2 Week of 02/13: natural numbers interface

In class we explored how a wide variety of functions on the natural numbers can be implemented in terms of just four "primitives":

- `zilch`, or the representation for zero
- `zilch?`, which checks equality to zero
- `succ`, which computes the successor of a natural number
- `pred`, which computes the predecessor of a (nonzero) natural number

A cool benefit of restricting ourselves to these functions when writing arithmetic functions on the natural numbers is that it gives us "implementation independence". There are many different ways of representing natural numbers, but if we write our arithmetic functions using only recursion and these four primitives, then the functions we write should be applicable to any representation of $\mathbb{N}$ that *implements this interface*, i.e. provides an implementation of these four primitives in such a way that they satisfy the same laws as the natural numbers. For instance, we could use Scheme's built-in naturals:

```
(define zilch 0)
(define zilch? (lambda (n) (= n 0)))
(define succ (lambda (n) (+ n 1)))
(define pred (lambda (n) (- n 1)))
```

or we could use nested lists (as seen in lecture), with zero being represented as the empty list `'()`, one being the list of the empty list `'(())`, two being `'((()))`, and so on, in which case our interface would look like this:

```
(define zilch '())
(define zilch? null?)
(define succ (lambda (n) (cons n '())))
(define pred car)
```

or we could use lists of varying length containing repetitions of some placeholder value (as seen in recitation), with zero again being the empty list `'()`, one being the singleton list `'(a)`, two being the list `'(a a)`, and so on, so that our interface would be the following:

```
(define zilch '())
(define zilch? null?)
(define succ (lambda (n) (cons 'a n)))
```

```
(define pred cdr)
```

or we could write a wacky (and *terribly* inefficient) implementation in which the natural number n is represented as a binary tree with depth n with the data `'z` at each of its nodes:

```
(define zilch 'z)
(define zilch? (lambda (n) (not (pair? n))))
(define succ (lambda (n) (cons n n)))
(define pred car)
```

The point is that we should write all of our arithmetic functions in such a way that they assume nothing about the actual structure being used to represent the natural numbers. Everything we write from this point onward should work no matter which of the above four interfaces we include at the beginning of our code.

First of all, let's write two functions that allow us to convert between our custom natural numbers and Racket's built-in integers. We will use these functions *only* for the purpose of readability, not for doing actual computations. Implementing arithmetic functions by converting our custom naturals to ordinary integers, performing a built-in arithmetic operation, and then converting them back would kind of defeat the point of using the interface. Here are two functions for converting between custom and built-in representations:

```
(define nat->number
  (lambda (n)
    (if (zilch? n) 0 (+ 1 (nat->number (pred n))))
  )
)

(define number->nat
  (lambda (n)
    (if (= n 0) zilch (succ (number->nat (sub1 n))))
  )
)
```

Now let's start writing some arithmetic functions. The simplest one will be addition, which we can write as follows:

```
 (define add
   (lambda (m n)
     (if (zilch? n) m (add (succ m) (pred n)))
   )
 )
```

The strategy with this function is to decrement the second argument until it is zero, all the while incrementing the first argument with each step. Once the second argument has reached zero, we simply return the first argument. What we're really taking advantage of here are the

following identities that are true for all natural numbers $m, n \in \mathbb{N}$:

$$m + 0 = m \qquad \text{base case}$$
$$m + n = (m + 1) + (n - 1) \qquad \text{recursive step}$$

By tracking the sequence of recursive calls that will be spawned by a single call to the `add` function, we can visualize how all the "mass" of the second argument gradually "accumulates" in the first argument until it's reduced to nothing. For instance, if we use the third interface listed above (the one we saw in recitation), then the sequence of recursive calls involved in computing `3 + 4` might look like this:

```
(add '(a a a) '(a a a a))
--> (add '(a a a a) '(a a a))
--> (add '(a a a a a) '(a a))
--> (add '(a a a a a a) '(a))
--> (add '(a a a a a a a) '())
--> '(a a a a a a a)
```

Now let's try writing a function to compute the product of two natural numbers. Here's a "bad" implementation that is *not* tail recursive:

```
(define mult-bad
  (lambda (m n)
    (if (zilch? m) zilch (add n (mult-bad (pred m) n)))
  )
)
```

This definition is based on the following fact about natural numbers $m, n \in \mathbb{N}$:

$$0 \times n = 0 \qquad \text{base case}$$
$$m \times n = n + (m - 1) \cdot n \qquad \text{recursive step}$$

but, as we said, this implementation is not tail-recursive. When `mult-bad` makes a tail call to itself, it doesn't simply return the result of this tail call, but *adds something* to it. This means that when evaluating this function in order to compute `3 x 2`, the intermediate steps will look something like this:

```
(mult-bad '(a a a) '(a a))
--> (add '(a a) (mult-bad '(a a) '(a a)))
--> (add '(a a) (add '(a a) (mult-bad '(a) '(a a))))
--> (add '(a a) (add '(a a) (add '(a a) (mult-bad '() '(a a)))))
--> (add '(a a) (add '(a a) (add '(a a) '())))
--> (add '(a a) (add '(a a) '(a a)))
--> (add '(a a) '(a a a a))
--> '(a a a a a a)
```

