1 Polynomials

1.1 Numeral Systems

A numeral system consists of a finite set of digits, D, and a base, b, for which $b = |D|$, *i.e.* b is the cardinality of D. The binary system, for instance, uses the digits $D = \{0, 1\}$. The cardinality of D in this case, hence, is 2. The decimal system uses the digits $D = \{0 \dots 9\}$ and, thus, has the base $b = 10$. The hexadecimal system uses the digits $D = \{0 \dots 15\}$, often given as $D = \{0 \dots 9, a, b, c, d, e, f\}$, and, therefore, has the base $b = 16$.

Numbers in any numeral system are usually represented as strings of digits. The string

$$10101010,$$

for instance, may represent a number in the binary system. (It could be a number in decimal format, too, though.) The string

$$170,$$

by contrast, cannot be a binary number, because it contains the digit 7, which is not element of D in the binary system. It can represent a decimal (or a hexadecimal number). The string

$$aa,$$

can represent a number in the hexadecimal system (but not one of in the binary or decimal system).

We interpret such a string, *i.e.* convert it to the decimal system, by rewriting it as a formula of the form:

$$a_n b^n + a_{n-1} b^{n-1} + \cdots + a_0 b^0,$$

where a_i are the digits that appear in the string, b is the base and n is position of the left-most digit starting to count with 0 on the right-hand side of the string. The string 10101010 in binary notation,hence, is interpreted as

$$1 \times 2^7 + 0 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0,$$

which can be simplified to

$$2^7 + 2^5 + 2^3 + 2,$$

which, in its turn, is

$$128 + 32 + 8 + 2 = 170.$$

The string 170 in decimal notation is interpreted as

$$10^2 + 7 \times 10 = 170.$$

Interpreting a string in the notation it is written in yields just that string. The string aa in hexadecimal notation is interpreted as

$$a \times 16 + a.$$

The digit a, however, is just 10. We, hence, get the equation

$$10 \times 16 + 10 = 160 + 10 = 170.$$

What do we get, when we relax some of the constraints defining a numeral system? Instead of using a finite set of digits, we could use a number field, F, (finite or infinite) so that any member of that field qualifies as coefficient in the formulas we used above to interpret numbers in the decimal system. We would then relax the rule that the base must be the cardinality of the field. Instead, we allow any member x of the field to serve as a base. Formulas we get from those new rules would follow the recipe:

$$a_n x^n + a_{n-1} x^{n-1} + \cdots + a_0 x^0$$

or shorter:

$$\sum_{i=0}^{n} a_i x^i$$

with $a_i, x \in F$.

Such beasts are indeed well-known. They are very prominent, in fact, and their name is *polynomial*.

The name *poly*nomial stems from the fact that polynomials may be composed of many terms; a monomial, by contrast, is a polynomial that consists of only one term. For instance,

$$5x^2$$

is a monomial. A binomial is a polynomial that consists of two terms. This is an example of a binomial:

$$x^5 + 2x.$$

There is nothing special about monomials and binomials, at least nothing that would affect their definition as polynomials. Monomials and binomials are just polynomials that happen to have only one or, respectively, two terms.

Polynomials share many properties with numbers. Like numbers, arithmetic, including addition, subtraction, multiplication and division as well as exponentiation, can be defined over polynomials. In some cases, numbers reveal their close relation to polynomials. The binomial theorem states, for instance, that a product of the form

$$(a + b)(a + b)$$

translates to a formula involving binomial coefficients:

$$a^2 + 2ab + b^2.$$

We can interpret this formula as the product of the polynomial $x + a$:

$$(x + a)(x + a),$$

which yields just another polynomial:

$$x^2 + 2ax + a^2$$

Let us replace a for the number 3 and fix $x = 10$. We get:

$$(10 + 3)(10 + 3) = 10^2 + 2 \times 3 \times 10 + 3^2 = 100 + 60 + 9 = 169, \tag{1.1}$$

which is just the result of the multiplication 13×13. Usually, it is harder to recognise this kind of relations numbers have with the binomial theorem (and, hence, with polynomials), because most binomial coefficients are too big to be represented by a single-digit number. Already in the product 14×14, the binomial coefficients are hidden. Let us look at this multiplication treated as the polynomial $(x + a)$ with $x = 10$ and $a = 4$:

$$(10 + 4)(10 + 4) = 10^2 + 2 \times 4 \times 10 + 4^2 = 100 + 2 \times 40 + 16.$$

When we look at the resulting number, we do not recognise the binomial coefficient anymore – they are *carried* away: $100 + 2 \times 40 + 16 = 100 + 80 + 16 = 196$.

