r)))
```

Evaluator

Example

```
>> (+ 1 (mystery 5 4 3 2 1))
10
```

Example

```
>> (mystery-c 5 4 3 2 1 (lambda (r) (+ 1 r)))
10
```

Evaluator

Example

```
>> (+ 1 (mystery 5 4 3 2 1))
10
>> (+ 1 (mystery 4 3 2 1 0))
```

Example

```
>> (mystery-c 5 4 3 2 1 (lambda (r) (+ 1 r)))
10
>> (mystery-c 4 3 2 1 0 (lambda (r) (+ 1 r)))
```

Evaluator

Example

```
>> (+ 1 (mystery 5 4 3 2 1))
10
>> (+ 1 (mystery 4 3 2 1 0))
ERROR ;;and the evaluator stops!!!!
```

Example

```
>> (mystery-c 5 4 3 2 1 (lambda (r) (+ 1 r)))
10
>> (mystery-c 4 3 2 1 0 (lambda (r) (+ 1 r)))
Can't divide (1 0)
```

Evaluator

Example

```
>> (+ 1 (mystery 5 4 3 2 1))
10
>> (+ 1 (mystery 4 3 2 1 0))
ERROR ;;and the evaluator stops!!!!
```

Example

```
>> (mystery-c 5 4 3 2 1 (lambda (r) (+ 1 r)))
10
>> (mystery-c 4 3 2 1 0 (lambda (r) (+ 1 r)))
Can't divide (1 0)
```

Continuations

- With implicit continuations, it is impossible for the program to influence the control flow from callers to callees.
- With explicit continuations, on each function call, the program decides what to do next.

Evaluator

Continuations

- Implicit continuation ⇒ program in *direct style*.
- Explicitly continuation ⇒ program in *continuation-passing style*.

`foo` and `bar` in Direct Style

```
(foo 2 (bar 3 4))
```

Evaluator

Continuations

- Implicit continuation ⇒ program in *direct style*.
- Explicitly continuation ⇒ program in *continuation-passing style*.

`foo` and `bar` in Direct Style

```
(foo 2 (bar 3 4))
```

`bar` in Continuation Passing Style

```
(bar-c 3 4
       (lambda (r0) (foo 2 r0)))
```

Evaluator

Continuations

- Implicit continuation $\Rightarrow$ program in *direct style*.
- Explicitly continuation $\Rightarrow$ program in *continuation-passing style*.

foo and bar in Direct Style

```
(foo 2 (bar 3 4))
```

bar in Continuation Passing Style

```
(bar-c 3 4
       (lambda (r0) (foo 2 r0)))
```

foo and bar in Continuation Passing Style

```
(bar-c 3 4
       (lambda (r0) (foo-c 2 r0
                           (lambda (r1) ...))))
```

Evaluator

Continuations

- Implicit continuation ⇒ program in *direct style*.
- Explicitly continuation ⇒ program in *continuation-passing style*.

Direct Style

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

Continuation Passing Style

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

Evaluator

Direct Style

```
>> (fact 10)
```

Continuation Passing Style

```
>> (fact 10 (lambda (r) r))
```

Evaluator

Direct Style

```
>> (fact 10)
3628800
```

Continuation Passing Style

```
>> (fact 10 (lambda (r) r))
3628800
```

Evaluator

Direct Style

```
>> (fact 10)
3628800
>> (+ (fact 10)
      (fact 5))
```

Continuation Passing Style

```
>> (fact 10 (lambda (r) r))
3628800
>> (fact 10 (lambda (r0)
              (fact 5 (lambda (r1) (+ r0 r1)))))
```

Evaluator

Direct Style

```
>> (fact 10)
3628800
>> (+ (fact 10)
      (fact 5))
3628920
```

Continuation Passing Style

```
>> (fact 10 (lambda (r) r))
3628800
>> (fact 10 (lambda (r0)
              (fact 5 (lambda (r1) (+ r0 r1)))))
3628920
```

Evaluator

Direct Style

```
>> (fact 10)
3628800
>> (+ (fact 10)
      (fact 5))
3628920
>> (let ((result (fact 20)))
```

Continuation Passing Style

```
>> (fact 10 (lambda (r) r))
3628800
>> (fact 10 (lambda (r0)
              (fact 5 (lambda (r1) (+ r0 r1)))))
3628920
>> (fact 20 (lambda (result)
```

Evaluator

Direct Style

```
>> (fact 10)
3628800
>> (+ (fact 10)
      (fact 5))
3628920
>> (let ((result (fact 20)))
     (* result result))
```

Continuation Passing Style

```
>> (fact 10 (lambda (r) r))
3628800
>> (fact 10 (lambda (r0)
              (fact 5 (lambda (r1) (+ r0 r1)))))
3628920
>> (fact 20 (lambda (result)
              (* result result)))
```

Evaluator

Direct Style

```
>> (fact 10)
3628800
>> (+ (fact 10)
      (fact 5))
3628920
>> (let ((result (fact 20)))
     (* result result))
5919012181389927685417441689600000000
```

Continuation Passing Style

```
>> (fact 10 (lambda (r) r))
3628800
>> (fact 10 (lambda (r0)
              (fact 5 (lambda (r1) (+ r0 r1)))))
3628920
>> (fact 20 (lambda (result)
              (* result result)))
5919012181389927685417441689600000000
```

Evaluator

In continuation-passing style:

- The continuation is represented as a function.
- The explicit representation of a continuation is the *reification* of the continuation.
- There are no *hidden* pending computations.
- Each computed result is explicitely named.
- All calls become *tail calls*.
- All recursive calls become *tail recursive calls*.
- A *stack* is not required.

Evaluator

Continuations

- The reification of a continuation allows its *capture*.

Capturing a continuation

```
>> (define pending-fact-computations #f)
#f
```

Evaluator

Continuations

- The reification of a continuation allows its *capture*.

Capturing a continuation

```
>> (define pending-fact-computations #f)
#f
>> (define (fact n c)
     (if (= n 0)
       (begin
         (set! pending-fact-computations c)
         (c 1))
       (fact (- n 1) (lambda (r) (c (* n r))))))
...
```

Evaluator

Continuations

- The reification of a continuation allows its *capture*.

Capturing a continuation

```
>> (define pending-fact-computations #f)
#f
>> (define (fact n c)
     (if (= n 0)
       (begin
         (set! pending-fact-computations c)
         (c 1))
       (fact (- n 1) (lambda (r) (c (* n r))))))
...
>> (fact 10 (lambda (r) r))
```

Evaluator

Continuations

- The reification of a continuation allows its *capture*.

Capturing a continuation

```
>> (define pending-fact-computations #f)
#f
>> (define (fact n c)
     (if (= n 0)
       (begin
         (set! pending-fact-computations c)
         (c 1))
       (fact (- n 1) (lambda (r) (c (* n r))))))
...
>> (fact 10 (lambda (r) r))
3628800
```

Evaluator

Continuations

- The reification of a continuation allows its *capture*.

Capturing a continuation

```
>> (define pending-fact-computations #f)
#f
>> (define (fact n c)
     (if (= n 0)
       (begin
         (set! pending-fact-computations c)
         (c 1))
       (fact (- n 1) (lambda (r) (c (* n r))))))
...
>> (fact 10 (lambda (r) r))
3628800
>> (pending-fact-computations 1)
```

Evaluator

Continuations

- The reification of a continuation allows its *capture*.

Capturing a continuation

```
>> (define pending-fact-computations #f)
#f
>> (define (fact n c)
     (if (= n 0)
       (begin
         (set! pending-fact-computations c)
         (c 1))
       (fact (- n 1) (lambda (r) (c (* n r))))))
...
>> (fact 10 (lambda (r) r))
3628800
>> (pending-fact-computations 1)
3628800
```

Evaluator

Continuations

- The reification of a continuation allows its *capture*.

Capturing a continuation

```
>> (define pending-fact-computations #f)
#f
>> (define (fact n c)
     (if (= n 0)
       (begin
         (set! pending-fact-computations c)
         (c 1))
       (fact (- n 1) (lambda (r) (c (* n r))))))
...
>> (fact 10 (lambda (r) r))
3628800
>> (pending-fact-computations 1)
3628800
>> (pending-fact-computations 10)
```

Evaluator

Continuations

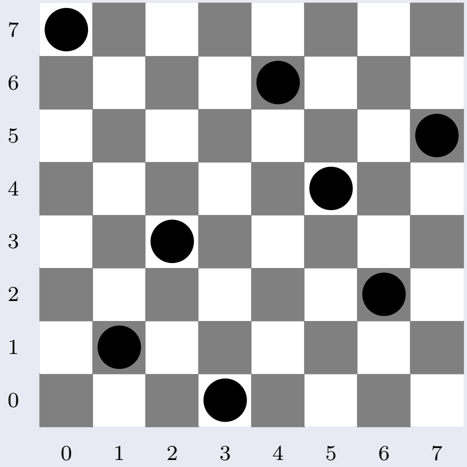
- The reification of a continuation allows its *capture*.

Capturing a continuation

```
>> (define pending-fact-computations #f)
#f
>> (define (fact n c)
     (if (= n 0)
       (begin
         (set! pending-fact-computations c)
         (c 1))
       (fact (- n 1) (lambda (r) (c (* n r))))))
...
>> (fact 10 (lambda (r) r))
3628800
>> (pending-fact-computations 1)
3628800
>> (pending-fact-computations 10)
36288000
```

Evaluator

Example - The *n*-Queens Problem



Evaluator

Example - The *n*-Queens Problem

- Place *n* Queens in a $n \times n$ chessboard so that no two queens attack each other.
- A solution requires that no two queens share the same row, column, or diagonal.

Is position (i_0, j_0) threatened by (i_1, j_1)?

```
(define (threat? i0 j0 i1 j1)
  (or (= i0 i1)
      (= j0 j1)
      (= (+ i0 j0) (+ i1 j1))
      (= (- i0 j0) (- i1 j1))))
```

Evaluator

Example - The *n*-Queens Problem

- We will represent the current board by the list of occupied positions.
- We will represent each position by the pair (*row* . *column*).

Is position (i, j) attacked by the currently placed queens?

```
(define (attacked? i j board)
  (and (not (null? board))
       (or (threat? i j (caar board) (cdar board))
           (attacked? i j (cdr board)))))
```

Evaluator

Solving the puzzle: Using *backtracking*

```
(define (queens n i j board)
  (if (= i n)
      board
      (if (= j n)
          #f
          (if (attacked? i j board)
              (queens n i (+ j 1) board)
              (or (queens n (+ i 1) 0 (cons (cons i j) board))
                  (queens n i (+ j 1) board)))))))
```

Evaluator

Solving the puzzle: Using *backtracking*

```
(define (queens n i j board)
  (if (= i n)
      board
      (if (= j n)
          #f
          (if (attacked? i j board)
              (queens n i (+ j 1) board)
              (or (queens n (+ i 1) 0 (cons (cons i j) board))
                  (queens n i (+ j 1) board))))))

(define (solve-queens n)
  (queens n 0 0 (list)))
```

Evaluator

Solution

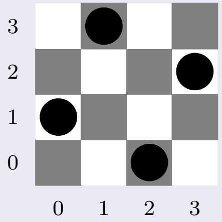
```
>> (solve-queens 4)
```

Evaluator

Solution

```
>> (solve-queens 4)
((3 . 2) (2 . 0) (1 . 3) (0 . 1))
```

Board



Evaluator

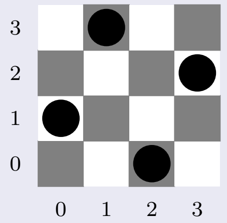
Solution

```
>> (solve-queens 4)
((3 . 2) (2 . 0) (1 . 3) (0 . 1))
```

Problem

- That's just one solution.
- Can we see the next one?

Board



Evaluator

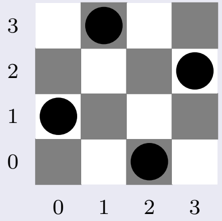
Solution

```
>> (solve-queens 4)
((3 . 2) (2 . 0) (1 . 3) (0 . 1))
```

Problem

- That's just one solution.
- Can we see the next one?

Board



Solution

- Clarify the choices made during the process of problem solving.
- "Either α or β" $\Longrightarrow$ "if α then α else β".

Evaluator

Choices - using or

```
(or (queens n (+ i 1) 0 (cons (cons i j) board))
    (queens n i (+ j 1) board)))
```

Evaluator

Choices - using or

```
(or (queens n (+ i 1) 0 (cons (cons i j) board))
    (queens n i (+ j 1) board)))
```

Choices - using if

```
(let ((solution?
       (queens n (+ i 1) 0 (cons (cons i j) board))))
  (if solution?
      solution?
      (queens n i (+ j 1) board)))
```

Choices

If a non-attacked place is found:

Evaluator

Choices - using or

```
(or (queens n (+ i 1) 0 (cons (cons i j) board))
    (queens n i (+ j 1) board)))
```

Choices - using if

```
(let ((solution?
       (queens n (+ i 1) 0 (cons (cons i j) board))))
  (if solution?
      solution?
      (queens n i (+ j 1) board)))
```

Choices

If a non-attacked place is found:

- Either place a queen and *try* to solve the rest of the board...

Evaluator

Choices - using or

```
(or (queens n (+ i 1) 0 (cons (cons i j) board))
    (queens n i (+ j 1) board)))
```

Choices - using if

```
(let ((solution?
       (queens n (+ i 1) 0 (cons (cons i j) board))))
  (if solution?
      solution?
      (queens n i (+ j 1) board)))
```

Choices

If a non-attacked place is found:

- Either place a queen and *try* to solve the rest of the board...
- ...or continue the search for a non-attacked place.

Evaluator

Next Solution

- The first solution is the sequence of the choices that avoided dead ends.
- In order to see the next solution, it is sufficient to consider the last successfull choice as a dead end.
- Currently, the choices are just pending computations...
- ...that disappear as soon as the first solution is returned.

Evaluator

Next Solution

- The first solution is the sequence of the choices that avoided dead ends.
- In order to see the next solution, it is sufficient to consider the last successfull choice as a dead end.
- Currently, the choices are just pending computations...
- ...that disappear as soon as the first solution is returned.
- But if we transform the program into continuation-passing style:
  - It becomes possible to save the sequence of choices that computed a solution.
  - It becomes possible to restore the sequence of choices using failure as an argument.

Evaluator

Solving the puzzle (in direct style)

```
(define (queens n i j board)
  (if (= i n)
      board
      (if (= j n)
          #f
          (if (attacked? i j board)
              (queens n i (+ j 1) board)
              (let ((solution?
                     (queens n (+ i 1) 0 (cons (cons i j) board))))
                (if solution?
                    solution?
                    (queens n i (+ j 1) board)))))))
```

Evaluator

Solving the puzzle (in direct style)

