1 Pairs and lists

Pairs and lists are the most commonly used data structures in Scheme, so familiarity with them is essential. We will mostly be dealing with lists, but in order to understand lists, there are a few things you need to know about pairs:

- To make a pair, we use the primitive procedure `cons`
- To get the first thing in a pair, we use `car`
- To get the second thing in a pair, we use `cdr`
- A pair with `null` as its second thing comprises a *list* with a single element: the first thing in the pair
- The procedure `pair?` tells you if something is a pair or not

A few useful things to know about lists:

- The most basic list is the empty list `()`, which is equivalent to `null`
- There are two ways to construct a list:
  - Using the primitive `list`, which takes in a variable number of arguments that become the list elements: `(list 1 2 3)` ⇒ `(1 2 3)`
  - Using `cons` to build up a list using pairs as follows: `(cons 1 (cons 2 (cons 3 null)))` ⇒ `(1 2 3)`
- The `car` of a list is simply the first item in a list
- The `cdr` of a list is a list with the first item removed
- The procedure `list?` tells you if something is a list or not

1.1 Example: map, filter, and reduce

These three high-order functions for lists are extremely handy, and can save you a lot of code.

- Map: Given a function of one argument and a list, returns a list with the function applied to each list item

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

- Filter: Given a function of one argument that returns true/false and a list, returns a list with all the items removed for which the function returns false

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

- Reduce: Given a function that "reduces" two arguments down to a single argument, repeatedly applies this function to the list, starting with the first two elements

```
(define (reduce proc init lst)
 (if (null? lst) init
     (proc (car lst) (reduce proc init (cdr lst)))))
```

Cool trick: We can implement `map` using `reduce`!

```
(define (map p sequence)
 (reduce (lambda (x y) (cons (p x) y)) null sequence))
```

1.2 Example: Censoring a list

The problem: we want to replace everything in a list that matches a certain symbol with '***'.

```
(define (censor symbol x)
(if (null? x)
    '()
    (let ((rest (censor symbol (cdr x))))
      (cons
       (if (eq? (car x) symbol)
           '***
           (car x))
       rest))))
```

Using our ever-so-useful friend `map`:

```
(define (censor symbol x)
(map (lambda (a)
       (if (eq? a symbol) '*** a))
     x))
```

2 Symbolic programming

Scheme lets us work with arbitrary symbols as well as numbers using the primitive `quote`. An expression that uses `quote` evaluates to *whatever you tell it to* via the argument you give to `quote`. Some examples:

- `(quote 3)` ⇒ `3`
- `(quote x)` ⇒ `x`
- `(quote foo)` ⇒ `foo`

We can use it with more complicated return values too, like lists and other data structures:

- `(quote (1 2))` ⇒ `(1 2)`
- `(quote (foo (bar) bat))` ⇒ `(foo (bar) bat)`

A shortcut for `quote` is to use the symbol `'` (the single quotation mark) in front of the argument, like such: `'foo` ⇒ `foo`, `'`(foo (bar) bat) ⇒ `(foo (bar) bat)`, etc. This is a really handy way to handle symbolic list processing (hint: this knowledge will come in handy for the next programming assignment!).

2.1 Example: Substitution with symbols

This was tutor problem 3.2.2 ("Babel Fish"). Here's what we were given:

```
(define *english-to-french*
       '((cat chat) (cake gateau) (present cadeau) (I je) (eat mange)
         (the le)))
```

and we made a simple lookup procedure:

```
(define lookup
 (lambda (word dictionary)
     (cond ((null? dictionary) #f)
           ((equal? (caar dictionary) word) (cadar dictionary))
           (else (lookup word (cdr dictionary))))))
```

Using `map`, we were able to make a really simple sentence translator:

```
(define translate
 (lambda (sentence dictionary)
   (map (lambda (w) (lookup w dictionary)) sentence)))
```

Without map, it takes a bit more code:

```
(define (translate2 sentence dictionary)
  (if (null? sentence) '()
      (cons (lookup (car sentence) dictionary)
            (translate2 (cdr sentence) dictionary))))
```

3 Hierarchical data

We sometimes want to use trees, a richer data structure than lists. We construct trees as lists of lists of varying depths. We refer to the nestings of lists as *branches* and the items at the ends of branches as *leaves*.

3.1 Example: Counting leaves of a tree

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

3.2 Example: Substitution on a tree

```
(define (deep-translate tree dictionary)
 (cond ((null? dictionary) #f)
       ((null? tree) '())
       ((not (pair? tree)) (lookup tree dictionary))
       (else (map (lambda (t) (deep-translate t dictionary)) tree))))
```

3.3 Example: Censoring a tree

Same problem as before, but now we want to censor everything in a tree. Note the use of `map`.

```
(define (deep-censor symbol x)
(cond ((null? x) '())
      ((not (pair? x))
       (if (eq? x symbol) '*** x))
      (else
       (map (lambda (a) (deep-censor symbol a))
            x))))
```