Principles

Practice
Principles

Practice
PRINCIPLES

1 2 3 4

Industrial co-op

PRACTICE
Principles

Practice

Industrial co-op
Principles

Practice
Principles

How to Design Classes
Data: Structure and Organization

Matthias Felleisen
Matthew Flatt
Robert Bruce Findler
Kathryn E. Gray
Shriram Krishnamurthi
Viera K. Proulx

Practice
EDUCATIONAL PEARL

The Structure and Interpretation of the Computer Science Curriculum

Matthias Felleisen, Northeastern University, Boston, MA, USA
Robert Bruce Findler, University of Chicago, Chicago, IL, USA
Matthew Flatt, University of Utah, Salt Lake City, UT, USA
Shriram Krishnamurthi, Brown University, Providence, RI, USA
Email: {matthias,robby,mflatt,shriram}@plt-scheme.org

Abstract

Twenty years ago Abelson and Sussman’s Structure and Interpretation of Computer Programs radically changed the intellectual landscape of introductory computing courses. Instead of teaching some currently fashionable programming language, it employed Scheme and functional programming to teach important ideas. Introductory courses based on the book showed up around the world and made Scheme and functional programming popular. Unfortunately, these courses quickly disappeared again due to shortcomings of the book and the whimsies of Scheme. Worse, the experiment left people with a bad impression of Scheme and functional programming in general.

In this pearl, we propose an alternative role for functional programming in the first-year curriculum. Specifically, we present a framework for discussing the first-year curriculum and, based on it, the design rationale for our book and course, dubbed How to Design Programs. The approach emphasizes the systematic design of programs. Experience shows that it works extremely well as a preparation for a course on object-oriented programming.

1 History and critique

The publication of Abelson and Sussman’s Structure and Interpretation of Computer Programs (sicp) (Abelson et al., 1985) revolutionized the landscape of the introductory computing curriculum in the 1980s. Most importantly, the book liberated the introductory course from the tyranny of syntax. Instead of arranging a course around the syntax of a currently fashionable programming language, sicp focused the first course on the study of important ideas in computing: functional abstraction, data abstraction, streams, data-directed programming, implementation of message-passing objects, interpreters, compilers, and register machines.

Over a short period, many universities in the US and around the world switched their first course to sicp and Scheme. The book became a major bestseller for MIT Press.¹ Along with sicp, the Scheme programming language (Sussman & Steele Jr.,

¹ According to Bob Prior (editor at MIT Press), sicp sold 45,000 copies in its first five years [personal communication, 9 June 2003].
EDUCATIONAL PEARL

The Structure and Interpretation of the Computer Science Curriculum

Matthias Felleisen, Northeastern University, Boston, MA, USA
Robert Bruce Findler, University of Chicago, Chicago, IL, USA
Matthew Flatt, University of Utah, Salt Lake City, UT, USA
Shriram Krishnamurthi, Brown University, Providence, RI, USA

Email: {matthias,robby,mflatt,shriram}@plt-scheme.org

Abstract

Twenty years ago Abelson and Sussman’s Structure and Interpretation of Computer Programs radically changed the intellectual landscape of introductory computing courses. Instead of teaching some currently fashionable programming language, it employed Scheme and functional programming to teach important ideas. Introductory courses based on the book showed up around the world and made Scheme and functional programming popular. Unfortunately, these courses quickly disappeared again due to shortcomings of the book and the whimsies of Scheme. Worse, the experiment left people with a bad impression of Scheme and functional programming in general.

In this pearl, we propose an alternative role for functional programming in the first-year curriculum. Specifically, we present a framework for discussing the first-year curriculum and, based on it, the design rationale for our book and course, dubbed How to Design Programs. The approach emphasizes the systematic design of programs. Experience shows that it works extremely well as a preparation for a course on object-oriented programming.

1 History and critique

The publication of Abelson and Sussman’s Structure and Interpretation of Computer Programs (sicp) (Abelson et al., 1985) revolutionized the landscape of the introductory computing curriculum in the 1980s. Most importantly, the book liberated the introductory course from the tyranny of syntax. Instead of arranging a course around the syntax of a currently fashionable programming language, sicp focused the first course on the study of important ideas in computing: functional abstraction, data abstraction, streams, data-directed programming, implementation of message-passing objects, interpreters, compilers, and register machines.

