JUSTUS-LIEBIG-UNIVERSITÄT GIESSEN

Mathematisches Institut, AG Algebra
Prof. Dr. Max Horn
Rechenkurs „Algebra mit GAP“

Worksheet 3

Version 2014-12-03

3 Higher-order functions

In GAP, functions are objects just like integers or lists. On this worksheet, we will discuss higher-order functions, i.e. functions that take other functions as arguments.

3.1 Applying a function to a list, in-place

A common programming task is to iterate over a list of things and apply a transformation to each list element. E.g. on might apply the map $x \mapsto 2x^2$ to each element of a list of integers. Here is one way to do that:

```
apply_2xx := function(list)
    local i;
    for i in [1..Length(list)] do
        list[i] := 2 * list[i]^2;
    od;
end;
```

```
gap> l := [17, 0, 8, -4, -99];;
gap> apply_2xx(l);
gap> l;
[ 578, 0, 128, 32, 19602 ]
```

Suppose we instead want to replace each entry by its absolute value. We might write:

```
apply_abs := function(list)
    local i;
    for i in [1..Length(list)] do
        list[i] := AbsInt(list[i]); # AbsInt is a handy function provided by GAP
    od;
end;
```

```
gap> l := [17, 0, 8, -4, -99];; apply_abs(l); l;
[ 17, 0, 8, 4, 99 ]
```

The two functions look very similar, which is no surprise, as they do something similar. Instead of re-writing this pattern again and again, we can write a higher-order function `apply` which takes a list and a unary function f, and then applies f to each entry of the list:

```
apply := function(list, func)
    local i;
    for i in [1..Length(list)] do
        list[i] := func(list[i]);
    od;
end;
```

```
gap> l := [17, 0, 8, -4, -99];; apply(l, x -> 2*x^2); l;
[ 578, 0, 128, 32, 19602 ]
gap> l := [17, 0, 8, -4, -99];; apply(l, AbsInt); l;
[ 17, 0, 8, 4, 99 ]
```

This is a quite useful programming technique. In fact it is so useful that GAP already includes a function doing just this under the name `Apply`.

Note that there is nothing magical about the `x -> 2*x^2` in the preceding example. This is just a shorthand notation for writing simple unary functions. We could also have written this (now using `Apply`):

```
gap> l := [17, 0, 8, -4, -99];; Apply(l, function(x) return 2*x^2; end); l;
[ 578, 0, 128, 32, 19602 ]
gap> # or even this:
gap> f := function(x) return 2*x^2; end;
gap> l := [17, 0, 8, -4, -99];; Apply(l, f); l;
[ 578, 0, 128, 32, 19602 ]
```

3.2 Mapping a list to a new list

An important variation of this theme is when you want to put the results into a new list, without modifying the original list. The following function does just that:

```
map := function(list, func)
    local x, result;
    result := [];
    for x in list do
        Add(result, func(x));
    od;
    return result;
end;
```

```
gap> l := [17, 0, 8, -4, -99];;
gap> map(l, AbsInt);
[ 17, 0, 8, 4, 99 ]
gap> l;
[17, 0, 8, -4, -99]
```

This, too, is already provided by the GAP library, under the name `List` (in other programming languages, it is known under such names as `map`, `transform`, `zip`, `zipWith`, ...)

```
gap> List([17, 0, 8, -4, -99], x -> 2*x^2);
[ 578, 0, 128, 32, 19602 ]
```

Exercise 3.2.1

Compute the first 20 square numbers. Do the same for cubes and some higher powers.

Other related functions are `Set` and `Perform`, you can read more about them in the GAP manual.

Exercise 3.2.2

Write a function `multiApply` which takes a list of functions $[f_1, \dots, f_n]$, and a single value x, and returns the list $[f_1(x), \dots, f_n(x)]$.
Hint: You can use `List` to write a very short solution.

3.3 Folding

Another important pattern is *folding* or *accumulation*: Suppose we want to sum up a list of integers.

```
sum := function(list)
    local s, x;
    s := 0;
    for x in list do
        s := s + x;
    od;
    return s;
end;
```

```
gap> sum([ -12, 5, -72, -18, 92 ]);
-5
```

Exercise 3.3.1

Modify the function to compute a product instead of a sum.

The pattern here is that we take an initial value (zero in this case), put that into an accumulator, then successively combine the accumulator plus the next list element together using a binary operation.

We can extract this into a higher-order function, which takes a list, a binary function, and an initial value.

```
foldLeft := function(list, func, init)
    local acc, x;
    acc := init;
    for x in list do
        acc := func(acc, x);
    od;
    return acc;
end;
```

With this, we can now also compute sums and products of list elements:

```
gap> list := [ -12, 5, -72, -18, 92 ];;
gap> foldLeft(list, function(a,b) return a+b; end, 0);
-5
gap> # Shorter version by directly passing the GAP function for addition:
gap> foldLeft(list, \+, 0);
-5
gap> foldLeft(list, \*, 1); # Or we can compute the product
-7153920
```

We can re-define the function `sum` using `foldLeft` as follows:

```
gap> sum := x -> foldLeft(x, \+, 0);;
gap> sum(list);
-5
```

The fold or accumulation pattern applies to many problems, for example also to computing the maximum and minimum of a list.

```
gap> min := function(a,b) if a < b then return a; else return b; fi; end;;
gap> minList := l -> foldLeft(l, min, infinity);
gap> minList(list);
-72
```

GAP provides pre-made functions for several of these computations: `Sum`, `Product`, `Maximum`, `Minimum`. Strangely enough, there is no analogue to `foldLeft`.