Indeed, polynomials are not numbers. Those are different concepts. Another important difference is that polynomials do not establish a clear order. For any two distinct numbers, we can clearly say which of the two is the greater and which is the smaller one. We cannot decide that based on the formula of the polynomial alone. One way to decide quickly which of two numbers is the grater one is to look at the number of their digits. The one with more digits is necessarily the greater one. In any numeral system it holds that:

$$a_3b^3 + a_2b^2 + a_1b + a_0 > c_2b^2 + c_1b + c_0$$

independent of the values of the as and the cs. For polynomials, this is not true. Consider the following example:

$$x^3 + x^2 + x + 1 > 100x^2?$$

For $x = 10$, the left-hand side of the inequation is $1000 + 100 + 10 + 1 = 1111$; the right-hand side, however, is $100 \times 100 = 10000$.

In spite of such differences, we can represent polynomials very similar to how we represented numbers, namely as a list of coefficients. This is a valid implementation in Haskell:

```
type Poly a = P [a]
  deriving (Show)
```

We add a safe constructor:

```
poly :: (Eq a, Num a) ⇒ [a] → Poly a
poly [] = error "not a polynomial"
poly as = P (cleanz as)

cleanz :: (Eq a, Num a) ⇒ [a] → [a]
cleanz xs = reverse $ go (reverse xs)
  where go []     = []
        go [0]    = [0]
        go (0 : xs) = go xs
        go xs     = xs
```

The constructor makes sure that the resulting polynomial has at least one coefficient and that all the coefficients are actually numbers and comparable for equality. The function *cleanz* called in the constructor removes leading zeros (which are redundant), just as we did when we defined natural numbers. But note that we reverse, first, the list of coefficients passed to *go* and, second, the result of *go*. This means that we store the coefficients from left to right in ascending order. Usually, we write polynomials out in descending order of their weight, *i.e.*

$$x^n + x^{n-1} + \cdots + x^0.$$

But, here, we store them in the order:

$$x^0 + x^1 + \cdots + x^{n-1} + x^n.$$

The following function gets the list of coefficients back:

```
coeffs :: Poly a → [a]
coeffs (P as) = as
```

Here is a function to pretty-print polynomials:

```
pretty :: (Num a, Show a, Eq a) ⇒ Poly a → String
pretty p = go (reverse $ weigh p)
  where go [] = ""
        go ((i, c) : cs) = let x | i ≡ 0       = ""
                                 | i ≡ 1       = "x"
                                 | otherwise = "x^" ++ show i
                           t | c ≡ 1       = x
                             | otherwise = show c ++ x
                           o | null cs     = ""
                             | otherwise = " + "
                           in if c ≡ 0 then go cs else t ++ o ++ go cs
weigh :: (Num a) ⇒ Poly a → [(Integer, a)]
weigh (P []) = []
weigh (P as) = (zip [0..] as)
```

The function demonstrates how we actually interpret the list of coefficients. We first *weigh* them by zipping the list of coefficients with a list of integers starting at 0. One could say: we count the coefficients. Note that we start with 0, so that the first coefficient gets the weight 0, the second gets the weight 1 and so on. That, again, reflects our descending ordering of coefficients.

The reversed weighted list is then passed to *go*, which does the actual printing. We first determine the substring describing x: if i, the weight, is 0, we do not want to write the x, since $x^0 = 1$. If $i = 1$, we just write x. Otherwise we write x^i.

Then we determine the term composed of coefficient and x. If the coefficient, c is 1, we just write x; otherwise, we concatenate c with x. Note, however, that we later consider an additional case, namely, when $c = 0$. In this case, we ignore the whole term.