Over a short period, many universities in the US and around the world switched their first course to sicp and Scheme. The book became a major bestseller for MIT Press. Along with sicp, the Scheme programming language (Sussman & Steele Jr., 1986, 1990)
1. Introduce only those language constructs that are necessary to
   teach programming principles
EDUCATIONAL PEARL

The Structure and Interpretation of the Computer Science Curriculum

Matthias Felleisen, Northeastern University, Boston, MA, USA
Robert Bruce Findler, University of Chicago, Chicago, IL, USA
Matthew Flatt, University of Utah, Salt Lake City, UT, USA
Shriram Krishnamurthi, Brown University, Providence, RI, USA
Email: {matthias,robby,mflatt,shriram}@plt-scheme.org

Abstract

Twenty years ago Abelson and Sussman's Structure and Interpretation of Computer Programs radically changed the intellectual landscape of introductory computing courses. Instead of teaching some currently fashionable programming language, it employed Scheme and functional programming to teach important ideas. Introductory courses based on the book showed up around the world and made Scheme and functional programming popular. Unfortunately, these courses quickly disappeared again due to short coming of the book and the whimsies of Scheme. Worse, the experiment left people with habit of implementing ideas.

In this pearl, we propose an alternative role for functional programming in the first-year curriculum. Specifically, we present a framework for discussing the first-year curriculum and, based on it, the design rationale for our book and course, dubbed How to Design Programs. The approach emphasizes systematic design of programs. Experience shows that it works extremely well as a preparation for a course on object-oriented programming.

History and critique

The publication of Abelson and Sussman's Structure and Interpretation of Computer Programs (sicp) (Abelson et al., 1985) revolutionized the landscape of the introductory computing curriculum in the 1980s. Most importantly, the book liberated the introductory course from the tyranny of syntax. Instead of arranging a course around the syntax of a currently fashionable programming language, sicp focused the first course on the study of important ideas in computing: functional abstraction, data abstraction, streams, data-directed programming, implementation of message-passing objects, interpreters, compilers, and register machines.

Over a short period, many universities in the US and around the world switched their first course to sicp and Scheme. The book became a major bestseller for MIT Press.1 Along with sicp, the Scheme programming language (Sussman & Steele Jr.,

---

1 Introduce only those language constructs that are necessary to teach programming principles

2. Choose a language with as few language constructs as possible, and one in which they can be introduced one at a time
#lang htdp/bsl
(require 2htdp/image)
(require 2htdp/universe)

; Use the rocket key to insert the rocket here.

(define ROCKET (define WIDTH 100)
(define HEIGHT 300)
(define MT-SCENE (empty-scene WIDTH HEIGHT))
; A World is a Number.
; Interp: distance from the ground in AU.
; render : World -> Scene
(check-expect (render 0)
  (place-image ROCKET (/ WIDTH 2) HEIGHT MT-SCENE))

(define (render h)
  (place-image ROCKET
  (/ WIDTH 2)
  (- HEIGHT h)
  MT-SCENE))

; next : World -> World
(check-expect (next 0) 7)
(define (next h)
  (+ h 7))

(big-bang 0
  (on-tick next)
  (to-draw render))
#lang htdp/bsl
(require 2http/image)
(require 2http/universe)

; Use the rocket key to insert the rocket here.

(define ROCKET)
(define WIDTH 100)
(define HEIGHT 300)
(define MT-SCENE (empty-scene WIDTH HEIGHT))
; A World is a Number.
; Interp: distance from the ground in AU.
; render: World -> Scene
(check-expect (render 0)
  (place-image ROCKET (/ WIDTH 2) HEIGHT MT-SCENE))

(define (render h)
  (place-image ROCKET
               (/ WIDTH 2)
               (- HEIGHT h)
               MT-SCENE))

; next : World -> World
(check-expect (next 0) 7)
(define (next h)
  (+ h 7))

