Lecture 7 — August 24, 2015

- Today:
  - Mutation and environments
  - The interpreter
- Readings:
  - Finish *SICP* 3.3, 4.1, 4.4

CSE130, Summer Session II

Benefits of assignment

- Similar to SICP 3.1.2

```
(define (estimate-pi trials)
  (set-rand-seed! 5)
  (sqrt (/ 6 (monte-carlo trials cesaro-test))))

(define (cesaro-test)
   (= (gcd (rand) (rand)) 1))

(define (monte-carlo trials experiment)
  (let iter ((remaining trials)
             (passed 0))
    (cond ((= remaining 0) (/ passed trials))
          ((experiment)
           (iter (- remaining 1) (+ passed 1)))
          (else
           (iter (- remaining 1) passed)))))
```

estimate-pi as purely functional

```
(define (estimate-pi trials)
  (sqrt (/ 6 (monte-carlo trials cesaro-test 5))))

(define (cesaro-test r)
  (let* ((r1 (rand-update r))
         (r2 (rand-update r1)))
    (cons (= (gcd r1 r2) 1) r2)))

(define (monte-carlo trials experiment seed)
  (let iter ((remaining trials)
             (passed 0)
             (r seed))
    (cond ((= remaining 0) (/ passed trials))
          (else
           (let* ((result (experiment r))
                  (test-result (car result))
                  (new-r (cdr result)))
             (if test-result
                 (iter (- remaining 1) (+ passed 1) new-r)
                 (iter (- remaining 1) passed new-r)))))))
```

Cost of assignment

- But order now matters! Compare:

```
(define (factorial n)
  (let ((product 1) (counter 1))
    (let iter ()
      (if (> counter n)
          product
          (begin
            (set! product (* counter product))
            (set! counter (+ counter 1))
            (iter))))))
```

- to:

```
(define (factorial n)
  (let iter ((product 1) (counter 1))
    (if (> counter n)
        product
        (iter (* counter product)
              (+ counter 1)))))
```

Local state

- Procedures (like the **`rand`** example before) can return different values on the same input.
- Exercise 3.1:

```
(define (make-accumulator value)
  (lambda (incr)
    (set! value (+ value incr))
    value))
```

```
> (define A (make-accumulator 5))
> (A 10)
15
> (A 10)
25
```

