Functional Programming

Paul Mellies

Lesson 7: Recursion and Pattern Matching
Guideline for writing recursive functions

In order to write a recursive function in Haskell, one should follow these guidelines:

1. Identify the end goal(s) of the problem.
2. Determine what happens when a goal is reached.
3. List all alternate possibilities.
4. Determine your "rinse and repeat" process.
5. Ensure that each alternative brings you closer to the goal.

We will see how to construct recursive functions in Haskell, using pattern-matching.
Illustration: the euclidian algorithm

Euclid as seen by the Renaissance (colored woodcut, 1584)
A millenium of prominent mathematicians in the Greco-Roman world
Illustration: the euclidian algorithm

The euclidian algorithm was described in Euclid’s Elements around 300 BC and is one of the oldest numeric algorithms in existence.

The algorithm is designed to compute recursively

\[
\text{the greatest common divisor (gcd) of two numbers } a \text{ and } b
\]

The algorithm works as follows:

1. Start with the two numbers \( a \) and \( b \).
2. If you divide \( a \) by \( b \) and the remainder \( r \) is equal to 0 then clearly \( b \) is the gcd.
3. Otherwise, replace the value of \( a \) by the value of the remainder \( r \) and then swap the values of \( a \) and \( b \),
4. Repeat the whole procedure from step 1. until the division \( a/b \) of \( a \) by \( b \) has its remainder \( r \) equal to 0.
Illustration: Euclid’s algorithm

Illustration: we apply the algorithm to the pair $a = 20$ and $b = 16$

1. $a = 20$ and $b = 16$,
2. $a/b = 20/16 = 1$ with remainder 4,
3. $a = 16$ and $b = 4$,
4. $a/b = 16/4 = 4$ with remainder 0,
5. $\text{gcd} = 4$

Exercise. Show that the algorithm necessarily terminates and provides the gcd.
Euclid’s algorithm in Haskell

The algorithm is implemented as follows in Haskell:

```haskell
myGCD a b = if remainder == 0
  then b
  else myGCD b remainder
  where remainder = a `mod` b
```

Here, the binary function \texttt{mod} computes the remainder of the division \texttt{a/b}
of its first argument \texttt{a} by its second argument \texttt{b}.

As explained in the previous lesson, \texttt{‘mod‘} is the corresponding infix operator.
Pattern matching

The `sayAmount` function takes a number and returns a string.

The function can be defined using the `case` operator:

```haskell
sayAmount n = case n of
  1 -> "one"
  2 -> "two"
  n -> "a bunch"
```

or using **pattern matching** in the following way:

```haskell
sayAmount 1 = "one"
sayAmount 2 = "two"
sayAmount n = "a bunch"
```

Pattern-matching looks like three separate definitions, each for one of the possible arguments. Note that, just like `case`, pattern-matching treats the options in order.
Pattern matching with wildcards

The `isEmpty` function is defined by pattern-matching.

It takes a list as argument and returns the boolean value `True` when the list is empty and the boolean value `False` otherwise. The function can be defined in this way:

```plaintext
isEmpty [] = True
isEmpty aList = False
```

It is common practice however to use `_` as a `wildcard` for values

```plaintext
isEmpty [] = True
isEmpty _ = False
```

which do not appear in the body of the function. The main advantage of using wildcards is that it makes the code for pattern-matching is generally easier to read.
General form of pattern-matching

In general, one uses more sophisticated forms of pattern-matching where the variables appearing in the patterns can then reappear to define the function.

Illustration: consider this version of the head function defined by pattern-matching

\[
\text{myHead} \ (x:xs) = x
\]

where the variable \( x \) appears in the pattern \((x:xs)\) as well as in the definition of the function \text{myHead} as the first element \( x \) of the list.

Note that a popular convention is to use the variable \( x \) to represent a single value and the variable \( xs \) to represent a list of values.
The Haskell code

\[ \text{myHead} \ [1,2,3] \]

can be rewritten as...

\[ \text{myHead} \ (1:\ [2,3]) \]

\[ \text{myHead} \ (x:xs) = x \]

Here, one should remember that the list \([1,2,3]\) is syntactic sugar for \(1:2:3:[]\).
Pattern-matching with wildcards

The previous head function may be refined and extended into

\[
\begin{align*}
\text{myHead} \ (x:xs) &= x \\
\text{myHead} \ [x] &= \text{error} \ "\text{No Head for empty list}\" \\
\end{align*}
\]

where the purpose of the error function is to throw an error.

Note that the head function can be equivalently written using a wildcard:

\[
\begin{align*}
\text{myHead} \ (x:_x) &= x \\
\text{myHead} \ [x] &= \text{error} \ "\text{No Head for empty list}\" \\
\end{align*}
\]

The resulting code does not need to introduce the variable xs and is thus more economic and easier to read.