CS450 - Structure of Higher Level Languages

Data Abstractions and Pairs

September 23, 2020

What to read in Chapter 1

We have already covered sections 1.1.1–1.2.4 and section 1.3.2 (lambda).

You should also read

- 1.3.1 Procedures as arguments. (Read this carefully.)
- 1.3.2 Lambda (We already did this.)
- 1.3.3 Processes as general methods.
- 1.3.4 Procedures as returned values.

These last two sections you can read quickly and not so carefully.

The `let` special form

Suppose we want to evaluate an expression like this:

$$f(x, y) = (x + y + xy)^2 + (x + y - xy)^2$$

It would be nice to be able to compute $x + y$ and xy only once, rather than twice. ($x + y$ and xy are what compiler writers call *common subexpressions*.) We can do this like this:

```
(define (f x y)
  (let ((a (+ x y))
        (b (* x y)))
    (+ (square (+ a b))
       (square (- a b))) ))
```

The `let` special form

Actually, the **let** special form is really "syntactic sugar" – it is transformed internally as follows:

```
(let ((var1 exp1)                ((lambda (var1 var2 ... varn)
      (var2 exp2)     ==>          body)
         ...                                 exp1 exp2 ... expn)
      (varn expn))
  body)
```

More Complex "Data Structures"

- So far we've seen primitive expressions and functions.
- We used abstractions to build higher order procedures.
- Sometimes we want to "glue together" several data objects.
- We can use abstraction to build compound data as well.
- For example, suppose Scheme did not contain support for fractions (i.e., for rational numbers represented as fractions).
- A rational number is essentially a pair of numbers – a numerator and a denominator.
- We could represent them separately but it would be complicated and confusing, especially if we want to define arithmetic operations on rational numbers.
- We'd need to keep track on which numerators and denominators belong where...
- We want to be able to "glue together" the two numbers.

Example – Rational Numbers

We could create a rational number package by writing procedures of the following form:

- A *constructor*

```
(make-rat n d)
```

- Two *selectors*

```
(numer x)
(denom x)
```

Example – Rational Numbers

These procedures must be related in the following ways:

```
(numer (make-rat n d)) ==> n
(denom (make-rat n d) ==> d
(make-rat (numer x) (denom x)) ==> x
```

Assuming we have these procedures, we can construct procedures to implement the usual operators. For instance, we know that

$$\frac{n_1}{d_1} + \frac{n_2}{d_2} = \frac{n_1 d_2 + n_2 d_1}{d_1 d_2}$$

with the other usual formulas for the other arithmetic operations.

Operations on Rational Numbers

```
(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)) ))
```

Operations on Rational Numbers

and also

```
(define (equal-rat? x y)
  (= (* (numer x)(denom y))
     (* (denom x)(numer y)) ))

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

So everything works out fine, as long as we can implement the constructor and the selectors.

Pairs

The basic constructors and selectors for data structures in Scheme are these:

constructor: cons

selectors: car and cdr (read: could-er)

```
(define x (cons 1 2))
```

We say that `cons` creates a *pair*. With this definition,

```
(car x) ==> 1
(cdr x) ==> 2
```

Suppose we also define

```
(define y (cons 3 4))
(define z (cons x y))
```

```
(car (car z)) ==> 1 ;; this is also produced by (caar z)
(car (cdr z)) ==> 3 ;; this is also produced by (cadr z)
```

So now we can implement our rational number package as follows:

```
(define (make-rat n d)
  (cons n d))

(define (numer x)
  (car x))

(define (denom x)
  (cdr x))
```

(Incidentally, we could just as well have written

```
(define make-rat cons)
```

Pairs

- Of course we have not dealt at all with questions of efficiency, representations, and lowest terms.
- For instance, should rational numbers be stored internally in lowest terms, or should they only be put into lowest terms when they are output?
- There are tradeoffs to be made here.

Pairs

But in any case, we have the following hierarchy of objects and procedures:

1. What the user sees is rational numbers, together with arithmetic operations on them (`add-rat`, etc.).
2. Those arithmetic operations are implemented in terms of the constructors and selectors (`make-rat`, etc.) and in terms of Scheme's primitive arithmetic operations on integers (+, etc.).
3. The constructors and selectors are implemented in terms of Scheme's primitive constructors and selectors (`cons`, `car`, and `cdr`).
4. `cons`, `car`, and `cdr` are implemented in some fashion. In fact, we have said nothing about how pairs are implemented in Scheme.

What is Data?

- What really is a pair? It could be implemented internally in various ways.
- It could be any data structure so long as `cons`, `car`, and `cdr` work together properly.
- In fact, *it might not be a data structure at all*, but a function:

```
(define (cons x y)
  (define (dispatch m)
    (cond ((= m 0) x)
          ((= m 1) y)
          (else (error "Improper argument")) ))
  dispatch)

(define (car z) (z 0))
(define (cdr z) (z 1))
```

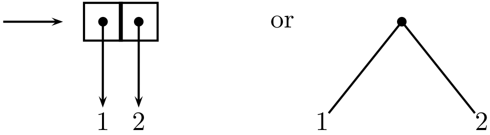
Another Way to Define cons