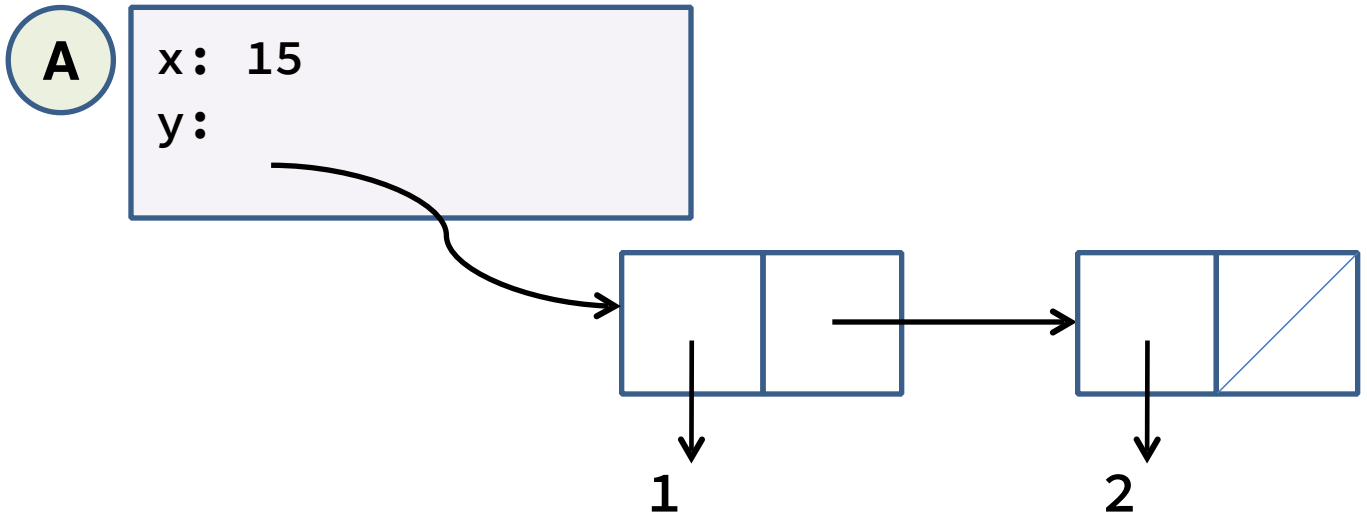
Environment Model

The EM is a precise, completely-mechanical description of a set of rules for determining the values associated with expressions in Scheme:

1. Name-rule – looking up the value of a variable
2. Define-rule – creating a new definition of a var
3. Set!-rule – changing the value of a variable
4. Lambda-rule – creating a procedure
5. Application – applying a procedure

