Computer und Software 1

Christof Köhler

7. Maple – Programmieren

Folien: Balint Aradi

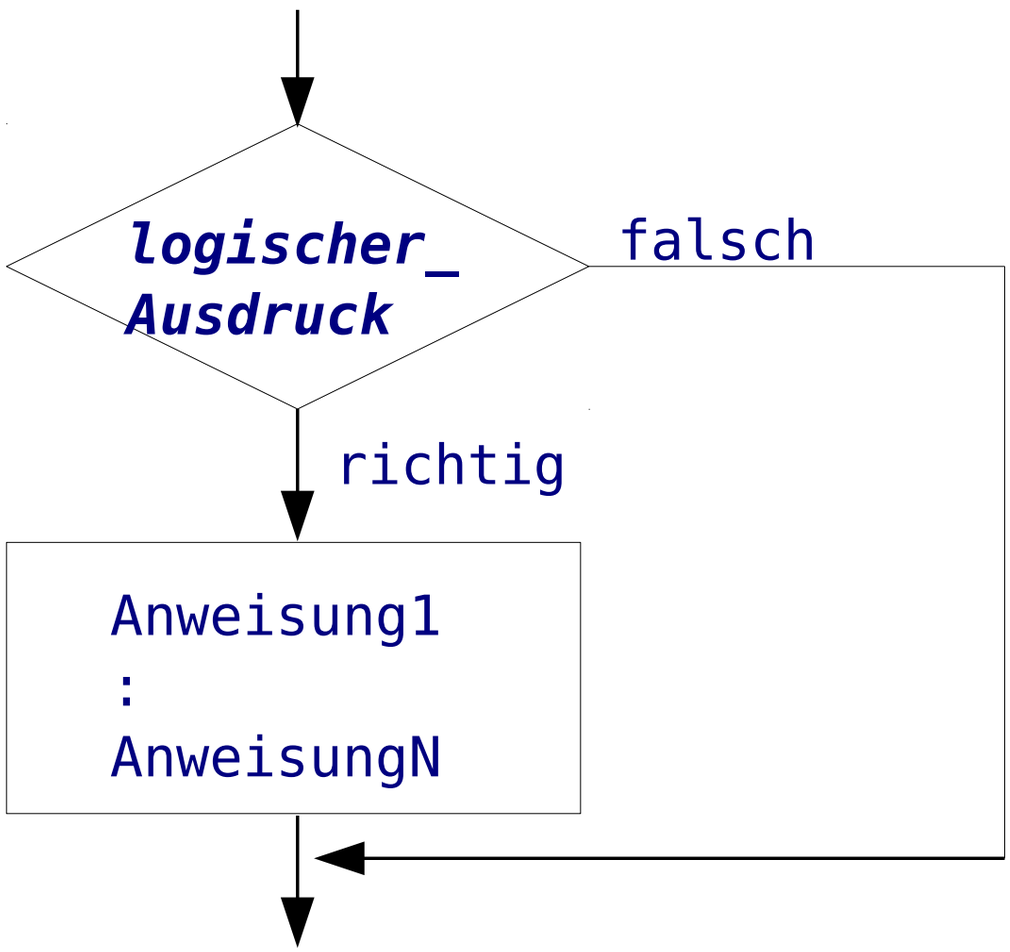
Details zu den Klausuren

- Die Klausuren finden am **11. bzw. 12. März** statt.
- Die genaue Einteilung erfolgt nach dem 31. Januar (wenn keine Abmeldungen mehr möglich sind). Die entsprechende Liste wird über StudIP verteilt. Es sind nur **Anmeldungen über PABO** zulässig!
- Die Klausuren finden entweder im CIP-Poolraum statt, oder im Poolraum des BCCMS im TAB-Gebäude (wird rechtzeitig bekannt gegeben).
- Es wird während der Klausur keinen Internetzugang geben.
- Als Hilfsmittel ist ein A4-Blatt mit Notizen erlaubt. Es dürfen jedoch keine Lösungen von Übungsaufgaben oder mehrzeilige Algorithmen darauf stehen . (Wenn das Notizblatt mit Computer erstellt wird, ist die minimale zu verwendende Schriftgröße 10px.)
- Zusätzlich darf selbstverständlich die in Maple eingebaute Hilfe verwendet werden.
- Bei Physikaufgaben werden die Formeln (im Gegensatz zu den Übungen) explizit vorgegeben.
- Die Klausur dauert ca. 2 Stunden.

