Funktionen; Rekursion

```
> restart;
```

Der Operator ->

Mit dem `->` -Operator definiert man eine Funktion (Abbildung).
'`->`' hat (zunächst) einen Namen (den *Formalparameter*) als linken und einen Ausdruck als rechten Operanden.
Das ganze sieht dann so aus, wie wir es aus der Mathematik gewohnt sind:

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

$$x \to x^2$$

Eine Funktion kann man auswerten; wir schreiben das Argument in runden Klammern hinter die Funktion.
Der Wert (`4`) heißt hier *Aktualparameter*, er ersetzt beim Auswerten den Formalparameter `x` in der Formel:

```
> (x->x^2)(4);
```

$$16$$

Dass das letzte Beispiel komisch aussah, lag nur daran, dass die Funktion noch keinen Namen hat.
Durch Zuweisen an eine Variable können wir dem abhelfen:

```
> f := x -> x^2;
> f(4);
```

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

$$16$$

Die Auswertung funktioniert ähnlich wie das `subs()` letzte Woche - zumindest bei diesem Beispiel:

```
> g := x^2;
> subs(x=4, g);
```

$$g := x^2$$

$$16$$

```
>
```

Mehrere Parameter

Wenn man mehr als einen Parameter hat, trennt man sie durch Kommas und macht runde Klammern drum:

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

$$g := (x, y) \to x^2 - y^2$$

```
> g(2);
```

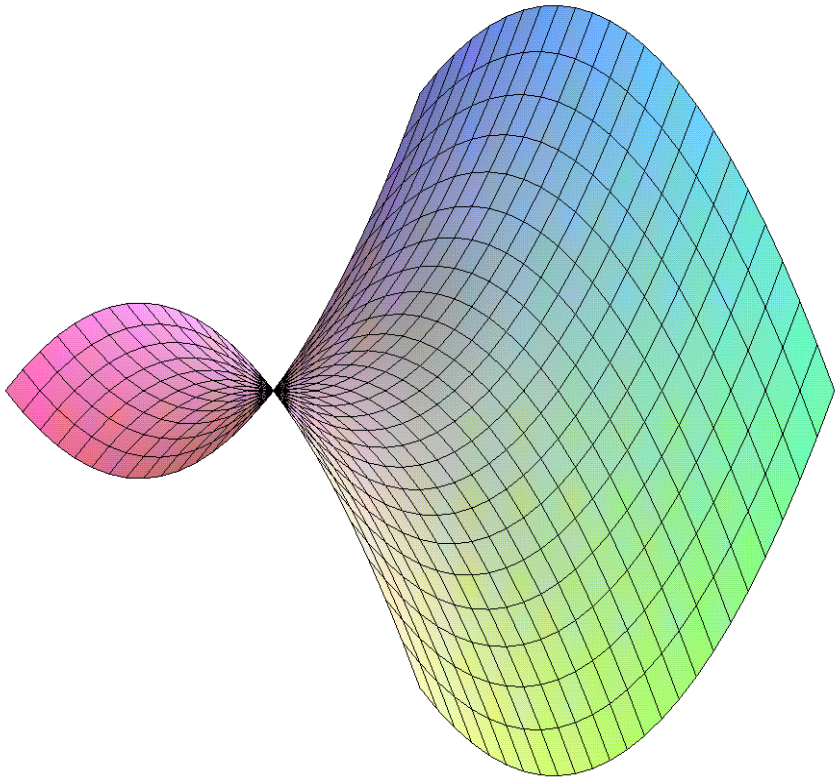
```
Error, (in g) g uses a 2nd argument, y, which is missing
```

```
> g(2,3);
```

$$-5$$

Mit der Funktion `plot3d` erhalten wir einen Graphen der Funktion `g`:

```
> plot3d(g, -1..1, -1..1);
```