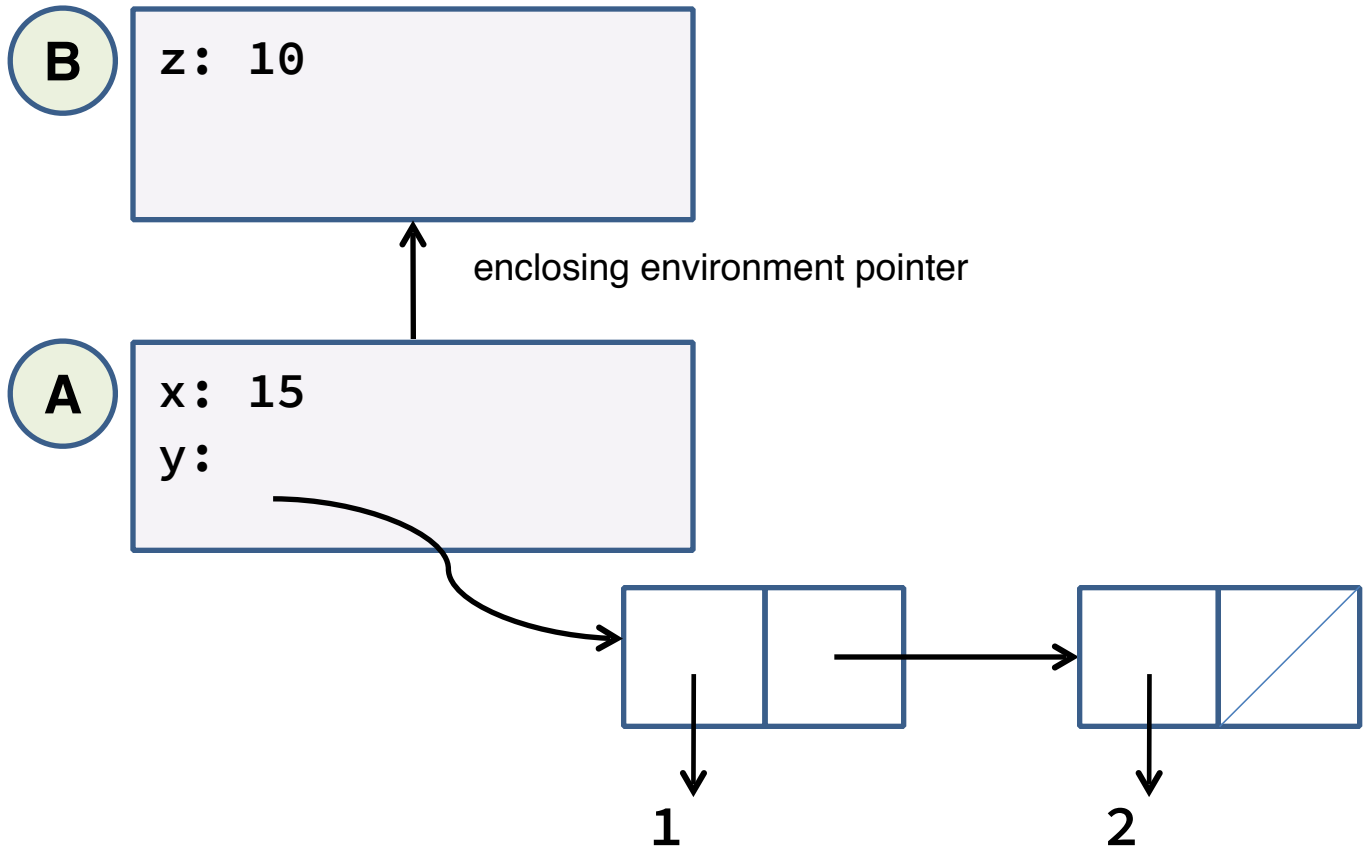
A frame is a table of bindings

- Binding: a pairing of a name and a value
- Example:
  - **x** is ***bound*** to **15** in frame A
  - **y** is bound to `(1 2)` in frame A
  - the value of the variable **x** in frame A is 15



An environment is a sequence of frames

- Example:
  - Environment E1 consists of frames A and B
  - Environment E2 consists of frame B only
- A frame may be shared by multiple environments