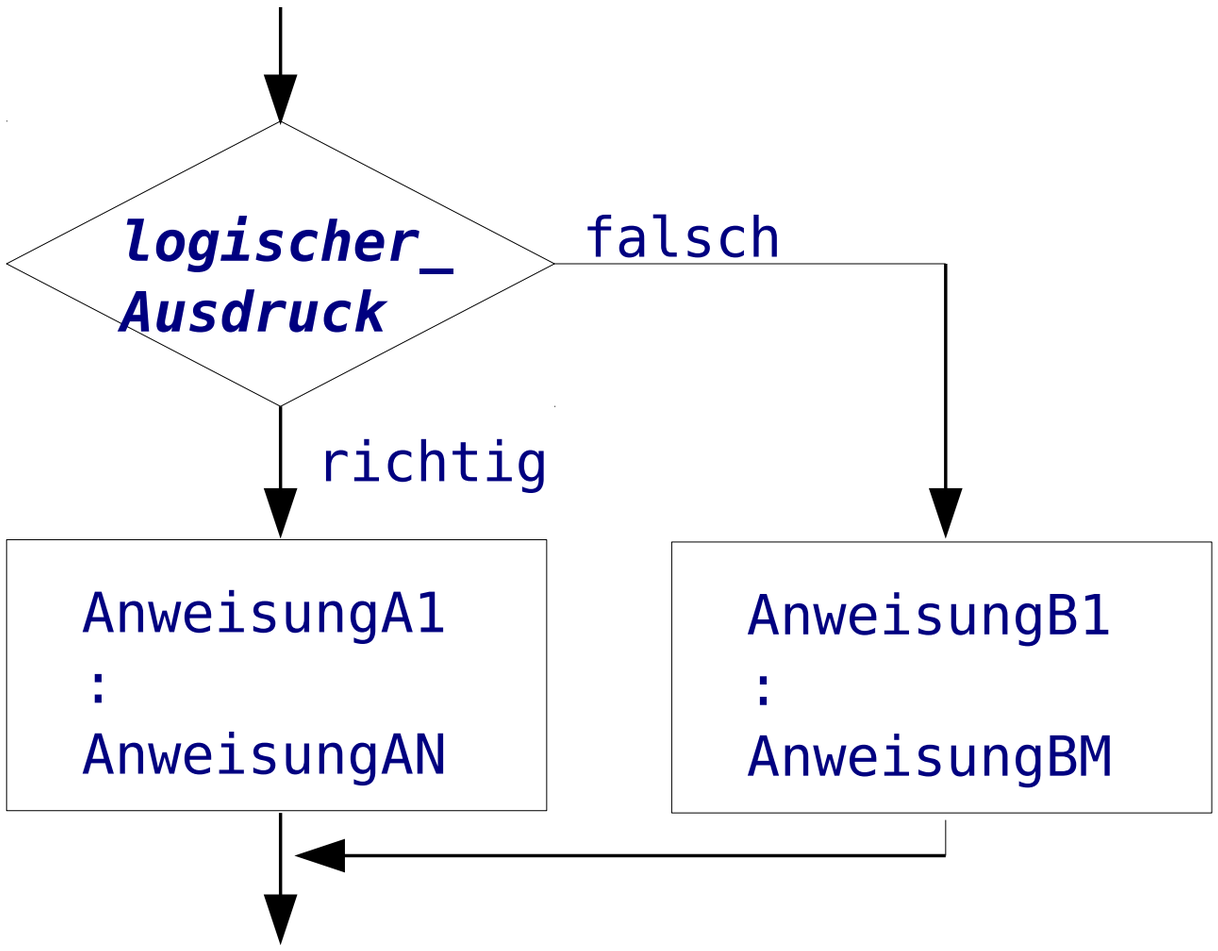
Fallunterscheidung

- Programmiersprachen enthalten Konstrukte für Fallunterscheidung/Verzweigung
- Abhängend vom Wert eines logischen Ausdrucks werden unterschiedliche Befehle ausgeführt:

Wenn-dann

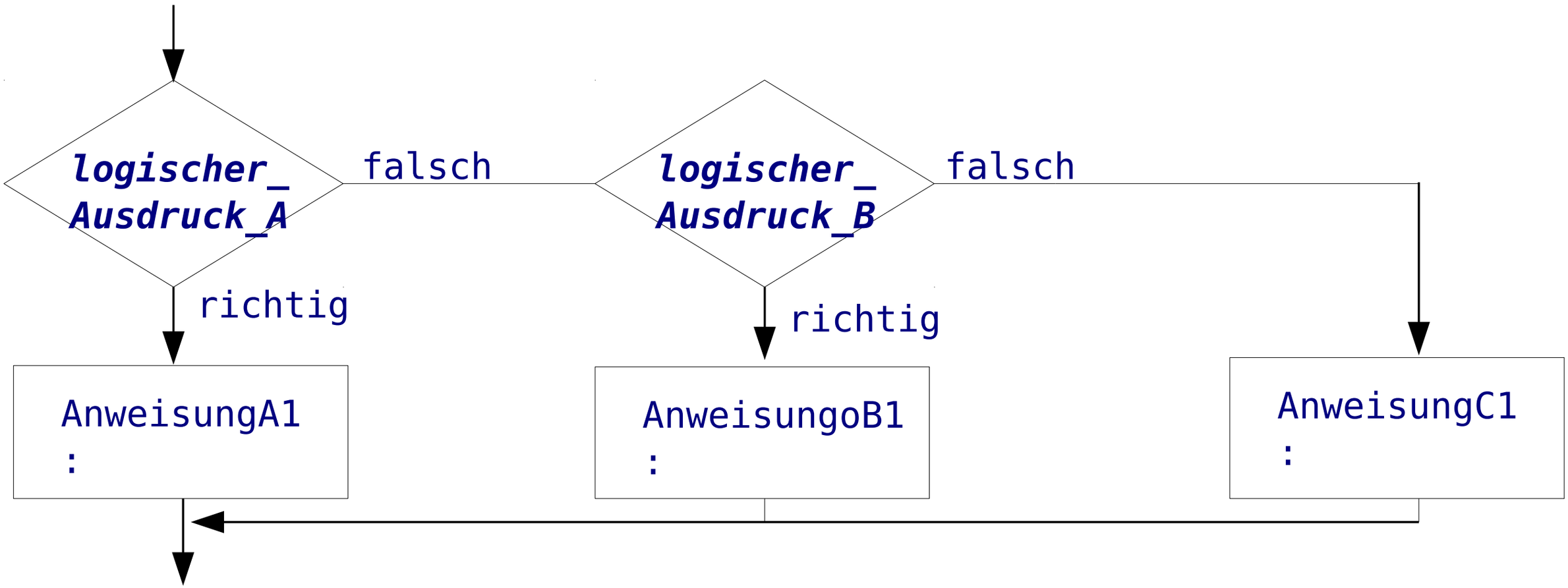


Entweder-oder



Fallunterscheidung

Verschachtelte Abfragen:



Fallunterscheidung in Maple

- Einfache Form:

if *logischer Ausdruck* **then** *Anweisung A* **else** *Anweisung B* **end if;**

```
> my_Heaviside := x -> if x < 0 then 0 else 1 end if;
```

$$my\_Heaviside := x \rightarrow \textbf{if } x < 0 \textbf{ then } 0 \textbf{ else } 1 \textbf{ end if;}$$

```
> my_Heaviside(-1);
```

$$0$$

```
> my_Heaviside(2);
```

$$1$$

liefert 0, wenn x < 0,
ansonsten 1

- Geschachtelte Form:

if *logischer Ausdruck* **then** *Anweisung A*
elif *logischer Ausdruk* **then** *Anweisung B*
:
else *Anweisung Z* **end if;**

```
> my_signum :=
    x -> if x < 0 then -1 elif x = 0 then 0 else 1 end if;
```

$$my\_signum := x \rightarrow \textbf{if } x < 0 \textbf{ then } -1 \textbf{ elif } x = 0 \textbf{ then } 0 \textbf{ else } 1 \textbf{ end if;}$$

```
> my_signum(-2), my_signum(0), my_signum(3);
```

$$-1, 0, 1$$

Logische Operatoren

- Einzelne Ausdrücke können mit **Relationsoperatoren** verglichen werden:

<	kleiner	**>=**	größer oder gleich
<=	kleiner oder gleich	**=**	gleich
>	größer	**<>**	ungleich

