■ *Mathematica*-Programme

Zum Schluss dieser Einleitung als Beispiele vier *Mathematica*-Programme. Wir werden gleich sehen wie es auch einfacher geht. Zuerst die Programmtexte, dann die Kurzfassungen mit speziellen Aufrufen aus *Mathematica* und schließlich die Kommentierungen und Erläuterungen der Texte.

Beispiel A:

```
In[1]:= exprod1[n_Integer] :=     (* zum baldigen Vergessen *)
         Module[ { t, k } ,       (* lokale Variablen *)
             t = 1 ;
             For [ k = 1, k <= n, k++, t = (x + k ) t ];
             t = Expand[t];
        (* wird weiter unten erklärt *)
              Return[t];
                 ]
```

Ein prozedurales Programm zur Entwicklung eines Produktes in Summanden.

Beispiel B:

HERONverfahren zur Berechnung von $\sqrt{2}$:

Die Schleife soll 5-mal durchlaufen werden.

```
In[2]:= s = 1.; (* Startwert 1. *)
        Do[s = 1 / s + s / 2, {5}]; s
Out[3]= 1.41421
```

Beispiel C:

HERONverfahren eleganter:

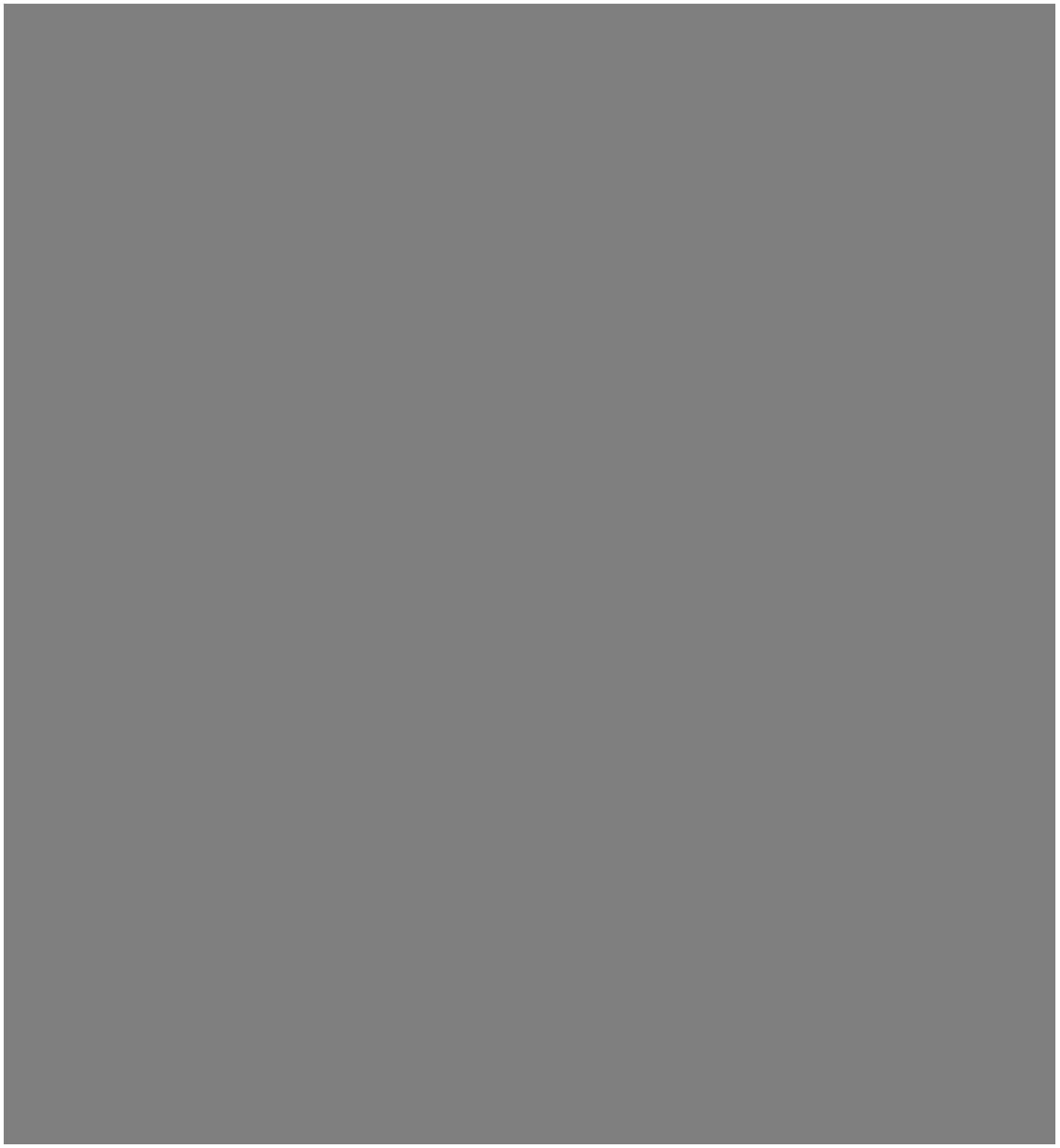
```
In[4]:= FixedPoint[ 1 / # + # / 2 &, N[1, 6] ]
Out[4]= 1.41421
```