```
(define (queens n i j board)
  (if (= i n)
      board
      (if (= j n)
          #f
          (if (attacked? i j board)
              (queens n i (+ j 1) board)
              (let ((solution?
                     (queens n (+ i 1) 0 (cons (cons i j) board))))
                (if solution?
                    solution?
                    (queens n i (+ j 1) board)))))))
```

Transformation

- Include additional parameter for the continuation.

Evaluator

Solving the puzzle (in direct style)

```
(define (queens n i j board c)
  (if (= i n)
      board
      (if (= j n)
          #f
          (if (attacked? i j board)
              (queens n i (+ j 1) board)
              (let ((solution?
                      (queens n (+ i 1) 0 (cons (cons i j) board))))
                (if solution?
                    solution?
                    (queens n i (+ j 1) board)))))))
```

Evaluator

Solving the puzzle (in direct style)

```
(define (queens n i j board c)
  (if (= i n)
      board
      (if (= j n)
          #f
          (if (attacked? i j board)
              (queens n i (+ j 1) board)
              (let ((solution?
                     (queens n (+ i 1) 0 (cons (cons i j) board))))
                (if solution?
                    solution?
                    (queens n i (+ j 1) board)))))))
```

Transformation

- Each returned value becomes the argument of a continuation call.

Evaluator

Solving the puzzle (in direct style)

```
(define (queens n i j board c)
  (if (= i n)
      (c board)
      (if (= j n)
          (c #f)
          (if (attacked? i j board)
              (queens n i (+ j 1) board)
              (let ((solution?
                      (queens n (+ i 1) 0 (cons (cons i j) board))))
                (if solution?
                    (c solution?)
                    (queens n i (+ j 1) board)))))))
```

Evaluator

Solving the puzzle (in direct style)

```
(define (queens n i j board c)
  (if (= i n)
      (c board)
      (if (= j n)
          (c #f)
          (if (attacked? i j board)
              (queens n i (+ j 1) board)
              (let ((solution?
                     (queens n (+ i 1) 0 (cons (cons i j) board))))
                (if solution?
                    (c solution?)
                    (queens n i (+ j 1) board)))))))
```

Transformation

- Each tail call receives the current continuation.

Evaluator

Solving the puzzle (in direct style)

```
(define (queens n i j board c)
  (if (= i n)
      (c board)
      (if (= j n)
          (c #f)
          (if (attacked? i j board)
              (queens n i (+ j 1) board c)
              (let ((solution?
                      (queens n (+ i 1) 0 (cons (cons i j) board))))
                (if solution?
                    (c solution?)
                    (queens n i (+ j 1) board c)))))))
```

Evaluator

Solving the puzzle (in direct style)

```
(define (queens n i j board c)
  (if (= i n)
      (c board)
      (if (= j n)
          (c #f)
          (if (attacked? i j board)
              (queens n i (+ j 1) board c)
              (let ((solution?
                     (queens n (+ i 1) 0 (cons (cons i j) board))))
                (if solution?
                    (c solution?)
                    (queens n i (+ j 1) board c)))))))
```

Transformation

- Each pending computation becomes a new continuation.

Evaluator

Solving the puzzle (in direct style)

```
(define (queens n i j board c)
  (if (= i n)
      (c board)
      (if (= j n)
          (c #f)
          (if (attacked? i j board)
              (queens n i (+ j 1) board c)
              (queens n (+ i 1) 0 (cons (cons i j) board)
                      (lambda (solution?)
                        (if solution?
                            (c solution?)
                            (queens n i (+ j 1) board c))))))))
```

Evaluator

Solving the puzzle (in direct style)

```
(define (queens n i j board c)
  (if (= i n)
      (c board)
      (if (= j n)
          (c #f)
          (if (attacked? i j board)
              (queens n i (+ j 1) board c)
              (queens n (+ i 1) 0 (cons (cons i j) board)
                      (lambda (solution?)
                        (if solution?
                            (c solution?)
                            (queens n i (+ j 1) board c)))))))))
```

Transformation

- The initial continuation is the identity function.

Evaluator

Solving the puzzle (in continuation-passing style)

```
(define (queens n i j board c)
  (if (= i n)
      (c board)
      (if (= j n)
          (c #f)
          (if (attacked? i j board)
              (queens n i (+ j 1) board c)
              (queens n (+ i 1) 0 (cons (cons i j) board)
                      (lambda (solution?)
                        (if solution?
                            (c solution?)
                            (queens n i (+ j 1) board c)))))))))

(define (solve-queens n)
  (queens n 0 0 (list) (lambda (solution) solution)))
```

Evaluator

Solving the puzzle (in continuation-passing style)

```
(define (queens n i j board c)
  (if (= i n)
      (c board)
      (if (= j n)
          (c #f)
          (if (attacked? i j board)
              (queens n i (+ j 1) board c)
              (queens n (+ i 1) 0 (cons (cons i j) board)
                      (lambda (solution?)
                        (if solution?
                            (c solution?)
                            (queens n i (+ j 1) board c))))))))

(define (solve-queens n)
  (queens n 0 0 (list) (lambda (solution) solution)))
```

Transformation

- Done!

Evaluator

Solving the puzzle (in continuation-passing style)

```
(define (queens n i j board c)
  (if (= i n)
      (c board)
      (if (= j n)
          (c #f)
          (if (attacked? i j board)
              (queens n i (+ j 1) board c)
              (queens n (+ i 1) 0 (cons (cons i j) board)
                      (lambda (solution?)
                        (if solution?
                            (c solution?)
                            (queens n i (+ j 1) board c))))))))
```

Using Continuations

- If a solution is found, the continuation is called with the current board.

Evaluator

Solving the puzzle (in continuation-passing style)

```
(define (queens n i j board c)
  (if (= i n)
      (c board)
      (if (= j n)
          (c #f)
          (if (attacked? i j board)
              (queens n i (+ j 1) board c)
              (queens n (+ i 1) 0 (cons (cons i j) board)
                      (lambda (solution?)
                        (if solution?
                            (c solution?)
                            (queens n i (+ j 1) board c))))))))
```

Using Continuations

- If a dead end is found, the continuation is called with false.

Evaluator

Solving the puzzle (in continuation-passing style)

```
(define (queens n i j board c)
  (if (= i n)
      (c board)
      (if (= j n)
          (c #f)
          (if (attacked? i j board)
              (queens n i (+ j 1) board c)
              (queens n (+ i 1) 0 (cons (cons i j) board)
                      (lambda (solution?)
                        (if solution?
                            (c solution?)
                            (queens n i (+ j 1) board c)))))))))
```

Using Continuations

- If the current position is attacked, we try the next one, using the same continuation.

Evaluator

Solving the puzzle (in continuation-passing style)

```
(define (queens n i j board c)
  (if (= i n)
      (c board)
      (if (= j n)
          (c #f)
          (if (attacked? i j board)
              (queens n i (+ j 1) board c)
              (queens n (+ i 1) 0 (cons (cons i j) board)
                      (lambda (solution?)
                        (if solution?
                            (c solution?)
                            (queens n i (+ j 1) board c))))))))
```

Using Continuations

- If the current position is not attacked, we place a queen and we solve the rest of the problem with a new continuation.

Evaluator

Solving the puzzle (in continuation-passing style)

```
(define (queens n i j board c)
  (if (= i n)
      (c board)
      (if (= j n)
          (c #f)
          (if (attacked? i j board)
              (queens n i (+ j 1) board c)
              (queens n (+ i 1) 0 (cons (cons i j) board)
                      (lambda (solution?)
                        (if solution?
                            (c solution?)
                            (queens n i (+ j 1) board c)))))))))
```

Using Continuations

- When the continuation receives the result ...

Evaluator

Solving the puzzle (in continuation-passing style)

```
(define (queens n i j board c)
  (if (= i n)
      (c board)
      (if (= j n)
          (c #f)
          (if (attacked? i j board)
              (queens n i (+ j 1) board c)
              (queens n (+ i 1) 0 (cons (cons i j) board)
                      (lambda (solution?)
                        (if solution?
                            (c solution?)
                            (queens n i (+ j 1) board c))))))))
```

Using Continuations

- ...we test it to see if it is a solution.

Evaluator

Solving the puzzle (in continuation-passing style)

```
(define (queens n i j board c)
  (if (= i n)
      (c board)
      (if (= j n)
          (c #f)
          (if (attacked? i j board)
              (queens n i (+ j 1) board c)
              (queens n (+ i 1) 0 (cons (cons i j) board)
                      (lambda (solution?)
                        (if solution?
                            (c solution?)
                            (queens n i (+ j 1) board c)))))))))
```

Using Continuations

- If it is, we send it to our continuation.

Evaluator

Solving the puzzle (in continuation-passing style)

```
(define (queens n i j board c)
  (if (= i n)
      (c board)
      (if (= j n)
          (c #f)
          (if (attacked? i j board)
              (queens n i (+ j 1) board c)
              (queens n (+ i 1) 0 (cons (cons i j) board)
                      (lambda (solution?)
                        (if solution?
                            (c solution?)
                            (queens n i (+ j 1) board c)))))))))
```

Using Continuations

- Otherwise, we continue the search for a non-attacked postion.

Evaluator

Saving continuations

```
(define (queens n i j board c)
  (if (= i n)
      (c board)
      (if (= j n)
          (c #f)
          (if (attacked? i j board)
              (queens n i (+ j 1) board c)
              (queens n (+ i 1) 0 (cons (cons i j) board)
                      (lambda (solution?)
                        (if solution?
                            (c solution?)
                            (queens n i (+ j 1) board c))))))))
```

Evaluator

Saving continuations

```
(define (queens n i j board c)
  (if (= i n)
      (begin (set! next-solution c) (c board))
      (if (= j n)
          (c #f)
          (if (attacked? i j board)
              (queens n i (+ j 1) board c)
              (queens n (+ i 1) 0 (cons (cons i j) board)
                      (lambda (solution?)
                        (if solution?
                            (c solution?)
                            (queens n i (+ j 1) board c)))))))))
```

Evaluator

Saving continuations

```
(define next-solution #f)

(define (queens n i j board c)
  (if (= i n)
      (begin (set! next-solution c) (c board))
      (if (= j n)
          (c #f)
          (if (attacked? i j board)
              (queens n i (+ j 1) board c)
              (queens n (+ i 1) 0 (cons (cons i j) board)
                      (lambda (solution?)
                        (if solution?
                            (c solution?)
                            (queens n i (+ j 1) board c))))))))
```

Evaluator

Saving continuations

```
(define next-solution #f)

(define (queens n i j board c)
  (if (= i n)
      (begin (set! next-solution c) (c board))
      (if (= j n)
          (c #f)
          (if (attacked? i j board)
              (queens n i (+ j 1) board c)
              (queens n (+ i 1) 0 (cons (cons i j) board)
                      (lambda (solution?)
                        (if solution?
                            (c solution?)
                            (queens n i (+ j 1) board c))))))))
```

Evaluator

Solution

```
>> (solve-queens 4)
```

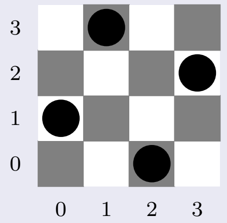
Board

Evaluator

Solution

```
>> (solve-queens 4)
((3 . 2) (2 . 0) (1 . 3) (0 . 1))
```

Board

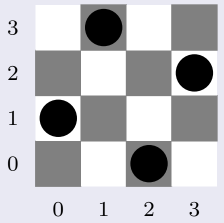


Evaluator

Solution

```
>> (solve-queens 4)
((3 . 2) (2 . 0) (1 . 3) (0 . 1))
>> (next-solution #f)
```

Board

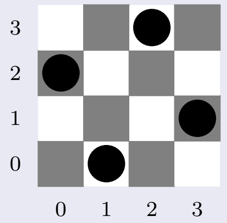


Evaluator

Solution

```
>> (solve-queens 4)
((3 . 2) (2 . 0) (1 . 3) (0 . 1))
>> (next-solution #f)
((3 . 1) (2 . 3) (1 . 0) (0 . 2))
```

Board

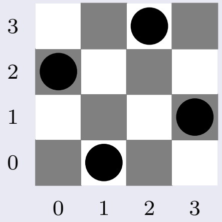


Evaluator

Solution

```
>> (solve-queens 4)
((3 . 2) (2 . 0) (1 . 3) (0 . 1))
>> (next-solution #f)
((3 . 1) (2 . 3) (1 . 0) (0 . 2))
>> (next-solution #f)
```

Board

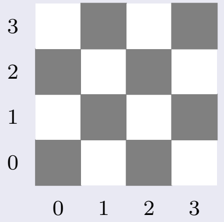


Evaluator

Solution

```
>> (solve-queens 4)
((3 . 2) (2 . 0) (1 . 3) (0 . 1))
>> (next-solution #f)
((3 . 1) (2 . 3) (1 . 0) (0 . 2))
>> (next-solution #f)
#f
```

Board

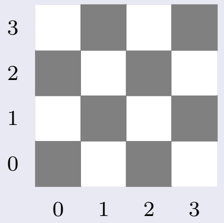


Evaluator

Solution

```
>> (solve-queens 4)
((3 . 2) (2 . 0) (1 . 3) (0 . 1))
>> (next-solution #f)
((3 . 1) (2 . 3) (1 . 0) (0 . 2))
>> (next-solution #f)
#f
>> (next-solution #f)
```

Board

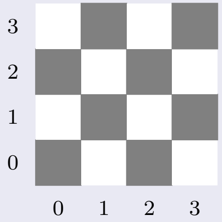


Evaluator

Solution

```
>> (solve-queens 4)
((3 . 2) (2 . 0) (1 . 3) (0 . 1))
>> (next-solution #f)
((3 . 1) (2 . 3) (1 . 0) (0 . 2))
>> (next-solution #f)
#f
>> (next-solution #f)
#f
```

Board



Nondeterministic Programming Languages

- Our program is now *nondeterministic*.
- It's just like Prolog.

Evaluator

Finding Solutions for 5-Queens

```
>> (solve-queens 5)
```

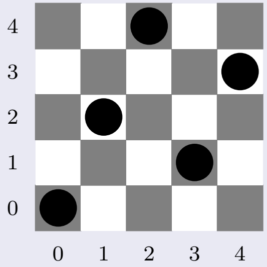
Board

Evaluator

Finding Solutions for 5-Queens

```
>> (solve-queens 5)
((4 . 3) (3 . 1) (2 . 4) (1 . 2) (0 . 0))
```

Board

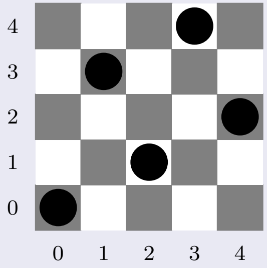


Evaluator

Finding Solutions for 5-Queens