- Logische Ausdrücke können miteinander verknüpft werde:

and	und
or	oder
not	nicht

- Manuelle Auswertung eines logischen Ausdruckes via **evalb():**

```
> x := 12;
```

$$x := 12$$

```
> evalb(3 < x and x < 7);
```

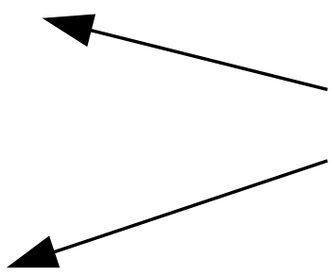
$$false$$

```
> x := 5;
```

$$x := 5$$

```
> evalb(3 < x and x < 7);
```

$$true$$



Überprüfe ob x zwischen 3 und 7 liegt

Symbole in logischen Ausdrücken

- **Vorsicht!** Bei logischen Ausdrücken werden die Operanden nicht automatisch ausgewertet:

```
> evalb(3 < Pi);
```

$$3 < \pi$$

Nicht entscheidbar, da kein numerischer Wert für Pi eingesetzt wurde

```
> evalb(3 < evalf(Pi));
```

$$true$$

- Die logischen Ausdrücke im **if**-Konstrukt müssen eindeutig entscheidbar sein:

```
> my_Heaviside := x -> if x < 0 then 0 else 1 end if;
```

$$my\_Heaviside := x \rightarrow \textbf{if } x < 0 \textbf{ then } 0 \textbf{ else } 1 \textbf{ end if;}$$

```
> my_Heaviside(-9), my_Heaviside(3.5);
```

$$0, 1$$

```
> my_Heaviside(Pi);
Error, (in my_Heaviside) cannot determine if this
expression is true or false: Pi < 0

> my_Heaviside(evalf(Pi));
```

$$1$$

Fehler wegen nicht entscheidbaren logischen Ausdrucks

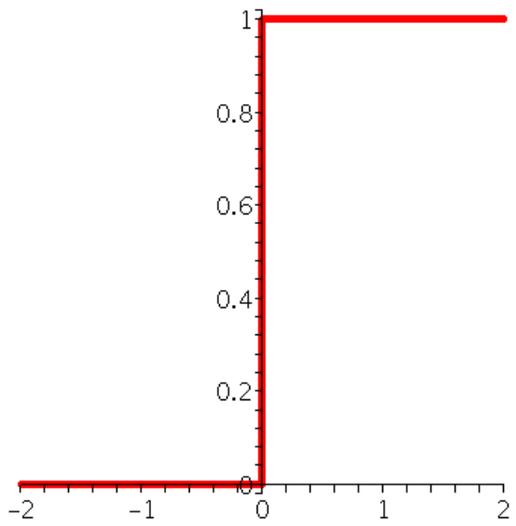
Darstellung von Funktionen mit if-Konstrukte

- Funktionen mit **if**-Konstrukte sind keine richtigen Funktionen (sondern **Prozeduren**), und müssen ohne Angabe des Argumentes dargestellt werden:

```
> plot(my_Heaviside(x), x=-2..2);
Error, (in my_Heaviside) cannot determine if this
expression is true or false: x < 0

> plot(my_Heaviside, x=-2..2);
Error, (in plot) invalid plotting of procedures, perhaps
you mean plot(my_Heaviside, -2 .. 2)

> plot(my_Heaviside, -2..2);
```



if-Konstrukt versus piecewise()-Funktion

- Funktionen mit **if**-Konstrukte können nicht abgeleitet oder integriert werden:

```
> int(my_heaviside(x), x=0..infinity);
Error, (in my_heaviside) cannot determine if this
expression is true or false: x < 0

> diff(my_heaviside(x), x);
Error, (in my_heaviside) cannot determine if this
expression is true or false: x < 0
```

- Deswegen wenn möglich, sollte **if** mit **piecewise()** ersetzt werden.
- **piecewise()** ist eine Funktion, kann also entsprechend integriert/differenziert werden:

```
> my_Heaviside2 := x -> piecewise(x < 0, 0, x >= 0, 1);
```

$$my\_Heaviside2 := x \to piecewise(x < 0, 0, 0 \le x, 1)$$

```
> int(my_Heaviside2(x), x=0..infinity);
```

$$\infty$$

```
> diff(my_Heaviside2(x), x);
```

- Wenn entsprechende Maple-Funktion mit der selben Funktionalität da ist, sollte selbstverständlich diese verwendet werden:

```
> int(Heaviside(x), x=0..infinity);
```