http://demonstrations.wolfram.com/SquareRootsWithNewtonsMethod/

http : // demonstrations.wolfram.com/LearningNewtonsMethod/

In[6]:=
```
Manipulate[
 Pane[
  Text@
   Column@Block[{$MinPrecision = precision,
      $MaxPrecision = precision},
     NestList[# - (#^2 - n) / (2 #) &,
      SetPrecision[n, precision], howmany]],
  ImageSize → {425, 250}],
 {{n, 2, "square root of"},
  Select[Range[100], ! IntegerQ[Sqrt[#]] &]},
 {{howmany, 6, "number of iterations"}, Range[1, 12]},
 {{precision, 50, "number of decimal places"},
  {5, 10, 20, 50}}]
```

Out[6]=



Beispiel D:

Beispiel A wird mit *Mathematica* eine Zeile:

```
In[5]:= exprod2[n_Integer] := Expand[ Product[ x + k, {k, 1, n}] ]
```

Ein funktionales Programm zur Entwicklung eines Produktes in Summanden.

Zu Beispiel A:

Dieses Beispiel soll erst einmal das Gefühl vermitteln, dass in *Mathematica* ähnlich wie in Pascal oder C programmiert werden kann: Mit *Mathematica* geht es viel besser!

`exprod1` ist der Programmname; er könnte z.B. auch `expandProduct` lauten.

`n_Integer` : Nur ganzzahlige Werte der Variable n werden akzeptiert.

`(* (* hier steht ein *) Kommentar *)`

" ; " trennt, wie gewohnt, die einzelnen Befehle.

`Module[ {t, k},  ]` : für dieses Modul werden die lokalen Variablen t, k reserviert und danach die folgenden Ausdrücke ausgewertet. Die eckigen Klammern [ ] spielen hier jetzt eine ähnliche Rolle wie `begin - end` in Pascal oder { } in C .

`Expand[t]` entwickelt das Produkt t als Ausdruck in eine Summe nach Potenzen.

`Return[t]` gibt t aus.

```
In[7]:= Clear[x]; t = (x + 1) (x + 2); Expand[t]
Out[7]= 2 + 3 x + x^2
```

Zu Beispiel B:

`Do[ expression, { n } ]` wertet *expression* *n*-fach aus.

Zu Beispiel C:

Hier beginnt keine undurchsichtige C-Programmiererei !

`#` ist in *Mathematica* ein Zeichen für eine Variable, die keinen Namen trägt. `&` nach einem Ausdruck fasst diesen Ausdruck zu einer Funktion zusammen. `#` wird *slot* genannt. Das Konzept der *pure function* ist aus der Sprache Lisp übernommen.