```
>> (solve-queens 5)
((4 . 3) (3 . 1) (2 . 4) (1 . 2) (0 . 0))
>> (next-solution #f)
((4 . 2) (3 . 4) (2 . 1) (1 . 3) (0 . 0))
```

Board

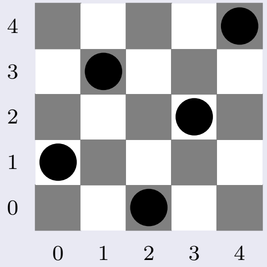


Evaluator

Finding Solutions for 5-Queens

```
>> (solve-queens 5)
((4 . 3) (3 . 1) (2 . 4) (1 . 2) (0 . 0))
>> (next-solution #f)
((4 . 2) (3 . 4) (2 . 1) (1 . 3) (0 . 0))
>> (next-solution #f)
((4 . 4) (3 . 2) (2 . 0) (1 . 3) (0 . 1))
```

Board

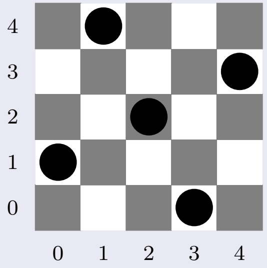


Evaluator

Finding Solutions for 5-Queens

```
>> (solve-queens 5)
((4 . 3) (3 . 1) (2 . 4) (1 . 2) (0 . 0))
>> (next-solution #f)
((4 . 2) (3 . 4) (2 . 1) (1 . 3) (0 . 0))
>> (next-solution #f)
((4 . 4) (3 . 2) (2 . 0) (1 . 3) (0 . 1))
>> (next-solution #f)
((4 . 3) (3 . 0) (2 . 2) (1 . 4) (0 . 1))
```

Board

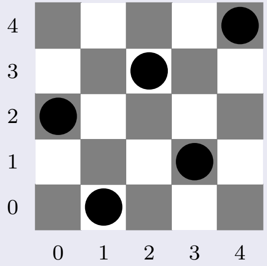


Evaluator

Finding Solutions for 5-Queens

```
>> (solve-queens 5)
((4 . 3) (3 . 1) (2 . 4) (1 . 2) (0 . 0))
>> (next-solution #f)
((4 . 2) (3 . 4) (2 . 1) (1 . 3) (0 . 0))
>> (next-solution #f)
((4 . 4) (3 . 2) (2 . 0) (1 . 3) (0 . 1))
>> (next-solution #f)
((4 . 3) (3 . 0) (2 . 2) (1 . 4) (0 . 1))
>> (next-solution #f)
((4 . 4) (3 . 1) (2 . 3) (1 . 0) (0 . 2))
```

Board

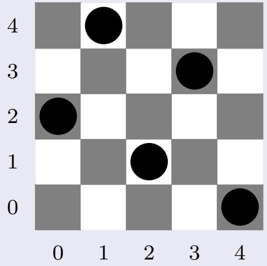


Evaluator

Finding Solutions for 5-Queens

```
>> (solve-queens 5)
((4 . 3) (3 . 1) (2 . 4) (1 . 2) (0 . 0))
>> (next-solution #f)
((4 . 2) (3 . 4) (2 . 1) (1 . 3) (0 . 0))
>> (next-solution #f)
((4 . 4) (3 . 2) (2 . 0) (1 . 3) (0 . 1))
>> (next-solution #f)
((4 . 3) (3 . 0) (2 . 2) (1 . 4) (0 . 1))
>> (next-solution #f)
((4 . 4) (3 . 1) (2 . 3) (1 . 0) (0 . 2))
>> (next-solution #f)
((4 . 0) (3 . 3) (2 . 1) (1 . 4) (0 . 2))
```

Board

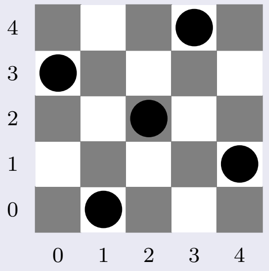


Evaluator

Finding Solutions for 5-Queens

```
>> (solve-queens 5)
((4 . 3) (3 . 1) (2 . 4) (1 . 2) (0 . 0))
>> (next-solution #f)
((4 . 2) (3 . 4) (2 . 1) (1 . 3) (0 . 0))
>> (next-solution #f)
((4 . 4) (3 . 2) (2 . 0) (1 . 3) (0 . 1))
>> (next-solution #f)
((4 . 3) (3 . 0) (2 . 2) (1 . 4) (0 . 1))
>> (next-solution #f)
((4 . 4) (3 . 1) (2 . 3) (1 . 0) (0 . 2))
>> (next-solution #f)
((4 . 0) (3 . 3) (2 . 1) (1 . 4) (0 . 2))
>> (next-solution #f)
((4 . 1) (3 . 4) (2 . 2) (1 . 0) (0 . 3))
```

Board

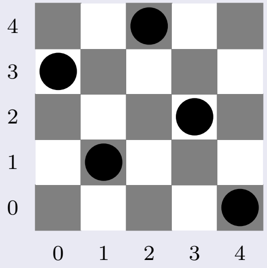


Evaluator

Finding Solutions for 5-Queens

```
>> (solve-queens 5)
((4 . 3) (3 . 1) (2 . 4) (1 . 2) (0 . 0))
>> (next-solution #f)
((4 . 2) (3 . 4) (2 . 1) (1 . 3) (0 . 0))
>> (next-solution #f)
((4 . 4) (3 . 2) (2 . 0) (1 . 3) (0 . 1))
>> (next-solution #f)
((4 . 3) (3 . 0) (2 . 2) (1 . 4) (0 . 1))
>> (next-solution #f)
((4 . 4) (3 . 1) (2 . 3) (1 . 0) (0 . 2))
>> (next-solution #f)
((4 . 0) (3 . 3) (2 . 1) (1 . 4) (0 . 2))
>> (next-solution #f)
((4 . 1) (3 . 4) (2 . 2) (1 . 0) (0 . 3))
>> (next-solution #f)
((4 . 0) (3 . 2) (2 . 4) (1 . 1) (0 . 3))
```

Board

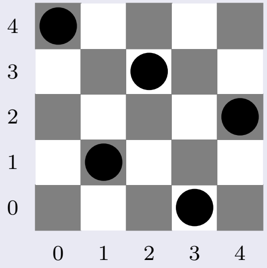


Evaluator

Finding Solutions for 5-Queens

```
>> (solve-queens 5)
((4 . 3) (3 . 1) (2 . 4) (1 . 2) (0 . 0))
>> (next-solution #f)
((4 . 2) (3 . 4) (2 . 1) (1 . 3) (0 . 0))
>> (next-solution #f)
((4 . 4) (3 . 2) (2 . 0) (1 . 3) (0 . 1))
>> (next-solution #f)
((4 . 3) (3 . 0) (2 . 2) (1 . 4) (0 . 1))
>> (next-solution #f)
((4 . 4) (3 . 1) (2 . 3) (1 . 0) (0 . 2))
>> (next-solution #f)
((4 . 0) (3 . 3) (2 . 1) (1 . 4) (0 . 2))
>> (next-solution #f)
((4 . 1) (3 . 4) (2 . 2) (1 . 0) (0 . 3))
>> (next-solution #f)
((4 . 0) (3 . 2) (2 . 4) (1 . 1) (0 . 3))
>> (next-solution #f)
((4 . 2) (3 . 0) (2 . 3) (1 . 1) (0 . 4))
```

Board

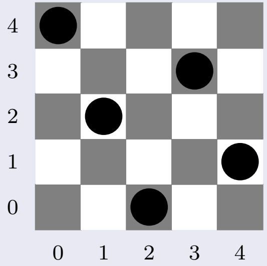


Evaluator

Finding Solutions for 5-Queens

```
>> (solve-queens 5)
((4 . 3) (3 . 1) (2 . 4) (1 . 2) (0 . 0))
>> (next-solution #f)
((4 . 2) (3 . 4) (2 . 1) (1 . 3) (0 . 0))
>> (next-solution #f)
((4 . 4) (3 . 2) (2 . 0) (1 . 3) (0 . 1))
>> (next-solution #f)
((4 . 3) (3 . 0) (2 . 2) (1 . 4) (0 . 1))
>> (next-solution #f)
((4 . 4) (3 . 1) (2 . 3) (1 . 0) (0 . 2))
>> (next-solution #f)
((4 . 0) (3 . 3) (2 . 1) (1 . 4) (0 . 2))
>> (next-solution #f)
((4 . 1) (3 . 4) (2 . 2) (1 . 0) (0 . 3))
>> (next-solution #f)
((4 . 0) (3 . 2) (2 . 4) (1 . 1) (0 . 3))
>> (next-solution #f)
((4 . 2) (3 . 0) (2 . 3) (1 . 1) (0 . 4))
>> (next-solution #f)
((4 . 1) (3 . 3) (2 . 0) (1 . 2) (0 . 4))
```

Board

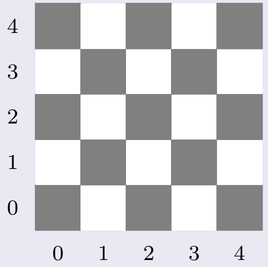


Evaluator

Finding Solutions for 5-Queens

```
>> (solve-queens 5)
((4 . 3) (3 . 1) (2 . 4) (1 . 2) (0 . 0))
>> (next-solution #f)
((4 . 2) (3 . 4) (2 . 1) (1 . 3) (0 . 0))
>> (next-solution #f)
((4 . 4) (3 . 2) (2 . 0) (1 . 3) (0 . 1))
>> (next-solution #f)
((4 . 3) (3 . 0) (2 . 2) (1 . 4) (0 . 1))
>> (next-solution #f)
((4 . 4) (3 . 1) (2 . 3) (1 . 0) (0 . 2))
>> (next-solution #f)
((4 . 0) (3 . 3) (2 . 1) (1 . 4) (0 . 2))
>> (next-solution #f)
((4 . 1) (3 . 4) (2 . 2) (1 . 0) (0 . 3))
>> (next-solution #f)
((4 . 0) (3 . 2) (2 . 4) (1 . 1) (0 . 3))
>> (next-solution #f)
((4 . 2) (3 . 0) (2 . 3) (1 . 1) (0 . 4))
>> (next-solution #f)
((4 . 1) (3 . 3) (2 . 0) (1 . 2) (0 . 4))
>> (next-solution #f)
#f
```

Board



Evaluator

Continuations

- Being forced to write programs in continuation-passing style is a **huge** disadvantage.
- Can we write an operator that captures the current continuation?

Continuations

- Yes, we can!
- Just convert the evaluator into continuation-passing style and implement the operator as simply "returning" the current continuation.

Evaluator

Continuations

- Being forced to write programs in continuation-passing style is a **huge** disadvantage.
- Can we write an operator that captures the current continuation?

Continuations

- Yes, we can!
- Just convert the evaluator into continuation-passing style and implement the operator as simply "returning" the current continuation.
- Or something like that.

Evaluator

Evaluator

```
(define (eval exp env)
  (cond ((self-evaluating? exp)
         exp)
        ...
        ((name? exp)
         (eval-name exp env))
        ((current-environment? exp)
         env)
        ((lambda? exp)
         (eval-lambda exp env))
        ((if? exp)
         (eval-if exp env))
        ((def? exp)
         (eval-def exp env))
        ...
        ((set? exp)
         (eval-set exp env))
        ((begin? exp)
         (eval-begin exp env))
        ((call? exp)
         (eval-call exp env))
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Evaluator

Evaluator in CPS

```
(define (eval exp env cont)
  (cond ((self-evaluating? exp)
         (cont exp))
        ...
        ((name? exp)
         (eval-name exp env cont))
        ((current-environment? exp)
         (cont env))
        ((lambda? exp)
         (eval-lambda exp env cont))
        ((if? exp)
         (eval-if exp env cont))
        ((def? exp)
         (eval-def exp env cont))
        ...
        ((set? exp)
         (eval-set exp env cont))
        ((begin? exp)
         (eval-begin exp env cont))
        ((call? exp)
         (eval-call exp env cont))
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Evaluator

Evaluator

```
(define (eval-name name env)
  (define (lookup-in-frame frame)
    (cond ((null? frame)
           ;;Search the rest of the environment
           (eval-name name (cdr env)))
          ((eq? name (caar frame))
           (cdar frame))
          (else
           ;;Search the rest of the frame
           (lookup-in-frame (cdr frame)))))
  (if (null? env)
      (error "Unbound name -- EVAL-NAME" name)
      (lookup-in-frame (car env))))
```

Evaluator

Evaluator in CPS

```
(define (eval-name name env cont)
  (define (lookup-in-frame frame)
    (cond ((null? frame)
           ;;Search the rest of the environment
           (eval-name name (cdr env) cont))
          ((eq? name (caar frame))
           (cont (cdar frame)))
          (else
           ;;Search the rest of the frame
           (lookup-in-frame (cdr frame)))))
  (if (null? env)
      (error "Unbound name -- EVAL-NAME" name)
      (lookup-in-frame (car env))))
```

Evaluator

Evaluator

```
(define (eval-lambda exp env)
  (make-function (lambda-parameters exp)
                 (lambda-body exp)
                 env))
```

Evaluator in CPS

```
(define (eval-lambda exp env cont)
  (cont
   (make-function (lambda-parameters exp)
                  (lambda-body exp)
                  env)))
```

Evaluator

Evaluator

```
(define (eval-lambda exp env)
  (make-function (lambda-parameters exp)
                 (lambda-body exp)
                 env))
```

Evaluator in CPS

```
(define (eval-lambda exp env cont)
  (cont
   (make-function (lambda-parameters exp)
                  (lambda-body exp)
                  env)))
```

Evaluator

Evaluator

```
(define (eval-if exp env)
  (if (true? (eval (if-condition exp) env))
      (eval (if-consequent exp) env)
      (eval (if-alternative exp) env)))
```

Evaluator in CPS

```
(define (eval-if exp env cont)
  (eval (if-condition exp) env
        (lambda (r)
          (if (true? r)
              (eval (if-consequent exp) env cont)
              (eval (if-alternative exp) env cont)))))
```

Evaluator

Evaluator

```
(define (eval-if exp env)
  (if (true? (eval (if-condition exp) env))
      (eval (if-consequent exp) env)
      (eval (if-alternative exp) env)))
```

Evaluator in CPS

```
(define (eval-if exp env cont)
  (eval (if-condition exp) env
        (lambda (r)
          (if (true? r)
              (eval (if-consequent exp) env cont)
              (eval (if-alternative exp) env cont)))))
```

Evaluator

Evaluator

```
(define (eval-if exp env)
  (if (true? (eval (if-condition exp) env))
      (eval (if-consequent exp) env)
      (eval (if-alternative exp) env)))
```

Evaluator in CPS