Exercise 3.3.2

1. Using `foldLeft`, write a function `LastElm(l)` which returns the last element of a list `l`, or `fail` if the list is empty. Use neither `if` clauses nor `Length`. Try to do it in one line with less than 80 characters.
2. Using `foldLeft`, write a function `rev(l)` which, given a list `l`, returns a new list containing the elements of `l` in reverse order. Use neither `Reversed` nor any loops.
   Hint: Use on of the functions `Concatenation(l1,l2)` or `Add(list,elm,pos)` to modify the accumulator suitably. With these, you can solve the problem in one line with less than 80 characters.

Remark 3.3.3

There is also a `foldRight` which iterates over a list in reverse order, and also changes the order of the arguments to the binary function. This is important if the function is not associative.

```
foldRight := function(list, func, init)
    local acc, x;
    acc := init;
    for x in Reversed(list) do
        acc := func(x, acc);
    od;
    return acc;
end;
```

Exercise 3.3.4

Compare what the following two functions compute for various values of n (say, for $-3 \leq n \leq 3$. What is going on?

```
f1 := n -> foldLeft([1..n], \-, 0);
f2 := n -> foldRight([1..n], \-, 0);
```

3.4 Predicates

A predicate is a unary function which maps things to booleans (true or false). Some examples of predicates built into GAP include `IsInt`, `IsEvenInt`, `IsOddInt`, `IsPrimeInt` but we can easily create our own, e.g.

```
IsMultipleOf3 := x -> (x mod 3) = 0;
```

Suppose we want to know how many elements of a list of integers are divisible by 3, and which of them are primes. We could write a loop. Or we could do this:

```
gap> list := [1111, 17, 105, 1361, 99, 191];;
gap> Number(list, IsMultipleOf3);
2
gap> Filtered(list, IsPrimeInt);
[ 17, 1361, 191 ]
```

Exercise 3.4.1

Implement a function `count(l,f)` which does exactly what `Number` does, but without using it.
Try to do it once with a loop, and once with `foldLeft`.

Sometimes we just need to know whether a property holds for any or even all elements of a list.

```
gap> ForAll(list, IsPrimeInt);
false
gap> ForAny(list, IsPrimeInt);
true
```

Or we just want the first element satisfying the property.

```
gap> First(list, IsPrimeInt);
17
```

Exercise 3.4.2

Write a function `Last(l,f)` which returns the last element of a list `l` for which `f` returns true. If no such element exists, return `fail`.
Try writing two versions, one with a loop, and one with `foldLeft`.

Two additional property related functions are `PositionProperty` and `PositionsProperty`. The GAP help system can tell you more about them.

3.5 Closures

So far, we looked at functions which take functions as parameters. But we can also write functions which *return* functions.

```
addN := function(n)
    local f;
    f := function(x)
        return x + n;
    end;
    return f;
end;
```

```
gap> add5 := addN(5);
function( x ) ... end
gap> add5(17);
22
```

This may seem a bit pointless for now (you could easily have written `add5` yourself – or simply added 5, without any functions). But it can be a very powerful tool, especially once you add in some additional *state*.

```
newCounter := function()
    local f, c;
    c := 0;
    f := function()
        c := c + 1;
        return c;
    end;
    return f;
end;
```

Note how the inner function `f` accesses the variable `c` declared in the outer function, without a `local` statement of its own. So `f` modifies the variable `c` belonging to the outer function – and this works even after the outer function has completed. This is called a closure.

```
gap> counter := newCounter();
function( ) ... end
gap> counter();
1
gap> counter();
2
gap> counter();
3
```

3.6 Additional exercises

1. In GAP, strings are represented as lists of characters. Moreover, an integer can be turned into a string with the `String` command, the reverse can be done using `Int`.

```
gap> str := String(123);
"123"
gap> Int(str);
123
gap> str[1];  # this is a character
'1'
gap> [str[1]]; # we can put the character back into a list to form a string
"1"
gap> Int([str[1]]); # a string of digits can be converted into an integer
1
```

   Consider this implementation of the `Checksum` function from exercise 1.6. Try to figure out how it works. Try writing a solution of the `Weird` homework S1H1 based on this.

```
Checksum := a -> Sum(List(String(AbsInt(a)), x -> Int([x])));
```

   Also try to understand why the following variants work (consult the documentation for `Sum`).

```
Checksum2 := a -> Sum(String(AbsInt(a)), x -> Int([x]));
Checksum3 := a -> Sum(ListOfDigits(a));
```

2. The GAP `Positions(l,x)` returns a list of all indices `i` for which `l[i]=x` holds. Implement a function `pos(l,x)` doing the same, but without using `Positions`. However, you may use `Filtered`.

3.7 Homework

(s3-h1) Implement a function `filt(l,f)` which does exactly what `Filtered` does, but without using it. That is, it should return the a list consisting of the elements `x` of `l` for which `f(x)=true`. Your function must be of the form `function(l,f) return foldLeft(l, ...); end`.

(s3-h2) Implement a function `elemPropNth(l,f,n)` which returns the n-th element in `l` satisfying the property `f`, or `fail` if less than n elements of l satisfy the property. You may assume that n is a positive integer.

(s3-h3) Write a function `iter(n,f)` which takes a non-negative integer n and an unary function f, and returns a new unary function g which applies f exactly n times to its argument. So $iter(3, f)$ should return a function which maps a value x to $f(f(f(x)))$. For $n = 0$, a function corresponding to the identity map should be returned.