Contents

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 is 2 and therefore $b = 2$. 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 or hexadecimal format, too. 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 number in the decimal (or the hexadecimal) system. The string

$$aa,$$

can represent a number in the hexadecimal system but not one 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:

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

where a_i are the digits that appear in the string, b is the base and n the 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 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 corresponds to 10 in the decimal system. We, therefore, 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 a 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 and their name is *polynomials*.

The name *poly*nomial stems from the fact that they 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':

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

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 of the two. 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. This is because the base b is fixed. In the case of 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:

```haskell
data Poly a = P [a]
  deriving (Show)
```

We add a safe constructor:

```haskell
poly :: (Eq a, Num a) => [a] -> Poly a
poly [] = P [0]
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:

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

We will soon see why that is an advantage.

The following function gets the list of coefficients back:

```haskell
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 with 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 measure 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 number of coefficients 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

We start with addition and subtraction, which, in German, are summarised by the beautiful word *strichrechnung* meaning literally "dash calculation" as opposed to *punktrechnung* or "dot calculation", which would be multiplication and division.

Polynomial *strichrechnung* is easy. Key is to realise that the structure of polynomials is already defined by *strichrechnung*: it is composed of terms each of which is a product of some number and a power of x. When we add (or subtract) two polynomials, we just merge them keeping order according to the exponents of their terms and add (or subtract) terms with equal exponents:

$$
\begin{array}{rrcrcccr}
 & ax^n & + & bx^{n-1} & + & \ldots & + & c \\
+ & dx^n & + & ex^{n-1} & + & \ldots & + & f \\
= & (a+d)x^n & + & (b+e)x^{n-1} & + & \ldots & + & c+f
\end{array}
\tag{1.2}
$$

With our polynomial representation, it is easy to implement this kind of operation. One might think it was designed especially to support addition and subtraction. Here is a valid implementation:

```
add :: (Num a, Eq a) => Poly a -> Poly a -> Poly a
add = strich (+)

sub :: (Num a, Eq a) => Poly a -> Poly a -> Poly a
sub = strich (-)

strich :: (Num a, Eq a) => (a -> a -> a) -> Poly a -> Poly a -> Poly a
strich o (P x) (P y) = P (strichlist o x y)

zeros :: Num a => Int -> [a]
zeros i = take i (repeat 0)

strichlist :: (Num a, Eq a) => (a -> a -> a) -> [a] -> [a] -> [a]
strichlist o xs ys = let   us | xd >= yd  = xs
                              | otherwise = xs ++ zeros (yd - xd)
                           vs | yd >= yd  = ys
                              | otherwise = ys ++ zeros (xd - yd)
                     in cleanz (go us vs)
  where xd = length xs
        yd = length ys
        go [] bs           = bs
        go as []           = as
        go (a : as) (b : bs) = a `o` b : go as bs
```

A bit tricky might be the use of *zeros*. The function generates a sequence of zeros of size i. We use it to add 0 coefficients at the end of the shorter coefficient list (if any). For addition this is not relevant (because we would just add the coefficients of the longer one to the end of the list). For subtraction, however, it is relevant, since we need to compute the additive inverse of the extra coefficients in the longer list. *zeros* does the trick.

Based on addition, we can also implement *sum* for polynomials:

```
sump :: (Num a, Eq a) => [Poly a] -> Poly a
sump = foldl' add (P [0])
```

Here is one more function that might be useful later on; it folds *strichlist* on a list of lists of coefficients:

```
strichf :: (Num a, Eq a) => (a -> a -> a) -> [[a]] -> [a]
strichf o = foldl' (strichlist o) []
```

What if we add a polynomial to itself more than once? With numbers, that would be multiplication. With polynomials, this is a bit different. There is in fact an operation that is between *strichrechnung* and *punktrechnung*, namely *scaling*. Scaling maps multiplication by n, for n some integer, on all coefficients and, as such, corresponds to adding a polynomial n times to itself:

$$
\begin{aligned}
&\mathit{scale} :: (\mathit{Num}\ a) \Rightarrow a \rightarrow \mathit{Poly}\ a \rightarrow \mathit{Poly}\ a\\
&\mathit{scale}\ n\ (P\ \mathit{cs}) = \mathit{poly}\ (\mathit{map}\ (n*)\ \mathit{cs})
\end{aligned}
$$

Punktrechnung, *i.e.* multiplication and division, is a bit more complex – because of the distribution law. Let us start with the simple case where we distribute a monomial over a polynomial:

$$
\begin{aligned}
&\mathit{mul1} :: \mathit{Num}\ a \Rightarrow (a \rightarrow a \rightarrow a) \rightarrow \mathit{Int} \rightarrow [\,a\,] \rightarrow a \rightarrow [\,a\,]\\
&\mathit{mul1}\ \ o\ i\ \mathit{cs}\ x = \mathit{zeros}\ i +\!\!\!+ [\,c\ \text{`}o\text{`}\ x \mid c \leftarrow \mathit{cs}\,]
\end{aligned}
$$

The function *mul1* takes a single term (the monomial) and distributes it over the coefficients of a polynomial using the operation o. Each term in the polynomial is combined with the single term. This corresponds to the operation:

$$
\begin{array}{ccccccccc}
dx^m & \times & (& ax^n & + & bx^{n-1} & + & \dots & + & c)\\
 & = & & adx^{m+n} & + & bdx^{n-1+m} & + & \dots & + & cdx^m
\end{array} \tag{1.3}
$$

The function *mul1* receives on more parameter, namely the *Int* i and uses it to generate a sequence of zeros that is put in front of the resulting coefficient list. As we will see shortly, the list of zeros reflects the weight of the single term. In fact, we do not implement the manipulation of the exponents we see in the abstract formula directly. Instead, the addition $+m$ is implicitly handled by placing m zeros at the head of the list resulting in a new polynomial of degree $m + d$ where d is the degree of the original polynomial. A simple example:

$$
5x^2 \times (4x^3 + 3x^2 + 2x + 1) = 20x^5 + 15x^4 + 10x^3 + 5x^2
$$

would be:

$\mathit{mul1}\ 2\ [1, 2, 3, 4]\ 5$

which is:

$\mathit{zero}\ 2 +\!\!\!+ (5 * [1, 2, 3, 4]) = [0, 0, 5, 10, 15, 20]$

We, hence, would add 2 zeros, since 2 is the degree of the monomial.

Now, when we multiply two polynomials, we need to map all terms in one of the polynomials on the other polynomial using *mul1*. We further need to pass the weight of the individual terms of the first polynomial as the *Int* parameter of *mul1*. What we want to do is:

$[\mathit{mul1}\ (*)\ i\ (\mathit{coeffs}\ p1)\ p \mid (i,p) \leftarrow \mathit{zip}\ [0\,..]\ (\mathit{coeffs}\ p2)]$.

What would we get applying this formula on the polynomials, say, $[1,2,3,4]$ and $[5,6,7,8]$? Let us have a look:

$[\mathit{mul1}\ (*)\ i\ ([5,6,7,8])\ p \mid (i,p) \leftarrow \mathit{zip}\ [0\,..]\ [1,2,3,4]]$
$[[5,6,7,8],[0,10,12,14,16],[0,0,15,18,21,24],[0,0,0,20,24,28,32]]$.

We see a list of four lists, one for each coefficient of $[1,2,3,4]$. The first list is the result of distributing 1 over all the coefficients in $[5,6,7,8]$. Since 1 is the first element, its weight is 0: no zeros are put before the resulting list. The second list results from distributing 2 over $[5,6,7,8]$. Since 2 is the second element, its weight is 1: we add one zero. The same process is repeated for 3 and 4 resulting in the third and fourth result list. Since 3 is the the third element, the third resulting list gets two zeros and, since 4 is the fourth element, the fourth list gets three zeros.

How do we transform this list of lists back into a single list of coefficients? Very easy: we add them together using *strichf*:

$\mathit{strichf}\ (+)\ [[5,6,7,8],[0,10,12,14,16],[0,0,15,18,21,24],[0,0,0,20,24,28,32]]$

which is

$[5,16,34,60,61,52,32]$.

This means that

$$\begin{aligned} &(4x^3+3x^2+2x+1)\times(8x^3+7x^2+6x+5)\\ =\ &32x^6+52x^5+61x^4+60x^3+34x^2+16x+5. \end{aligned} \tag{1.4}$$

Here is the whole algorithm:

```
mul :: (Show a, Num a, Eq a) => Poly a -> Poly a -> Poly a
mul p1 p2 | d2 > d1   = mul p2 p1
          | otherwise = P (strichf (+) ms)
  where d1 = degree p1
        d2 = degree p2
        ms = [mul1 (*) i (coeffs p1) p | (i,p) <- zip [0..] (coeffs p2)]
```

On top of multiplication, we can implement power. We will, of course, not implement a naïve approach based on repeated multiplication alone. Instead, we will use the *square-and-multiply* approach we have already used before for numbers. Here is the code:

$$
\begin{array}{l}
\mathit{powp} :: (\mathit{Show}\ a, \mathit{Num}\ a, \mathit{Eq}\ a) \Rightarrow \mathit{Natural} \rightarrow \mathit{Poly}\ a \rightarrow \mathit{Poly}\ a \\
\mathit{powp}\ f\ \mathit{poly} = \mathit{go}\ f\ (P\ [1])\ \mathit{poly} \\
\quad \textbf{where}\ \mathit{go}\ 0\ y\ \_ = y \\
\qquad\qquad \mathit{go}\ 1\ y\ x = \mathit{mul}\ y\ x \\
\qquad\qquad \mathit{go}\ n\ y\ x \mid \mathit{even}\ n \quad\ = \mathit{go}\ (n\ \text{`}\mathit{div}\text{`}\ 2)\ y \qquad\quad (\mathit{mul}\ x\ x) \\
\qquad\qquad\qquad\qquad \mid \mathit{otherwise} = \mathit{go}\ ((n-1)\ \text{`}\mathit{div}\text{`}\ 2)\ (\mathit{mul}\ y\ x) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad (\mathit{mul}\ x\ x)
\end{array}
$$

The function *powp* receives a natural number, that is the exponent, and a polynomial. We kick off by calling *go* with the exponent f, a base polynomial $P\ [1]$, *i.e.* unity, and the polynomial we want to raise to the power of f. If $f = 0$, we are done and return the base polynomial. This reflects the case $x^0 = 1$. If $f = 1$, we multiply the base polynomial by the input polynomial. It we have called *powp* with one, this has no effect, since the base polynomial, in this case, is unity.

Otherwise, if the exponent is even, we halve it, pass the base polynomial on and square the input. Otherwise, if the exponent is odd, we subtract one form the exponent and half the result and pass the product of the base polynomial and the input on instead of the base polynomial as it is and, of course, still square the input.

This implementation differs a bit from the implementation we presented before for numbers, but it implements the same algorithm.

Here is a simple example: we raise the polynomial $x + 1$ to the power of 5. In the first round, we compute

$\mathit{go}\ 5\ (P\ [1])\ (P\ [1,1])$,

which, since 5 is odd, results in

$\mathit{go}\ 2\ (P\ [1,1])\ (P\ [1,2,1])$.

This, in its turn, results in

$\mathit{go}\ 1\ (P\ [1,1])\ (P\ [1,4,6,4,1])$.

This is the final step and results in

$\mathit{mul}\ (P\ [1,1])\ (P\ [1,4,6,4,1])$,

which is

$P\ [1,5,10,10,5,1]$,

the polynomial $x^5 + 5x^4 + 10x^3 + 10x^2 + 5x + 1$.

You might have noticed that the different states of the algorithm given in our Haskell notation shows the binomial coefficients $\binom{n}{k}$ for $n = 1$, $n = 2$, $n = 4$ and $n = 5$. We never see $n = 3$, which would be $P\ [1,3,3,1]$, because we leave the multiplication $\mathit{mul}\ (P\ [1,1])\ (P\ [1,2,1])$ out. For this specific case with exponent 5, leaving out this step is where square-and-multiply is more efficient than multiplying five times. With growing exponents, the saving quickly grows to a significant order.

Division is, as usual, still more complicated than multiplication. But it is not too different from number division. First, we define polynomial division as Euclidean division, that is we search the solution for the equation

$$\frac{a}{b} = q + r \tag{1.5}$$

where $r < b$ and $bq + r = a$.

The manual process is as follows: we divide the first term of a by the first term of b. The quotient goes to the result; then we multiply it by b and set a to a minus that result. Now we repeat the process until the degree of a is less than that of b.

Here is an example:

$$\frac{4x^5 - x^4 + 2x^3 + x^2 - 1}{x^2 + 1}.$$

We start by dividing $4x^5$ by x^2. The quotient is $4x^3$, which we add to the result. We multiply: $4x^3 \times (x^2 + 1) = 4x^5 + 4x^3$ and subtract the result from a:

$$\begin{array}{cccccccccc} & 4x^5 & - & x^4 & + & 2x^3 & + & x^2 & - & 1 \\ - & 4x^5 & & & + & 4x^3 & & & & \\ = & & - & x^4 & - & 2x^3 & + & x^2 & - & 1 \end{array} \tag{1.6}$$

We continue with $-x^4$ and divide it by x^2, which is $-x^2$. The overall result now is $4x^3 - x^2$. We multiply $-x^2 \times (x^2 + 1) = -x^4 - x^2$ and subtract that from what remains from a:

$$\begin{array}{cccccccccc} & - & x^4 & - & 2x^3 & + & x^2 & - & 1 \\ - & - & x^4 & & & - & x^2 & & \\ = & & & - & 2x^3 & + & 2x^2 & - & 1 \end{array} \tag{1.7}$$

We continue with $-2x^3$, which, divided by x^2 is $-2x$. This goes to the result: $4x^3 - x^2 - 2x$. We multiply $-2x \times (x^2 + 1) = -2x^3 - 2x$ and subtract:

$$\begin{array}{cccccccccc} & - & 2x^3 & + & 2x^2 & + & & - & 1 \\ - & - & 2x^3 & & & - & 2x & & \\ = & & & & 2x^2 & + & 2x & - & 1 \end{array} \tag{1.8}$$

We continue with $2x^2$, which, divided by x^2 is 2. We multiply $2 \times (x^2 + 1) = 2x^2 + 2$ and subtract:

$$
\begin{array}{rrrrrrr}
 & 2x^2 & + & 2x & - & 1 \\
- & 2x^2 & & & + & 2 \\
= & & & 2x & - & 3
\end{array}
\tag{1.9}
$$

The result now is $4x^3 - x^2 - 2x + 2$. We finally have $2x - 3$, which is smaller in degree than b. The result, hence, is $(4x^3 - x^2 - 2x + 2, 2x - 3)$.

Here is an implementation of division in Haskell:

```
divp ::  (Show a, Num a, Eq a, Fractional a, Ord a) ⇒
         Poly a → Poly a → (Poly a, Poly a)
divp (P as) (P bs) = let (q, r) = go [ ] as in (P q, P r)
  where db = degree (P bs)
        go q r | degree (P r) < db = (q, r)
               | null r ∨ r ≡ [0]  = (q, r)
               | otherwise         =
                let t  = last r / last bs
                    d  = degree (P r) − db
                    ts = zeros d ++ [t]
                    m  = mulist ts bs
                in go (cleanz $ strichlist (+) q ts)
                      (cleanz $ strichlist (−) r m)
mulist :: (Show a, Num a, Eq a) ⇒ [a] → [a] → [a]
mulist c1 c2 = coeffs $ mul (P c1) (P c2)
```

First note that division expects its arguments to be polynomials over a *Fractional* data type. We do not allow polynomials over integers to be used with this implementation. The reason is that we do not want to use Euclidean division on the coefficients. That could indeed be very confusing. Furthermore, polynomials are most often used with rational or real coefficients. Restricting division to integers (using Euclidean division) would, therefore, not make much sense.

Observe further that we call *go* with an empty set – that is the initial value of q, *i.e.* the final result – and *as* – that is initially the number to be divided, the number we called a above. The function *go* has two base cases: if the degree of r, the remainder and initially *as*, is less than the degree of the divisor b, we are done. The result is our current (q, r). The same is true if r is *null* or contains only the constant 0. In this case, there is no remainder: b divides a.

Otherwise, we divide the *last* of r by the *last* of b. Note that those are the terms with the highest degree in each polynomial. This division is just a number division of the two coefficients. We still have to compute the new exponent, which is the exponent of *last r* minus the exponent of *last* b, *i.e.* their weight. We do this by subtracting their degrees and then inserting zeros at the head of the result ts. This result, ts, is then added to q. We further compute $ts \times bs$ and subtract the result from r. The function *mulist* we use for this purpose is just a wrapper around *mul* using lists of coefficients instead of *Poly* variables. With the resulting (q, r), we go into the next round.

Let us try this with our example from above:

$$\frac{4x^5 - x^4 + 2x^3 + x^2 - 1}{x^2 + 1}.$$

We call *divp* $(P\ [-1, 0, 1, 2, -1, 4])$ $(P\ [1, 0, 1])$ and get $(P\ [2, -2, -1, 4], P\ [-3, 2])$, which translates to the polynomials $4x^3 - x^2 - 2x + 2$ and $2x - 3$. This is the same result we obtained above with the manual procedure.

From here on, we can implement functions based on division, such as *divides*:

$$\begin{aligned}
&\mathit{divides} :: (\mathit{Show}\ a, \mathit{Num}\ a, \mathit{Eq}\ a, \mathit{Fractional}\ a, \mathit{Ord}\ a) \Rightarrow \\
&\qquad\quad \mathit{Poly}\ a \to \mathit{Poly}\ a \to \mathit{Bool} \\
&\mathit{divides}\ a\ b = \mathbf{case}\ b\ \text{`}\mathit{divp}\text{`}\ a\ \mathbf{of} \\
&\qquad\qquad\quad (_, P\ [0]) \to \mathit{True} \\
&\qquad\qquad\quad _ \qquad\qquad \to \mathit{False}
\end{aligned}$$

the remainder:

$$\begin{aligned}
&\mathit{remp} :: (\mathit{Show}\ a, \mathit{Num}\ a, \mathit{Eq}\ a, \mathit{Fractional}\ a, \mathit{Ord}\ a) \Rightarrow \\
&\qquad\quad \mathit{Poly}\ a \to \mathit{Poly}\ a \to \mathit{Poly}\ a \\
&\mathit{remp}\ a\ b = \mathbf{let}\ (_, r) = b\ \text{`}\mathit{divp}\text{`}\ a\ \mathbf{in}\ r
\end{aligned}$$

and, of course, the GCD:

$$\begin{aligned}
&\mathit{gcdp} :: (\mathit{Show}\ a, \mathit{Num}\ a, \mathit{Eq}\ a, \mathit{Fractional}\ a, \mathit{Ord}\ a) \Rightarrow \\
&\qquad\quad \mathit{Poly}\ a \to \mathit{Poly}\ a \to \mathit{Poly}\ a \\
&\mathit{gcdp}\ a\ b \mid \mathit{degree}\ b > \mathit{degree}\ a = \mathit{gcdp}\ b\ a \\
&\qquad\quad\ \mid \mathit{zerop}\ b \qquad = a \\
&\qquad\quad\ \mid \mathit{otherwise} = \mathbf{let}\ (_, r) = \mathit{divp}\ a\ b\ \mathbf{in}\ \mathit{gcdp}\ b\ r
\end{aligned}$$

We use a simple function to check whether a polynomial is zero:

$$\begin{aligned}
&\mathit{zerop} :: (\mathit{Num}\ a, \mathit{Eq}\ a) \Rightarrow \mathit{Poly}\ a \to \mathit{Bool} \\
&\mathit{zerop}\ (P\ [0]) = \mathit{True} \\
&\mathit{zerpo}\ _ \qquad\quad = \mathit{False}
\end{aligned}$$

We can demonstrate *gcdp* nicely on binomial coefficients. For instance, the GCD of the polynomials $x^5 + 5x^4 + 10x^3 + 10x^2 + 5x + 1$ and $x^3 + 3x^2 + 3x + 1$, thus

gcdp $(P\ [1, 5, 10, 10, 5, 1])$ $(P\ [1, 3, 3, 1])$

is $x^3 + 3x^2 + 3x + 1$.

Since polynomials consisting of binomial coefficients of n, where n is the degree of the polynomial, are always a product of polynomials composed of smaller binomial coefficients, the GCD of two polynomials consisting only of binomial coefficients, is always the smaller of the two. In other cases, that is, when the smaller does not divide the greater, this implementation of the GCD can lead to confusing results. For instance, we multiply $P\ [1,2,1]$ by another polynomial, say, $P\ [1,2,3]$. The result is $P\ [1,4,8,8,3]$. Now,

$gcdp\ (P\ [1,5,10,10,5,1])\ (P\ [1,4,8,8,3])$

does not yield the expected result $P\ [1,2,1]$, but polynomials with fractions as coefficients. The reason is that the GCD is an operation defined on integers, but we implemented it on top of fractionals. That is not what we want. In fact, we confuse concepts: the GCD is a concept defined on integral numbers, not on fractions.

And this is the prompt to turn our attention to polynomial arithmetic over a finite field and, thus, to modular polynomial arithmetic. With modular arithmetic, all coefficients in the polynomial are modulo n. That means we have to reduce those numbers. This, of course, does only make sense with integers. We first implement some helpers to reduce numbers modulo n reusing functions implemented in the previous chapter.

The first function takes an integer modulo n:

```
mmod :: (Integral a) => a -> a -> a
mmod n p | n < 0 && (-n) > p = mmod (-(mmod (-n)) p) p
         | n < 0              = mmod (p + n) p
         | otherwise          = n `rem` p
```

Equipped with this function, we can easily implement multiplication:

```
modmul :: (Integral a) => a -> a -> a -> a
modmul p f1 f2 = (f1 * f2) `mmod` p
```

For division, we reuse the *inverse* function:

```
modiv :: (Integral a) => a -> a -> a -> a
modiv p n d = modmul p n d'
  where d' = fromIntegral (M.inverse (fromIntegral d)
                                     (fromIntegral p))
```

Now, we turn to polynomials. Here is, first, a function that transforms a polynomial into one modulo n:

```
pmod :: (Integral a) => Poly a -> a -> Poly a
pmod (P cs) p = P [c `mmod` p | c <- cs]
```

In other words, we just map *mmod* on all coefficients. Let us look at some polynomials modulo a number, say, 7. The polynomial $P\ [1,2,3,4]$ we already used above is just the same modulo 7. The polynomial $P\ [5,6,7,8]$, however, changes:

$P\ [5,6,7,8]$ ‘*pmod*‘ 7

is $P\ [5,6,0,1]$ or, in other words, $8x^3 + 7x^2 + 6x + 5$ turns, modulo 7, into $x^3 + 6x + 5$.

The polynomial $x+1$ raised to the power of 5 is $x^5+5x^4+10x^3+10x^2+5x+1$. Modulo 7, this reduces to $x^5+5x^4+3x^3+3x^3+5x+1$. That is: the binomial coefficients modulo n change. For instance,

$map\ (choose2\ 6)\ [0\mathinner{.\,.}6]$

is

1,6,15,20,15,6,1.

Modulo 7, we get

1,6,1,6,1,6,1.

$map\ (choose2\ 7)\ [0\mathinner{.\,.}7]$

is

1,7,21,35,35,21,7,1.

Without big surprise, we see this modulo 7 drastically simplified:

1,0,0,0,0,0,0,1.

Here are addition and subtraction, which are very easy to convert to modular arithmetic:

```
addmp :: (Integral a) => a -> Poly a -> Poly a -> Poly a
addmp n p1 p2 = strich (+) p1 p2 `pmod` n
submp :: (Integral a) => a -> Poly a -> Poly a -> Poly a
submp n p1 p2 = strich (-) p1 p2 `pmod` n
```

Multiplication:

```
mulmp :: (Integral a) => a -> Poly a -> Poly a -> Poly a
mulmp p p1 p2 | d2 > d1   = mulmp p p2 p1
              | otherwise = P [m `mmod` p | m <- strichf (+) ms]
  where ms = [mul1 o i (coeffs p1) c | (i, c) <- zip [0..] (coeffs p2)]
        d1 = degree p1
        d2 = degree p2
        o  = modmul p
```

and product:

```
mulmp :: (Integral a) => a -> [a] -> [a] -> [a]
mulmlist p c1 c2 = coeffs $ mulmp p (P c1) (P c2)
```

We repeat the multiplication from above

$mul\ (P\ [1,2,3,4])\ (P\ [5,6,7,8])$

which was

$P\ [5, 16, 34, 60, 61, 52, 32]$

Modulo 7, this result is

$P\ [5, 2, 6, 4, 5, 3, 4]$.

The modulo multiplication

$\mathit{mulmp}\ 7\ (P\ [1, 2, 3, 4])\ (P\ [5, 6, 0, 1])$

yields the same result:

$P\ [5, 2, 6, 4, 5, 3, 4]$

Division:

$$
\begin{array}{l}
\mathit{divmp} :: (\mathit{Integral}\ a) \Rightarrow a \to \mathit{Poly}\ a \to \mathit{Poly}\ a \to (\mathit{Poly}\ a, \mathit{Poly}\ a) \\
\mathit{divmp}\ p\ (P\ \mathit{as})\ (P\ \mathit{bs}) = \textbf{let}\ (q, r) = \mathit{go}\ [0]\ \mathit{as}\ \textbf{in}\ (P\ q, P\ r) \\
\quad \textbf{where}\ \mathit{db} = \mathit{degree}\ (P\ \mathit{bs}) \\
\qquad\qquad \mathit{go}\ q\ r \mid \mathit{degree}\ (P\ r) < \mathit{db} = (q, r) \\
\qquad\qquad\qquad\quad \mid \mathit{null}\ r \vee r \equiv [0] \quad = (q, r) \\
\qquad\qquad\qquad\quad \mid \mathit{otherwise} \qquad\quad = \\
\qquad\qquad\qquad\quad\ \ \textbf{let}\ t \ \ = \mathit{modiv}\ p\ (\mathit{last}\ r)\ (\mathit{last}\ \mathit{bs}) \\
\qquad\qquad\qquad\qquad\quad\ d \ \ = \mathit{degree}\ (P\ r) - \mathit{db} \\
\qquad\qquad\qquad\qquad\quad\ \mathit{ts} = \mathit{zeros}\ d + \!\! + [t] \\
\qquad\qquad\qquad\qquad\quad\ m = \mathit{mulmlist}\ p\ \mathit{ts}\ \mathit{bs} \\
\qquad\qquad\qquad\quad\ \ \textbf{in}\ \mathit{go}\ (\mathit{cleanz}\ [c\ `\mathit{mmod}`\ p \mid c \leftarrow \mathit{strichlist}\ (+)\ q\ \mathit{ts}]) \\
\qquad\qquad\qquad\qquad\qquad (\mathit{cleanz}\ [c\ `\mathit{mmod}`\ p \mid c \leftarrow \mathit{strichlist}\ (-)\ r\ m])
\end{array}
$$

Division works exactly like the variant for infinite fields, except that we now use multiplication with the modulo inverse instead of fractional division.

Here is the GCD:

$$
\begin{array}{l}
\mathit{gcdmp} :: (\mathit{Integral}\ a)\ a \to \mathit{Poly}\ a \to \mathit{Poly}\ a \to \mathit{Poly}\ a \\
\mathit{gcdmp}\ p\ a\ b \mid \mathit{degree}\ b > \mathit{degree}\ a = \mathit{gcdmp}\ p\ b\ a \\
\qquad\qquad\quad\ \mid \mathit{zerop}\ b = a \\
\qquad\qquad\quad\ \mid \mathit{otherwise} = \textbf{let}\ (\_, r) = \mathit{divmp}\ p\ a\ b\ \textbf{in}\ \mathit{gcdmp}\ p\ b\ r
\end{array}
$$

Let us try *gcdmp* on the variation we already tested above. We multiply the polynomial $x^2 + 2x + 1$ by $3x^2 + 2x + 1$ modulo 7:

$\mathit{mulmp}\ 7\ (P\ [1, 2, 1])\ (P\ [1, 2, 3])$.