```
(define (eval-if exp env cont)
  (eval (if-condition exp) env
        (lambda (r)
          (if (true? r)
              (eval (if-consequent exp) env cont)
              (eval (if-alternative exp) env cont)))))
```

Evaluator

Evaluator

```
(define (eval-if exp env)
  (if (true? (eval (if-condition exp) env))
      (eval (if-consequent exp) env)
      (eval (if-alternative exp) env)))
```

Evaluator in CPS

```
(define (eval-if exp env cont)
  (eval (if-condition exp) env
        (lambda (r)
          (if (true? r)
              (eval (if-consequent exp) env cont)
              (eval (if-alternative exp) env cont)))))
```

Evaluator

Evaluator

```
(define (eval-def exp env)
  (let ((value (eval (def-init exp) env)))
    (define-name! (def-name exp) value env)
    value))
```

Evaluator in CPS

```
(define (eval-def exp env cont)
  (eval (def-init exp) env
        (lambda (value)
          (define-name! (def-name exp) value env)
          (cont value))))
```

Evaluator

Evaluator

```
(define (eval-def exp env)
  (let ((value (eval (def-init exp) env)))
    (define-name! (def-name exp) value env)
    value))
```

Evaluator in CPS

```
(define (eval-def exp env cont)
  (eval (def-init exp) env
        (lambda (value)
          (define-name! (def-name exp) value env)
          (cont value))))
```

Evaluator

Evaluator

```
(define (eval-def exp env)
  (let ((value (eval (def-init exp) env)))
    (define-name! (def-name exp) value env)
    value))
```

Evaluator in CPS

```
(define (eval-def exp env cont)
  (eval (def-init exp) env
        (lambda (value)
          (define-name! (def-name exp) value env)
          (cont value))))
```

Evaluator

Evaluator

```
(define (eval-set exp env)
  (let ((value (eval (assignment-expression exp) env)))
    (update-name! (assignment-name exp) value env)
    value))
```

Evaluator in CPS

```
(define (eval-set exp env cont)
  (eval (assignment-expression exp) env
        (lambda (value)
          (update-name! (assignment-name exp) value env)
          (cont value))))
```

Evaluator

Evaluator

```
(define (eval-set exp env)
  (let ((value (eval (assignment-expression exp) env)))
    (update-name! (assignment-name exp) value env)
    value))
```

Evaluator in CPS

```
(define (eval-set exp env cont)
  (eval (assignment-expression exp) env
        (lambda (value)
          (update-name! (assignment-name exp) value env)
          (cont value))))
```

Evaluator

Evaluator

```
(define (eval-set exp env)
  (let ((value (eval (assignment-expression exp) env)))
    (update-name! (assignment-name exp) value env)
    value))
```

Evaluator in CPS

```
(define (eval-set exp env cont)
  (eval (assignment-expression exp) env
        (lambda (value)
          (update-name! (assignment-name exp) value env)
          (cont value))))
```

Evaluator

Evaluator

```
(define (eval-begin exp env)
  (define (eval-sequence expressions env)
    (if (null? (cdr expressions))
        (eval (car expressions) env)
        (begin
          (eval (car expressions) env)
          (eval-sequence (cdr expressions) env))))
  (eval-sequence (begin-expressions exp) env))
```

Evaluator in CPS

```
(define (eval-begin exp env cont)
  (define (eval-sequence expressions env)
    (if (null? (cdr expressions))
        (eval (car expressions) env cont)
        (eval (car expressions) env
              (lambda (r) ;;ignore the result
                (eval-sequence (cdr expressions) env)))))
  (eval-sequence (begin-expressions exp) env))
```

Evaluator

Evaluator

```
(define (eval-begin exp env)
  (define (eval-sequence expressions env)
    (if (null? (cdr expressions))
        (eval (car expressions) env)
        (begin
          (eval (car expressions) env)
          (eval-sequence (cdr expressions) env))))
  (eval-sequence (begin-expressions exp) env))
```

Evaluator in CPS

```
(define (eval-begin exp env cont)
  (define (eval-sequence expressions env)
    (if (null? (cdr expressions))
        (eval (car expressions) env cont)
        (eval (car expressions) env
              (lambda (r) ;;ignore the result
                (eval-sequence (cdr expressions) env)))))
  (eval-sequence (begin-expressions exp) env))
```

Evaluator

Evaluator

```
(define (eval-begin exp env)
  (define (eval-sequence expressions env)
    (if (null? (cdr expressions))
        (eval (car expressions) env)
        (begin
          (eval (car expressions) env)
          (eval-sequence (cdr expressions) env))))
  (eval-sequence (begin-expressions exp) env))
```

Evaluator in CPS

```
(define (eval-begin exp env cont)
  (define (eval-sequence expressions env)
    (if (null? (cdr expressions))
        (eval (car expressions) env cont)
        (eval (car expressions) env
              (lambda (r) ;;ignore the result
                (eval-sequence (cdr expressions) env)))))
  (eval-sequence (begin-expressions exp) env))
```

Evaluator

Evaluator

```
(define (eval-begin exp env)
  (define (eval-sequence expressions env)
    (if (null? (cdr expressions))
        (eval (car expressions) env)
        (begin
          (eval (car expressions) env)
          (eval-sequence (cdr expressions) env))))
  (eval-sequence (begin-expressions exp) env))
```

Evaluator in CPS

```
(define (eval-begin exp env cont)
  (define (eval-sequence expressions env)
    (if (null? (cdr expressions))
        (eval (car expressions) env cont)
        (eval (car expressions) env
              (lambda (r) ;;ignore the result
                (eval-sequence (cdr expressions) env)))))
  (eval-sequence (begin-expressions exp) env))
```

Evaluator

Evaluator

```
(define (eval-call exp env)
  (let ((func (eval (call-operator exp) env)))
    (if (macro? func)
        (let ((expansion (apply-function func (call-operands exp))))
          (eval expansion env))
        (apply-function func (eval-exprs (call-operands exp) env)))))
```

Evaluator in CPS

```
(define (eval-call exp env cont)
  (eval (call-operator exp) env
        (lambda (func)
          (if (macro? func)
              (apply-function func (call-operands exp)
                              (lambda (expansion)
                                (eval expansion env cont)))
              (eval-exprs (call-operands exp) env
                          (lambda (args)
                            (apply-function func args cont)))))))
```

Evaluator

Evaluator

```
(define (eval-call exp env)
  (let ((func (eval (call-operator exp) env)))
    (if (macro? func)
        (let ((expansion (apply-function func (call-operands exp))))
          (eval expansion env))
        (apply-function func (eval-exprs (call-operands exp) env)))))
```

Evaluator in CPS

```
(define (eval-call exp env cont)
  (eval (call-operator exp) env
        (lambda (func)
          (if (macro? func)
              (apply-function func (call-operands exp)
                              (lambda (expansion)
                                (eval expansion env cont)))
              (eval-exprs (call-operands exp) env
                          (lambda (args)
                            (apply-function func args cont)))))))
```

Evaluator

Evaluator

```
(define (eval-call exp env)
  (let ((func (eval (call-operator exp) env)))
    (if (macro? func)
        (let ((expansion (apply-function func (call-operands exp))))
          (eval expansion env))
        (apply-function func (eval-exprs (call-operands exp) env)))))
```

Evaluator in CPS

```
(define (eval-call exp env cont)
  (eval (call-operator exp) env
        (lambda (func)
          (if (macro? func)
              (apply-function func (call-operands exp)
                              (lambda (expansion)
                                (eval expansion env cont)))
              (eval-exprs (call-operands exp) env
                          (lambda (args)
                            (apply-function func args cont)))))))
```

Evaluator

Evaluator

```
(define (eval-call exp env)
  (let ((func (eval (call-operator exp) env)))
    (if (macro? func)
        (let ((expansion (apply-function func (call-operands exp))))
          (eval expansion env))
        (apply-function func (eval-exprs (call-operands exp) env)))))
```

Evaluator in CPS

```
(define (eval-call exp env cont)
  (eval (call-operator exp) env
        (lambda (func)
          (if (macro? func)
              (apply-function func (call-operands exp)
                              (lambda (expansion)
                                (eval expansion env cont)))
              (eval-exprs (call-operands exp) env
                          (lambda (args)
                            (apply-function func args cont)))))))
```

Evaluator

Evaluator

```
(define (eval-call exp env)
  (let ((func (eval (call-operator exp) env)))
    (if (macro? func)
        (let ((expansion (apply-function func (call-operands exp))))
          (eval expansion env))
        (apply-function func (eval-exprs (call-operands exp) env)))))
```

Evaluator in CPS

```
(define (eval-call exp env cont)
  (eval (call-operator exp) env
        (lambda (func)
          (if (macro? func)
              (apply-function func (call-operands exp)
                              (lambda (expansion)
                                (eval expansion env cont)))
              (eval-exprs (call-operands exp) env
                          (lambda (args)
                            (apply-function func args cont)))))))
```

Evaluator

Evaluator

```
(define (eval-call exp env)
  (let ((func (eval (call-operator exp) env)))
    (if (macro? func)
        (let ((expansion (apply-function func (call-operands exp))))
          (eval expansion env))
        (apply-function func (eval-exprs (call-operands exp) env)))))
```

Evaluator in CPS

```
(define (eval-call exp env cont)
  (eval (call-operator exp) env
        (lambda (func)
          (if (macro? func)
              (apply-function func (call-operands exp)
                              (lambda (expansion)
                                (eval expansion env cont)))
              (eval-exprs (call-operands exp) env
                          (lambda (args)
                            (apply-function func args cont)))))))
```

Evaluator

Evaluator

```
(define (eval-call exp env)
  (let ((func (eval (call-operator exp) env)))
    (if (macro? func)
        (let ((expansion (apply-function func (call-operands exp))))
          (eval expansion env))
        (apply-function func (eval-exprs (call-operands exp) env)))))
```

Evaluator in CPS

```
(define (eval-call exp env cont)
  (eval (call-operator exp) env
        (lambda (func)
          (if (macro? func)
              (apply-function func (call-operands exp)
                              (lambda (expansion)
                                (eval expansion env cont)))
              (eval-exprs (call-operands exp) env
                          (lambda (args)
                            (apply-function func args cont)))))))
```

Evaluator

Evaluator

```
(define (apply-function func args)
  (if (primitive? func)
      (apply-primitive-function func args)
      (let ((extended-environment
             (augment-environment (function-parameters func)
                                  args
                                  (function-environment func))))
        (eval (function-body func) extended-environment))))
```

Evaluator in CPS

```
(define (apply-function func args cont)
  (if (primitive? func)
      (cont (apply-primitive-function func args))
      (let ((extended-environment
             (augment-environment (function-parameters func)
                                  args
                                  (function-environment func))))
        (eval (function-body func) extended-environment cont))))
```

Evaluator

Evaluator

```
(define (apply-function func args)
  (if (primitive? func)
      (apply-primitive-function func args)
      (let ((extended-environment
              (augment-environment (function-parameters func)
                                   args
                                   (function-environment func))))
        (eval (function-body func) extended-environment))))
```

Evaluator in CPS

```
(define (apply-function func args cont)
  (if (primitive? func)
      (cont (apply-primitive-function func args))
      (let ((extended-environment
              (augment-environment (function-parameters func)
                                   args
                                   (function-environment func))))
        (eval (function-body func) extended-environment cont))))
```

Evaluator

Evaluator

```
(define (apply-function func args)
  (if (primitive? func)
      (apply-primitive-function func args)
      (let ((extended-environment
              (augment-environment (function-parameters func)
                                   args
                                   (function-environment func))))
        (eval (function-body func) extended-environment))))
```

Evaluator in CPS

```
(define (apply-function func args cont)
  (if (primitive? func)
      (cont (apply-primitive-function func args))
      (let ((extended-environment
              (augment-environment (function-parameters func)
                                   args
                                   (function-environment func))))
        (eval (function-body func) extended-environment cont))))
```

Evaluator

Evaluator

```
(define (eval-exprs exprs env)
  (if (null? exprs)
      (list)
      (cons (eval (car exprs) env)
            (eval-exprs (cdr exprs) env))))
```

Evaluator in CPS

```
(define (eval-exprs exprs env cont)
  (if (null? exprs)
      (cont (list))
      (eval (car exprs) env
            (lambda (r)
              (eval-exprs (cdr exprs) env
                          (lambda (l)
                            (cont (cons r l))))))))
```

Evaluator

Evaluator

```
(define (eval-exprs exprs env)
  (if (null? exprs)
      (list)
      (cons (eval (car exprs) env)
            (eval-exprs (cdr exprs) env))))
```

Evaluator in CPS

```
(define (eval-exprs exprs env cont)
  (if (null? exprs)
      (cont (list))
      (eval (car exprs) env
            (lambda (r)
              (eval-exprs (cdr exprs) env
                          (lambda (l)
                            (cont (cons r l))))))))
```

Evaluator

Evaluator

```
(define (eval-exprs exprs env)
  (if (null? exprs)
      (list)
      (cons (eval (car exprs) env)
            (eval-exprs (cdr exprs) env))))
```

Evaluator in CPS

```
(define (eval-exprs exprs env cont)
  (if (null? exprs)
      (cont (list))
      (eval (car exprs) env
            (lambda (r)
              (eval-exprs (cdr exprs) env
                          (lambda (l)
                            (cont (cons r l))))))))
```

Evaluator

Evaluator

```
(define (eval-exprs exprs env)
  (if (null? exprs)
      (list)
      (cons (eval (car exprs) env)
            (eval-exprs (cdr exprs) env))))
```

Evaluator in CPS

```
(define (eval-exprs exprs env cont)
  (if (null? exprs)
      (cont (list))
      (eval (car exprs) env
            (lambda (r)
             (eval-exprs (cdr exprs) env
                          (lambda (l)
                            (cont (cons r l))))))))
```

Evaluator

Evaluator

```
(define (eval-exprs exprs env)
  (if (null? exprs)
      (list)
      (cons (eval (car exprs) env)
            (eval-exprs (cdr exprs) env))))
```

Evaluator in CPS

```
(define (eval-exprs exprs env cont)
  (if (null? exprs)
      (cont (list))
      (eval (car exprs) env
            (lambda (r)
              (eval-exprs (cdr exprs) env
                          (lambda (l)
                            (cont (cons r l))))))))
```

Evaluator

Evaluator

```
(define (repl)
  (prompt-for-input)
  (let ((input (read)))
    (let ((output (eval input initial-environment)))
      (myprint output)))
  (repl))
```

Evaluator in CPS

```
(define (repl)
  (prompt-for-input)
  (let ((input (read)))
    (eval input initial-environment
          (lambda (output)
            (myprint output)
            (repl)))))
```

Evaluator

Evaluator

```
(define (repl)
  (prompt-for-input)
  (let ((input (read)))
    (let ((output (eval input initial-environment)))
      (myprint output)))
  (repl))
```

Evaluator in CPS