`1/# + #/2&` entspricht dem Ausdruck `Function[ x, 1/x + x/2 ]`, oder wie Sie vielleicht gewohnt sind: $x \mapsto \frac{1}{x}+\frac{x}{2}$.

Zu Beispiel D:

`Product[ x + k, {k, 1, n} ]` entspricht $\prod_{k=1}^{n}(x+k)$, also eine sehr naheliegende Umsetzung.

```
In[8]:= Product[ x + k, {k, 1, 4} ]
Out[8]= (1 + x) (2 + x) (3 + x) (4 + x)

In[9]:= exprod2[4]
Out[9]= 24 + 50 x + 35 x^2 + 10 x^3 + x^4
```

Für verschiedene n, etwa für $n = 500$, können Sie das Verhalten der Funktionen testen:

```
In[10]:= {Timing[ exprod1[n = 500]][[1]], Timing[exprod2[n]][[1]] }
Out[10]= {0.969, 0.688}
```

In der neuen *Mathematica*-Version sind die Zeitunterschiede nicht mehr so groß.

■ Funktionen

Im Handbuch zu *Mathematica* werden als Beispiele häufig die “Quadratfunktion”, die Fakultät oder eine Logarithmusfunktion benutzt. Die Bedeutung von “ := ” wird weiter unten kurz erklärt. Teilweise haben wir solche Beispiele schon benutzt. Jetzt eine instruktive Spielerei mit quasi-trivialen background. Faites votre jeu! Welche Funktionen werden da definiert?

```
In[1]:= Clear[f];
        f[0] = 0; f[1] = 1;
        f[ n_Integer ] := f[1] + f[n - 1] /; n > 1
```

n soll dabei nur ganzzahlig positiv zugelassen sein!

In[3]:=
```
f[3]
```
Out[3]=
```
3
```

In[4]:=
```
f[33]
```
Out[4]=
```
33
```

In[5]:=
```
f[-3]
```
Out[5]=
```
f[-3]
```

In[6]:=
```
f[333]
```

$RecursionLimit::reclim: Recursion depth of 256 exceeded. >>

Out[6]=
```
254 + Hold[RuleCondition[
    $ConditionHold[$ConditionHold[f[1] + f[79 - 1]]], 79 > 1]]
```

Hold

Hold[*expr*] maintains *expr* in an unevaluated form. >>

In[7]:=
```
254 + f[1] + f[79 - 1]
```
Out[7]=
```
333
```

In[8]:=
```
f[1 / 3]
```
Out[8]=
$$f\left[\frac{1}{3}\right]$$

In[9]:=
```
Clear[f]
```

In[10]:=
```
f[0] = 0; f[1] = 1;
f[ n_Integer ] :=
 f[1] + 100 Sign[n - 1] * f[ Abs[ Quotient[ n - 1, 100]]] +
  f[ Mod[ n - 1, 100]]
```

In[11]:=
```
f[1111]
```
Out[11]=
```
1111
```

In[12]:=
```
f[-11]
```
Out[12]=
```
-11
```

In[13]:=
```
f[1 / 2]
```
Out[13]=
$$f\left[\frac{1}{2}\right]$$

```
In[14]:= f[ q_Rational ] := f[ Numerator[q] ] / f[ Denominator[q] ]

In[15]:= f[1 / 2]
Out[15]= 1/2

In[16]:= f[64 / 46]
Out[16]= 32/23

In[17]:= f[ x_Real ] := f[ Rationalize[ x, 10^-6] ]

In[18]:= f[3.14159292]
Out[18]= 355/113

In[19]:= f[π]
Out[19]= f[π]

In[20]:= f[N[π]]
Out[20]= 355/113
```

π ist ein Symbol; `N[π]` ist `Real`.

```
In[21]:= Definition[f]
Out[21]= f[0] = 0

         f[1] = 1

         f[n_Integer] :=
          f[1] + 100 Sign[n - 1] f[Abs[Quotient[n - 1, 100]]] +
           f[Mod[n - 1, 100]]

         f[q_Rational] := f[Numerator[q]] / f[Denominator[q]]

         f[x_Real] := f[Rationalize[x, 1/10^6]]
```

Geben Sie einen einfachen Ausdruck an, der f gut approximiert!

