Building Abstractions with Data
Module 2, SICP

Marriette Katarahweire

September 25, 2023

References

- Chapter 2, SICP book
- Lecture 2b, MIT Videos

Module 2 Outline

- Data Abstraction
- Hierarchical Data and the Closure Property
- Symbolic data(programs as data)
- Multiple Representations for Abstract Data
- Systems with Generic Operations

Status so far

	Data	Procedures
Primitive	✓	✓
Combinations	-	✓
Abstractions	-	✓

Data Abstraction

- Data Abstraction: means the programming language provides to combine data objects to form compound data
- Programs are typically designed to model complex phenomena, and more often than not one must construct computational objects that have several parts in order to model real-world phenomena that have several aspects
- in this module, we cover another key aspect of any programming language: the means it provides for building abstractions by combining data objects to form compound data
- The general technique of isolating the parts of a program that deal with how data objects are represented from the parts of a program that deal with how data objects are used is a powerful design methodology called **data abstraction**
  - makes programs much easier to design, maintain, and modify.

Data Abstraction

- The basic idea of data abstraction is to structure the programs that are to use compound data objects so that they operate on "abstract data."
- with **abstract data**, our programs should use data in such a way as to make no assumptions about the data that are not strictly necessary for performing the task at hand
- a **concrete data representation** is defined independent of the programs that use the data
- The interface between these two parts of our system will be a set of procedures, called **selectors** and **constructors**, that implement the abstract data in terms of the concrete representation
- To illustrate this technique, we will consider how to design a set of procedures for manipulating rational numbers.

Why Compound Data

- to elevate the conceptual level at which we can design our programs
- to increase the modularity of our designs
- to enhance the expressive power of our language
- the ability to construct compound data objects enables us to deal with data at a higher conceptual level than that of the primitive data objects of the language

Example: Rational Numbers

- design a system to perform arithmetic with rational numbers
- a rational number can be thought of as two integers: a numerator and a denominator
- Want to add, multiply, subtract, divide and test rational numbers for equality
- an operation *add-rat* that takes two rational numbers and produces their sum
- design a program in which each rational number would be represented by two integers (a numerator and a denominator) *add-rat* would be implemented by two procedures (one producing the numerator of the sum and one producing the denominator)
  - need to explicitly keep track of which numerators corresponded to which denominators
  - such bookkeeping details would clutter the programs and our minds substantially

Example: Rational Numbers

- much better if we could "glue together" a numerator and denominator to form a pair that our programs could manipulate in a way that would be consistent with regarding a rational number as a single conceptual unit
- A pair: a compound data object
- The use of compound data also enables us to increase the modularity of our programs. If we can manipulate rational numbers directly as objects in their own right, then we can separate the part of our program that deals with rational numbers per se from the details of how rational numbers may be represented as pairs of integers

Example: Rational Numbers

- want to add, subtract, multiply,and divide them and to test whether two rational numbers are equal
- assume the constructor and selectors are available as procedures
- Constructor:
- (make-rat $< n >< d >$) returns the rational number whose numerator is the integer $< n >$ and whose denominator is the integer $< d >$
- Selectors:
- (numer $< x >$) returns the numerator of the rational number $< x >$
- (denom $< x >$) returns the denominator of the rational number $< x >$

Example: Rational Numbers

- wishful thinking: a powerful strategy of synthesis. We haven't yet said how a rational number is represented, or how the procedures *numer*, *denom*, and *make-rat* should be implemented
- if we did have these three procedures, we could then add, subtract, multiply, divide, and test equality by using known relations

Possible Operations

```
(define (add-rat x y)
   (make-rat (+ (* (numer x) (denom y))
                (* (numer y) (denom x)))
             (* (denom x) (denom y))))
(define (sub-rat x y)
   (make-rat (- (* (numer x) (denom y))
                (* (numer y) (denom x)))
             (* (denom x) (denom y))))
(define (mul-rat x y)
   (make-rat (* (numer x) (numer y))
             (* (denom x) (denom y))))
(define (div-rat x y)
   (make-rat (* (numer x) (denom y))
             (* (denom x) (numer y))))
(define (equal-rat? x y)
   (= (* (numer x) (denom y))
      (* (numer y) (denom x))))
```

Pairs

- need some way to glue together a numerator and a denominator to form a rational number
- To enable us to implement the concrete level of our data abstraction, Scheme provides a compound structure called a **pair**, which can be constructed with the primitive procedure **cons**
- This procedure *cons* takes two arguments and returns a compound data object that contains the two arguments as parts
- Given a pair,we can extract the parts using the primitive procedures **car** and **cdr**
- pairs can be used as general-purpose building blocks to create all sorts of complex data structures
- The single compound-data primitive *pair*,implemented by the procedures *cons*, *car*, and *cdr*, is the only glue we need
- Data objects constructed from pairs are called **list-structured data**