(big-bang 0
  (on-tick next)
  (to-draw render))
regular, minimal syntax
regular, minimal syntax

complicated irregular syntax
regular, minimal syntax

untyped

complicated irregular syntax
regular, minimal syntax
untyped

complicated irregular syntax
typed
regular, minimal syntax
untyped
pedagogical environment

complicated irregular syntax
typed
regular, minimal syntax
untyped
pedagogical environment

complicated irregular syntax
typed
industrial environment
regular, minimal syntax  complicated irregular syntax
untyped  typed
pedagogical environment  industrial environment
mathematical numbers
regular, minimal syntax
untyped
pedagogical environment
mathematical numbers
complicated irregular syntax
typed
industrial environment
machine numbers
regular, minimal syntax

complicated irregular syntax

untyped

typed

pedagogical environment

industrial environment

mathematical numbers

machine numbers

images as values
regular, minimal syntax
untyped
pedagogical environment
mathematical numbers
images as values
complicated irregular syntax
typed
industrial environment
machine numbers
?
regular, minimal syntax
untyped
pedagogical environment
mathematical numbers
images as values
interaction

complicated irregular syntax
typed
industrial environment
machine numbers
?
regular, minimal syntax

untyped

pedagogical environment

mathematical numbers

images as values

interaction

complicated irregular syntax

typed

industrial environment

machine numbers

? 

interaction
regular, minimal syntax
untyped
pedagogical environment
mathematical numbers
images as values
interaction
functions and structures
complicated irregular syntax
typed
industrial environment
machine numbers
?
compilation
regular, minimal syntax
untyped
pedagogical environment
mathematical numbers
images as values
interaction
functions and structures
objects
1. Introduce only those language constructs that are necessary to teach programming principles

2. Choose a language with as few language constructs as possible, and one in which they can be introduced one at a time
1. Introduce only those language constructs that are necessary to teach programming principles

2. Choose a language with as few language constructs as possible, and one in which they can be introduced one at a time

```
#lang class

;; A Tree is one of:
;; - (new leaf Number)
;; - (new node Tree Number Tree)
;; and implements
;; sum : Tree -> Number
;; sums the elements of this tree

(define-class leaf
  (fields v)
  (define (sum) (this . v))

(define-class node
  (fields left v right)
  (define (sum)
    (+ (this . left . sum)
      (this . v)
      (this . right . sum))

(check-expect ((new leaf 7) . sum) 7)
(check-expect ((new node (new leaf 1) 5 (new node (new leaf 0) 10 (new leaf 0)))) . sum) 16)
```

Figure 1: Binary tree sum in Beginning Student and in the Class language
#lang class/0
(define-class posn (fields x y))
(new posn 3 4)
(send (new posn 3 4) x) ;=> 3
(send (new posn 3 4) y) ;=> 4
#lang class/1
(define-class posn (fields x y))
(new posn 3 4)
((new posn 3 4) . x) ;=> 3
((new posn 3 4) . y) ;=> 4
#lang class/1

(define-class posn (fields x y)
  ;; Posn -> Number
  ;; Distance between this posn and that posn
  (check-expect ((new posn 0 0) . dist (new posn 3 4)) 5)
  (define (dist that)
    (sqrt (+ (sqr (- (this . x) (that . x)))
             (sqr (- (this . y) (that . y)))))))

  ;; -> Number
  ;; Distance of this posn from the origin
  (check-expect ((new posn 0 0) . dist-origin) 0)
  (check-expect ((new posn 3 4) . dist-origin) 5)
  (define (dist-origin)
    (this . dist (new posn 0 0))))

objects
#lang class/1

;; A Posn is a (new posn Number Number),
;; which represents a point on the Cartesian plane
(define-class posn (fields x y)
  ;; Posn -> Number
  ;; Distance between this posn and that posn
  (check-expect ((new posn 0 0) . dist (new posn 3 4)) 5)
  (define (dist that)
  (sqrt (+ (sqr (- (this . x) (that . x)))
           (sqr (- (this . y) (that . y))))))
  ;; -> Number
  ;; Distance of this posn from the origin
  (check-expect ((new posn 0 0) . dist-origin) 0)
  (check-expect ((new posn 3 4) . dist-origin) 5)
  (define (dist-origin)
  (this . dist (new posn 0 0)))

