1 Polynomials

1.1 Numeral Systems

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

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

$$10101010,$$

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

$$170,$$

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

$$aa,$$

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

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

$$a_nb^n + a_{n-1}b^{n-1} + \dots + a_0b^0,$$

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

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

which can be simplified to

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

which, in its turn, is

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

The string 170 in decimal notation is interpreted as

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

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

$$a \times 16 + a.$$

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

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

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

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

or shorter:

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

with $a_i, x \in F$.

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

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

$$5x^2$$

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

$$x^5 + 2x.$$

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

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

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

translates to a formula involving binomial coefficients:

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

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

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

which yields just another polynomial:

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

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

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

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

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

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

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

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

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

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

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

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

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

We add a safe constructor:

```
poly :: (Eq a, Num a) ⇒ [a] → Poly a
poly [] = error "not a polynomial"
poly as = P (cleanz as)
cleanz :: (Eq a, Num a) ⇒ [a] → [a]
cleanz xs = reverse $ go (reverse xs)
  where go []       = []
        go [0]      = [0]
        go (0 : xs) = go xs
        go xs       = xs
```

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

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

But, here, we store them in the order:

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

The following function gets the list of coefficients back:

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

Here is a function to pretty-print polynomials:

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

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

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

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

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

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

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

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

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

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

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

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

1.2 Polynomial Arithmetic

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 sort them according to the exponents of their terms and add (or subtract) terms with equal exponents:

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

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 (−)

sump :: (Num a, Eq a) ⇒ [Poly a] → Poly a
sump = foldl′ add (P [0])

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)

strichlist :: (Num a, Eq a) ⇒ (a → a → a) → [a] → [a] → [a]
strichlist o xs ys = cleanz (go xs ys)
  where go [] bs            = bs
        go as []            = as
        go (a : as) (b : bs) = a ‘o‘ b : go as bs
```

Note that *sump* function, which implements *sum* for polynomials. 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) []
```

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

```
mul1 :: Num a ⇒ (a → a → a) → Int → [a] → a → [a]
mul1 o i as a = zeros i ++ go as a
  where go [] _       = []
        go (c : cs) x = c ‘o‘ x : go cs x

zeros :: Num a ⇒ Int → [a]
zeros i = take i $ repeat 0
```

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*:

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

which is

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

This means that

$$(4x^3+3x^2+2x+1)\times(8x^3+7x^2+6x+5) = 32x^6+52x^5+61x^4+60x^3+34x^2+16x+5. \quad (1.4)$$

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:

```
powp :: (Show a, Num a, Eq a) ⇒ Natural → Poly a → Poly a
powp f poly = go f (P [1]) poly
  where go 0 y _ = y
        go 1 y x = mul y x
        go n y x | even n    = go (n `div` 2) y       (mul x x)
                 | otherwise = go ((n − 1) `div` 2) (mul y x)
                                                   (mul x x)
```

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. Otherwise, if the exponent is even, we halve it, pass the base polynomial on and square the input. Otherwise, we pass the product of the base polynomial and the input on instead of the base polynomial as it is. 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

$go\ 5\ (P\ [1])\ (P\ [1,1])$,

which, since 5 is odd, results in

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

This, in its turn, results in

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

This is the final step and results in

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 our Haskell notation shows the binomial coefficients $\binom{n}{k}$ for $n=0$, $n=1$, $n=2$, $n=4$ and $n=5$. We never see $n=3$, which would be $P\ [1,3,3,1]$, because we leave the multiplication *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, a bit 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:

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

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

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

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

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

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 term 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*:

```
divides :: (Show a, Num a, Eq a, Ord a) ⇒
           Poly a → Poly a → Bool
divides a b = case b `divp` a of
                (_, P [0]) → True
                _          → False
```

the remainder:

```
remp :: (Show a, Num a, Eq a, Ord a) ⇒
        Poly a → Poly a → Bool
remp a b = let (_, r) = b `d` a in r
```

and, of course, the GCD:

```
gcdp :: (Show a, Num a, Eq a, Fractional a, Ord a) ⇒
        Poly a → Poly a → Poly a
gcdp a b | degree b > degree a = gcdp b a
         | zerop b    = a
         | otherwise = let (_, r) = divp a b in gcdp b r
```

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

```
zerop :: (Num a, Eq a) ⇒ Poly a → Bool
zerop (P [0]) = True
zerpo _       = False
```

