JUSTUS-LIEBIG-UNIVERSITÄT GIESSEN

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

Worksheet 5

Version 2015-01-28

5 Miscellaneous

5.1 Print versus return

There was some confusion about printing and returning a value. In general, returning a value is more useful than printing it, as it makes it possible to do further work with the returned value, while something we printed is visible to the user, but not to the rest of the program. To demonstrate the difference, compare the following two examples.

```
gap> f := function(x) return x + 42; end;;
gap> f(10);      # f really returns a value ...
52
gap> f(10);;     # ... we can suppress printing it with a second semicolon ...
gap> x := f(10); # ... or assign it to a variable ...
52
```

(Note that we use `Display` instead of `Print` in this example. One difference between the two is that `Display` automatically adds `\n` at the end of the output, thus starting a new line.)

```
gap> g := function(x) Display(x + 42); end;;
gap> g(10); # g does not return a value, it prints it ...
52
gap> g(10);; # ... thus a second semicolon does not suppress the output ...
52
gap> y := g(10); # ... and we cannot assign it to a variable
52
Error, Function call: <func> must return a value
not in any function at line 22 of *stdin*
gap>
```

Exercise 5.1.1

A natural number n is perfect if it equals the sum of all its proper divisors.
Write a function `isPerfect(n)` which takes an integer n and returns `true` if n is perfect, `false` otherwise. You may use `DivisorsInt` for this. Next, write a function `isPerfectOdd(n)` which takes an integer n and returns true if the value n is perfect and odd. Use `isPerfect` for this.

5.2 Variables versus values, and copying lists

In GAP, as in many other programming languages, a variable is simply a convenient handle, a reference, to a value respectively object. One must not confuse this handle with the actual value. Think of a variable as of your own name: You can be addressed via your name, but of course you and your name are not identical. In particular, multiple variables can point to the same object, just like you might be known under multiple names. To illustrate this, consider this example:

```
gap> a := [1,2,3];
[ 1, 2, 3 ]
gap> b := a;
[ 1, 2, 3 ]
gap> b[1] := 23;           # Modifying b ...
23
gap> a;                    # ... also modifies a ...
[ 23, 2, 3 ]
gap> IsIdenticalObj(a, b); # ... because they refer to the same (identical) object
true
```

In this program, we create a new list, and change the variable a to reference this list. We next let b also reference the same list. Then, we “modify b” and this also “modifies a”.

While this might seem confusing at first, the actual problem is that our informal description of the situation is not accurate. To be formally correct, we should say something like this: “We modify the list referenced by the variable b; since a references the same list, of course we can see this change also by accessing a”.

Again, this closely mirrors names used to address persons. If “Angela Merkel” changes the color of her hair, you won’t be surprised to find out that the the “Bundeskanzlerin” did the same thing, and so did “Angie”.

But sometimes, we really want to make copies of a value. In GAP, this can be done using ShallowCopy and StructuralCopy. The difference between them becomes visible for nested lists (such as matrices).

```
gap> a := [ [1], 2 ];;
gap> b := StructuralCopy(a);;
gap> c := ShallowCopy(a);;
gap> IsIdenticalObj(a, b) or IsIdenticalObj(a, c) or IsIdenticalObj(b, c);
false
gap> b[1][1] := 42;; b[2] := -42;;
gap> c[1][1] := 23;; c[2] := -23;;
gap> a; b; c;
[ [ 23 ], 2 ]
[ [ 42 ], -42 ]
[ [ 23 ], -23 ]
gap> IsIdenticalObj(a[1], b[1]); # StructuralCopy makes copies of everything...
false
gap> IsIdenticalObj(a[1], c[1]); # ... while ShallowCopy goes only one level deep.
true
```

Exercise 5.2.1

Write a function transMut(m) which takes a square matrix m and transposes it in-place. The function should return nothing, but rather modify the matrix m itself. For example:

```
gap> m := [[1, 2], [3, 4]];;
gap> transMut(m);;
gap> m;
[ [ 1, 3 ], [ 2, 4 ] ]
```

In order to prevent accidental changes to a matrix or any other mutable object, one can make it immutable.

```
gap> m := [[1, 2], [3, 4]];;
gap> IsMutable(m);
true
gap> MakeImmutable(m);;
gap> IsMutable(m);
false
gap> m[1][1]:=5;
Error, Lists Assignment: <list> must be a mutable list
not in any function at line 12 of *stdin*
you can 'return;' and ignore the assignment
brk>
```

