

## Hinweise zu diesem Worksheet:

Die Bearbeitung dieses Worksheets ist freiwillig. Der Fokus des Begleitpraktikums liegt zwar auf der Mathematik und nicht dem Programmieren, trotzdem wird es empfohlen dieses Worksheet zu bearbeiten, um einen möglichst angenehmen Einstieg in den Umgang mit Maple zu gewährleisten.

- 1.) Der erste Teil vermittelt alle grundlegenden Programmierkonzepte die du zum Bestehen des Praktikums benötigst. Am Ende jedes Abschnitts findest du kurze Übungen mit deren Hilfe du überprüfen kannst, ob du den entsprechenden Stoff (schon) beherrschst.
- 2.) Der zweite Teil behandelt die Aussagenlogik und ihre Erweiterung um Quantoren.
- 3.) Teil drei erklärt wie man mit Maple einfache Programme schreibt.

### Für Programmiererfahrene

An ein paar Stellen in Teil eins und drei finden sich für Studierende, die schon Erfahrung im Programmieren gesammelt haben, ausklappbare Abschnitte mit zusätzlichen Hinweisen / Herausforderungen.

Viel Spaß!

## 1.) Einführung in die Programmierung mit Maple

Aufgaben: 4 freiwillige

### Benutzung der Oberfläche

Unten siehst du die Maple-Eingabeaufforderung in der du Befehle eingeben kannst. Gib dort "4+4;" ein (ohne die Anführungsstriche) und drücke ENTER.

Jede Maple Anweisung muss mit einem ";" oder einem ":" abgeschlossen werden. Der Doppelpunkt unterdrückt hierbei die Ausgabe des Ergebnisses.

Mit SHIFT-ENTER kann man in der Eingabeaufforderung eine neue Zeile erstellen. Gib unten zusätzlich ein:

```
4+1;
```

```
4+2:
```

**> # Hier sollen zwei zusätzliche Zeilen hin.**

Mit "#" kann man Kommentare einfügen, das heißt Maple ignoriert jegliche Zeichen die in derselben Zeile folgen.

Eine neue Maple-Eingabeaufforderung kann man übrigens über die Menüleiste oben hinzufügen: klicke dazu auf Insert > Execution Group > After Cursor oder drücke einfach Control-J.

*Hinweis:* Du willst vermutlich die Standardeinstellung für eine neue Eingabeaufforderung ändern, gehe hierfür auf Tools->Options->Display, ändere "Input Display" auf "Maple Notation" und bestätige alles mit "Apply Globally".

Variablen definieren und ihnen Werte zuweisen kann man mit dem "!=" Operator. Führe die nächsten Zeilen aus, indem du ENTER drückst.

```
> a := 2;  
b := 2;  
a := a*b;
```

Die Variable "b" besitzt nun den Wert 2 und "a" damit den Wert 4.

Wichtig ist hier "a\*b" und nicht etwa "ab" zu schreiben, da "ab" als eine neue, noch nicht definierte Variable aufgefasst würde.

```
> a := ab;  
a;
```

Man sieht, dass "ab" bislang kein Wert zugewiesen wurde. Deswegen enthält die Variable jetzt keine konkrete Zahl, sondern sie selbst enthält wiederum die Variable "ab".

Nachdem wir nun "a" und "b" Werte zugewiesen haben, ist es höchste Zeit unsere bisherige Arbeit zu sichern.

Drücke dazu Control-S (bzw. Strg-S auf einer deutschen Tastatur).

**Tipp:** Man kann bei Maple eine Autosave-Funktion aktivieren. Klicke dazu oben in der Menüleiste auf Tools > Options: Im sich öffnenden Fenster sollte sich ein Häkchen bei "Auto save every 3 minutes" befinden. Falls nicht, füge es hinzu und klicke dann unten auf "Apply Globally".

**Tipp:** Oben in der Werkzeugleiste kann man über die Symbole mit der Lupe den Zoom auf 150% stellen und spart damit Arztkosten.

### FREIWILLIGE ÜBUNG [01]:

- 1.) Welches Zeichen steht in Maple für Kommentare?
- 2.) Wie weist man einer Variable einen Wert zu?

## Zahlen

In Maple kann man Variablen natürlich nicht nur ganze Zahlen zuweisen, sondern auch rationale, reelle oder komplexe Zahlen.

```
> a := 5/12;  
a := 4*a;
```

Maple unterscheidet dabei zwischen verschiedenen Typen und rechnet beispielsweise mit Bruchzahlen ohne sie in Gleitkommazahlen umzuwandeln.

Dies nennt man auch "Symbolisches Rechnen" und ermöglicht es Maple ohne Rundungsfehler zu arbeiten.

