Harvey Mudd College
Computer Science 42
Assignments 3 and 4

Part 1: Due Wednesday, October 3, by 11:59pm

Part 2: Due Wednesday, October 10, by 11:59pm

I realize that the Grace Hopper Conference is coming up for some of you. Dividing the assignment into two parts gives everyone, both attendees and others, flexibility in how to schedule work. If you think you won't have enough time before leaving for the conference, then try to do more before. Grading on assignment 3 will not be rigid.

Unicalc CLI

CLI stands for Command Line Interface. Unlike an API, which is intended for use by developers, a CLI is a primitive interface for the end-user. It may sometimes be used as a stepping stone toward a GUI (Graphical User Interface) as well, and some applications have both a CLI and a GUI.

Having already constructed an API for Unicalc, this problem puts our API to work behind a CLI. The inputs will be single lines typed after a prompt. Each line will be an expression that Unicalc is supposed to evaluate. The features of our expressions include:

- Numbers, ideally integers, floats, and maybe rationals
- Units (both basic and defined, using our terminology from the previous assignment)
- Operators
    - ^ for raising to an integer power, as in `meter^3`
    - * (optional) or juxtaposition for multiply, as in `acre foot` or `foot * pound`
    - / for divide, as in `foot/second`
    - + for add, as in `meter + yard`
    - – for subtract, as in `9 - 6`
    - parentheses for grouping, as in `(foot / second) year`

The operators are listed in order of precedence or binding strength, but multiply and divide have the same strength and group left-to-right, as do add and subtract. The CLI will loop forever, offering a prompt, then allowing the user to enter an expression on a line. The line ends when return is pressed.

In part 1, you do not have to worry about the actual command line. Instead, you will just make calls to a function `uniparse`, which has a single string argument and returns a Racket list structure for the string similar to the tests for the previous assignment. In part 2, this expression will be given to the Racket `eval` function, which will evaluate it as if it were typed on the Racket command line, and print out the value.

Thus a part 1 test, when done from the Racket command line, will look like this:

```
(uniparse "345 newton meter/second")
```

The result should look like something like this:

```
'(divide (multiply (make-numeric-quantity 345)
                   (multiply (normalize-unit 'newton)
                             (normalize-unit 'meter)))
         (normalize-unit 'second))
```

Note that the whole expression is quoted, plus there are additionally quoted symbols inside. The above form of expression is the "meaning" of the argument to the string argument to `uniparse`. Returning this form of meaning, rather than just the final value, is an exercise in seeing the form that a meaning might take.

When such a meaning expression is given to the `eval` function, with the Unicalc API present, the returned value will be

```
'(345 (kg meter meter) (second second second))
```

To this you would apply a formatting function to get it back into the original form. In the second part of the assignment, we will define the user interface, reading input from the command line, formatting, and printing it.

What to do for Part 1:

First, we need a **grammar** for the input language. Such a grammar was discussed at length in class on Thursday, September 27, 2012, but I will probably suggest some slight changes on Tuesday, October 2.

Second, we need **parse functions** corresponding to the non-terminals in the grammar. The parse function for the start symbol of the grammar will be called by `uniparse`.

Note: It is ***not*** recommended that you try to recognized individual Unicalc *Quantity* expressions in parsing. Instead, recognize numbers and units and leave it up to evaluation to form *Quantities*. This will be much simpler.

Submit: Working parse for a subset of the language, plus test cases showing what works. For part 1, we don't insist that your function cover all expressions. However, it should cover *some* expressions and do so cleanly, without failure. In part 2, it will need to cover test cases entered from the command line.

Note: If the Unicalc API that you submitted for the last assignment doesn't quite work, you can fix it in this one to recover up to 50% of lost points. Alternatively, email me and I will provide a working API that you can use. Please note on your assignment 4 if you are exercising either of these options.

Design, commenting, testing and formatting

Design, formatting and testing will account for roughly a quarter of the credit.

- Format your definitions so that they are **readable** as if on a sheet of paper in portrait orientation, in a 12 point font. Using 80 characters per line as a soft limit is a guide. Align arguments of long lines vertically. Readable code is important.

- Your functions should have a single, well-documented purpose. If a function is doing too many different things, break it up further into cleanly defined pieces. Individual functions should have the purpose at the top. Explanation of workings should come after purpose, below the function, or on the side. Try not to interleave comment lines and code lines. If you feel the need, leave blank lines to make it more readable.

- You should document your overall approach in the comments of the top-level functions. This will help you track the different facets of the computation being composed into an overall solution.

API Requirements

These are the function names that you are allowed to use in the API. Don't use different names.

Function Call Form	Meaning
(**multiply** *Q1 Q2*)	Returns the normalized result of multiplying normalized quantity *Q1* by *Q2*.
(**divide** *Q1 Q2*)	Returns the normalized result of dividing normalized quantity *Q1* by *Q2*.
(**add** *Q1 Q2*)	Returns the result of adding normalized *Q1* to normalized *Q2*, provided the quantities are interconvertible. The result should be in units of Q1. Returns a list `('error ('add Q1 Q2))` value if not.
(**subtract** *Q1 Q2*)	Returns the result of subtracting normalized *Q2* from normalized *Q1*, provided the quantities are interconvertible. The result should be in units of Q1. Returns a list `('error ('subtract Q1 Q2))` if not.
(**power** *Q1 N*)	Returns the result of raising *Q1* to the integer power *N* returning a normalized Quantity. If *N* is not an integer, returns a list `('error ('power Q1 N))` if not. You should allow for negative powers.
(**normalize-unit** *Unit*)	Make a ***normalized*** Quantity from a singe unit.
(**make-numeric-quantity** *Number*)	Make a ***normalized*** Quantity from a singe number.