Auch der Fall "gar kein Parameter" ist erlaubt::

```
> g2 := () -> 17;
```

$$g2 := ( ) \rightarrow 17$$

```
> g2();
```

$$17$$

Ohne Namen sieht das schon recht kryptisch aus :-)

```
> (()->17)();
```

$$17$$

```
>
```

Die Funktion `if()`

`if` ist eine Funktion mit drei Parametern, von denen der erste eine Bedingung sein muss (also ein Ausruck, der ausgewertet **`true`** oder **`false`** ergibt).
Bei erfüllter Bedingung (**`true`**) ist das Ergebnis der zweite, sonst der dritte Parameter.
(Weil es sich bei **`if`** um ein *reserviertes Wort* handelt, müssen wir es in `` `...` `` einschließen, wenn

wir es als Funktionsnamen verwenden)

```
> gerade := n -> `if`(n mod 2 = 0, 'gerade', 'ungerade');
```

$$gerade := n \to \text{`if`}(n \bmod 2 = 0, \text{'gerade'}, \text{'ungerade'})$$

```
> evalb(5 mod 2 = 0);
```

$$false$$

```
> gerade(5);
```

$$ungerade$$

```
> Max := (a,b) -> `if`(a>=b, a, b);
```

$$Max := (a, b) \to \text{`if`}(b \le a, a, b)$$

```
> Max(7,12);
```

$$12$$

```
>
```

Rekursive Funktionsdefinitionen

Wir bauen eine simple Funktion, die 2^n berechnet. So ginge es ganz einfach:

```
> zweihoch := n -> 2^n;
> zweihoch(10);
```

$$zweihoch := n \to 2^n$$

$$1024$$

Wir wollen aber ein Beispiel für eine rekursive Funktion haben:

- Wenn n=0 ist, ist das Ergebnis 1,
- wenn n>0 ist, benutzen wir $2^n = 2\ 2^{(n-1)}$,
- und um den Fall n<0 kümmern wir uns erstmal nicht.

```
> zweihoch := n -> `if`(n=0, 1, 2*zweihoch(n-1));
```

$$zweihoch := n \to \text{`if`}(n = 0, 1, 2\ \text{zweihoch}(n-1))$$

```
> zweihoch(0);
```

$$1$$

```
> zweihoch(48);
```

$$281474976710656$$

```
> zweihoch(-1);
Error, (in zweihoch) too many levels of recursion
```

```
> besser := n -> `if`(n<0, 1/zweihoch(-n), zweihoch(n));
```

$$besser := n \to \text{`if`}\left(n < 0, \frac{1}{\text{zweihoch}(-n)}, \text{zweihoch}(n)\right)$$

```
> besser(-10);
```

$$\frac{1}{1024}$$

```
>
```

Jetzt tun wir noch etwas für die Effizienz unserer Funktion.
Bisher führt die Auswertung von **`zweichoch(n)`** zu n+1 Funktionsaufrufen.

Billiger wird es, wenn wir für gerades n die Beziehung $2^n = \left(2^{\left(\frac{n}{2}\right)}\right)^2$ benutzen.
Dann brauchen wir allerdings ein weiteres if für die Unterscheidung gerade/ungerade:

```
> zweihoch := n -> `if`(n=0, 1, `if`(n mod 2 = 0,
  zweihoch(n/2)^2, 2*zweihoch(n-1)));
```

$$zweihoch := n \to \text{`if`}\left(n = 0,\ 1,\ \text{`if`}\left(n \bmod 2 = 0,\ \text{zweihoch}\left(\frac{1}{2}n\right)^2,\ 2\ \text{zweihoch}(n-1)\right)\right)$$

```
> zweihoch(14);
```

$$16384$$

```
>
```

Verschränkte Rekursion

Ein einfaches Beispiel für eine kompliziertere Rekursion:
wir bestimmen wieder, ob eine natürliche Zahl gerade oder ungerade ist:

- 1 ist nicht gerade,
- n>1 ist genau dann gerade, wenn n-1 ungerade ist.

Aber wann ist eine Zahl ungerade?

- 1 ist ungerade,
- n>1 ist genau dann ungerade, wenn n-1 gerade ist.

```
> restart;
> gerade := n -> `if`(n=1, false, ungerade(n-1));
```

$$gerade := n \to \text{`if`}(n = 1,\ false,\ \text{ungerade}(n-1))$$

```
> ungerade := n -> `if`(n=1, true, gerade(n-1));
```

$$ungerade := n \to \text{`if`}(n = 1,\ true,\ \text{gerade}(n-1))$$

```
> gerade(5);
```

$$false$$

```
> ungerade(5);
```

$$true$$

```
>
```

ggT berechnen

Die Berechnung des größten gemeinsamen Teilers ggT(a,b) zweier natürlicher Zahlen a und b mittels des euklidischen Algorihmus ist ein weiteres Standardbeispiel für rekursive Berechnung.
Wir nehmen zunächst $a \le b$ an, dann gilt:

- Wenn a b teilt, ist das Ergebnis a,
- sonst gilt ggT(a,b) = ggT(b mod a, a) und 0 < (b mod a) < a, so dass wir mit zunehmender Rekursionstiefe immer kleinere natürliche Zahlen als Parameter haben, daher muss die Rekursion terminieren (es gibt nämlich eine kleinste natürliche Zahl!).

```
> restart;
> ggT := (a, b) -> `if`(b mod a = 0, a, ggT(b mod a, a));
```

$$ggT := (a, b) \to \text{`if`}(b \bmod a = 0,\ a,\ \text{ggT}(b \bmod a, a))$$

```
> ggT(13*17, 13*19);
```

$$13$$

```
>
```

Maple hat noch eine zweite Aufschreibung für Funktionen (die behandeln wir eigentlich erst später).
Hier benutzen wir sie, um dieselbe Funktion wie oben mit der Option 'trace' auszustatten, die bewirkt, dass wir den Verlauf der Berechnung verfolgen können.

```
> ggT := proc(a, b) option arrow, operator, trace;
    `if`(b mod a = 0, a, ggT(b mod a, a))
  end;
```

$$ggT := (a, b) \rightarrow \text{`if`}(b \bmod a = 0, a, \text{ggT}(b \bmod a, a))$$

```
> ggT(13*17, 13*19);
{--> enter ggT, args = 221, 247
{--> enter ggT, args = 26, 221
{--> enter ggT, args = 13, 26
```

$$13$$

```
<-- exit ggT (now in ggT) = 13}
```

$$13$$

```
<-- exit ggT (now in ggT) = 13}
```

$$13$$

```
<-- exit ggT (now at top level) = 13}
```

$$13$$

Fibonacci-Zahlen

Ein Beispiel, an dem man schön sieht, dass eine elegante Aufschreibung zu einem sehr ineffizienten Verfahren führen kann, sind die Fibonacci-Zahlen: deren Folge beginnt mit 1, 1 und jedes weitere Folgenglied ist die Summe der beiden vorhergehenden:
1, **1**, **2**=1+1, **3**=1+2, **5**=2+3, **8**=3+5, **13**, **21**, **34**, **55**, ...
Die Berechnung der n-ten Fibonacci-Zahl F(n) erfordert also n-2 Additionen.
Obige Definition lässt sich auch schnell in eine Funktion umsetzen:

```
> F := n -> `if`(n<=2, 1, F(n-1) + F(n-2));
```

$$F := n \rightarrow \text{`if`}(n \le 2, 1, \text{F}(n-1) + \text{F}(n-2))$$

```
> F(1);
```

$$1$$

```
> F(2);
```

$$1$$

```
> F(3);
```

$$2$$

```
> F(4);
```

$$3$$

```
> F(10);
```

$$55$$

```
> F(30);
```

$$832040$$

Das hat jetzt ganz schön lange gedauert für 28 Additionen - mit unserer Funktion ist wohl noch etwas faul.

Überlegen wir uns dazu, wieviele Additionen wirklich ausgeführt werden, deren Zahl nennen wir A(n):

- n=1 oder 2: keine Addition, A(1)=A(2)=0
- n>2: Additionen für F(n-1), für F(n-2) und eine weitere: A(n) = A(n-1)+A(n-2)+1

Das sieht ganz ähnlich aus wie die Definition von F(n) und tatsächlich gilt A(n) = F(n)-1. Unsere Implementierung braucht für F(30) also nicht 28 Additionen, sondern 832039.

Um eine direkte Formel für A(n) zu bekommen, müsste man Differenzengleichungen lösen können - zum Glück kann Maple das für uns erledigen (nur das Ergebnis anschauen, der Aufruf ist für uns nicht weiter relevant):

```
> A := rsolve({A(n)=A(n-1)+A(n-2)+1,A(1)=0,A(2)=0},A(n));
```

