WS 2020/21

Begleitpraktikum I

Blatt 0 – Einführung

Hinweise

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.

Am Ende der meisten Abschnitte findest Du kurze Übungen mit deren Hilfe Du überprüfen kannst, ob Du den entsprechenden Stoff (schon) beherrschst.

Der erste Teil vermittelt alle grundlegenden Programmierkonzepte die Du zum Bestehen des Praktikums benötigst. Der zweite Teil behandelt die Aussagenlogik und ihre Erweiterung um Quantoren. Im dritten Teil erfährst Du, 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 die folgenden Befehle 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 CTRL-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 werden 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 CTRL-S (bzw. STRG-S auf einer deutschen Tastatur).

Hinweis. 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".

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

Freiwillige Übung 1.1

(i) Welches Zeichen steht in Maple für Kommentare?

```
>
```

(ii) 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;
```

$$a := \frac{5}{12}$$

$$a := \frac{5}{3} \quad \textbf{(1)}$$

Maple unterscheidet dabei zwischen verschiedenen Typen und rechnet beispielsweise mit Bruchzahlen, ohne sie in Gleitkommazahlen umzuwandeln.
Dies nennt man auch *Symbolisches Rechnen* und es ermöglicht 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 );
```

integer

fraction

float **(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)  );
```

$$\frac{10}{7}$$

$$\sqrt{2}$$

$$1.428571429$$

$$1.414213562 \quad \textbf{(3)}$$

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 );
```

$$\cos(x)^2 + \sin(x)^2$$

$$1 \quad \textbf{(4)}$$

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. Ebensowenig 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);
```

symbol **(5)**

Freiwillige Übung 1.2

(i) Welche Zahl ist größer: $\sqrt{2}$ oder $\frac{30547}{21600}$?

```
>
```

(ii) 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);
```

$L := [3, 1, 4, 1]$

list **(6)**

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

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

1 **(7)**

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 nächstem Sheet aus Sicht der Mathematik betrachten.

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

$$M := \{1, 3, 4\}$$

$$set$$

$$3$$ **(8)**

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

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

$$3$$

$$3$$ **(9)**

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

Freiwillige Übung 1.3

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
```

$$1, 2, [3, 4], x = x$$

$$exprseq$$ **(10)**

Ein nützlicher Befehl zur Erstellung von Folgen ist **seq** (für sequence). Diese Folgen 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 ) ];
```

$$5, 6, 7, 8, 9, 10$$

$$10, 15, 20$$

$$5, 6, 7, 8, 9, 10, 10, 15, 20$$

(11)

$$A := [5, 6, 7, 8, 9, 10, 10, 15, 20] \quad \textbf{(11)}$$

```
> ?seq
```

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

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

$$5, 6, 7, 8, 9, 10$$

$$5, 6, 7, 8, 9, 10 \quad \textbf{(12)}$$

Freiwillige Übung 1.4

(i) Erstelle eine Liste, die alle ungeraden Zahlen zwischen 40 und 80 enthält.

```
>
```

(ii) 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;
```

$$A := [5, 13, 7, 8, 9, 10, 10, 15, 20] \quad \textbf{(13)}$$

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);
```

$$5, 13, 7, 8, 9, 10, 10, 15, 20 \quad \textbf{(14)}$$

Freiwillige Übung 1.5

(i) Füge hinten an die Liste `A` die Zahl 0 hinzu. Verwende dabei `op`.

```
>
```

(ii) 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;
```

$$squares := [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]$$

$$[1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256, 289, 324, 361, 400] \quad \textbf{(15)}$$

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;
```

$$x := 0$$

$$2870 \quad \textbf{(16)}$$

Freiwillige Übung 1.6

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

```
>
```

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 für "ungleich"
    x := x / y:
  else
    x := x+1:
  end if:
  x;
```

$$16 \quad \textbf{(17)}$$

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 Übung 1.7

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 ];
```

$$L := [2, 1, 3, 1, 34, 13, 0, -5, 6, 1, 657, 34, -3, 1, 42] \quad \textbf{(18)}$$

```
>
```

Eine sehr elegante Möglichkeit, eine bestimmte Funktion auf alle Elemente einer Liste oder Menge anzuwenden, ermöglicht der Befehl **map**.

