

# Section Notes 3

CS51—Spring, 2009

Week of February 15, 2009

## Outline

1. Cons Cells
2. Filter
3. Graphs
4. More Practice with Recursion
5. Style
6. Note: in the interest of saving paper, please return these notes to your section leader at the end of section if you do not intend to keep them.

## 1 Review

By now you should be familiar with the following concepts:

- Writing in Scheme
- Programming with recursion
- Abstract Data Structures
- Concrete Data Structures, especially lists and trees

## 2 Cons cells

It has been said that the universe is made of cons cells. Believe it or not, lists in Scheme are made of the same substance. Coincidence? You be the judge.

A cons cell is an object that references two other objects. The two references in a cons cell are called the *car* and the *cdr*. `cons` is a useful function that creates a cons cell with the given *car* (first) and *cdr* (second):

```
> (cons 1 empty)
(list 1)
> (cons "left" (cons "right" empty))
(list "left" "right")
```

## 2.1 Review of proper lists

Quick review: there is a specific definition for what constitutes an actual list in Scheme, and here it is:

A list is either the empty list, or a pair whose cdr is a list.

Notice again that this definition is recursive. Also, in order to define lists, we have to introduce the empty list. The empty list is written as `empty` and is defined to be the list with no elements. What is the reason for such fussiness in defining lists? (Answer: it provides a base case to test in functions that operate on lists.) Note that the empty list is emphatically not a cons cell.

```
(define (my-list? lst)
```

When dealing with cons cell structures, it is helpful to draw them out with boxes and arrows. Try drawing the structure of the nested list `(list (list "a") (list (list "b") (list "c"))) (list (list (list "d"))))` (which we will call `lst`):

Now work out the values of these expressions (note that if you find yourself using functions like `caaddr` too often, you probably need more abstraction).

```
(car lst), (cdr lst), (cadr lst), (cadadr lst), (caaddr lst)
```

Note that the contract for `cons` is: `(-> any/c list? list?)`

That is, `cons` takes anything and a list, and produces a list. What would happen if we changed the contract to: `(-> any/c any/c list?)`

## 3 Filter

A filter is a device that lets some inputs through while blocking others. We can imagine applying this idea to Scheme lists. Let's say we have a list of integers representing the addresses of internet users who have visited our website. We would like to process this list to extract statistics like the number of unique users, the number of users from a particular network (such as Harvard's campus network), and so on.

Filtering functions are quite useful for this task. One thing we must do first is remove addresses that don't count for various reasons. For example, we know that we obsessively visit our own site every day, so we should remove our address from the list to get more realistic data.

### 3.1 Filtering numbers

For practice, let's consider trying to write a function that filters a list of numbers according to some predicate. First, write a filter that will filter a list of numbers so that only those elements that are less than 3 will be retained:

```
(define (filter-<-3 lst)
```

Write a function that filters numbers less than 5:

```
(define (filter-<-5 lst)
```

How can we abstract to `filter-<` without writing separate functions for each number?

```
(define (filter-< n lst)
```

Write a function that filters numbers greater than 7:

```
(define (filter->-7 lst)
```

As we can see, we can't abstract further without passing the predicate as well. How can we write a function in which we can pass the predicate? Hint: a function to which a predicate is passed for filtering is essentially the conventional filter function.

How would we write a pirate filter function that filters all strings which end in 'rrr'? (Hint: Scheme has a function called `(string-suffix? PATTERN STRING)`)

Write a function that can remove all occurrences of a particular number from a list of numbers (don't worry too much about error checking):

```
(define (remove-eq-to-n lst n)
```

Next, we want to count unique users. We can do that by removing duplicate entries from the list. The function we just wrote should be helpful.

```
(define (remove-duplicates lst)
```

### 3.2 Generalize that

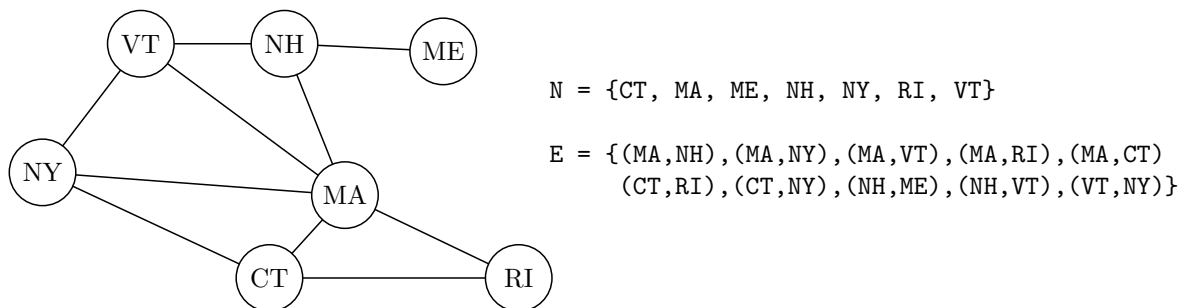
Good software engineers should always be on the lookout for code that can be made re-usable. The function `remove-eq-to-n` would be a lot more useful if it could work on lists of anything. Write a more general filter function that takes a predicate function, `pred?`, as an argument, and removes all items that fail to satisfy the predicate. Note that predicates can be passed as arguments; just substitute the argument (`pred?`) in the function argument, treating it as a function when relevant.

```
(define (filter lst pred?)
```

## 4 Graphs

### 4.1 Definition

Graphs are general models useful for representing any situation involving entities with relationships or connections between them. A graph is defined as a set of *vertices* (or *nodes*) and *edges* connecting them. The classic real-world analogy would be that nodes are cities and edges are roads, train tracks, airline routes, etc. from one city to another. Consider the following graph:



This particular graph is *undirected*, meaning that whenever A is connected to B it is implied that B is connected to A. What are some common situations that can be modeled by undirected graphs?