We still consider the operation. If the remainder of the list is *null*, *i.e.* we are now handling the last term, o is the empty string. Otherwise, it is the plus symbol. Here is room for improvement: when the coefficient is negative, we do not really need the operation, since we then write $+ -cx$. Nicer would be to write only $-cx$.

Finally, we put everything together concatenating a string composed of term, operation and *go* applied on the remainder of the list.

Here is a list of polynomials and how they are represented in our Haksell type:

$x^2 + x + 1$	*poly* $[1, 1, 1]$
$5x^5 + 4x^4 + 3x^3 + 2x^2 + x$	*poly* $[0, 1, 2, 3, 4, 5]$
$5x^4 + 4x^3 + 3x^2 + 2x + 1$	*poly* $[1, 2, 3, 4, 5]$
$5x^4 + 3x^2 + 1$	*poly* $[1, 0, 3, 0, 5]$

An important concept related to polynomials is the *degree*. The degree is a measurement of the *size* of the polynomial. In concrete terms, it is the greatest exponent in the polynomial. For us, it is the weight of the right-most element in the polynomial or, much simpler, the length of the list of coefficients minus one – since, we start with zero! The following function computes the degree of a given polynomial:

```
degree :: Poly a → Int
degree (P as) = length as − 1
```

Note, by the way, that polynomials of degree 0, those with only one trivial term, are just constant numbers.

Finally, here is a useful function that creates random polynomials with *Natural* coefficients:

```
randomPoly :: Natural → Int → IO (Poly Natural)
randomPoly n d = do
  cs ← cleanz <$> mapM (\_ → randomCoeff n) [1 .. d]
  if length cs < d then randomPoly n d
    else return (P cs)
randomCoeff :: Natural → IO Natural
randomCoeff n = randomNatural (0, n − 1)
```

The function receives a *Natural* and an *Int*. The *Int* indicates the degree of the polynomial we want to obtain. The *Natural* is used to restrict the size of the coefficients we want to see in the polynomial. In *randomCoeff*, we use the *randomNatural* defined in the previous chapter to generate a random number between 0 and $n-1$. You might suspect already where that will lead us: to polynomials modulo some number. But before we get there, we will study polynomial arithmetic.

1.2 Polynomial Arithmetic

1.3 The Difference Engine

1.4 Factoring Polynomials

1.5 Roots

1.6 Vieta's Formula

1.7 The Method of partial Fractions

1.8 Polynomials and Binomial Coefficients

1.9 Generationfunctionology 1

1.10 The closed Form of the Fibonacci Sequence

$$G(x) = F_0 + F_1 x + F_2 x^2 + F_3 x^3 + \dots \tag{1.2}$$

$$G(x) = 0 + x + x^2 + 2x^3 + 3x^4 + 5x^5 + 8x^6 + \dots \tag{1.3}$$

$$xG(x) = F_0 x + F_1 x^2 + F_2 x^3 + F_3 x^4 + \dots \tag{1.4}$$

$$x^2 G(x) = F_0 x^2 + F_1 x^3 + F_2 x^4 + F_3 x^5 + \dots \tag{1.5}$$

$$G(x) - xG(x) - x^2 G(x) = (1 - x - x^2)G(x). \tag{1.6}$$

$$\begin{array}{lllllll} (1-x-x^2)G(x) = & (F_0 & +F_1 x & +F_2 x^2 & +F_3 x^3 & +\dots) & - \\ & (& F_0 x & +F_1 x^2 & +F_2 x^3 & +\dots) & - \\ & (& & +F_0 x^2 & +F_1 x^3 & +\dots) & \end{array}$$

$$
\begin{aligned}
(1 - x - x^2)G(x) = \quad & F_0 + (F_1 - F_0)x \\
& + (F_2 - F_1 - F_0)x^2 \\
& + (F_3 - F_2 - F_1)x^3 \\
& + \dots
\end{aligned}
$$

$$(1 - x - x^2)G(x) = x. \tag{1.7}$$

$$G(x) = \frac{x}{1 - x - x^2}. \tag{1.8}$$