Data Abstractions

Abelson & Sussman & Sussman sections: 2.1 & (first part)2.2

Preview

- In the next few lectures we will examine various aspects of data abstraction including
  - Motivation for using/supporting data abstractions.
    - The use of abstraction barriers to represent different parts of a program.
  - How compound data is represented in Scheme
  - How programs can be constructed from general procedures pasted together using compound data as a conventional interface.
  - How symbolic data is represented in Scheme
  - How generic procedures can be constructed in Scheme
  - The relationships between different types of abstraction.

Data abstraction

- We are all familiar with the concept of data abstraction.
  - Concealing aspects of a data representation to make it more abstract.
- We are all familiar with the motivations for data abstraction
  - Making data simpler from the users' point of view.
  - Better matching the capabilities of the data abstractions to the needs of the users.
  - Allowing the underlying representation to change without affecting users programs.
  - Admitting the possibility of generic procedures, able to treat different abstractions with similar behaviour in the same way, at some level.
- Data abstraction is a tool for the programmer
  - reduces problems associated with making large or growing systems.

Data Abstraction in Scheme

- Scheme provides no explicit mechanism for data-hiding
  - This is both a weakness and a strength
- It is a weakness because most abstractions can be broken by the programmer.
  - programmers need to know what they are doing
- From an expository point of view it is a strength
  - There is no mandated way to abstract data. Allows us to explore different ways of providing data abstraction.
- In this part of the course we will use Scheme as a vehicle to explore
  - different ways data can be represented.
  - different ways of abstracting over data and
  - different ways abstractions interact.

Data Abstraction example: rational numbers

- Rational numbers are numbers expressible using fractions:
  - e.g. 1/2, 3/18,384/17 and so on.
- We want to be able to treat rational numbers like ordinary numbers
  - we want to add, subtract, multiply, divide and test for equality.
- To do this we need to imagine that we have ways of putting together and pulling apart rational numbers:
  - `make-rat` - to make a rational out of two numbers.
  - `numer` - to return the numerator of a rational number.
  - `denom` - to return the denominator of a rational number.
- For now we will imagine that we have these operations and implement add, subtract, etc. in terms of them.
  - pretending operations exist is an important aspect of abstraction.

Rational numbers (2.1.1)

```
(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))))
```


- And so on for `mult-rat` and `div-rat`

- Now, we need to create definitions for the auxiliary procedures
  - `numer`, `denom`, and `make-rat`

Representing Rational Numbers

- Rational numbers have two components
  - a numerator and denominator
- A mechanism is needed to glue these together
  - we use the Scheme primitives `cons`, `car` and `cdr`.
- `cons` glues together two things

```
(cons 1 2)
```

- `car` accesses the first thing in a cons

```
(car (cons 1 2)) => 1
```

- cdr accesses the second thing in a cons

```
(cdr (cons 1 2)) => 2
```

Representing Rational numbers -cont'd

- Now we can define our representation in terms of `cons`, `car` and `cdr`:

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

- We might want to print rational numbers in a pretty way:
  - the standard way is a bit ugly.

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

Using rational numbers

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

- This is answer is un-normalised - we can fix this by modifying make-rat:


```
(define (make-rat n d)
        (let ((g (gcd n d)))
             (cons (/ n g) (/ d g))))
```


- Now...