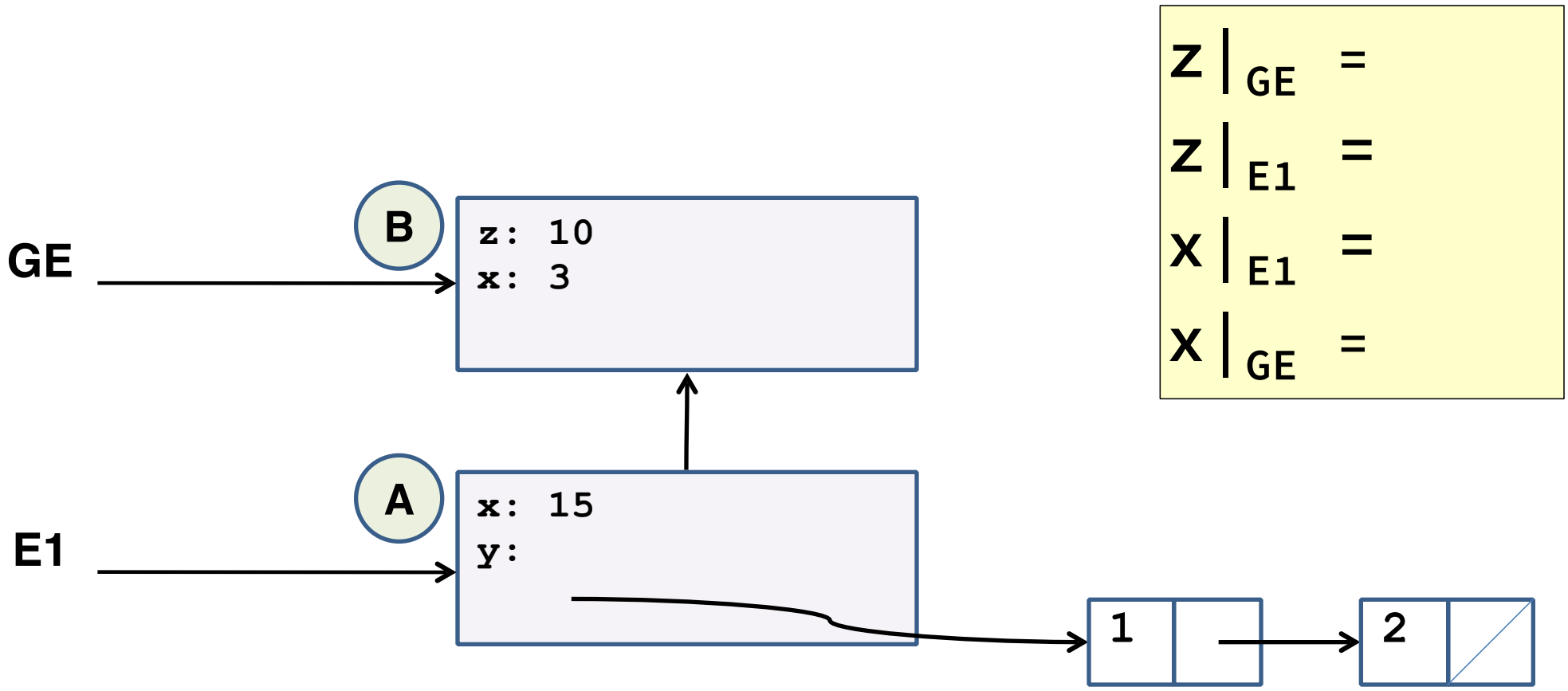


Evaluation in the EM

- All evaluation occurs in an environment
  - The current environment changes when the interpreter applies a procedure
- The top environment is called the global environment (GE)
  - Only the GE has no enclosing environment
  - Holds bindings for primitives
- To evaluate a combination:
  - Evaluate the sub-expressions in the current environment
  - Apply the value of the first to the values of the rest

The name rule

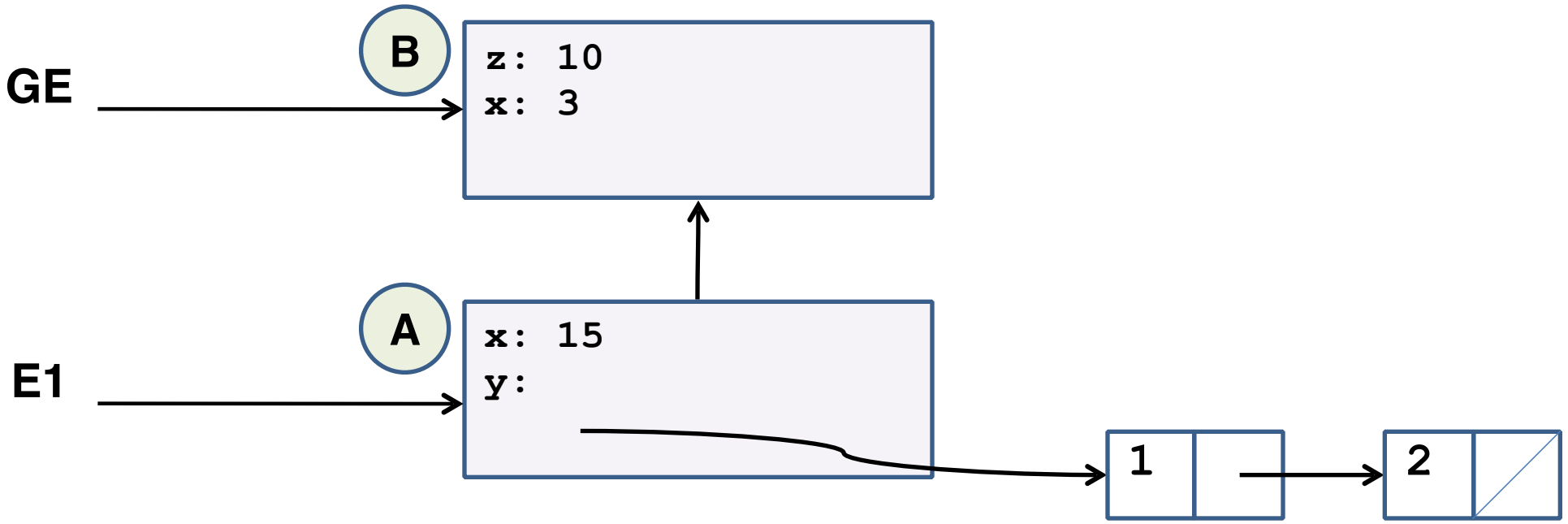
- A name X evaluated in environment E gives the value of X in the **first frame** of E where X is bound
- In E1, the binding of x in frame A ***shadows*** the binding of x in B