Man kann sich über den Befehl "**whattype**" von Maple den Typ einer Variable oder eines bestimmten Ausdrucks anzeigen lassen.

```
> whattype( 1 );  
whattype( a );  
whattype( 4.2 );
```

Möchte man jedoch herausfinden welche der beiden Zahlen

$\frac{10}{7}$  oder  $\sqrt{2}$  größer ist, so kann

man den Befehl "**evalf**" benutzen (für evaluate as float, float = Gleitkommazahl).

```
> 10 / 7;  
sqrt(2);  
evalf( 10 / 7 );  
evalf( sqrt(2) );
```

Die Fähigkeit zum symbolischen Rechnen beschränkt sich natürlich nicht nur auf Bruchzahlen. Beispielsweise kann man mit dem Befehl "**simplify**" Maple dazu auffordern Gleichungen, Brüche uvm. zu vereinfachen.

```
> cos(x)^2 + sin(x)^2;  
simplify( cos(x)^2 + sin(x)^2 );
```

Maple besitzt eine Vielzahl nützlicher Kommandos und mit "?Befehl" kann man sich die jeweilige Hilfe anzeigen lassen.

```
> ?evalf
```

### Für Programmiererfahrene

Anders als in Java oder C++ kann eine Variable in Maple im Laufe ihres Lebens verschiedene Typen haben. Ebenso wenig muss man bei der Definition einer Variablen ihren Typ angeben. Folgendes etwa ist vollkommen legitim:

```
> a := 1:  
a := 1/2:  
a := 0.5:
```

Diese Eigenschaft macht Maple natürlich bequemer zu benutzen als Java oder C++, bedeutet während des Programmierens allerdings nur, dass man umso mehr ein Auge darauf haben muss, welche Variablen zu welcher Zeit welchen Typ besitzen.

Nicht-belegte Variablen haben übrigens den Typ "symbol".

```
> a := 'a': # löscht den Inhalt der Variable a  
whattype(a);
```

### FREIWILLIGE ÜBUNG [02]:

- 1.) Welche Zahl ist größer:  $\sqrt{2}$  oder  $\frac{30547}{21600}$  ?
- 2.) Wie ruft man die Maple-Hilfe zu einem Befehl auf?

### Datenstrukturen

Maple bietet verschiedene sogenannte Datenstrukturen in denen man Objekte, Zahlen, etc. zusammen speichern kann.

Die grundlegenden Datenstrukturen, die wir im Praktikum benutzen werden, sind sogenannte *Listen* und *Mengen*.

Eine Liste definiert man über

```
> L := [ 3,1,4,1 ];  
whattype(L);
```

und auf den i-ten Eintrag greift man mithilfe von "`L[i]`" zu.

```
> L[2];
```

Eine Liste zeichnet dabei aus, dass sowohl die Reihenfolge der Elemente bei der Definition relevant ist, als auch, dass sich Listenelemente wiederholen können.

Eine Menge hingegen speichert jedes vorkommende Element nur einmal und sortiert diese automatisch um. Welche Reihenfolge Maple dabei nimmt, kann man von außen nicht beeinflussen. Wir werden Mengen auf dem nächsten Sheet aus Sicht der Mathematik betrachten.

```
> M := { 3,1,4,1 };  
whattype(M);  
M[2];
```

Die Länge oder Anzahl Elemente einer Liste bzw. Menge kann man über die Befehle "`nops`" (number of operators) oder "`numelems`" (number of elements) erhalten.

```
> nops(M);  
numelems(M);
```

Listen und Mengen können in Maple beliebige Elemente - also auch andere Listen oder Mengen - haben. Listen die Listen als Elemente enthalten nennt man auch *verschachtelt*.

**Aufgabe:** Definiere eine Liste "A" und eine Menge "B" mit den drei Elementen [1,2], 1, 2. Welches Element steht jeweils an dritter Stelle?

Listen und Mengen kann man auch über sogenannte *Folgen* konstruieren. Dabei ist eine Folge nichts anderes als eine durch Kommata getrennte Abfolge von Objekten und Ausdrücken.

```
> 1,2,[3,4], x=x;  
whattype( 1,2,[3,4],x=x ); # exprseq steht für expression  
sequence
```

Ein nützlicher Befehl um Folgen zu erstellen ist "`seq`" (für sequence), diese können dann benutzt werden um Listen oder Mengen zu definieren.

```
> seq( 5..10 );  
seq( 10..20, 5 );  
seq( 5..10 ), seq( 10..20, 5 );  
A := [ seq( 5..10 ), seq( 10..20, 5 ) ];
```

```
> ?seq
```

Alternativ kann man als Kurzschreibweise für "`seq( 5..10 )`" auch "`$5..10`" benutzen.

```
> seq( 5..10 );  
$5..10;
```

**Aufgabe:** Erstelle eine Liste, die alle ungeraden Zahlen zwischen 40 und 80 enthält.

**Aufgabe:** Erstelle eine Menge, die alle ganzen Zahlen zwischen 10 und 40 enthält, die sich durch 5 oder 7 teilen lassen.