$$\infty$$

Heaviside-Funktion ist in Maple schon definiert

Schleife

- Wiederholte Ausführung bestimmter Anweisungen:

for *Variable* **from** *Startwert* **to** *Endwert* **do** *Anweisung(en)* **end do;**
for *Variable* **from** *Startwert* **by** *Schrittweite* **to** *Endwert* **do** *Anweisung(en)* **end do;**

```
> answer := 0;
  for i from 1 to 5 do
    answer := answer + i
  end do;
```

$$answer := 0$$
$$answer := 1$$
$$answer := 3$$
$$answer := 6$$
$$answer := 10$$
$$answer := 15$$

Addieren der Zahlen von 1 bis 5 (nur als Beispiel, sollte eigentlich mit der **sum()** Funktion realisiert werden.)

```
> answer := 0;
  for i from 1 by 2 to 5 do
    answer := answer + i;
  end do;
```

$$answer := 0$$
$$answer := 1$$
$$answer := 4$$
$$answer := 9$$

Addieren der ungeraden Zahlen von 1 bis 5 (sollte auch eigentlich via sum() gemacht werden)

Schleife

- Schleife mit Doppelpunkt abgeschlossen: Ergebnis der Anweisungen in der Schleife nicht sichtbar:

```
> answer := 0;
  for i from 1 by 2 to 5 do
    answer := answer + i;
  end do:
  answer;
```

Unterdrücke Output von der Schleife

$$answer := 0$$

$$9$$

- Bei Variablenneudefinitionen wird zuerst die rechte Seite ausgewertet und erst dann das Ergebnis der linken Seite zugeordnet.

```
> answer := 5;
```

$$answer := 5$$

```
> answer := answer + 1;
```

$$answer := 6$$

zuerst rechte Seite ausgewertet und dann Variable mit dem Ergebnis überschrieben

Schleife

- Schleifenvariable braucht vor der Schleife nicht deklariert worden zu sein.
- Wenn Variable vorher schon existiert, wird sie während der Schleifenausführung automatisch überschrieben.
- Nach Abschluss der Schleifenausführung hat die Schleifenvariable den letzten verwendeten Wert + die Schrittweite:

```
> i := 122;
```

$$i := 122$$

```
> for i from 1 by 2 to 5 do i; end do;
```

$$1$$
$$3$$
$$5$$

← Vorheriger Wert von i in der Schleife überschrieben

```
> i;
```

$$7$$

```
> for i from 1 by 2 to 4 do
    i;
  end do;
```

$$1$$
$$3$$

Es zählt der letze verwendete Wert für die Schleifenvariable, nicht die Obergrenze der Schleife!

```
> i;
```

$$5$$

Schleife mit Bedingung

- Statt Schleifenvariablenobergrenze kann beliebiges Schleifenausführungskriterium verwendet werden.

for *Variable* **from** *Startwert* **while** Ausführungs*kriterium* **do** *Anweisung(en)* **end do;**
for *Variable* **from** *Startwert* **by** *Schrittweite* **while** *Kriterium* **do** *Anweisung(en)* **end do;**

```
> x := 0;
  xold := -infinity;
```

$$x := 0$$

$$xold := -\infty$$

```
> for i from 1 while abs(evalf(x-xold)) > 1.0e-5 do
    xold := x;
    x := evalf(cos(x));
  end do;
> evalf(x);
```

$$0.7390822985$$

Schleife ausgeführt, solange Ausführungskriterium erfüllt wird (Wenn keine Konvergenz, **endlose Schleife**!)

Fixpunktiteration $x_{n+1} = \cos(x_n)$, solange Fixpunkt nicht auf 5 absolute Stellen genau ist.

Vorzeitiges Verlassen der Schleife

- Mit dem **break** Befehl kann eine Schleife vorzeitig verlassen werden:

```
> for i from 1 to 1000 do;
    xold := x;
    x := evalf(cos(x));
    if abs(x - xold) < 1e-5 then break; end if;
  end do;
```

- Beim Vorzeitigen verlassen der Schleife behält die Schleifenvariable den letzten zugewiesenen Wert:

```
> for i from 1 to 10000 do;
    if i = 10 then break; end if;
  end do:
> i;
```

10

Prozeduren / Unterprogramme

- Öfter verwende Algorithmen (Anweisungsreihenfolgen) können als Unterprogramme (Prozeduren) gespeichert werden.