Pairs

```
(car (cons x y)) => x
(cdr (cons x y)) => y

(define x (cons 10 22))
(car x)  => 10
(cdr x)   => 22

(define x (cons 3 4))
(define y (cons 5 9))
(define z (cons x y))

(car (car z))    => 3
(cdr (cdr z))    => 9
```

Representing Rational numbers

```
(define (make-rat n d)
   (cons n d))
(define (numer r)
   (car r))
(define (denom r)
   (cdr r))

(define (print-rat x)
   (newline)
   (display (numer x))
   (display "/")
   (display (denom x)))
```

Representing Rational numbers

```
(define one-half (make-rat 1 2))
(print-rat one-half) => 1/2
(define one-third (make-rat 1 3))
(print-rat (add-rat one-half one-third))  => 5/6
(print-rat (mul-rat one-half one-third))  => 1/6
(print-rat (add-rat one-third one-third)) => 6/9
```

Representing Rational Numbers

//reduce rational numbers to lowest terms, use *gcd* to reduce the numerator and the denominator to lowest terms before constructing the pair
perform the *gcd* at construction time

```
(define (make-rat n d)
   (let ((g (gcd n d)))
      (cons (/ n g) (/ d g))))
(define (numer r)
   (car r))
(define (denom r)
   (cdr r))
```

Representing Rational Numbers

//perform the *gcd* at selection time.

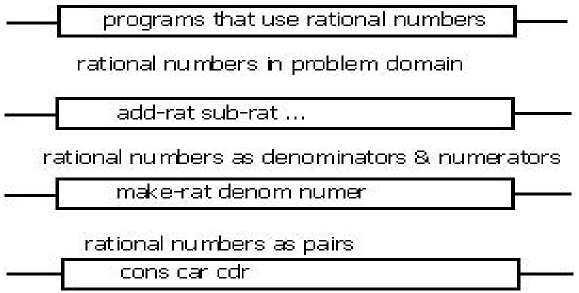
```
(define (make-rat n d)
   (cons n d))
(define (numer x)
   (let ((g (gcd (car x) (cdr x))))
      (/ (car x) g)))
(define (denom x)
   (let ((g (gcd (car x) (cdr x))))
      (/ (cdr x) g)))
```

Abstraction Barriers

Abstraction Barriers

- Principal idea of Data abstraction is to identify for each type of data object a basic set of operations in terms of which all manipulations of data objects of that type will be expressed, and then to use only those operations in manipulating the data

Abstraction Barriers


programs that use rational numbers
rational numbers in problem domain
add-rat sub-rat ...
rational numbers as denominators & numerators
make-rat denom numer
rational numbers as pairs
cons car cdr

- Horizontal lines => abstraction barriers
- Isolate different levels of the system
- At each level, separate programs above that use the data abstraction from those below that implement the data abstraction
- procedures at each level are the interfaces that define the abstraction barriers and connect the different levels

Exercise

- Create a constructor *student-name* that captures the first and last names of a student. Create corresponding selectors to get the student's first and last names. Example Student1 is "Agaba Peter". Use pretty printing to display student1's first and last names.
- Create a *point* object that has an *x-coordinate* and a y-coordinate. Provide selectors that when given a point p, are able to extract the x and y coordinates of the point.
- Create a *shoe* object which is made up of the left and right foot shoes.

What is Data?

- rational number operations are being defined in terms of data objects – numerators, denominators, and rational numbers – whose behavior was specified by the three procedures *make-rat*, *numer* and *denom*
- **Data**: some collection of selectors and constructors, together with specified conditions that these procedures must fulfill in order to be a valid representation
- *make-rat*, *numer*, and *denom* must satisfy the condition that, for any integer n and any non-zero integer d, if x is $(make - rat\ n\ d)$ then

$$x = (\text{make-rat } n\ d)$$

$$\frac{\text{numer } x}{\text{denom } x} = \frac{n}{d}$$

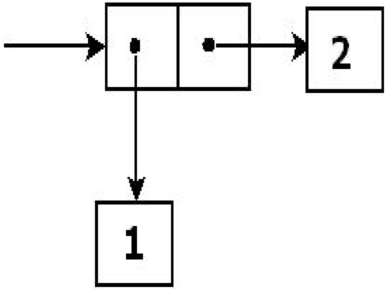
Hierarchical Data & the Closure Property

