Winfried Auzinger
Klaus Felsenstein
Dirk Praetorius

SS 2009
(23. April 2009)

Übungen zur Vorlesung Computermathematik

Serie 6

Die Aufgaben mit Stern (*) sind bis zur Übung in der kommenden Woche fix vorzubereiten und werden dort abgeprüft (Minimalerfordernis).

Kopieren Sie Ihre Worksheets auf Ihren Account auf `lva.student.tuwien.ac.at`. *Überprüfen Sie vor der Übung, ob Ihre Codes unter Maple 12 auf* `lva` *einwandfrei funktionieren. Es wird empfohlen, für jedes Beispiel im Verzeichnis* `serie06` *ein eigenes Worksheet mit dem Namen* `aufgabe_xy.mw` *anzulegen. Alle zu verfassenden Prozeduren sind zu kommentieren und – je nach Angabe – mit verschiedenen Werten für die Parameter auszutesten.*

Verwenden Sie konsequent die Online-Hilfe – das ist wichtig für den erfolgreichen praktischen Umgang mit Maple. Insbesondere kommt es immer wieder vor, dass etwas in der Vorlesung aus [noch] nicht im Detail besprochen wurde – dann muss man sich zu helfen wissen. Explizite Verweise auf die Hilfe (z.B. `?set`*) deuten an, dass man sich die entsprechende Hilfe-Seite auf jeden Fall ansehen sollte. Manchen Aufgaben dienen auch der Wiederholung bzw. Vertiefung von Themen, die in der Vorlesung behandelt wurden.*

Hinweis zur Rückgabewerten von Prozeduren: Das kann im Prinzip etwas ganz beliebiges sein, z.B. auch eine exprseq, *wenn man mehrere Werte gleichzeitig zurückgeben will. Normalerweise wird zurückgegeben, was unmittelbar vor* `end proc` *berechnet wurde. Im allgemeinen verwendet man*

```
> return ... ;
```

um an einer beliebigen Stelle abzubrechen und `...` *zurückzugeben (wichtig z.B. bei Verzweigungen innerhalb der Prozedur). Man beachte auch, dass die Argumente von Prozeduren nur der Parameterübergabe, aber nicht der Rückgabe von Werten dienen.*

Aufgabe 6.1*. Eine Liste (`?list`) ist ein Objekt der Form [*exprseq*], wobei *exprseq* eine beliebige Folge von Ausdrücken ist, in der angegebenen Ordnung (im Gegensatz zu einer Menge, `?set`). Manche Operationen an Listen sind ein wenig umständlich, z.B. Hinzufügen eines Elementes:

```
> l := [1,zwei,'drei'];
                          l := [1, zwei, drei]
> l[4] := "vier";
                          Error, out of bound assignment to a list
> l:=[op(l),"vier"];
                          l := [1, zwei, drei, "vier"]
```

Schreiben Sie

- eine Prozedur `ladd(l,element)`, die ein Element an eine Liste `l` hinten anfügt und die modifizierte Liste zurückgibt;
- eine Prozedur `lcut(l,anzahl)`, die `anzahl` Elemente am Ende entfernt und die verkürzte Liste zurückgibt. Falls `anzahl>=nops(l)`, ist die leere Liste `[]` zurückzugeben.

Zu beachten: Die Anzahl der Elemente erhält man mittels `nops(...)`. `length(...)` ist etwas anderes.

Das package `ListTools` enthält diverse Operationen an Listen. Da gibt es z.B. `?Join`. Schreiben Sie eine modifizierte Version `MyJoin`, die dasselbe tut, den 'Separator' jedoch auch am Beginn und am Ende hinzufügt. `MyJoin` soll `Join` intern *nicht* verwenden.