```
(define (repl)
  (prompt-for-input)
  (let ((input (read)))
    (eval input initial-environment
          (lambda (output)
            (myprint output)
            (repl)))))
```

Evaluator

Evaluator

```
(define (repl)
  (prompt-for-input)
  (let ((input (read)))
    (let ((output (eval input initial-environment)))
      (myprint output)))
  (repl))
```

Evaluator in CPS

```
(define (repl)
  (prompt-for-input)
  (let ((input (read)))
    (eval input initial-environment
          (lambda (output)
            (myprint output)
            (repl)))))
```

Evaluator

Evaluator

```
(define (repl)
  (prompt-for-input)
  (let ((input (read)))
    (let ((output (eval input initial-environment)))
      (myprint output)))
  (repl))
```

Evaluator in CPS

```
(define (repl)
  (prompt-for-input)
  (let ((input (read)))
    (eval input initial-environment
          (lambda (output)
            (myprint output)
            (repl)))))
```

Evaluator

Continuations

- The evaluator is now in continuation-passing style.
- The transformation preserves the semantics of the program.
- But we now have an explicit representation of the pending computations of the evaluator.
- Note that this is **a procedure in the evaluating language**.
- The pending computations of the evaluator are directly related to the pending computations of the evaluated program.
- It is possible to make this continuation available to the evaluated program as **a procedure in the evaluated language**.
- Just wrap the continuation of the evaluator into a primitive procedure of the evaluated language.

Evaluator

Implementing `current-continuation`

```
;;(current-continuation)
```

Evaluator

Implementing current-continuation

```
;;(current-continuation)

(define (current-continuation? exp)
  (and (pair? exp)
       (eq? (car exp) 'current-continuation)))
```

Evaluator

Implementing current-continuation

```
;;(current-continuation)

(define (current-continuation? exp)
  (and (pair? exp)
       (eq? (car exp) 'current-continuation)))

(define (eval exp env cont)
  (cond ((self-evaluating? exp)
         (cont exp))
        ...
        ((name? exp)
         (eval-name exp env cont))
        ((current-environment? exp)
         (cont env))


        ((lambda? exp)
         (eval-lambda exp env cont))
        ...
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Evaluator

Implementing current-continuation

```
;;(current-continuation)

(define (current-continuation? exp)
  (and (pair? exp)
       (eq? (car exp) 'current-continuation)))

(define (eval exp env cont)
  (cond ((self-evaluating? exp)
         (cont exp))
        ...
        ((name? exp)
         (eval-name exp env cont))
        ((current-environment? exp)
         (cont env))
        ((current-continuation? exp)
         (cont (make-primitive cont)))
        ((lambda? exp)
         (eval-lambda exp env cont))
        ...
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Evaluator

Testing `current-continuation`

```
>> (define foo (current-continuation))
```

Evaluator

Testing current-continuation

```
>> (define foo (current-continuation))
(primitive #<closure 0x975264c0>)
```

Evaluator

Testing current-continuation

```
>> (define foo (current-continuation))
(primitive #<closure 0x975264c0>)
>> foo
```

Evaluator

Testing current-continuation

```
>> (define foo (current-continuation))
(primitive #<closure 0x975264c0>)
>> foo
(primitive #<closure 0x975264c0>)
```

Evaluator

Testing `current-continuation`

```
>> (define foo (current-continuation))
(primitive #<closure 0x975264c0>)
>> foo
(primitive #<closure 0x975264c0>)
>> (foo 1)
```

Evaluator

Testing current-continuation

```
>> (define foo (current-continuation))
(primitive #<closure 0x975264c0>)
>> foo
(primitive #<closure 0x975264c0>)
>> (foo 1)
1
```

Evaluator

Testing current-continuation

```
>> (define foo (current-continuation))
(primitive #<closure 0x975264c0>)
>> foo
(primitive #<closure 0x975264c0>)
>> (foo 1)
1
>> foo
```

Evaluator

Testing `current-continuation`

```
>> (define foo (current-continuation))
(primitive #<closure 0x975264c0>)
>> foo
(primitive #<closure 0x975264c0>)
>> (foo 1)
1
>> foo
1
```

Evaluator

Testing current-continuation

```
>> (define foo (current-continuation))
(primitive #<closure 0x975264c0>)
>> foo
(primitive #<closure 0x975264c0>)
>> (foo 1)
1
>> foo
1
```

Explanation

- When the continuation was captured, we were waiting for a value to assign to `foo`.
- When we called the continuation with argument 1, it means that the value we were waiting to assign to `foo` was the number 1.
- It means that we now are evaluating `(define foo 1)`.

Evaluator

Testing `current-continuation`

```
>> (current-continuation)
```

Evaluator

Testing `current-continuation`

```
>> (current-continuation)
(primitive #<closure 0x970f48c0>)
```

Evaluator

Testing current-continuation

```
>> (current-continuation)
(primitive #<closure 0x970f48c0>)
>> ((current-continuation) 1)
```

Evaluator

Testing `current-continuation`

```
>> (current-continuation)
(primitive #<closure 0x970f48c0>)
>> ((current-continuation) 1)
ERROR ;;Why?
```

Evaluator

Testing `current-continuation`

```
>> (current-continuation)
(primitive #<closure 0x970f48c0>)
>> ((current-continuation) 1)
ERROR ;;Why?
```

Explanation

- When the continuation was captured, we were waiting for a function to call with argument 1.
- When we called the continuation with argument 1, the current continuation was replaced with the captured continuation as if **the function** we were waiting to call with argument 1 was **the number** 1.
- It means that we now are trying to evaluate `(1 1)`.

Evaluator

Testing current-continuation

```
>> (let ((c (current-continuation)))
     (display "Hi!")
     (+ 1 (c (lambda (x) 1))))
```

Evaluator

Testing current-continuation

```
>> (let ((c (current-continuation)))
     (display "Hi!")
     (+ 1 (c (lambda (x) 1))))
Hi!Hi!2
```

Evaluator

Testing `current-continuation`

```
>> (let ((c (current-continuation)))
     (display "Hi!")
     (+ 1 (c (lambda (x) 1))))
Hi!Hi!2
```

Explanation

- When the continuation was captured, we were waiting for a value to assign to c **and then** to display "Hi!" **and then** to add 1 to the invocation of c with a function that returns 1.
- We then did all the computations we were waiting to do.
- At the call to c, we went back to the assignment to c that now becomes bound to the function `(lambda (x) 1)` and then we did all the computations we were waiting to do until we called c that now is a function that simply returns 1.

Evaluator

Testing current-continuation

```
>> (let ((longjmp (current-continuation)))
     (if longjmp
         (begin
           (display "Before the longjmp")
           (newline)
           (longjmp #f)
           (display "Did I survive the longjmp?"))
         (begin
           (display "After the longjmp"))))
```

Evaluator

Testing current-continuation

```
>> (let ((longjmp (current-continuation)))
     (if longjmp
         (begin
           (display "Before the longjmp")
           (newline)
           (longjmp #f)
           (display "Did I survive the longjmp?"))
         (begin
           (display "After the longjmp"))))
Before the longjmp
```

Evaluator

Testing current-continuation

```
>> (let ((longjmp (current-continuation)))
     (if longjmp
         (begin
           (display "Before the longjmp")
           (newline)
           (longjmp #f)
           (display "Did I survive the longjmp?"))
         (begin
           (display "After the longjmp"))))
Before the longjmp
After the longjmp
```

Evaluator

Testing current-continuation

```
>> (let ((longjmp (current-continuation)))
     (if longjmp
         (begin
           (display "Before the longjmp")
           (newline)
           (longjmp #f)
           (display "Did I survive the longjmp?"))
         (begin
           (display "After the longjmp"))))
Before the longjmp
After the longjmp
```

Explanation

- Go see the setjmp/longjmp available in C.

Evaluator

More Useful Continuations

- In our current implementation, the *use* of a continuation is usually part of the continuation itself.
- This means that (in general) the use of the continuation will re-evaluate its use when, presumably, that second use was not intended.
- It's preferable to separate the part of the program that *captures* the continuation from the part of the program that *uses* the continuation.
- That's the purpose of the function `call/cc` (also known as `call-with-current-continuation`).

Evaluator

Implementing call/cc

```
;; (call/cc (lambda (c) ...))
```

Evaluator

Implementing call/cc

```
;; (call/cc (lambda (c) ...))

(define (call/cc? exp)
  (and (pair? exp)
       (eq? (car exp) 'call/cc)))
```

Evaluator

Implementing call/cc

```
;; (call/cc (lambda (c) ...))

(define (call/cc? exp)
  (and (pair? exp)
       (eq? (car exp) 'call/cc)))

(define (call/cc-expression exp)
  (cadr exp))
```

Evaluator

Implementing call/cc

```
;; (call/cc (lambda (c) ...))

(define (call/cc? exp)
  (and (pair? exp)
       (eq? (car exp) 'call/cc)))

(define (call/cc-expression exp)
  (cadr exp))

(define (eval exp env cont)
  (cond ...


        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Evaluator

Implementing call/cc

```
;; (call/cc (lambda (c) ...))

(define (call/cc? exp)
  (and (pair? exp)
       (eq? (car exp) 'call/cc)))

(define (call/cc-expression exp)
  (cadr exp))

(define (eval exp env cont)
  (cond ...
        ((call/cc? exp)
         (eval-call/cc exp env cont))
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

Evaluator

Implementing call/cc

```
;; (call/cc (lambda (c) ...))

(define (call/cc? exp)
  (and (pair? exp)
       (eq? (car exp) 'call/cc)))

(define (call/cc-expression exp)
  (cadr exp))

(define (eval exp env cont)
  (cond ...
        ((call/cc? exp)
         (eval-call/cc exp env cont))
        (else
         (error "Unknown expression type -- EVAL" exp))))

(define (eval-call/cc exp env cont)
  (eval `(,(call/cc-expression exp)
          ',(make-primitive cont))
        env
        cont))
```

Evaluator

Implementing call/cc

- Note that call/cc is implemented as special syntax in our evaluator but is a “normal” function in the Scheme language.

Implementing current-continuation

```
(define (current-continuation)
  (call/cc (lambda (c) c)))
```

call/cc *vs* current-continuation

- call/cc is sufficient for implementing current-continuation.
- call/cc makes it easier to implement other control structures.

Evaluator

Example

```
>> (define foo (call/cc (lambda (c) c)))
```

Evaluator

Example

```
>> (define foo (call/cc (lambda (c) c)))
(primitive #<closure 0xbd961420>)
```

Evaluator

Example

```
>> (define foo (call/cc (lambda (c) c)))
(primitive #<closure 0xbd961420>)
>> foo
```

Evaluator

Example

```
>> (define foo (call/cc (lambda (c) c)))
(primitive #<closure 0xbd961420>)
>> foo
(primitive #<closure 0xbd961420>)
```

Evaluator

Example

```
>> (define foo (call/cc (lambda (c) c)))
(primitive #<closure 0xbd961420>)
>> foo
(primitive #<closure 0xbd961420>)
>> (foo 1)
```

Evaluator

Example

```
>> (define foo (call/cc (lambda (c) c)))
(primitive #<closure 0xbd961420>)
>> foo
(primitive #<closure 0xbd961420>)
>> (foo 1)
1
```

Evaluator

Example

```
>> (define foo (call/cc (lambda (c) c)))
(primitive #<closure 0xbd961420>)
>> foo
(primitive #<closure 0xbd961420>)
>> (foo 1)
1
>> foo
```

Evaluator

Example

```
>> (define foo (call/cc (lambda (c) c)))
(primitive #<closure 0xbd961420>)
>> foo
(primitive #<closure 0xbd961420>)
>> (foo 1)
1
>> foo
1
```

Evaluator

Example

```
>> (define foo (call/cc (lambda (c) c)))
(primitive #<closure 0xbd961420>)
>> foo
(primitive #<closure 0xbd961420>)
>> (foo 1)
1
>> foo
1
>> (call/cc (lambda (c) (set! foo c)))
```

Evaluator

Example

```
>> (define foo (call/cc (lambda (c) c)))
(primitive #<closure 0xbd961420>)
>> foo
(primitive #<closure 0xbd961420>)
>> (foo 1)
1
>> foo
1
>> (call/cc (lambda (c) (set! foo c)))
(primitive #<closure 0xbd721560>)
```

Evaluator

Example

```
>> (define foo (call/cc (lambda (c) c)))
(primitive #<closure 0xbd961420>)
>> foo
(primitive #<closure 0xbd961420>)
>> (foo 1)
1
>> foo
1
>> (call/cc (lambda (c) (set! foo c)))
(primitive #<closure 0xbd721560>)
>> (foo 1)
```

Evaluator

Example

```
>> (define foo (call/cc (lambda (c) c)))
(primitive #<closure 0xbd961420>)
>> foo
(primitive #<closure 0xbd961420>)
>> (foo 1)
1
>> foo
1
>> (call/cc (lambda (c) (set! foo c)))
(primitive #<closure 0xbd721560>)
>> (foo 1)
1
```

Evaluator

Example

```
>> (define foo (call/cc (lambda (c) c)))
(primitive #<closure 0xbd961420>)
>> foo
(primitive #<closure 0xbd961420>)
>> (foo 1)
1
>> foo
1
>> (call/cc (lambda (c) (set! foo c)))
(primitive #<closure 0xbd721560>)
>> (foo 1)
1
>> foo
```

Evaluator

Example

```
>> (define foo (call/cc (lambda (c) c)))
(primitive #<closure 0xbd961420>)
>> foo
(primitive #<closure 0xbd961420>)
>> (foo 1)
1
>> foo
1
>> (call/cc (lambda (c) (set! foo c)))
(primitive #<closure 0xbd721560>)
>> (foo 1)
1
>> foo
(primitive #<closure 0xc85d7c80>)
```

Evaluator

Example

```
>> (define foo (call/cc (lambda (c) c)))
(primitive #<closure 0xbd961420>)
>> foo
(primitive #<closure 0xbd961420>)
>> (foo 1)
1
>> foo
1
>> (call/cc (lambda (c) (set! foo c)))
(primitive #<closure 0xbd721560>)
>> (foo 1)
1
>> foo
(primitive #<closure 0xc85d7c80>)
>> (+ 2 (foo 1))
```

Evaluator

Example

```
>> (define foo (call/cc (lambda (c) c)))
(primitive #<closure 0xbd961420>)
>> foo
(primitive #<closure 0xbd961420>)
>> (foo 1)
1
>> foo
1
>> (call/cc (lambda (c) (set! foo c)))
(primitive #<closure 0xbd721560>)
>> (foo 1)
1
>> foo
(primitive #<closure 0xc85d7c80>)
>> (+ 2 (foo 1))
1
```

Evaluator

Example