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 in the same way, 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 $P\ [1,2,1]$. The reason is that the GCD is an operation defined on integers, but we implemented it on top of fractionals. That is often not what we want. Anyway, here, we will actually use the GCD only in finite fields. Until now, we have discussed polynomials in infinite fields. We now turn our attention to polynomial arithmetic in a finite field and, hence, 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 :: Zahl → Zahl → Zahl
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:

$modmul :: Zahl \to Zahl \to Zahl \to Zahl$
$modmul\ p\ f1\ f2 = (f1 * f2)\ `mmod`\ p$

For division, we reuse the *inverse* function:

$modiv :: Zahl \to Zahl \to Zahl \to Zahl$
$modiv\ p\ n\ d = modmul\ p\ n\ d'$
$\quad \textbf{where}\ d' = M.inverse\ d\ p$

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

$pmod :: Poly\ Zahl \to Zahl \to Poly\ Zahl$
$pmod\ (P\ cs)\ p = P\ [c\ `mmod`\ p \mid c \leftarrow cs]$

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:

$addmp :: Zahl \to Poly\ Zahl \to Poly\ Zahl \to Poly\ Zahl$
$addmp\ n\ p1\ p2 = strich\ (+)\ p1\ p2\ `pmod`\ n$
$submp :: Zahl \to Poly\ Zahl \to Poly\ Zahl \to Poly\ Zahl$
$submp\ n\ p1\ p2 = strich\ (-)\ p1\ p2\ `pmod`\ n$

Multiplication:

```
mulmp :: Zahl → Poly Zahl → Poly Zahl → Poly Zahl
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
```

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

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:

```
divmp :: Zahl → Poly Zahl → Poly Zahl → (Poly Zahl, Poly Zahl)
divmp p (P as) (P bs) = let (q, r) = go [0] 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  = modiv p (last r) (last bs)
                     d  = degree (P r) − db
                     ts = zeros d ++ [t]
                     m  = mulmlist p ts bs
                 in go [c ‘mmod‘ p | c ← cleanz $ strichlist (+) q ts]
                       [c ‘mmod‘ p | c ← cleanz $ strichlist (−) r m]
```

GCD:

```
gcdmp :: Zahl → Poly Zahl → Poly Zahl → Poly Zahl
gcdmp p a b | degree b > degree a = gcdmp p b a
            | zerop b = a
            | otherwise = let (_, r) = divmp p a b in gcdmp p b r
```

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:

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:

gcdmp 7 (P [1, 5, 3, 3, 5, 1]) (P [1, 4, 1, 1, 3]).

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

Finally, power:

```
powmp :: Zahl → Zahl → Poly Zahl → Poly Zahl
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* ($\lambda x \to$ *powmp* 7 x (P [1, 1]) [1 . . 14]:

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

It is especially interesting to look at greater powers using exponents that are multiples of 7. 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 ⇒ Poly a → a → a
apply (P cs) x = sum [c ∗ x ↑ i | (i, c) ← 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 ⇒ Poly a → [a] → [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 $x + 1$, 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 *mappy* (P $[1, 2, 3, 4]$) $[1\,..\,10]$, for instance, does not resemble such a simple sequence: 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 *mappy* (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 we recognise 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 19th 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 single numbers:

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

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

$$\begin{array}{l} \mathit{diffs} :: [\mathit{Zahl}] \rightarrow [\mathit{Zahl}] \\ \mathit{diffs}\ [\,] \qquad\quad = [\,] \\ \mathit{diffs}\ [\_] \qquad\ \ = [\,] \\ \mathit{diffs}\ (a : b : cs) = (b - a) : \mathit{diffs}\ (b : cs) \end{array}$$

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 sequence. It yields:

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

And once again:

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

And one more time:

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 machine, which takes differences, until it reaches a constant sequence.