Once an object is immutable, it must of course stay immutable (otherwise, it wouldn't be immutable in the first place). But you can make a mutable copy with help of the `ShallowCopy` command.

```
gap> m := [[1, 2], [3, 4]];;
gap> MakeImmutable(m);;
gap> m2 := ShallowCopy(m);
[ [ 1, 2 ], [ 3, 4 ] ]
gap> IsMutable(m2);
true
gap> m2[1] := [-5,4];  # the list m2 itself can now be modified...
[ -5, 4 ]
gap> m2[2][1] := 4;    # ...but its sublists (= matrix rows) still are immutable
Error, Lists Assignment: <list> must be a mutable list
not in any function at line 10 of *stdin*
you can 'return;' and ignore the assignment
brk> return;
gap> m2 := List(m, ShallowCopy); # ... but higher order functions come to our rescue
[ [ 1, 2 ], [ 3, 4 ] ]
gap> m2[2][1] := 4;
4
```

5.3 Short-circuit evaluation

Look at the following program and think about what it does, without running in the computer. What output do you expect? In particular, what value do you expect count to have at the end? Now run the program and compare what you get with your expectation. Do they agree? Discuss your observations.

```
count := 0;
CountAndRet := function(x)
    count := count + 1;
    return x;
end;
for x in [true, false] do
    for y in [true, false] do
        z := CountAndRet(x) or CountAndRet(y);
        Print(x, " or ", y, " = ", z, "\n");
    od;
od;
Print("count = ", count, "\n");
```

What happens here is called "short-circuit evaluation": When evaluating the expression `A and B`, if the term `A` evaluates to `false`, then GAP never evaluates `B`, as it already knows the final value of `A and B`.

This influences both correctness and efficiency of a program. Consider the following computation. By rearranging the predicate expression, we get a considerable improvement in speed.

```
gap> Number([1..10^6], n -> isPerfect(n) and IsEvenInt(n));; time;
5575
gap> Number([1..10^6], n -> IsEvenInt(n) and isPerfect(n));; time;
2688
```

5.4 Efficiency: Timings

We already introduced the `time` variable which measures how long the last command took (in milliseconds).

```
gap> Number(SymmetricGroup(8), x -> Order(x) = 2);; time;
177
```

However, these measurements can fluctuate quite a bit. Ideally, one would measure several times and take the average; perhaps also throw away outliers. The following helper function does that for you.

```
Time := function(f,x)
    local i, t, y;
    t := [];
    for i in [1..5] do
        y := ShallowCopy(x);
        t[i] := Runtime();
        f(y);
        t[i] := Runtime() - t[i];
    od;
    Sort(t);
    return QuoInt(t[2]+t[3]+t[4], 3);
end;
```

```
gap> f := G -> Number(G, x -> Order(x) = 2);;
gap> Time(f, SymmetricGroup(8));
167
```

5.5 Efficiency: Loops

Compare the following functions (they all should return the same thing).

1. Which one do you think will be fastest, which will be slowest?
2. Make a table: One row for each of the functions below, and one column for each of the group Σ_n for $5 \leq n \leq 9$. In the row f and column G, enter the value `Time(f,G)`. (You could write a little function which prints this table for your.)
3. Which function is really fastest, which slowest? Does this agree with your expectations?

```
h1 := G -> Sum(G, Order);

h2 := G -> Sum(Elements(G), Order);

h3 := function(G)
    local sum, g;
    sum := 0;
    for g in G do
        sum := sum + Order(g);
    od;
    return sum;
end;

h4 := function(G)
    local sum, elems, i;
    sum := 0;
    elems := Elements(G);
    for i in [1..Length(elems)] do
        sum := sum + Order(elems[i]);
    od;
    return sum;
end;

h5 := function(G)
    local sum, elems, i;
    sum := 0;
    elems := Elements(G);
    i := 1;
    while i <= Length(elems) do
        sum := sum + Order(elems[i]);
        i := i + 1;
    od;
    return sum;
end;
```

5.6 Efficiency: Caching

```
gap> h3 := G -> Sum(Elements(G), Order);;
gap> G := SymmetricGroup(9);;
gap> h3(G);; time;
334
gap> h3(G);; time;
67
gap> h3(SymmetricGroup(9));; time;
344
gap> h3(SymmetricGroup(9));; time;
345
```

What happens here is that GAP *caches* the results of various computations, i.e. it stores them and reuses them if the values are needed again later on. In this case, the value of `Elements` is cached. In the first two calls to `h3`, we pass in the same group twice, and the second time, we benefit from the cached result. In the third and fourth call to `h3`, we each time give it a fresh new group, and hence we do not see this benefit.