Aufgabe 6.2*. Unter *impliziter Differentiation* versteht man folgenden Vorgang. Angenommen, eine Kurve im $\mathbb{R}^2$ wird durch eine Gleichung der Form $f(x,y)=0$ beschrieben (f stetig differenzierbar in beiden Argumenten). Sei (x_0,y_0) mit $f(x_0,y_0)=0$ ein Punkt auf der Kurve mit $\frac{\partial}{\partial y}f(x_0,y_0)\neq 0$. Ein Satz aus der Analysis besagt, dass die Kurve dann in der Umgebung dieser Stelle in der Form $y=y(x)$ dargestellt werden kann.

Falls f eine hinreichend komplizierte Gestalt hat, kann man jedoch nicht explizit formelmäßig nach y auflösen, d.h. die Funktion $y(x)$ kann man nicht 'hinschreiben'. Angenommen, wir wollen trotzdem den Wert von $y'(x)$ an einer konkreten Stelle x berechnen. Das geht mit Hilfe der Kettenregel,

$$0 \equiv \frac{d}{dx} f(x, y(x)) = \frac{\partial}{\partial x} f(x, y(x)) + \frac{\partial}{\partial y} f(x, y(x))\, y'(x),$$

was man unter der Annahme $\frac{\partial f}{\partial y} \neq 0$ nach dem gesuchten Wert $y'(x)$ auflösen kann.

Schreiben Sie eine Prozedur `impdiff(f,x,y)`, die das für eine vorgegebene Funktion `f` an einer gegebenen Stelle (x, y) durchführt. Falls $f(x, y) \neq 0$ oder $\frac{\partial}{\partial y} f(x, y) = 0$, wird `NULL` (d.h. nichts) zurückgegeben.
Alles in exakter Rechnung. Man beachte, dass `f` als Prozedur definiert sein muss; ihr Name wird an `impdiff` als Parameter übergeben. Für die Berechnung der partiellen Ableitungen von `f` verwendet man `diff` oder `D`.

Ein einfaches Testbeispiel: Für den Kreis, $f(x, y) = x^2 + y^2 - 1$, funktioniert das für $x \neq \pm 1$. Hier kann man auch explizit nach y auflösen und vergleichen.

Aufgabe 6.3*. Eine `table` ist nichts anderes als ein assoziatives Array, z.B.

```
> t := table():
> t["ich"] := "Müller":
> t["du"] := "Kuh":
> t["er"] := "Esel";
> t;
>                   t
> print(t);
                    table(["du" = "Kuh", "ich" = "Müller", "er" = "Esel"])
```

(In diesem Beispiel werden Strings verwendet, aber das ist nicht wesentlich.) Die Ausgabe mittels `print` ist nicht besonders übersichtlich. Schreiben Sie zunächst eine Prozedur `tprint(t)`, die die Tabelleneinträge separat Zeile für Zeile untereinander mittels `print` in folgender Form ausgibt:

```
> tprint(t);
                    t
                    "du", "Kuh"
                    "ich", "Müller"
                    "er", "Esel"
```

(Auf die Reihenfolge hat man hier keinen direkten Einfluss, weil ein assoziatives Array nicht automatisch angeordnet ist.)

Hinweis: `op(op(t))` konvertiert die Tabelle in eine *Liste von Gleichungen*, die bequem weiterverarbeitet werden kann. (`?lhs, ?rhs`).

Noch 'schönere' Lösungen sind natürlich nicht verboten (freiwillige Spielerei); eigentlich benötigt man dafür jedoch formatierte Ausgabe (wird später besprochen).

Aufgabe 6.4*. Ein `for`-Konstrukt wie etwa in

```
> l := [a,b,c]:
> for i in l do i end do;
                              a
                              b
                              c
```

durchläuft automatisch alle Elemente einer Datenstruktur (in diesem Fall einer Liste; `?for`). Verwenden Sie dieses Konstrukt, um aus zwei gegebenen Mengen `A` und `B` von Maple-Objekten die Menge

$$\{[x, y] :\ x \in A,\, y \in B,\, x \neq y\}$$

zurückzugeben.