```
dengine :: [Zahl] → [[Zahl]]
dengine cs | constant cs = [ ]
           | otherwise = ds : dengine ds
  where ds = diffs cs
        constant [ ]      = True
        constant [_]      = True
        constant (x : xs) = all (≡ x) xs
```

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:

length (*dengine* (*mapply* (P [0, 1]) [0 . . 32])): 1
length (*dengine* (*mapply* (P [0, 0, 1]) [0 . . 32])): 2
length (*dengine* (*mapply* (P [0, 0, 0, 1]) [0 . . 32])): 3
length (*dengine* (*mapply* (P [0, 0, 0, 0, 1]) [0 . . 32])): 4
length (*dengine* (*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 this sequence. Here is a function that does that for us:

```
predict :: [[Zahl]] → [Zahl] → Maybe Zahl
predict ds xs = case go (reverse ds) of
                  0 → Nothing
                  d → Just (d + (last xs))
  where go [] = 0
        go (a : cs) = last a + go cs
```

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

Let us go back to the question of how to find the polynomial given the sequence that this polynomial generates. 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" an 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 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:

```
genCoeff :: Zahl → Zahl → Zahl → [Zahl]
genCoeff d n x = go 0 x
  where go i x | i > d     = [x]
               | otherwise = n ↑ i : go (i + 1) x
```

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. The local function *go* repeats from 0 to d, raising n, the input value, to the current weight and adding it to the resulting list. At the end, when we have reached a value greater than the degree, we add the value from the sequence as known constant 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 first column, the squares in the third, the cubes in the fourth and the powers in the fifth and sixth column. This are just the values for x^i, for $i \in \{0 \dots 4\}$. 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 use *genCoeff* to create a matrix representing the entire system of equations:

```
findCoeffs :: [[Zahl]] → [Zahl] → L.Matrix Zahl
findCoeffs ds seq = L.M (go 0 seq)
  where d = fromIntegral (length ds)
        go _ [] = []
        go n (x : xs) | n > d      = []
                      | otherwise = genCoeff d n x : go (n + 1) xs
```

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:

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 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** $\mathit{ds}' = \mathit{tail}\ \mathit{ds}$ **in** *findGen* ($\mathit{tail}\ \mathit{ds}'$) ($\mathit{head}\ \mathit{ds}'$) 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** $\mathit{ds}'' = \mathit{tail}\ (\mathit{tail}\ \mathit{ds})$ **in** *findGen* ($\mathit{tail}\ \mathit{ds}''$) ($\mathit{head}\ \mathit{ds}''$) 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 chapter; 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

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

the derivative is

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

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. 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) ⇒ Poly a → Poly a
derivative (P as) = P (cleanz (go $ zip [1 ..] (drop 1 as)))
  where go []            = []
        go ((x, c) : cs) = (x ∗ c) : go cs
```

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* $p'\ [0 \mathinner{..} 10]$ 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 as **let** $p'' =$ *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** $p''' =$ *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

$$2, 2, 2, 2, 2, 2, 2, 2, 2, \dots$$

1.4 Differences and Binomial Coefficients

The ingenious 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, in the first line, the value of n, *i.e.* the value to which the polynomial is applied; in the second line, we see the result for this n; in the first column we have the first value from the sequence and from the 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:

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

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 is left-to-right:

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

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:

```
newton :: Zahl → [[Zahl]] → [Zahl] → Zahl
newton n ds seq = sum ts
  where hs = getHeads seq ds
        ts = [h ∗ (choose n k) | (h, k) ← zip hs [0 . . n]]
getHeads :: [Zahl] → [[Zahl]] → [Zahl]
getHeads seq ds = map head (seq : ds)
```

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 = *mapply* P [14, 9, 11, 16, 12] [0 . . 10] and d = *dengine* s. Now we perform some tests:

newton 0 d s gives 14.
newton 1 d s gives 62.
newton 5 d s gives 9834.
newton 11 d s gives 198432.
newton 13 d s gives 379874.

The approach, hence, 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 it:

```
bin2poly :: Zahl → Zahl → Poly Quoz
bin2poly h 0   = P [h % 1]
bin2poly h 1   = P [0, h % 1]
bin2poly h k   = P [h % (B.fac k)] `mul` go (k % 1)
  where go 1 = P [0, 1]
        go i = P [−(i − 1), 1] `mul` (go (i − 1))
```

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 binomials: $\binom{n}{0} = 1$. We, hence, return a constant polynomial consisting of the factor. This corresponds to $h_0 \times \binom{n}{0}$. The result is just h. 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:

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

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:

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

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.

The model consists of a data type:

```
data Newton = H | X | Y | Z
  deriving (Show, Eq)
```

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 $H = 0$:

```
cn :: Newton → Natural → [Newton]
cn H 0 = [H]
cn H n = cn H (n − 1) ++ cn X (n − 1)
```

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

```
cn X 0 = [X]
cn X n = cn X (n − 1) ++ cn Y (n − 1)
```

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

```
cn Y 0 = [Y]
cn Y n = Z : cn Y (n − 1)
```

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
$cn\ H\ 0 + cn\ X\ 0$ which is
$[H] + [X]$.