The result is $P\ [1, 4, 1, 1, 3]$.

Now, we compute the GCD with $P\ [1, 5, 10, 10, 5, 1]$ modulo 7:

$\mathit{gcdmp}\ 7\ (P\ [1, 5, 3, 3, 5, 1])\ (P\ [1, 4, 1, 1, 3])$.

The result is $P\ [1,2,1]$, as expected.

The GCD is a very useful concept with modular arithmetic. Therefore, we should also implement the variants, *mgcd* (gcd on a list), *xgcd* (the extended Euclidean algorithm) and *mxgcd* (the *xgcd* on a list).

As we have already seen in the chapter on arithmetic modulo a prime, we can just fold the argument list with gcd:

```haskell
mgcdmp :: (Integer a) => a -> [Poly a] -> Poly a
mgcdmp _ [] = P [1]
mgcdmp _ [a] = a
mgcdmp p (a : as) = foldl' (gcdmp p) a as
```

Here is the modular extended Euclidean algorithm for polynomials:

```haskell
xgcdmp :: (Integral a) => a -> Poly a -> Poly a -> (Poly a, (Poly a, Poly a))
xgcdmp p a b = go a b (P [1]) (P [0]) (P [0]) (P [1])
  where go c d uc vc ud vd | zerop c    = (d, (ud, vd))
                           | otherwise =
                             let (q, r) = divmp p d c
                             in go r c (subp p ud (mulmp p q uc))
                                       (subp p vd (mulmp p q vc)) uc vc
```

And its variant for lists:

```haskell
mxgcdmp :: (Integral a) => a -> [Poly a] -> (Poly a, [Poly a])
mxgcdmp p []       = (P [1], [])
mxgcdmp p [x]      = (x, [P [1]])
mxgcdmp p (a : as) = let (g, rs) = go [] a as in (g, reverse $ ks rs)
  where go rs i [j] =  let (g, (x, y)) = xgcdmp p i j
                       in (g, [y, x] ++ rs)
        go rs i is   = let (g, (x, y)) = xgcdmp p i (head is)
                       in go ([y, x] ++ rs) g (tail is)
        ks = M.distr (mulmp p) (P [1])
```

Finally, we implement power:

```haskell
powmp :: (Integral a) => a -> a -> Poly a -> Poly a
powmp p f poly = go f (P [1]) poly
  where go 0 y _ = y
        go 1 y x = mulmp p y x
        go n y x | even n    = go (n `div` 2) y       (mulmp p x x)
                 | otherwise = go ((n - 1) `div` 2) (mulmp p y x)
                                                     (mulmp p x x)
```

Here is a nice variant of Pascal's triangle generated by
$map\ (\lambda x \rightarrow powmp\ 7\ x\ (P\ [1,1]))\ [1\,..\,14]$:

$$
\begin{gathered}
P\ [1,1] \\
P\ [1,2,1] \\
P\ [1,3,3,1] \\
P\ [1,4,6,4,1] \\
P\ [1,5,3,3,5,1] \\
P\ [1,6,1,6,1,6,1] \\
P\ [1,0,0,0,0,0,0,1] \\
P\ [1,1,0,0,0,0,0,1,1] \\
P\ [1,2,1,0,0,0,0,1,2,1] \\
P\ [1,3,3,1,0,0,0,1,3,3,1] \\
P\ [1,4,6,4,1,0,0,1,4,6,4,1] \\
P\ [1,5,3,3,5,1,0,1,5,3,3,5,1] \\
P\ [1,6,1,6,1,6,1,1,6,1,6,1,6,1] \\
P\ [1,0,0,0,0,0,0,2,0,0,0,0,0,0,1]
\end{gathered}
$$

Before we continue with modular arithmetic, which we need indeed to understand some of the deeper problems related to polynomials, we will investigate the application of polynomials using a famous device: Babbage's difference engine.

1.3 The Difference Engine

Polynomial arithmetic, as we have seen, is very similar to number arithmetic. What is the correspondent of interpreting a number in a given numeral system in the domain of polynomials? Well, that is the *application* of the polynomial to a given number. We would substitute x for a number in the Field in which we are working and just compute the formula. For instance, the polynomial

$$x^2 + x + 1$$

can be applied to, say, 2. Then we get the formula

$$2^2 + 2 + 1,$$

which is $4 + 2 + 1 = 7$.

For other values of x, it would of course generate other values. For $x = 0$, for instance, it would give $0^2+0+1 = 1$; for $x = 1$, it is $1^2+1+1 = 3$; for $x = 3$, it yields $3^2+3+1 = 13$.

How would we apply a polynomial represented by our Haskell type? We would need to go through the list of coefficients, raise x to the power of the weight of each particular coefficient, multiply it by the coefficient and, finally, add all the values together. Here is an implementation:

$$apply :: Num\ a \Rightarrow Poly\ a \rightarrow a \rightarrow a$$
$$apply\ (P\ cs)\ x\ = sum\ [c * x \uparrow i \mid (i, c) \leftarrow zip\ [0\,..]\ cs]$$

Let us try with a very simple polynomial, $x + 1$:

apply $(P\ [1,1])\ 0$ gives 1.
apply $(P\ [1,1])\ 1$ gives 2.
apply $(P\ [1,1])\ 2$ gives 3.
apply $(P\ [1,1])\ 3$ gives 4.

This polynomial, apparently, just counts the integers adding one to the value to which we apply it. It implements `i++`.

On the first sight, this result appears to be boring. However, after a quick thought, there is a lesson to learn: we get to know the polynomial, when we look at the *sequence* it produces. So, let us implement a function that maps *apply* to lists of numbers:

$$mapply :: Num\ a \Rightarrow Poly\ a \rightarrow [\,a\,] \rightarrow [\,a\,]$$
$$mapply\ p = map\ (apply\ p)$$

For simple polynomials, the sequences are predictable. x^2, obviously, just produces the squares; x^3 produces the cubes and so on. Sequences created by powers of the simple polynomial $x + 1$, like $(x + 1)^2$, $(x + 1)^3$ and so on, still, are quite predictable, *e.g.*

mapply $(P\ [1,2,1])\ [0\,..\,10]$:	1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121
mapply $(P\ [1,3,3,1])\ [0\,..\,10]$:	1, 8, 27, 64, 125, 216, 343, 512, 729, 1000, 1331
mapply $(P\ [1,4,6,4,1])\ [0\,..\,10]$:	1, 16, 81, 256, 625, 1296, 2401, 4096, 6561, 10000, 14641
mapply $(P\ [1,5,10,10,5,1])\ [0\,..\,10]$:	1, 32, 243, 1024, 3125, 7776, 16807, 32768, 59049, 100000, 161051

The first line, easy to recognise, is the squares, but pushed one up, *i.e.* the application to 0 yields the value for 1^2, the application to 1 yields the value for 2^2 and so on. The second, still easy to recognise, is the cubes – again pushed up by one. The third line is the powers of four and the fourth line is the powers of five, both pushed up by one.

That is not too surprising at the end, since $P\ [1,2,1]$ is the result of squaring $P\ [1,1]$, which generates the integers pushed one up; $P\ [1,3,3,1]$ is the result of raising $P\ [1,1]$ to the third power and so on.

Things become more interesting, when we deviate from binomial coefficients. The sequence produced by *mapply* $(P\ [1,2,3,4])\ [1\,..\,10]$, for instance, does not resemble such a simple pattern:

1, 10, 49, 142, 313, 586, 985, 1534, 2257, 3178, 4321.

Even the Online Encyclopedia has nothing interesting to say about it.

The same is true for $mapply\ (P\ [5, 6, 7, 8])\ [1 \,..\, 10]$, which is

5, 26, 109, 302, 653, 1210, 2021, 3134, 4597, 6458, 8765.

This raises another interesting question: given a sequence, is there a method by which we can identify the polynomial that created it? Yes, there is. In fact, there are. There was even a machine that helped guessing polynomials from sequences. It was built in the early 19^{th} century by Charles Babbage (1791 – 1871), an English polymath, mathematician, philosopher, economist and inventor.

Babbage stands in the tradition of designers and constructors of early computing machinery; predecessors of his in this tradition were, for instance, Blaise Pascal (1623 – 1662) and Gottfried Wilhelm Leibniz (1646 – 1716). Babbage designed two series of machines, first, the difference engines and, later, the analytical engines.

The analytical engine, unfortunately, was not built in his lifetime. The final collapse of the project came in 1878, after Babbage's death in 1871, due to lack of finance. The analytical engine would have been a universal (Turing-complete) computer very similar to our computers today, but not working on electricity, but on steam and brawn. It would have been programmed by punch cards that, in Babbage's time, were used for controlling looms. Programs would have resembled modern assembly languages allowing control structures like selection and iteration. In the context of a description of the analytical engine, Ada Lovelace (1815 – 1852), a friend of Babbage and daughter of Lord Byron, described how to compute Bernoulli numbers with the machine. She is, therefore, considered the first computer programmer in history.

The difference engine, at which we will look here, is much simpler. It was designed to analyse polynomials and what it did was, according to Babbage, "computing differences". During Babbage's lifetime, a first version was built and sucessfully demonstrated. The construction of a second, much more powerful version which was financially backed by the government, failed due to disputes between Babbage and his engineers. This machine was finally built by the London Science Museum in 1991 using material and engineering techniques available in the 19^{th} century proving this way that it was actually possible for Babbage and his engineers to build such a machine.

The difference engine, as Babbage put it, computes differences, namely the differences in a sequence of numbers. It would take as input a sequence of the form

0,1,16,81,256,625,1296,2401,4096,6561,10000

and compute the differences between the individual numbers:

$$\begin{array}{rcrcr} 1 & - & 0 & = & 1 \\ 16 & - & 1 & = & 15 \\ 81 & - & 16 & = & 65 \\ 256 & - & 81 & = & 175 \\ \dots & & & & \end{array} \tag{1.10}$$

Here is a simple function that does this job for us:

$$
\begin{aligned}
&\mathit{diffs} :: [\mathit{Zahl}] \to [\mathit{Zahl}]\\
&\mathit{diffs}\ [\,] \qquad\quad\ = [\,]\\
&\mathit{diffs}\ [\_] \qquad\quad = [\,]\\
&\mathit{diffs}\ (a : b : cs) = (b - a) : \mathit{diffs}\ (b : cs)
\end{aligned}
$$

Applied on the sequence above, *diffs* yields:

1,15,65,175,369,671,1105,1695,2465,3439

What is so special about it? Perhaps, nothing. But let us repeat the process using this result. The repetition yields:

14,50,110,194,302,434,590,770,974

One more time:

36,60,84,108,132,156,180,204

And once again:

24,24,24,24,24,24,24

Suddenly, we have a constant list. How often did we apply *diffs*? Four times – and, as you may have realised, the original sequence was generated by the polynomial x^4, a polynomial of degree 4. Is that coincidence?

For further investigation, we implement the complete difference engine, which takes differences, until it reaches a constant sequence.

$$
\begin{aligned}
&\mathit{dengine} :: [\mathit{Zahl}] \to [[\mathit{Zahl}]]\\
&\mathit{dengine}\ cs \mid \mathit{constant}\ cs = [\,]\\
&\qquad\qquad\quad \mid \mathit{otherwise} = ds : \mathit{dengine}\ ds\\
&\quad \mathbf{where}\ ds = \mathit{diffs}\ cs\\
&\qquad\qquad \mathit{constant}\ [\,] \qquad\ = \mathit{True}\\
&\qquad\qquad \mathit{constant}\ (x : xs) = \mathit{all}\ (\equiv x)\ xs
\end{aligned}
$$

Note that we restrict coefficients to integers. This is just for clarity. Usually, polynomials are defined over a field, such as the rational or the real numbers.

To confirm our suspicion that the difference engine creates n difference sequences for a polynomial of degree n, we apply the engine on x, x^2, x^3, x^4 and x^5 and count the sequences it creates:

$\mathit{length}\ (\mathit{dengine}\ (\mathit{mapply}\ (P\ [0,1])\ [0\,..\,32]))$: 1
$\mathit{length}\ (\mathit{dengine}\ (\mathit{mapply}\ (P\ [0,0,1])\ [0\,..\,32]))$: 2
$\mathit{length}\ (\mathit{dengine}\ (\mathit{mapply}\ (P\ [0,0,0,1])\ [0\,..\,32]))$: 3
$\mathit{length}\ (\mathit{dengine}\ (\mathit{mapply}\ (P\ [0,0,0,0,1])\ [0\,..\,32]))$: 4
$\mathit{length}\ (\mathit{dengine}\ (\mathit{mapply}\ (P\ [0,0,0,0,0,1])\ [0\,..\,32]))$: 5

The engine already has a purpose: it tells us the degree of the polynomial that generates a given sequence. It can do much more, though. For instance, it lets us predict the next value in the sequence. To do so, we take the constant difference from the last sequence and add it to the last difference of the previous sequence; we take that result and add it to the previous sequence and so on, until we reach the first sequence. Consider the sequence and its differences from above:

0,1,16,81,256,625,1296,2401,4096,6561,10000
1,15,65,175,369,671,1105,1695,2465,3439
14,50,110,194,302,434,590,770,974
36,60,84,108,132,156,180,204
24,24,24,24,24,24,24

We start at the bottom and compute $204 + 24 = 228$. This is the next difference of the previous sequence. We compute $974 + 228 = 1202$. We go one line up and compute $3439+1202 = 4641$. This, finally, is the difference to the next value in the input sequence, which, hence, is $10000 + 4641 = 14641$ and, indeed, 11^4. Even without knowing the polynomial that actually generates the sequence, we are now able to continue it. Here is a function that does that for us:

$$
\begin{aligned}
&\mathit{predict} :: [[\mathit{Zahl}]] \to [\mathit{Zahl}] \to \mathit{Maybe\ Zahl}\\
&\mathit{predict\ ds}\ [\,] = \ \mathit{Nothing}\\
&\mathit{predict\ ds\ xs} = \mathbf{case}\ \mathit{go}\ (\mathit{reverse\ ds})\ \mathbf{of}\\
&\qquad\qquad\qquad\quad 0 \to \mathit{Nothing}\\
&\qquad\qquad\qquad\quad d \to \mathit{Just}\ (d + (\mathit{last\ xs}))\\
&\quad \mathbf{where}\ \mathit{go} = \mathit{foldl'}\ (\lambda x\ c \to \mathit{last}\ c + x)\ 0
\end{aligned}
$$

The function takes two arguments: the first is the list of difference sequences and the second is the original sequence. We apply *go* on the reverse of the sequences (because we are working backwards). For each sequence in this list, we get the last and add it to the last of the previous until we have exhausted the list. If *go* yields 0, we assume that something went wrong. The list of sequences may have been empty in the first place. Otherwise, we add the result to the last of the original list.

Here are some more examples:

let $s = \mathit{mapply}\ (P\ [0,1])\ [0\,..\,10]$ **in** $\mathit{predict}\ (\mathit{dengine}\ s)\ s$: 11
let $s = \mathit{mapply}\ (P\ [0,0,1])\ [0\,..\,10]$ **in** $\mathit{predict}\ (\mathit{dengine}\ s)\ s$: 121
let $s = \mathit{mapply}\ (P\ [0,0,0,1])\ [0\,..\,10]$ **in** $\mathit{predict}\ (\mathit{dengine}\ s)\ s$: 1331
let $s = \mathit{mapply}\ (P\ [0,0,0,0,1])\ [0\,..\,10]$ **in** $\mathit{predict}\ (\mathit{dengine}\ s)\ s$: 14641
let $s = \mathit{mapply}\ (P\ [0,0,0,0,0,1])\ [0\,..\,10]$ **in** $\mathit{predict}\ (\mathit{dengine}\ s)\ s$: 161051

But how can we find the polynomial itself that generates the given sequence? With the help of the difference engine, we already know the degree of the polynomial. Supposed, we know that the first element in the sequence was generated applying 0 to the unknown polynomial and the second one was generated applying 1, the third by applying 2 and so on, we have all information we need.

From the degree, we know the form of the polynomial. A polynomial of degree 1 has the form $a_1x + a_2$; a polynomial of degree 2 has the form $a_1x^2 + a_2x + a_3$; a polynomial of degree 3 has the form $a_1x^3 + a_2x^2 + a_3x + a_4$ and so on.

Since we know the values to which the polynomial is applied, we can easily compute the value of the x-part of the terms. They are that value raised to the power of the weight. The challenge, then, is to find the coefficient by which that value is multiplied.

The first element in the sequence, the one created by applying the polynomial to 0, is just the last coefficient, the one "without" any x, since the other terms "disappear", when we apply to 0. Consider for example a polynomial of the form $x^2 + x + a$. When we apply it to 0, we get $0^2 + 0 + a = c$, where c is the first (or, in this notation, the last) value in the sequence. Thus, $a = c$.

The second element is 1 applied to the formula and, therefore, all terms equal their coefficients, since cx^n, for $x = 1$, is just c. The third element results from applying 2 to the polynomial, it hence adheres to a formula where unknown values (the coefficients) are multiplied by 2, $2^2 = 4$, $2^3 = 8$ and so on.

In other words, for a polynomial of degree n, we can devise a system of linear equations with $n + 1$ unknowns and the $n + 1$ first elements of the sequence as constant values. A polynomial of degree 2, for instance, yields the system

$$\begin{array}{ccccccc} & & & & a & = & a_1 \\ a & + & b & + & c & = & a_2 \\ a & + & 2b & + & 4c & = & a_3 \end{array} \tag{1.11}$$

where the constant numbers a_1, a_2 and a_3 are the first three elements of the sequence. A polynomial of degree 3 would generate the system

$$\begin{array}{ccccccccc} & & & & & & a & = & a_1 \\ a & + & b & + & c & + & d & = & a_2 \\ a & + & 2b & + & 4c & + & 8d & = & a_3 \\ a & + & 3b & + & 9c & + & 27d & = & a_4 \end{array} \tag{1.12}$$

We have already learnt how to solve such systems: we can apply Gaussian elimination. The result of the elminiation is the coefficients of the generating polynomial, which are the unknowns in the linear equations. The known values (which we would call the coefficients in a linear equation) are the values obtained by computing x^i where i is the weight of the coefficient. Here is a function to extract the known values, the $x$es raised to the weight, from a given sequence with a given degree:

$$\begin{array}{l} genCoeff :: Zahl \to Zahl \to Zahl \to [Zahl] \\ genCoeff\ d\ n\ x = map\ (n\uparrow)\ [0\,..\,d] + [x] \end{array}$$

Here, d is the degree of the polynomial, n is the value to which the polynomial is applied and x is the result, *i.e.* the value from the sequence. We create the sequence n^i, for $0 \leq i \leq d$ and append x yielding one line of the system of linear equations.

When we apply *genCoeff* on the the sequence generated by x^4, we would have:

genCoeff 4 0 0 resulting in $[1,0,0,0,0,0]$
genCoeff 4 1 1 resulting in $[1,1,1,1,1,1]$
genCoeff 4 2 16 resulting in $[1,2,4,8,16,16]$
genCoeff 4 3 81 resulting in $[1,3,9,27,81,81]$
genCoeff 4 4 256 resulting in $[1,4,16,64,256,256]$

Note that the results are very regular: we see constant 1 in the first column, the natural numbers in the second column, the squares in the third, the cubes in the fourth and n^4 in the fifth and sixth column. Those are just the values for x^i, for $i \in \{0 \ldots 4\}$. Since the value in the sixth column, the one we took from the sequence, equals the value in the fifth column, we can already guess that the polynomial is simply x^4. Here is another sequence, generated by a secret polynomial:

14, 62, 396, 1544, 4322, 9834, 19472, 34916, 58134, 91382, 137204

We compute the difference lists using *dengine* as *ds* and compute the degree of the polynomial using *length ds*. The result is 4. Now we call *genCoeff* on the first four elements of the sequence:

genCoeff 4 0 14 resulting in $[1,0,0,0,0,14]$
genCoeff 4 1 62 resulting in $[1,1,1,1,1,62]$
genCoeff 4 2 396 resulting in $[1,2,4,8,16,396]$
genCoeff 4 3 1544 resulting in $[1,3,9,27,81,1544]$
genCoeff 4 4 4322 resulting in $[1,4,16,64,256,4322]$

We already see that this is a less trivial case: the last two numbers are not equal!

Now we use *genCoeff* to create a matrix representing the entire system of equations:

$$
\begin{array}{l}
\mathit{findCoeffs} :: [[\mathit{Zahl}]] \rightarrow [\mathit{Zahl}] \rightarrow \mathit{L.Matrix}\ \mathit{Zahl} \\
\mathit{findCoeffs}\ \mathit{ds}\ \mathit{sq} = \mathit{L.M}\ [\mathit{genCoeff}\ d\ n\ x \mid (n,x) \leftarrow \mathit{zip}\ [0\,..\,d]\ \mathit{sq}] \\
\quad \mathbf{where}\ d = \mathit{fromIntegral}\ (\mathit{length}\ \mathit{ds})
\end{array}
$$

The function *findCoeffs* receives the list of difference sequences created by *dengine* and the original sequence. It computes the degree of the generating polynomial as *length ds* and, then, it goes through the first d elements of the sequence calling *genCoeff* with d, the known input value n, and x, the element of the sequence. For the sequence generated by x^4, we obtain $M\ [[1,0,0,0,0,0],[1,1,1,1,1,1],[1,2,4,8,16,16],$ $[1,3,9,27,81,81],[1,4,16,64,256,256]]$, which corresponds to the matrix

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 4 & 8 & 16 & 16 \\ 1 & 3 & 9 & 27 & 81 & 81 \\ 1 & 4 & 16 & 64 & 256 & 256 \end{pmatrix}$$

For the sequence of the unknown polynomial, we obtain $M\ [[1,0,0,0,0,14], |\ [1,1,1,1,1,62], [1,2,4,8,16,396], [1,3,9,27,81,1544], [1,4,16,64,256,4322]]$, which corresponds to the matrix:

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 14 \\ 1 & 1 & 1 & 1 & 1 & 62 \\ 1 & 2 & 4 & 8 & 16 & 396 \\ 1 & 3 & 9 & 27 & 81 & 1544 \\ 1 & 4 & 16 & 64 & 256 & 4322 \end{pmatrix}$$

The next steps are simple. We create the echelon form and solve by back-substitution. The following function puts all the bits together to find the generating polynomial:

$$\begin{aligned} &\mathit{findGen} :: [[\mathit{Zahl}]] \to [\mathit{Zahl}] \to [\mathit{Quoz}] \\ &\mathit{findGen}\ \mathit{ds} = \mathit{L.backsub} \circ \mathit{L.echelon} \circ \mathit{findCoeffs}\ \mathit{ds} \end{aligned}$$

Applied on the difference list and the sequence generated by x^4, *findGen* yields:

$[0 \% 1, 0 \% 1, 0 \% 1, 0 \% 1, 1 \% 1]$,

which indeed corresponds to the polynomial x^4. For the sequence generated by the unknown polynomial, we get:

$[14 \% 1, 9 \% 1, 11 \% 1, 16 \% 1, 12 \% 1]$,

which corresponds to the polynomial $12x^4 + 16x^3 + 11x^2 + 9x + 14$. Let us test:

$\mathit{mapply}\ (P\ [14, 9, 11, 16, 12])\ [0\,..\,10]$ yields:

14,62,396,1544,4322,9834,19472,34916,58134,91382,137204,

which indeed is the same sequence as we saw above!

Now, what about the differences generated by the difference engine? Those, too, are sequences of numbers. Are there polynomials that generate those sequences? The first difference sequence of our formerly unknown polynomial is

48,334,1148,2778,5512,9638,15444,23218,33248,45822

The next three difference sequences could be derived from this sequence – so, we can assume that this sequence is generated by a polynomial of degree 3. Let us see what *findGen* (*tail ds*) (*head ds*) yields (with *ds* being the list of difference sequences of that polynomial):

$[48 \,\%\, 1, 118 \,\%\, 1, 120 \,\%\, 1, 48 \,\%\, 1]$,

which corresponds to the polynomial $48x^3 + 120x^2 + 118x + 48$. Let us test again:

mapply (P [48, 118, 120, 48]) [0 . . 10] yields:

48,334,1148,2778,5512,9638,15444,23218,33248,45822,61228

The next difference sequence should then be generated by a polynomial of degree 2. We try with
let $ds' = tail\ ds$ **in** *findGen* ($tail\ ds'$) ($head\ ds'$)
and get

$[286 \,\%\, 1, 384 \,\%\, 1, 144 \,\%\, 1]$,

which corresponds to the polynomial $144x^2 + 384x + 286$.

mapply (P [286, 384, 144]) [0 . . 10] yields:

286,814,1630,2734,4126,5806,7774,10030,12574,15406,18526

which, indeed, is the third difference sequence.

Finally, the last but one sequence, the last that is not constant, should be generated by a polynomial of degree 1. We try with
let $ds'' = tail\ (tail\ ds)$ **in** *findGen* ($tail\ ds''$) ($head\ ds''$)
and get

$528 \,\%\, 1, 288 \,\%\, 1$

which corresponds to the polynomial $288x + 528$.

mapply (P [528, 288]) [0 . . 10] yields:

528,816,1104,1392,1680,1968,2256,2544,2832,3120,3408

which, again is the expected difference sequence.

The differences are closely related to the tremendously important concept of the *derivative* of a function. The derivative of a polynomial π of degree n is a polynomial π' of degree $n - 1$ that measures the *rate of change* or *slope* of π. The derivative expresses the rate of change precisely for any point in π. We will look at this with much more attention in the next section; the third part will then be entirely dedicated to derivatives and related concepts.

The difference sequences and the polynomials that generate them are also a measure of the rate of change. Actually, the difference between two points *is* the rate of change of that polynomial between those two points. The difference, however, is a sloppy measure.

Without going into too much detail here, we can quickly look at how the derivative of a polynomial is computed, which, in fact, is very easy. For a polynomial of the form

$$ax^n + bx^m + \cdots + cx + d,$$

the derivative is

$$nax^{n-1} + mbx^{m-1} + \cdots + c.$$

In other words, we drop the last term (which is the first term in our Haskell representation of polynomials) and, for all other terms, we multiply the term by the exponent and reduce the exponent by one.

The derivative of the polynomial x^4, for instance, is $4x^3$; in the notation of our polynomial type, we have $P\ [0,0,0,0,1]$ and its derivative $P\ [0,0,0,4]$. The derivative of $4x^3$ is $12x^2$, whose derivative then is $24x$, whose derivative is just 24 (a number you have already seen in this very section!). The deriviative of our polynomial

$$12x^4 + 16x^3 + 11x^2 + 9x + 14$$

is

$$48x^3 + 48x^2 + 22x + 9.$$

Note that the first term equals the first term of the polynomial that we identified as the generator of the first difference sequence. Indeed, the differences are sloppy as a measure for the rate of change – but they are not completely wrong!

Here is a function to compute the derivative:

```
derivative :: (Eq a, Num a, Enum a) =>
              (a -> a -> a) -> Poly a -> Poly a
derivative o (P as) = P (cleanz (map op (zip [1..] (drop 1 as))))
  where op (x, c) = x `o` c
```

Note that we keep the implementation of *derivative* flexible. Instead of hardcoding $\times$, we use a function parameter 'o', so we can pass in the operation we need. We will later see how this is useful.

What is the sequence generated by the derivative of our polynomial? Well, we define the derivative as **let** $p' = derivative\ (P\ [14,9,11,16,12])$, which is $P\ [9,22,48,48]$, apply it using $mapply\ p'\ [0\,..\,10]$ and see:

9,127,629,1803,3937,7319,12237,18979,27833,39087,53029

Quite different from the first difference sequence we saw above!

What about the second derivative? We define **let** $p'' = \textit{derivative}\ p'$ and get $P\ [22, 96, 144]$. This polynomial creates the sequence

22,262,790,1606,2710,4102,5782,7750,10006,12550,15382

The next derivative, **let** $p''' = \textit{derivative}\ p''$, is $P\ [96, 288]$ and generates the sequence

96,384,672,960,1248,1536,1824,2112,2400,2688,2976.

