Tutorial 10: Lots of Math

CS 135 Fall 2007

November 14-16, 2007

This week's tutorial covers the second half of lecture module 10 (on lambda) and the beginning of module 11. Make sure you look at the updated course notes for module 10 (http://www.student.cs.uwaterloo.ca/ cs135/handouts/10-funcabst-handout.pdf). And here's a link to the starter code for this tutorial (t10-starter.scm).

1 Lambda Calculus

The reason we use the word lambda for anonymous functions comes from the formal model of computation invented by Church and Kleene the better part of a century ago. In this model, basically everything is a lambda application, so there aren't actually any `define`s or anything like that.

It can be rather tricky to write complicated functions under these restrictions. But you now have the tools to evaluate lambda expressions using semantic substitution rules. So evaluate the following Scheme expression. For an added challenge, try to figure out what the top-level lambda function is actually doing. Don't cheat and use DrScheme!

```
((lambda (x)
   ((lambda (x y)
      (x x y))
    (lambda (x y)
      (cond ((= 1 y) 1)
            (else (* y (x x (sub1 y))))))
    x))
 3)
```

2 More Polynomial Adventures

Today, let's represent polynomials in a more traditional way, as a list of *all* coefficients from low to high order. So for example the polynomial

$$2x^5 - 3x^3 + x^2 + 8x$$

would be represented as

```
(list 0 8 1 -3 0 2)
```

It may be useful to note that, a polynomial of the form $f(x) = a + xg(x)$, where $g(x)$ is also a polynomial, would be represented as `(cons a g)`, where `g` is the representation of $g(x)$.

2.1 Scalar Multiplication

Note that multiplying a polynomial by a scalar (i.e. a number) just means multiplying each coefficient by that scalar.

Without using any recursive calls, write a function `scalar-mul` which consumes a polynomial and a number and produces the result of multiplying that polynomial by the given number.

2.2 Evaluation Function

Every polynomial is in fact a function in just one variable, which we have been calling x. But we are representing polynomials as lists, not as functions. Give a function `fun-for` to convert from the list representation of a polynomial to the functional representation. That is, `fun-for` should consume a single polynomial and produce a function which takes one argument and returns the value of the polynomial evaluated at that point.

2.3 Adder

Sometimes we might want to add to the same polynomial repeatedly. Write a function `adder` which consumes a single polynomial and produces another Scheme function. The function produced will consume another polynomial and produce the sum of the two polynomials.

3 Generative Recursion in Number Theory

3.1 Factorization

Write a function `factors` which consumes a single natural number and produces a list of all the prime factors of that number, with repeats, sorted from least to greatest. So for example, `(factor 20)` should produce `(list 2 2 5)`.

Hint: Write (and use) a helper function to find the least factor of a given number.

3.2 Chinese Remaindering

An operation of central importance in cryptography and computational number theory is called the Chinese Remainder Algorithm (CRA). Given a set of *images* of the form (v_i, m_i), where all the m_i's are relatively prime, the CRA computes the smallest positive integer congruent to each v_i modulo each m_i. In fact, the final result will always be less than the product of the m_i's, so this really produces a single new image which combines all the input images, in some sense.

In Scheme, we will represent each image with the structure

```
(define-struct img (value modulus)).
```

The starter code provides you with a function `two-cra` which consumes two images and produces a single image using the CRA. So, for example, the expression

```
(two-cra (make-img 1 3) (make-img 2 7)
```

produces the value `(make-img 16 21)`, since 16 is congruent to 1 mod 3 and 2 mod 7.

Your task is to write a function `multi-cra` which consumes a list of images and produces the single image which is the combination of all of them.

1. Write `multi-cra` using the `foldr` function. Note that every number is congruent to 0 mod 1.

2. The `two-cra`function works much more efficiently when the two moduli are close to the same size. With our first implementation of `multi-cra`

using `foldr`, this won't be the case. So write a different version of `multi-cra` that always calls `two-cra` on images with similarly-sized moduli. **Hint**: The size of the list in your recursive call should be roughly half the size of the original input list at each step.