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

$$[1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256, 289, 324, 361, 400] \quad \textbf{(19)}$$

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 $x \to x^2$.

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

$$x \mapsto x^2 \quad \textbf{(20)}$$

Für sehr Programmiererfahrene

Falls Dich der **map**-Befehl und seine Verwendung an das Lambda-Kalkül und Funktionen höherer Ordnung erinnert, so liegst Du richtig. Man kann dies sogar ziemlich weit treiben, wie das folgende Beispiel demonstriert.

```
> x := 'x':
  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
```

$$M := \begin{bmatrix} 1 & x & x^2 \\ 2 & 2x & 2x^2 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 1 & 2x \\ 0 & 2 & 4x \end{bmatrix}$$

$$\begin{bmatrix} 0 & 1 & 2x \\ 0 & 2 & 4x \end{bmatrix} \quad \textbf{(21)}$$

Freiwillige Übung 1.8

Erstelle eine Liste der Länge 20, die nur Nullen enthält.

```
>
```

```
>
```

Analog zum **map**-Befehl, der **for**-Schleifen ersetzen kann, gibt es 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] **(22)**

Freiwillige Übung 1.9

Beantworte die folgenden Fragen über die Liste

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

mithilfe von Maple-Befehlen.

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.

(i) Wieviele positive Zahlen enthält die Liste **L**?

```
>
```

(ii) Multipliziere die Quadratwurzeln aller positiven Zahlen, die in **L** enthalten sind.

```
>
```

2 Aussagenlogische Ausdrücke und ihre Verknüpfungen

Aufgaben: 2 freiwillige

```
> restart; # Wir wollen alle Variabeln wieder freigeben.
```

Beispiele von Aussagen und ihren Wahrheitswerten

Mathematik & 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
- Formulierungen von mathematischen Sätzen,
- Beweisen und
- Beschreibungen von Mengen,

als auch eine programmiertechnische Bedeutung bei
- Abfragen im interaktiven Modus und
- Abfragen zur Programmsteuerung.

```
> evalf( 42*100 / 1337 ) < evalf(Pi);
```

3.141361257 < 3.141592654 **(23)**

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 **(24)**

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(%);
```

$$42 \in \{23, 42, 73\}$$

true **(25)**

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(%);
```

$$42::\mathbb{Z}$$

true **(26)**

Einschränkung in der Auswertung von Aussagen in Maple

Maple. Maple kann nur strikte, mathematische Aussagen auswerten. Gewisse Fragen zum Leben, dem Universum und dem ganzen Rest wird Maple uns leider nicht beantworten können.

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

true **(27)**

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

Mathematik. 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. mathematischmatisch 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*.

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

A	**Nicht A**
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 wie folgt 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 aus – diesmal durch Maple erzeugt. (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");
  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

Mathematik. 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.

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**.

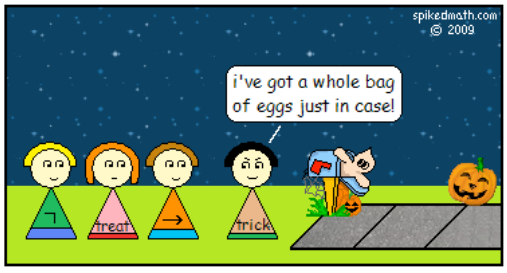
Die beiden Aussagen "**a** und **b**" und "**b** und **a**" nehmen in der folgenden Wahrheitstafel immer die selben Werte an. Wir unterscheiden hierbei zwischen diesen beiden Aussagen, fassen diese also nicht als *gleiche* Aussagen auf, sondern nennen sie stattdessen **äquivalente Aussagen**.

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

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.

Zum Abschluss des Themas, hier noch ein *comic relief*.



Freiwillige Übung 2.1

Zeige die folgenden Äquivalenzen mithilfe von Maple.