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
$cn\ H\ 1 + cn\ X\ 1$, which is
$cn\ H\ 0 + cn\ X\ 0$ and $cn\ X\ 0 + cn\ Y\ 0$.

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. When we generate a random polynomial of degree 3, say, $P\ [2, 28, 15, 22]$, this is the polynomial $22x^3 + 15x^2 + 28x + 2$, we get 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:

```
new2a :: (a, a, a, a) → Newton → a
new2a (h, x, y, z) n = case n of
  H → h
  X → x
  Y → y
  Z → z
subst :: (a, a, a, a) → [Newton] → [a]
subst as = map (new2a as)
```

The head of the sequence is 2; the head 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 with 0).

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

```
ccn :: [Newton] → (Int, Int, Int, Int)
ccn ls = (length (filter (≡ H) ls),
          length (filter (≡ X) ls),
          length (filter (≡ Y) ls),
          length (filter (≡ Z) ls))
```

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

$(1, 3, 3, 1)$

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

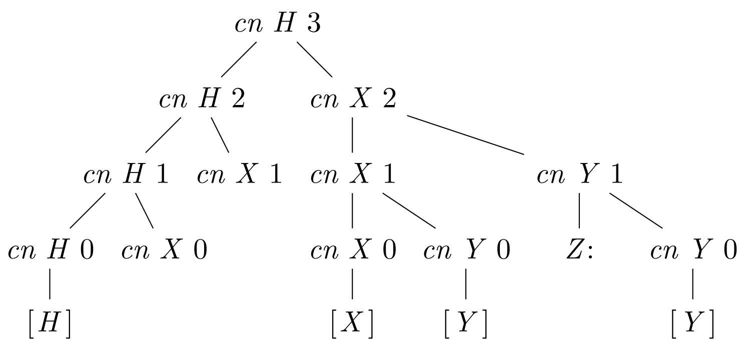
To see some more examples we call *map* (*ccn* $\circ$ *cn* H) $[4 \mathinner{..} 10]$ and get

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

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 $cn\ X\ 0 + cn\ Y\ 0$; 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 $cn\ H\ (n-1)$, $cn\ H\ (n-2)$, $\ldots$, $cn\ H\ 0$ 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. □

1.5 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:

$$a_n x^n + a_{n-1} x^{n-1} + \cdots + a_0 = a \tag{1.14}$$

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

$$a_n x^n + a_{n-1} x^{n-1} + \cdots + a_0 = 0. \tag{1.15}$$

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 negative coefficients that, for smaller numbers, result in numbers that are greater than those resulting from the term of highest degree. The polynomial $x^2 - 4$, for instance, 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 $x^3 + 9$ has its root at -3, while $x^3 - 9$ has its 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 set 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 fulful 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 regularily is that we only use basic arithmetic functions in their definition: we add, multiply and raise to powers. Those operations together with the integers form monoids and, with rational and real numbers, they form groups. Both, monoids and groups, are closed over their base set. We can therefore be sure that, for any input value from the base set, the result is in the same base set, too.

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 exponents, *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 also polynomials with a quite confusing behaviour that make it hard to guess the roots, for instance, Wilkinson's polynomial named for James Hardy Wilkinson (1919 – 1986), an American mathematician and computer scientist. The Wilkinson polynomial is defined as

$$w(x) = \prod_{i=1}^{20} (x - i). \tag{1.16}$$

We can generate it in terms of our polynomial type as

> *wilkinson* :: (*Num a*, *Enum a*, *Show a*, *Eq a*) ⇒ *Poly a*
> *wilkinson* = *prodp mul* [*P* [−*i*, 1] | *i* ← [1 . . 20]]

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 $1 \ldots 10$, 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 . . .
apply wilkinson 1.1 is −8.4600 . . .
apply wilkinson 1.9 is −8.1111 . . .
apply wilkinson 2.1 is 4.9238 . . .

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

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

The results when applying the polynomial, however, look quite irregular and, on the first sight, completely unrelated to the coefficients. 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. We then choose two values that limit this interval on the lower and on the upper side.