You can already predict the next derivative, which is a polynomial of degree 0: it is $P\ [288]$. This is a constant polynomial and will generate a constant sequence, namely the sequence 288. That, however, was also the constant sequence generated by the difference engine. Of course, when the rate of change is the same everywhere in the original polynomial, then precision does not make any difference anymore. The two methods shall come to the same result.

Consider the simple polynomial x^2. It generates the sequence

$$0, 1, 4, 9, 16, 25, 36, 49, \ldots$$

The differences are

$$1, 3, 5, 7, 9, 11, 13, \ldots$$

The differences of this list are all 2.

The derivative of x^2 is $2x$. It would generate the sequence

$$0, 2, 4, 6, 8, 10, 12, 14, \ldots$$

which does not equal the differences. However, we can already see that the derivative of $2x$, 2, is constant and generates the constant sequence

$$2, 2, 2, 2, 2, 2, 2, 2, \ldots$$

1.4 Differences and Binomial Coefficients

Isaac Newton studied the relation between sequences and their differences intensely and came up with a formula. Before we go right to it, let us observe on our own. The following table shows the values and differences of a certain polynomial. In the first row, it shows the value of n, *i.e.* the value to which the polynomial is applied; in the second row, we see the result for this n; in the first column we have the first values from the sequence and its difference lists:

	0	1	2	3	4
	14	62	396	1544	4322
14	1	1	1	1	1
48	0	1	2	3	4
286	0	0	1	3	6
528	0	0	0	1	4
288	0	0	0	0	1

What we see in the cells of the table are factors. With their help, we can compute the values in the sequence by formulas of the type:

$$\begin{array}{ccccccccccccccccccccc}
1 & \times & 14 & & & & & & & & & & & & & & & & & = & 14 \\
1 & \times & 14 & + & 1 & \times & 48 & & & & & & & & & & & & & = & 62 \\
1 & \times & 14 & + & 2 & \times & 48 & + & 1 & \times & 286 & & & & & & & & & = & 396 \\
1 & \times & 14 & + & 3 & \times & 48 & + & 3 & \times & 286 & + & 1 & \times & 528 & & & & & = & 1544 \\
1 & \times & 14 & + & 4 & \times & 48 & + & 6 & \times & 286 & + & 4 & \times & 528 & + & 1 & \times & 288 & = & 4322
\end{array} \tag{1.13}$$

The next question would then be: what are those numbers? But, here, I have to ask you to look a bit more closely at the table. What we see in the columns left-to-right is:

$$\begin{array}{ccccccccc}
 & & & & 1 & & & & \\
 & & & 1 & & 1 & & & \\
 & & 1 & & 2 & & 1 & & \\
 & 1 & & 3 & & 3 & & 1 & \\
1 & & 4 & & 6 & & 4 & & 1
\end{array}$$

Those are binomial coefficients! Indeed. We could rewrite the table as

	0	1	2	3	4
	14	62	396	1544	4322
14	$\binom{0}{0}$	$\binom{1}{0}$	$\binom{2}{0}$	$\binom{3}{0}$	$\binom{4}{0}$
48	$\binom{0}{1}$	$\binom{1}{1}$	$\binom{2}{1}$	$\binom{3}{1}$	$\binom{4}{1}$
286	$\binom{0}{2}$	$\binom{1}{2}$	$\binom{2}{2}$	$\binom{3}{2}$	$\binom{4}{2}$
528	$\binom{0}{3}$	$\binom{1}{3}$	$\binom{2}{3}$	$\binom{3}{3}$	$\binom{4}{3}$
288	$\binom{0}{3}$	$\binom{1}{4}$	$\binom{2}{4}$	$\binom{3}{4}$	$\binom{4}{4}$

If this were universally true, we could devise a much better prediction function. The one we wrote in the previous section has the disadvantage that we can only predict the next number in the sequence. To predict a value way ahead we need to generate number by number before we are there. With Newton's trick, we could compute any number in the sequence in one step. All we have to do is to get the *head*s of the sequences and to calculate the formula:

$$\sum_{k=0}^{d} h_k \binom{n}{k}$$

where d is the degree of the polynomial, n the position in the sequence, *i.e.* the number to which we apply the polynomial, and h_k the head of the sequence starting to count with the original sequence as $k = 0$. The sixth value ($n = 5$) of the sequence would then be

$$14 \times \binom{5}{0} + 48 \times \binom{5}{1} + 286 \times \binom{5}{2} + 528 \times \binom{5}{3} + 288 \times \binom{5}{4},$$

which is

$$14 + 48 \times 5 + 286 \times 10 + 528 \times 10 + 288 \times 5,$$

which, in its turn, is

$$14 + 240 + 2860 + 5280 + 1440 = 9834,$$

which is indeed the next value in the sequence.

Here is an implementation:

$$
\begin{array}{l}
\mathit{newton} :: \mathit{Zahl} \to [[\mathit{Zahl}]] \to [\mathit{Zahl}] \to \mathit{Zahl} \\
\mathit{newton}\ n\ \mathit{ds}\ \mathit{seq} = \mathit{sum}\ \mathit{ts} \\
\quad \textbf{where}\ \mathit{hs} = \mathit{getHeads}\ \mathit{seq}\ \mathit{ds} \\
\qquad\qquad \mathit{ts}\ = [h * (\mathit{choose}\ n\ k) \mid (h,k) \leftarrow \mathit{zip}\ \mathit{hs}\ [0\mathinner{..}n]] \\
\mathit{getHeads} :: [\mathit{Zahl}] \to [[\mathit{Zahl}]] \to [\mathit{Zahl}] \\
\mathit{getHeads}\ \mathit{seq}\ \mathit{ds} = \mathit{map}\ \mathit{head}\ (\mathit{seq} : \mathit{ds})
\end{array}
$$

To perform some experiments, here, as a reminder, are the first 14 numbers of the sequence generated by our polynomial $P\ [14,9,11,16,12]$:

14,62,396,1544,4322,9834,19472,34916,58134,91382,137204,198432,278186,379874

We set $s = \mathit{mapply}\ P\ [14,9,11,16,12]\ [0\mathinner{..}10]$ and $d = \mathit{dengine}\ s$. Now we perform some tests:
$\mathit{newton}\ 0\ d\ s$ gives 14.
$\mathit{newton}\ 1\ d\ s$ gives 62.
$\mathit{newton}\ 5\ d\ s$ gives 9834.
$\mathit{newton}\ 11\ d\ s$ gives 198432.
$\mathit{newton}\ 13\ d\ s$ gives 379874.

The approach seems to work. But there is more. The function *newton* gives us a closed form to compute any number in the sequence, given that we have the beginning of that sequence and its difference lists. A closed form, however, is a generating formula – it is the polynomial that generates the entire sequence. We just need a way to make the formula implicit in *newton* explicit.

We can do that using our polynomial data type. When we can express the binomial coefficients in terms of polynomials and apply them to the formula used above, we will get the polynomial out that generates this sequence. Here is a function that does that:

$$
\begin{array}{l}
\mathit{bin2poly} :: \mathit{Zahl} \to \mathit{Zahl} \to \mathit{Poly}\ \mathit{Quoz} \\
\mathit{bin2poly}\ h\ 0 \quad = P\ [h \,\%\, 1] \\
\mathit{bin2poly}\ h\ 1 \quad = P\ [0, h \,\%\, 1] \\
\mathit{bin2poly}\ h\ k \quad = P\ [h \,\%\, (B.\mathit{fac}\ k)]\ \text{`}\mathit{mul}\text{`}\ \mathit{go}\ (k \,\%\, 1) \\
\quad \textbf{where}\ \mathit{go}\ 1 = P\ [0,1] \\
\qquad\qquad \mathit{go}\ i = P\ [-(i-1),1]\ \text{`}\mathit{mul}\text{`}\ (\mathit{go}\ (i-1))
\end{array}
$$

The function receives two integers: the first one is a factor (the head) by which we multiply the resulting binomial polynomial and the second one is k in $\binom{n}{k}$. Note that we do not need n, since n is the unknown, the base of our polynomial.

If $k = 0$, the binomial is 1, since for all binomial coefficients: $\binom{n}{0} = 1$. We, hence, return a constant polynomial consisting of the factor. This corresponds to $h_0 \times \binom{n}{0}$. The result is just h_0. Note that we convert the coefficients to rational numbers, since that is the type the function is supposed to yield.

If $k = 1$, the binomial is n, since for all binomials: $\binom{n}{1} = n$. Because n is the base of the polynomial, n itself is expressed by $P\ [0,1]$. This is just $n+0$ and, hence, n. Since we multiply with h, the result in this case is $h \times n = hn$, or, in the language of our Haskell polynomials $P\ [0,h]$.

Otherwise, we go into the recursive *go* function. The function receives one rational number, namely k (which, de facto, is an integer). The base case is $k = 1$. In that case we yield $P\ [0,1]$, which is just n. Otherwise, we create the polynomial $P\ [-(i-1),1]$, that is $n-(k-1)$ and multiply with the result of *go* applied to $i-1$. The function, hence, creates the numerator of the fraction formula of the binomial coefficient:

$$n(n-1)(n-2)\ldots(n-k+1).$$

The result of the function is then multiplied by h divided by $k!$. The former, still, is some head from the difference sequences and the latter is the denominator of the fraction formula. We, thus, compute:

$$\frac{hn(n-1)(n-2)\ldots(n-k+1)}{k!}.$$

Now, we can use this formula represented by a polynomial to compute the generating polynomial. The function that does so has exactly the same structure as the *newton* function. The difference is just that it expresses binomial coefficients as polynomials and that it does not receive a concrete number n for which we want to compute the corresponding value (because we want to compute the formula generating all the values):

```
newtonGen :: [[Zahl]] → [Zahl] → Poly [Quoz]
newtonGen ds seq = sump ts
  where hs = getHeads seq ds
        ts  = [bin2poly h k | (h, k) ← zip hs [0 .. n]]
        n   = fromIntegral (length $ ds)
```

When we call *newtonGen ds s*, *ds* still being the difference lists and s the sequence in question, we see:

$P\ [14\,\%\,1, 9\,\%\,1, 11\,\%\,1, 16\,\%\,1, 12\,\%\,1]$,

which we immediately recognise as our polynomial $12x^4 + 16x^3 + 11x^2 + 9x + 14$.

For another test, we apply the monomial x^5 as

let $s = mapply\ (P\ [0,0,0,0,0,1])\ [0\,..\,10]$ **in** *newtonGen* (*dengine s*) *s*

and see

$P\ [0\,\%\,1, 0\,\%\,1, 0\,\%\,1, 0\,\%\,1, 0\,\%\,1, 1\,\%\,1]$,

which is indeed the polynomial x^5.

But now comes the hard question: why does that work at all???

To answer this question, we should make sure to understand how Newton's formula works. The point is that we restrict ourselves to the heads of the sequences as basic building blocks. When we compute some value x_n in the sequence, we need to recursively compute x_{n-1} and the difference between x_{n-1} and x_n and add them together. Let us build a model that simulates this approach and that allows us to reason about what is going on more easily.

We use as a model a polynomial of degree 3; that model is sufficiently complex to simulate the problem completely and is, on the other hand, somewhat simpler than a model based on a polynomial of degree 4, like the one we have studied above – not to mention a model for polynomials of any degree.

The model consists of a data type:

$$
\begin{aligned}
&\textbf{data}\ \mathit{Newton} = H \mid X \mid Y \mid Z\\
&\quad \textbf{deriving}\ (\mathit{Show}, \mathit{Eq})
\end{aligned}
$$

The *Newton* type has four constructors: H represents the head of the original sequence; X is the head of the first difference list; Y is the head of the second difference list and Z is the constant element repeated in the last difference list. (Remember that a polynomial of degree 3 generates 3 difference lists.)

The model also contains a function to compute positions in the sequence. This function, called *cn* (for "computeNewton"), takes two arguments: a *Newton* constructor and an integer. The integer tells us the position we want to compute starting with the head $H = 0$:

$$
\begin{aligned}
&\mathit{cn} :: \mathit{Newton} \to \mathit{Natural} \to [\mathit{Newton}]\\
&\mathit{cn}\ H\ 0 = [H]\\
&\mathit{cn}\ H\ n = \mathit{cn}\ H\ (n-1) +\!\!+\ \mathit{cn}\ X\ (n-1)
\end{aligned}
$$

When we want to compute the first element in the sequence, $\mathit{cn}\ H\ 0$, we just return $[H]$. When we want to compute any other number, we recursively call $\mathit{cn}\ H\ (n-1)$, which computes the previous data point, and add $\mathit{cn}\ X\ (n-1)$, which computes the difference between n and $n-1$. Here is how we compute the difference:

$$
\begin{aligned}
&\mathit{cn}\ X\ 0 = [X]\\
&\mathit{cn}\ X\ n = \mathit{cn}\ X\ (n-1) +\!\!+\ \mathit{cn}\ Y\ (n-1)
\end{aligned}
$$

If we need the first difference, $\mathit{cn}\ X\ 0$, we just return $[X]$. Otherwise, we call $\mathit{cn}\ X\ (n-1)$, this computes the previous difference, and compute $\mathit{cn}\ Y\ (n-1)$, the difference between the previous and the current difference. Here is how we compute the difference of the difference:

$$
\begin{aligned}
&\mathit{cn}\ Y\ 0 = [Y]\\
&\mathit{cn}\ Y\ n = Z : \mathit{cn}\ Y\ (n-1)
\end{aligned}
$$

If we need the first difference, $cn\ Y\ 0$, we just return $[Y]$. Otherwise, we compute the previous difference $cn\ Y\ (n-1)$ adding Z, the constant difference, to the result.

The simplest case is of course computing the first in the sequence. This is just:

$cn\ H\ 0$, which yields $[H]$.

Computing the second in the sequence is slightly more work:

$cn\ H\ 1$ goes to
$cn\ H\ 0 \mathbin{+\!\!+} cn\ X\ 0$ which is
$[H] \mathbin{+\!\!+} [X]$.

We, hence, get $[H, X]$. That is the head of the sequence plus the head of the first difference list.

Computing the third in the sequence

$cn\ H\ 2$ calls
$cn\ H\ 1 \mathbin{+\!\!+} cn\ X\ 1$, which is
$cn\ H\ 0 \mathbin{+\!\!+} cn\ X\ 0$ and $cn\ X\ 0 \mathbin{+\!\!+} cn\ Y\ 0$.

We hence get $[H, X, X, Y]$. This is the head of the original sequence plus the head of the first difference sequence (we are now at $H\ 1$) plus this difference plus the first of the second difference sequence.

This looks simple, but already after a few steps, the result looks weird. For $cn\ H\ 5$, for example, we see

$[H, X, X, Y, X, Y, Z, Y, X, Y, Z, Y, Z, Z, Y, X, Y, Z, Y, Z, Z, Y, Z, Z, Z, Y]$,

which is somewhat confusing. The result, however, is correct. We can illustrate that by comparing the result with a real polynomial of degree 3, say, $P\ [2, 28, 15, 22]$, this is the polynomial $22x^3 + 15x^2 + 28x + 2$; this polynomial generates the sequence 2, 67, 294, 815, 1762, 3267, 5462, 8479, 12450, 17507, 23782.

We now define a function that substitutes the symbols of our model by the heads of the sequence and the difference lists:

$$
\begin{array}{l}
\mathit{new2a} :: (a, a, a, a) \to \mathit{Newton} \to a \\
\mathit{new2a}\ (h, x, y, z)\ n = \textbf{case}\ n\ \textbf{of} \\
\quad H \to h \\
\quad X \to x \\
\quad Y \to y \\
\quad Z \to z \\
\mathit{subst} :: (a, a, a, a) \to [\mathit{Newton}] \to [a] \\
\mathit{subst}\ \mathit{as} = \mathit{map}\ (\mathit{new2a}\ \mathit{as})
\end{array}
$$

The head of the sequence is 2; the heads of the difference sequences are 65, 162 and 132.

We call the function as $subst\ (2, 65, 162, 132)\ (cn\ H\ 5)$ and see

2, 65, 65, 162, 65, 162, 132, 162, 65, 162, 132, 162, 132, 132, 162, 65, 162, 132, 162, 132, 132, 162, 132, 132, 132, 162.

When we sum this together, $sum\ (subst\ (2, 65, 162, 132)\ (cn\ H\ 5))$, we get 3267, which is indeed the number appearing at position 5 in the sequence (starting to count from 0).

We implement one more function: ccn, for "count cn":

$$
\begin{aligned}
& ccn :: [Newton] \rightarrow (Int, Int, Int, Int) \\
& ccn\ ls = (length\ (filter\ (\equiv H)\ ls), \\
& \qquad\qquad length\ (filter\ (\equiv X)\ ls), \\
& \qquad\qquad length\ (filter\ (\equiv Y)\ ls), \\
& \qquad\qquad length\ (filter\ (\equiv Z)\ ls))
\end{aligned}
$$

When we apply this function, *e.g.* $ccn\ (cn\ H\ 3)$, we see:

$(1, 3, 3, 1)$

The binomial coefficients $\binom{3}{k}$, for $k \in \{0 \ldots 3\}$.

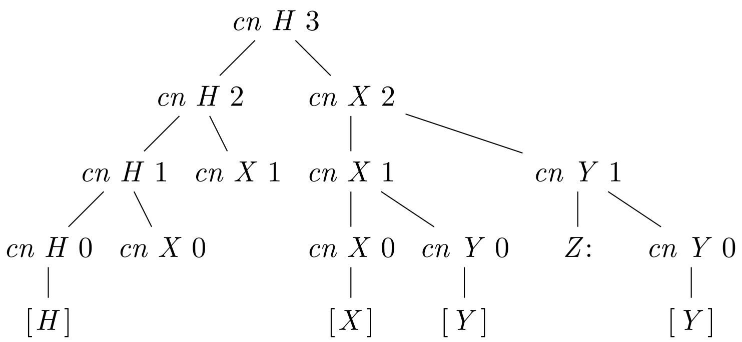
To see some more examples we call $map\ (ccn \circ cn\ H)\ [4\,..\,10]$ and get

$[(1, 4, 6, 4),$
$(1, 5, 10, 10),$
$(1, 6, 15, 20),$
$(1, 7, 21, 35),$
$(1, 8, 28, 56),$
$(1, 9, 36, 84),$
$(1, 10, 45, 120)]$

What we see, in terms of the table we used above, is

	0	1	2	3
	n_0	n_1	n_2	n_3
H	$\binom{0}{0}$	$\binom{1}{0}$	$\binom{2}{0}$	$\binom{3}{0}$
X	$\binom{0}{1}$	$\binom{1}{1}$	$\binom{2}{1}$	$\binom{3}{1}$
Y	$\binom{0}{2}$	$\binom{1}{2}$	$\binom{2}{2}$	$\binom{3}{2}$
Z	$\binom{0}{3}$	$\binom{1}{3}$	$\binom{2}{3}$	$\binom{3}{3}$

So, why do we see binomial coefficients and can we prove that we will always see binomial coefficients? To answer the first question, we will analyse the execution tree of cn. Here is the tree for $cn\ H\ 3$:



On the left-hand side of the tree, you see the main execution path calling $cn\ H\ (n-1)$ and $cn\ X\ (n-1)$ on each level. The sketch expands $cn\ X$ only for one case, namely the top-level call $cn\ X\ 2$ on the right-hand side. Otherwise, the tree would be quite confusing.

Anyway, what we can see:

- Any top-level call of type $cn\ A$ (for $A \in \{H, X, Y\}$) creates only one A; we therefore have always exactly one H.
- Every call to $cn\ H\ n$, for $n > 0$, calls one instance of $cn\ X$. We therefore have exactly $n\ X$.
- Every call to $cn\ X\ n$, for $n > 0$, calls one instance of $cn\ Y$. We therefore have exactly $n\ Y$ per $cn\ X\ n$, $n > 0$.
- Every call to $cn\ Y\ n$, for $n > 0$, creates one Z.
- The call to $cn\ X\ 1$ would expand to $cn\ X\ 0 \mathbin{+\!\!+} cn\ Y\ 0$; it would, hence, create one more X and one more Y.
- The call to $cn\ X\ 0$ would create one more X.
- This execution, thus, creates 1 H, 3 X, 3 Y and 1 Z.

We now prove by induction that if a call to $cn\ H\ n$ creates

$$\binom{n}{0}H, \binom{n}{1}X, \binom{n}{2}Y \text{ and } \binom{n}{3}Z$$

(and the previous calls to $cn\ H\ (n-1)$, $cn\ H\ (n-2)$, $\dots$, $cn\ H\ 0$ created similar patterns including the binomial coefficients), then $cn\ H\ (n+1)$ creates

$$\binom{n+1}{0}H, \binom{n+1}{1}X, \binom{n+1}{2}Y \text{ and } \binom{n+1}{3}Z.$$

Note that the number of H does not increase, because, as observed, each top-level call to *cn A n* creates exactly one A. If *cn H n* creates one H, *cn H* $(n+1)$ creates exactly one H, too. We conclude that we create $\binom{n+1}{0}H$ as requested.

When we call *cn H* $(n+1)$, we will call *cn H n*. We, therefore, create all instances of X created by *cn H n* plus those created in the first level of *cn H* $(n+1)$. This new level calls *cn X n* exaclty once, which creates one X (because any top-level call to *cn A n* creates exactly one A). We, hence, create one X more. This, however, is $\binom{n}{0} + \binom{n}{1} = \binom{n+1}{1}$ according to Pascal's Rule. We conclude that we create $\binom{n+1}{1}X$ as requested.

Since we call *cn H n*, when we call *cn H* $(n+1)$, we also create all instances of Y that were created by *cn H n*. We additionally create all instances of Y that are created by the new call to *cn X n*. This, in its turn, calls n instances of *cn Y*. Since $n = \binom{n}{1}$ and any top-level call to *cn Y n* creates exactly one Y, we create $\binom{n}{1} + \binom{n}{2} = \binom{n+1}{2}Y$ as requested.

Finally, since we call *cn H n*, when we call *cn H* $(n+1)$, we also create all instances of Z that were created before. But we call one more instance of *cn X n*, which creates a certain amount of new Z. How many? We create again all Z that were created anew by *cn H n*, those that did not exist in *cn H* $(n-1)$. Let us call the number of Z created by *cn H n* z_n and the number of Z created by *cn H* $(n-1)$ z_{n-1}. The number of Z created anew in *cn H n* is then $z_n - z_{n-1}$.

But since, in *cn H* $(n+1)$, we call *cn X* one level up, more Z are created than before. All calls to *cn Y* 0, those that did not create a new Z in *cn H n*, are now called as *cn Y* 1 and, hence, create a Z that was not created before. The calls to *cn Y* 0 create Y that were not created by *cn H* $(n-1)$. We, therefore, need to add to the number of Z the number of Y that did not exist in *cn H* $(n-1)$. We use the same convention as for Z, *i.e.* the number of Y created anew in *cn H n* is $y_n - y_{n-1}$. The number of additional Z created by the additional call to *cn X n*, hence, is

$$y_n - y_{n-1} + z_n - z_{n-1}$$

But we are dealing with binomial coefficients. We, therefore, have $z_n = y_{n-1} + z_{n-1}$ by Pascals' Rule applied backwards. When we substitute this back, we get

$$y_n - y_{n-1} + y_{n-1} + z_{n-1} - z_{n-1},$$

which simplifies to y_n, *i.e.* the number of instances of Y created by *cn H n*. In other words: the number of Z we additionally create in *cn H* $(n+1)$ is the number of Y in *cn H n*. So, the complete number of Z we have in *cn H* $(n+1)$ is the number of Y in *cn H n* plus the number Z in *cn H n*. Since the number of Y is $\binom{n}{2}$ and the number of Z is $\binom{n}{3}$, we now have $\binom{n}{2} + \binom{n}{3} = \binom{n+1}{3}$ according to Pascal's Rule as requested and this completes the proof. $\square$

1.5 Umbral Calculus

We saw that the differences and the derivative is not the same concept. Despite of many similarities, the polynomial of degree $n-1$ that generates the differences of a given polynomial of degree n is not necessarily the derivative of that polynomial. There is a class of polynomials, however, for which derivative, well, a very special kind of derivative, and differences are actually the same. Those are the *factorial polynomials*.

A factorial polynomial $x^{(n)}$ is a polynomial of the form

$$x^{(n)} = x(x-1)(x-2)\dots(x-n+1). \tag{1.14}$$

A factorial polynomial, hence, is generated by the *falling factorial* of x. The simplest factorial polynomial $x^{(1)}$ is

$$x^{(1)} = x. \tag{1.15}$$

The, arguably, even simpler than simplest factorial polynomial $x^{(0)}$ is, according to the definition of the factorials, 1.

Here is a Haskell function that shows the factors of the n^{th} factorial polynomial:

```
fpfacs :: (Integral a) => a -> [Poly a]
fpfacs 0 = [P [1]]
fpfacs n = [poly [-k,1] | k <- [0..n-1]]
```

Let us look at the first factorial polynomials:

fpfacs 0: $[P\ [1]]$
fpfacs 1: $[P\ [0,1]]$
fpfacs 2: $[P\ [0,1], P\ [-1,1]]$
fpfacs 3: $[P\ [0,1], P\ [-1,1], P\ [-2,1]]$
fpfacs 4: $[P\ [0,1], P\ [-1,1], P\ [-2,1], P\ [-3,1]]$
fpfacs 5: $[P\ [0,1], P\ [-1,1], P\ [-2,1], P\ [-3,1], P\ [-4,1]]$
fpfacs 6: $[P\ [0,1], P\ [-1,1], P\ [-2,1], P\ [-3,1], P\ [-4,1], P\ [-5,1]]$
fpfacs 7: $[P\ [0,1], P\ [-1,1], P\ [-2,1], P\ [-3,1], P\ [-4,1], P\ [-5,1], P\ [-6,1]]$

This suggests that the factorial polynomials, just as the factorials, can be defined recursively. The following equation describes the recursive formula:

$$x^{(n+1)} = (x-n)x^{(n)}, \tag{1.16}$$

which we can translate to Haskell as

$$
\begin{array}{l}
\mathit{rfacpoly} :: (\mathit{Integral}\ a) \Rightarrow a \rightarrow \mathit{Poly}\ a \\
\mathit{rfacpoly}\ 0 = P\ [1] \\
\mathit{rfacpoly}\ n = \mathit{mul}\ (\mathit{rfacpoly}\ (n-1))\ (P\ [-(n-1), 1])
\end{array}
$$

The recursive formula is, of course, not an efficient computing tool. For the factorial polynomial $x^{(n)}$, we would need n recursive steps, namely $x^{(n-1)}(x-n+1)$, $x^{(n-2)}(x-n+2)$, ..., $x^{(0)}x$. To compute, for instance, $n = 3$, we need to compute:

$$
\begin{array}{lllll}
x^{(1)} & = & x^{(0)}x & = & x \\
x^{(2)} & = & x(x-1) & = & (x^2 - x) \\
x^{(3)} & = & (x^2 - x)(x-2) & = & (x^3 - 3x^2 + 2x)
\end{array}
$$

A better way to compute the polynomial, once we have its factors, is to just multiply them out, like: *prodp mul*. The following implementation first creates the factors and then builds their product:

$$
\begin{array}{l}
\mathit{facpoly} :: (\mathit{Integral}\ a) \Rightarrow a \rightarrow \mathit{Poly}\ a \\
\mathit{facpoly} = \mathit{prodp}\ \mathit{mul} \circ \mathit{fpfacs}
\end{array}
$$

The two functions, *rfacpoly* and *facpoly*, create exactly the same result. When we apply one of them to $[1\mathinner{..}7]$ as above we get

facpoly 1: $P\ [0, 1]$
facpoly 2: $P\ [0, -1, 1]$
facpoly 3: $P\ [0, 2, -3, 1]$
facpoly 4: $P\ [0, -6, 11, -6, 1]$
facpoly 5: $P\ [0, 24, -50, 35, -10, 1]$
facpoly 6: $P\ [0, -120, 274, -225, 85, -15, 1]$
facpoly 7: $P\ [0, 720, -1764, 1624, -735, 175, -21, 1]$

which corresponds to the polynomials (in mathematical notation):

$$
\begin{array}{c}
x \\
x^2 - x \\
x^3 - 3x^2 + 2x \\
x^4 - 6x^3 + 11x^2 - 6x \\
x^5 - 10x^4 + 35x^3 - 50x^2 + 24x \\
x^6 - 15x^5 + 85x^4 - 225x^3 + 274x^2 - 120x \\
x^7 - 21x^6 + 175x^5 - 735x^4 + 1624x^3 - 1764x^2 + 720x
\end{array}
$$

Note, by the way, the last coefficient in each polynomial. Those are factorials. More precisely, the last coefficient of $x^{(n)}$ is $(n-1)!$. Does this pattern remind you of something? Not? Don't worry, we will look into it later.