```
>> (define foo (call/cc (lambda (c) c)))
(primitive #<closure 0xbd961420>)
>> foo
(primitive #<closure 0xbd961420>)
>> (foo 1)
1
>> foo
1
>> (call/cc (lambda (c) (set! foo c)))
(primitive #<closure 0xbd721560>)
>> (foo 1)
1
>> foo
(primitive #<closure 0xc85d7c80>)
>> (+ 2 (foo 1))
1
>> (+ 2 (* 3 (foo 1)))
```

Evaluator

Example

```
>> (define foo (call/cc (lambda (c) c)))
(primitive #<closure 0xbd961420>)
>> foo
(primitive #<closure 0xbd961420>)
>> (foo 1)
1
>> foo
1
>> (call/cc (lambda (c) (set! foo c)))
(primitive #<closure 0xbd721560>)
>> (foo 1)
1
>> foo
(primitive #<closure 0xc85d7c80>)
>> (+ 2 (foo 1))
1
>> (+ 2 (* 3 (foo 1)))
1
```

Evaluator

Example

```
>> (define foo (call/cc (lambda (c) c)))
(primitive #<closure 0xbd961420>)
>> foo
(primitive #<closure 0xbd961420>)
>> (foo 1)
1
>> foo
1
>> (call/cc (lambda (c) (set! foo c)))
(primitive #<closure 0xbd721560>)
>> (foo 1)
1
>> foo
(primitive #<closure 0xc85d7c80>)
>> (+ 2 (foo 1))
1
>> (+ 2 (* 3 (foo 1)))
1
>> (foo (+ 2 (foo (* 2 (foo 1)))))
```

Evaluator

Example

```
>> (define foo (call/cc (lambda (c) c)))
(primitive #<closure 0xbd961420>)
>> foo
(primitive #<closure 0xbd961420>)
>> (foo 1)
1
>> foo
1
>> (call/cc (lambda (c) (set! foo c)))
(primitive #<closure 0xbd721560>)
>> (foo 1)
1
>> foo
(primitive #<closure 0xc85d7c80>)
>> (+ 2 (foo 1))
1
>> (+ 2 (* 3 (foo 1)))
1
>> (foo (+ 2 (foo (* 2 (foo 1)))))
1
```

Evaluator

Aborting a Computation

```
>> (define abort-top-level #f)
#f
>> (call/cc (lambda (c) (set! abort-top-level c)))
(primitive #<closure 0xbca29d20>)
```

Remembering `mystery`

Evaluator

Aborting a Computation

```
>> (define abort-top-level #f)
#f
>> (call/cc (lambda (c) (set! abort-top-level c)))
(primitive #<closure 0xbca29d20>)
```

Remembering mystery

```
(define (mystery a b c d e)
  (+ a (* b (safe-/ (- c d) e))))

(define (safe-/ x y)
  (if (= y 0)
      (error "Can't divide " (list x y))
      (/ x y)))

(define (error msg args)
```

Evaluator

Aborting a Computation

```
>> (define abort-top-level #f)
#f
>> (call/cc (lambda (c) (set! abort-top-level c)))
(primitive #<closure 0xbca29d20>)
```

Remembering mystery

```
(define (mystery a b c d e)
  (+ a (* b (safe-/ (- c d) e))))

(define (safe-/ x y)
  (if (= y 0)
      (error "Can't divide " (list x y))
      (/ x y)))

(define (error msg args)
  (display msg)
  (display args)
```

Evaluator

Aborting a Computation

```
>> (define abort-top-level #f)
#f
>> (call/cc (lambda (c) (set! abort-top-level c)))
(primitive #<closure 0xbca29d20>)
```

Remembering mystery

```
(define (mystery a b c d e)
  (+ a (* b (safe-/ (- c d) e))))

(define (safe-/ x y)
  (if (= y 0)
      (error "Can't divide " (list x y))
      (/ x y)))

(define (error msg args)
  (display msg)
  (display args)
  (abort-top-level #f))
```

Evaluator

Breaking a Computation

```
(define (palindrome n f)
  (if (= n 0)
      (f n)
      (begin
        (display n)
        (palindrome (- n 1) f)
        (display n))))
```

Evaluator

Breaking a Computation

```
(define (palindrome n f)
  (if (= n 0)
      (f n)
      (begin
        (display n)
        (palindrome (- n 1) f)
        (display n))))

>> (palindrome 9 display)
```

Evaluator

Breaking a Computation

```
(define (palindrome n f)
  (if (= n 0)
      (f n)
      (begin
        (display n)
        (palindrome (- n 1) f)
        (display n))))

>> (palindrome 9 display)
9876543210123456789...
```

Evaluator

Breaking a Computation

```
(define (palindrome n f)
  (if (= n 0)
      (f n)
      (begin
        (display n)
        (palindrome (- n 1) f)
        (display n))))

>> (palindrome 9 display)
9876543210123456789...
>> (call/cc (lambda (return)
              (palindrome 9 return)))
```

Evaluator

Breaking a Computation

```
(define (palindrome n f)
  (if (= n 0)
      (f n)
      (begin
        (display n)
        (palindrome (- n 1) f)
        (display n))))

>> (palindrome 9 display)
9876543210123456789...
>> (call/cc (lambda (return)
              (palindrome 9 return)))
9876543210
```

Evaluator

Breaking a Computation

```
(define (palindrome n f)
  (if (= n 0)
      (f n)
      (begin
        (display n)
        (palindrome (- n 1) f)
        (display n))))

>> (palindrome 9 display)
9876543210123456789...
>> (call/cc (lambda (return)
              (palindrome 9 return)))
9876543210
;;That was the Scheme way.
;;What about the Common Lisp way?
```

Evaluator

Breaking a Computation

```
(define (palindrome n f)
  (if (= n 0)
      (f n)
      (begin
        (display n)
        (palindrome (- n 1) f)
        (display n))))

>> (palindrome 9 display)
98765432101234567 89...
>> (call/cc (lambda (return)
              (palindrome 9 return)))
9876543210
;;That was the Scheme way.
;;What about the Common Lisp way?
>> (block end
     (palindrome 9 (lambda (r)
                     (return-from end r))))
9876543210
```

Evaluator

Common Lisp's `block`/`return-from`

```
(mdef block (name body)
  `(call/cc (lambda (,name)
                ,body)))

(define (return-from name value)
  (name value))
```

Common Lisp's `block`/`return-from`

- The operators `block` and `return-from` provide a structured, lexical, non-local exit facility.
- At any point lexically contained within forms, `return-from` can be used with the given name to return control and values from the block form.

Evaluator

Java's try-catch/throw

```
(define (foo n)
  (+ 1 (try (bar n)
        catch error-a
        (display "Handling error-a ")
        0)))

(define (bar n)
  (+ 1 (try (baz n)
        catch error-b
        (display "Handling error-b ")
        1)))

(define (baz n)
  (if (= n 0)
      (throw 'error-a)
      (if (= n 1)
          (throw 'error-b)
          n)))
```

Output

Evaluator

Java's try-catch/throw

```
(define (foo n)
  (+ 1 (try (bar n)
        catch error-a
        (display "Handling error-a ")
        0)))

(define (bar n)
  (+ 1 (try (baz n)
        catch error-b
        (display "Handling error-b ")
        1)))

(define (baz n)
  (if (= n 0)
      (throw 'error-a)
      (if (= n 1)
          (throw 'error-b)
          n)))
```

Output

```
>> (foo 0)
```

Evaluator

Java's try-catch/throw

```
(define (foo n)
  (+ 1 (try (bar n)
         catch error-a
         (display "Handling error-a ")
         0)))

(define (bar n)
  (+ 1 (try (baz n)
         catch error-b
         (display "Handling error-b ")
         1)))

(define (baz n)
  (if (= n 0)
      (throw 'error-a)
      (if (= n 1)
          (throw 'error-b)
          n)))
```

Output

```
>> (foo 0)
Handling error-a 1
```

Evaluator

Java's try-catch/throw

```
(define (foo n)
  (+ 1 (try (bar n)
         catch error-a
         (display "Handling error-a ")
         0)))

(define (bar n)
  (+ 1 (try (baz n)
         catch error-b
         (display "Handling error-b ")
         1)))

(define (baz n)
  (if (= n 0)
      (throw 'error-a)
      (if (= n 1)
          (throw 'error-b)
          n)))
```

Output

```
>> (foo 0)
Handling error-a 1
>> (foo 1)
```

Evaluator

Java's try-catch/throw

```
(define (foo n)
  (+ 1 (try (bar n)
        catch error-a
        (display "Handling error-a ")
        0)))

(define (bar n)
  (+ 1 (try (baz n)
        catch error-b
        (display "Handling error-b ")
        1)))

(define (baz n)
  (if (= n 0)
      (throw 'error-a)
      (if (= n 1)
          (throw 'error-b)
          n)))
```

Output

```
>> (foo 0)
Handling error-a 1
>> (foo 1)
Handling error-b 3
```

Evaluator

Java's try-catch/throw

```
(define (foo n)
  (+ 1 (try (bar n)
        catch error-a
        (display "Handling error-a ")
        0)))

(define (bar n)
  (+ 1 (try (baz n)
        catch error-b
        (display "Handling error-b ")
        1)))

(define (baz n)
  (if (= n 0)
      (throw 'error-a)
      (if (= n 1)
          (throw 'error-b)
          n)))
```

Output

```
>> (foo 0)
Handling error-a 1
>> (foo 1)
Handling error-b 3
>> (foo 2)
```

Evaluator

Java's try-catch/throw

```
(define (foo n)
  (+ 1 (try (bar n)
         catch error-a
         (display "Handling error-a ")
         0)))

(define (bar n)
  (+ 1 (try (baz n)
         catch error-b
         (display "Handling error-b ")
         1)))

(define (baz n)
  (if (= n 0)
      (throw 'error-a)
      (if (= n 1)
          (throw 'error-b)
          n)))
```

Output

```
>> (foo 0)
Handling error-a 1
>> (foo 1)
Handling error-b 3
>> (foo 2)
4
```

Evaluator

Java's try-catch/throw

```
(define error-handlers (list))
```

Evaluator

Java's try-catch/throw

```
(define error-handlers (list))

(define (push-error-handler tag eh)
  (set! error-handlers (cons (cons tag eh) error-handlers)))
```

Evaluator

Java's try-catch/throw

```
(define error-handlers (list))

(define (push-error-handler tag eh)
  (set! error-handlers (cons (cons tag eh) error-handlers)))

(define (pop-error-handler)
  (set! error-handlers (cdr error-handlers)))
```

Evaluator

Java's try-catch/throw

```
(define error-handlers (list))

(define (push-error-handler tag eh)
  (set! error-handlers (cons (cons tag eh) error-handlers)))

(define (pop-error-handler)
  (set! error-handlers (cdr error-handlers)))

(define (find-error-handler tag)
  (if (null? error-handlers)
      abort-top-level
      (let ((tag-eh (car error-handlers)))
        (set! error-handlers (cdr error-handlers))
        (if (eq? tag (car tag-eh))
            (cdr tag-eh)
            (find-error-handler tag)))))
```

Evaluator

Java's try-catch/throw

```
(mdef try (expr catch tag . handler)
  `(call-with-error-handler
    (lambda () ,expr)
    ',tag
    (lambda () ,@handler)))
```

Evaluator

Java's try-catch/throw

```
(mdef try (expr catch tag . handler)
  `(call-with-error-handler
    (lambda () ,expr)
    ',tag
    (lambda () ,@handler)))

(define (call-with-error-handler f tag eh)
  (call/cc (lambda (c)
             (push-error-handler
               tag
               (lambda () (c (eh))))
             (let ((result (f)))
               (pop-error-handler)
               result))))
```

Evaluator

Java's try-catch/throw

```
(mdef try (expr catch tag . handler)
  `(call-with-error-handler
    (lambda () ,expr)
    ',tag
    (lambda () ,@handler)))

(define (call-with-error-handler f tag eh)
  (call/cc (lambda (c)
             (push-error-handler
               tag
               (lambda () (c (eh))))
             (let ((result (f)))
               (pop-error-handler)
               result))))

(define (throw tag)
  ((find-error-handler tag)))
```

Evaluator

Nondeterministic Programs

- Certain expressions in a nondeterministic language can have a different value in each different *world*.
- Each *world* encodes one of many possible sequences of *choices*, each choice corresponding to one of the possible values of a nondeterministic expression.
- In practice:
  - The evaluator computes just one of the possible values of a nondeterministic expression.
  - But remembers that there were more choices available.
  - If the choice taken does not satisfy subsequent requirements, a different choice is attempted.
  - This process is repeated until the evaluation succeeds or the evaluation runs out of choices.

Evaluator

A Pythagorean Triple

```
(define (a-pythagorean-triple n)
  (let ((i (an-integer-between 1 n)))
    (let ((j (an-integer-between i n)))
      (let ((k (an-integer-between j n)))
        (require (= (+ (* i i) (* j j)) (* k k)))
        (list i j k)))))
```

Nondeterministic Programs

- The function `a-pythagorean-triple` uses the function `(an-integer-between` *a* *b*`)` that nondeterministically returns one integer between *a* *b*.
- The function `require` accepts a boolean expression and *requires* that the computation must satisfy the expression. Otherwise, the computation *fails*.

Evaluator

An Integer Between a and b

```
(define (an-integer-between a b)
  (if (> a b)
      (fail)
      (amb a
           (an-integer-between (+ a 1) b))))
```

The `amb` (ambiguous) operator

- Invented by John McCarthy in 1961.
- Nondeterministically returns the value of one of the expressions.

The `fail` operator

- Cause the *current* computation to fail.
- It means that the choices that were taken in the current computation were not good.

Evaluator

amb and fail

```
>> (amb 1 2)
```

Evaluator

amb and fail

```
>> (amb 1 2)
1
```

Evaluator

amb and fail

```
>> (amb 1 2)
1
>> (fail)
```

Evaluator

amb and fail

```
>> (amb 1 2)
1
>> (fail)
2
```

Evaluator

amb and fail

```
>> (amb 1 2)
1
>> (fail)
2
>> (fail)
```

Evaluator

amb and fail

```
>> (amb 1 2)
1
>> (fail)
2
>> (fail)
no-more-choices
```

Evaluator

amb and fail

```
>> (amb 1 2)
1
>> (fail)
2
>> (fail)
no-more-choices
>> (+ 1 (amb 10 20))
```

Evaluator

amb and fail

```
>> (amb 1 2)
1
>> (fail)
2
>> (fail)
no-more-choices
>> (+ 1 (amb 10 20))
11
```

Evaluator

amb and fail

```
>> (amb 1 2)
1
>> (fail)
2
>> (fail)
no-more-choices
>> (+ 1 (amb 10 20))
11
>> (fail)
```

Evaluator

amb and fail

```
>> (amb 1 2)
1
>> (fail)
2
>> (fail)
no-more-choices
>> (+ 1 (amb 10 20))
11
>> (fail)
21
```

Evaluator

amb and fail