Einzelne Einträge einer Liste kann man nachträglich noch über "[:=" verändern,

```
> A[2] := 13;
```

leider ist es auf diese Art und Weise jedoch nicht möglich eine Liste zu verlängern.

Dazu kann man sich des "`op`" Befehls bedienen. "`op`" gibt die Elemente einer Liste oder Menge als Folge zurück.

```
> op(A);
```

**Aufgabe:** Füge hinten an die Liste "A" eine 0 hinzu. Verwende dabei "op".

### FREIWILLIGE ÜBUNG [03]:

Erstelle mit Maple eine Menge, die alle ganzen Zahlen zwischen 5 und 20 enthält, die durch 3 oder 5 teilbar sind. Nutze dabei den "seq" Befehl.

## Steuerkonstrukte

Oft möchte man einen bestimmten Befehl mehrfach ausführen oder auf jedes Element einer Liste anwenden. Zu diesem Zweck gibt es in Maple sogenannte **for**-Schleifen.

```
> squares := [ $1..20 ];
for i from 1 to 20 do
    squares[i] := i^2;    # ^2 steht für "hoch 2"
end do;
squares;
```

Die Schleifenvariable "i" startet bei 1 und wird nach jedem Schritt um 1 erhöht. In jedem Schritt wird dabei der Befehl "squares[i] := i^2;" ausgeführt. Für i=20 wird der Befehl dabei auch ausgeführt.

Man kann eine **for**-Schleife auch alle Elemente einer bestimmten Liste oder Menge durchlaufen lassen.

```
> x := 0;
for i in squares do
    x := x + i;
end do;
x;
```

**Aufgabe:** Bestimme die Zahl  $40! = 1 * 2 * \dots * 40$  mittels einer **for**-Schleife.

```
> 40!;
```

## Für Programmiererfahrene

Mache dich mit der Syntax der **while**-Schleife vertraut:

```
> ?while
```

Außer Schleifen, die bestimmte Befehle mehrfach ausführen gibt es auch noch sogenannte Verzweigungen. Mit ihnen kann man Befehle nur unter bestimmten Voraussetzungen ausführen lassen.

```
> x := 15: y := 0:
if (y <> 0) then    # <> steht hier für "ungleich"
    x := x / y:
else
    x := x+1:
end if:
x;
```

Falls die auf das "if" folgende Bedingung wahr ist, werden die auf "then" folgenden Befehle ausgeführt. Ist die Bedingung falsch, so werden die auf "else" folgenden Befehle ausgeführt. Das "else" und die darauf folgenden Befehle kann man selbstverständlich auch weglassen. Auch das hier wäre eine korrekte Maple-Anweisung:

```
> if (y <> 0) then
    x := x / y;
end if;
```

**freiwillige Aufgabe:** Zähle mithilfe von "for" und "if" wieviele Einsen in der folgenden Liste vorkommen:

```
> L := [ 2,1,3,1,34,13,0,-5,6,1,657,34,-3,1,42 ];
```

Eine sehr elegante Möglichkeit eine bestimmte Funktion auf alle Elemente einer Liste oder Menge anzuwenden ist es den "map" Befehl zu benutzen.

```
> map( x -> x^2, [ $1..20 ] );
```

Mit seiner Hilfe kann man Anweisungen für die man mit for-Schleifen mehrere Zeilen bräuchte eleganter und auch lesbarer in einer einzelnen Zeile formulieren.

Hierbei ist "x -> x^2" die Maple-Kurzschreibweise für die Funktion  $x \rightarrow x^2$ .

```
> x -> x^2;
```

### ▼ Für *sehr* Programmiererfahrene

Falls dich der map-Befehl und seine Verwendung an das Lambda-Kalkül und Funktionen höherer Ordnung erinnert, so liegst du richtig. So kann man dies sogar ziemlich weit treiben:

```
> M := Matrix( 2 , 3 , (i,j) -> i*x^(j-1) );
map( i -> diff( i , x ) , M );# diff( f , x ) leitet die
Funktion f nach x ab
map( diff , M , x );# alle Argumente ab dem dritten werden
von map an das erste Argument als Parameter für das 2te,
3te usw. Argument übergeben
```

**freiwillige Aufgabe:** Erstelle eine Liste von Länge 20, die nur Nullen enthält.

Ähnlich wie der "map"-Befehl, der for-Schleifen ersetzen kann, gibt es auch ein Analagon für if-Abfragen: den "select"-Befehl.

```
> squares;
select( x -> (x < 10), squares );
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256, 289, 324, 361, 400]
```

[1, 4, 9]

(2.4.1)

### ▼ Für Programmiererfahrene

**Aufgabe:** Benutze "map" und "op" um das Kartesische Produkt  $\{1,2\} \times \{1,2,3\}$  zu konstruieren, also die Menge der Tupel  $\{[a,b] \mid a \in \{1,2\}, b \in \{1,2,3\}\}$ .