Name **:= proc**(*arg1*, *arg2*, ...)
  local *lokaleVariable1*, *lokaleVariable2*, …;
  global *globaleVariable1*, *globaleVariable2*, ...;
  Anweisungen
end proc;

Deklarationen **local** und **global** sind optional.

Beispiel: Summation jeder zweiten Zahl in einem Intervall:

```
> sum2 := proc(startval, endval)
    local i, answer;

    answer := 0;
    for i from startval by 2 to endval do
      answer := answer + i;
    end do;
    answer;
  end proc;

> sum2(2,7);
```

Teile eines Unterprogramms

- Beim Aufruf könne Prozeduren (genau wie bei Funktionen) Parameter übergeben werden. Anzahl und Name der Parameter wird bei der Deklaration festgelegt:

```
> sum2 := proc(startval, endval)
```

Prozedure sum2() akzeptiert zwei Parameter

- Alle Variablen, die nur innerhalb des Unterprogrammes verwendet werden, müssen nach einer **local** Anweisung aufgezählt werden:

```
local i, answer;
```

- Es ist nicht erlaubt eine Variable ohne Lokalanweisung im Unterprogramm zu verwenden (obwohl Maple das schluckt):

```
> myproce := proc(inp)
    for i from 1 to 10 do
      j := i * inp;
    end do;
  end proc;
Warning, `i` is implicitly declared local to procedure
`myproce`
Warning, `j` is implicitly declared local to procedure
`myproce`
```

Teile eines Unterprogramms

- Sollten Variablen im Arbeitsblatt schon mit denselben Namen wie lokale Variablen in einer Prozedur deklariert worden sein, verlieren sie im Unterprogramm ihren Wert (als wären sie noch nicht verwendete Variablen.)
- Nach Verlassen des Unterprogrammes wird ihr alter Wert wieder hergestellt.

```
> myproc := proc()
    local i;
    i := 25;
  end proc;
```

$$myproc := \mathbf{proc()}\ \mathbf{local}\ i;\ i := 25\ \mathbf{end\ proc};$$

```
> i := 12;
```

$$i := 12$$

```
> myproc();
```

$$25$$

```
> i;
```

$$12$$

Teile eines Unterprogramms

- Das Unterprogramm kann beliebige Maple-Anweisungen enthalten:

```
answer := 0;
for i from startval by 2 to endval do
  answer := answer + i;
end do;
answer;
```

- Die Unterprogramme werden genauso aufgerufen, wie die Funktionen

```
> sum2(1,10);
```

25

- Das „Ergebnis“ des Aufrufes ist das Ergebnis der letzten ausgeführten Anweisung im Unterprogramm. Alles andere, das während der Ausführung passiert ist von außen unsichtbar.

```
   answer := 0;
   for i from startval by 2 to endval do
     answer := answer + i;
   end do;
   answer;
end proc;
```

Vor Verlassen des Unterprogrammes wird noch *anwer* ausgewertet → Ergebnis = letzter Wert von *answer*

```
> sum2(1,10);
```

25

letzter Wert von answer: 25

Globale Variablen in Unterprogrammen

- Sollte ein Unterprogramm direkt Variablen im Maple-Arbeitsblatt manipulieren, sind die entsprechenden Variablen im Unterprogramm mit **global** zu deklarieren:

```
> myproc := proc()
    global a;

    a := 12;
  end proc;
```

$$myproc := \mathbf{proc}() \ \mathbf{global}\ a;\ a := 12\ \mathbf{end\ proc};$$

```
> a := 4;
```

$$a := 4$$

```
> myproc();
```

$$12$$

```
> a;
```

$$12$$

- Verwendung von globalen Variablen kann zu unerwarteten Nebeneffekten führen.

Bitte möglichst **keine globalen Variablen** verwenden!

Fehlersuche

- Details der Unterprogrammausführung können zwecks Fehlersuche mit **trace()** angezeigt werden:

```
> trace(sum2);
```

Information über detailierten Ablauf soll angezeigt werden.

sum2

```
> sum2(2, 7);
{--> enter sum2, args = 2, 7
```

$answer := 0$
$answer := 2$
$answer := 6$
$answer := 12$

answer in jeder Iteration neu definiert

```
12
<-- exit sum2 (now at top level) = 12}
```

12

- Fehlersuchinfomation kann mit **untrace()** wieder abgeschaltet werden:

```
> untrace(sum2);
```

sum2

```
> sum2(2,7);
```

12

Programmbeispiel: Mandelbrotmenge

- **Mandelbrotmenge:** Menge diejeniger komplexen Zahlen c, für die die Folge nach der Iterationsvorschrift

$$z_0 = 0$$

$$z_{n+1} = z_n^2 + c$$

beschränkt bleibt.