Beispiel (neue Prozedur `setset`):

```
> A := {1,2,2};
                    {1,2}
> B := {1,3};
                    {1,3}

> setset(A,B);
                    {[1,3],[2,1],[2,3]}
```

Am besten macht man zunächst ein geschachteltes `for ... in` – `for ... in` Konstrukt und 'filtert' dann entsprechend.

Aufgabe 6.5. Fortsetzung von Aufgabe 6.3: Modifizieren Sie `tprint` so, dass die Ausgabe nach den Indizes sortiert erscheint (du, er, ich), indem Sie die Indizes als Liste extrahieren, mittels `sort` sortieren und das Ganze wieder zusammensetzen.

Aufgabe 6.6. Man schreibe zwei Maple-Prozeduren `innerprod3(a,b)` und `outerprod3(a,b)`, die zwei Listen `a` und `b` der Länge 3 als Vektoren im $\mathbb{R}^3$ interpretieren und das innere (Skalarprodukt, $a \cdot b$) bzw. das äußere (Vektorprodukt, $a \times b$, letzteres wieder als Liste) zurückgeben. Falls `a` oder `b` nicht die Länge 3 hat, ist jeweils `NULL` zurückzugeben.

'Beweisen' Sie damit die Tatsache $a \times b \perp a,\ a \times b \perp b$, und die Lagrange-Identität

$$(a \times b) \cdot (c \times d) = (a \cdot c)(b \cdot d) - (b \cdot c)(a \cdot d)$$

indem Sie beliebige 'generische' Vektoren (ohne konkrete Werte) übergeben.

Aufgabe 6.7. Schreiben Sie zwei Maple-Prozeduren `encode(d::integer)` und `decode(e::string)`: `encode` 'verschlüsselt' die Zahl `d` → `e`, und `decode` entschlüsselt sie. Dabei soll es nicht um Sicherheit gehen ('Datenschutz'), sondern nur darum, dass der Wert nicht direkt verwendbar und als 'verschlüsselt' erkennbar ist, aber jederzeit rekonstruiert werden kann.

Vorschlag: Wandle z.B. `12345`, `-12345` um in

`"ENCODED+ABCDE"` `"ENCODED-ABCDE"`

(`1` → `A`, etc.). Ein bisschen Handarbeit mit Strings. Sie dürfen sich auch etwas Trickreicheres einfallen lassen ('echte' Verschlüsselung?!), aber das ist nicht verlangt.

Aufgabe 6.8. Ein *Stack* (Stapelspeicher) S ist eine einfache lineare Datenstruktur, vorzustellen als ein Vektor $(s_1, s_2, \ldots, s_n)$ mit variabler Höhe (Länge) n (kann auch leer sein, $n = 0$), wobei jedoch nur folgende Operationen zulässig sind:

- Man kann oben ein neues Element hinzufügen (*push*).
- Man kann das oberste Element entfernen, falls S nicht leer ist (*pop*).

Wählen Sie eine globale Variable mit dem fest vorgegebenem Namen `S` (wird in jeder der Prozeduren als `global S;` deklariert) in Form einer Liste als zugrunde liegende Datenstruktur und verwenden Sie `op()`, um auf die Elemente von `S` zuzugreifen. Der leere Stack ist die leere Liste `[]`. Damit implementieren Sie folgende Prozeduren:

- `S_ini()` erzeugt den leeren Stack, `S:=[];` (Höhe 0).
- `S_push(x)` fügt bei `S` das Element `x` oben an.
- `S_pop()` entfernt das oberste Element und gibt es als Wert zurück (nur zulässig, falls `S` nicht leer, ansonsten Ausgabe `NULL`).
- `S_peek()` liest das oberste Element aus, ohne es aus `S` zu entfernen.
- `S_height()` liefert die aktuelle Höhe von `S`.