### ▼ Tipp

[Benutze verschachtelte "map"-Anweisungen.

## Lösung

```
[> map( x -> op( map( y -> [x,y], {1,2,3} ) ), {1,2} );
```

### FREIWILLIGE ÜBUNG [04]:

Löse die folgenden Aufgaben mithilfe von Maple-Befehlen:

- 1.) Wieviele positive Zahlen enthält die unten definierte Liste "L"?
- 2.) Multipliziere die Quadratwurzeln aller positiven Zahlen, die in "L" enthalten sind.

**Hinweis:** Falls du gerne "map" und "select" statt "for" und "if" benutzen möchtest, kannst du mit "nops" die Länge einer Liste und mit "mul" das Produkt aller Elemente einer Liste berechnen lassen.

```
[> L := [ 4,3,-16,9,-25,49 ];
```

```
[> L;
```

## 2.) Aussagenlogische Ausdrücke und ihre Verknüpfungen

[Aufgaben: 2 freiwillige

```
[> restart;# Wir wollen alle Variablen wieder freigeben.
```

### Beispiele von Aussagen und ihren Wahrheitswerten

**MATH/MAPLE:** Ausgangspunkt der klassischen (zweiwertigen) Aussagenlogik sind **Aussagen** mit eindeutig bestimmtem **Wahrheitswert** "wahr" oder "falsch". Eine Aussage muss also entweder wahr oder falsch sein. In Maple schreiben wir dafür **true** oder **false**.

Für uns hat die Aussagenlogik sowohl eine innermathematische Bedeutung bei

- 1) Formulierungen von mathematischen Sätzen,
- 2) Beweisen,
- 3) Beschreibungen von Mengen

als auch eine programmiertechnische Bedeutung bei

- 1) Abfragen im interaktiven Modus,
- 2) Abfragen zur Programmsteuerung.

```
[> evalf( 42*100 / 1337 ) < evalf(Pi);  
3.141361257 < 3.141592654 (3.1.1)
```

In der letzten Zeile steht eine Aussage. MAPLE kann in diesem Fall entscheiden, ob diese Aussage wahr (**true**) oder falsch (**false**) ist. Dazu verwendet MAPLE das Kommando **evalb** ("evaluate boolean", oder auf deutsch "Wahrheitswert auswerten"):

```
[> evalb(%);  
true (3.1.2)
```

Hierbei bezieht sich % auf das zuletzt berechnete Ergebnis.

Die Aussage "42 ist ein Element der Menge {23, 42, 73}" oder in Symbolen " $42 \in \{23, 42, 73\}$ " können wir auch von MAPLE auf den Wahrheitswert überprüfen lassen:

```
> 42 in {23,42,73};  
evalb(%);  
  
42 ∈ {23, 42, 73}  
true
```

(3.1.3)

Hier sehen wir das Ganze für die Aussage "42 ist eine ganze Zahl" oder als Maple-Ausdruck "42 ist ein Objekt vom Typ integer":

```
> 42::integer;  
evalb(%);
```

### Einschränkung in der Auswertung von Aussagen in Maple

Maple kann nur strikte, mathematische Aussagen auswerten. Gewisse Fragen zum Leben, dem Universum und dem ganzen Rest kann MAPLE nicht sinnvoll auswerten:

```
> coulditbe(42 = Number_Of_Roads_A_Man_Must_Walk_Down);  
true
```

(3.2.1)

Wir werden uns in diesem Worksheet auf einfache Aussagen beschränken, welche sowohl wir als auch MAPLE mathematisch sinnvoll beantworten können.

### Zusammensetzung neuer Aussagen durch Verknüpfung

**MATH:** Unser Augenmerk soll jetzt der Konstruktion neuer Aussagen aus vorgegeben Aussagen gelten:

- Aussagen können negiert werden durch **nicht (Negation)**, d.h., ist **A** eine Aussage, so ist auch "nicht **A**", in Symbolen " $\neg A$ ", (ausführlicher: "**A** gilt nicht") eine Aussage.
- Aussagen können miteinander verknüpft werden. Mathematisch besonders wichtig sind Verknüpfungen durch **und (Konjunktion,  $\wedge$ )**, **oder (Disjunktion,  $\vee$ )**, **impliziert (Implikation,  $\Rightarrow$ )** und **äquivalent (Äquivalenz,  $\Leftrightarrow$ )**.

