Harvey Mudd College
Computer Science 60
Spring 2010

Assignment 2
Unicalc API
Due. 11:59 p.m., Wed., 3 February 2010

This is the first part of a 2-part assignment. **API** stands for "Application Programming Interface", essentially the visible functions in a library used to build complete applications. **Unicalc** is a calculator that includes physical and other units, rather than just numbers.

Units are important to computation because they eliminate an element sometimes left to human interpretation. In the area of engineering, failure to interpret numbers without their units specified has been known to lead to failure of space missions for example. In the wikipedia article about NASA's Mars Climate Orbiter, http://en.wikipedia.org/wiki/Mars_Climate_Orbiter, it is stated:

> "The Mars Climate Orbiter was intended to enter orbit at an altitude of 140–150 km (460,000-500,000 ft.) above Mars. However, a navigation error caused the spacecraft to reach as low as 57 km (190,000 ft.). The spacecraft was destroyed by atmospheric stresses and friction at this low altitude. The navigation error arose because a NASA subcontractor (Lockheed Martin) used Imperial units (pound-seconds) instead of the metric system."

A **Unicalc Quantity** consists of three parts:

- A number, called the **multiplier**
- A flat list of symbols, called the **numerator**
- A flat list of symbols, called the **denominator**

Our API specifies that a Quantity is represented by a list of these components, in this order, for example, using a Scheme literal, the following would represent 2.5 kg m/s^2:

```
'(2.5 (kg meter) (second second))
```

For this problem, Quantities will be entered as test cases as literals. In the next assignment, we will construct a Scheme-like interpreter that accepts arithmetic expressions containing quantities.

On the main course webpage, you will find **unicalc-db.scm**. This is a Scheme source file for the Unicalc **database**. The database is, in effect, a set of equations defining single symbols, called **units**, in terms of Quantities. As a Scheme structure, it is an association list, **unicalc-db**. You will want to download this file and load it with your solution to this assignment: `(load "unicalc-db.scm")`.

The API is a set of functions that you will construct, as described below, along with other helper functions. To describe our functions, we need a couple more definitions.

A symbol that is defined in the database (as the first element of an element of the association list) is called a **defined unit**. A symbol that is not defined is called a **basic unit**. Examples of defined units are: mile, day, pound. Examples of basic units are: kg, second, meter.

Defined units can be **expanded** into equivalent Quantities consisting of only basic units in one or more steps. (You can assume for now that there are no circular definitions in the database.) Also, basic units can be converted into quantities by making them the only element in the numerator list, accompanied by a multiplier of 1 and an empty denominator list.

A Quantity is called **normalized** provided that the following conditions are true:

- The numerator and denominator consist of only basic units.
- The numerator and denominator are *sorted* in ascending alphabetic order.
- No unit appears in both the numerator and denominator.

We don't require the **user** to provide normalized Quantities, but it helps make the processing more efficient if we use them exclusively inside our functions. For example, the sorted aspect allows us to use the "merge pattern" described in the appendix for normalization.

Here are the functions you need to provide in the API. Note that **divide** effectively achieves conversion from one kind of normalized Quantity to another.

Function Call Form	Meaning
`(normalize-unit` Unit`)`	Returns a normalized Quantity for a single unit.
`(normalize` Quantity`)`	Converts any Quantity to a normalized Quantity.
`(multiply` Quantity1 Quantity2`)`	Multiplies two normalized Quantities, returning a normalized Quantity.
`(divide` Quantity1 Quantity2`)`	Divides normalized Quantity1 by normalized Quantity2, returning a normalized Quantity.

Note that the database, unicalc-db, is global and behind the scenes, rather than being passed as an argument. Test cases will be provided as **`uncialc-tests.scm`**.

Appendix: The Merge Pattern

This is a pattern that has many applications. We'll illustrate it in the manipulation of sets represented as lists. Sets are not lists, of course. Duplicates do not matter in sets, and neither does order, but they do with lists. However, we can represent the set abstraction using lists. It behooves us to keep the elements of the sets sorted, assuming the data has a natural order, as strings and integers do. It also helps to keep duplicates out of sets. They just take up extra space, for one thing.

Given that we have chosen to represent sets as sorted lists, we can define some operations on the sets, and this is where the merge pattern comes in. We start with set union:

```
(define (union A B)
  (cond
    ((null? A)                B)
    ((null? B)                A)
    ((= (first A) (first B)) (cons (first A) (union (rest A) (rest B))))
    ((< (first A) (first B)) (cons (first A) (union (rest A) B)))
    (else                    (cons (first B) (union A (rest B))))))
```

You might not have used the `cond` form up to this point. It is basically like a set of cascaded `if`'s, pairing up tests with results, and finally using the keyword else to give a result for the default case.

Here is the logic behind union. First, we are assuming that the arguments A and B are sorted and contain no duplicates. Otherwise this code will not work. It also will return results with those properties.

The first two cond lines are obvious: if A is empty, then the result is the same as B, and vice-versa. The next cond line handles the case where the first elements of A and B are the same. We only want to return one of those two, so we use (first A). The rest of the union is a recursive call to union, without (first A) and (first B) any longer present.

The third line uses the sorted nature of the arguments. If the first element of A is less than the first of B, and B is sorted, then we know that the first element of A cannot occur anywhere in B. Thus we include it in the result, and move on to take the union of the rest of A with B.

As we have disposed of the case whether first element of A is either equal to, or less than, the first element of B, the only possibility remaining is that the first element of B is less than the first element of A. So by the same reasoning as in the preceding paragraph, we include the first element of B in the result, and move on to take the union of the rest of B with A.

A similar strategy can be used for set intersection. See if you can explain it:

```
 (define (intersection A B)
  (cond
    ((null? A)                ())
    ((null? B)                ())
    ((= (first A) (first B)) (cons (first A) (intersection (rest A) (rest B))))
    ((< (first A) (first B)) (intersection (rest A) B))
    (else                    (intersection A (rest B)))))
```

Finally, try difference, which means elements that are in the first set but not the second:

```
(define (difference A B)
  (cond
    ((null? A)                ())
    ((null? B)                A)
    ((= (first A) (first B)) (difference (rest A) (rest B)))
    ((< (first A) (first B)) (cons (first A) (difference (rest A) B)))
    (else                    (difference A (rest B)))))
```