The **define** rule

- A set! of variable X evaluated in environment E changes the binding of X in the first frame of E where X is bound

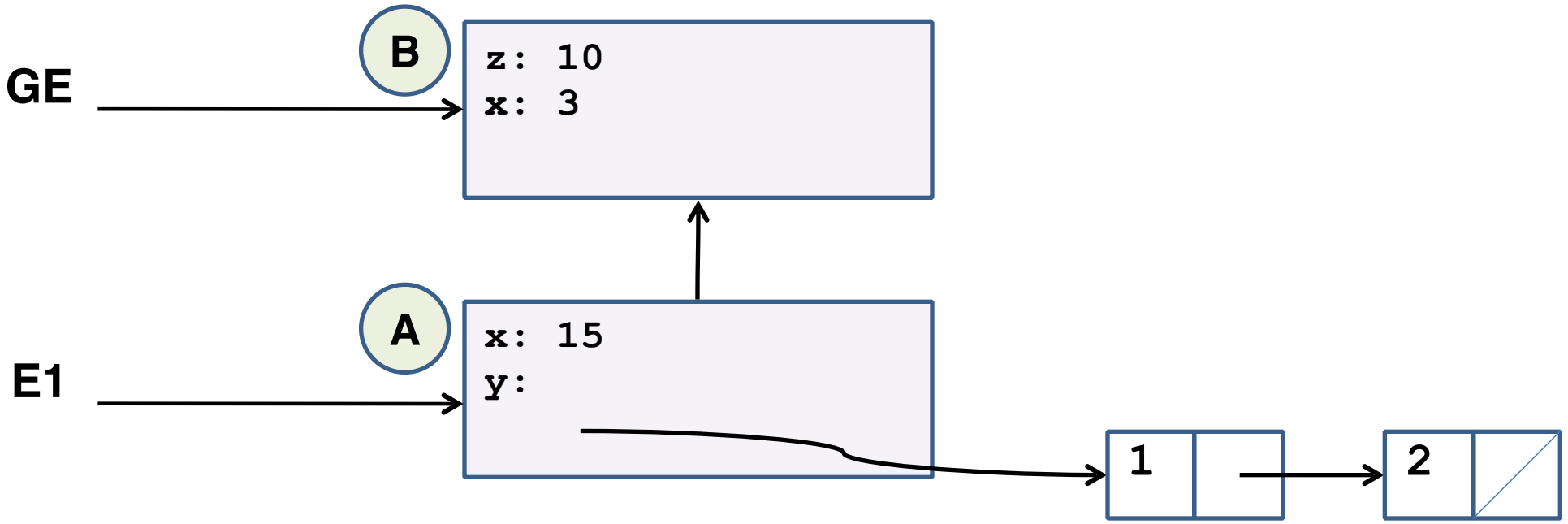
`(set! z 20)`$|_{GE}$ `(set! z 25)`$|_{E1}$



The **`set!`** rule

- A **`define`** evaluated in environment E creates or replaces a binding in the first frame of E.

`(define z 20)`$|_{GE}$ `(define z 25)`$|_{E1}$



The `lambda` rule

- Procedures are created by lambda expressions
- The value of the procedure is a pair:
  - A pointer to the code of the procedure, including its parameters list and the procedure's body; and
  - A pointer to the environment in which the lambda expression was evaluated

The apply rule

To apply a compound procedure P to arguments:

1. Create a new frame A
2. Make A into an environment E:
3. A's enclosing environment pointer goes to the same frame as the environment pointer of P
4. In A, bind the parameters of P to the argument values
5. Evaluate the body of P with E as the current environment

Derived expressions: **`let`**

- The code:

```
(let ((name1 val1)
      (name2 val2))
  body)
```

- is equivalent to:

```
((lambda (name1 name2) body)
 val1 val2)
```

- Derivation: So, if we've already implemented lambdas and applications, our interpreter could handle **`let`** by making the same transformation!