Ouch. See how the repeated additions of `'(a a)` build up quite a bit before they can be simplified at the end? This could cause a lot of overhead for the interpreter, in the form of activation

registers accumulating on the call stack for function evaluations that don't end with tail calls. Here's a better implementation of the multiplication function that makes use of a tail-recursive helper function, defined using `letrec`:

```
(define mult
  (lambda (m n)
    (letrec
     ((mult-tail (lambda (m n acc) (if (zilch? m) acc (mult-tail (pred m) n (add n acc))))))
      (mult-tail m n zilch)
    )
  )
)
```

The tail-recursive helper function `mult-tail` accumulates the successive additions in a third *accumulator* argument, to avoid the overhead of unfinished function calls. Here's what the sequence of recursive calls to `mult-tail` might look like, if it is to compute `|3 x 2|`:

```
(mult-tail '(a a a) '(a a) '())
--> (mult-tail '(a a) '(a a) (add '(a a) '()))
--> (mult-tail '(a a) '(a a) '(a a))
--> (mult-tail '(a) '(a a) (add '(a a) '(a a)))
--> (mult-tail '(a) '(a a) '(a a a a))
--> (mult-tail '() '(a a) (add '(a a) '(a a a a)))
--> (mult-tail '() '(a a) '(a a a a a a))
--> '(a a a a a a)
```

Notice how with this implementation, we never have more than one pending addition operation at a time.

Let's take a quick look at a couple more arithmetic functions. We can subtract two natural numbers (tail-recursively) as follows:

```
(define subtract
  (lambda (m n)
    (cond
      ((zilch? n) m)
      ((zilch? m) zilch)
      (else (subtract (pred m) (pred n)))
    )
  )
)
```

Note that if we try subtracting a natural number from another natural number that is smaller than it, we would end up with a negative number - and that's not something that is accounted for by our interface. Because of this, we're going to use the convention that `(subtract m n)` should return `zilch` for the result of any subtraction that would produce a negative output,

so that, for instance, computing `3 - 5` would produce zero. This definition makes use of the following facts about natural numbers $m, n \in \mathbb{N}$:

$$\begin{aligned} m - 0 &= m && \text{base case 1} \\ 0 - n &\text{ is negative if } n \neq 0 && \text{base case 2} \\ m - n &= (m - 1) - (n - 1) && \text{recursive step} \end{aligned}$$

Using this implementation, it is straightforward to write a function `leq?` that compares the size of two natural numbers, determining if its first argument is less than or equal to its second argument:

```
(define leq?
  (lambda (m n) (zilch? (subtract m n)))
)
```

Using this as a helper function, we can write a "divisibility checker" called `divides?` that tests whether its second argument is evenly divisible by its first argument, which was one of the two lab exercises:

```
(define divides?
  (lambda (d n)
    (if (leq? d n) (divides? d (subtract n d)) (zilch? n))
  )
)
```

This function is based on the following facts about natural numbers $d, n \in \mathbb{N}$ with $d \neq 0$:

$$\begin{aligned} d > n &\implies d|n \text{ iff } n = 0 && \text{base case} \\ d|n &\iff d|(n - d) && \text{recursive step} \end{aligned}$$

In other words: the only time when it's possible for a natural number to evenly divide a natural number less than itself is when that second number is equal to zero, and subtracting a number does not affect divisibility by that number. This means that if n is less than d, we know that it can only be divisible by d if it's zero, but if it's not less than d, we can iteratively subtract d from it until it has become smaller than d. We can also write a tail-recursive function to calculate the remainder of a division problem, like this:

```
(define my-remainder
  (lambda (n d)
    (if (leq? d n) (my-remainder (subtract n d) d) n)
  )
)
```

The second lab exercise for the Monday section was to write a function called `count-powers-of-two` that counts the number of perfect powers of two (starting from $2^0 = 1$) less than or equal to a given input number. Here's one possible implementation:

```
(define count-powers-of-two
  (lambda (n)
    (letrec
        (
      (helper (lambda (up acc) (if (leq? up n) (helper (add up up) (succ acc)) acc)))
            )
      (helper (succ zilch) zilch)
      )
  )
)
```

For this function, we're using a tail-recursive helper function called `helper` that repeatedly doubles its first argument through recursive calls to itself until this value surpasses *n*, tallying up the number of doubles in its second accumulator argument. (For the Wednesday section, the second exercise was to write a function called `count-powers-of-three`, which can be done by making a slight change to the above code.) The sequence of recursive calls spawned by `helper` when calculating the number of powers of two less than or equal to 9 might look like this:

```
(helper '(a) '())
--> (helper '(a a) '(a))
--> (helper '(a a a a) '(a a))
--> (helper '(a a a a a a a a) '(a a a))
--> (helper '(a a a a a a a a a a a a a a a a) '(a a a a))
--> '(a a a a)
```

If you're looking for an extra challenge, you can try writing an efficient tail-recursive function to test whether a given natural number is prime or composite, or to find the prime factorization of a given natural number (in the form of a list of primes). Give it a try!