Let us now turn to differences. Instead of just applying the polynomial to a sequence of numbers and then compute the differences, we could try to find a formula that expresses the differences for a given polynomial. When we take a formula like $x^{(3)}$, we can compute its differences by applying two consecutive values and compute the difference of the results, *e.g.*:

$$\begin{aligned} & 3^{(3)} - 2^{(3)} \\ = \; & (3^3 - 3 \times 3^2 + 2 \times 3) - (2^3 - 3 \times 2^2 + 2 \times 2) \\ = \; & (27 - 27 + 6) - (8 - 12 + 4) \\ = \; & 6 - 0 \\ = \; & 6. \end{aligned}$$

Instead of using concrete numbers, we can use a placeholder like a:

$$\begin{aligned} & (a+1)^{(3)} - a^{(3)} \\ = \; & ((a+1)^3 - 3(a+1)^2 + 2(a+1)) - (a^3 - 3a^2 + 2a) \\ = \; & ((a^3 + 3a^2 + 3a + 1) - (3a^2 + 6a + 3) + (2a + 2)) - (a^3 - 3a^2 + 2a) \\ = \; & (a^3 - a) - (a^3 - 3a^2 + 2a) \\ = \; & 3a^2 - 3a \end{aligned}$$

Let us test this result. We first apply $x^{(3)}$ on a sequence and compute the differences: *diffs* (*mapply* (*facpoly* 3) $[0\,..\,11]$. From this we get

0, 0, 6, 18, 36, 60, 90, 126, 168, 216, 270.

Now we apply the polynomial $3x^2 - 3x$ on the same sequence (minus one, because *diffs* has one element less than the sequence it is applied to): *mapply* (P $[0, -3, 3]$) $[0\,..\,10]$ and get

0, 0, 6, 18, 36, 60, 90, 126, 168, 216, 270.

The same sequence.

But what is so special about the result $3x^2 - 3x$ in the first place? Well, we can factor 3 out and get $3(x^2 - x)$, whose second part is $x^{(2)}$ and whose first part is $n = 3$. In other words, what we see here is that the differences of $x^{(n)}$ can be computed by the polynomial $nx^{(n-1)}$ and that formula is very similar to the concept of the derivative. Of course, it is not really the derivative, since the derivative of a polynomial deals with powers. The derivative of the polynomial x^n is, according to the power rule, nx^{n-1}. We see the same pattern here, but the exponent is not really an exponent, but a falling factorial.

A system that establishes a calculus that follows the same rules as the *infinitesimal calculus*, to which the derivative belongs, is often called *umbral calculus*. Most typical *umbral calculi* are systems of computations based on *Bernoulli polynomials* and *Bernoulli* *numbers.* But factorial polynomials, too, establish an *umbral calculus.*

Here is a Haskell function to compute the umbral derivative of the factorial polynomial $x^{(n)}$:

$$
\begin{aligned}
&\mathit{uderivative} :: (\mathit{Integral}\ a) \Rightarrow a \to \mathit{Poly}\ a\\
&\mathit{uderivative}\ n = \mathit{scale}\ n\ (\mathit{facpoly}\ (n-1))
\end{aligned}
$$

But we are moving fast. We have just looked at one special case, namely the differences of $x^{(3)}$. To be sure that the equation

$$\Delta_{x^{(n)}} = nx^{(n-1)}, \tag{1.17}$$

holds for all factorial polynomials, *i.e.* that the differences of $x^{(n)}$ equal $nx^{(n-1)}$, we first need to show it for the general case.

To do this, we start as above. We plug in the "value" a and compute the difference $\Delta_{a^{(n)}} = (a+1)^{(n)} - a^{(n)}$. When we expand the formula for the falling factorial, we get

$$
\begin{array}{rcccc}
\Delta_{a^{(n)}} & = & (a+1) & a(a-1)\dots(a-n+2) & \\
 & - & & a(a-1)\dots(a-n+2) & (a-n+1)
\end{array}
$$

On the right-hand side of this equation we see a middle part that is identical in both lines, namely $a(a-1)\dots(a-n+2)$, which is composed of the common factors of $(a+1)^{(n)}$ and $a^{(n)}$.

We zoom out to get a better overview of the equation by setting $b = a(a-1)\dots(a-n+2)$ and obtain:

$$\Delta_{a^{(n)}} = (a+1)b - (a-n+1)b. \tag{1.18}$$

By regrouping, we get $(a+1-a+n-1)b$. In the sum, we have a and $-a$ as well as 1 and -1. These terms, hence, cancel out and we are left with $\Delta_{a^{(n)}} = nb$. But b is $a(a-1)\dots(a-n+2)$, *i.e.* the same as the second line, but with one factor removed, namely $(a-n+1)$. That, however, is $a^{(n-1)}$ and, thus, we have

$$\Delta_{a^{(n)}} = na^{(n-1)}. \quad \square \tag{1.19}$$

This rule can be used to provide an elegant proof for Pascal's rule, which, as you may remember, states that

$$\binom{k+1}{n+1} = \binom{k}{n+1} + \binom{k}{n}. \tag{1.20}$$

We start by subtracting $\binom{k}{n+1}$ from both sides, obtaining

$$\binom{k+1}{n+1} - \binom{k}{n+1} = \binom{k}{n}. \tag{1.21}$$

This corresponds to

$$\frac{(k+1)^{(n+1)}}{(n+1)!} - \frac{k^{(n+1)}}{(n+1)!} = \binom{k}{n}. \tag{1.22}$$

When we join the fractions on the left-hand side, we get in the numerator the formula to compute the differences of $k^{(n+1)}$:

$$\frac{(k+1)^{(n+1)} - k^{(n+1)}}{(n+1)!} = \frac{\Delta_{k^{(n+1)}}}{(n+1)!}.$$

We have shown that $\Delta_{k^{(n+1)}} = (n+1)k^{(n)}$. If we substitute this back into the original equation, we see

$$\binom{k+1}{n+1} - \binom{k}{n+1} = \frac{(n+1)k^{(n)}}{(n+1)!}. \tag{1.23}$$

We now see in the fraction on the right-hand side that there is one factor that appears in numerator and denominator, namely $n+1$. When we cancel $n+1$ out we need to reduce $(n+1)!$ in the denominator by this factor. $(n+1)!$, however, is $(n+1)n!$. We therefore get:

$$\binom{k+1}{n+1} - \binom{k}{n+1} = \frac{k^{(n)}}{n!} = \binom{k}{n}. \quad \square \tag{1.24}$$

The difference between $x^{(n)}$ and x^n is, as already stated above, that the former is a falling factorial, while the latter is a power. Those are distinct concepts. For instance, x^2 is xx, while $x^{(2)}$ is $x(x-1)$. The falling factorial of n, hence, is smaller than the corresponding power n. We can even say precisely how much smaller it is. We just have to look at the list of factorial polynomials we have created above:

$$x^{(2)} = x(x-1) = x^2 - x. \tag{1.25}$$

So, we could express x^2 as $x^{(2)} + x$ adding the part that we subtract from x^2 to get $x^{(2)}$. If we wanted to express x^2 strictly in terms of falling factorials, we could say:

$$x^2 = x^{(2)} + x^{(1)}. \tag{1.26}$$

With the same technique, we can establish what x^3 is in terms of factorial polynomials. Since

$$x^{(3)} = x^3 - 3x^2 + 2x, \tag{1.27}$$

we have

$$x^3 = x^{(3)} + 3x^2 - 2x. \tag{1.28}$$

Using the previous result, we arrive at

$$x^3 = x^{(3)} + 3(x^{(2)} + x^{(1)}) - 2x^{(1)} = x^{(3)} + 3x^{(2)} + x^{(1)}. \tag{1.29}$$

For x^4, we have

$$x^4 = x^{(4)} + 6x^3 - 11x^2 + 6x \tag{1.30}$$

and, hence,

$$\begin{aligned} x^4 &= x^{(4)} + 6(x^{(3)} + 3x^{(2)} + x^{(1)}) - 11(x^{(2)} + x^{(1)}) + 6x^{(1)} \\ &= x^{(4)} + 6x^{(3)} + 7x^{(2)} + x^{(1)}. \end{aligned}$$

In this way, we can go on and create formulas for all powers (and, once we have shown that we can express powers by factorial polynomials, we can show that we can represent polynomials as factorial polynomials). We can even show that each power has a unique representation as sum of factorial polynomials, just as each number has a unique representation as product of prime numbers.

To prove this, suppose that, for a power x^n, there were two different representations as sums of factorial polynomials, such that

$$\begin{aligned} x^n &= A_1x^{(1)} + A_2x^{(2)} + \dots + A_nx^{(n)} \\ &= B_1x^{(1)} + B_2x^{(2)} + \dots + B_nx^{(n)}. \end{aligned} \tag{1.31}$$

When we subtract one representation from the other, the result shall be zero, since both represent the same value x^n. So, we have:

$$A_1x^{(1)} + A_2x^{(2)} + \dots + A_nx^{(n)} - B_1x^{(1)} + B_2x^{(2)} + \dots + B_nx^{(n)} = 0. \tag{1.32}$$

Regrouping we get

$$(A_1 - B_1)x^{(1)} + (A_2 - B_2)x^{(2)} + \cdots + (A_n - B_n)x^{(n)} = 0. \tag{1.33}$$

There are two ways for this sum to become zero. Either the $A_n - B_n$ parts are all zero or the $x^{(n)}$ parts are all zero (or, of course, in some cases it is like this and in others like that). The value of $x^{(n)}$, however, depends on the value to which we apply the polynomial. But the formula requires that the sum is zero for any value we may fill in for x. We are therefore left with the first option: the $A_n - B_n$ parts must be zero. These differences, however, are zero only if $A_n = B_n$. That shows that the two representations are equal. □

We have proved that powers can be represented uniquely by factorial polynomials. Here is a list of representations of powers (starting with x^1 in the first line) as factorial polynomials:

$$\begin{gathered}
x^{(1)}\\
x^{(2)} + x^{(1)}\\
x^{(3)} + 3x^{(2)} + x^{(1)}\\
x^{(4)} + 6x^{(3)} + 7x^{(2)} + x^{(1)}\\
x^{(5)} + 10x^{(4)} + 25x^{(3)} + 15x^{(2)} + x^{(1)}\\
x^{(6)} + 15x^{(5)} + 65x^{(4)} + 90x^{(3)} + 31x^{(2)} + x^{(1)}\\
x^{(7)} + 21x^{(6)} + 140x^{(5)} + 350x^{(4)} + 301x^{(3)} + 63x^{(2)} + x^{(1)}
\end{gathered}$$

Those of you who still suffer from triangle paranoia: you have probably realised that this is already the second triangle appearing in this section. When you scroll back to certain triangle-intense chapters, you will recognise the coefficients above as *Stirling numbers of the second kind.* Of course the table above is inverted, because we start with the largest k in $x^{(k)}$ going down to $k = 1$, while the triangle for the Stirling numbers shows the coefficients in the order $\left\{ {n \atop 1} \right\} \dots \left\{ {n \atop n} \right\}$. As a reminder, here they are:

1							1						
2						1		1					
3					1		3		1				
4				1		7		6		1			
5			1		15		25		10		1		
6		1		31		90		65		15		1	
7	1		63		301		350		140		21		1

Well, we see for some cases that the numbers by which we scale factorial polynomials so that they sum up to powers are Stirling numbers. Can we prove it for all cases?

Let's give it a try with a proof by induction. Any of the examples above serves as base case that shows that

$$x^n = \left\{ {n \atop n} \right\} x^{(n)} + \left\{ {n \atop n-1} \right\} x^{(n-1)} + \dots + \left\{ {n \atop 1} \right\} x^{(1)}. \tag{1.34}$$

We need to show that, if this equation holds for x^n, it holds for x^{n+1} that

$$x^{n+1} = \left\{ {n+1 \atop n+1} \right\} x^{(n+1)} \left\{ {n+1 \atop n} \right\} x^{(n)} + \dots + \left\{ {n+1 \atop 1} \right\} x^{(1)} \tag{1.35}$$

We start with the base case and multiply x on both sides. On the left-hand side, we get x^{n+1}. But what do we get on the right-hand side? Well, for each term $x^{(k)}$, we get $xx^{(k)}$. We have never really thought about what the result of $xx^{(k)}$ is. We only know that $(x-k)x^{(k)} = x^{(k+1)}$. So, let us stick to what we know and try to get it in. A simple way is to express x as an expression with a cameo of $x - k$, for instance: $x = x - k + k$. With this expression, we have $(x - k + k)x^{(k)}$. We distribute $x^{(k)}$ over the sum and get

$$(x-k)x^{(k)} + kx^{(k)} = x^{(k+1)} + kx^{(k)}.$$

On the right-hand side, we, hence, get such a sum for each term:

$$\left\{ {n+1 \atop n+1} \right\} \left(x^{(n+1)} + nx^{(n)} \right) + \left\{ {n+1 \atop n} \right\} \left(x^{(n)} + (n-1)x^{(n-1)} \right) + \dots + \left\{ {n+1 \atop 1} \right\} \left(x^{(2)} + x^{(1)} \right)$$

We can now regroup the terms, so that the elements with equal "exponents" appear together. This yields pairs composed of the $x^{(k)}$ that was already there and the new one that we generated by multiplying by x:

$$\begin{aligned} \textstyle\left\{ {n \atop n} \right\} x^{(n+1)} & \quad + \\ n\textstyle\left\{ {n \atop n} \right\} x^{(n)} + \left\{ {n \atop n-1} \right\} x^{(n)} & \quad + \\ (n-1)\textstyle\left\{ {n \atop n-1} \right\} x^{(n-1)} + \left\{ {n \atop n-2} \right\} x^{(n-1)} & \quad + \\ \dots & \quad + \\ \textstyle\left\{ {n \atop 1} \right\} x^{(1)} & \end{aligned}$$

We regroup a bit more, in particular, we factor $x^{(k)}$ out, so that we obtain factors that consist only of expressions containing Stirling numbers in front of the $x$es:

$$
\begin{array}{rll}
\left\{{n \atop n}\right\} & x^{(n+1)} & + \\
\left(n\left\{{n \atop n}\right\} + \left\{{n \atop n-1}\right\}\right) & x^{(n)} & + \\
\left((n-1)\left\{{n \atop n-1}\right\} + \left\{{n \atop n-2}\right\}\right) & x^{(n-1)} & + \\
\dots & & + \\
\left(2\left\{{n \atop 2}\right\} + \left\{{n \atop 1}\right\}\right) & x^{2} & + \\
\left\{{n \atop 1}\right\} & x^{(1)} &
\end{array}
$$

You might remember the identity

$$\left\{{n+1 \atop k+1}\right\} = k\left\{{n \atop k+1}\right\} + \left\{{n \atop k}\right\}, \tag{1.36}$$

which is "Pascal's rule" for Stirling numbers of the second kind. This is exactly what we see in each group! Compare the factors in front of the first Stirling number that read n, $n-1$ and so on with what you see in the Stirling number in the place of k (*i.e.* in the bottom). For instance, in the formula

$$\left((n-1)\left\{{n \atop n-1}\right\} + \left\{{n \atop n-2}\right\}\right) x^{(n-1)}$$

we have $k = n - 1$.

Now, all terms that show this pattern, can be simplified to

$$\left\{{n+1 \atop k+1}\right\}$$

leaving only the first and the last term. But since the first and the last are $\left\{{n \atop n}\right\}$ and $\left\{{n \atop 1}\right\}$ respectively, which are both just 1, that is not a problem. We get as desired

$$x^{n+1} = \left\{{n+1 \atop n+1}\right\}x^{(n+1)} + \left\{{n+1 \atop n}\right\}x^{(n)} + \dots + \left\{{n+1 \atop 1}\right\}x^{(1)} \quad \square \tag{1.37}$$

and that completes the proof.

The following function exploits Stirling numbers to compute powers by means of factorial polynomials:

$$\begin{array}{l}
\mathit{stirpow} :: \mathit{Natural} \to \mathit{Poly\ Natural}\\
\mathit{stirpow}\ n = \mathit{sump}\ [\mathit{scale}\ (\mathit{Perm.stirling2}\ n\ k)\ (\mathit{facpoly}\ k) \mid k \leftarrow [1\mathinner{..}n]]
\end{array}$$

This is a lame function, of course. Powers are not difficult to compute at all, so why using factorial polynomials in the first place? More interesting, at least from theoretical perspective, is the opposite function that, for a given power, shows the factorial polynomials and the coefficients that indicate how often each factorial polynomial appears:

$$\begin{array}{l}
\mathit{fpPowTerms} :: \mathit{Natural} \to [(\mathit{Natural}, \mathit{Poly\ Natural})]\\
\mathit{fpPowTerms}\ 0 = [(1, P\ [1])]\\
\mathit{fpPowTerms}\ n = [(\mathit{Perm.stirling2}\ n\ k, \mathit{facpoly}\ k) \mid k \leftarrow [1\mathinner{..}n]]
\end{array}$$

The function, just like the previous one, makes use of the *stirling2* function that we defined in the first chapter and so we are obliged to use the concrete type *Natural*.

Here is a function to test the results:

$$\begin{array}{l}
\mathit{sumFpPolyTerms} :: [(\mathit{Integer}, \mathit{Poly\ Integer})] \to \mathit{Poly\ Integer}\\
\mathit{sumFpPolyTerms} = \mathit{sump} \circ \mathit{map}\ (\mathit{uncurry\ scale})
\end{array}$$

The function, basically, just sums up the list we pass in scaling the polynomials by their coefficient. Here is a test for the first 7 powers,
$\mathit{map}\ (\mathit{sumFpPolyTerms} \circ \mathit{fpPowTerms})\ [0\mathinner{..}6]$:

$$\begin{array}{c}
P\ [1]\\
P\ [0,1]\\
P\ [0,0,1]\\
P\ [0,0,0,1]\\
P\ [0,0,0,0,1]\\
P\ [0,0,0,0,0,1]\\
P\ [0,0,0,0,0,0,1]\\
P\ [0,0,0,0,0,0,0,1]
\end{array}$$

Once we can represent powers by factorial polynomials, we are able to represent any polynomial by factorial polynomials, since polynomials are just sums of scaled powers of x. Here is a function that does that:

$$\begin{array}{lll}
\multicolumn{3}{l}{\mathit{fpPolyTerms} :: \mathit{Poly\ Natural} \to [(\mathit{Natural}, \mathit{Poly\ Natural})]}\\
\multicolumn{3}{l}{\mathit{fpPolyTerms}\ (P\ cs) = [\mathit{foldl}\ \mathit{ab}\ \mathit{p0}\ p \mid p \leftarrow \mathit{p2}]}\\
\quad\mathbf{where}\ \mathit{p0} & & = (0, P\ [0])\\
\quad\quad \mathit{p1} & & = \mathit{concat}\ [\mathit{map}\ (s\ c)\ (\mathit{fpPowTerms}\ k) \mid (c,k) \leftarrow \mathit{zip}\ \mathit{cs}\ [0\mathinner{..}]]\\
\quad\quad \mathit{p2} & & = \mathit{groupBy}\ ((\equiv)\ `\mathit{on}`\ \mathit{snd})\ (\mathit{sortOn}\ (\mathit{degree} \circ \mathit{snd})\ \mathit{p1})\\
\quad\quad \mathit{ab}\ a\ b & & = (\mathit{fst}\ a + \mathit{fst}\ b, \mathit{snd}\ b)\\
\quad\quad s\ c\ (n,p) & & = (c * n, p)
\end{array}$$

The function looks a bit confusing on the first sight. It is not too horrible, though. We start by computing $p1$. We apply *fpPowTerms* on the exponents of the original polynomial ($[0\mathinner{..}]$) and multiply the coefficients of the original (*cs*) and the coefficients that tell us how often each factorial polynomial occurs in the respective power. The latter is done by function s which is mapped on the result of *fpPowTerms*. The result is a list of lists of pairs (n, p), where n is a *Natural* and p a polynomial. We concat this list, so we obtain a flat list of such pairs.

In the next step, we compute $p2$ by sorting and then grouping this flat list by the degree of the polynomials. The result is a list of lists of polynomials of equal degree with differing coefficients.

In the final step we sum the coefficients of each such groups starting with zero $p0 = (0, P\ [0])$.

We test this function by factoring arbitrary polynomials into their terms and summing the result together again:

sumFpPolyTerms (*fpPolyTerms* (P [0, 0, 0, 0, 1]))
P [0, 0, 0, 0, 1]
sumFpPolyTerms (*fpPolyTerms* (P [1, 1, 1, 1, 1]))
P [1, 1, 1, 1, 1]
sumFpPolyTerms (*fpPolyTerms* (P [5, 4, 3, 2, 1]))
P [5, 4, 3, 2, 1]
sumFpPolyTerms (*fpPolyTerms* (P [1, 2, 3, 4, 5]))
P [1, 2, 3, 4, 5]

In the next experiment we retrieve the coefficients for polynomials of the form

$$x^n + x^{n-1} + \cdots + 1,$$

i.e. polynomials with all coefficient equal to 1.

We apply *map* (*map fst* $\circ$ *fpPolyTerms*) to the first 7 polynomials of that form, *i.e.* 1, $x + 1$, $x^2 + x + 1$ and so on and get

$$\begin{array}{c} [1] \\ [1, 1] \\ [1, 2, 1] \\ [1, 3, 4, 1] \\ [1, 4, 11, 7, 1] \\ [1, 5, 26, 32, 11, 1] \\ [1, 6, 57, 122, 76, 16, 1] \end{array}$$

This again is a triangle and it is the simplest that we can obtain this way, since the input coefficients are all 1. One could think that other polynomials could now be generated by means of these coefficients just multiplying the coeficients of the polynomial with these ones. Unfortunately, that is too simple. The coefficients here indicate only how often each factorial polynomial appears in the respective polynomial; they are not coefficients of that polynomial (which are all 1 anyway).

The sequence as such is the result of a matrix multiplication (a topic we will study soon) with one matrix being a lower-left triangle of ones and the other a lower-left triangle containing the Stirling numbers of the second kind:

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 3 & 1 & 0 & 0 \\ 1 & 7 & 6 & 1 & 0 \\ 1 & 15 & 25 & 10 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 & 0 \\ 3 & 4 & 1 & 0 & 0 \\ 4 & 11 & 7 & 1 & 0 \\ 5 & 26 & 32 & 11 & 1 \end{pmatrix} \tag{1.38}$$

Meanwhile, you may have guessed or even verified that the coefficients of factorial polynomials, those appearing in the very first triangle in this section, are Stirling numbers of the first kind. But they are special: some are negative. Indeed, there are two variants of the Stirling numbers of the first kind: signed and unsigned. Since we were discussing combinatorial problems related to permutations, when we first introduced Stirling numbers, we did not consider the signed variety. Here are the signed Stirling numbers of the first kind:

1							1						
2						-1		1					
3					2		-3		1				
4				-6		11		-6		1			
5			24		-50		35		-10		1		
6		-120		274		-225		85		-15		1	
7	720		-1764		1624		-735		175		-21		1

The recursive formula to compute these numbers is

$$\begin{bmatrix} n+1 \\ k+1 \end{bmatrix} = -n \begin{bmatrix} n \\ k+1 \end{bmatrix} + \begin{bmatrix} n \\ k \end{bmatrix}. \tag{1.39}$$

Note that, when the first Stirling number, $\left[{n \atop k}\right]$, on the right-hand side is positive, then the second, $\left[{n \atop k-1}\right]$, is negative. Since we multiply the first by a negative number, the first term becomes positive, when the Stirling number is negative and negative otherwise. Therefore, both terms are either negative or positive and the absolute value of the whole expression does not change compared to the unsigned Stirling number.

So, can we prove that the coefficients of factorial polynomials are Stirling numbers of the first kind? We prove by induction with any of the above given polynomials as base case

$$x^{(n)} = \begin{bmatrix} n \\ n \end{bmatrix} x^n + \begin{bmatrix} n \\ n-1 \end{bmatrix} x^{n-1} + \dots + \begin{bmatrix} n \\ 1 \end{bmatrix} x, \tag{1.40}$$

where the Stirling numbers, here, are to be understood as signed.

We need to prove that, if that equation holds, then the following holds as well:

$$x^{(n+1)} = \begin{bmatrix} n+1 \\ n+1 \end{bmatrix} x^{n+1} + \begin{bmatrix} n+1 \\ n \end{bmatrix} x^{n} + \cdots + \begin{bmatrix} n+1 \\ 1 \end{bmatrix} x. \tag{1.41}$$

We start with the observation that

$$x^{(n+1)} = (x-n)x^{(n)}. \tag{1.42}$$

So, we can go from 1.40 to 1.41 by multiplying both sides of 1.40 by $x-n$. The right-hand side would then become:

$$(x-n)\begin{bmatrix} n \\ n \end{bmatrix} x^{n} + (x-n)\begin{bmatrix} n \\ n-1 \end{bmatrix} x^{n-1} + \cdots + (x-n)\begin{bmatrix} n \\ 1 \end{bmatrix} x.$$

For each term, we distribute the factors over the sum $x-n$:

$$\begin{bmatrix} n \\ n \end{bmatrix} x^{n+1} - n\begin{bmatrix} n \\ n \end{bmatrix} x^{n} + \begin{bmatrix} n \\ n-1 \end{bmatrix} x^{n} - n\begin{bmatrix} n \\ n-1 \end{bmatrix} x^{n-1} + \cdots + \begin{bmatrix} n \\ 1 \end{bmatrix} x^{2} - n\begin{bmatrix} n \\ 1 \end{bmatrix} x$$

and regroup so that we get pairs of terms with equal $x$es:

$$\begin{array}{cccccccc}
 & \left[\begin{smallmatrix} n \\ n \end{smallmatrix}\right] & x^{n+1} & & & & +\\
-n & \left[\begin{smallmatrix} n \\ n \end{smallmatrix}\right] & x^{n} & + & \left[\begin{smallmatrix} n \\ n-1 \end{smallmatrix}\right] & x^{n} & +\\
-n & \left[\begin{smallmatrix} n \\ n-1 \end{smallmatrix}\right] & x^{n-1} & + & \left[\begin{smallmatrix} n \\ n-2 \end{smallmatrix}\right] & x^{n-1} & +\\
 & & & & \dots & & +\\
-n & \left[\begin{smallmatrix} n \\ 2 \end{smallmatrix}\right] & x^{2} & + & \left[\begin{smallmatrix} n \\ 1 \end{smallmatrix}\right] & x^{2} & +\\
-n & \left[\begin{smallmatrix} n \\ 1 \end{smallmatrix}\right] & x & & & &
\end{array}$$

When we factor the $x$es out again, we get

$$
\begin{array}{rll}
{n \brack n} & x^{n+1} & + \\
\left(-n{n \brack n} + {n \brack n-1}\right) & x^n & + \\
\left(-n{n \brack n-1} + {n \brack n-2}\right) & x^{n-1} & + \\
\dots & & + \\
\left(-n{n \brack 2} + {n \brack 1}\right) & x^2 & + \\
-n{n \brack 1} & x &
\end{array}
$$

In each line but the first and the last, we now have the formula to compute ${n+1 \brack k+1}$ and can simplify all these lines accordingly:

$$
\begin{bmatrix} n \\ n \end{bmatrix} x^{n+1} + \begin{bmatrix} n+1 \\ n \end{bmatrix} x^n + \begin{bmatrix} n+1 \\ n-1 \end{bmatrix} x^{n-1} + \dots + \begin{bmatrix} n+1 \\ 2 \end{bmatrix} x^2 - n \begin{bmatrix} n \\ 1 \end{bmatrix} x
$$

For the first term, the same argument we already used before still holds: ${n \brack n} = {n+1 \brack n+1} = 1$.