Converting let

```
private static Value evaluateLet(LinkedList<Value> body,
                                 Environment environment) {
  // (let ([n e] [m f]) body) => ((lambda (n m) body) e f)
  matchSymbol(body, "let");

  List<LexemeDatum.SymbolDatum> names = new LinkedList<>();
  List<Datum> initializers = new LinkedList<>();
  getLetVariablesAndValues(next(body, CompoundDatum.class),
                           names, initializers);
  Datum letBody = expressionListToBody(body);

  Datum lambdaText = CompoundDatum.newList(LexemeDatum.newSymbol(
    "lambda"), CompoundDatum.newList(names), letBody);

  List<Datum> applTextList = new LinkedList<>();
  appTextList.add(lambdaText);
  appTextList.addAll(initializers);
  return evaluate(CompoundDatum.newList(appTextList), environment);
}
```

Representing environments

```
public interface Environment {
  void defineVariable(String variable, Value value);
  Value lookupVariable(String variable);
  void setVariable(String variable, Value newValue);
}
```

```
public class NullEnvironment implements Environment {
  public void defineVariable(String variable, Value value) {
    throw Problem.unboundVariable(variable);
  }

  public Value lookupVariable(String variable) {
    throw Problem.unboundVariable(variable);
  }

  public void setVariable(String variable, Value newValue) {
    throw Problem.unboundVariable(variable);
  }
}
```

```
public class ExtendedEnvironment implements Environment {
  private Map<String, Value> frame;
  private  Environment base;

  ExtendedEnvironment(Environment base) {
    this.frame = new HashMap<>()
    this.base = base;
  }

  public void defineVariable(String variable, Value value) {
    frame.put(variable, value);
  }

  public Value lookupVariable(String variable) {
    if (frame.containsKey(variable)) {
      return frame.get(variable);
    }
    return base.lookupVariable(variable);
  }

  public void setVariable(String variable, Value newValue) {
    if (frame.containsKey(variable)) {
      frame.put(variable, newValue);
    } else {
      base.setVariable(variable, newValue);
    }
  }
}
```

Environment factories

- Both classes are just implemented in Environment.java:

```
public abstract class Environment {
  /* Creates an empty environment where new bindings can be added to it. */
  public static Environment newEmptyEnvironment() {
    return new NullEnvironment().extend(
        Collections.emptyList(), Collections.emptyList());
  }

  /* Creates a new environment by extending the given one. Any bindings not found in the
extended environment will be checked in the base environment. */
  public Environment extend(List<String> varNames,
                            List<Value> values) {
    Environment extended = new ExtendedEnvironment(this);
    if (varNames.size() != values.size()) { /* throw */ }
    for (int i = 0; i < varNames.size(); ++i) {
      extended.defineVariable(
          varNames.get(i), values.get(i));
    }
    return extended;
  }
}
```

The name rule

```
private static Value evaluate(Datum sExpr,
                              Environment environment) {
  if (sExpr.isSymbol()) {
    return environment.lookupVariable(sExpr.toString());
  } else if (sExpr.isEmptyList()) {
    throw Problem.invalidExpression("Application must …");
  } else if (sExpr.isPair()) {
    return evaluateNonEmptyListForm(sExpr, environment);
  } else {
    // then just assume it's self-evaluating
    return sExpr;
  }
}
```

Considerations

- Since we are not implementing a Scheme interpreter in Scheme, the question presents itself: Should our interpreter use Scheme lists or Java lists?
- Approach One: Convert s-expressions to Java Lists and back again
  - Pro: Offers reuse of Java libraries
  - The **`Deque`** interface of **`LinkedList`** is particularly helpful
- Approach Two: Parse to Java Lists in the first place
  - Con: coupling to specific implementation strategy
- Approach Three: Maintain the s-expression representation throughout
  - Con: reinventing the wheel