Welcome to DrRacket, version 5.3.3.5--2013-02-25(08800641/d) [3m].
Language: class/1; memory limit: 128 MB.
All 3 tests passed!
> ((new posn 6 10) . dist-origin)
11.661903789690601

Background expansion finished
;; A Tree is one of:
;; - (make-leaf Number)
;; - (make-node Tree Number Tree)
(define-struct leaf (v))
(define-struct node (left v right))

;; sum : Tree -> Number
;; sums the elements of the given tree
(define (sum a-tree)
  (cond
   [(leaf? a-tree) (leaf-v a-tree)]
   [else
     (+ (sum (node-left a-tree))
        (node-v a-tree)
        (sum (node-right a-tree)))]))

#lang class/1
;; A Tree is one of:
;; - (new leaf Number)
;; - (new node Tree Number Tree)
;; and implements
;; sum : -> Number
;; sums the elements of this tree
(define-class leaf
  (fields v)
  (define (sum) (this . v)))

(define-class node
  (fields left v right)
  (define (sum)
    (+ (this . left . sum)
      (this . v)
      (this . right . sum)))

objects
(require 2htdp/image 2htdp/universe)

;;; A World is a Number
(define-struct world (n))

;;; tock : World -> World
(define (tock w)
  (make-world (+ (world-n w) 1)))

;;; draw : World -> Image
(define (draw w)
  (rotate (modulo (world-n w) 360))
)

;;; KeyEvent World -> World
(define (press k w) (make-world 0))

(big-bang (make-world 0)
  [on-tick tock]
  [on-draw draw]
  [on-key press])
5. included with each method definition, following the principles of the design recipe studied in the first semester. In fact, the `check-expect` mechanism works exactly as it did before.

Methods can be defined to consume any number of arguments, but they are implicitly parameterized over this, the object that was sent the message.

### 3.2 Where did the `cond` go?

Unions, and recursive unions in particular, are a fundamental kind of data definition that students are well-versed in from the previous semester. A fundamental early lesson is how to represent (recursive) unions using classes and how to write recursive methods. As an example, figure 1 defines binary trees of numbers (an archetypal recursive union data definition) using the BSL language and the Class language.

```scheme
#lang bsl
;; A Tree is one of:
;; - (make-leaf Number)
;; - (make-node Tree Number Tree)
(define-struct leaf (v))
(define-struct node (left v right))
;; sum : Tree -> Number
;; sums the elements of the given tree
(define (sum a-tree)
  (cond [(leaf? a-tree) (leaf-v a-tree)]
        [else (+ (sum (node-left a-tree))
                (node-v a-tree)
                (sum (node-right a-tree)))]))
(check-expect (sum (make-leaf 7)) 7)
(check-expect (sum (make-node (make-leaf 1) 5
                              (make-node (make-leaf 0) 10
                                          (make-leaf 0)))) 16)
```

```
#lang class/1
;; A Tree is one of:
;; - (new leaf Number)
;; - (new node Tree Number Tree)
;; and implements
;; sum : -> Number
;; sums the elements of this tree
(define-class leaf
  (fields v)
  (define (sum) (this . v)))
(define-class node
  (fields left v right)
  (define (sum)
    (+ (this . left . sum)
       (this . v)
       (this . right . sum))))
(check-expect ((new leaf 7) . sum) 7)
(check-expect ((new node (new leaf 1) 5
                          (new node (new leaf 0) 10
                                      (new leaf 0)))) . sum) 16)
```

Figure 1: Binary tree sum in Beginning Student and in the Class language

The structure of this data definition is analogous to the approach of the previous semester but this example brings to light an important difference with the functional approach. The method for computing the sum of the empty tree is defined in the `leaf` class, while the method for computing the sum of a node...
Methods can be defined to consume any number of arguments, but they are implicitly parameterized over this, the object that was sent the message.