• "not (**a** or **b**)" (kurz $\neg($**a** $\vee$ **b**$)$) ist äquivalent zu "(not **a**) and (not **b**)" (kurz $\neg$**a** $\wedge$ $\neg$**b**).
• "not (**a** and **b**)" (kurz $\neg($**a** $\wedge$ **b**$)$) ist äquivalent zu "(not **a**) or (not **b**)" (kurz $\neg$**a** $\vee$ $\neg$**b**).

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.

```
>
```

Quantoren

Mathematik. Die aussagenlogischen Operatoren *und* und *oder* haben wichtige Verallgemeinerungen in der sogenannten *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 die neue Aussage
$\forall$ x $\in M : A(x)$ (in Worten "für alle x aus M gilt $A(x)$").
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* $\exists$ und erhalten
$\exists\, x \in M : A(x)$ (in Worten "es existiert ein x aus M, so dass $A(x)$ gilt").
Im Gegensatz zum All-Quantor ist diese Aussage *falsch*, falls die Menge M leer ist.

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 die Aussage
$\exists\, x \in M : \neg\, A(x)$ (oder gebräuchlicher in der Umgangssprache "es gibt ein x in M, für das $A(x)$ 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
$\forall\, x \in M : \neg\, A(x)$.

Freiwillige Übung 2.2

Verneine die folgenden Aussagen (jeweils umgangssprachlich und in Symbolen.

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

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

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

3 Algorithmen und Programme

Aufbauend auf: "Aussagenlogische Ausdrücke und ihre Verknüpfungen"

Aufgaben: 4 freiwillige

```
> restart;
```

Algorithmen und Programme in Maple

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);
```

$$f := x \mapsto x^2$$

$$9 \qquad \textbf{(28)}$$

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 3.1

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

```
>
```

(ii) Rufe `Hello()` auf.

```
>
```

Mathematik. 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 $n! := 1 \cdot 2 \cdot \ldots \cdot n$ ausrechnen.

Algorithmus Fak
Eingabedaten: eine nichtnegative ganze Zahl n.
Ausgabedaten: $n!$
Rechenschritte:
Wenn $n = 0$ gilt, dann gib 1 aus.
Ansonsten gib $1 \cdot 2 \cdot \ldots \cdot n$ aus.

Maple. Eine Implementierung dieses Algorithmus findet sich in der folgenden Prozedur.

```
> 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);
```

6 **(29)**

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

1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800, 39916800, 479001600, 6227020800, 87178291200, 1307674368000, 20922789888000, 355687428096000, 6402373705728000, 121645100408832000, 2432902008176640000 **(30)**

```
> 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 3.2

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. Schaue Dir gegebenenfalls nochmal den Abschnitt zu den Datenstrukturen in Maple an.

```
>
```

Rekursion

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

Rekursionen sind ein beliebtes Mittel, um komplexe Sachverhalte elegant zu beschreiben. Ein typisches Beispiel sind die *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 **(31)**

Leonardo da Pisa, auch Fibonacci genannt, soll über das folgende Problem nachgedacht haben.

Ein neugeborenes Paar Kaninchen wird in einem Feld ausgesetzt. Im zweiten Monat werden beide Kaninchen geschlechtsreif (und tun das Erwartete). Damit leben im dritten Monat zwei Kaninchenpaare auf dem Feld – ein ausgewachsenes und ein neugeborenes Paar. Angenommen jedes Kaninchen würde ewig leben und nach dem Erreichen der Geschlechtsreife jeden Monat ein neues Paar zur Welt bringen, wieviele Kaninchenpaare würden nach Ablauf eines Jahres auf dem Feld leben?

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

Maple. Wir können diesen Algorithmus wie folgt realisieren.

```
> 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 **(32)**

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

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144 **(33)**

Freiwillige Übung 3.3

(i) 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.

(ii) Implementiere eine **rekursive** Version `Fak2` des Programms `Fak`.

```
>
```

Quersumme als Beispiel eines Programms

Freiwillige Übung 3.4

Implementiere ein **rekursives** Programm `Quer`, das die Quersumme einer nicht-negativen ganzen Zahl berechnet und versieh es mit Kommentaren.

Hinweis. Die Maple-Befehle `mod` oder `floor` könnten nützlich sein.

```
>
```