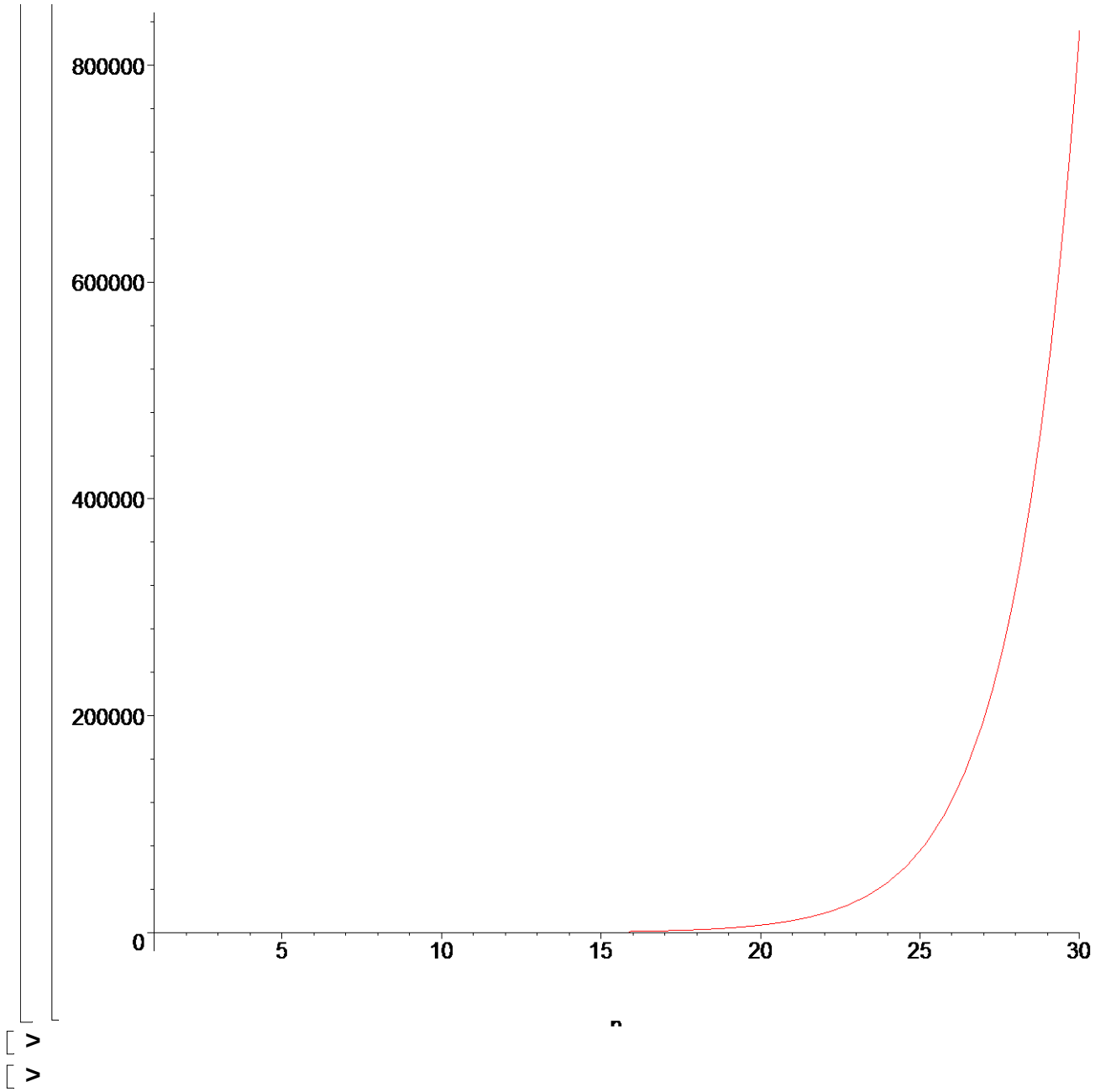
$$A := -1 - \frac{\sqrt{5}\left(-\frac{2}{1+\sqrt{5}}\right)^n}{5} - \frac{(-1+\sqrt{5})\sqrt{5}\left(-\frac{2}{1-\sqrt{5}}\right)^n}{5(1-\sqrt{5})}$$

```
> evalf(A);
```

$$-1. - 0.4472135954\,(-0.6180339888)^n + 0.4472135954\,1.618033989^n$$

Die Zahl der Additionen wächst also exponentiell (1.618... ist größer als 1), das Programm ist völlig unbrauchbar!

```
> plot(A, n=1..30);
```


800000
600000
400000
200000
0
5
10
15
20
25
30
n
V
V