3.2 Where did the cond go?

Unions, and recursive unions in particular, are a fundamental kind of data definition that students are well-versed in from the previous semester. A fundamental early lesson is how to represent (recursive) unions using classes and how to write recursive methods. As an example, figure 1 defines binary trees of numbers (an archetypal recursive union data definition) using the BSL language and the Class language.

```scheme
#lang bsl
;; A Tree is one of:
;; - (make-leaf Number)
;; - (make-node Tree Number Tree)
(define-struct leaf (v))
(define-struct node (left v right))

;; sum : Tree -> Number
;; sums the elements of the given tree
(define (sum a-tree)
  (cond [(leaf? a-tree) (leaf-v a-tree)]
        [else (+ (sum (node-left a-tree))
                  (node-v a-tree)
                  (sum (node-right a-tree)))]))
(check-expect (sum (make-leaf 7)) 7)
(check-expect (sum (make-node (make-leaf 1) 5
                             (make-node (make-leaf 0) 10
                                       (make-leaf 0)))) 16)
```

```scheme
#lang class/1
;; A Tree is one of:
;; - (new leaf Number)
;; - (new node Tree Number Tree)
;; and implements
;; sum : -> Number
;; sums the elements of this tree
(define-class leaf
  (fields v)
  (define (sum) (this . v)))
(define-class node
  (fields left v right)
  (define (sum)
    (+ (this . left . sum)
       (this . v)
       (this . right . sum))))
(check-expect ((new leaf 7) . sum) 7)
(check-expect ((new node (new leaf 1) 5
                         (new node (new leaf 0) 10
                                   (new leaf 0)))) . sum) 16)
```

Figure 1: Binary tree sum in Beginning Student and in the Class language

The structure of this data definition is analogous to the approach of the previous semester but this example brings to light an important difference with the functional approach. The method for computing the sum of the empty tree is defined in the leaf class, while the method for computing the sum of a node...
Methods can be defined to consume any number of arguments, but they are implicitly parameterized over this, the object that was sent the message.

3.2 Where did the cond go?

Unions, and recursive unions in particular, are a fundamental kind of data definition that students are well-versed in from the previous semester. A fundamental early lesson is how to represent (recursive) unions using classes and how to write recursive methods. As an example, figure 1 defines binary trees of numbers (an archetypal recursive union data definition) using the BSL language and the Class language.

```
#lang bsl
;; A Tree is one of:
;; - (make-leaf Number)
;; - (make-node Tree Number Tree)
(define-struct leaf (v))
(define-struct node (left v right))
;; sum : Tree -> Number
;; sums the elements of the given tree
(define (sum a-tree)
  (cond [(leaf? a-tree) (leaf-v a-tree)]
        [else (+ (sum (node-left a-tree))
                 (node-v a-tree)
                 (sum (node-right a-tree)))]))
(check-expect (sum (make-leaf 7)) 7)
(check-expect (sum (make-node (make-leaf 1) 5
                              (make-node (make-leaf 0)
                                         10
                                         (make-leaf 0)))) 16)
```

```
#lang class
;; A Tree is one of:
;; - (new leaf Number)
;; - (new node Tree Number Tree)
(define-class leaf (fields v)
  (define (sum) (this . v)))
(define-class node (fields left v right)
  (define (sum)
    (+ (this . left . sum)
       (this . v)
       (this . right .sum))))
(check-expect ((new leaf 7) . sum) 7)
(check-expect ((new node (new leaf 1) 5
                         (new node (new leaf 0) 10
                                   (new leaf 0)))) . sum) 16)
```

Figure 1: Binary tree sum in Beginning Student and in the Class language

The structure of this data definition is analogous to the approach of the previous semester but this example brings to light an important difference with the functional approach. The method for computing the sum of the empty tree is defined in the leaf class, while the method for computing the sum of a node object's sum.
Methods can be defined to consume any number of arguments, but they are implicitly parameterized over this, the object that was sent the message.