Hierarchical Data & the Closure Property

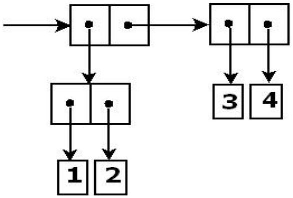
- pairs provide a primitive "glue" that we can use to construct compound data objects
- the figure shows a standard way to visualize a pair – in this case, the pair formed by (cons 1 2)
- In this representation, called **box-and-pointer notation**, each object is shown as a pointer to a box
- The box for a primitive object contains a representation of the object
- For example, the box for a number contains a numeral
- The box for a pair is actually a double box, the left part containing (a pointer to) the *car* of the pair and the right part containing the *cdr*

Box-and-pointer Notation

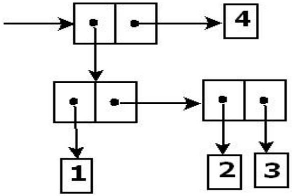
```
(cons 1 2)
```



Box-and-pointer Notation



```
(cons (cons 1 2)
      (cons 3 4))
```



```
(cons (cons 1
            (cons 2 3)
        4))
```

Exercise

Draw a box-and-pointer notation for

- the *student whose name is "Agaba Peter" and registration number "2023/HD05/121212U"*, and *point p(3,4)* objects created in an earlier exercise
- $(cons\ 1\ (cons\ 4\ 5))$
- $(cons 1(cons 2(cons 3(cons 45))))$

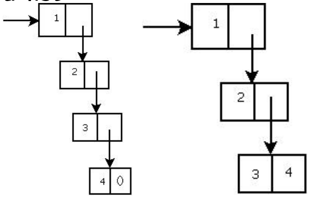
Closure Property

- Ability to create pairs whose elements are pairs
- Permits us to create hierarchical structures – structures made up of parts, which themselves are made up of parts, and so on e.g. lists & trees

Representing Sequences

Sequence: Ordered collection of data objects

List: sequence of pairs either empty list '() or any pair whose cdr is a list



Examples

```
(cons 1 (cons 2 (cons 3 (cons 4 ‘()))))  >> (1 2 3 4)
(list 1 2 3 4)   >> (1 2 3 4)
(define one-to-four (list 1 2 3 4))  >> (1 2 3 4)
(car one-to-four)  >> 1
(cdr one-to-four)  >> (2 3 4)
(car (cdr one-to-four))
(cdr (cdr (cdr one-to-four)))
(null? ’())  >> #t
(null? one-to-four)  >> #f
(caar (cons (cons 1 2 ) (cons 3 4)))   >> 1
(caddr one-to-four)  >>  3
```

Common Pitfall

Difference between (list 1 2 3 4) & (1 2 3 4)
(1 2 3 4)

```
 >>procedure application: expected procedure, given: 1;
arguments were: 2 3 4
```

(list 1 2 3 4)

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

Remember evaluation rule for combinations

List Operations

```
(define (list-ref items n)
(if (= n 0)
(car items)
(list-ref (cdr  items) (- n 1))))

(list-ref one-to-four 3) >> 4

(define (length items)
(if (null? items)
0
(+ 1 (length (cdr items)))))

(length one-to-four) >> 4
```

```
(define (lengthie items)
(define (length-iter a count)
(if (null? a)
count
(length-iter (cdr a) (+ 1 count))))
(length-iter items 0))

(lengthie one-to-four) >> 4
```

List Operations

```
(define (append list1 list2)
(if (null? list1)
list2
(cons (car list1) (append (cdr list1) list2))))

(define five-to-ten
(list 5 6 7 8 9 10))

five-to-ten
>> (5 6 7 8 9 10)

(append one-to-four five-to-ten)
>> (1 2 3 4 5 6 7 8 9 10)
```

Mapping over Lists

apply a transformation to each element in a list and generate the list of results

```
(define (scale-list items factor)
(if (null? items)
'()
(cons (* (car items) factor)
(scale-list (cdr items) factor))))

(scale-list (list 1 2 3 4 ) 10)
>> (10 20 30 40)

(define (map proc items)
(if (null? items)
'()
(cons (proc (car items))
(map proc (cdr items)))))

(map abs (list -10 2.5 -11.6 17))
>> (10 2.5 11.6 17)
```

Higher-order Procedures

```
(define (scale-list items factor)
(map (lambda (x) (* x factor))
items))

(scale-list five-to-ten 3)
>> (15 18 21 24 27 30)
```

Trees

A tree: a list whose elements are lists

Tree's branches: elements of the list

```
(define (count-leaves x)
(cond ((null? x) 0)
((not (pair? x)) 1)
(else (+ (count-leaves (car x))
(count-leaves (cdr x))))))

(define x
(cons (list 1 2) (list 3 4)))
(length x)     >> 3
(count-leaves x)    >> 4
(length (list x x))   >> 2
(count-leaves (list x x)) >> 8
```