Recursive functions

We have seen how to define and use functions in Haskell, and how to work with lists. Some very nice things happen when one combines the two using *recursion.* A recursive function definition gives the function in terms of itself. Let's start with a simple example. Suppose we want to count the number of times an element occurs in a list, that is, we want to define a function

```
elemCount :: Eq a => a -> [a] -> Int
```

so that `elemCount x aList` returns the number of appearances of `x` in the `alist`. We already know a quick way to do this using list manipulation:

```
elemCount x aList = length ( filter ( x == ) aList )
```

but we will define it recursively using *pattern matching:*

```
elemCount x [ ] = 0
elemCount x (y:ys)
  | x == y    = 1 + elemCount x ys
  | otherwise = elemCount x ys
```

Pattern matching is basically giving cases of the function. Haskell looks through the patterns and applies *the first one* that fits what it is trying to evaluate. In this case, the first line says that if the list is empty, then `elemCount x aList` is 0. If the list is nonempty, then Haskell proceeds to the next line. The pattern of the list is `(y:ys)`, where `y` is the head of the list and `ys` is the remainder of the list, which may be empty. In case the head `y` of the list matches `x`, the count should be one more than the number of appearances of `x` in `ys`. Otherwise, that is, if `not x == y`, the count is just the number of appearances of `x` in the remainder `ys`.

Let's think through how the interpreter would evaluate the expression `elemCount 3 [3, 1, 0, 3, 7]`. The first pattern that matches is `elemCount x (y:ys)` with `x` equal to 3, `y` equal to 3, and `ys` equal to `[1, 0, 3, 7]`. Since `3 == 3` is true, the expression `elemCount 3 [3, 1, 0, 3, 7]` is evaluated to `1 + elemCount 3 [1, 0, 3, 7]`. Continuing, we have

```
1 + elemCount 3 [1, 0, 3, 7]
1 + elemCount 3 [0, 3, 7]
1 + elemCount 3 [3, 7]
2 + elemCount 3 [7]
2 + elemCount 3 [ ]
2 + 0
2
```

Here is a more sophisticated example. Suppose we want a function that will sort a list of orderable things (the Prelude has a built-in `sort` function). Among the many sorting algorithms is the *insertion sort.* The idea is to take your list of things and build a new list, inserting each of the elements so that the new list is always in order. We start by writing a function that takes an element and a list that is already in order, and inserts the new element in the right place:

```
insertElement :: Ord a => a -> [a] -> [a]
insertElement x [ ] = [x]
insertElement x (y:ys)
  | x <= y = (x:y:ys)
  | otherwise = y:  ( insertElement x ys )
```

The `Ord a` signifies that the type `a` must have a concept of order. Now, the insertion sort is recursive, making use of `insertElement`:

```
iSort :: Ord a => [a] -> [a]
iSort [ ] = [ ]
iSort (x:xs) = insertElement x ( iSort xs )
```

Let's think through how the interpreter would evaluate this expression:

```
iSort [2,2,1,3]
= insertElement 2 (iSort [2,1,3])
= insertElement 2 (insertElement 2 (iSort [1,3]))
= insertElement 2 (insertElement 2 (insertElement 1 [3]))
= insertElement 2 (insertElement 2 (insertElement 1 (insertElement 3 [ ])))
= insertElement 2 (insertElement 2 (insertElement 1 ([3])))
= insertElement 2 (insertElement 2 ([1,3]))
= insertElement 2 (1 :  insertElement 2 ([3]))
= insertElement 2 (1 :  [2, 3])
= insertElement 2 ([1, 2, 3])
= 1 :  insertElement 2 ([2, 3])
= 1 :  [ 2, 2, 3])
= [ 1, 2 , 2, 3])
```

Here is another example, the `unique` function that returns the elements of a list that appear only once:

```
unique :: Eq a => [a] -> [a]
unique [ ] = [ ]
unique (x:xs)
  | elem x xs = unique ( filter ( x /= ) xs )
  | otherwise x : (unique xs)
```

Notice the Haskell notation `/=` for not equals.