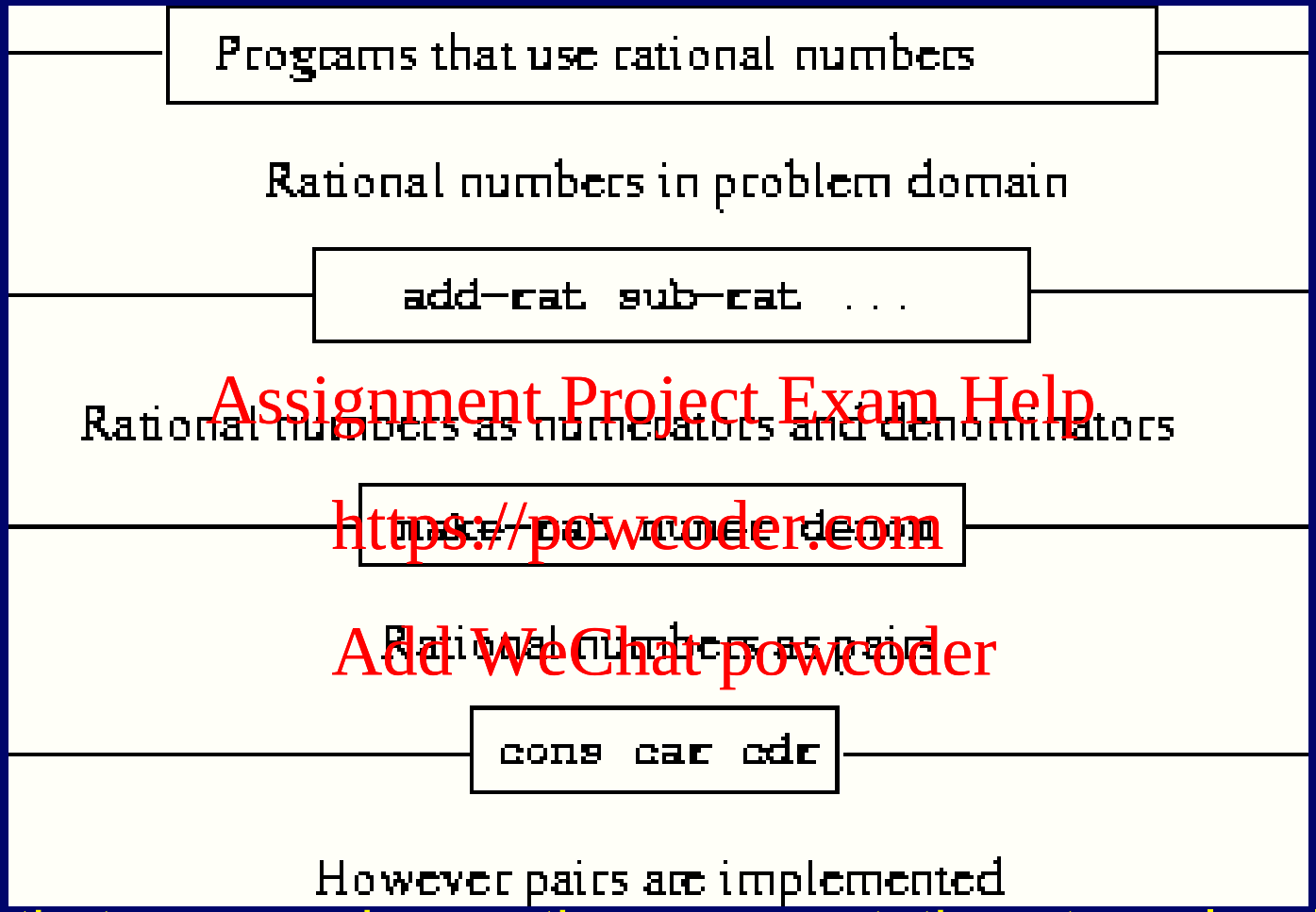
```
(define one-third (make-rat 1 3))
(print-rat (add-rat one-third one-third))
=> 2/3
```

- That's better

Abstraction Barriers (2.1.2)

- Notice how we defined our `add-rat`, `sub-rat`, etc. procedures before we knew exactly how the underlying data was represented.
  - we made use of an abstraction barrier.
- There are several abstraction barriers in the rational number system...

Abstraction Barriers



- Note that we can change the representation at any level and, as long as the interface is the same, our program still works.

Back to data representations (2.1.3)

- We used the primitive functions `cons`, `car` and `cdr` to hold and access the numerator and denominator of the rational numbers.
  - Is there a level below these primitive functions?
  - Is there something more primitive than these primitives?
- It turns out that `cons`, `car` and `cdr` can be represented in terms of other things...

Representing cons, car, cdr

```
(define (cons x y)
   (lambda (m)
     (cond ((= m 0) x)
           ((= m 1) y)
           (else
            (error "Argument not 0 or 1 -- CONS" m)))))

(define (car z) (z 0))

(define (cdr z) (z 1))
```


- Our new representation of `cons` is indistinguishable from the old:

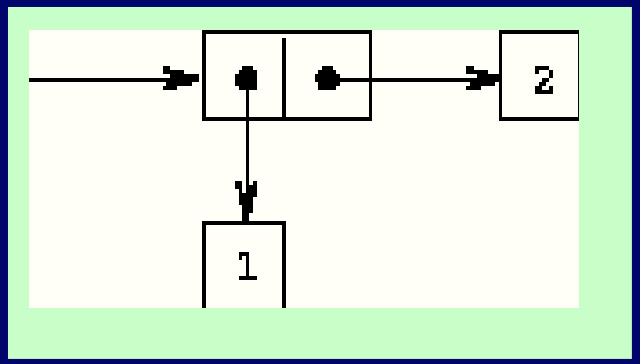
```
(car (cdr (cons 1 (cons 3 2)))) => 3

(cdr (cdr (cons 1 (cons 3 2)))) => 2
```

- Data can be represented purely as functions (as above)
  - The actual representation isn't important as long as the behaviour is what we expect.