Sind also **A** und **B** Aussagen, so sind auch

"**A** und **B**" ("**A**  $\wedge$  **B**"),

"**A** oder **B**" ("**A**  $\vee$  **B**"),

"Entweder **A** oder **B**",

"**A** impliziert **B**" ("**A**  $\Rightarrow$  **B**", ausführlicher: Wenn **A** gilt, dann gilt auch **B**) und

"**A** ist äquivalent zu **B**" ("**A**  $\Leftrightarrow$  **B**", ausführlicher: **A** gilt genau dann, wenn **B** gilt)

Aussagen. Man beachte, dass hier mit oder das nicht ausschließende oder gemeint ist, im Gegensatz zu entweder oder.

**Wichtigster Punkt:** Der Wahrheitswert einer zusammengesetzten Aussage hängt nur von den Wahrheitswerten der Teilaussagen ab (gemäß der nachfolgenden Tabelle):

<b>A</b>		Nicht <b>A</b>
wahr		falsch
falsch		wahr



A	B	A und B	A oder B	Entw. A oder B	A impliziert B
wahr	wahr	wahr	wahr	falsch	wahr
wahr	falsch	falsch	wahr	wahr	falsch
falsch	wahr	falsch	wahr	wahr	wahr
falsch	falsch	falsch	falsch	falsch	wahr

Diese Tabellen sind so zu lesen: Ist z.B. die Aussage **A** wahr und die Aussage **B** falsch, so ist die Aussage "**A impliziert B**" auch falsch.

**MAPLE:** Anstelle von *nicht*, *und*, *oder*, *entweder oder* und *impliziert* schreiben wir in Maple **not**, **and**, **or**, **xor** und **implies**. Man bezeichnet diese Symbole auch als **Boolesche Operatoren**.

Wir drucken nochmals die Wahrheitstafeln - diesmal durch Maple erzeugt - aus (wie der **printf** Befehl funktioniert ist hierbei nicht relevant).

```
> printf("      a      b | not a | a and b | a or b | a xor
b | a implies b \n");
printf("-----\n");
-----\n");
for a in [true,false] do
  for b in [true,false] do
    printf("%7s%7s | %7s | %7s | %7s | %7s | %11s \n",
          a, b, not a, a and b, a or b, a xor b, a implies
b);
  end do
end do;
```

a	b	not a	a and b	a or b	a xor b	a implies b
true	true	false	true	true	false	true
true	false	false	false	true	true	false
false	true	true	false	true	true	true
false	false	true	false	false	false	true

## Äquivalenz von Aussagen

**MATH:** Aus technischen Gründen nimmt man zu den bisherigen Aussagen noch zwei künstliche Aussagen hinzu. In der mathematischen Logik bezeichnet man diese beiden Aussagen mit **VERUM** (eine immer wahre Aussage) und **FALSUM** (eine immer falsche Aussage). In Maple benutzt man dafür ebenfalls **true** und **false**. Die Übereinstimmung mit den entsprechenden Wahrheitswerten ist unproblematisch. Man mache sich nur klar, dass diese Größen hier in einer Doppelbedeutung (einmal als Aussage und einmal als Wahrheitswert einer Aussage) auftreten. Anschaulich kann sich das Verhältnis von **true** und **VERUM** so ähnlich vorstellen, wie das der Zahl 1 und einer konstanten Funktion die immer den Wert 1 annimmt.

**MATH:** Eine weitere Aussage die immer wahr ist, ist zum Beispiel die Aussage "**a** oder nicht **a**":

```
> printf("      a | a or not a \n");
printf("-----\n");
```

```

for a in [true,false] do
  printf(" %8s | %8s \n",a,a or not a);
end do;

```

a	a or not a
true	true
false	true

Man nennt solche Aussagen **Tautologien**.

**MATH:** Die beiden Aussagen "**a** und **b**" und "**b** und **a**" nehmen in der folgenden Wahrheitstafel immer die selben Werte an. Wir unterscheiden hierbei zwischen diesen beiden Aussagen, fassen diese also nicht als *gleiche* Aussagen auf, sondern nennen sie stattdessen **äquivalente Aussagen**.

```

> printf("          a          b | a and b | b and a \n");
printf(" -----\n");
for a in [true,false] do
  for b in [true,false] do
    printf("%8s%8s | %8s | %8s \n", a, b, a and b, b and a)
  ;
  end do;
end do;

```

a	b	a and b	b and a
true	true	true	true
true	false	false	false
false	true	false	false
false	false	false	false

Man beachte, dass unsere Wahrheitstafel jetzt  $2 \cdot 2 = 4$  Zeilen hat, weil alle Kombinationen der Belegung für **a** und **b** berücksichtigt werden müssen.

**MATH:** Eine Aussage ist also genau dann eine Tautologie, wenn sie äquivalent zu der Aussage **VERUM** ist.

**MATH:** Die Frage, ob zwei (endliche) Aussagen äquivalent sind, ist ein endliches Problem. Das bedeutet, dass man durch Einsetzen von endlich vielen Kombinationen von Wahrheitswerten die Äquivalenz von zwei Aussagen entscheiden kann.