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:

```
bisect :: (Num a, Eq a, Ord a, Fractional a, Show a)
          ⇒ Poly a → a → a → a → a
bisect p t a b  | abs fc < abs t           = c
                | signum fc ≡ signum fa = bisect p t c b
                | otherwise                = bisect p t a c
  where fa = apply p a
        fb = apply p b
        fc = apply p c
        c  = (a + b) / 2
```

The function receives four arguments. The first is the polynomial. The second is a tolerance. When we reach a result that is smaller than the tolerance, we return the result. 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:

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

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 inprecision.

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 this parameters returns almost instantanious. 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.

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, by Joseph Raphson in 1690. 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:

```
newguess :: (Num a, Eq a, Ord a, Enum a, Fractional a)
            ⇒ Poly a → Natural → a → a → a
newguess p m t a | abs pa < t = a
                 | m ⩽ 0      = a
                 | otherwise  = newguess p (m − 1) t (a − pa / p′a)
  where p′  = derivative (∗) p
        pa  = apply p a
        p′a = apply p′ a
```

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 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 with $a - \frac{\pi(a)}{\pi'(a)}$.

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

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

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

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

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

newguess wilkinson 10 (0.0001) 1.5

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

newguess 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 that the usual representation with coefficients. Wilkinson's polynomial expressed as a product was just

$$w(x) = \prod_{i=1}^{20} (x - i), \tag{1.17}$$

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, because it makes the product and, as such, the whole polynomial zero. Any integer in the range $[1\dots20]$ would make one of the factors of Wilkinson's polynomial zero. The integers $[1\dots20]$ are therefore the roots of Wilkinson's polynomial.

Factoring polynomials, however, is an advanced problem in its own right and we will dedicate the next section to its study. Anyway, what algebraists did for centuries was to find 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 not such formulas. This dicovery 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 higher degrees, we need to understand polynomials much better. First, we will look at the formula to solve polynomials of the second degree. Note that we skip the first degree, since finding the roots of polynomials of the first degree is just solving linear equations.

1.6 Vieta's Formula

1.7 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 either a *unit* or 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 $p = qr$ can be 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 are $\pm 2, \pm 3, \pm 5, \pm 7, \dots$. We can easily verify that 1 and -1 obey to 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 $\frac{1}{6}$:

$$\begin{array}{cccccc} \frac{1}{3} & \times & 6 & = & 2 \\ \frac{7}{2} & \times & 6 & = & 21 \\ 2 & \times & 6 & = & 12 \\ 1 & \times & 6 & = & 6 \end{array}$$

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

$$x^2 - 9 = (x - 3)(x + 3). \tag{1.18}$$

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 polynomial 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 two.

The rationale of applying the polynomial is the following: When the polynomial we want to factor generates a given 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:

```
divs :: Zahl → [Zahl]
divs i | i < 0       = divs (−i)
       | otherwise = ds ++ map negate ds
  where ds = [d | d ← [1 .. i], rem i d ≡ 0]
```

The divisors are now combined to yield n-tuples, n, still the degree of the factor plus one, where each divisor represents 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):

```
kronecker :: Poly Zahl → [Zahl] → [Poly Quoz]
kronecker (P cs) is = nub [a | a ← as, snd (r ‘divp‘ a) ≡ P [0]]
  where ds = map divs is
        ps = concatMap perms (listcombine ds)
        as = map (P ∘ map fromInteger) ps
        r  = P [c % 1 | c ← cs]
```

Note that, since we use *divp*, we need to convert the integer polynomial to a rational polynomial.

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:

```
listcombine :: [[a]] → [[a]]
listcombine []       = []
listcombine ([]: _)  = []
listcombine (x : xs) = inshead (head x) (listcombine xs) ++
                         listcombine ((tail x) : xs)
inshead :: a → [[a]] → [[a]]
inshead x [] = [[x]]
inshead x zs = map (x:) zs
```

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 coefficients by factoring out -1 first. However, we can easily see that all of them are actually factors of $x^2 - 9$, since

$$(x + 3)(x - 3) = (x^2 - 9) \tag{1.19}$$

and

$$(-x + 3)(-x - 3) = (x^2 - 9). \tag{1.20}$$

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:

$$(x^2 + x + 1)(x^3 - x + 2) = x^5 + x^4 + x^2 + x + 2. \tag{1.21}$$

Kronecker's method is just a brute force search. It is obvious that it is not efficient and it fails with growing degrees and coefficients. Modern methods to factor polynomials use much more sophisticated techniques.

