Worksheet 7

Version 2015-09-14

7 Polynomials und Test Driven Development

7.1 Polynomials

The following session log shows how to create a polynomial ring in GAP, as well as some polynomials.

```
gap> R := PolynomialRing(Rationals, "t");
Rationals[t]
gap> IsRing(R);
true
gap> t := R.1;
t
gap> f := 3*t^4 + 5*t^3+t+1; g := 2*t^2+5;
3*t^4+5*t^3+t+1
2*t^2+5
gap> f * g;
6*t^6+10*t^5+15*t^4+27*t^3+2*t^2+5*t+5
gap> QuotientRemainder(f, g);
[ 3/2*t^2+5/2*t-15/4, -23/2*t+79/4 ]
gap> f / g;    # GAP also knows aboutwith rational functions...
(3*t^4+5*t^3+t+1)/(2*t^2+5)
```

You can also create polynomials directly:

```
gap> x := Indeterminate(Rationals, "x");
x
gap> f := x^4 + 3 * x - 1;
x^4+3*x-1
```

The above are all examples of a polynomial ring in one “variable” or indeterminate, also called univariate polynomial rings. You can also create polynomials in multiple indeterminates, which are called multivariate polynomials.

```
gap> x := Indeterminate(Rationals, "x");;
gap> y := Indeterminate(Rationals, "y");;
gap> x = y;
false
gap> IsUnivariatePolynomial(x^2 + 1);
true
gap> IsUnivariatePolynomial( (x+y)^2 );
false
gap> x = IndeterminateOfLaurentPolynomial(x^2 + 1);
true
```

To compute the degree of a univariate polynomial, you can use `Degree`. For multivariate polynomials, you need to tell GAP with regards to which indeterminate you want to know the degree.

```
gap> Degree(x^2);
2
gap> f := x^3 + x*y + y^2;; DegreeIndeterminate(f, x); DegreeIndeterminate(f, y);
3
2
```

When working with polynomials, it is often useful to access their coefficients.

```
gap> f := 12 * x^3 - 2*x + 7;;
gap> LeadingCoefficient(f);
12
gap> coeffs := CoefficientsOfUnivariatePolynomial(f);
[ 7, -2, 0, 12 ]
gap> # The reverse is, unfortunately, rather tedious
gap> ind := IndeterminateNumberOfUnivariateRationalFunction(f);;
gap> fam := CoefficientsFamily(FamilyObj(f));;
gap> g := UnivariatePolynomialByCoefficients(fam, coeffs, ind);
gap> f = g;
true
```

Polynomial rings with coefficients in a field are vector spaces, so methods for vector spaces apply to them.

```
gap> R := PolynomialRing(GF(5), ["u", "v"]);;
gap> IsVectorSpace(R);
true
gap> CoefficientsRing(R);
GF(5)
gap> R.1; R.2;
u
v
gap> Dimension(R);
infinity
gap> U := Subspace(R, [R.1^0, R.1, R.1^2]);
<vector space over GF(3), with 3 generators>
gap> Dimension(U);
3
gap> Basis(U);
Basis( <vector space of dimension 3 over GF(5)>, [ Z(5)^0, u, u^2 ] )
```

You can also evaluate polynomials, and much more – consult the GAP manual for further information.

```
gap> Value(x^3 - 1, 2);
7
gap> Value(x^2 + y^2, [x], [3]);
y^2+9
gap> Value(x^2 + y^2, [x,y], [3,4]);
25
```

7.2 Tests

When writing programs, it is usually necessary to verify that it adheres to a given specification (this does not just apply to the homework in this course, but also when you program as part of a job). Ensuring that is a tricky process.

Moreover, whenever one makes a change to an existing program, there is a risk of breaking something that previously worked. Most of you have experienced this already.

Here are some hints on how to systematically ensure that your program works correctly.

First, of course you need to read the program specification very carefully. For example, if an input variable n is specified as being an integer, then it can be more than just a positive number – it could also be zero or negative.

Secondly, try to write so-called tests for your program, which check that it actually complies to the specification. That is, instead of manually running your program with different inputs, you automate this, by writing a second program which tests the first. The idea being that this second program will be quite a bit simpler, will change less often, and does something that computers are perfect for and humans are very bad at: It performs some boring tests again and again, unfailingly and always complete.

For example, recall the function `binarySearch(l,x)`. A simple set of tests for it could look like this:

```
l := [1,17,33];
for i in [1..Length(l)] do
    pos := binarySearch(l, l[i]);
    if pos <> i then Error("wrong"); fi;
od;
```

Now, whenever you change your program, you can easily re-run these tests. And if you broke something, you have at least a chance of detecting this.

So suppose you submitted your program, but got a reply that told you something in your program does not yet work. You are told that it fails for the empty list. The first thing you do is to add a new test case to your test program which tests this. For example:

```
pos := binarySearch([], 1);
if pos <> fail then Error("wrong"); fi;
```

Then, you modify your binary search program and re-run the test program until all tests pass. While doing this, you noticed that lists with just also one element also sometimes triggered a problem. You add a few test cases for this, too, then change your program again until all tests pass.

```
pos := binarySearch([1], 1);
if pos <> 1 then Error("wrong"); fi;
pos := binarySearch([1], 2);
if pos <> fail then Error("wrong"); fi;
```

This process of first adding a test for a potentially or actually problematic case, then modifying your program until it passes all tests, is known as test driven development, or TDD. It can be a bit tedious at first, but on the long run and for somewhat complicated programs, it pays of greatly. I recommend experimenting with this a bit.

