Initialisierung von Maple

Maple hat einen Kernel, der neu gestartet werden kann. Dabei werden alle Zuweisungen gelöscht:

```
> restart;
```

Einzelne Zuweisungen lassen sich mit **unassign** löschen. Der Variablenname steht dabei in Hochkommas:

```
> k := 2; k; unassign ('k'); k;
```

$$k := 2$$

$$2$$

$$k$$

Mit dem Befehl **assign** lassen sich übrigens Ergebnisse zuweisen: Hier werden die Ergebnisse für x und y in diese Variablen explizit gespeichert.

```
> s := solve( {x+y=1, 2*x+y=3}, {x,y} );
> assign(s);
  x,y;
```

$$s := \{x = 2, y = -1\}$$

$$2, -1$$

Zusätzliche Bibliotheken werden mit **with** geladen:

```
> with (combinat);
Warning, new definition for Chi
```

[Chi, *bell*, binomial, *cartprod*, *character*, *choose*, *composition*, *conjpart*, *decodepart*, *encodepart*, *fibonacci*, *firstpart*, *graycode*, *inttovec*, *lastpart*, *multinomial*, *nextpart*, *numbcomb*, *numbcomp*, *numbpart*, *numbperm*, *partition*, *permute*, *powerset*, *prevpart*, *randcomb*, *randpart*, *randperm*, *stirling1*, *stirling2*, *subsets*, *vectoint*]

Wir können aber auch einzelne Teile einer Bibliothek laden, normalerweise einzelne Prozeduren:

```
> with (linalg, [transpose, matrix]);
```

[*matrix*, *transpose*]

Definition von Funktionen in Maple

```
> restart;
```

Einfache Funktionen werden in Maple in starker Anlehnung an die mathematische Schreibweise definiert:
Die Funktion *f* bildet *x* auf $x^2 + 2$ ab.

```
> f := x -> x^2+2; f(2);
```

$$f := x \to x^2 + 2$$

$$6$$

Das geht natürlich auch komplizierter mit zwei Variablen:

```
> g := (x,y) -> x+y; g(1,2);
```

$$g := (x, y) \to x + y$$

$$3$$

Für algorithmische Beschreibungen bietet sich eher der **proc** - Aufruf an:

```
> fibonacci := proc (i)
                if (i = 0) then RETURN(0); fi;
                if (i = 1) then RETURN(1); fi;
                RETURN (fibonacci (i-1) + fibonacci (i-2));
                end;
```

fibonacci := **proc**(*i*)

if $i = 0$ **then** RETURN(0) **fi**;

if $i = 1$ **then** RETURN(1) **fi**;

RETURN(fibonacci($i - 1$) + fibonacci($i - 2$))

end

Diese Funktion hat eine Laufzeit von O(e^n). Mit der Option **remember** laeßt sich die Zeitkomplexität auf linear verringern:

```
> fibonacci_lin := proc (i) option remember;
```

```
                    if (i<2) then i else fibonacci_lin (i-1) + fibonacci_lin
  (i-2); fi; end:
```

Beachte, daß hier der Befehl RETURN fehlt. Wenn keine Zuweisung erfolgt, wird der berechnete Wert automatisch zurückgeliefert.
Der Doppelpunkt hinter end verhindert, daß die Definition der Prozedur angezeigt wird, das spart Platz.
Die Zeitkomplexität zeigt sich mehr als deutlich in den folgenden Aufrufen:

```
> fibonacci_lin (22);
```

$$17711$$

```
> fibonacci (22);
```

$$17711$$

Listen und Mengen

Ein Grund, Maple in der Diskreten Mathematik zu verwenden ist sicherlich die einfache Behandlung von Listen und Mengen.
Eine Liste enthält Objekte in der Reihenfolge des Einfügens, eine Menge in einer beliebigen Reihenfolge, zur Darstellung werden sie der Größe nach sortiert.

```
> Liste := [3,1,2,5,4]; Menge := {3,1,2,5,4};
```

$$Liste := [3, 1, 2, 5, 4]$$

$$Menge := \{1, 2, 3, 4, 5\}$$

Auf die Elemente von Listen oder Mengen läßt sich mit Hilfe von Operatoren zugreifen:
op (i, Liste) liefert das i-te Element von Liste.

```
> op (2,Liste);
```

$$1$$

Damit kann man aber auch auf alle anderen Strukturen zugreifen:

```
> term := x=3^n; op (0, term), op (1, term), op (2, term), op ([2,0], term),
  op ([2,1], term), op ([2,2], term);
```

$$term := 23 = 3^n$$

$$=, 23, 3^n, ^, 3, n$$

Anzahl der Elemente in einer Liste:

```
> nops (Liste);
```

$$5$$

Konkatenation einer Liste (**op(Liste)** gibt alle Elemente des Objekts aus):

```
> Liste := [op(Liste), 99]; op (Liste);
```

$$Liste := [3, 1, 2, 5, 4, 99]$$

$$3, 1, 2, 5, 4, 99$$

Löschen eines Elements aus einer Liste:

```
> Liste := subsop (2=NULL, Liste);
```

$$Liste := [3, 2, 5, 4, 99]$$

Wir können aber auch das Element 99 aus der Liste löschen:

```
> Liste := subs (99=NULL, Liste);
```

$$Liste := [3, 2, 5, 4]$$

Wir möchten eine Teilliste mit den Elementen 2-4:

```
> Liste := Liste[2..4];
```

$$Liste := [2, 5, 4]$$

Wir bauen eine neue Liste mit den Werten 0 bis 10, in die Liste werden die ausgewerteten Ausdrücke eingetragen:

```
> Liste := [seq (i, i=0..10)];
```

$$Liste := [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$$

Genauso läßt sich eine Liste mit den Fibonacci-Zahlen erzeugen:

```
> [seq (fibonacci_lin (i), i=0..10)];
```

$$[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55]$$

Auf die Liste mit den Zahlen von 0 bis 10 wenden wir nun eine beliebige Operation an.
Der Operator **map** bildet jedes Element des zweiten Parameters mittels der Abbildung im ersten Parameter ab,

das funktioniert auch mit Matrizen / Vektoren.

```
> map (x -> x^2+2, Liste);
```