### 3.2 Where did the cond go?

Unions, and recursive unions in particular, are a fundamental kind of data definition that students are well-versed in from the previous semester. A fundamental early lesson is how to represent (recursive) unions using classes and how to write recursive methods. As an example, figure 1 defines binary trees of numbers (an archetypal recursive union data definition) using the BSL language and the Class language.

```
#lang bsl
;; A Tree is one of:
;; - (make-leaf Number)
;; - (make-node Tree Number Tree)
(define-struct leaf (v))
(define-struct node (left v right))

;; sum : Tree -> Number
;; sums the elements of the given tree
(define (sum a-tree)
  (cond [(leaf? a-tree) (leaf-v a-tree)]
        [else (+ (sum (node-left a-tree))
                 (node-v a-tree)
                 (sum (node-right a-tree)))]))
(check-expect (sum (make-leaf 7)) 7)
(check-expect (sum (make-node (make-leaf 1) 5 (make-node (make-leaf 0) 10 (make-leaf 0)))) 16)
```

```
#lang class/1
;; A Tree is one of:
;; - (new leaf Number)
;; - (new node Tree Number Tree)
;; and implements
;; sum : -> Number
;; sums the elements of this tree
(define-class leaf
  (fields v)
  (define (sum) (this . v)))
(define-class node
  (fields left v right)
  (define (sum) (+ (this . left . sum) (this . v) (this . right . sum))))
(check-expect ((new leaf 7) . sum) 7)
(check-expect ((new node (new leaf 1) 5 (new node (new leaf 0) 10 (new leaf 0)))) . sum) 16)
```

Figure 1: Binary tree sum in Beginning Student and in the Class language

The structure of this data definition is analogous to the approach of the previous semester but this example brings to light an important difference with the functional approach. The method for computing the sum of the empty tree is defined in the leaf class, while the method for computing the sum of a node...
Methods can be defined to consume any number of arguments, but they are implicitly parameterized over this, the object that was sent the message.  

3.2 Where did the cond go?  

Unions, and recursive unions in particular, are a fundamental kind of data definition that students are well-versed in from the previous semester. A fundamental early lesson is how to represent (recursive) unions using classes and how to write recursive methods. As an example, figure 1 defines binary trees of numbers (an archetypal recursive union data definition) using the BSL language and the Class language.

```bsl
#lang bsl
;; A Tree is one of:
;; - (make-leaf Number)
;; - (make-node Tree Number Tree)
(define-struct leaf (v))
(define-struct node (left v right))
;; sum : Tree -> Number
;; sums the elements of the given tree
(define (sum a-tree)
  (cond
     [(leaf? a-tree) (leaf-v a-tree)]
    [else (+ (sum (node-left a-tree))
             (node-v a-tree)
             (sum (node-right a-tree)))]))
(check-expect (sum (make-leaf 7)) 7)
(check-expect (sum (make-node
                      (make-leaf 1)
                      5
                      (make-node (make-leaf 0)
                                 10
                                 (make-leaf 0)))) 16)
```

```class/1
#lang class/1
;; A Tree is one of:
;; - (new leaf Number)
;; - (new node Tree Number Tree)
;; and implements
;; sum : -> Number
;; sums the elements of this tree
(define-class leaf
  (fields v)
  (define (sum) (this . v)))
(define-class node
  (fields left v right)
  (define (sum)
    (+ (this . left . sum)
       (this . v)
       (this . right .sum))))
(check-expect ((new leaf 7) . sum) 7)
(check-expect ((new node
                (new leaf 1)
                5
                (new node (new leaf 0)
                           10
                           (new leaf 0)))) . sum) 16)
```

Figure 1: Binary tree sum in Beginning Student and in the Class language

The structure of this data definition is analogous to the approach of the previous semester but this example brings to light an important difference with the functional approach. The method for computing the sum of the empty tree is defined in the leaf class, while the method for computing the sum of a node 0. objects 1. shorthand method call 2. inheritance 3. overriding.
Methods can be defined to consume any number of arguments, but they are implicitly parameterized over this, the object that was sent the message.