```
(define (cons x y)
  (lambda (m)
    (cond ((= m 0) x)
          ((= m 1) y)
          (else (error "Improper argument")) )))
```

Thus in Scheme, the difference between data and procedures fades away. This will be very important in this course.

Representing `cons`

The conventional way to represent a pair (say, `(cons 1 2)`) is like this:



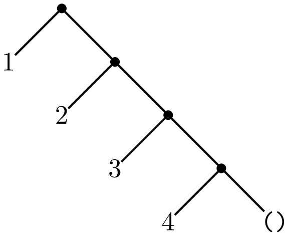
Lists (or sequences)

- The *empty list* is represented in a Scheme expression as `'()`. (We'll talk about that single quote mark later.)
- The list consisting of the numbers 1, 2, 3, 4 in that order is created by the constructor `(list 1 2 3 4)`.
- This is syntactic sugar for the following:

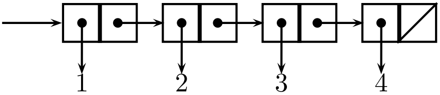
```
(cons 1
      (cons 2
            (cons 3
                  (cons 4
                        '() ))))
```

Lists (or sequences)

That is, it is turned internally into the following data structure:



or



Lists (or sequences)

- So a list is actually constructed internally out of pairs.
- That is, it's really a special kind of binary tree.
- On output, pairs are represented with dots. For instance:

```
==> (cons 1 2)

(1 . 2)
==>
```

However, when the data structure is actually a list, it is represented as such.

Lists (or sequences)

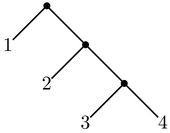
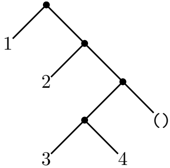
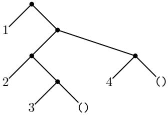
The above list could be represented on output as

```
(1 . (2 . (3 . (4 . ())))))
```

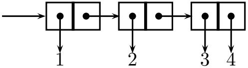
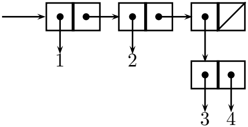
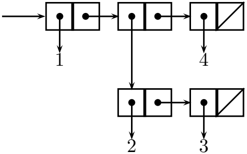
But that's not what actually happens:

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

(1 2 3 4)
==> (cons 1
          (cons 2
                (cons 3
                      (cons 4
                            '() ))))

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

Display vs. Representation

Internal Data Structure	Output Representation	Input Representation
	`(1 2 3 . 4)`	`(cons 1 (cons 2 (cons 3 4)))`
	`(1 2 (3 . 4))`	`(list 1 2 (cons 3 4))`
	`(1 (2 3) 4)`	`(list 1 (list 2 3) 4)`

Display vs. Representation (box + pointer)

Internal Data Structure	Output Representation	Input Representation
	`(1 2 3 . 4)`	`(cons 1 (cons 2 (cons 3 4)))`
	`(1 2 (3 . 4))`	`(list 1 2 (cons 3 4))`
	`(1 (2 3) 4)`	`(list 1 (list 2 3) 4)`

Note that if

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

Then

`(car x)`	is	`1`	an element of the list
`(cdr x)`	is	`(2 3 4)`	the rest of the list

and further,

`(cons 10 x)`	is	`(10 1 2 3 4)`
`(caddr x)`	is	3
`(cddddr x)`	is	()

Procedures that manipulate lists

```
(define (count-down-from n)
  (cond ((= n 0) (display 0) (newline))
        (else (display n) (newline)
        (count-down-from (- n 1))) ))
```

Question: Why can't we just use "if" here?

```
(define (count-up-to n)
  (cond ((= n 0) (display 0)(newline))
        (else (count-up-to (- n 1))
        (display n)(newline)) ))
```

```
(define (add-elements items)
  (if (null? items)
      0
      (+ (car items) (add-elements (cdr items))) ))
```

Procedures that manipulate lists

```
(define (add-squares items)
  (if (null? items)
      0
      (+ (square (car items))
         (add-squares (cdr items))) ))

(define (add-transformed items func)
  (if (null? items)
      0
      (+ (func (car items))
         (add-transformed (cdr items) func)) ))

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

Procedures that manipulate lists

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

Here is an iterative form of `length`:

```
(define (length items)
  (length-iter items 0))

(define (length-iter items result)
  (if (null? items)
      result
      (length-iter (cdr items) (+ result 1)) ))
```

Procedures that manipulate lists

```
(define (append x y)
  (if (null? x)
      y
      (cons (car x) (append (cdr x) y)) ))

==> (define squares (list 1 4 9 16 25))

==> (define odds (list 1 3 5 7 9))

==> (append squares odds)

(1 4 9 16 25 1 3 5 7 9)
==>
```

Procedures that manipulate lists

- See a pattern here? We have a function f, we apply it to the `car` and then recursively to the `cdr` .
- Most examples we saw behave in a similar way, so we can abstract even more...
- Write a higher order procedure that takes a function and applies it to a sequence.

```
(define (map func sequence)
  (if (null? sequence)
      '()
      (cons (func (car sequence))
            (map func (cdr sequence))) ))
```

Procedures that manipulate lists

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

(10 2.5 11.6 17)
==> (map (lambda (x) (* x x))
         (list 1 2 3 4))

(1 4 9 16)
==>

(define (filter predicate sequence)
  (cond ((null? sequence) '())
        ((predicate (car sequence))
         (cons (car sequence)
                    (filter predicate (cdr sequence))))
        (else (filter predicate (cdr sequence))) ))
```