Back to our tests. The last two (for empty lists and lists of length 1) are examples of edge cases, i.e. cases where some parameters take on some extremal values which are unusual, and hence often forgotten by implementors. It is usually a good idea to try to discover such edge cases right from the start, and adding tests cases for them. However, do *not* immediately start adding code for them. E.g. in the binary search example, a proper implementation actually covers all edge cases automatically. Trying to handle them all explicitly just makes your program longer, and thus increases the chance of it being buggy.

Back to our test program (also called a test suite). After adding a bunch of tests, the fact that all errors lead to the same error message becomes annoying, as you have to do extra work to figure out which tests failed. We could change our test cases to look like this:

```
pos := binarySearch([1], 1);
if pos <> 1 then Error("wrong result ",pos," for input l=[1] and x=1"); fi;
```

But this means a lot of extra typing work, and also introduces the risk of the test performed differing from the error message... I.e. we risk a bug in our test code. But this is all about *avoiding* bugs. In this case, this turns out to be easy: Just automate the message generation, too:

```
testBinarySearch := function(l, x)
    local pos;
    pos := binarySearch(l, x);
    if pos = fail then
        if x in l then
            Error("Input l=",l,"; x=",x," got fail, but x in l");
        fi;
    elif IsInt(pos) then
        if l[pos] <> x then
            Error("Input l=",l,"; x=",x," got wrong answer ", pos);
        fi;
    else
        Error("Input l=",l,"; x=",x," got invalid answer ", pos);
    fi;
end;

l := [1,17,33];
testBinarySearch(l, 1);
testBinarySearch(l, 17);
testBinarySearch(l, 33);
testBinarySearch(l, 10); # check negative case where fail is returned
testBinarySearch(l, -1); # check negative case where element is before first entry
testBinarySearch(l, 50); # check negative case where element is after last entry
testBinarySearch([], 1); # empty list
testBinarySearch([1], 1); # list of length 1, match
testBinarySearch([1], 2); # list of length 1, no match
for i in [0..4] do
    testBinarySearch([1,2,2,3], i); # test with repetition in list
    testBinarySearch([1,2,2,2,2,2,2,2,2,2,2,3], i);
od;
```

This test suite is now quite easy to understand, and also easy to extend.

One last suggestion: All your tests, no matter how you do them, are useless if you by accident test the wrong code. For example, if you use copy & paste to transfer code from an editor into GAP, then sometimes this gets messed up, and as a result some characters are not transferred. The program then will work differently than expected; and it can even happen that GAP refuses to accept the garbled pasted code. If you don't notice this, it will keep using the previous version of the program, and you will wonder why your change seemingly did not have any effect.

To avoid this, always use `Read` to load your code, and double check that you are reading from the right path. Also, on occasion, exit your GAP sessions and start a new one. Then read your program in – this ensures that no "old" version of your code is used by accident, may help uncover syntax errors or missing `local` statements or the accidental use of global variables, all of which you may not have noticed before, after repeatedly reading and testing variants of your program.

7.3 Homework

(s7-h1) Let $f := \sum_{i=0}^{n} a_i t^i \in \mathbb{Z}[t]$ be a polynomial. Then $\mathrm{cont}(f) := \gcd(a_0, a_1, \ldots, a_n)$ is the content of f, and f is primitive if $\mathrm{cont}(f) = 1$.

Implement a function `cont(f)` which takes a univariate polynomial with integers coefficients, and returns its content. Return `fail` if f does not satisfy the input requirements.

Hint: You can use `IsInt` to test whether an object is an integer.

(s7-h2) Let $f := \sum_{i=0}^{n} a_i t^i \in \mathbb{Z}[t]$ be a polynomial. Then f is irreducible over $\mathbb{Q}$ if it is not constant and cannot be written as the product of two non-constant polynomials in $\mathbb{Q}[t]$. Eisenstein's criterion says: If the degree of f is $n \in \mathbb{N}$, and there is a prime p such that (i) p divides a_i for $0 \leq i < n$, (ii) p does not divide a_n, and (iii) p^2 does not divide a_0, then f is irreducible over $\mathbb{Q}$.

Implement a function `Eisenstein(f)` which takes a polynomial $f \in \mathbb{Z}[t]$ and returns `true` if f is irreducible over $\mathbb{Q}$ according to Eisenstein's criterion for any prime p, and `false` otherwise. (Note that f may still be irreducible.) Return `fail` if f does not satisfy the input requirements.

(s7-h3) Let $\mathbb{K}$ be a field. Implement a function `polyDivRem(f,g)` which takes two univariate polynomials f and g with coefficients in $\mathbb{K}$, with $g \neq 0$, and which returns a pair $[q, r]$ of univariate polynomials such that $f = qg + r$ and $\deg(r) < \deg(g)$. Return `fail` if f, g are not univariate polynomials or if g if zero.

Do not use the GAP functions for polynomial division (such as `QuotientRemainder`). You may use those for adding and multiplying polynomials, and also `IndeterminateOfLaurentPolynomial`.

(s7-h4*) Let R be a commutative ring and $f := \sum_{i=0}^{n} a_i t^i \in R[t]$ a polynomial. The derivative of f is $f' := \sum_{i=1}^{n} i a_i t^{i-1}$.

Implement a function `der(f)` which takes a univariate polynomial and returns its derivative. Return `fail` if f does not satisfy the input requirements.

Do not use the GAP function `Derivative`. You may use `IndeterminateOfLaurentPolynomial`.

(s7-h5*) Write a test program for `polyDivRem`. That is, first implement a function `testPolyDivRem(f,g)` which invokes `polyDivRem` and then verifies that its output is correct. Try to do this without using `QuotientRemainder` etc.

Then, write a set of test cases that covers typical cases, but also edge cases. What edge cases can you think of?

Hint: Consider following the test driven development approach described above, and solving this exercise concurrently with implementing `polyDivRem`.