By contrast, in a *directed* graph, it matters which way vertices are listed in the ordered pair representing an edge (such as  $(CT, NY)$ ). For nodes to be mutually connected in a directed graph, we have to specify, for example, that NY connects to CT and that CT connects to NY. A directed graph can be used to represent a conceptually undirected graph by providing such pairs for every edge. Be sure that you understand this concept, as it is used in this week's problem set. The graph drawn above would be represented by the following directed graph:

$N = \{CT, MA, ME, NH, NY, RI, VT\}$

$E = \{(MA, NH), (NH, MA), (MA, NY), (NY, MA), (MA, VT), (VT, MA), (MA, RI), (RI, MA), (MA, CT), (CT, MA), (CT, RI), (RI, CT), (CT, NY), (NY, CT), (NH, ME), (ME, NH), (NH, VT), (VT, NH), (VT, NY), (NY, VT)\}$

Note, that mutually connected edges in a directed graph may serve a purpose other than modeling an undirected graph. Directed graphs are sometimes useful when we want to label edges with weights (representing length, travel time, cost, etc.) and the directions between two places are unequal. Think of the George Washington Bridge, which connects New York and New Jersey. Nobody really wants to go to New Jersey, so that side of the road moves smoothly with little delay. The other side, filled with drivers desperate to get to Manhattan, is often more crowded. To model this scenario we would need two directed edges with different travel times, even though the two edges connect the same places.

### 4.2 Representing Graphs in Scheme

There are many ways that this Abstract Data Type can be represented in Scheme, however all graph implementations should share a common interface. In Scheme, we enforce the contracts provided by an

interface using a module. At this point in CS51, you have already seen two concrete representations of this Abstract Data Type. Before we look at the implementations, let's define the interface of a graph module. Remember that a module in Scheme is used to help organize our code into reusable libraries. Also, recall that the `provide/contract` mechanism permits us to impose constraints and promises on the use of a module's interface. With that in mind, fill in the rest of the `provide/contract` component of the following module appropriately. What other functions or constants should be part of this interface?

```
(module graph scheme
  ...
  (provide/contract
    [graph? (-> any/c boolean?)])
)
```

#### 4.2.1 Edge-list representation

Given some a defined interface, we can now think about how we would implement this interface (i.e. fill in the ... above). In Problem Set 2, we represent a directed graph as a list of `edge` structures, where an `edge` is simply defined as:

```
(define-struct edge (source dest))
```

For the Problem Set, we leave it up to you to incorporate this simple type definition into an interface for a directed graph. Although we don't ask you to implement your graph according to a specific interface, you may find that the module/contract system provides useful tools for designing your code. What functions in the interface above do you think would be challenging to write? Which might be challenging to make efficient given this interface?

#### 4.2.2 Neighbor association-list representation

In Project 1a we provided you with an interface for an undirected graph: `ugraph`. We also provided an implementation of `ugraph` that happens to use a special kind of list, but put that out of your mind for a minute.

What functions/methods are valid for you to call on a `ugraph?`, as defined in the comments of `proj1a.scm`:

Do the functions that we provided in our interface match the functions you came up with earlier in section? If not, how difficult would it be to change these interfaces so they match each other?

In Project 1a we also provide the implementation of `ugraph`, which is built out of an *association list*. An *association list* is a list that contains lists where each of the element lists has a specific structure. An element of an association list represents two values that are “associated” as a key-value pair. Association lists are a commonly used type in Scheme. Consider, for example, how you might represent information about students and their residences as an association list:

```
(list (list "Juan" "Eliot House") (list "Mae" "Currier House")
      (list "John" "Delta House") (list "Bart" "Simpsons House"))
```

In this example, the “key” is a name and the “value” is the residence. What would the list look like if we changed the value to be a list of courses that are associated with the student (i.e. the courses the student is taking)?

Now that you are familiar with the association list, consider how this type can be used to represent a graph. A graph can be represented as an association list where the keys represent the nodes and the values represent the neighbor lists. The neighbor list of a node, *N*, contains all of the nodes for which there exists an edge between *N* and the prospective neighbor.

Represent the graph of US States from earlier in this section as an association/neighbor list as just described:

*Note:* Despite the fact that you may know that a graph is represented as an association list or a list of edges, you should never write code that relies on the particular implementation of an Abstract Data Type, always write code against the public interface. If we had made `ugraph` into a module, you wouldn’t even be allowed to code against its internal representation! If you write code that makes assumptions about the guts of an Abstract Data Type, then your code will most likely break should the representation change.

## 5 Recursion example

How many different ways can we make change, given half-dollars, quarters, dimes, nickels, and pennies? Consider that the number of ways to change amount  $a$  using  $n$  kinds of coins equals

- the number of ways to change amount  $a$  using all but the first kind of coin, plus
- the number of ways to change amount  $a - d$  using  $n$  kinds of coins, where  $d$  is the denomination of the first coin

Which 3 base cases can we specify?

We are given the following function, which takes as input the number of kinds of coins available and returns the denomination of the first kind. Write `(count-change amount)` with one other helper function. (Hint: The helper function should look like `(cc-helper amount kinds-of-coins)`.)

```
(define (first-denomination kinds-of-coins)
  (cond [(= kinds-of-coins 1) 1]
        [(= kinds-of-coins 2) 5]
        [(= kinds-of-coins 3) 10]
        [(= kinds-of-coins 4) 25]
        [(= kinds-of-coins 5) 50]))
```

## 6 Style notes

- Predicates should not return 'true' or 'false' explicitly.
- Try to avoid elaborate nested `cond/if` expressions.