Hierarchical data and lists 2.2

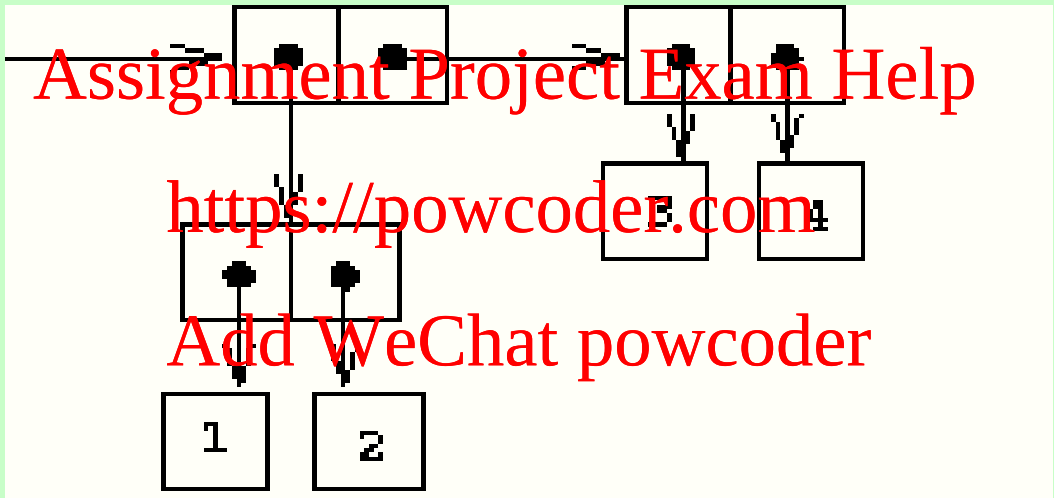
- Almost all compound data in Scheme programs uses pairs, formed from `cons`, in its underlying representation.
- Pairs can be used to form arbitrarily complex data structures.
  - lists, trees and graphs can all be represented using pairs
  - graphs are covered in chapter 3.
- Representing a single pair: `(cons 1 2)`



Other representations: trees

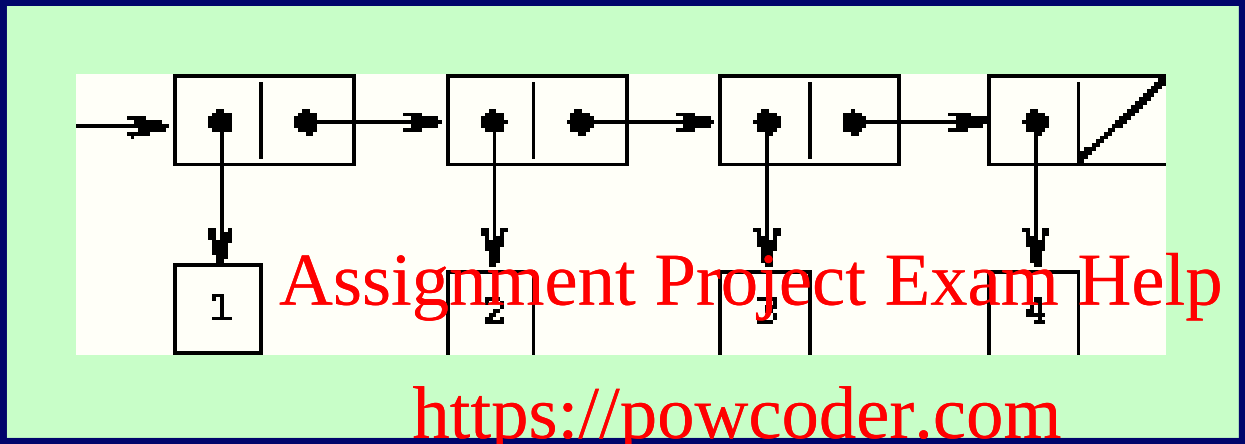
- Representing tree-like data

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



Other representations: Lists

- Cons-lists: `(cons ( 1 (cons 2 (cons 3 (cons 4 ()))))`



- `()` (nil) is a special value used to terminate a list.

- Lists are a very frequently used compound data structure in Scheme.
- There is even a special primitive used to construct lists, like the one above called `list`, e.g.

```
(list 1 2 3 4)
```

Other List operations

- `null?` - primitive procedure to determine if a list is `()`

```
(null? (list 1 2)) => ()
```

- `length` - primitive procedure returning the length of a list

```
(length (list 1 2 3)) => 3
```


- `append` - primitive procedure to join two lists


```
(append (list 1 2 3) (list 5 6)) => (1 2 3 5 6)
```


- `list-ref` - primitive procedure to access a list element

```
(list-ref (list 1 2 3) 2) => 3
```

More list operations

- `map` - primitive to apply an operation to every element of a list.

```
(map sqr (list 1 2 3)) => (1 4 9)
```

- `filter` - a programmer-defined procedure to filter a list

```
(filter less-than-3 (list 1 2 3 2)) => (1 2 2)
```

The definition is in the book but see if you can write your own version.

- `reduce` - a primitive to insert a binary operation between list elements

```
(reduce + 0 (list 1 2 3)) => 6
```

Review, Preview and Questions

- In this lecture we covered
  - data abstractions in Scheme
  - abstraction barriers
  - cons, alternate representations for cons
  - some basic procedures on cons-lists
  - hierarchical data

- *Exercises 2.3, 2.6, 2.17, 2.18, 2.20, 2.26*