For the last term, we know that ${n \brack 1} = \pm(n-1)!$. We hence see the product $(-n)(\pm((n-1)!))$, which is $-(\pm(n!))$. If, for n, the factorial was positive, it will now be negative. If it was negative, it will now be positive. This complies with the signed Stirling numbers of the first kind and completes the proof. $\square$

What have we learnt in the last sections? Well, factorial polynomials have coefficients that count the number of permutations that can be expressed by a given number of cycles. When factorial polynomials are used to represent powers, we need to scale them by factors that count the number of ways to partition a set into a given number of distinct subsets.

Furthermore, we can express any polynomial by combinations of scaled factorial polynomials and the coefficients of those are products of the differences and the binomial coefficients which count the number of ways to choose k out of n. "The Lord is subtle" said Einstein, "but he is not plain mean". That is a quantum of solace for us mere mortals! Let us go on to see what is there more to discover.

1.6 Roots

In the previous sections, we looked at the results, when applying polynomials to given values. That is, we applied a polynomial $\pi(x)$ to a given value (or sequence of values) for x and studied the result $y = \pi(x)$. Now we are turning this around. We will look at a given y and ask which value x would create that y. In other words, we look at polynomials as equations of the form:

$$a_n x^n + a_{n-1} x^{n-1} + \cdots + a_0 = a \tag{1.43}$$

and search for ways to solve such equations. In the focus of this investigation is usually the special case $a = 0$, *i.e.*

$$a_n x^n + a_{n-1} x^{n-1} + \cdots + a_0 = 0. \tag{1.44}$$

The values for x fulfilling this equation are called the *root*s of the polynomial. A trivial example is x^2, whose root is 0. A slightly less trivial example is $x^2 - 4$, whose roots are $x_1 = -2$ and $x_2 = 2$, since

$$(-2)^2 - 4 = 4 - 4 = 0$$

and

$$2^2 - 4 = 4 - 4 = 0.$$

Note that these examples are polynomials of even degree. Polynomials of even degree do not need to have any roots. Since even powers are always positive (or zero), negative values are turned into positive numbers and, since the term of highest degree is even, the whole expression may always be positive. This is true for the polynomial $x^2 + 1$. Since all negative values are transformed into positive values by x^2, the smallest value that we can reach is the result for $x = 0$, which is $0 + 1 = 1$.

On the other hand, even polynomials may have negative values, namely when they have terms with coefficients that, for small absolute values, result in negative numbers whose absolute value is greater than those resulting from the term of highest degree. The polynomial $x^2 - 4$, once again, is negative in the interval $] - 2 \dots 2[$. It, therefore, must have two roots: one at -2, where the polynomial results become negative, and the other at 2, where the polynomial results become positive again.

Odd polynomials, by contrast, usually have negative values, because the term with the highest degree may result in a negative or a positive number depending on the signedness of the input value and that of the coefficient. The trivial polynomial x^3, for instance, is negative for negative values and positive for positive values. The slightly less trivial polynomial $x^3 + 27$ has a root at -3, while $x^3 - 27$ has a root at 3.

In summary, we can say that even polynomials do not necessarily have negative values and, hence, do not need to have a root. Odd polynomials, on the other hand, usually have both, negative and positive values, and, hence, must have a root.

Those are strong claims. They are true, because polynomials belong to a specific class of *functions*, namely *continuous* functions. That, basically, means that they have no *holes*, *i.e.* for any value x of a certain number type there is a result y of that number type. For instance, when the coefficients of the polynomial are all integers and the x-value is an integer, then the result is an integer, too. When the polynomial is defined over a field (all coefficients are part of that field and the values to which we apply the polynomial lie in that field), then the result is in that field, too. Rational polynomials, for instance, have rational results. Real polynomials have real results.

Furthermore, the function does not "jump", *i.e.* the results grow with the input values – not necessarily at the same rate, in fact, for polynomials of degree greater than 1, the result grows much faster than the input – but the growth is regular.

These properties appear to be "natural" at the first sight. But there are functions that do not fulfil these criteria. In the next chapter, when we properly define the term *function*, we will actually see functions with holes and jumps.

The reason that polynomials behave regularly is that we only use basic arithmetic operations in their definition: we add, multiply and raise to powers. All those operations are closed, *i.e.* their results lie in the same fields as their inputs.

Furthermore, the form of polynomials guarantees that they develop in a certain way. For very large numbers (negative or positive), it is the term with the greatest exponent, *i.e.* the degree of the polynomial, that most significantly determines the outcome, that is, the result for very large numbers approaches the result for the term with the largest exponent. For smaller values, however, the terms of lower degree have stronger impact. The terms "large" and "small", here, must be understood relative to the coefficients. If the coefficients are very large, the values to which the polynomial is applied must be even larger to approach the result for the first term.

There are polynomials whose behaviour is hard to predict, for instance, *Wilkinson's polynomial* named for James Hardy Wilkinson (1919 – 1986), an American mathematician and computer scientist. The Wilkinson polynomial is defined as

$$w(x) = \prod_{i=1}^{20} (x - i). \tag{1.45}$$

It is thus a factorial polynomial, namely $x^{(21)}$. We can generate it in terms of our polynomial type as

$$\begin{aligned}
&\mathit{wilkinson} :: (\mathit{Num}\ a, \mathit{Enum}\ a, \mathit{Show}\ a, \mathit{Eq}\ a) \Rightarrow \mathit{Poly}\ a\\
&\mathit{wilkinson} = \mathit{prodp}\ \mathit{mul}\ [P\ [-i, 1] \mid i \leftarrow [1\mathinner{..}20]]
\end{aligned}$$

It looks like this:

P [
$2432902008176640000, -8752948036761600000, 13803759753640704000,$
$-12870931245150988800, 8037811822645051776, -3599979517947607200,$
$1206647803780373360, -311333643161390640, 63030812099294896,$
$-10142299865511450, 1307535010540395, -135585182899530, 11310276995381,$
$-756111184500, 40171771630, -1672280820, 53327946, -1256850, 20615, -210, 1]$

The first terms are

$$x^{20} - 210x^{19} + 20615x^{18} - 1256850x^{17} \ldots$$

When we apply Wilkinson's polynomial to the integers $1 \ldots 25$, we see:

$0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2432902008176640000,$
$51090942171709440000, 562000363888803840000, 4308669456480829440000,$
$25852016738884976640000,$

which looks very confusing. When we try non-integers, we see

apply wilkinson 0.9 is $1.7213\ldots$
apply wilkinson 1.1 is $-8.4600\ldots$
apply wilkinson 1.9 is $-8.1111\ldots$
apply wilkinson 2.1 is $4.9238\ldots$

As we see, the results switch sign at the integers or, more precisely, at the integers in the interval $[1 \ldots 20]$, which are the roots of Wilkinson's polynomial. Looking at the factors of the polynomial

$$(x-1)(x-2)\ldots(x-20),$$

this result is much less surprising, since, obviously, when any of these factors becomes 0, then the whole expression becomes 0. So, for the value $x = 3$, we would have

$$2 \times 1 \times 0 \times \cdots \times -17 = 0.$$

When we look at the coefficients, however, the results look quite irregular and, on the first sight, completely unrelated. When we say that polynomials show a regular behaviour, that must be taken with a grain of salt. Anyway, that they behave like this gives rise to a number of simple methods to find roots based on approximation, at least when we start with a fair guess, which requires some knowledge about the rough shape of the polynomial in the first place.

These methods can be split into two major groups: *bracketing* methods and *open* methods. Bracketing methods start with two distinct values somewhere on the "left" and the "right" of the root. Bracketing methods, hence, require a pre-knowledge about where, more or less, a root is located.

The simplest variant of bracketing is the *bisect* algorithm. It is very similar to Heron's method to find the square root of a given number. We start with two values a and b and, on each step, we compute the average $(a + b)/2$ and substitute either a or b by this value depending on the side the value is located relative to the root. Here is an implementation:

$$
\begin{array}{l}
\mathit{bisect} :: (\mathit{Num}\ a, \mathit{Eq}\ a, \mathit{Ord}\ a, \mathit{Fractional}\ a, \mathit{Show}\ a) \\
\qquad\quad \Rightarrow \mathit{Poly}\ a \to a \to a \to a \to a \\
\mathit{bisect}\ p\ t\ a\ b \quad \begin{array}{ll} \mid \mathit{abs}\ \mathit{fc} < \mathit{abs}\ t & = c \\ \mid \mathit{signum}\ \mathit{fc} \equiv \mathit{signum}\ \mathit{fa} & = \mathit{bisect}\ p\ t\ c\ b \\ \mid \mathit{otherwise} & = \mathit{bisect}\ p\ t\ a\ c \end{array} \\
\quad \mathbf{where}\ \begin{array}{ll} \mathit{fa} & = \mathit{apply}\ p\ a \\ \mathit{fb} & = \mathit{apply}\ p\ b \\ \mathit{fc} & = \mathit{apply}\ p\ c \\ c & = (a + b)\ /\ 2 \end{array}
\end{array}
$$

The function receives four arguments. The first is the polynomial. The second is a tolerance. When, on applying the polynomial, we get a result that is smaller than the tolerance, we return the obtained x value. a and b are the starting values.

We distinguish three cases:

- The result for the new value, c, is below the tolerance threshold. In this case, c is sufficiently close to the root and we yield this value.
- the sign of the result for the new value equals the sign of a. Then we replace a by c.
- the sign of the result for the new value equals the sign of b. In this case, we replace b by c.

We try *bisect* on the polynomial x^2 with the initial guess $a = -1$ and $b = 1$ (because we assume that the root should be close to 0) and a tolerance of 0.1:

$\mathit{bisect}\ (P\ [0, 0, 1])\ 0.1\ (-1)\ 1$

and see the correct result 0.0.

For the polynomial $x^2 - 4$, which has two roots, we try

$\mathit{bisect}\ (P\ [-4, 0, 1])\ 0.1\ (-3)\ (-1)$,

which yields -2 and

bisect (P $[-4, 0, 1]$) 0.1 1 3,

which yields 2.

With Wilkinson's polynomial, however, we get a surprise:

bisect wilkinson 0.1 0.5 1.5,

for which we expect to find the root 1. But the function does not return. Indeed, when we try *apply wilkinson* 1.0, we see

1148.0,

a somewhat surprising result. Wilkinson used this polynomial to demonstrate the sensivity of coefficients to small differences in the input values. Using Haskell real numbers, The computation leads to a loss of precision in representing the terms. Indeed, considering terms raised to the 20^{th} power and multiplied by large coefficients, the number 1148 appears to be a tiny imprecision.

We can work around this, using rational numbers:

apply wilkinson (1 % 1)

gives without any surprise 0 % 1. So, we try

bisect wilkinson (1 % 10) (1 % 2) (3 % 2)

and get the correct result 1 % 1. The function with these parameters returns almost instantly. That is because the average of 0.5 and 1.5 is already 1. The function finds the root in the first step. A more serious challenge is

bisect wilkinson (1 % 10) (1 % 3) (3 % 2),

which needs more than one recursion. The function, now, runs for a short while and comes up with the result

1729382256910270463 % 1729382256910270464,

which is pretty close to 1 and, hence, the correct result.

Open methods need only one value. The most widely known open method is Newton's method, also called Newton-Raphson method. It was first developed by Newton in about 1670 and then, in 1690, again by Joseph Raphson. Newton's version was probably not known to Raphson, since Newton did not publish his work. Raphson's version, on the other hand, is simpler and, therefore, usually preferred.

Anyway, the method starts with only one approximation and is therefore not a bracketing method. The approximation is then applied to the polynomial π and the derivative of that polynomial, π'. Then, the quotient of the results, $\frac{\pi(x)}{\pi'(x)}$ is computed and subtracted from the initial guess. Here is an implementation:

$$
\begin{aligned}
&\mathit{newguess} :: (\mathit{Num}\ a, \mathit{Eq}\ a, \mathit{Ord}\ a, \mathit{Enum}\ a, \mathit{Fractional}\ a) \\
&\qquad\qquad \Rightarrow \mathit{Poly}\ a \to \mathit{Natural} \to a \to a \to a \\
&\mathit{newguess}\ p\ m\ t\ a \mid \mathit{abs}\ pa < t = a \\
&\qquad\qquad\qquad\quad \mid m \leqslant 0 \qquad\ = a \\
&\qquad\qquad\qquad\quad \mid \mathit{otherwise}\ = \mathit{newguess}\ p\ (m-1)\ t\ (a - pa\ /\ p'a) \\
&\quad \textbf{where}\ p' \ = \mathit{derivative}\ (*)\ p \\
&\qquad\qquad pa \ = \mathit{apply}\ p\ a \\
&\qquad\qquad p'a = \mathit{apply}\ p'\ a
\end{aligned}
$$

The function receives four parameters. The polynomial p, the natural number m, the tolerance t and the initial guess a. The natural number m is a delimiter. It is not guaranteed that the value increases in precision with always more repetitions. It may get worse at some point. It is therefore useful – and a lesson learnt from experimenting with *bisect* – to restrict the number of iterations.

The function terminates when we have reached either the intended precision or the number of repetitions, m. Otherwise, we repeat with $m - 1$ and $a - \frac{\pi(a)}{\pi'(a)}$.

For the polynomial $x^2 - 4$, we call first

$\mathit{newguess}\ (P\ [-4, 0, 1])\ 10\ 0.1\ 1$

and get $2.00069\ldots$, which is very close to the known root 2. For the other root we call

$\mathit{newguess}\ (P\ [-4, 0, 1])\ 10\ 0.1\ (-1)$

and get the equally close result $-2.00069\ldots$ For the Wilkinson polynomial, we call

$\mathit{newguess}\ \mathit{wilkinson}\ 10\ (0.0001)\ 1.5$

and get $1.99999\ldots$, which is very close to the real root 2. We can further improve precision by increasing the number of iterations:

$\mathit{newguess}\ \mathit{wilkinson}\ 20\ (0.0001)\ 1.5$

The difference is at the 12^{th} decimal digit.

Note that the Newton-Raphson method is not only more precise (that is: converges earlier with a good result), but also more robust against real representation imprecision.

To understand why this method works at all, we need to better understand what the derivative is. We will come back to this issue in the next chapter. In the strict sense, the derivative does not belong here anyway, since the concept of derivative is analysis, not algebra. Both kinds of methods, the bracketing and the open methods, in fact, come from numerical analysis. They do not have the "look and feel" of algebraic methods. So, how would an algebraist tackle the problem of finding the roots of a polynomial?

One possibility is factoring. Polynomials may be represented as the product of their factors (just like integers). We have experienced with Wilkinson's polynomial that the factor representation may be much more convenient than the usual representation with coefficients. Wilkinson's polynomial expressed as a product was just

$$w(x) = \prod_{i=1}^{20} (x - i), \tag{1.46}$$

i.e.: $(x-1)(x-2)\dots(x-20)$.

As for all products, when one of the factors is zero, then the whole product becomes zero. For the root problem, this means that, when we have the factors, we can find a value for x, so that any of the factors becomes zero and this value is then a root. Any integer in the range $[1 \dots 20]$ would make one of the factors of Wilkinson's polynomial zero. The integers $[1 \dots 20]$ are therefore the roots of this polynomial.

Factoring polynomials, however, is an advanced problem in its own right and we will dedicate some of the next sections to its study. Anyway, what algebraists did for centuries was searching formulas that would yield the roots for any kind of polynomials. In some cases they succeeded, in particular for polynomials of degrees less than 5. For higher degrees, there are no such formulas. This discovery is perhaps much more important than the single formulas developed over the centuries for polynomials of the first four degrees. In fact, the concepts that led to the discovery are the foundations of modern (and *postmodern*) algebra.

But first things first. To understand why there cannot be general formulas for solving polynomials of higher degrees, we need to understand polynomials much better. First, we will look at the formula to solve polynomials of the second degree.

Polynomials of the first degree are just linear equations of the form

$$ax + b = 0. \tag{1.47}$$

We can easily solve by subtracting b and dividing by a:

$$x = -\frac{b}{a}. \tag{1.48}$$

In Haskell, this is just:

```
solve1 :: (Fractional a) => Poly a -> [a]
solve1 (P [b, a]) = [-b / a]
```

Note the order of a and b in the definition of the polynomial. This is consistent with the equation we gave above, since, in our definition of polynomials in Haskell, the head of the list of the coefficients is the coefficient of x^0.

Polynomials of the second degree can be solved with a technique we already used in the previous chapter, namely *completing the square*. We will now apply this technique on symbols and, as a result, will obtain a formula that can be applied on any polynomial of second degree. We start with the equation

$$ax^2 + bx + c = 0. \tag{1.49}$$

We subtract c and divide by a obtaining:

$$x^2 + \frac{b}{a}x = -\frac{c}{a}. \tag{1.50}$$

Now, we want to get a binomial formula on the left-hand side of the equation. A binomial formula has the form:

$$(\alpha + \beta)^2 = \alpha^2 + 2\alpha\beta + \beta^2. \tag{1.51}$$

When we set $\alpha = x$, we have on the right-hand side:

$$x^2 + 2\beta x + \beta^2.$$

In our equation, we see the term $\frac{b}{a}x$ at the position where, here, we have $2\beta x$. We, therefore, have $\frac{b}{a} = 2\beta$ and $\beta = \frac{b}{2a}$. The missing term, hence, is $\left(\frac{b}{2a}\right)^2 = \frac{b^2}{4a^2}$. We add this term to both sides of the equation:

$$x^2 + \frac{b}{a}x + \frac{b^2}{4a^2} = -\frac{c}{a} + \frac{b^2}{4a^2}. \tag{1.52}$$

We can simplify the right-hand side of the equation a bit:

$$x^2 + \frac{b}{a}x + \frac{b^2}{4a^2} = \frac{b^2 - 4ac}{4a^2}. \tag{1.53}$$

To get rid of all the squares, we now take the square root on both sides of the equation. Since we have a binomial formula on the left-hand side, we get:

$$x + \frac{b}{2a} = \frac{\pm\sqrt{b^2 - 4ac}}{2a}. \tag{1.54}$$

When we solve this equation for x, we get

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}. \tag{1.55}$$

Voilà, this is the formula for solving polynomials of the second degree.

We immediately see that polynomials with rational coefficients may have irrational roots, because the solution involves a square root, which leads either to an integer or an irrational number.

We also see that polynomials of the second degree may have two roots, namely the result of the expression on the right-hand side, when we take the positive root, *i.e.*

$$\frac{-b + \sqrt{b^2 - 4ac}}{2a},$$

and the one, when we take the negative root, *i.e.*

$$\frac{-b - \sqrt{b^2 - 4ac}}{2a}.$$

However, when the square root is zero then it makes no difference whether we add or subtract. The square root becomes zero, when the expression $b^2 - 4ac$ is zero. So, when this expression is zero, there is only one root.

But there is one more thing: When the expression $b^2 - 4ac$ is negative, then we will try to take a square root from a negative term and that is not defined, since a number multiplied by itself is always positive, independent of that number itself being positive or negative.

Well, it is not defined for *real* numbers. When we assume that $\sqrt{-1}$ is actually a legal expression, we could *extend* the field of the real numbers to another, more complex field that includes this beast. We have already looked at how to extend fields in the previous chapter and we will indeed do this extension for $\mathbb{R}$ to create the *complex numbers*, $\mathbb{C}$. In that field, the root of a negative number is indeed defined and we have a valid result in both cases.

For instance the polynomial $x^2 + 1$ is never negative and, therefore, has no roots in $\mathbb{R}$. But when we assume that there is a number, say, i, for which $i^2 = -1$, then this value i would make the polynomial zero: $i^2 + 1 = -1 + 1 = 0$.

But, again, first things first. The expression $b^2 - 4ac$ is called the *discriminant* of the polynomial, because it determines how many roots there are: 2, 1 or (in $\mathbb{R}$) none. The discriminant for polynomials of degree 2 with real coefficients may be implemented in Haskell as follows:

$$
\begin{array}{l}
\mathit{dis2} :: (\mathit{Num}\ a) \Rightarrow \mathit{Poly}\ a \to a \\
\mathit{dis2}\ (P\ [c, b, a]) = b \uparrow 2 - 4 * a * c
\end{array}
$$

On top of this we implement a *root counter*:

$$
\begin{array}{lll}
\mathit{countRoots} :: (\mathit{Num}\ a, \mathit{Ord}\ a) \Rightarrow \mathit{Poly}\ a \to \mathit{Int} & & \\
\mathit{countRoots}\ p & \mid \mathit{dis2}\ p > 0 & = 2 \\
 & \mid \mathit{dis2}\ p < 0 & = 0 \\
 & \mid \mathit{otherwise} & = 1
\end{array}
$$

The polynomial $x^2 + 4$, for instance, has no roots in $\mathbb{R}$, since

$\mathit{countRoots}\ (P\ [4, 0, 1])$

gives 0. Indeed $0^2 - 4 \times 1 \times 4$ is negative.

The polynomial $x^2 - 4$, by contrast has

$\mathit{countRoots}\ (P\ [-4, 0, 1])$,

2 roots. Indeed, $0^2 - 4 \times 1 \times -4$ is $0 + 16$ and, hence, positive.

The polynomial x^2 has 1 root, since

$\mathit{countRoots}\ (P\ [0, 0, 1])$

is 1. Indeed, $0^2 - 4 \times 1 \times 0$ is 0.

We finally implement the solution for polynomials of the second degree:

$$
\begin{array}{lll}
\mathit{solve2} :: (\mathit{Floating}\ a, \mathit{Fractional}\ a, \mathit{Real}\ a) \Rightarrow \mathit{Poly}\ a \to [\,a\,] & & \\
\mathit{solve2}\ p@(P\ [c, b, a]) & \mid \mathit{dis2}\ p < 0 & = [\,] \\
 & \mid \mathit{x1} \not\equiv \mathit{x2} & = [\mathit{x1}, \mathit{x2}] \\
 & \mid \mathit{otherwise} & = [\mathit{x1}] \\
\quad \textbf{where}\ d & = \mathit{sqrt}\ (\mathit{dis2}\ p) & \\
\quad \quad \mathit{x1} & = (-b + d)\ /\ 2 * a & \\
\quad \quad \mathit{x2} & = (-b - d)\ /\ 2 * a &
\end{array}
$$

When we call $\mathit{solve2}\ (P\ [0, 0, 1])$, that is, we solve the polynomial x^2, we get the root $[0]$, which is one root as predicted.

To solve the polynomial $x^2 + 4$, we call $\mathit{solve2}\ (P\ [4, 0, 1])$ and get $[\,]$; as predicted, this polynomial has no roots. It is everywhere positive.

The polynomial $x^2 - 4$, by contrast, shall have two roots. We call $\mathit{solve2}\ (P\ [-4, 0, 1])$ and get $[2, -2]$. When we check this by applying the polynomial to 2 and -2 like $\mathit{map}\ (\mathit{apply}\ (P\ [-4, 0, 1]))\ [2, -2]$, we get $[0, 0]$.

What about the polynomial $-x^2 - x + 1$, which we factored in the previous chapter? We try $\mathit{solve2}\ (P\ [1, -1, -1])$ and get

$[-1.618033988749895, 0.6180339887498949]$,

which is $-\Phi$ and $-\Psi$, just as we saw before.

Which polynomial has the roots Φ and Ψ? Well, let us try:

$mul\ (P\ [-phi, 1])\ (P\ [-psi, 1])$

yields:

$P\ [1.0, -2.23606797749979, 1.0]$,

which corresponds to $x^2 - \sqrt{5} + 1$. The coefficients are 1 for x^2, $-\sqrt{5}$ for $(-\Phi - \Psi)x$ and 1 for $(-\Phi)(-\Psi)$.

What is the result for the "simple" polynomial x^2+x+1? We try with *solve2* $(P\ [1, 1, 1])$ and get $[\,]$ – the empty list. Indeed, $1^2 - 4 \times 1 \times 1$ is negative!

Let us pretend to be optimistic like the "reckoning masters" in the 15 and 16 hundreds. We already have a formula to compute the roots for polynomials of the first two degrees. It will certainly be easy to find formulas for the remaining (infinitely many) degrees. We can then define a function of the form:

$$
\begin{aligned}
&solve :: (Fractional\ a, Floating\ a, Real\ a) \Rightarrow Poly\ a \rightarrow [\,a\,]\\
&solve\ p \mid degree\ p \equiv 0 = [\,]\\
&\qquad\quad \mid degree\ p \equiv 1 = solve1\ p\\
&\qquad\quad \mid degree\ p \equiv 2 = solve2\ p\\
&\qquad\quad \mid degree\ p \equiv 3 = \perp\\
&\qquad\quad \mid degree\ p \equiv 4 = \perp\\
&\qquad\quad \mid degree\ p \equiv 5 = \perp
\end{aligned}
$$

and so on. With this optimism, our goal is to replace the $\perp$ implementations by functions of the form *solve3*, *solve4*, *etc.* We come back to this endevour in a later chapter.

1.7 Vieta's Formulas

The binomial theorem describes regularities in the coefficients that turn up when multiplying a polynomial (repeatedly) by it itself. For the simple case $(a + b)(a + b)$, we get the result $(a^2 + 2ab + b^2)$. The linear factors of polynomials have a similar structure: sums of numbers that are multiplied with each other, *e.g.*:

$$x^2 - 1 = (x + 1)(x - 1). \tag{1.56}$$

Should we not expect similar regularities with the coefficients of the resulting polynomials in those cases? When we look at this in an algebraic way, we would see:

$$(x + a)(x + b) = x^2 + xb + xa + ab = x^2 + (a + b)x + ab. \tag{1.57}$$

The coefficients of the resulting polynomial are 1, $a + b$ and ab. We immediately see the relation to the binomial theorem: if $a = b$, we would have $2a$ and a^2, where, in the binomial theorem, the final coefficent is interpreted as 1, for the number of occurrences of a^2. We, hence, get $1, 2, 1$.

Let us check the theoretic result against the concrete example $(x + 1)(x - 1)$. We set $a = 1$ and $b = -1$ and see:

$$x^2 + (1 - 1)x + (1 \times (-1)) = x^2 - 1. \tag{1.58}$$

That appears to be correct. But who are those a and b guys that appear in the formula? Well, those are the additive inverses of the roots of the polynomial in question, since, if $(x + a)(x + b)\dots$ are the linear factors, then the polynomial becomes 0 if any of those factors becomes 0. The factor $(x+a)$, obviously, becomes 0 if $x = -a$. $-a$ is therefore a root of the polynomial. It follows that we have a direct relation between the roots and the coefficients.