```
>> (amb 1 2)
1
>> (fail)
2
>> (fail)
no-more-choices
>> (+ 1 (amb 10 20))
11
>> (fail)
21
>> (cons (amb 1 2) (amb 3 4))
```

Evaluator

amb and fail

```
>> (amb 1 2)
1
>> (fail)
2
>> (fail)
no-more-choices
>> (+ 1 (amb 10 20))
11
>> (fail)
21
>> (cons (amb 1 2) (amb 3 4))
(1 . 3)
```

Evaluator

amb and fail

```
>> (amb 1 2)
1
>> (fail)
2
>> (fail)
no-more-choices
>> (+ 1 (amb 10 20))
11
>> (fail)
21
>> (cons (amb 1 2) (amb 3 4))
(1 . 3)
>> (fail)
```

Evaluator

amb and fail

```
>> (amb 1 2)
1
>> (fail)
2
>> (fail)
no-more-choices
>> (+ 1 (amb 10 20))
11
>> (fail)
21
>> (cons (amb 1 2) (amb 3 4))
(1 . 3)
>> (fail)
(1 . 4)
```

Evaluator

amb and fail

```
>> (amb 1 2)
1
>> (fail)
2
>> (fail)
no-more-choices
>> (+ 1 (amb 10 20))
11
>> (fail)
21
>> (cons (amb 1 2) (amb 3 4))
(1 . 3)
>> (fail)
(1 . 4)
>> (fail)
```

Evaluator

amb and fail

```
>> (amb 1 2)
1
>> (fail)
2
>> (fail)
no-more-choices
>> (+ 1 (amb 10 20))
11
>> (fail)
21
>> (cons (amb 1 2) (amb 3 4))
(1 . 3)
>> (fail)
(1 . 4)
>> (fail)
(2 . 3)
```

Evaluator

amb and fail

```
>> (amb 1 2)
1
>> (fail)
2
>> (fail)
no-more-choices
>> (+ 1 (amb 10 20))
11
>> (fail)
21
>> (cons (amb 1 2) (amb 3 4))
(1 . 3)
>> (fail)
(1 . 4)
>> (fail)
(2 . 3)
>> (fail)
```

Evaluator

amb and fail

```
>> (amb 1 2)
1
>> (fail)
2
>> (fail)
no-more-choices
>> (+ 1 (amb 10 20))
11
>> (fail)
21
>> (cons (amb 1 2) (amb 3 4))
(1 . 3)
>> (fail)
(1 . 4)
>> (fail)
(2 . 3)
>> (fail)
(2 . 4)
```

Evaluator

amb and fail

```
>> (amb 1 2)
1
>> (fail)
2
>> (fail)
no-more-choices
>> (+ 1 (amb 10 20))
11
>> (fail)
21
>> (cons (amb 1 2) (amb 3 4))
(1 . 3)
>> (fail)
(1 . 4)
>> (fail)
(2 . 3)
>> (fail)
(2 . 4)
>> (fail)
```

Evaluator

amb and fail

```
>> (amb 1 2)
1
>> (fail)
2
>> (fail)
no-more-choices
>> (+ 1 (amb 10 20))
11
>> (fail)
21
>> (cons (amb 1 2) (amb 3 4))
(1 . 3)
>> (fail)
(1 . 4)
>> (fail)
(2 . 3)
>> (fail)
(2 . 4)
>> (fail)
no-more-choices
```

Evaluator

Implementation

- The ambiguous operator `amb` evaluates just one expression.
- Each of the remaining expressions is saved so that, if necessary, the entire computation can be repeated using its evaluation.

Saving Choices

```
(define choices (list))
```

Evaluator

Implementation

- The ambiguous operator `amb` evaluates just one expression.
- Each of the remaining expressions is saved so that, if necessary, the entire computation can be repeated using its evaluation.

Saving Choices

```
(define choices (list))

(define (add-choices! cs)
  (set! choices (append cs choices)))
```

Evaluator

Implementation

- The ambiguous operator `amb` evaluates just one expression.
- Each of the remaining expressions is saved so that, if necessary, the entire computation can be repeated using its evaluation.

Saving Choices

```
(define choices (list))

(define (add-choices! cs)
  (set! choices (append cs choices)))

(define (pop-choice!)
  (let ((choice (car choices)))
    (set! choices (cdr choices))
    choice))
```

Evaluator

amb and fail

```
>> (* 2 (+ 3 (amb 4 5 6)))
```

amb and fail

Evaluator

amb and fail

```
>> (* 2 (+ 3 (amb 4 5 6)))
```

amb and fail

```
>> (* 2 (+ 3 (call/cc (lambda (c)
                        (add-choices! (list (lambda () (c 5))
                                            (lambda () (c 6))))
                        4))))
```

Evaluator

amb and fail

```
>> (* 2 (+ 3 (amb 4 5 6)))
```

amb and fail

```
>> (* 2 (+ 3 (call/cc (lambda (c)
                        (add-choices! (list (lambda () (c 5))
                                            (lambda () (c 6))))
                        4)))))
```

Evaluator

amb and fail

```
>> (* 2 (+ 3 (amb 4 5 6)))
```

amb and fail

```
>> (* 2 (+ 3 (call/cc (lambda (c)
                        (add-choices! (list (lambda () (c 5))
                                            (lambda () (c 6))))
                       4))))
```

Evaluator

amb and fail

```
>> (* 2 (+ 3 (amb 4 5 6)))
14
```

amb and fail

```
>> (* 2 (+ 3 (call/cc (lambda (c)
                        (add-choices! (list (lambda () (c 5))
                                            (lambda () (c 6))))
                        4))))
14
```

Evaluator

amb and fail

```
>> (* 2 (+ 3 (amb 4 5 6)))
14
>> (fail)
```

amb and fail

```
>> (* 2 (+ 3 (call/cc (lambda (c)
                        (add-choices! (list (lambda () (c 5))
                                            (lambda () (c 6))))
                        4))))
14
>> ((pop-choice!))
```

Evaluator

amb and fail

```
>> (* 2 (+ 3 (amb 4 5 6)))
14
>> (fail)
16
```

amb and fail

```
>> (* 2 (+ 3 (call/cc (lambda (c)
                        (add-choices! (list (lambda () (c 5))
                                            (lambda () (c 6))))
                        4))))
14
>> ((pop-choice!))
16
```

Evaluator

amb and fail

```
>> (* 2 (+ 3 (amb 4 5 6)))
14
>> (fail)
16
>> (fail)
18
```

amb and fail

```
>> (* 2 (+ 3 (call/cc (lambda (c)
                        (add-choices! (list (lambda () (c 5))
                                            (lambda () (c 6))))
                        4))))
14
>> ((pop-choice!))
16
>> ((pop-choice!))
18
```

Evaluator

From

```
(amb e0 e1 ... en)
```

To

```
(call/cc
  (lambda (c)
    (add-choices!
      (list (lambda () (c e1))
            ...
            (lambda () (c en))))
    e0))
```

Evaluator

From

(amb e_0 e_1 ... e_n)

To

```
(call/cc
  (lambda (c)
    (add-choices!
      (list (lambda () (c e₁))
            ...
            (lambda () (c eₙ))))
    e₀))
```

The amb Macro

```
(mdef amb (e . es)
  `(call/cc (lambda (c)
              (add-choices!
                (list ,@(map (lambda (e) `(lambda () (c ,e)))
                             es)))
              ,e)))
```

Evaluator

Failing

```
(define (fail)
  (if (null? choices)
      (abort-top-level 'no-more-choices)
      ((pop-choice!))))
```

Failing

- When a world *fails* it means that the computation took some wrong choice and cannot continue.
- In order to proceed, take another choice and continue the computation from there.
- What should be the meaning of `(amb` *e*`)`?
- What should be the meaning of `(amb)`?

Evaluator

A Pythagorean Triple

```
(define (a-pythagorean-triple n)
  (let ((i (an-integer-between 1 n)))
    (let ((j (an-integer-between i n)))
      (let ((k (an-integer-between j n)))
        (require (= (+ (* i i) (* j j)) (* k k)))
        (list i j k)))))
```

Evaluator

A Pythagorean Triple

```
(define (a-pythagorean-triple n)
  (let ((i (an-integer-between 1 n)))
    (let ((j (an-integer-between i n)))
      (let ((k (an-integer-between j n)))
        (require (= (+ (* i i) (* j j)) (* k k)))
        (list i j k)))))
```

The `require` Function

- The function `require` wants its argument to be true.
- If it is not true, then the computation fails.

Evaluator

A Pythagorean Triple

```
(define (a-pythagorean-triple n)
  (let ((i (an-integer-between 1 n)))
    (let ((j (an-integer-between i n)))
      (let ((k (an-integer-between j n)))
        (require (= (+ (* i i) (* j j)) (* k k)))
        (list i j k)))))
```

The require Function

- The function require wants its argument to be true.
- If it is not true, then the computation fails.

The require Function

```
(define (require e)
  (or e (fail)))
```

Evaluator

Example

```
>> (a-pythagorean-triple 25)
```

Evaluator

Example

```
>> (a-pythagorean-triple 25)
(3 4 5)
```

Evaluator

Example

```
>> (a-pythagorean-triple 25)
(3 4 5)
>> (fail)
(5 12 13)
```

Evaluator

Example

```
>> (a-pythagorean-triple 25)
(3 4 5)
>> (fail)
(5 12 13)
>> (fail)
(6 8 10)
```

Evaluator

Example

```
>> (a-pythagorean-triple 25)
(3 4 5)
>> (fail)
(5 12 13)
>> (fail)
(6 8 10)
>> (fail)
(7 24 25)
>> (fail)
(8 15 17)
>> (fail)
(9 12 15)
>> (fail)
(12 16 20)
>> (fail)
(15 20 25)
>> (fail)
no-more-choices
```

Evaluator

Solving the puzzle: Using *backtracking*

```
(define (queens n i j board)
  (if (= i n)
      board
      (if (= j n)
          #f
          (if (attacked? i j board)
              (queens n i (+ j 1) board)
              (or (queens n (+ i 1) 0 (cons (cons i j) board))
                  (queens n i (+ j 1) board))))))

(define (solve-queens n)
  (queens n 0 0 (list)))
```

Evaluator

Solving the puzzle: Using *nondeterminism*

```
(define (queens n i j board)
  (if (= i n)
      board
      (let ((j (an-integer-between 0 (- n 1))))
        (require (not (attacked? i j board)))
        (queens n (+ i 1) 0 (cons (cons i j) board)))))




(define (solve-queens n)
  (queens n 0 0 (list)))
```

Evaluator

Finding Solutions for 5-Queens

```
>> (solve-queens 5)
```

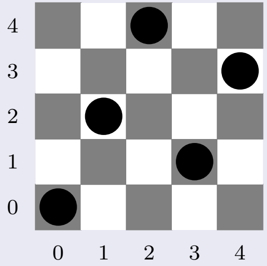
Board

Evaluator

Finding Solutions for 5-Queens

```
>> (solve-queens 5)
((4 . 3) (3 . 1) (2 . 4) (1 . 2) (0 . 0))
```

Board

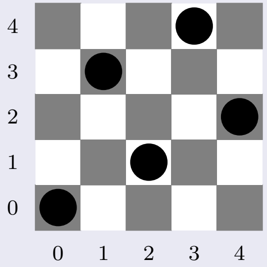


Evaluator

Finding Solutions for 5-Queens

```
>> (solve-queens 5)
((4 . 3) (3 . 1) (2 . 4) (1 . 2) (0 . 0))
>> (fail)
((4 . 2) (3 . 4) (2 . 1) (1 . 3) (0 . 0))
```

Board

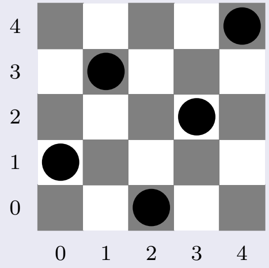


Evaluator

Finding Solutions for 5-Queens

```
>> (solve-queens 5)
((4 . 3) (3 . 1) (2 . 4) (1 . 2) (0 . 0))
>> (fail)
((4 . 2) (3 . 4) (2 . 1) (1 . 3) (0 . 0))
>> (fail)
((4 . 4) (3 . 2) (2 . 0) (1 . 3) (0 . 1))
```

Board

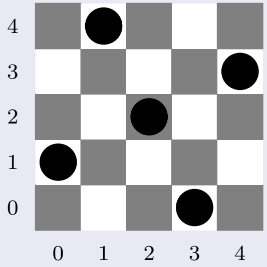


Evaluator

Finding Solutions for 5-Queens

```
>> (solve-queens 5)
((4 . 3) (3 . 1) (2 . 4) (1 . 2) (0 . 0))
>> (fail)
((4 . 2) (3 . 4) (2 . 1) (1 . 3) (0 . 0))
>> (fail)
((4 . 4) (3 . 2) (2 . 0) (1 . 3) (0 . 1))
>> (fail)
((4 . 3) (3 . 0) (2 . 2) (1 . 4) (0 . 1))
```

Board

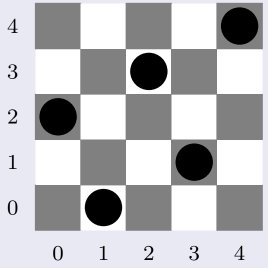


Evaluator

Finding Solutions for 5-Queens

```
>> (solve-queens 5)
((4 . 3) (3 . 1) (2 . 4) (1 . 2) (0 . 0))
>> (fail)
((4 . 2) (3 . 4) (2 . 1) (1 . 3) (0 . 0))
>> (fail)
((4 . 4) (3 . 2) (2 . 0) (1 . 3) (0 . 1))
>> (fail)
((4 . 3) (3 . 0) (2 . 2) (1 . 4) (0 . 1))
>> (fail)
((4 . 4) (3 . 1) (2 . 3) (1 . 0) (0 . 2))
```

Board

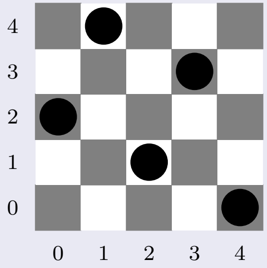


Evaluator

Finding Solutions for 5-Queens

```
>> (solve-queens 5)
((4 . 3) (3 . 1) (2 . 4) (1 . 2) (0 . 0))
>> (fail)
((4 . 2) (3 . 4) (2 . 1) (1 . 3) (0 . 0))
>> (fail)
((4 . 4) (3 . 2) (2 . 0) (1 . 3) (0 . 1))
>> (fail)
((4 . 3) (3 . 0) (2 . 2) (1 . 4) (0 . 1))
>> (fail)
((4 . 4) (3 . 1) (2 . 3) (1 . 0) (0 . 2))
>> (fail)
((4 . 0) (3 . 3) (2 . 1) (1 . 4) (0 . 2))
```

Board

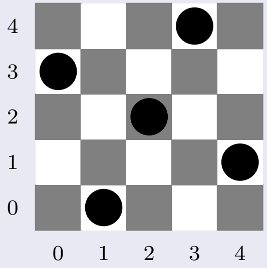


Evaluator

Finding Solutions for 5-Queens