They are, in particular, based on modular arithmetic of polynomials 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.

1.8 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 methematician, 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

$$a^{\varphi(n)} \equiv 1 \pmod{n}, \tag{1.22}$$

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 an example thereof.

However, we need to redefine Euler's theorem to make clear what is meant by it in the new context. First, we are dealing with the polynomial field $K[x]$ and a polynomial $m \in K[x]$. Then we define the totient function as

$$\varphi(m) = |\{f \in K[x] : 0 \leq f \leq m \wedge \gcd(m, f) = 1\}|,$$

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 field and any $f \in K[x] : \gcd(m, f) = 1$, the following holds:

$$f^{\varphi(m)} \equiv 1 \pmod{m}. \tag{1.23}$$

This is easy to prove. 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. From $gcd(m, f) = 1$, it follows that $f_m = f \mod m \in K^*_m$. We, for sure, have $f_m^{\varphi(m)} \equiv 1 \pmod{m}$, since $\varphi(m)$ is the size of the group. This 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, and we unmistakenly have $f^{\varphi(m)} \equiv 1 \pmod{m}$ $\square$.

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 $1 \dots p - 1$, which contains only $p - 1$ elements. 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

$$f^{q^d} \equiv f \pmod{g}. \tag{1.24}$$

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 this polynomial is zero for which division is not defined. For the irreducible polynomial g, we therefore have $r-1$ elements that do not share divisors with g, *i.e.* $\varphi(g) = r-1$. So, according to Euler's theorem, we have

$$f^{r-1} \equiv 1 \pmod{g}. \tag{1.25}$$

Multiplying both sides by f, we get

$$f^{r} \equiv f \pmod{g}. \tag{1.26}$$

Since $r = q^d$, this is equivalent to 1.24 and this concludes the proof. $\square$

From Fermat's theorem, we can derive a nice and useful corollary. Note that when we subtract f from both sides of the equivalence, then 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:

$$x^{q^d} - x \equiv 0 \pmod{g}. \tag{1.27}$$

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:

```
irreducible :: Natural → Poly Natural → Bool
irreducible p u  | d < 2       = False
                 | otherwise = go 1 x
  where d        = degree u
        x        = P [0,1]
        go i z = let z′ = powmp p p z
                 in case pmmod p (addp p z′ (P [0,p − 1])) u of
                    P [_]  → i ≡ d
                    g      → if i < d then go (i + 1) (pmmod p z′ u)
                                      else False
```

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 a constant polynomial and the degree counter i equals d, then equation 1.27 is fulfilled. (Note that we consider any constant polynomial as zero, since a constant polynomial is just the content, which usually should have been removed before we start to search factors.) 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 constant, 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 *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:

$g \leftarrow$ *randomPoly* 7 4

I get the polynomial $P\ [3,3,3,4]$. (Note that you may get another one!) Calling *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 certainly not constant. g is therefore not irreducible.

Let us try another one:

$g \leftarrow$ *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 $P\ [0]$. $P\ [3,1,4,4]$, hence, is irreducible.

The formula, however, is not only interesting for testing irreducibility. What the formula states is in fact that all irreducible polynomials of degree d are factors of $x^{q^d} - x$. Whenever we construct this expression, we have created the product of all irreducible polynomials of degree d. The irreducible factors of our polynomial are part of this product and we can get them out just by asking for the greatest common divisor. This would give us the product of all factors of our polynomial of a given degree.

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 :: Integer → Poly Integer → Poly Integer
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.

```
ddfac :: Natural → Poly Natural → [(Int, Poly Natural)]
ddfac p u   = go 1 u (P [0,1])
  where n = degree u
        go d v x | degree v ⩽ 0 = [ ]
                 | otherwise    =
                   let x'      = powmp p p x
                       t       = addp p x' (P [0, p − 1])
                       g       = gcdmp p t v
                       (v', _) = divmp p v g
                       r       = (d, monicp p g)
                   in case g of
                       P [_] →     go (d + 1) v' (pmmod p x' u)
                       _      → r : go (d + 1) v' (pmmod p x' u)
```

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 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 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

$$t^{p^d} - t = t(t^{(p^d-1)/2} + 1)(t^{(p^d-1)/2} - 1). \tag{1.28}$$

Make sure that the equality holds by multiplying the right-hand side out. 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:

$$u = \gcd(u,t) \times \gcd(u, (t^{(p^d-1)/2} + 1)) \times \gcd(u, ((t^{(p^d-1)/2} - 1)) \tag{1.29}$$