As a first approximation (which is wrong!), we could describe a second degree polynomial with the roots α and β as:

$$x^2 + (-\alpha - \beta)x + \alpha\beta,$$

We check again with $\alpha = -1$ and $\beta = 1$:

$$x^2 + (1 - 1)x + (-1 \times 1) = x^2 - 1. \tag{1.59}$$

Correct until here. Let us express this result as a formula that defines the coefficients in terms of roots. We have $-\alpha - \beta = b$ where b is the second coefficient in a polynomial of the form $x^2 + bx + c$. We can factor "$-$" out and get $-(\alpha + \beta) = b$ or nicer even (but still wrong!):

$$\alpha + \beta = -b. \tag{1.60}$$

Correspondly, we have for c:

$$\alpha\beta = c. \tag{1.61}$$

What about other examples, for instance: $x^2 + x - 1$. We already know the roots are $-\Phi$ and $-\Psi$. So, we set $\alpha = -\Phi$ and $\beta = -\Psi$:

$$x^2 + (\Phi + \Psi)x + ((-\Phi) \times (-\Psi)) = x^2 + x - 1. \tag{1.62}$$

The polynomial $x^2 - 4$ has the roots 2 and -2:

$$x^2 + (-2 + 2)x + (2 \times (-2)) = x^2 - 4. \tag{1.63}$$

The polynomial $x^2 + 5 + 6$ has the roots -2 and -3:

$$x^2 + (2 + 3)x + (-2 \times (-3)) = x^2 + 5x + 6. \tag{1.64}$$

Note, by the way, the multiplication $12 \times 13 = 156$. Once again, this is a nice illustration of the similarity of numbers and polynomials.

Now, what about the polynomial $-x^2 - x + 1$. We know it has the same roots as the polynomial $x^2 + x - 1$. But how can we get the coefficients from the roots with the same formula? Something seems to be wrong...

Well, until now, we have looked only at *monic* polynomials, that is polynomials with the first coefficient being 1. But the polynomial $-x^2 - x + 1$ is not monic. The first coefficient is -1. In fact, the complete factorisation of this polynomial is

$$-1(x + \Phi)(x + \Psi).$$

We have to adjust our formula above to this case – and that is where we said the formulas given above are wrong. The adjustment, however, is quite easy. We just divide all coefficients by the leading one and get:

$$\alpha + \beta = -\frac{b}{a} \tag{1.65}$$

and

$$\alpha\beta = \frac{c}{a}. \tag{1.66}$$

Then, in the polynomial formula, we need to multiply a to get the coefficients back, *e.g.*:

$$ax^2 - a(\alpha + \beta)x + a\alpha\beta,$$

When we now test with roots $\alpha = -\Phi$ and $\beta = -\Psi$ and coefficient $a = -1$, we get

$$\begin{array}{rl} & -x^2 - (-1)(-\Phi - \Psi)x - 1(-\Phi)(-\Psi) \\ = & -x^2 - (-1)(-1)x + (-1)(-1) \\ = & -x^2 - x + 1 \end{array}$$

and everything seems to be in joint again.

When we advance beyond degree 2, how should these formulas evolve? Let us look at roots in terms of linear factors. For a polynomial of degree n, we have up to n factors of the form

$$(x+\alpha)(x+\beta)(x+\gamma)\dots$$

When we multiply that out, we get combinations as products and sums of products of the coefficients of the linear factors $\alpha, \beta, \gamma, \dots$ which are the additive inverses of the roots of the resulting polynomial (so watch out for signs in the following formulas!):

$$(x^2 + \beta x + \alpha x + \alpha\beta)(x+\gamma),$$

which is

$$x^3 + (\alpha+\beta+\gamma)x^2 + (\alpha\beta + \alpha\gamma + \beta\gamma)x + \alpha\beta\gamma.$$

This already begins to reveal a pattern. The first coefficient of the resulting polynomial (counting without the coefficient of x^3) is the sum of all the linear coefficients; the second coefficient is the sum of all their tuple products; the third is a triple product. We could suspect that the third, in a four-degree polynomial, would be the sum of all triple products and the fourth a single quadruple product. Let us check: we compute

$$(x^3 + (\alpha+\beta+\gamma)x^2 + (\alpha\beta + \alpha\gamma + \beta\gamma)x + \alpha\beta\gamma)(x+\delta)$$

and get

$$\begin{array}{lll} & & x^4 \\ + & (\alpha+\beta+\gamma+\delta) & x^3 \\ + & (\alpha\beta + \alpha\gamma + \alpha\delta + \beta\gamma + \beta\delta + \gamma\delta) & x^2 \\ + & (\alpha\beta\gamma + \alpha\beta\delta + \alpha\gamma\delta + \beta\gamma\delta) & x \\ + & \alpha\beta\gamma\delta. & \end{array}$$

The result, indeed, continues the pattern we saw above. For the first coefficient we see the simple sum of all the linear coefficients; for the second one, we see the sum of all tuple products; for the third one, we see the sum of all triple products and then we see a single quadruple product.

When we now bring the negative sign of the roots in (we used their additive inverses) and the first coefficient, we get the following sequence of formulas:

$$x_1 + x_2 + \cdots + x_n = -\frac{a_{n-1}}{a_n} \tag{1.67a}$$

$$x_1x_2 + \cdots + x_1x_n + x_2x_3 + \cdots + x_2x_n + \cdots + x_{n-1}x_n = \frac{a_{n-2}}{a_n} \tag{1.67b}$$

$$x_1x_2x_3 + \cdots + x_1x_2x_n + \cdots + x_{n-2}x_{n-1}x_n = -\frac{a_{n-3}}{a_n} \tag{1.67c}$$

$$\ldots = \ldots \tag{1.67d}$$

$$(x_1x_2 \ldots x_n) = (-1)^n \frac{a_0}{a_n} \tag{1.67e}$$

to describe the relation of roots and coefficients of a polynomial of the form

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

with roots $x_1, x_2, \ldots, x_n$.

The equations 1.67 are known as *Vieta's formulas*, after the French lawyer and mathematician François Viète (1540 – 1603) who we already know as author of an elegant formula to express π.

But what are those constructs on the left-hand side of the formulas? One answer is: those are *elementary symmetric polynomials*, which are building blocks for *symmetric polynomials*. Symmetric polynomials will be very important for us further down the road. At the moment, they only delay a good answer to the question...

A better answer at this stage is that those beasts are the sums of all *distinct* combinations of the roots in 1-tuples, 2-tuples, 3-tuples and so on. For the first case, the '1-tuples', that is just the sum of all the roots; for the second case, the '2-tuples', we have all combinations of *2 elements out of* n, where n is the number of roots; for the third case, we have all combinations of *3 elements out of* n and so on.

You probably guess where this is leading us. When we have four roots, the first coefficient, the one in front of x^{n-1}, is basically the sum of $\binom{4}{1} = 4$ terms; the second coefficient is the sum of $\binom{4}{2} = 6$ terms; the third coefficient is the sum of $\binom{4}{3} = 4$ terms and the last coefficient, the one without an x, is the sum of only $\binom{4}{4} = 1$ term.

In general, for n roots, we get, for the k^{th} coefficient, $\binom{n}{k}$ terms of products of k roots. Those are $\sum_{k=0}^{n} \binom{n}{k} = 2^n$ terms in total (including the coefficient in front of x^n, which corresponds to $\binom{n}{0} = 1$). Once again, algebra boils down to combinatorial problems induced by the distributive law.

Let us devise a function that gives us the right-hand sides of Vieta's formula, when we provide the left-hand sides. That is, we write a function that receives the list of roots of the polynomial and that returns the list of the coefficients divided by the first coefficient.

On the first sight, it seems to be tricky to get the sums of products right. But, in fact, we already know everything we need. What we want to do is to generate all possible k-combinations for $k = 1 \dots n$ of the n elements, but without duplicates, *i.e.* ab is the same as ba (since multiplication is commutative). This, however, is the structure of the powerset, which, for a set with n elements, contains indeed 2^n subsets – just the number of all possibilities to choose k out of n for $k = 0 \dots n$.

For instance, the set of roots $\{\alpha, \beta, \gamma, \delta\}$ has the powerset (ordered according to the size of the subsets):

$$\begin{gathered}\{\varnothing, \\ \{\alpha\}, \{\beta\}, \{\gamma\}, \{\delta\}, \\ \{\alpha,\beta\}, \{\alpha,\gamma\}, \{\alpha,\delta\}, \{\beta,\gamma\}, \{\beta,\delta\}, \{\delta,\gamma\}, \\ \{\alpha,\beta,\gamma\}, \{\alpha,\beta,\delta\}, \{\alpha,\gamma,\delta\}, \{\beta,\gamma,\delta\}, \\ \{\alpha,\beta,\gamma,\delta\}\}\end{gathered}$$

We can transform the powerset into the coefficients by dropping $\varnothing$ (which represents a in a monic polynomial) and then adding up the products of the subsets of the same size. The following function does that:

```
vieta :: (Real a) => [a] -> [a]
vieta = c ∘ g ∘ d ∘ s ∘ Perm.ps
  where d   = drop 1
        g   = groupBy ((≡) `on` length)
        s   = sortOn length
        c p = [(-1) ↑ n * sum (map product x) | (x, n) <- zip p [1 ..]]
```

We first create the powerset (*Perm.ps*). We then sort it by the lengths of the subsets (that is the *cardinalities* in set theory jargon) and drop the first one (the empty set). We then introduce one more level of separation, *i.e.* we group the subsets by their size. From this result, we create a new set by zipping the result with the natural numbers starting from 1 so that each group of equal length gets paired with a number n. We, then, map *product* on these lists and add the resulting products together. Finally, we multiply this number by -1 raised to the power of n.

This last step takes care of signedness. Since, in the linear factors, we use the additive inverses of the roots, the effect of the signs of the roots must be flipped around. Therefore, we flip the sign of every second result, namely those with an odd number of factors. The negative signs of the roots that enter products with an even number of factors cancel out by themselves.

Let us look at some examples. We start with our favourite: $x^2 + x - 1$. and call *vieta* $[-phi, -psi]$:

$[1.0, -1.0]$.

That are the coefficient of x and the constant -1. To complicate, we check some variants of those roots:

- *vieta* $[phi, -psi]$ gives $[-\sqrt{5}, 1]$ and, hence, the polynomial $x^2 - \sqrt{5} + 1$, whose roots are indeed Φ and $-\Psi$.
- *vieta* $[-phi, psi]$ gives $[\sqrt{5}, 1]$ and that is the polynomial $x^2 + \sqrt{5} + 1$, whose roots are $-\Phi$ and Ψ.
- *vieta* $[phi, psi]$ gives $[-1, -1]$, the polynomial $x^2 - x - 1$, whose roots are Φ and Ψ.

A simpler example that shows the signedness of roots and coefficients is x^2-1. *vieta* $[1, -1]$ gives $[0, -1]$, which, indeed, corresponds to $x^2 - 1$.

What about a third-degree polynomial, *e.g.* $(x+1)(x+1)(x+1) = x^3 + 3x^2 + 3x + 1$? We call *vieta* $[-1, -1, -1]$ and see $[3, 3, 1]$.

Another experiment: we compute *mul* $(P\ [1, 1])$ $(mul\ (P\ [2, 1])\ (P\ [3, 1]))$, which is $(x+1)(x+2)(x+3)$ and get $P\ [6, 11, 6, 1]$, which represents the polynomial $x^3+6x^2+11x+6$. We call *vieta* $[-1, -2, -3]$ and get $[6, 11, 6]$. (You may realise that the coefficients are unsigned Stirling numbers of the first kind and now might want to contemplate why those guys show up again...)

A fifth-degree polynomial: *prodp mul* $[P\ [1, 1], P\ [2, 1], P\ [3, 1], P\ [4, 1], P\ [5, 1]]$: $P\ [120, 274, 225, 85, 15, 1]$, that is

$$x^5 + 15x^4 + 85x^3 + 225x^2 + 274x + 120.$$

vieta $[-1, -2, -3, -4, -5]$: $[15, 85, 225, 274, 120]$.

Well, we can go on playing around like this forever. The point of Vieta's formulas, however, is not so much practical. It is not an efficient way to compute roots from coefficients or coefficients from roots. That should be clear immediately, when we look at the Haskell function *vieta*. It generates the powerset of the set of roots – and that cannot be efficient at least for large (or better worded perhaps: unsmall) numbers. Vieta's formulas, instead, are a theoretical device. They help us understand the relation between coefficients and roots and they will play an important role in our further investigations.

1.8 Discriminant and Resultant

When we started to discuss roots, we mentioned the discriminant for polynomials of degree 2,

$$b^2 - 4ac,$$

which tells us, just looking at the coefficients, how many real roots the polynomial has, namely two, if the discriminant is positive, one repeated root, if it is 0, and none at all, if it is negative.

Wouldn't it be nice to have a discriminant for any degree? It turns out, there is. At its very heart, this generalised discriminant is the product of the differences of the roots; for instance, if we have the three roots α, β, γ, the core of the discriminant is

$$(\alpha - \beta)(\alpha - \gamma)(\beta - \gamma).$$

In general we have

$$\prod_{i<j} (x_i - x_j). \tag{1.68}$$

Notice that there are $\binom{n}{2}$ factors for n the number of roots. For two roots, there is only one factor, namely $\alpha - \beta$; for three roots, there are three factors, namely those given above; for four roots, there are six factors and so on.

Now, when there is a repeated root, then one of the factors will be 0 and, as such, the whole product will be 0. If at least one of the roots is non-real, then the product will be non-real too. We, hence, can read off the same properties from this discriminant as we already could read from the old one.

But the new discriminant tells us even more: if it is irrational, then all roots are real, but at least one is irrational. The other way round, if it is rational, then all roots are rational.

There is a snag though. In the generalised formula, we assumed a given order of the roots – but there is no such privileged order. Imagine a polynomial with the roots $-1, 2, -3, 4$, which is

$$(x+1)(x-2)(x+3)(x-4) = x^4 - 2x^3 - 13x^2 + 14x + 24.$$

When we compute the differences of the roots in this order (and note that there are $\binom{4}{2} = 6$ factors), *i.e.*

$$(-1-2)(-1+3)(-1-4)(2+3)(2-4)(-3-4),$$

we get the result 2100. But when we compute, just changing the order of the roots to $2, -1, -3, 4$,

$$(2+1)(2+3)(2-4)(-1+3)(-1-4)(-3-4)$$

we get -2100. Indeed, of the $4! = 24$ permutations of the roots, 12 lead to the positive result and 12 to the negative one. That means: the discriminant in this form, is not well defined!

We can get rid of the problem by squaring the differences like this:

$$\prod_{i<j}(x_i - x_j)^2. \tag{1.69}$$

We now would have

$$(-1-2)^2(-1+3)^2(-1-4)^2(2+3)^2(2-4)^2(-3-4)^2,$$

which is 4410000, and

$$(2+1)^2(2+3)^2(2-4)^2(-1+3)^2(-1-4)^2(-3-4)^2,$$

which is 4410000 as well.

Of course, we have to adapt our principles to this new form: there are non-real roots, if the discriminant is negative and there are irrational roots if the discriminant is not a perfect square.

You now may ask: is this discriminant actually the same as the naïve one? Or, in other words, is the naïve discriminant a special case of this general form for second-degree polynomials?

Let us look at a second-degree polynomial, *e.g.*:

$$(x-2)(x+3) = x^2 + x - 6.$$

The naïve discriminant ($b^2 - 4ac$) is

$$1^2 - 4 \times (-6) = 1 + 24 = 25.$$

The new discriminant is

$$(2 + 3)^2 = (-3 - 2)^2 = 25.$$

This seems to be correct. Let's try a non-monic polynomial, *e.g.*

$$3(x - 2)(x + 3) = 3x^2 + 3x - 18.$$

The naïve discriminant is

$$3^2 - 4 \times 3 \times (-18) = 225.$$

The roots are, of course, still 2 and -3. So the new discriminant is still 25. What are we missing?

Let's be practical and compare the two numbers. What is their ratio? It is $225/25 = 9$. 9, however, is the leading coefficient squared: 3^2. So, if we multiplied the discriminant by the square of the leading coefficient, the results would be equal again.

Is this just by chance or can we prove that it is always the case? We need to prove, for the case of a second-degree polynomial, that

$$a^2(\alpha - \beta)^2 = b^2 - 4ac, \tag{1.70}$$

where a, b, c are the coefficients and α, β are, as usual, the roots of the polynomial.

First we observe that $(\alpha - \beta)^2$ can be expressed as $(\alpha + \beta)^2 - 4\alpha\beta$. This is true because, when we multiply $(\alpha + \beta)^2$ out, we get

$$\alpha^2 + 2\alpha\beta + \beta^2.$$

When we subtract $4\alpha\beta$, we obtain

$$\alpha^2 - 2\alpha\beta + \beta^2,$$

which clearly is $(\alpha - \beta)^2$. Thus:

$$(\alpha - \beta)^2 = (\alpha + \beta)^2 - 4\alpha\beta. \tag{1.71}$$

According to Vieta's formulas, which we discussed in the previous section, we have:

$$\alpha + \beta = -\frac{b}{a} \tag{1.72}$$

and

$$\alpha\beta = \frac{c}{a}. \tag{1.73}$$

We therefore have

$$(\alpha - \beta)^2 = \left(-\frac{b}{a}\right)^2 - 4\frac{c}{a}. \tag{1.74}$$

We multiply a^2 on both sides and get

$$a^2(\alpha - \beta)^2 = a^2\left(\left(-\frac{b}{a}\right)^2 - 4\frac{c}{a}\right). \tag{1.75}$$

The right-hand side is

$$a^2\left(\frac{b^2}{a^2} - 4\frac{c}{a}\right).$$

Distributing a^2 over the terms, we get

$$\frac{a^2b^2}{a^2} - 4\frac{a^2c}{a}.$$

and can now simplify to

$$b^2 - 4ac,$$

which leads to the desired result

$$a^2(\alpha - \beta)^2 = b^2 - 4ac. \quad \square \tag{1.76}$$

In the general form, which we won't prove here, the discriminant can be computed as

$$a^{2d-2}\prod_{i<j}(x_i - x_j)^2, \tag{1.77}$$

where a is the leading coefficient and d the degree of the polynomial. For $d = 2$, the funny exponent $2d - 2$ is $2 \times 2 - 2 = 2$. For $d = 3$, it would be $6 - 2 = 4$, for $d = 4$, we have $8 - 2 = 6$ and so on.

Of course, we again have to adapt our principles to this new formula. To say something about irrationality of the roots, we need to divide the discriminant by a^{2d-2}. If (and only if) the result is a perfect square, the polynomial has only rational roots. Note that we do not need to change anything to decide whether there are non-real roots. Since $2d-2$ is always even, a raised to such a power is always positive. It will, hence, not affect the sign of the discriminant. Therefore, if (and only if) the discriminant is negative, there are non-real roots.

But isn't there a real issue? The discriminant is supposed to tell us something about the roots. But from what we see here, we need to know the roots to compute the discriminant in the first place. That is not very useful! The so called "naïve" discriminant is not too naïve after all! At least, it has a function!

Well, here comes the esoteric part of this section. There is in fact a way to compute the discriminant without knowing the roots. The discriminant can be computed from the *resultant* of the polynomial and its derivative. The resultant is a magic number computed from two polynomials and is often used to decide whether two polynomials have a common root.

There are two ways to compute the resultant. One comes from linear algebra and is very inefficient, the other is related to polynomial arithmetic and pretty efficient. We will start with the first one. It is inefficient, but the algorithm is quite interesting and it introduces some concepts from linear algebra. This way, the current section is also a teaser for one of the next chapters to come.

The resultant of two polynomials can be computed as the *determinant* of the *Sylvester matrix* of these polynomials. Wow! That are two new concepts in one sentence! We look at them one by one. First, the *Sylvester matrix*.

We already met the concept of a matrix in the previous part: a matrix is just a table of numbers. The Sylvester matrix is a square matrix (one with an equal number of rows and columns) that contains the coefficients of the two polynomials. It has $d_1 + d_2$ rows and columns, where d_1 and d_2 are the degrees of the polynomials.

It is constructed in the following way:

1. The first row of the matrix consists of the coefficients of the first polynomial and $d_2 - 1$ zeros on the right.

2. The next row contains one zero on the left followed by the coefficients of the first polynomial and $d_2 - 2$ zeros on the right.

3. We continue this way, incrementing the number of zeros on the left and decrementing it on the right until the number of zeros on the right is zero, that is, the last

coefficient of the polynomial (the one of lowest degree) hits the end of the row.

4. Then we repeat the process with the second polynomial starting in the next row.

Imagine the two polynomials $P\ [4,3,2,1]$ and $P\ [7,6,5]$ (in mathematical notation: $x^3 + 2x^2 + 3x + 4$ and $5x^2 + 6x + 7$) which are of degree 3 and 2 respectively. The Sylvester matrix, hence, is the 5×5 square matrix:

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 0 \\ 0 & 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 0 & 0 \\ 0 & 5 & 6 & 7 & 0 \\ 0 & 0 & 5 & 6 & 7 \end{pmatrix}$$

Here is an Haskell implementation using the matrix type from the previous part:

```
sylvester :: (Num a) => Poly a -> Poly a -> L.Matrix a
sylvester a b = L.M (go 0 xs ys)
  where la = degree a
        lb = degree b
        ll = la + lb
        xs = (reverse $ coeffs a) ++ zeros (lb - 1)
        ys = (reverse $ coeffs b) ++ zeros (la - 1)
        go _ [] []                  = []
        go i l1 l2 | i == ll        = []
                   | i >= lb        = l2 : go (i + 1) [] (0 : init l2)
                   | otherwise      = l1 : go (i + 1) (0 : init l1) l2
```

When we call *sylvester* like this:

$sylvester\ (P\ [4,3,2,1])\ (P\ [7,6,5])$

we get

$M\ [[1,2,3,4,0],[0,1,2,3,4],[5,6,7,0,0],[0,5,6,7,0],[0,0,5,6,7]]$

which corresponds to the matrix above.

The other concept mentioned above is the determinant of a matrix. The determinant is defined only for square matrices and can be seen as an encoding of certain *linear transformations* described by the matrix. Those are all concepts from linear algebra and will remain somewhat mysterious in this section. However, we just want to compute the determinant and that is an interesting recursive algorithm.

The determinant for a 2×2 square matrix m of the form

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

also written $|m|$, is defined as

$$det(m) = |m| = ad - bc. \tag{1.78}$$

It is, thus, a kind of "cross product difference", *i.e.* the difference of the products resulting from multiplying the elements in the square matrix that share neither row nor column. In terms of (row,column)-coordinates we have $m[0,0] \times m[1,1] - m[0,1] \times m[1,0]$.

When we have an $n \times n$ square matrix with $n > 2$, we proceed as follows:

1. We cut off the first row.
2. For each element x_i in that row (the subscript i representing the column within that row), we compute $(-1)^i x_i |minor_i(m)|$ where $minor_i$ is a square matrix formed from the original matrix without the first row and with the ith column removed.
3. We sum up the results.

The determinant of the 3×3 matrix m

$$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix}$$

would be computed as

$$|m| = a \begin{vmatrix} e & f \\ h & i \end{vmatrix} - b \begin{vmatrix} d & f \\ g & i \end{vmatrix} + c \begin{vmatrix} d & e \\ g & h \end{vmatrix} \tag{1.79}$$

For $n \times n$ matrices with $n > 3$, the process repeats recursively on each sub-determinant. The process is called *minor expansion formula* (MEF), since it expands into always more minor matrices as it proceeds further.

To clarify how the MEF proceeds, here is a Haskell implementation. We start with a simple function to copy the columns of a matrix cutting one column out:

```
copyWithout :: Int -> [a] -> [a]
copyWithout p rs = go 0 rs
  where go _ []                  = []
        go i (c : cs) | i == p    = go (i + 1) cs
                      | otherwise = c : go (i + 1) cs
```

which we use to create the rows and columns of a minor matrix

$$
\begin{aligned}
&\mathit{minor} :: \mathit{Int} \to [[\,a\,]] \to [[\,a\,]]\\
&\mathit{minor}\ p\ [\,] \qquad\ \ = [\,]\\
&\mathit{minor}\ p\ (r : rs) = \mathit{copyWithout}\ p\ r : \mathit{minor}\ p\ rs
\end{aligned}
$$

Here is the MEF:

$$
\begin{aligned}
&\mathit{mef} :: (\mathit{Num}\ a) \Rightarrow [[\,a\,]] \to a\\
&\mathit{mef}\ [\,] \qquad\qquad\quad\ = 0\\
&\mathit{mef}\ [[\,a\,]] \qquad\qquad = a\\
&\mathit{mef}\ [[\,a, b\,], [\,c, d\,]] = a * d - b * c\\
&\mathit{mef}\ (r : rs) \qquad\ \ = \mathit{sum}\ [(-1) \uparrow i * c * (\mathit{go}\ i\ rs) \mid (c, i)\\
&\qquad\qquad\qquad\qquad\qquad\qquad \leftarrow \mathit{zip}\ r\ [0\,..\,(\mathit{length}\ r) - 1]]\\
&\quad \textbf{where}\ \mathit{go}\ i = \mathit{mef} \circ \mathit{minor}\ i
\end{aligned}
$$

Notice that we already cut out the first row in *mef* and, this way, call *minor* with an $(n-1) \times n$ matrix, which, in its turn, creates an $(n-1) \times (n-1)$ matrix. The recursion enters in the *go* function, which is called for each element in the sum.

Finally, we add a top-level function to compute the determinant:

$$
\begin{aligned}
&\mathit{det} :: (\mathit{Num}\ a) \Rightarrow \mathit{Matrix}\ a \to a\\
&\mathit{det}\ m \mid \neg\ (\mathit{square}\ m) = \mathit{error}\ \texttt{"not a square matrix"}\\
&\qquad\ \ \mid \mathit{otherwise} \qquad = \mathit{mef}\ (\mathit{rows}\ m)
\end{aligned}
$$

It is one of the many amazing results of linear algebra that the determinant of the Sylvester matrix of two polynomials is the resultant of these polynomials. Furthermore, the discriminant of a polynomial can be computed from the resultant of this polynomial and its derivative. Concretely, it can be computed as

$$
dis(p) = (-1)^{d(d-1)/2} \times \frac{res(p, p')}{lc(p)}, \tag{1.80}
$$

where d is the degree of the polynomial and lc is the leading coefficient. Here is an implementation:

$$
\begin{aligned}
&\mathit{dis} :: (\mathit{Num}\ a, \mathit{Integral}\ a) \Rightarrow \mathit{Poly}\ a \to a\\
&\mathit{dis}\ p = (-1) \uparrow x * (\mathit{res}\ p\ p')\ \lq \mathit{div} \lq\ l\\
&\quad \textbf{where}\ x\ = d * (d - 1)\ \lq \mathit{div} \lq\ 2\\
&\qquad\qquad\ d\ = \mathit{degree}\ p\\
&\qquad\qquad\ p' = \mathit{derivative}\ (*)\ p\\
&\qquad\qquad\ l\ \ = \mathit{lc}\ p
\end{aligned}
$$

Notice that we have not yet defined *res*. We assume for the moment that *res* is the determinant of the Sylvester Matrix. We will implement it differently, however, for reasons that we will understand in a minute.

But let us first test our implementation. We start with our polynomial with the four roots $-1, 2, -3, 4$, which is $x^4 - 2x^3 - 13x^2 + 14x + 24$. Here is a Haskell session:

let $p = P\ [24, 14, -13, -2, 1]$
let $p' = \mathit{derivative}\ (*)\ p$
let $m = \mathit{sylvester}\ p\ p'$
let $d = \mathit{det}\ m$

What is d right now? Note that the degree of p is 4. The exponent of -1 in the computation of *dis* above, hence, is $4 \times 3/2$, which is 6 and even. The minus sign, therefore, disappears; furthermore, the leading coefficient is 1. The computation of the discriminant, this way, reduces to the determinant: d, therefore, shall be $4\,410\,000$, the discriminant of this polynomial. Try it out!

Let us look at the second-degree polynomial we already used above, namely $(x-2)(x+3) = x^2 + x - 6$.

let $p = P\ [-6, 1, 1]$
let $p' = \mathit{derivative}\ (*)\ p$
let $m = \mathit{sylvester}\ p\ p'$
let $d = \mathit{det}\ m$

The determinant d now is -25. When we compute the discriminant, the exponent will be $2 \times 1/2$, which is 1 and, as such, odd. We will hence multiply the determinant by -1 and arrive at the discriminant 25, which we already computed above.

For the polynomial $3(x-2)(x+3) = 3x^2 + 3x - 18$, we get, following the same recipe, -675. The minus sign will disappear and, when we divide the result by the leading coefficient, *i.e.* 675/3, we get 225 as expected.