3.2 Where did the cond go?

Unions, and recursive unions in particular, are a fundamental kind of data definition that students are well-versed in from the previous semester. A fundamental early lesson is how to represent (recursive) unions using classes and how to write recursive methods. As an example, figure 1 defines binary trees of numbers (an archetypal recursive union data definition) using the BSL language and the Class language.

```plaintext
#lang bsl
;; A Tree is one of:
;; - (make-leaf Number)
;; - (make-node Tree Number Tree)
(define-struct leaf (v))
(define-struct node (left v right))
;; sum : Tree -> Number
;; sums the elements of the given tree
(define (sum a-tree)
  (cond [(leaf? a-tree) (leaf-v a-tree)]
        [else (+ (sum (node-left a-tree))
                 (node-v a-tree)
                 (sum (node-right a-tree)))]))
(check-expect (sum (make-leaf 7)) 7)
(check-expect (sum (make-node (make-leaf 1) 5
                              (make-node (make-leaf 0) 10
                                          (make-leaf 0)))) 16)
```

```plaintext
#lang class/1
;; A Tree is one of:
;; - (new leaf Number)
;; - (new node Tree Number Tree)
;; and implements
;; sum : -> Number
;; sums the elements of this tree
(define-class leaf
  (fields v)
  (define (sum) (this . v)))
(define-class node
  (fields left v right)
  (define (sum) (+ (this . left . sum)
                   (this . v)
                   (this . right . sum))))
(check-expect ((new leaf 7) . sum) 7)
(check-expect ((new node (new leaf 1) 5
                         (new node (new leaf 0) 10
                                    (new leaf 0)))) . sum) 16)
```

Figure 1: Binary tree sum in Beginning Student and in the Class language

The structure of this data definition is analogous to the approach of the previous semester but this example brings to light an important difference with the functional approach. The method for computing the sum of the empty tree is defined in the leaf class, while the method for computing the sum of a node...
Methods can be defined to consume any number of arguments, but they are implicitly parameterized over this, the object that was sent the message.

3.2 Where did the cond go?

Unions, and recursive unions in particular, are a fundamental kind of data definition that students are well-versed in from the previous semester. A fundamental early lesson is how to represent (recursive) unions using classes and how to write recursive methods. As an example, figure 1 defines binary trees of numbers (an archetypal recursive union data definition) using the BSL language and the Class language.

```scheme
#lang bsl
;; A Tree is one of:
;; - (make-leaf Number)
;; - (make-node Tree Number Tree)
(define-struct leaf (v))
(define-struct node (left v right))
;; sum : Tree -> Number
;; sums the elements of the given tree
(define (sum a-tree)
  (cond [(leaf? a-tree) (leaf-v a-tree)]
        [else (+ (sum (node-left a-tree))
                 (node-v a-tree)
                 (sum (node-right a-tree)))]))
(check-expect (sum (make-leaf 7)) 7)
(check-expect (sum (make-node (make-leaf 1) 5
                                (make-node (make-leaf 0) 10
                                           (make-leaf 0)))) 16)
```

```class/1
#lang class/1
;; A Tree is one of:
;; - (new leaf Number)
;; - (new node Tree Number Tree)
;; and implements
;; sum : -> Number
;; sums the elements of this tree
(define-class leaf
  (fields v)
  (define (sum) (this . v)))
(define-class node
  (fields left v right)
  (define (sum)
    (+ (this . left . sum)
       (this . v)
       (this . right . sum))))
(check-expect ((new leaf 7) . sum) 7)
(check-expect ((new node (new leaf 1) 5
                          (new node (new leaf 0) 10
                                     (new leaf 0)))) . sum) 16)
```

Figure 1: Binary tree sum in Beginning Student and in the Class language

The structure of this data definition is analogous to the approach of the previous semester but this example brings to light an important difference with the functional approach. The method for computing the sum of the empty tree is defined in the leaf class, while the method for computing the sum of a node.

---

0. objects

1. shorthand method call

2. inheritance

3. overriding

4. first-class classes

5. mutation
the sum of the empty tree is defined in the example brings to light an important difference with the functional approach. The method for computing

```
(check-expect
(define (sum a-tree)
    ;; sums the elements of the given tree
    (cond [(leaf? a-tree) (leaf-v a-tree)]
          [else (+ (sum (node-left a-tree)) (sum (node-right a-tree)))])

(define-struct node (left v right))
(define-struct leaf (v))
;; - (make-node Tree Number Tree)
;; A Tree is one of:

#lang bsl

numbers (an archetypal recursive union data definition) using the BSL language and the Class language. Unions, and recursive unions in particular, are a fundamental kind of data definition that students are

3.2 Where did the semester. In fact, the included with each method definition, following the principles of the design recipe studied in the first

S. Tobin-Hochstadt & D. Van Horn

Java