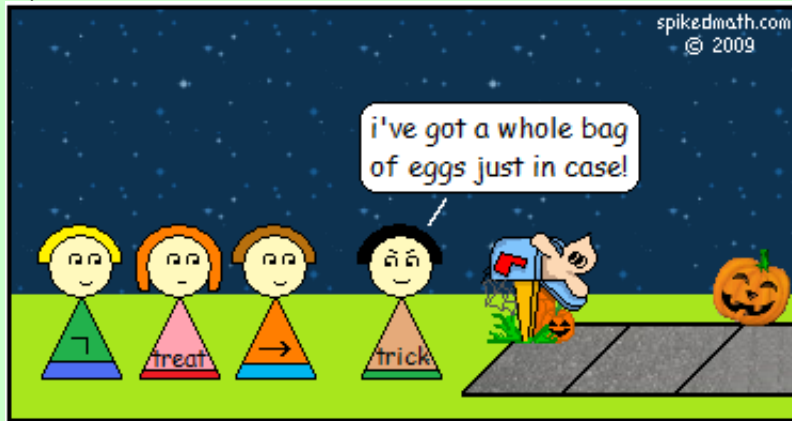
Insbesondere kann man, wie oben geschehen, mithilfe einer Wahrheitstafel beweisen, ob zwei Aussagen äquivalent sind.

### FREIWILLIGE ÜBUNG [01]:

- 1.) Zeige mit Maple, dass "not (**a** or **b**)" (kurz " $\neg(\mathbf{a} \vee \mathbf{b})$ ") und "(not **a**) and (not **b**)" (kurz " $\neg\mathbf{a} \wedge \neg\mathbf{b}$ ") äquivalente Aussagen sind.
- 2.) Zeige mit Maple, dass "not (**a** and **b**)" (kurz " $\neg(\mathbf{a} \wedge \mathbf{b})$ ") und "(not **a**) or (not **b**)" (kurz " $\neg\mathbf{a} \vee \neg\mathbf{b}$ ") äquivalente Aussagen sind.

**Hinweis:** Die korrekte Benutzung des **printf**-Befehls sowie das Aussehen der Tabellen sind hier nebensächlich. Die "**%Ns**" in dem ersten Argument werden der Reihe nach durch die anderen Argumente des Befehls ersetzt. Das "**N**" gibt dabei, wieviel Platz in der Ausgabe eingenommen werden soll.

3.) Erkläre den Witz im Comic:



## Quantoren

**MATH:** Die Aussagenlogischen Operatoren *und* und *oder* haben wichtige Verallgemeinerungen in der sog. Prädikatenlogik. Wollen wir ausdrücken, dass eine bestimmte, von  $x$  abhängige, Aussage  $A(x)$  für alle Elemente  $x$  einer Menge  $M$  gilt, benutzen wir dafür den *All-Quantor*  $\forall$  und erhalten so eine neue Aussage:

$\forall x \in M: A(x)$  (in Worten "für alle  $x$  aus  $M$ :  $A(x)$  gilt").

Ein wichtiger Spezialfall ist hierbei, dass wir - unabhängig von der Teilaussage  $A(x)$  - obige Aussage als "wahr" auffassen, falls die Menge  $M$  leer ist.

Wollen wir ausdrücken, dass eine bestimmte Aussage  $A(x)$  für (mindestens) ein Objekt  $x$  gilt, benutzen wir dafür den *Existenz-Quantor*  $\exists$  :

$\exists x \in M: A(x)$  (in Worten "es existiert ein  $x$  aus  $M$  mit:  $A(x)$  gilt").

Im Gegensatz zu der obigen Situation ist obige Aussage "falsch", falls die Menge  $M$  leer ist.

**MATH:** Die ersten beiden Aussagen der letzten Übung lassen sich dementsprechend ebenfalls in der Prädikatenlogik verallgemeinern: Die Verneinung der Aussage

$\forall x \in M: A(x)$

ist

$\exists x \in M: \neg A(x)$  (oder gebräuchlicher in der Umgangssprache "es gibt ein  $x$ , so dass  $A(x)$  nicht gilt").

Falls  $x$  nur eine zweielementige Menge durchläuft, wird die erste Aussage zu einer gewöhnlichen *und*-Verbindung, während die zweite in diesem Fall zu einer *oder*-Verbindung der Verneinungen wird.

Entsprechend ist die Verneinung der Aussage

$\exists x \in M: A(x)$

die Aussage

$\forall x \in M: \neg A(x)$

**DENKANSTOSS:** Wenn  $x$  eine dreielementige Menge, etwa  $\{1, 2, 3\}$ , durchläuft, dann bedeutet

für alle  $x$  aus  $M$ :  $a(x)$  gilt

dasselbe wie

$a(1)$  und  $a(2)$  und  $a(3)$ .