Bei der Definition von Funktionen in *Mathematica* wird deutlich, wieviele kontextabhängige Konventionen in der Mathematik gebräuchlich sind und meist beachtet werden. In einem

formalisierten System ist Kontextabhängigkeit jedoch äußert problematisch. $f(a) = b$ ist für MathematikerInnen klar die Formel für: Die Funktion f hat bei dem konstanten Argument a den konstanten Wert b. Was unterscheidet aber formal $f(x) = y$ von $f(a) = b$?

In *Mathematica* ist `f[a] = b` eine Ersetzungsregel. Immer, wenn der spezielle Ausdruck $f[a]$ auftaucht, wird er durch b ersetzt. Sollen jedoch für beliebige Ausdrücke x die Werte $f(x)$ angegeben werden, so besitzt *Mathematica* das *Muster* (engl. *pattern*) `x_`, um dies zu ermöglichen. Muster stehen für Klassen von Ausdrücken. Das Muster `x_` steht für einen beliebigen Ausdruck und gibt ihm den Namen x.

`f[x]` = *wert*	Definition für einen *speziellen Ausdruck* x.
`f[x_]` = *wert*	Definition für einen *beliebigen Ausdruck*, der x genannt wird.

Lesen Sie im Immediate and Delayed Definitions !

Dort finden Sie insbesondere den folgenden Text als Merkregel:

As you can see from the example above, both = and := can be useful in defining functions, but they have different meanings, and you must be careful about which one to use in a particular case.

One rule of thumb is the following. If you think of an assignment as giving the final "value" of an expression, use the = operator. If instead you think of the assignment as specifying a "command" for finding the value, use the := operator. If in doubt, it is usually better to use the := operator than the = one.

Es kann manchmal sinnvoll sein — insbesondere bei rekursiven Definitionen — schon einmal berechnete Funktionswerte zu speichern.

```
In[22]:= f[0] = f[1] = 1; f[x_] := f[x] = f[x - 1] + f[x - 2]
```

FIBONACCI läßt grüßen: `f[x_]:= f[x] = ...` bewirkt diese Speicherung.

In unserem obigen längeren Beispiel kamen Ausdrücke wie `f[n_Integer]`, `f[x_Real]` vor. Es ist oft nötig, Muster einzuschränken, also im mathematischen Sinne Definitionsbereiche anzugeben. Im Versuch der letzten Definition von f wäre es nicht gut gewesen etwa `In[ ]:= f[-1]` einzugeben. Die Abbruchbedingung `f[0]` läge jenseits von Gut und Böse. Um das abzufangen, stehen folgende Möglichkeiten bereit:

```
In[23]:= Clear[ f ];
         f[0] = 0;
         f[1] = 1;
         f[ n_Integer ] := f[1] + f[n - 1] /; n > 0

In[24]:= f[-1]
Out[24]= f[-1]
```

Etwas umständlicher geht das so: Dabei steht `&&` für logisches “und”:

```
In[25]:= Clear[ f ] ; f[0] = 0; f[1] = 1;
         f[n_] := f[1] + f[n - 1] /; IntegerQ[n - 1] && Positive[n - 1]
```

Damit ist dieser Fall abgesichert:

```
In[27]:= f[-1]
Out[27]= f[-1]
```

Wenn es in *Mathematica* die Signumfunktion `Sign` nicht gäbe, könnte sie so definiert werden:

```
In[28]:= sign[x_] := -1 /; x < 0;
         sign[x_] :=  1 /; x > 0;
         sign[0]   =  0
Out[30]= 0
```

Weitere Möglichkeiten bietet der Ausdruck `Piecewise` .