By the way, `unique` has a one-line definition using filtration:

```
unique :: Eq a => [a] -> [a]
unique xs = [ x | x <- xs, length ( filter ( x == ) xs ) == 1 ]
```

Here is a famous application of Haskell recursion, the one the a Haskell salesman would show you. One of the most powerful sorting methods is the *quicksort* algorithm. In most programming languages, setting up a quicksort is a tricky little exercise. The Haskell version is a two-line function using filtration *and* recursion:

```
qSort :: Ord a => [a] -> [a]
qSort [ ] = [ ]
qSort (x:xs)
 = qSort [ y | y<- xs, y <= x ] ++ [x] ++ qSort [ y | y<- xs, y > x ]
```

Now, let's look at some of our continued fraction calculations recursively. Here is a short Haskell script:

```
import Prelude
import Ratio

integerPart ::  Rational -> Integer
integerPart x = quot (numerator ( x ) ) (denominator ( x ))

modZPart ::  Rational -> Rational
modZPart x = x - toRational( integerPart x )

cFrac ::  Rational -> [Integer]
cFrac r
  | denominator r == 1 = [numerator r]
  | r > 0 = (integerPart r) :  cFrac( 1/(modZPart r))
  | r < 0 = map (\n -> -n) (cFrac (-r))

unCFrac ::  [Integer] -> Rational
unCFrac [n] = toRational n
unCFrac (n:ns) = (toRational n) + 1/unCFrac(ns)
```

First, we import `Prelude` and `Ratio`, the latter being the (rather minimal) library for working with rational numbers. The `Ratio` library includes the `numerator` and `denominator` functions.

The `integerPart` function is defined using the integer quotient function `quot`, which is exactly like the GAP QuoInt function. In defining `modZPart`, we have to convert the integer part back to a rational number using the built-in `toRational` function.

`cFrac` is defined recursively exactly as we define it mathematically. For integral rational numbers, we just take the numerator as the unique term. A fine point here is that denominators of rational numbers are always positive. Check this by importing `Ratio` and then have Haskell evaluate the rational number $3/(-2)$ using Haskell's slightly clunky percent notation for a rational number:

```
Hugs> 3 % -2
(-3) % 2
```

To finish the definition of `cFrac`, for positive `r`, we use the mathematical definition, and for negative `r`, we reduce to the positive case. `unCFrac` is also just defined the way we would do it mathematically.

There are many uses of recursion. Here is one common device, for situations when we want to repeat something until it no longer has any effect. The problem we will consider is the normal form of finitely generated abelian groups. There are a couple of standard forms for finitely generated abelian groups. One is $\mathbb{Z} \oplus \cdots \oplus \mathbb{Z} \oplus \mathbb{Z}/m_1 \oplus \mathbb{Z}/m_2 \oplus \cdots \oplus \mathbb{Z}/m_k$ where $m_{i+1} | m_i$ for each $i > 1$. Thus, for example, you can write $\mathbb{Z}/2 \oplus \mathbb{Z}/4 \oplus \mathbb{Z}/8 \oplus \mathbb{Z}/9 \oplus \mathbb{Z}/10$ as $\mathbb{Z}/360 \oplus \mathbb{Z}/4 \oplus \mathbb{Z}/2 \oplus \mathbb{Z}/2$. How can we attain this form algorithmically?

A special case of the normal form is $\mathbb{Z}/a \oplus \mathbb{Z}/b \cong \mathbb{Z}/\operatorname{lcm}(a,b) \oplus \mathbb{Z}/\gcd(a,b)$. There are, of course, many ways to see this isomorphism. Here is a way that will fit in well with some later examples. If an abelian group A is generated by n elements, then there is a surjective homomorphism $\mathbb{Z}^n \to A$. Math shows that its kernel is a free abelian group of some rank $m \leq n$. We can think of this as an exact sequence

$$0 \to \mathbb{Z}^m \xrightarrow{\phi} \mathbb{Z}^n \to A \to 0 \ .$$

If there are bases $\{v_1, \ldots, v_n\}$ of $\mathbb{Z}^m$ and $\{w_1, \ldots, w_n\}$ of $\mathbb{Z}^n$ such that $\phi(v_i) = n_i \cdot w_i$ for $i \leq m$, then $A \cong \mathbb{Z}/n_1 \oplus \cdots \oplus \mathbb{Z}/n_m \oplus \mathbb{Z} \oplus \cdots \oplus \mathbb{Z}$, and if A has this form then we can easily construct a corresponding $\phi\colon \mathbb{Z}^m \to \mathbb{Z}^n$ just by sending $\mathbb{Z}^n \to A$ taking e_i to a generator of $\mathbb{Z}/m_i$, and $\phi(e_i) = m_i e_i$, where the e_i are the standard basis vectors. In the particular case of $\mathbb{Z}/a \oplus \mathbb{Z}/b$, we have