Well, we computed the determinant with a recursive formula that expands a minor for each column per recursion step. For an $n \times n$ matrix with $n > 2$, we will compute n minors in the first recursion step. In the next recursion step, if $n-1 > 2$, we will for each of those minors compute $n-1$ minors and so on. How many minors do we compute?

We compute n for the first iteration, $n-1$ for the second, $n-2$ for the third and so on until $n = 2$. We, hence, have $n!/2$ recursion steps for an $n \times n$ square matrix. For a degree-5 polynomial, since its derivative has degree 4, we have a 9×9 matrix and need $9!/2 = 181440$ recursion steps. For a degree-6 polynomial, we already have an 11×11 matrix and, thus, need $11!/2 = 19958400$ recursion steps. That is obviously inefficient already for polynomials of moderate degree. The algorithm is cute, but does not scale very well.

So, let us look at the alternative related to polynomial arithmetic. It is based on the concept of *pseudo remainders*. This concept is motivated by the fact that, for polynomials over a ring (like the integers), there is no Euclidean division, *i.e.* a division resulting in a quotient and a remainder. In fact, when we divide polynomials, the coefficients of the polynomials do not lie in the original ring anymore. They are now fractional and, as such, belong to a field ($\mathbb{Q}$ for the ring of integers) and that is quite annoying since the GCD does not make much sense in a field.

To deal with this, mathematicians invented pseudo remainders. The pseudo remainder of two polynomials p and q is defined as

$$prem(p, q) = rem(lc(q)^{deg(p)-deg(q)+1}p, q). \tag{1.81}$$

In plain English: we scale p by the leading coefficient of q raised to the power of the difference of the degrees of p and q plus one. When you go back to the division formula, it should be intuitively clear that this guarantees that the coefficients of the remainder are all integral and, thus, still in the ring over which the polynomials were defined.

We can define *prem* in Haskell like this:

```haskell
prem :: (Integral a) => Poly a -> Poly a -> Poly a
prem a b = poly $ map numerator (coeffs $ snd (divp x y))
  where l  = lc b
        k  = l ^ (da - db + 1)
        da = fromIntegral $ degree a
        db = fromIntegral $ degree b
        a' = scale k a
        x  = P (map (%1) $ coeffs a')
        y  = P (map (%1) $ coeffs b)
```

This appears to be a lot of code for such a simple formula. Most of the code, however, is related to convert the coefficients back and forth between fractional (for which *divp* is defined) and integral.

Of course, these remainders are now too big, since the first polynomial has been scaled. Therefore mathematicians invented yet another trick, namely to reduce the resulting remainder by dividing by some number α.

To compute the GCD, one replaces the equation $r_{i+1} = rem(r_{i-1}, r_i)$ by

$$r_{i+1} = \frac{prem(r_{i-1}, r_i)}{\alpha} \tag{1.82}$$

Calling this recursively and preserving the intermediate results leads to a *pseudo-remainder sequence*:

$$
\begin{array}{l}
\mathit{pgcd} :: (\mathit{Integral}\ a) \Rightarrow \\
\qquad (\mathit{Poly}\ a \to a) \to \mathit{Poly}\ a \to \mathit{Poly}\ a \to [\mathit{Poly}\ a] \\
\mathit{pgcd}\ \mathit{alpha}\ p\ q \mid \mathit{zerop}\ q = [p] \\
\qquad\qquad\quad \mid \mathit{otherwise} = \\
\qquad\qquad\quad\ \ \mathbf{let}\ r = \mathit{prem}\ p\ q \\
\qquad\qquad\qquad\quad a = \mathit{alpha}\ r \\
\qquad\qquad\qquad\quad x = P\ [\mathit{numerator}\ (c\,\%\,a) \mid c \leftarrow \mathit{coeffs}\ r] \\
\qquad\qquad\quad\ \ \mathbf{in\ if}\ \mathit{zerop}\ r\ \mathbf{then}\ [\,]\ \mathbf{else}\ x : \mathit{pgcd}\ \mathit{alpha}\ q\ x
\end{array}
$$

The function *pgcd* accepts, besides two polynomials, a function that converts a polynomial into an integral. On each recursion step, it produces the pseudo remainder r and an α for the remainder and then divides the coefficients of the remainder by that number.

The trivial pseudo-remainder sequence, for instance, results from setting $\alpha = 1$:

$$
\begin{array}{l}
\mathit{tpgcd} :: (\mathit{Integral}\ a) \Rightarrow \mathit{Poly}\ a \to \mathit{Poly}\ a \to [\mathit{Poly}\ a] \\
\mathit{tpgcd} = \mathit{pgcd}\ (\backslash\_ \to 1)
\end{array}
$$

The primitive pseudo-remainder sequence, by contrast, uses the content of the polynomial resulting in a sequence of primitive polynomials:

$$
\begin{array}{l}
\mathit{ppgcd} :: (\mathit{Integral}\ a) \Rightarrow \mathit{Poly}\ a \to \mathit{Poly}\ a \to [\mathit{Poly}\ a] \\
\mathit{ppgcd} = \mathit{pgcd}\ \mathit{content}
\end{array}
$$

And now there is a pseudo-remainder sequence whose final element by some weird magic has the resultant of the two polynomials as its only coefficient! We could implement the sequence using a second-order function similar to *pgcd*. Unfortunately, we need state, since α does not only depend on the current remainder polynomial, but also on the previous α. To keep it simple we present a completely new implementation (which is inspired by *sympy*, a Python package for symbolic computation):

$$
\begin{array}{l}
\mathit{spgcd} :: \mathit{Poly\ Integer} \to \mathit{Poly\ Integer} \to [\mathit{Poly\ Integer}] \\
\mathit{spgcd}\ a\ b = \mathbf{let}\ n = \mathit{degree}\ a \\
\qquad\qquad\qquad\quad m = \mathit{degree}\ b \\
\qquad\qquad\qquad\quad d = n - m \\
\qquad\qquad\qquad\quad c = \mathit{scale}\ ((-1) \uparrow (d+1))\ (\mathit{prem}\ a\ b) \\
\qquad\qquad\qquad\quad l = \mathit{lc}\ b \\
\qquad\qquad\qquad\quad z = -(l \uparrow d) \\
\qquad\qquad\qquad\quad k = \mathit{degree}\ c \\
\qquad\qquad\quad \mathbf{in}\ c : \mathit{go}\ z\ l\ b\ c\ k\ (m-k) \\
\quad \mathbf{where}\ \mathit{go}\ c\ l\ f\ g\ n\ d \mid \mathit{degree}\ (g) \equiv 0 = [\,] \\
\qquad\qquad\qquad\qquad\qquad \mid \mathit{otherwise} \qquad = \\
\qquad\qquad\qquad\qquad\qquad\quad \mathbf{let}\ y\ \ = (-l) * c \uparrow d \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad h\ \ = P\ [x\ \text{`}\mathit{div}\text{`}\ y \mid x \leftarrow \mathit{coeffs}\ (\mathit{prem}\ f\ g)] \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad m = \mathit{degree}\ h \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad l'\ \ = \mathit{lc}\ g \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad c' \mid d > 1 \qquad = (-l') \uparrow d\ \text{`}\mathit{div}\text{`}\ (c \uparrow (d-1)) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid \mathit{otherwise} = -l' \\
\qquad\qquad\qquad\qquad\qquad\quad \mathbf{in}\ h : \mathit{go}\ c'\ l'\ g\ h\ m\ (n-m)
\end{array}
$$

The function has two parts: the first step which is performed in the body of the function itself and the recursive part which is implemented in *go*. The first step computes the degrees of the polynomials and their difference (n, m and d), an initial value to compute the first alpha (z) and the first pseudo remainder (c).

The subroutine *go* receives z (named c in *go*), l, the second polynomial (b named f in *go*), the first pseudo remainder (c, named g in *go*), the degree of c and the difference of the degrees of the two polynomials passed in ($m - k$ named d in *go*).

go computes y (the alpha), the new pseudo remainder h, the degree of h (m) and the next value to compute alpha (c'). Then, the process repeats. We terminate when the degree of the second pseudo remainder is 0.

The function is certainly somewhat confusing. But the outstanding fact is that it allows computing the resultant of two polynomials at roughly the same cost as the GCD. There is in particular nothing special involved in computing the alpha.

We can now define *res* (which we left out above) as:

$$
\begin{array}{l}
\mathit{res} :: \mathit{Poly\ Integer} \to \mathit{Poly\ Integer} \to \mathit{Integer} \\
\mathit{res}\ a\ b = \mathit{head}\ \$\ \mathit{coeffs}\ \$\ \mathit{last}\ \$\ \mathit{spgcd}\ a\ b
\end{array}
$$

With that, the implementation of *dis* is complete. We can compute the discriminant of our polynomials just using the *dis* function, which, in its turn, computes the discriminant of the polynomial without using the roots:

dis (P $[24, 14, -13, -2, 1]$) yields 4410000, *dis* $(P\ [-6,1,1])$ yields 25 and

dis $(P\ [-18,3,3])$ yields 225.

1.9 Factoring Polynomials

Polynomials can be factored in different contexts, for instance a field or the integers (which, as you may remember, do not form a field, but a ring). These contexts can be generalised to what is called a *unique factorisation domain*. A unique factorisation domain is a commutative ring R, where

- $uv \neq 0$, whenever $u, v \in R$ and $u \neq 0$ and $v \neq 0$;
- every nonzero element is a *unit*, a *prime* or can be uniquely represented as a product of primes;
- every unit u has an inverse v, such that $uv = 1$.
- a prime p is a nonunit element for which an equation of the form $p = qr$ is true, only if either q or r is a unit.

The integers form a unique factorisation domain, with the units 1 and -1 and the primes $\pm 2, \pm 3, \pm 5, \pm 7, \dots$ We can easily verify that 1 and -1 obey the definition of unit, when we assume that each one is its own inverse. We can also agree that the primes are primes in the sense of the above definition: for any prime in $p \in \mathbb{Z}$, if $p = qr$, then either q or r must be a unit and the other must equal p. That is the definition of primes.

A field is trivially a unique factorisation domain without primes where all elements are units.

The simplest notion of factoring in such a domain is the factoring into *primitive part* and *content*. This, basically, splits a polynomial into a number (in the domain we are dealing with) and a *primitive polynomial*.

With the integers, the content is the GCD of the coefficients. For instance, the GCD of the coefficients of the polynomial $9x^5 + 27x^2 + 81$ is 9. When we divide the polynomial by 9 we get $x^5 + 3x^2 + 9$.

For rational numbers, we would choose a fraction that turns all coefficients into integers that do not share divisors. The polynomial

$$\frac{1}{3}x^5 + \frac{7}{2}x^2 + 2x + 1,$$

for instance, can be factored dividing all coefficients by $\frac{1}{6}$:

$$
\begin{array}{lcccr}
\frac{1}{3} & \times & 6 & = & 2 \\
\frac{7}{2} & \times & 6 & = & 21 \\
2 & \times & 6 & = & 12 \\
1 & \times & 6 & = & 6
\end{array}
$$

We, hence, get the product $\frac{1}{6}(2x^5 + 21x^2 + 12x + 6)$.

This, however, is not the end of the story. Consider the polynomial

$$3x^2 - 27.$$

We can factor this one into $3(x^2 - 9)$, with the second part being primitve: the GCD of its coefficients is 1. But we can factor it further. Obviously, we have

$$x^2 - 9 = (x - 3)(x + 3). \tag{1.83}$$

The complete factorisation of the polynomial $3x^2 - 27$, hence, is $3(x - 3)(x + 3)$.

For factoring primitive polynomials manually, there are many different methods (most of which have a video on youtube). They share one property: they are highly inefficient, when it comes to polynomials of larger degrees or with big coefficients. They, basically, all use integer factorisation of which we know that it is extremely expensive in terms of computation complexity. Instead of going through all of them, we will here present a typical classical method, namely Kronecker's method.

Kronecker's method is a distinct-degree approach. That is, it searches for the factors of a given degree. We start by applying the polynomial to n distinct values, for n the degree of the factors plus 1. That is because, to represent a polynomial of degree d, we need $d+1$ coefficients, *e.g.* $P\ [0, 0, 1]$ has three coefficients and represents the polynomial x^2, which is of degree 2.

The rationale of applying the polynomial is the following: When the polynomial we want to factor generates a certain set of values, then the product of the factors of that polynomial must generate the same values. Any factor must, hence, consist of divisors of those values. The number of integer divisors of those values, however, is limited. We can therefore afford, at least for small polynomials with small coefficients, trying all the combinations of the divisors.

We have already defined a function to find the divisors of a given number, when we discussed Euler's totient function. However, that function dealt with natural numbers only. We now need a variant that is able to compute negative divisors. It would be also nice if that function could give us not only the divisors, but additionally the additive inverse, *i.e.* the negation of the divisors, because, in many cases, we need to look at the negative alternatives too. Here is an implementation:

$$\begin{aligned}
&divs :: Zahl \to [\,Zahl\,]\\
&divs\ i \mid i < 0 \qquad\ = divs\ (-i)\\
&\qquad\quad \mid otherwise = ds \mathbin{+\!\!\!+} map\ negate\ ds\\
&\quad \mathbf{where}\ ds = [\,d \mid d \leftarrow [1\,..\,i], rem\ i\ d \equiv 0\,]
\end{aligned}$$

The divisors are now combined to yield n-tuples with n still the degree of the factor plus one and each divisor representing one coefficient of the resulting polynomial. But before we can convert the n-tuples into polynomials, we need to create all possible permutations, since the polynomial $P\ [\,a, b\,]$ is not the same as $P\ [\,b, a\,]$ if $a \neq b$. From this we obtain a (potentially very large) list of n-tuples that we then convert into polynomials. From that list, we finally filter those polynomials for which p '$divp$' $k \equiv (\_, 0)$, where p is the input polynomial and k the candidate in the list of polynomials. Here is an implementation (using lists instead of n-tuples):

$$\begin{aligned}
&kronecker :: Poly\ Zahl \to [\,Zahl\,] \to [\,Poly\ Quoz\,]\\
&kronecker\ (P\ cs)\ is = nub\ [\,a \mid a \leftarrow as, snd\ (r\ \text{`}divp\text{`}\ a) \equiv P\ [0]\,]\\
&\quad \mathbf{where}\ ds = map\ divs\ is\\
&\qquad\qquad ps = concatMap\ perms\ (listcombine\ ds)\\
&\qquad\qquad as = map\ (P \circ map\ fromInteger)\ ps\\
&\qquad\qquad r\ \ = P\ [\,c \,\%\, 1 \mid c \leftarrow cs\,]
\end{aligned}$$

The function takes two arguments. The first is the polynomial we want to factor and the second is the list of results obtained by applying the polynomial. We then get the divisors ds, create all possible combinations of the divisors and all possible permutations of the resulting lists. We then convert the coefficients to rational numbers (since we later use $divp$). Finally, we filter all polynomials that leave no remainder when the input polynomial is divided by any one of them.

There are two combinatorial functions, *perms* and *listcombine*. We have already defined *perms*, when discussing permutations. The function generates all permutations of a given list. The other function, *listcombine*, however, is new. It creates all possible combinations of a list of lists. Here is a possible implementation:

$$\begin{aligned}
&listcombine :: [\,[\,a\,]\,] \to [\,[\,a\,]\,]\\
&listcombine\ [\,] \qquad\quad = [\,]\\
&listcombine\ ([\,]{:}\ _) \quad = [\,]\\
&listcombine\ (x : xs) = inshead\ (head\ x)\ (listcombine\ xs) \mathbin{+\!\!\!+}\\
&\qquad\qquad\qquad\qquad\quad listcombine\ ((tail\ x) : xs)\\
&inshead :: a \to [\,[\,a\,]\,] \to [\,[\,a\,]\,]\\
&inshead\ x\ [\,]\ = [\,[\,x\,]\,]\\
&inshead\ x\ zs = map\ (x{:})\ zs
\end{aligned}$$

Let us try *kronecker* on some polynomials. First, we need to apply the input polynomial to get n results. For instance, we know that the polynomial $x^2 - 9$ has factors of first degree. We, therefore, apply it on two values: **let** $vs = mapply\ (P\ [-9, 0, 1])\ [0, 1]$ and get for vs: $[-9, -8]$. Now we call $kronecker\ (P\ [-9, 0, 1])\ [-9, -8]$ and get:

$P\ [3\,\%\,1, 1\,\%\,1]$
$P\ [3\,\%\,1, (-1)\,\%\,1]$
$P\ [(-3)\,\%\,1, 1\,\%\,1]$
$P\ [(-3)\,\%\,1, (-1)\,\%\,1]$

Those are the polynomials $x + 3$, $-x + 3$, $x - 3$ and $-x - 3$. By convention, we exclude the polynomials starting with a negative coefficient by factoring -1 out. However, we can easily see that all of them are actually factors of $x^2 - 9$, since

$$(x + 3)(x - 3) = (x^2 - 9) \tag{1.84}$$

and

$$(-x + 3)(-x - 3) = (x^2 - 9). \tag{1.85}$$

Here is another example: $x^5 + x^4 + x^2 + x + 2$. We want to find a factor of degree 2, so we apply the polynomial to three values, say, $[-1, 0, 1]$. The result is $[2, 2, 6]$. We run $kronecker\ (P\ [2, 1, 1, 0, 1, 1])\ [2, 2, 6]$ and, after a short while, we get:

$P\ [1\,\%\,1, 1\,\%\,1, 1\,\%\,1]$
$P\ [2\,\%\,1, 2\,\%\,1, 2\,\%\,1]$
$P\ [(-1)\,\%\,1, (-1)\,\%\,1, (-1)\,\%\,1]$
$P\ [(-2)\,\%\,1, (-2)\,\%\,1, (-2)\,\%\,1]$,

which corresponds to the polynomials x^2+x+1, $2x^2+2x+2$, $-x^2-x-1$ and $-2x^2-2x-2$. Ony the first one is a primitive polynomial. We can factor out 2 from the second one, leaving just the first one; polynomials three and four, simply, are the negative counterparts of one and two, so we can factor out -1 and -2, respectively, to obtain again the first one.

To check if the first one is really a factor of the input polynomial we divide:

$P\ [2, 1, 1, 0, 1, 1]$ ‘*divp*‘ $P\ [1, 1, 1]$
and get $P\ [2, -1, 0, 1]$, which corresponds to $x^3 - x + 2$. Indeed:

$$(x^2 + x + 1)(x^3 - x + 2) = x^5 + x^4 + x^2 + x + 2. \tag{1.86}$$

Kronecker's method is just a brute force search. It is obvious that it is not efficient and will fail with growing degrees and coefficients. Modern methods to factor polynomials use much more sophisticated techniques.

They are, in particular, based on modular arithmetic and make use of theorems that we have already discussed in the ring of integers. Polynomials with coefficients in a ring (or field) form a ring too, a polynomial ring. Theorems that hold in any ring, hence, hold also in a polynomial ring. We, therefore, do not need to prove them here again.

We will discuss the methods for factoring polynomials in a finite field in the next section. Let us here assume that we already knew such a method. We could then call it to factor a given polynomial in a finite field and then reinterpret the result in the domain we started with.

1.10 Practical Factoring Techniques

1.11 Factoring Polynomials in a finite Field

Famous factorisation algorithms using modular arithmetic are *Berlekamp's algorithm* developed by the American mathematician Elwyn Berlekamp in the late Sixties and the *Cantor-Zassenhaus algorithm* developed in the late Seventies and early Eighties by David Cantor, an American mathematician, not to be confused with Georg Cantor, and Hans Zassenhaus (1912 – 1991), a German-American mathematician. We will here focus on Cantor-Zassenhaus, which is by today probably the most-used algorithm implemented in many computer algebra systems.

The contribution of Cantor-Zassenhaus, strictly speaking, is just one of several pieces. The whole approach is based on Euler's theorem, which, as you may remember, states that

$$a^{\varphi(n)} \equiv 1 \pmod{n}, \tag{1.87}$$

where $\varphi(n)$ is the totient function of n counting the numbers $1 \dots n-1$ that are coprime to n, *i.e.* that share no divisors with n.

Euler's theorem is defined as theorem over the ring of integers, which, by modular arithmetic, transforms into the finite field of the integers $0 \dots n-1$. Polynomial rings can be seen as extensions of the underlying ring (of integers). When we introduce modular arithmetic, that is, when we build polynomials on a finite field, they still constitute a ring, but now a ring built on top of a finite field. Notationally, this is usually expressed as $K[x]$, where K is a field and $K[x]$ the polynomial ring defined on top of K.

When we now take polynomials modulo a polynomial, we again get a finite field, this time a polynomial field of the form $K[x]/m$ (pronounced "over" m), where m is a polynomial. The point in doing this is that many properties of the original field K are preserved in $K[x]/m$ and Euler's theorem is one of them.

However, we need to redefine Euler's theorem to make clear what is meant by it in the new context. We are now dealing with the polynomial ring $K[x]$ and a polynomial $m \in K[x]$. Based on this, we can define the totient function as

$$\varphi(m) = |\{f \in K[x] : 0 \le f \le m \wedge \gcd(m, f) = 1\}|,$$

i.e. the cardinality of the set of all polynomials f less or equal than m that do not share divisors with m. For any such ring $K[x]$ and any $f \in K[x] : \gcd(m, f) = 1$, the following holds:

$$f^{\varphi(m)} \equiv 1 \pmod{m}. \tag{1.88}$$

The resulting structure $K[x]/(m)$ has a multiplicative group K_m^* (just as the integers $\pmod{n}$). The members of this group are all polynomials that do not share divisors with m and $\varphi(m)$ is the cardinality of this group. The equivalence may hold also for other numbers, a, such that $f^a \equiv 1 \pmod{m}$, but according to Lagrange's theorem (that the cardinality of subgroups of G divides the cardinality of G), all these numbers a must divide $\varphi(m)$, the size of the group. But independent of the possibility that other number may fulfil the equivalence, we unmistakenly have $f^{\varphi(m)} \equiv 1 \pmod{m}$.

From this theorem, Fermat's little theorem follows. Let K be a field with q elements; when using arithmetic modulo a prime p, then K_m^* is the group of numbers $1 \dots p-1$, which has $q = p - 1$ elements. Note that, when we refer to the multiplicative group of this field, we usually refer only to the numbers $1 \dots p-1$, *i.e.* $p - 1$ numbers. Now, let g be an *irreducible* polynomial, *i.e.* a non-constant polynomial that cannot be further factored and, hence, a "prime" in our polynomial ring, with degree d, $d > 0$. Then it holds for any polynomial f from this field

$$f^{q^d} \equiv f \pmod{g}. \tag{1.89}$$

We can prove this easily: We know that K has q elements. From this q elements we can create a limited number of polynomials. When you look at our Haskell representation of polynomials, you will easily convince yourself that the number of valid polynomials of a given degree d equals the number of valid numbers that can be presented in the numeral system base q with $d + 1$ digits. If, for instance, $q = 2$, then we have (without the zero-polynomial $P\ [0]$)

degree	size	polynomials
0	1	$P\ [1]$
1	2	$P\ [0,1], P\ [1,1]$
2	4	$P\ [0,0,1], P\ [1,0,1], P\ [0,1,1], P\ [1,1,1]$
3	8	$P\ [0,0,1,1], P\ [1,0,1,1], P\ [0,1,1,1], P\ [1,1,1,1]$ $P\ [0,0,0,1], P\ [1,0,0,1], P\ [0,1,0,1], P\ [1,1,0,1]$
...	...	...

We, hence, can precisely say how many polynomials of degree $< d$ there are, namely $r = q^d$. For the example $q = 2$, we see that there are 16 polynomials with degree less than 4, which is 2^4. One of those polynomials, however, is $P\ [0]$, which we must exclude, when asking for $\varphi(g)$ (since, for this polynomial, division is not defined). For the irreducible polynomial g, we therefore have $r - 1$ polynomials that do not share divisors with g, *i.e.* $\varphi(g) = r - 1$. So, according to Euler's theorem, we have

$$f^{r-1} \equiv 1 \pmod{g}. \tag{1.90}$$

Multiplying both sides by f, we get

$$f^r \equiv f \pmod{g}. \tag{1.91}$$

Since $r = q^d$, this is equivalent to 1.89 and this concludes the proof. □

From Fermat's theorem, we can derive a nice and useful corollary. Note that, when we subtract f from both sides of the equivalence, we would get 0 on the right-hand side, which means that g divides the expression on the left-hand side. Set $x = f$, then we have:

$$x^{q^d} - x \equiv 0 \pmod{g}. \tag{1.92}$$

This is the basis for a nice test for irreducibility. Since the group established by a non-irreducible polynomial of degree d has less than $p^d - 1$ elements, it will divide $x^{p^c} - x$ for some $c < d$, but an irreducible polynomial will not. Here is a Haskell implementation:

$$
\begin{array}{l}
\mathit{irreducible} :: \mathit{Natural} \rightarrow \mathit{Poly\ Natural} \rightarrow \mathit{Bool} \\
\mathit{irreducible}\ p\ u \quad \mid d < 2 \qquad\quad = \mathit{False} \\
\qquad\qquad\qquad\quad\ \mid \mathit{otherwise} = \mathit{go}\ 1\ x \\
\quad \textbf{where}\ d \qquad\ = \mathit{degree}\ u \\
\qquad\qquad\ x \qquad = P\ [0,1] \\
\qquad\qquad\ \mathit{go}\ i\ z = \textbf{let}\ z' = \mathit{powmp}\ p\ p\ z \\
\qquad\qquad\qquad\qquad\ \textbf{in case}\ \mathit{pmmod}\ p\ (\mathit{addp}\ p\ z'\ (P\ [0,p-1]))\ u\ \textbf{of} \\
\qquad\qquad\qquad\qquad\quad P\ [0] \quad \rightarrow i \equiv d \\
\qquad\qquad\qquad\qquad\quad g \qquad\ \rightarrow \textbf{if}\ i < d\ \textbf{then}\ \mathit{go}\ (i+1)\ (\mathit{pmmod}\ p\ z'\ u) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\ \mathit{False}
\end{array}
$$

The function receives two arguments: the modulus and the polynomial we want to check. First, we compute the degree of the polynomial. When the polynomial is of degree 0 or 1, there are by definition only trivial, *i.e.* constant factors. It is, hence, not irreducible (it is not reducible either, it is just uninteresting). Then we start the algorithm beginning with values 1 and x, where x is the simple polynomial x. In *go*, we raise this polynomial to the power of p, and subtract it from the result. Note that we add $p-1$, which, in modular arithmetic, is the same as subtracting 1. We take the result modulo the input polynomial u. This corresponds to $x^{p^d} - x$ for degree $d = 1$.

If the result is $P\ [0]$, *i.e.* zero, and the degree counter i equals d, then equation 1.92 is fulfilled. Otherwise, if the degree counter does not equal d, this polynomial fulfils the equation with a “wrong” degree. This is possible only if the input was not irreducible in the first place.

Finally, if we have a remainder that is not zero, we either continue (if we have not yet reached the degree in question) or, if we had already reached the final degree, we return with False, since the polynomial is certainly not irreducible.

Note that we continue with $\mathit{pmmod}\ p\ z'\ u$, that is, with the previous power modulo u. This is an important optimisation measure. If we did not do that, we would create gigantic polynomials. Imagine a polynomial of degree 8 modulo 11. To check that polynomial we would need to raise x to the power of 11^8, which would result in a polynomial of degree 214 358 881. Since the only thing we want to know is a value modulo u, we can reduce the overhead of taking powers by taking them modulo u in the first place.

Let us look at an example. We generate a random polynomial of degree 3 modulo 7:

$g \leftarrow \mathit{randomPoly}\ 7\ 4$

I get the polynomial $P\ [3,3,3,4]$. (Note that you may get another one!) Calling $\mathit{irreducible}\ 7\ g$ says: *False*.