Caching also has drawbacks: It increases memory usage.

```
gap> G := SymmetricGroup(9);;
gap> MemoryUsage(G);
296
gap> Elements(G);;
gap> MemoryUsage(G);
18397566
gap> Int(MemoryUsage(G) / Size(G)); # Here, GAP needs ~50 bytes per group element
50
```

5.7 Efficiency: Variations on a theeme

1. Take a look at the following functions (each of which takes a list of integers, and returns the sum of the even integers in that list), and guess which one is the fastest, second fastest etc.
2. Copy & paste them into your computer and compare them: Make a table, with one row for each function, and one column for each of the intervals $[1..10^n]$ for $1 \leq n \leq 6$. In row f and column l, enter the value `Time(f,l)`. Feel free to skip entries if you think they'd become too large.
3. Do these results agree with your expectations? Can you explain them?
4. Repeat the test after replacing `IsEvenInt` by `IsPrimeInt`. Make a second table for these. How does this change affect the relative ranking of the various functions?

```
f1 := list -> Sum(Filtered(list, IsEvenInt));
```

```
f2 := function(list)
    local x, i;
    for i in Reversed([1..Length(list)]) do
        x := list[i];
        if not IsEvenInt(x) then
            Remove(list, i);
        fi;
    od;
    return Sum(list);
end;
f3 := function(list)
    local x, i;
    for i in [1..Length(list)] do
        x := list[i];
        if not IsEvenInt(x) then
            list[i] := 0;
        fi;
    od;
    return Sum(list);
end;
f4 := function(list)
    local x, i;
    i := 1;
    while i <= Length(list) do
        x := list[i];
        if not IsEvenInt(x) then
            list[i] := 0;
        fi;
        i := i + 1;
    od;
    return Sum(list);
end;
f5 := function(list)
    local x, sum;
    sum := 0;
    for x in list do
        if IsEvenInt(x) then
            sum := sum + x;
        fi;
    od;
    return sum;
end;
```

5.8 Homework

(s5-h1) This time you are supposed to help another student with his homework: Karl was asked to write a function `minLst` doing the following: Given two integer lists a and b, the function shall return a new list c such that $c[i] = min(a[i], b[i])$; if one of the two lists is longer than the other, treat the shorter list as if it was filled up to equal length with entries equal to ∞. For example, given the lists $[5, 6, 7]$ and $[10, 3]$, it should return $[5, 3, 7]$. The input lists must not be modified.

Karl’s attempt follows. Unfortunately, his program suffers from various problems. Your task is to help Karl by identifying and correcting all problems, and telling him about them.

So, write a short summary with a list of brief bullet points of what is wrong; suggest for each problem a fix that is as self-contained as possible (ideally, only requires changing a single line). Add these as comments in your submission file. Finally, include a corrected version of the program where you implemented your hints for Karl. Your final program really should be obtained from Karl’s attempt in incremental steps. You are not supposed to write a new program from scratch. This maximizes the learning effect for Karl.

```
blaBla := function(a, b)
    local c;
    c := a;
    for i in [1..Length(b)] do
        if a[i] > b[i] or i > Length(a) then
            c[i] := b[i];
        fi;
    od;
    return c;
end;
```

(s5-h2) Karl also came up with another potential implementation for the problem in Section 5.7. Unfortunately, his implementation contains a bug, and sometimes returns incorrect results.

```
f6 := function(list)
    local x;
    for x in list do
        if not IsEvenInt(x) then
            Remove(list, Position(list, x));
        fi;
    od;
    return Sum(list);
end;
```

Find five integer lists where the function returns the correct answer, and five where it returns a wrong answer. Put them into two lists, `correct` and `wrong`. Your submission should look like this:

```
correct := [ [...], ... ];
wrong := [ [...], ... ];
```

Also, explain what is going wrong and put your explanation into a comment.

(s5-h3) Implement a function `binarySearch(l,x)` which takes a sorted integer list l, and an integer x, and which returns either an index i such that $l[i] = x$, or `fail` if no such index exists. Your program should run for lists with 10^{10} elements in less than 100 milliseconds.

Hint: Exploit that l is sorted and perform a binary search: First, look at the middle element m of l. If it is a match, we are done. Otherwise, we can compare it with x: If $x < m$ then we only need to look at the first half of the list for x, otherwise only at the second half. Repeat. This way, you need to only look at about $\log_2(|l|)$ elements on average, instead of about $|l|/2$.