$$0 \to \mathbb{Z}^2 \xrightarrow{\phi} \mathbb{Z}^2 \to \mathbb{Z}/a \oplus \mathbb{Z}/b \to 0 \ ,$$

where $\phi(e_1) = ae_1$ and $\phi(e_2) = be_2$. Its matrix with respect to the standard bases is $\begin{pmatrix} a & 0 \\ 0 & b \end{pmatrix}$. Now, we can do any row and column operations on this matrix, achieving a matrix for ϕ with respect to some other bases. In particular, putting $d = \gcd(a,b)$, we have

$$\begin{pmatrix} a & 0 \\ 0 & b \end{pmatrix} \to \begin{pmatrix} a & d \\ 0 & b \end{pmatrix} \to \begin{pmatrix} a & d \\ (-b/d)a & 0 \end{pmatrix} \to \begin{pmatrix} 0 & d \\ (-b/d)a & 0 \end{pmatrix} \to \begin{pmatrix} -\operatorname{lcm}(a,b) & 0 \\ & \gcd(a,b) \end{pmatrix} \ ,$$

the last by a column interchange.

So here is our algorithm:

1. Represent an initial $\mathbb{Z} \oplus \cdots \mathbb{Z} \oplus \mathbb{Z}/a_1 \oplus \mathbb{Z}/a_2 \oplus \cdots \oplus \mathbb{Z}/a_k$ as a list $[0, \ldots, 0, a_1, \ldots, a_k]$.
2. Do replacement moves that replace pairs $[a_i, a_{i+1}]$ by $[\operatorname{lcm}(a_i, a_{i+1}), \gcd(a_i, a_{i+1})]$.
3. Whenever some a_{i+1} does not divide a_i, a (nontrivial) replacement move will be possible.
4. A replacement move decreases $[a_1, \ldots, a_k]$ in the lexicographical ordering, so this process cannot continue forever. Therefore it must terminate with a_{i+1} dividing a_i for every i.

We may assume that all the a_i are nonzero, and the first step is to write a "combing" process that makes one round of moves:

```
comb :: [Int] -> [Int]
-- Assumes that all integers are nonzero, since gcd 0 0 is undefined
comb [ ] = [ ]
comb [n] = [n]
comb (m:n:rest) = (lcm m n) :  (comb (gcd m n:  rest))
```

Here is what `comb` does in the case of our earlier example $\mathbb{Z}/2 \oplus \mathbb{Z}/4 \oplus \mathbb{Z}/8 \oplus \mathbb{Z}/9 \oplus \mathbb{Z}/10$:

```
Abelian> comb [2,4,8,9,10]
[4,8,18,10,1]
Abelian> comb $$
[8,36,10,2,1]
Abelian> comb $$
[72,20,2,2,1]
Abelian> comb $$
[360,4,2,2,1]
```

giving $\mathbb{Z}/360 \oplus \mathbb{Z}/4 \oplus \mathbb{Z}/2 \oplus \mathbb{Z}/2 \oplus \mathbb{Z}/1$, with $\mathbb{Z}/1$ the trivial quotient $\mathbb{Z}/(1 \cdot \mathbb{Z}) = \{0\}$. Notice the convenient hugs interpreter notation `$$` for "the result of the previous evaluation". Now, we use guards to continue our recursion until replacement moves have no effect:

```
normalForm :: [Int] -> [Int]
normalForm list
  | nonzeros == comb nonzeros = nonzeros ++ zeros
  | otherwise = zeros ++ normalForm ( (comb nonzeros) + zeros )
  where
  zeros = [ x | x <- list, x == 0 ]
  nonzeros = [ x | x <- list, x /= 0, x /= 1 ]
```

A final note is that the `$$` notation does not work in scripts, because there is no "order" of the statements. In evaluating functions, Haskell might carry out the steps in any number of different possible orders, so there is no "previous" statement when one executes a script.