Was ist bei  $n$ -elementigen Mengen? Wie formuliert sich

es existiert ein  $x$  aus  $M$ :  $a(x)$  gilt

in der alten Sprache mit "oder"?

### FREIWILLIGE ÜBUNG [02]:

(Ohne Maple, dafür aber mit Papier und Schreibgerät):

1.) Verneine die Aussage: "Es existiert ein  $x$ , so dass für jedes  $y$  die Aussage  $a(x,y)$  gilt" (kurz " $\exists x: \forall y: a(x,y)$ ").

2.) Verneine die Aussage: "Für jedes  $x$  gibt es ein  $y$ , so dass  $a(x,y)$  gilt" (kurz " $\forall x: \exists y: a(x,y)$ ").

Schreibe die Verneinung jeweils umgangssprachlich und in Symbolen hin.

Hinweis: Klammern setzen und die Regel aus dem letzten MATH-Abschnitt zweimal verwenden.

## 3.) Algorithmen und Programme

Aufbauend auf: "Aussagenlogische Ausdrücke und ihre Verknüpfungen"

Aufgaben: 4 freiwillige

`> restart;`

### Algorithmen und Programme in Maple

Nun wollen wir unsere eigenen Programme erstellen. Eine Kurzschreibweise um einfache Programme zu definieren hast du schon kennengelernt: "`x -> x^2`".

```
> f := x -> x^2;  
f(3);
```

Maple nennt Programme und Funktionen übrigens "Prozeduren". Die volle Syntax um Prozeduren zu definieren lautet wie folgt (wobei die Teile inklusive der eckigen Klammern zu ersetzen sind):

```
prozedurname := proc( [Eine Folge von Argumenten die übergeben  
wird] )  
# . . . Befehle . . .  
return [Der Wert den die Prozedur zurückgibt];
```

`end proc:`

Du bist nun schon in der Lage selbst ein kleines Programm zu schreiben.

### FREIWILLIGE ÜBUNG [01]:

- 1.) Definiere eine Prozedur `"Hello()"` ohne Argumente, die nichts weiteres tut als `"Hello World"` zurückzugeben. Achte darauf die roten Anführungsstriche mit einzugeben.
- 2.) Rufe `"Hello()"` auf.

**MATH:** Ein **Algorithmus** ist ein Rechenverfahren, welches in *endlich* vielen Schritten für jeden Datensatz bestimmter Spezifikation ein wohldefiniertes Resultat liefert. Die Anzahl der Schritte kann sehr wohl von den Eingabedaten abhängen. Ein **Programm** ist die Umsetzung eines Algorithmus im Computer, mit dessen Hilfe bei Eingabe von Daten das Ergebnis berechnet wird.

Hier ein einfaches Beispiel: Wir wollen  $n! := 1 \cdot 2 \cdot \dots \cdot n$  ausrechnen.

#### Algorithmus Fak:

Eingabedaten: eine nichtnegative ganze Zahl  $n$

Ausgabedaten:  $n!$

Rechenschritte:

Wenn  $n = 0$  gilt, dann gib 1 aus.

Ansonsten gib  $1 \cdot 2 \cdot \dots \cdot n$  aus.

Programm = Implementation des Algorithmus:

```
> Fak := proc( n::nonnegint )
  local result, i;
  result := 1;
  for i in seq( 1..n ) do
    result := result * i;
  end do;
  return result;
end proc;
```

Das Stichwort `"local"` sorgt übrigens dafür, dass die Variablen `"result"` und `"i"` nur innerhalb der Prozedur `"Fak"` verwendet werden. Falls wir die `"local"`-Zeile weglassen würden, würde z.B. die Variable `"result"` bei jedem Aufruf von `"Fak"` überschrieben werden. Hätten wir in `"result"` vorher andere Daten gespeichert, wäre dies natürlich äußerst unpraktisch.

```
> Fak(3);
```

1 (4.1.1)

```
> seq( Fak(k), k=0..20 );
```

1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800, 39916800, 479001600, (4.1.2)  
6227020800, 87178291200, 1307674368000, 20922789888000, 355687428096000,  
6402373705728000, 121645100408832000, 2432902008176640000

```
> Fak(-1);
```

```
Error, invalid input: FAK expects its 1st argument, n, to be  
of type nonnegint, but received -1
```

Für weitere Integer-Typen kann man übrigens folgende Hilfe-Seite zurate ziehen:

```
> ?nonnegint
```