A reasonable choice for t is a polynomial of degree $2d-1$, since in this case there is high probability that the factors are spread equally among the three factors of equality 1.29. With high probability, we reduce the problem significantly, when 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 polynomial. 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, to illustrate that, 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

$$w^2 + w = t^2 + t^4 + \dots + t^{2^d} + t + t^2 + \dots + t^{2^{d-1}},$$

when we rearrange according to exponents, we again get pairs of equal terms:

$$w^2 + w = t + 2t^2 + 2t^4 + \dots + 2t^{2^{d-1}} + t^{2^d}.$$

When we compute this modulo 2, again all terms with even coefficients fall away and we finally get

$$w^2 + w = t^{2^d} + t. \tag{1.30}$$

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 the implementation of the Cantor-Zassenhaus algorithm:

```
cz :: Natural → Int → Poly Natural → IO [Poly Natural]
cz p d u | n ⩽ d     = return [monicp p u]
         | otherwise = do
  x ← monicp p <$> randomPoly p (2 ∗ d)
  let t    | p ≡ 2     = addsquares (d − 1) p x u
           | otherwise = addp p (powmodp p m x u) (P [p − 1])
  let r = gcdmp p u t
  if degree r ≡ 0 ∨ degree r ≡ n then cz p d u
    else do r1 ← cz p d r
            r2 ← cz p d (fst $ divmp p u r)
            return (r1 ++ r2)
  where n = degree u
        m = (p ↑ d − 1) ‘div‘ 2
```

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, 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 1.29. We compute the gcd and, if the result has either degree 0 (no factor was found) or degree equal to u (all factors are in this one), we just try again with another random polynomial. Otherwise, we continue with the gcd and the quotient $u/\gcd$.

Let us try this for the result $(1, P\ [2,4,1])$ we obtained earlier from applying *ddfac* on $P\ [5,4,3,1,1]$. $cz\ 7\ 1\ (P\ [2,4,1])$ gives

$[P\ [6,1], P\ [5,1]]$

two irreducible polynomials of degree 1. The complete factorisation of $P\ [5,4,3,1,1]$, hence, is

$[P\ [6,1], P\ [5,1], P\ [6,4,1]]$,

which we can test by calling $prodp\ (mulmp\ 7)\ [P\ [6,1], P\ [5,1], P\ [6,4,1]]$ and we, indeed, get $P\ [5,4,3,1,1]$ back.

For the case where $p = 2$, we use the function addsquares:

```
addsquares :: Int → Natural → Poly Natural → Poly Natural → Poly Natural
addsquares i p x u = go i x x
  where go 0 w _ = w
        go k w t = let t'  = pmmod p (powmp p p t) u
                       w'  = addp p w t'
                   in go (k − 1) w' t'
```

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 irreducible (and, per definition, monic). The call $ddfac\ 2\ (P\ [0,1,1,1,0,0,1,1,1])$ 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 already have the specified degree. The first one, however, is of degree 2, but shall contain factors of degree 1. So, let us see what $cz\ 2\ 1\ (P\ [0,1,1])$ will yield:

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

The complete factorisation of $P\ [0,1,1,1,0,0,1,1,1]$, hence, is

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

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. The point is that a polynomial and its derivative share only those divisors that appear more than once in the factorisation. We have not enough knowledge on derivatives to prove that here rigorously, but we can give an intuition.

Consider a polynomial with the factorisation

$$(x + a)(x + b) \dots$$

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

$$(f \times g)' = fg' + f'g, \tag{1.31}$$

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 $(x+a)(x+b)$ 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:

$$1 \times (x + a) + 1 \times (x + b) = (x + a) + (x + b).$$

Let us compute the polynomial with the factors $(x + a)(x + b)$. The polynomial is $x^2 + ax + bx + ab$. Its derivative is $2x + a + b$. When we apply the product rule to the factors, we get $(x + a) + (x + b) = x + a + x + b = 2x + a + b$, 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 and, even further, that none of the factors is preserved. They all disappear in favour of others they do not share divisors with, because, since the factors are coprime to each other, they do not share divisors with their sum either.