```
>> (solve-queens 5)
((4 . 3) (3 . 1) (2 . 4) (1 . 2) (0 . 0))
>> (fail)
((4 . 2) (3 . 4) (2 . 1) (1 . 3) (0 . 0))
>> (fail)
((4 . 4) (3 . 2) (2 . 0) (1 . 3) (0 . 1))
>> (fail)
((4 . 3) (3 . 0) (2 . 2) (1 . 4) (0 . 1))
>> (fail)
((4 . 4) (3 . 1) (2 . 3) (1 . 0) (0 . 2))
>> (fail)
((4 . 0) (3 . 3) (2 . 1) (1 . 4) (0 . 2))
>> (fail)
((4 . 1) (3 . 4) (2 . 2) (1 . 0) (0 . 3))
```

Board

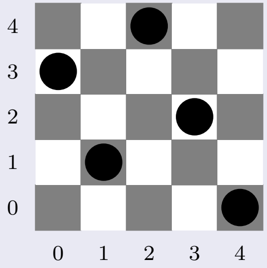


Evaluator

Finding Solutions for 5-Queens

```
>> (solve-queens 5)
((4 . 3) (3 . 1) (2 . 4) (1 . 2) (0 . 0))
>> (fail)
((4 . 2) (3 . 4) (2 . 1) (1 . 3) (0 . 0))
>> (fail)
((4 . 4) (3 . 2) (2 . 0) (1 . 3) (0 . 1))
>> (fail)
((4 . 3) (3 . 0) (2 . 2) (1 . 4) (0 . 1))
>> (fail)
((4 . 4) (3 . 1) (2 . 3) (1 . 0) (0 . 2))
>> (fail)
((4 . 0) (3 . 3) (2 . 1) (1 . 4) (0 . 2))
>> (fail)
((4 . 1) (3 . 4) (2 . 2) (1 . 0) (0 . 3))
>> (fail)
((4 . 0) (3 . 2) (2 . 4) (1 . 1) (0 . 3))
```

Board

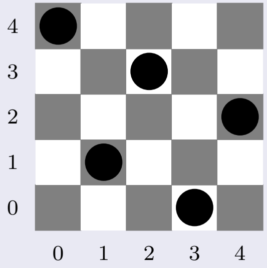


Evaluator

Finding Solutions for 5-Queens

```
>> (solve-queens 5)
((4 . 3) (3 . 1) (2 . 4) (1 . 2) (0 . 0))
>> (fail)
((4 . 2) (3 . 4) (2 . 1) (1 . 3) (0 . 0))
>> (fail)
((4 . 4) (3 . 2) (2 . 0) (1 . 3) (0 . 1))
>> (fail)
((4 . 3) (3 . 0) (2 . 2) (1 . 4) (0 . 1))
>> (fail)
((4 . 4) (3 . 1) (2 . 3) (1 . 0) (0 . 2))
>> (fail)
((4 . 0) (3 . 3) (2 . 1) (1 . 4) (0 . 2))
>> (fail)
((4 . 1) (3 . 4) (2 . 2) (1 . 0) (0 . 3))
>> (fail)
((4 . 0) (3 . 2) (2 . 4) (1 . 1) (0 . 3))
>> (fail)
((4 . 2) (3 . 0) (2 . 3) (1 . 1) (0 . 4))
```

Board

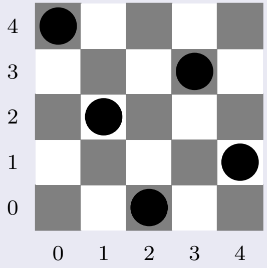


Evaluator

Finding Solutions for 5-Queens

```
>> (solve-queens 5)
((4 . 3) (3 . 1) (2 . 4) (1 . 2) (0 . 0))
>> (fail)
((4 . 2) (3 . 4) (2 . 1) (1 . 3) (0 . 0))
>> (fail)
((4 . 4) (3 . 2) (2 . 0) (1 . 3) (0 . 1))
>> (fail)
((4 . 3) (3 . 0) (2 . 2) (1 . 4) (0 . 1))
>> (fail)
((4 . 4) (3 . 1) (2 . 3) (1 . 0) (0 . 2))
>> (fail)
((4 . 0) (3 . 3) (2 . 1) (1 . 4) (0 . 2))
>> (fail)
((4 . 1) (3 . 4) (2 . 2) (1 . 0) (0 . 3))
>> (fail)
((4 . 0) (3 . 2) (2 . 4) (1 . 1) (0 . 3))
>> (fail)
((4 . 2) (3 . 0) (2 . 3) (1 . 1) (0 . 4))
>> (fail)
((4 . 1) (3 . 3) (2 . 0) (1 . 2) (0 . 4))
```

Board

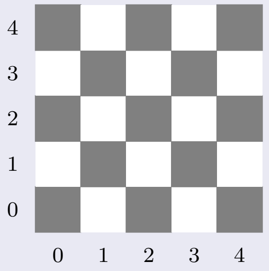


Evaluator

Finding Solutions for 5-Queens

```
>> (solve-queens 5)
((4 . 3) (3 . 1) (2 . 4) (1 . 2) (0 . 0))
>> (fail)
((4 . 2) (3 . 4) (2 . 1) (1 . 3) (0 . 0))
>> (fail)
((4 . 4) (3 . 2) (2 . 0) (1 . 3) (0 . 1))
>> (fail)
((4 . 3) (3 . 0) (2 . 2) (1 . 4) (0 . 1))
>> (fail)
((4 . 4) (3 . 1) (2 . 3) (1 . 0) (0 . 2))
>> (fail)
((4 . 0) (3 . 3) (2 . 1) (1 . 4) (0 . 2))
>> (fail)
((4 . 1) (3 . 4) (2 . 2) (1 . 0) (0 . 3))
>> (fail)
((4 . 0) (3 . 2) (2 . 4) (1 . 1) (0 . 3))
>> (fail)
((4 . 2) (3 . 0) (2 . 3) (1 . 1) (0 . 4))
>> (fail)
((4 . 1) (3 . 3) (2 . 0) (1 . 2) (0 . 4))
>> (fail)
no-more-choices
```

Board



Evaluator

Puzzle

- **A**drian, **B**ob, **C**harles, **D**avid, and **E**dward live on different floors of an apartment house that contains only five floors.
- **A**drian does not live on the top floor.
- **B**ob does not live on the bottom floor.
- **C**harles does not live on either the top or the bottom floor.
- **D**avid lives on a higher floor than does **B**ob.
- **E**dward does not live on a floor adjacent to **C**harles's.
- **C**harles does not live on a floor adjacent to **B**ob's.
- Where does everyone live?

Evaluator

The apartment-puzzle Function

```
(define (apartment-puzzle)
  (let ((a (amb 1 2 3 4 5))
        (b (amb 1 2 3 4 5))
        (c (amb 1 2 3 4 5))
        (d (amb 1 2 3 4 5))
        (e (amb 1 2 3 4 5)))
    (require (distinct? (list a b c d e)))
```

Puzzle

- **A**drian, **B**ob, **C**harles, **D**avid, and **E**dward live on **different** floors of an apartment house that contains only **five** floors.

Evaluator

The apartment-puzzle Function

```
(define (apartment-puzzle)
  (let ((a (amb 1 2 3 4))
        (b (amb 1 2 3 4 5))
        (c (amb 1 2 3 4 5))
        (d (amb 1 2 3 4 5))
        (e (amb 1 2 3 4 5)))
    (require (distinct? (list a b c d e)))
```

Puzzle

- **A**drian does not live on the top floor.

Evaluator

The apartment-puzzle Function

```
(define (apartment-puzzle)
  (let ((a (amb 1 2 3 4))
        (b (amb 2 3 4 5))
        (c (amb 1 2 3 4 5))
        (d (amb 1 2 3 4 5))
        (e (amb 1 2 3 4 5)))
    (require (distinct? (list a b c d e))
```

Puzzle

- **B**ob does not live on the bottom floor.

Evaluator

The apartment-puzzle Function

```
(define (apartment-puzzle)
  (let ((a (amb 1 2 3 4))
        (b (amb 2 3 4 5))
        (c (amb 2 3 4))
        (d (amb 1 2 3 4 5))
        (e (amb 1 2 3 4 5)))
    (require (distinct? (list a b c d e)))
```

Puzzle

- Charles does not live on either the top or the bottom floor.

Evaluator

The apartment-puzzle Function

```
(define (apartment-puzzle)
  (let ((a (amb 1 2 3 4))
        (b (amb 2 3 4 5))
        (c (amb 2 3 4))
        (d (amb 1 2 3 4 5))
        (e (amb 1 2 3 4 5)))
    (require (distinct? (list a b c d e)))
    (require (> d b))
```

Puzzle

- **D**avid lives on a higher floor than does **B**ob.

Evaluator

The apartment-puzzle Function

```
(define (apartment-puzzle)
  (let ((a (amb 1 2 3 4))
        (b (amb 2 3 4 5))
        (c (amb 2 3 4))
        (d (amb 1 2 3 4 5))
        (e (amb 1 2 3 4 5)))
    (require (distinct? (list a b c d e)))
    (require (> d b))
    (require (not (= (abs (- e c)) 1)))
```

Puzzle

- **E**dward does not live on a floor adjacent to **C**harles's.

Evaluator

The apartment-puzzle Function

```
(define (apartment-puzzle)
  (let ((a (amb 1 2 3 4))
        (b (amb 2 3 4 5))
        (c (amb 2 3 4))
        (d (amb 1 2 3 4 5))
        (e (amb 1 2 3 4 5)))
    (require (distinct? (list a b c d e)))
    (require (> d b))
    (require (not (= (abs (- e c)) 1)))
    (require (not (= (abs (- c b)) 1)))
```

Puzzle

- **C**harles does not live on a floor adjacent to **B**ob's.

Evaluator

The apartment-puzzle Function

```
(define (apartment-puzzle)
  (let ((a (amb 1 2 3 4))
        (b (amb 2 3 4 5))
        (c (amb 2 3 4))
        (d (amb 1 2 3 4 5))
        (e (amb 1 2 3 4 5)))
    (require (distinct? (list a b c d e)))
    (require (> d b))
    (require (not (= (abs (- e c)) 1)))
    (require (not (= (abs (- c b)) 1)))
    (list (list 'adrian a) (list 'bob b) (list 'charles c)
          (list 'david d) (list 'edward e))))
```

Puzzle

- Where does everyone live?

Evaluator

Solving the Puzzle

```
(define (apartment-puzzle)
  (let ((a (amb 1 2 3 4))
        (b (amb 2 3 4 5))
        (c (amb 2 3 4))
        (d (amb 1 2 3 4 5))
        (e (amb 1 2 3 4 5)))
    (require (distinct? (list a b c d e)))
    (require (> d b))
    (require (not (= (abs (- e c)) 1)))
    (require (not (= (abs (- c b)) 1)))
    (list (list 'adrian a) (list 'bob b) (list 'charles c)
          (list 'david d) (list 'emil e)))))
```

Evaluator

Solving the Puzzle

```
(define (apartment-puzzle)
  (let ((a (amb 1 2 3 4))
        (b (amb 2 3 4 5))
        (c (amb 2 3 4))
        (d (amb 1 2 3 4 5))
        (e (amb 1 2 3 4 5)))
    (require (distinct? (list a b c d e)))
    (require (> d b))
    (require (not (= (abs (- e c)) 1)))
    (require (not (= (abs (- c b)) 1)))
    (list (list 'adrian a) (list 'bob b) (list 'charles c)
          (list 'david d) (list 'emil e))))


>> (apartment-puzzle)
```

Evaluator

Solving the Puzzle

```
(define (apartment-puzzle)
  (let ((a (amb 1 2 3 4))
        (b (amb 2 3 4 5))
        (c (amb 2 3 4))
        (d (amb 1 2 3 4 5))
        (e (amb 1 2 3 4 5)))
    (require (distinct? (list a b c d e)))
    (require (> d b))
    (require (not (= (abs (- e c)) 1)))
    (require (not (= (abs (- c b)) 1)))
    (list (list 'adrian a) (list 'bob b) (list 'charles c)
          (list 'david d) (list 'emil e)))))
```

```
>> (apartment-puzzle)
((adrian 3) (bob 2) (charles 4) (david 5) (edward 1))
```

Evaluator

Solving the Puzzle

```
(define (apartment-puzzle)
  (let ((a (amb 1 2 3 4))
        (b (amb 2 3 4 5))
        (c (amb 2 3 4))
        (d (amb 1 2 3 4 5))
        (e (amb 1 2 3 4 5)))
    (require (distinct? (list a b c d e)))
    (require (> d b))
    (require (not (= (abs (- e c)) 1)))
    (require (not (= (abs (- c b)) 1)))
    (list (list 'adrian a) (list 'bob b) (list 'charles c)
          (list 'david d) (list 'emil e))))


>> (apartment-puzzle)
((adrian 3) (bob 2) (charles 4) (david 5) (edward 1))
>> (fail)
```

Evaluator

Solving the Puzzle

```
(define (apartment-puzzle)
  (let ((a (amb 1 2 3 4))
        (b (amb 2 3 4 5))
        (c (amb 2 3 4))
        (d (amb 1 2 3 4 5))
        (e (amb 1 2 3 4 5)))
    (require (distinct? (list a b c d e)))
    (require (> d b))
    (require (not (= (abs (- e c)) 1)))
    (require (not (= (abs (- c b)) 1)))
    (list (list 'adrian a) (list 'bob b) (list 'charles c)
          (list 'david d) (list 'emil e))))


>> (apartment-puzzle)
((adrian 3) (bob 2) (charles 4) (david 5) (edward 1))
>> (fail)
no-more-choices
```

Evaluator

Conclusions

Continuations are the basis for the implementation of specialized control operators:

- Lexical transfer of control (Lisp's `block&return-from`, Java's `break` and `continue`, etc)
- Dynamical transfer of control (Lisp's `catch&throw`)
- Exceptions (Lisp's `signal` and `error`, Java's `throw`)
- Exception Handling (Lisp's `handler-bind` and `handler-case` and `unwind-protect`, Java's `try-catch-finally`)
- Generators (Python, Ruby and C#'s `yield`)
- Nondeterminism (McCarthy's `amb`, Prolog, Icon)
- Coroutines (cooperative multithreading)
- ...

Harold Abelson and Gerald Jay Sussman.
Structure and Interpretation of Computer Programs.
The MIT Press, Cambridge, Massachusetts, 2nd edition, July 1996.