```
import tester.*;

interface Tree {
    // sums the elements of this tree
    Integer sum();
}

class Leaf {
    Integer v;
    Leaf(Integer v) { this.v = v; }
    public Integer sum() { return this.v; }
}

class Node {
    Tree left; Integer v; Tree right;
    Node(Tree l, Integer v, Tree r) {
        this.left = l;
        this.v = v;
        this.right = r;
    }

    public Integer sum() {
        return this.left.sum() + this.v + this.right.sum();
    }
}
```
3.2 Where did the cond go?

Unions, and recursive unions in particular, are a fundamental kind of data definition that students are well-versed in from the previous semester. A fundamental early lesson is how to represent (recursive) unions using classes and how to write recursive methods. As an example, figure 1 defines binary trees of numbers (an archetypal recursive union data definition) using the BSL language and the Class language.

```scheme
#lang bsl
;; A Tree is one of:
;; - (make-leaf Number)
;; - (make-node Tree Number Tree)
(define-struct leaf (v))
(define-struct node (left v right))
;; sum : Tree -> Number
;; sums the elements of the given tree
(define (sum a-tree)
  (cond [(leaf? a-tree) (leaf-v a-tree)]
        [else (+ (sum (node-left a-tree))
                  (node-v a-tree)
                  (sum (node-right a-tree)))]))
(check-expect (sum (make-leaf 7)) 7)
(check-expect (sum (make-node (make-leaf 1) 5
                             (make-node (make-leaf 0) 10
                                        (make-leaf 0)))) 16)
```

```scheme
#lang class/1
;; A Tree is one of:
;; - (new leaf Number)
;; - (new node Tree Number Tree)
(define-class leaf (fields v) (define (sum) (this . v)))
(define-class node (fields left v right) (define (sum)
  (+ (this . left . sum)
      (this . v)
      (this . right . sum))))
(check-expect ((new leaf 7) . sum) 7)
(check-expect ((new node (new leaf 1) 5
                          (new node (new leaf 0) 10
                                     (new leaf 0)))) . sum) 16)
```

Figure 1: Binary tree sum in Beginning Student and in the Class language

The structure of this data definition is analogous to the approach of the previous semester but this example brings to light an important difference with the functional approach. The method for computing the sum of the empty tree is defined in the leaf class, while the method for computing the sum of a node...
3.2 Where did the cond go?

Unions, and recursive unions in particular, are a fundamental kind of data definition that students are well-versed in from the previous semester. A fundamental early lesson is how to represent (recursive) unions using classes and how to write recursive methods. As an example, figure 1 defines binary trees of numbers (an archetypal recursive union data definition) using the BSL language and the Class language.

```plaintext
#lang bsl
;; A Tree is one of:
;; - (make-leaf Number)
;; - (make-node Tree Number Tree)
(define-struct leaf (v))
(define-struct node (left v right))
;; sum : Tree -> Number
;; sums the elements of the given tree
(define (sum a-tree)
  (cond [(leaf? a-tree) (leaf-v a-tree)]
        [else (+ (sum (node-left a-tree))
                 (node-v a-tree)
                 (sum (node-right a-tree)))]))
(check-expect (sum (make-leaf 7)) 7)
(check-expect (sum (make-node (make-leaf 1) 5 (make-node (make-leaf 0) 10 (make-leaf 0)))) 16)
```

```plaintext
#lang class/1
;; A Tree is one of:
;; - (new leaf Number)
;; - (new node Tree Number Tree)
(define-class leaf
  (fields v)
  (define (sum) (this . v)))
(define-class node
  (fields left v right)
  (define (sum)
    (+ (this . left . sum)
       (this . v)
       (this . right . sum))))
(check-expect ((new leaf 7) . sum) 7)
(check-expect ((new node (new leaf 1) 5 (new node (new leaf 0) 10 (new leaf 0)))) . sum) 16)
```

Figure 1: Binary tree sum in Beginning Student and in the Class language

The structure of this data definition is analogous to the approach of the previous semester but this example brings to light an important difference with the functional approach. The method for computing the sum of the empty tree is defined in the leaf class, while the method for computing the sum of a node.