To elaborate on this, consider now polynomials with more than two factors of the form

$$abc \dots,$$

where a, b and c stand for irreducible polynomials like $(x + \alpha)$, $(x + \beta)$ 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. This can be represented nicely as:

$$\left(\prod_{i=0}^{k} a_i\right)' = \sum_{i=0}^{k}\left(a_i' \prod_{j\neq i} a_j\right) \tag{1.32}$$

There is a remarkable 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 irreducible factors (and, hence, 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 a multiple of the original factor, which, in its turn, is therefore preserved.

Suppose we want to compute the factorisation of

$$f = a_1 a_2^2 a_3^3 \dots a_k^k, \tag{1.33}$$

where the as represent the products of all the factors raised to the indicated exponent, then, since the derivative preserves the factors, the gcd of f and its derivative f' (whatever that looks like) is:

$$\gcd(f, f') = a_2^1 a_3^2 \dots a_k^{k-1}, \tag{1.34}$$

i.e. the repeated factors with the exponent decreased by one. Then f divided by the gcd gives us

$$\frac{f}{\gcd(f, f')} = a_1 a_2 a_3 \dots a_k, \tag{1.35}$$

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 1.34 reduced one more ($a_3 a_4^2 \dots$) and 1.35 with the head chopped off ($a_2 a_3 \dots$). The quotient of the two versions of 1.35, *i.e.*

$$\frac{a_1 a_2 a_3 \dots}{a_2 a_3 \dots},$$

would give us the head. The head, however, is the product of all factors with a given exponent.

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. As in the algorithm for infinte fields, we would 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.*

$$V_{k+1} = \begin{cases} \gcd(T_k, V_k) & \textit{if } p \nmid k \text{ (as before)} \\ V_k & \textit{if } p \mid k \end{cases}$$

At each step, we have

$$V_k = \prod_{i \geq k, p \nmid i} a_i, \tag{1.36}$$

i.e., the product of all as with exponents greater than those that we have already processed and that do not divide p, and

$$T_k = \prod_{i \geq k, p \nmid i} a_i^{i-k} \prod_{i \geq k, p \mid i} a_i^i, \tag{1.37}$$

i.e., the product of the powers greater than those we have already processed for both cases $p \mid k$ and $p \nmid k$. For the cases $p \nmid k$, everything is as before. For the cases $p \nmid k$, 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 multiples of p.

To get as out, we divide all 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:

```
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} 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 $e = 0$:

```
squarefactormod :: Integer → Poly Integer → [(Integer, Poly Integer)]
squarefactormod p = sqmp p 0
```

Finally, we are ready to put everything together:

```
cantorzassenhaus :: Integer → Poly Integer → IO [(Integer, Poly Integer)]
cantorzassenhaus p u | irreducible p m = return [(1, m)]
                     | otherwise       =
                       concat < $ > mapM mexpcz [(e, ddfac p f) |
                                          (e, f) ← squarefactormod p m]
  where m = monicp p u
        expcz e (d, v)    = map (λf → (e, f)) < $ > cz p d v
        mexpcz (e, dds) = concat < $ > mapM (expcz e) dds
```

Hans Zassenhaus worked most of his life as a computeralgeabrist and was important for the development of 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. Unfortunately, the societal engagement of the Zassenhaus siblings is hardly remembered today.

1.9 The Method of Partial Fractions

1.10 The closed Form of the Fibonacci Sequence

$$G(x) = F_0 + F_1x + F_2x^2 + F_3x^3 + \dots \tag{1.38}$$

$$G(x) = 0 + x + x^2 + 2x^3 + 3x^4 + 5x^5 + 8x^6 + \dots \tag{1.39}$$

$$xG(x) = F_0x + F_1x^2 + F_2x^3 + F_3x^4 + \dots \tag{1.40}$$

$$x^2G(x) = F_0x^2 + F_1x^3 + F_2x^4 + F_3x^5 + \dots \tag{1.41}$$

$$G(x) - xG(x) - x^2G(x) = (1 - x - x^2)G(x). \tag{1.42}$$

$$\begin{array}{llllllll}
(1 - x - x^2)G(x) = & (F_0 & +F_1x & +F_2x^2 & +F_3x^3 & +\dots) & - \\
 & (& F_0x & +F_1x^2 & +F_2x^3 & +\dots) & - \\
 & (& & +F_0x^2 & +F_1x^3 & +\dots) &
\end{array}$$

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

$$(1 - x - x^2)G(x) = x. \tag{1.43}$$

$$G(x) = \frac{x}{1 - x - x^2}. \tag{1.44}$$