When we raise the polynomial $P\ [0,1]$ to the power of $7^3 = 343$, we get a polynomial of degree 343 with the leading coefficient 1. When we subtract $P\ [0,1]$ from it, it will have -1, which is 6 in this case, as last but one coefficient. Taking this modulo to the random polynomial g, we get the polynomial $P\ [0, 3, 6]$, which is $6x^2 + 3x$ and definitely not constant. g is therefore not irreducible.

Let us try another one:

$g \leftarrow \mathit{randomPoly}\ 7\ 4$

This time, I get $P\ [3, 1, 4, 4]$. Calling *irreducible* 7 g says: *True*. When we take $x^{7^3} - x$ modulo g, we get $P\ [0]$. But we do not get a constant polynomial for $x^7 - x$ or $x^{7^2} - x$. $P\ [3, 1, 4, 4]$, hence, is irreducible.

The formula, however, is not only interesting for testing irreducibility. What the formula states is in fact that all irreducible polynomials up to degree d are factors of $x^{q^d} - x$. The irreducible factors of the polynomial we want to factor are part of this product and we can get them out just by asking for the greatest common divisor of $x^{p^d} - x$ and the polynomial we want to factor. This would give us the product of all factors of our polynomial of a given degree.

Consider for example the polynomials modulo 2 of degree 2 in the table above. There is only one irreducible polynomial in this row, namely $P\ [1, 1, 1]$. We compute $x^4 - x$, which is $P\ [0, 1, 0, 0, 1]$ and now divide this one by $P\ [1, 1, 1]$:

$\mathit{divmp}\ 2\ (P\ [0, 1, 0, 0, 1])\ (P\ [1, 1, 1])$

The result is $P\ [0, 1, 1]$, yet another polynomial of degree 2. This one, however, is not irreducible. It can be factored into the polynomials $P\ [1, 1]$ and $P\ [0, 1]$. $x^4 - x$, hence, can be factored into three irreducible polynomials, one of degree 2 and two of degree 1.

We need to add one more qualification however. Since we are searching for a *unique* factorisation, we should make sure that we always make the polynomial *monic*, that is, we should remove the leading coefficient by dividing all coefficients by it. This corresponds to content-and-primitive-part factorisation as already discussed above, but in the case of modular arithmetic it is much simpler. Whatever the leading coefficients is, we can just multiply all coefficients by its inverse without worrying about coefficients becoming fractions. Here is an implementation:

```
monicp :: Natural -> Poly Natural -> Poly Natural
monicp p u = let cs = coeffs u
                 k  = last cs `M.inverse` p
             in P (map (modmul p k) cs)
```

The following function, obtains the products of the factors of a given (monic) polynomial degree by degree. Note that we give the result back as a monic polynomial again. Each result is a tuple of the degree and the corresponding factor product.

$$
\begin{array}{l}
\mathit{ddfac} :: \mathit{Natural} \to \mathit{Poly\ Natural} \to [(\mathit{Int}, \mathit{Poly\ Natural})]\\
\mathit{ddfac}\ p\ u \quad = \mathit{go}\ 1\ u\ (P\ [0,1])\\
\quad \textbf{where}\ n = \mathit{degree}\ u\\
\qquad\qquad \mathit{go}\ d\ v\ x \mid \mathit{degree}\ v \leqslant 0 = [\,]\\
\qquad\qquad\qquad\qquad\quad \mid \mathit{otherwise} \quad =\\
\qquad\qquad\qquad\qquad\quad \textbf{let}\ x' \qquad = \mathit{powmp}\ p\ p\ x\\
\qquad\qquad\qquad\qquad\qquad\ t \qquad\ = \mathit{addp}\ p\ x'\ (P\ [0, p-1])\\
\qquad\qquad\qquad\qquad\qquad\ g \qquad\ = \mathit{gcdmp}\ p\ t\ v\\
\qquad\qquad\qquad\qquad\qquad\ (v', \_) \ \ = \mathit{divmp}\ p\ v\ g\\
\qquad\qquad\qquad\qquad\qquad\ r \qquad\ = (d, \mathit{monicp}\ p\ g)\\
\qquad\qquad\qquad\qquad\quad \textbf{in case}\ g\ \textbf{of}\\
\qquad\qquad\qquad\qquad\qquad P\ [\_] \to \quad\ \ \mathit{go}\ (d+1)\ v'\ (\mathit{pmmod}\ p\ x'\ u)\\
\qquad\qquad\qquad\qquad\qquad \_ \qquad\ \to r : \mathit{go}\ (d+1)\ v'\ (\mathit{pmmod}\ p\ x'\ u)
\end{array}
$$

The real work is done by function *go*. It starts with degree $d = 1$, the polynomial u we want to factor and, again, the simple polynomial x. We then raise x to the power p^1 for the first degree, subtract x from the result and compute the *gcd*. If the result is a constant polynomial, there are no non-trivial factors of this degree and we continue. Otherwise, we store the result with the degree, making g monic.

We continue with the next degree, $d+1$, the quotient of the polynomial we started with and the factor product g we obtained and the power of x' reduced to the modulo u. The latter is again an optimisation. The former, however, is essential to avoid generating the same factor product over and over again. By dividing the input polynomial by g, we make sure that the factors we have already found are taken out. This works only if the polynomial is squarefree of course. (You might remember the discussion of squarefree numbers in the context of Euler's theorem where we found that, if n is squarefree, then $\varphi(n) = \prod_{p|n} p - 1$, *i.e.* the totient number of n is the product of the primes in the factorisation of n all reduced by 1.) We need to come back to this topic and, for the moment, make sure that we only apply polynomials that are squarefree and monic.

We try *ddfac* on the 4-degree polynomial $u(x) = x^4 + x^3 + 3x^2 + 4x + 5$ modulo 7 and call *ddfac* 7 u and obtain the result

$[(1, P\ [2,4,1]), (2, P\ [6,4,1])]$,

i.e. the factor product x^2+4x+2 for degree 1 and the factor product x^2+4x+6 for degree 2. First, we make sure that these are really factors of u by calling *divmp* 7 $(P\ [2,4,1])$, which shows

$(P\ [6,4,1], P\ [0])$.

We can conclude that these are indeed all the factors of u. But, obviously, $P\ [2,4,1]$ or x^2+4x+2 is not irreducible, since it is a second-degree polynomial, but it was obtained for the irreducible factors of degree 1. $P\ [6,4,1]$, on the other hand, was obtained for degree 2 and is itself of degree 2. We can therefore assume that it is already irreducible, but let us check: *irreducible* 7 $(P\ [6,4,1])$, indeed, yields *True*.

But what about the other one? How can we get the irreducible factors out of that one? Here Cantor and Zassenhaus come in. They proposed a simple algorithm with the following logic. We, again, use the magic polynomial $x^{p^d} - x$, but choose a specific polynomial for x, say t. We already have that chunk of irreducible polynomials hidden in $(P\ [2,4,1])$, let us call it u, and know that those polynomials are factors of both, $t^{p^d} - t$ and u. The approach of Cantor and Zassenhaus is to split the factors so that the problem reduces significantly. We can split t into three parts using the equality

$$t^{p^d} - t = t(t^{(p^d-1)/2} + 1)(t^{(p^d-1)/2} - 1). \tag{1.93}$$

By a careful choice of t, we can make sure that the factors are likely to be more or less equally distributed among the latter two factors. That, indeed, would reduce the problem significantly.

Since u and $t^{p^d} - t$ share factors, we can transform the equality into the following variant:

$$u = \gcd(u,t) \times \gcd(u,(t^{(p^d-1)/2} + 1)) \times \gcd(u,((t^{(p^d-1)/2} - 1)) \tag{1.94}$$

A reasonable choice for t is a polynomial of degree $2d - 1$. With high probability, the factors are equally distributed among the latter two factors of the equation and we indeed reduce the problem significantly. To do so, we compute one of the gcds and continue splitting this gcd and the quotient of u and the gcd further. Should we be unlucky (the gcd contains either no or all of the factors), we just try again with another choice for t. After some tries (less than three according to common wisdom), we will hit a common factor.

There is an issue, however, for $p = 2$. Because in that case, $t^{(p^d-1)/2} - 1 = t^{(p^d-1)/2} + 1$. Consider a polynomial modulo 2, for instance $P\ [0,1,1]$ and $d = 3$. Then we have

$$(p^d - 1)/2 = (2^3 - 1)/2 = 7/2 = 3.$$

We raise the polynomial to the power of 3 and get $[0,0,0,1,1,1,1]$. When we add $P\ [1]$, we get $[1,0,0,1,1,1,1]$. But what do we subtract? Let us try *modp* 2 $(P\ [-1])$. We get back $P\ [1]$. Adding and subtracting 1 is just the same thing here.

But that would mean that our formula would be much poorer. We would not have three different factors, but only two, namely t and $t^{(p^d-1)/2} + 1$. Unfortunately, it is very likely that all the factors end up in the second one and with this, we would not simplify the problem.

The fact that we are now working modulo 2 may help. We first observe that, modulo 2, there is no difference between the polynomials $t^{2^d} - t$ (the magic one with $p = 2$) and $t^{2^d} + t$. The second one, however, is easy to split, when we set

$$w = t + t^2 + t^4 + \cdots + t^{2^{d-1}}.$$

Then, w^2 would be

$$t^2 + t^4 + \cdots + t^{2d}.$$

This may shock you on the first sight. But remember, we are still working modulo p and we have (*freshman's dream*):

$$(a + b)^p \equiv a^p + b^p \pmod{p}.$$

When multiplying w by itself, we would get

$$t^2 + 2t^3 + t^4 + 2t^5 + 2t^6 + t^8.$$

Since we are working modulo 2, all terms with even coefficients cancel out, we, hence, get

$$t^2 + t^4 + t^8.$$

Now, observe that

$$w^2 + w = t^2 + t^4 + \cdots + t^{2^d} + t + t^2 + \cdots + t^{2^{d-1}},$$

when we rearrange according to exponents, we again get pairs of equal terms:

$$w^2 + w = t + 2t^2 + 2t^4 + \cdots + 2t^{2^{d-1}} + t^{2^d}.$$

When we compute this modulo 2, again all terms with even coefficients fall away and we finally get

$$w^2 + w = t^{2^d} + t. \tag{1.95}$$

The point of all this is that we can split the expression $w^2 + w$ into two more or less equal parts, just by factoring w out: $w(w + 1)$. Now, it is again very probable that we find common divisors in both of the factors, w or $w + 1$ making it likely that we can reduce the problem by taking the gcd with one of them. Here is an implementation of the Cantor-Zassenhaus algorithm:

$$
\begin{array}{l}
cz :: \mathit{Natural} \rightarrow \mathit{Int} \rightarrow \mathit{Poly\ Natural} \rightarrow \mathit{IO}\ [\mathit{Poly\ Natural}] \\
cz\ p\ d\ u \mid n \leqslant d \qquad = \mathit{return}\ [\mathit{monicp}\ p\ u] \\
\qquad\qquad \mid \mathit{otherwise} = \mathbf{do} \\
\quad x \leftarrow \mathit{monicp}\ p < \$ > \mathit{randomPoly}\ p\ (2 * d) \\
\quad \mathbf{let}\ t \quad \mid p \equiv 2 \qquad = \mathit{addsquares}\ (d-1)\ p\ x\ u \\
\qquad\qquad \mid \mathit{otherwise} = \mathit{addp}\ p\ (\mathit{powmodp}\ p\ m\ x\ u)\ (P\ [p-1]) \\
\quad \mathbf{let}\ r = \mathit{gcdmp}\ p\ u\ t \\
\quad \mathbf{if}\ \mathit{degree}\ r \equiv 0 \vee \mathit{degree}\ r \equiv n\ \mathbf{then}\ cz\ p\ d\ u \\
\qquad \mathbf{else\ do}\ r1 \leftarrow cz\ p\ d\ r \\
\qquad\qquad\qquad r2 \leftarrow cz\ p\ d\ (\mathit{fst}\ \$\ \mathit{divmp}\ p\ u\ r) \\
\qquad\qquad\qquad \mathit{return}\ (r1 +\!\!+\ r2) \\
\quad \mathbf{where}\ n = \mathit{degree}\ u \\
\qquad\qquad m = (p \uparrow d - 1)\ \text{`}\mathit{div}\text{`}\ 2
\end{array}
$$

The function receives a natural number, that is the modulus p, an *Int*, d, for the degree, and the polynomial u, the factor product, which we both obtained from *ddfac*. When the degree is equal or greater than n, the degree of u, we are done: we already have a factor of the predicted degree. Otherwise, we generate a random monic polynomial of degree $2d-1$. Note that, since *randomPoly* expects the number of coefficients, which is $d+1$, we just pass $2d$.

Then we calculate t. If p is 2, we use *addsquares*, at which we will look in a moment. Otherwise, we raise the random polynomial to the power of $(p^d-1)/2$ and subtract 1. That is the third factor of equation 1.94. We compute the gcd and, if the result has either degree 0 (no factor was found) or the same degree as u (all factors are in this one), we just try again with another random polynomial. Otherwise, we continue with the gcd and the quotient $u/\,\mathrm{gcd}$.

Let us try this for the result $(1, P\ [2,4,1])$ we obtained earlier from applying *ddfac* on $P\ [5,4,3,1,1]$. We call $cz\ 7\ 1\ (P\ [2,4,1])$ and see

$[P\ [6,1], P\ [5,1]]$,

two irreducible polynomials of degree 1. The complete factorisation of $P\ [5,4,3,1,1]$ is therefore

$[P\ [6,1], P\ [5,1], P\ [6,4,1]]$,

which we can test by calling $\mathit{prodp}\ (\mathit{mulmp}\ 7)\ [P\ [6,1], P\ [5,1], P\ [6,4,1]]$ and we, indeed, get $P\ [5,4,3,1,1]$ back.

For the case where $p=2$, we use the function addsquares:

$$
\begin{array}{l}
\mathit{addsquares} :: \mathit{Int} \to \mathit{Natural} \to \mathit{Poly\ Natural} \to \mathit{Poly\ Natural} \to \mathit{Poly\ Natural} \\
\mathit{addsquares}\ i\ p\ x\ u = \mathit{go}\ i\ x\ x \\
\quad \textbf{where}\ \mathit{go}\ 0\ w\ \_ = w \\
\qquad\qquad \mathit{go}\ k\ w\ t = \textbf{let}\ t' \ = \mathit{pmmod}\ p\ (\mathit{powmp}\ p\ p\ t)\ u \\
\qquad\qquad\qquad\qquad\qquad\quad w' = \mathit{addp}\ p\ w\ t' \\
\qquad\qquad\qquad\qquad\quad \textbf{in}\ \mathit{go}\ (k-1)\ w'\ t'
\end{array}
$$

which just computes w as $t + t^2 + t^4 + \dots t^{2^{d-1}}$.

Let us try *ddfac* and *cz* with a polynomial modulo 2, *e.g.* $P\ [0, 1, 1, 1, 0, 0, 1, 1, 1]$, which is of degree 8 and is squarefree (and, per definition, monic). The call

$\mathit{ddfac}\ 2\ (P\ [0, 1, 1, 1, 0, 0, 1, 1, 1])$

gives us three chunks of factors:

$[(1, P\ [0, 1, 1]), (2, P\ [1, 1, 1]), (4, P\ [1, 1, 1, 1, 1])]$.

We see at once that the second and third polynomials are already irreducible, since they have the specified degree. The first one, however, is of degree 2, but shall contain factors of degree 1. So, let us see what $\mathit{cz}\ 2\ 1\ (P\ [0, 1, 1])$ will yield:

$[P\ [0, 1], P\ [1, 1]]$.

The complete factorisation of $P\ [0, 1, 1, 1, 0, 0, 1, 1, 1]$ is therefore

$[P\ [0, 1], P\ [1, 1], P\ [1, 1, 1], P\ [1, 1, 1, 1, 1]]$.

We can test with

$\mathit{prodp}\ (\mathit{mulmp}\ 2)\ [P\ [0, 1], P\ [1, 1], P\ [1, 1, 1], P\ [1, 1, 1, 1, 1]]$

which indeed results in $P\ [0, 1, 1, 1, 0, 0, 1, 1, 1]$.

Now, we still have to solve the problem of polynomials containing squared factors, *i.e.* repeated roots. There is in fact a method to find such factors adopted from calculus and, again, related to the derivative. It is based on the observation that a polynomial π and its derivative π' share only those factors that appear more than once in the factorisation of π. We have not enough knowledge on derivatives yet to prove that here rigorously, but we can get an intuition.

Consider a polynomial with the factorisation

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

This is a product and, to find the derivative of this polynomial, we need to apply the *product rule* (which we will study in part 3). The product rule states that

$$(fg)' = fg' + f'g, \tag{1.96}$$

i.e. the derivative of the product of f and g is the sum of the product of f and the derivative of g and the product of the derivative of f and g.

The derivatives of the individual factors $(x + a)(x + b)$ in this example all reduce to 1, since for $f = x^1 + a$, $f' = 1 \times x^0 = 1$. The product of factors, hence, turns into a sum of factors:

$$1 \times (x + a) + 1 \times (x + b) = (x + a) + (x + b) = 2x + a + b.$$

Let us check this result: when we build the product of the factors $(x+a)(x+b)$, we get the polynomial $x^2 + (a + b)x + ab$. Its derivative is $2x + a + b$, which is indeed the same result.

It is intuitively clear that the sum of the factors is not the same as the product of those same factors. Furthermore, the factors are irreducible and do not share divisors among each other; they are coprime. In consequence, the original factors disappear in favour of others they also do not share divisors with, because, since the factors are coprime to each other, they do not share divisors with their sum either.

Now consider polynomials with more than two factors of the form

$$abc\dots,$$

where a, b and c stand for irreducible polynomials like $(x + \alpha)$, $(x + \beta)$, $(x + \gamma)$ and so on.

We apply the product rule on the first two factors and get:

$$(a'b + ab')\dots$$

When we now apply the product rule once again, we would multiply c with the derivative of ab (which is $a'b + ab'$) and the derivative of c, c', with the original ab and get:

$$(a'b + ab')c + abc' = a'bc + ab'c + abc'.$$

We see that we end up with the sum of the products of the original factors, with the current factor i substituted by something else, namely the derivative of this factor. For the example above where the derivative was 1, we would have:

$$bc + ac + ab.$$

The general result can be represented by the following remarkable formula:

$$\left(\prod_{i=0}^{k} a_i\right)' = \sum_{i=0}^{k} \left(a_i' \prod_{j \neq i} a_j\right) \tag{1.97}$$

There is a striking similarity to the structure we found in analysing the Chinese remainder theorem, when we divided the product of all remainders by the current remainder. Just as in the Chinese remainder theorem, each of the terms resulting from the product rule is coprime to the original factor at the same position, since it is the product of all other irreducible factors (which, hence, are coprime to each other) and the derivative of that factor, which, for sure, does not share divisors with the original factor at that position.

When we have a repeated factor, however, as in the following polynomial

$$(x + a)(x + a)(x + b) \dots,$$

then this factor is preserved. The product rule will create the factor $x+a+x+a = 2x+2a$, which is the original factor scaled up. This factor is therefore preserved.

Suppose we want to compute the factorisation of

$$f = a_1 a_2^2 a_3^3 \dots a_k^k, \tag{1.98}$$

where the as represent the products of all the factors that are raised to the same exponent, then, since the derivative preserves the repeated factors, the gcd of f and its derivative f' is:

$$\gcd(f, f') = a_2^1 a_3^2 \dots a_k^{k-1}, \tag{1.99}$$

i.e. the repeated factors with the exponent decreased by one. Then f divided by the gcd gives us

$$\frac{f}{\gcd(f, f')} = a_1 a_2 a_3 \dots a_k, \tag{1.100}$$

all the factors reduced to their first power. Now, if we continue this scheme using the $\gcd(f, f')$ and $f/\gcd(f, f')$ as input, we would get 1.99 reduced once more ($a_3 a_4^2 \dots$) and 1.100 with the head chopped off ($a_2 a_3 \dots$). The quotient of the two versions of 1.100, *i.e.*

$$\frac{a_1 a_2 a_3 \dots}{a_2 a_3 \dots},$$

would give us the head. This leads to an iterative algorithm where we can process the factors with different exponents one by one advancing by chopping off the factors that we have already treated on each step.

In a finite field, this, unfortunately does not work in all cases. Problematic are all coefficients with exponents that are multiples of the modulus. When we compute nc^{n-1}, for n an exponent in the original polynomial that is a multiple of the modulus, the coefficient itself becomes zero. If we are unlucky, the derivative *disappears*, *i.e.* it becomes zero. A simple example is the polynomial $x^4 \pmod 2$. When we compute the derivative, we get $4x^3$. Unfortunately, 4 is a multiple of 2 and, therefore, the only nonzero coefficient we had in the original polynomial becomes zero and the entire derivative disappears.

What we can do, however, is to keep the coefficients with exponents that are multiples of the modulus separated from those that are not. We would still iteratively compute two sequences of values, namely $T_{k+1} = T_k/V_{k+1}$ with $T_1 = \gcd(f, f')$ and $V_{k+1} = \gcd(T_k, V_k)$ with $V_1 = f/T_1$. But we would now deviate for all k that are multiples of p, *viz.*

$$V_{k+1} = \begin{cases} \gcd(T_k, V_k) & if\ p \nmid k \text{ (as before)} \\ V_k & if\ p \mid k \end{cases}$$

At each step, we have

$$V_k = \prod_{i \geq k, p \nmid i} a_i, \tag{1.101}$$

i.e., the product of all as with exponents greater than those that we have already processed and that do not divide p, and

$$T_k = \prod_{i \geq k, p \nmid i} a_i^{i-k} \prod_{i \geq k, p \mid i} a_i^i, \tag{1.102}$$

i.e., the product of the powers greater than those we have already processed for both cases $p \mid i$ and $p \nmid i$. For the cases $p \nmid i$, everything is as before. For the cases $p \mid i$, we will end up, when we have reduced V_k to a constant polynomial, with a product of all the powers of the as with exponents that are multiples of p.

To get these as out, we divide all exponents by p and repeat the whole algorithm. For the return value, *i.e.* the factors, we need to remember the original exponent, but that is easily done as shown below.

Note that for polynomials with many coefficients, this recursion step will occur more than once. The exponents that are multiples of p in such a polynomial have the form

$$0p, p, 2p, 3p, 4p, \dots$$

Dividing by p, we get

$$0, 1, p, 2p, 3p, \dots$$

So, we need to repeat, until there are no more multiples of p. Here is the algorithm:

```haskell
sqmp :: Integer -> Integer -> Poly Integer -> [(Integer, Poly Integer)]
sqmp p e u | degree u < 1 = []
           | otherwise    = let u' = derivative (modmul p) u
                                t  = gcdmp p u u'
                                v  = fst (divmp p u t)
                            in go 1 t v
  where go k tk vk = let vk' | k `rem` p /= 0 = gcdmp p tk vk
                             | otherwise      = vk
                         tk' = fst (divmp p tk vk')
                         k'  = k + 1
                     in case divmp p vk vk' of
                          (P [_], _) ->                nextStep k' tk' vk'
                          (f, _)     -> (k * p ^ e, f) : nextStep k' tk' vk'
        nextStep k tk vk | degree vk > 0 = go k tk vk
                         | degree tk > 0 = sqmp p (e + 1) (dividedTk tk)
                         | otherwise     = []
        dividedTk tk = poly (divExp 0 (coeffs tk))
        divExp _ [] = []
        divExp i (c : cs) | i `rem` p == 0 = c : divExp (i + 1) cs
                          | otherwise      =     divExp (i + 1) cs
```

As usual, the hard work is done in the local function *go*, which takes three arguments, k, t_k and v_k. We initialise $k = 1$, $t_k = \gcd(u, u')$ and $v_k = u/t_k$. We set $v_{k+1} = \gcd(t_k, v_k)$, if $p \nmid k$, and $v_{k+1} = v_k$, otherwise. We further set $t_{k+1} = t_k/v_{k+1}$ and $k = k + 1$. If v_k/v_{k+1} (this is the head) is not constant (otherwise it is irrelevant), we remember the result as the product of factors with this exponent. Note that the overall result is a list of tuples, where the first element represents the exponent and the second the factor product. The exponent is calculated as $k \times p^e$. The number e, here, is not the Euler-Napier constant, but a variable passed in to *sqmp*. We would start the algorithm with $e = 0$. We, hence, get $k \times p^0 = k \times 1 = k$ for the first recursion.

The function *nextStep* is just a convenient wrapper for the decision of how to continue. If v_k is not yet constant, we continue with *go* $(k + 1)$ t_{k+1} v_{k+1}. Otherwise, if t_k is not yet constant, we continue with *sqmp* with $e+1$ and t_k with exponents that are multiples of p divided by p.

For bootstrapping the algorithm, we can define a simple function with a reasonable name that calls *sqmp* with $e = 0$:

$$\begin{aligned}
&\mathit{squarefactormod} :: \mathit{Integer} \to \mathit{Poly\ Integer} \to [(\mathit{Integer}, \mathit{Poly\ Integer})]\\
&\mathit{squarefactormod}\ p = \mathit{sqmp}\ p\ 0
\end{aligned}$$

Finally, we are ready to put everything together:

$$\begin{aligned}
&\mathit{cantorzassenhaus} :: \mathit{Integer} \to \mathit{Poly\ Integer} \to \mathit{IO}\ [(\mathit{Integer}, \mathit{Poly\ Integer})]\\
&\mathit{cantorzassenhaus}\ p\ u \mid \mathit{irreducible}\ p\ m = \mathit{return}\ [(1, m)]\\
&\qquad\qquad\qquad\qquad\quad \mid \mathit{otherwise} \qquad =\\
&\qquad\qquad\qquad\qquad\qquad \mathit{concat} <\$> \mathit{mapM}\ \mathit{mexpcz}\ [(e, \mathit{ddfac}\ p\ f) \mid\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad (e, f) \leftarrow \mathit{squarefactormod}\ p\ m]\\
&\quad \textbf{where}\ m = \mathit{monicp}\ p\ u\\
&\qquad\qquad \mathit{expcz}\ e\ (d, v) \quad = \mathit{map}\ (\lambda f \to (e, f)) <\$> \mathit{cz}\ p\ d\ v\\
&\qquad\qquad \mathit{mexpcz}\ (e, \mathit{dds}) = \mathit{concat} <\$> \mathit{mapM}\ (\mathit{expcz}\ e)\ \mathit{dds}
\end{aligned}$$

The function returns a list of pairs. The first of the pair is the exponent of the factor that, in its turn, is the second of the pair.

We first test whether the polynomial is irreducible. If so, we just return that polynomial as its only factor.

Otherwise, we create a list of exponents and factors using *squarefactormod* and pass the factors (without the exponents) to *ddfac* creating pairs of exponent and the result of *ddfac*. On this list, we map cz performing some acrobatics to pass the correct parameters, since the result of *ddfac* is a list of pairs itself, namely of pairs (*degree*, *factor product*). Finally, we reassign the exponent per result creating the list that we return to the caller of *cantorzassenhaus*.

Hans Zassenhaus worked most of his life as computeralgebraist and pioneered this area of mathematics and computer science. He was born in Germany before the second world war and studied mathematics under Emil Artin, one of the founders of modern algebra. Zassenhaus' father was strongly influenced by Albert Schweitzer and, as such, opposed to Nazi ideology. Hans shared this antipathy and, to avoid being drafted to a significant war effort like, as it would appear natural for an algebraist, cryptography, he left university and volonteered for the army weather forecast where he survived the war. Later, he would follow invitations first to the UK and later to the USA, where he remained until his death.

His sister Hiltgunt (who, after emigrating to the USA, preferred to use her second name Margret) studied Scandinavistics. During the war, she worked as translator for censorship in camps for Norwegian and Danish prisoners. She undermined censorship in this position, maintained contact between prisioners and helped smuggling medicine, tobacco and food into the prisons. For her efforts during and after the war, she was nominated for the Nobel Peace Prize in 1974.

1.12 Hensel Lifting

1.13 Enumerating the Algebraic Numbers