### FREIWILLIGE ÜBUNG [02]:

Definiere eine Prozedur "**Konkatenation**( **a::list**, **b::list** )" die die Verkettung der zwei Listen "a" und "b" zurückgibt. Zum Beispiel wäre:

```
Konkatenation( [1,2], [3,4] ) = [ 1,2,3,4 ]
```

**Hinweis:** Schau dir gegebenenfalls nochmal den Abschnitt zu den Datenstrukturen in Maple an.

## Rekursion

**Rekursion** werden wir hauptsächlich in zwei Zusammenhängen antreffen:

- 1) Eine mathematische Funktion oder eine Eigenschaft kann **rekursiv** definiert sein. Das heißt, dass sie in ihrer Definition auf ("kleinere" Instanzen von) sich selbst verweist.
- 2) Ein Algorithmus kann ebenfalls **rekursiv** sein. Das bedeutet, dass er sich selber wiederholt aufruft, um ein kompliziertes Problem in kleinere Teilprobleme zu zerlegen.

Sie ist ein beliebtes Mittel um komplexe Sachverhalte elegant zu beschreiben. Ein beliebtes Beispiel um Rekursion zu veranschaulichen ist die Folge sogenannter Fibonacci-Zahlen.

```
> 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144;
```

```
1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144
```

(4.2.1)

Leonardo da Pisa, auch Fibonacci genannt, soll über folgendes Problem nachgedacht haben:

Ein neugeborenes Paar Kaninchen wird in einem Feld ausgesetzt. Im zweiten Monat werden beide Kaninchen geschlechtsreif. Damit leben im dritten Monat zwei Kaninchenpaare auf dem Feld: ein ausgewachsenes und ein neugeborenes. Angenommen jedes Kaninchen lebt ewig und jedes Kaninchenpaar bringt nach dem Erreichen der Geschlechtsreife jeden Monat ein neues Paar zur Welt. Wieviele Kaninchen leben auf dem Feld nach Ablauf eines Jahres?

Im vierten Monat leben damit 3 Paare auf dem Feld: das Paar aus dem dritten Monat ist nun geschlechtsreif geworden und das erste Paar hat ein neues Paar zur Welt gebracht. Im fünften Monat sind es 5 etc.

In jedem Monat kommen also soviele neue Paare hinzu, wie zwei Monate zuvor auf dem Feld gelebt haben, da diese nun ausgewachsen sind. Diese Fibonacci-Folge unterliegt also folgendem Rekursionsprinzip:

$$a_1 := 1; a_2 := 1;$$

$$a_{n+2} := a_{n+1} + a_n$$

Wir erhalten damit für die Berechnung der Fibonacci-Zahlen einen einfachen rekursiven Algorithmus. Es wäre allerdings ziemlich kompliziert diesen Algorithmus mithilfe von **for**-Schleifen und ohne Rekursion zu beschreiben.

### Algorithmus Fib:

Eingabedaten: Eine positive ganze Zahl  $n$

Ausgabedaten: Die  $n$ -te Fibonacci-Zahl

Rechenschritte:

Wenn  $n = 1$  oder  $n = 2$  gilt, dann gib 1 aus.

Ansonsten gib  $Fib(n - 1) + Fib(n - 2)$  aus.

```
> Fib := proc( n::posint )
    if n=1 then
        return 1;
    end if;
    if n=2 then
        return 1;
    end if;
    return Fib(n-1) + Fib(n-2);
end proc;
```

**Hinweis:** Der Programmablauf endet sofort, falls ein **return** Befehl ausgeführt wird. Die Befehle in den darauf folgenden Zeilen des Programms werden dann nicht mehr ausgeführt.

```
> Fib(1);
Fib(2);
Fib(3);
```

1

1

2

(4.2.2)

```
> seq( Fib(x), x=1..12 );
```

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144

(4.2.3)

**HINWEIS:** Die Rekursion ist kein triviales Konzept, insbesondere wenn man noch nie programmiert hat. Bitte gegebenenfalls die Hiwis das Prinzip der Rekursion noch einmal mit dir zu besprechen. Wer das Prinzip der Rekursion nicht kennt und deshalb nicht versteht, wird in den Testaten der folgenden Worksheets Probleme haben.

### FREIWILLIGE ÜBUNG [03]:

1) Führe den Programmaufruf "**Fib(4)**" von Hand - also mit Stift und Papier - aus, um ein Gefühl für die Rekursion zu bekommen. Du kannst dir den Ablauf zum Beispiel als Baum veranschaulichen.

2) Implementiere eine **rekursive** Version "**Fak2**" des Programms "**Fak**".

### Quersumme als Beispiel eines Programms

#### FREIWILLIGE ÜBUNG [04]:

Implementiere ein **rekursives** Programm "**Quer**", das die Quersumme einer nicht-negativen ganzen Zahl berechnet und versieh es mit Kommentaren. Die Maple-Befehle "**mod**" und "**floor**" sind dabei hilfreich.