[2, 3, 6, 11, 18, 27, 38, 51, 66, 83, 102]

Beispiel: Permutation von Mengen

Als Beispiel, das Mengen / Listen und Prozedurdefinitionen in Maple vereint, bietet sich die Permutation an:
Die Funktion **print()** gibt hier das Ergebnis an die Standardausgabe aus. Variablen, die innerhalb des Prozedurkontextes verändert werden, müssen als local definiert werden, sie sind nur innerhalb des Kontextes definie
Die als Parameter übergebenen Variablen können nicht verändert werden.
Soll Variablen, die außerhalb definiert wurden, etwas zugewiesen werden, müssen diese als global definiert werden.
Die **for** - Schleife wird mit **do** begonnen und ended mit **od;**

```
> permute1 := proc (k, p, l)
              local i;
              if (nops (l)=0) then print (p); fi;
              for i from 1 to nops(l) do
                  permute1 (k+1, [op(p), op(i,l)], subsop((i)=NULL,l)):
              od:
              end:
```

Wir permutieren eine Menge der Zahlen 1 bis 3 durch:

```
> permute1 (1, [], {seq (i, i=1..3)});
```

[1, 2, 3]
[1, 3, 2]
[2, 1, 3]
[2, 3, 1]
[3, 1, 2]
[3, 2, 1]

Ein / Ausgabe mit Maple

Maple kann über eine Schnittstelle auch mit Dateien umgehen.

Speichern / Einlesen von Definitionen

Wir können Zwischergebnisse / Zuweisungen speichern:

```
> save `test.m`;
```

Dann starten wir den Kernel neu:

```
> restart;
```

Natürlich fehlt uns jetzt die Definition der Liste:

```
> Liste;
```

Liste

Wir lesen alle Definitionen neu ein:

```
> read (`test.m`);
```

Und dann ist auch die Liste wieder da:

```
> Liste;
```

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

Ein- / Ausgabe von Zahlen

```
> restart;
```

Maple kann normale ASCII-Dateien schreiben, lesen und parsen.
Maple kennt alle Möglichkeiten der Kommunikation, die in C definiert sind, also auch Streams und Pipes.

Einlesen von Datentabellen:

Achtung: Das Öffnen von Dateien erfordert eine **absolute Pfadangabe** in UNIX-Konvention!
Das Beispiel hier ist die Adjazenzliste aus der Hausaufgabe. Hier stejht in der ersten Zeile die Anzahl der Kanten (interessiert uns eigentlich gar nicht), und in den nachfolgenden Zeilen jeweils ein Paar von Knoten.

```
> fd := fopen (`/usr/stud/rodler/ds1/ue5-graph.in`, READ, TEXT);
```

$$fd := 0$$

Wir lesen die Daten zeilenweise ein, wobei wir jeweils die ersten 2 Spalte berücksichtigen.
Genausogut können wir auch float parsen, bzw. eine Kombination von numerischen Typen (siehe Hilfe).
Achtung: Die folgende Zeile funktioniert zwar sehr wohl, allerdings kann man auch auf den Suns waehrend des Einlesens Kaffeetrinken gehen!

```
> t := readdata (fd, integer, 2):
Warning, computation interrupted
> fclose (fd);
```

Besser, wir lesen die Datei zeilenweise ein und kümmern uns jeweils um diese Zeile:

```
> fd := fopen (`/usr/stud/rodler/ds1/ue5-graph.2.in`, READ, TEXT);
```

$$fd := 0$$

```
> t := readline (fd);
```

$$t := \text{"1000000"}$$

```
> fclose (fd);
```

Noch geschickter ist es natürlich, die Datei gleich zeilenweise zu parsen und in diese Schleife die Berechnung einzufügen:

```
> fd := fopen (`/usr/stud/rodler/ds1/ue5-graph.in`, READ, TEXT);
```

$$fd := 0$$

Die nachfolgende Schleife liest jeweils Paare von Integern ein und speichert sie zeilenweise in der Variablen line:

```
> while line <> 0 do
    line := fscanf (fd, "%d %d");
  od;
```

$$line := [1000000, 0]$$
$$line := [111572, 0]$$
$$line := [11596, 0]$$
$$line := [120117, 0]$$
$$line := [124511, 0]$$
$$line := [474060, 0]$$

```
> fclose (fd);
```

Und das Ergebnis können wir irgendwohin ausgeben:

```
> fd2 := fopen (`/usr/stud/rodler/ds1/testausgabe.txt`, WRITE, TEXT);
```

$$fd2 := 0$$

```
> writedata (fd2, [[1,2,3],[4,5],[1]], integer);
```

Das geht natürlich auch beliebig formatiert (wie in C) auf das Terminal:

```
> writedata (terminal, [[1,2,3],[4,5],[1]], integer, proc(f,x)
  fprintf(f,`%a`,x) end);
1JK
4M
1
> fclose (fd2);
```

Benutzerinteraktion

Wenn wir eine Benutzereingabe benötigen, können wir den Befehl **readstat** benutzen:
Die Eingabe muß mit einem Semikolon abgeschlossen werden.

```
> x := readstat (`Eine beliebige Zahl: `);
Eine beliebige Zahl: 23;
```

$$x := 23$$