Aufgabe 6.9. Modifikation von Aufgabe 6.8, unter der Annahme, dass alle Elemente auf dem Stack vom Type `integer` sind:

Auf die Liste `S`, die den Stack repräsentiert, kann man auch ganz normal zugreifen, das ist bei einem 'echten' Stack aber eigentlich 'verboten'. Modifizieren Sie das Ganze daher so, dass jede Zahl `x` auf `S` automatisch intern 'verschlüsselt', und beim Auslesen 'entschlüsselt' wird. Beim direkten Zugriff (Lesen) sieht man nur die verschlüsselte Version. Zusätzlich kann man sich mittels `protect('S')` am Beginn des Worksheets dagegen 'schützen', `S` 'irrtümlich' zu verändern. Die modifizierten Prozeduren `S_pop` etc. verwenden dazu intern am Beginn `unprotect('S')` und am Ende `protect('S')`. Außerdem wird `S_pop` so modifiziert, dass sie nur `integer`-Werte akzeptiert: `S_pop:=proc(x::integer)`.

'Ver/Ent-schlüsselung' unter Verwendung der Prozeduren aus Aufgabe 6.7.

Aufgabe 6.10. Sei $K := \{\{0, 1, \ldots, p\}, \oplus, \odot\}$ (p Primzahl) der endliche (Restklassen-)Körper mit den Operationen

$$x \oplus y := (x + y) \mod p, \qquad x \odot y := (xy) \mod p.$$

(vgl. `? mod`). Diese algebraische Struktur erfüllt alle Axiome für einen *Körper*.[1]

Schreiben Sie eine Maple-Prozedur `generate_field(p::prime)`, die für einen vorgegebenen Wert von p zwei als global deklarierte `Arrays` mit den fest vorgegebenen Namen `Plus`, `Mal`, der Dimension `0..p-1,0..p-1` zurückliefert, die die beiden Operationen $\oplus$ und $\odot$ repräsentieren, d.h. so dass `Plus[i,j] = i` $\oplus$ `j`, `Mal[i,j] = i` $\odot$ `j`. Weiters belegt man zwei eindimensionale globale Arrays `Pinv` und `Minv` der Dimension `0..p-1` so dass `Pinv[i] = -i` und `Minv[i] = 1/i` im Sinne der Körperoperationen. (Sonderfall: `Minv[0]:=NULL`.)

Hinweis: Man deklariert die Arrays Prozedur-intern als globale Variablen,

```
> global Plus, Mal, Pinv, Minv;
> Plus := Array(0..p-1,0..p-1);
> Mal := Array(0..p-1,0..p-1);
> Pinv := Array(0..p-1);
> Minv := Array(0..p-1);
```

und belegt sie mit den entsprechenden Werten. Damit sind sie nach dem Aufruf `generate_field(p)` für einen bestimmten Wert von p außerhalb der Prozedur (in dem umgebenden Worksheet) verfügbar.

Jetzt könnte man schon bequem damit umgehen, z.B. ($p = 5$):

```
> Plus[1,4];
                          0
> Pinv[1];
                          4
> Plus[1,5]
Error, Array index out of range
```

Schreiben Sie vier weitere Prozeduren `plus`, `mal`, `pinv`, `minv`, die auf diese globalen Arrays zugreifen, so dass man wie folgt rechnen kann:

```
> plus(1,4);
                          0
> pinv(1);
                          4
> plus(1,5);
>
```

etc. Diese Prozeduren sollen `NULL` zurückgeben, falls sie mit unzulässigen Parametern aufgerufen werden.

Freiwillig[2]: Schreiben Sie eine weitere Prozedur `svmul(s,v)`, die eine Liste `v` beliebiger Länge, mit Komponenten in K als Vektor interpretiert und mit dem Skalar `s` $\in K$ multipliziert (Operation in Vektorraum über K). Wie immer: `NULL`, falls unzulässige Argumente.

[1] 'Dummy'-Definition eines Körpers: Ein Körper (engl. *field*) ist eine algebraische Struktur, in der für Addition, Multiplikation bzw. Kombinationen davon dieselben Rechenregeln gelten wie in $\mathbb{R}$ oder $\mathbb{C}$.