- Um die Beschränkheit für eine Zahl *c* zu ermitteln, werden wir die Iteration mit der Zahl ausführen, bis

  – **entweder** der Betrag von z_n einen vorgegebenen Maximalwert überschreitet
  (Iteration für die Zahl *c* **unbeschränkt**)

  – **oder** eine vorgegeben maximale Anzahl von Iterationen erreicht worden ist
  (Iteration für die Zahl *c* **beschränkt**)

- Wir wollen die Anzahl der Iterationen für jede untersuchte komplexe Zahl als farbkodiertes Bild in der x,y-Ebene darstellen. Wir machen “3D-Plot” einer 2D-Funktion.

Anmerkung: numerische Iterationen sollte man eher in einer **kompilierten** Sprache (Fortran, C/C++) programmieren: kann mehr als **3 Größenordnungen schneller** sein!

Programmbeispiel: Mandelbrotmenge (#2)

Schritt 1: Unterprogramm für die Iteration erstellen:

Prozedur Mandelbrot

Eingangsparameter:

- Komplexe Zahl c
- Anzahl der Maximalen Iterationen
- Maximaler Wert für Betrag

Rückgabewert:

- Anzah der tatsächlichen Iterationen

```
> mandelbrot := proc(c, maxiter, maxval)
    local i, z;

    z := 0;
    for i from 1 to maxiter do
      z := evalf(z)^2 + c;
      if abs(z) > maxval then break; end if;
    end do;
    i;
end proc;

> mandelbrot(0.3 + 0.4*I, 200, 4);
                                  201

> mandelbrot(-1.0 - 1.2*I, 200, 4);
                                    3
```

Programmbeispiel: Mandelbrotmenge (#3)

Schritt 2: Eine 2D-Funktion f(x,y) erzeugen, die zu jedem Punkt x,y die Iterationsanzahl für die enstprechende komplexe Zahl x + iy zuordnet:

```
> mandelbrot_xy := (x, y) -> mandelbrot(x + I*y, 200, 4);
```

$$mandelbrot\_xy := (x, y) \rightarrow mandelbrot(x + I\,y, 200, 4)$$

```
> mandelbrot_xy(0.3, 0.4);
```

$$201$$

```
> mandelbrot_xy(-1.0, -1.2);
```

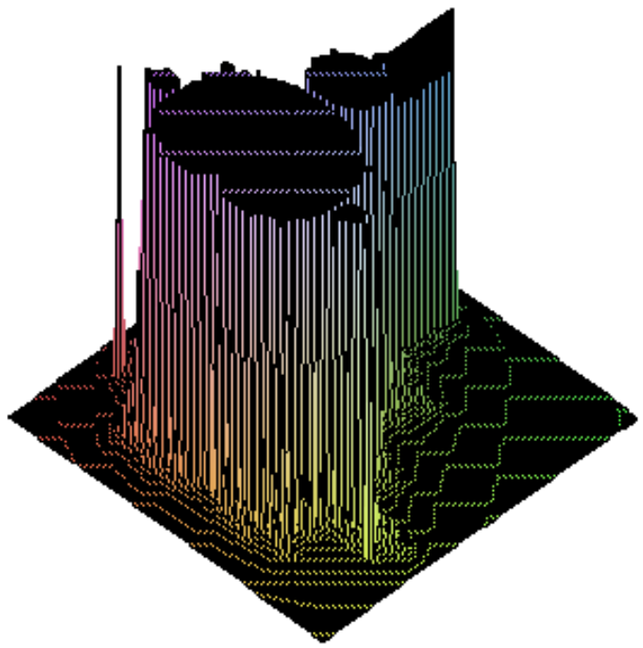
$$3$$

Wir verwenden feste Werte für die Anzahl der Iterationen und den maximalen Betrag

Programmbeispiel: Mandelbrotmenge (#4)

Schritt 3: f(x,y) als 3D-Plot darstellen

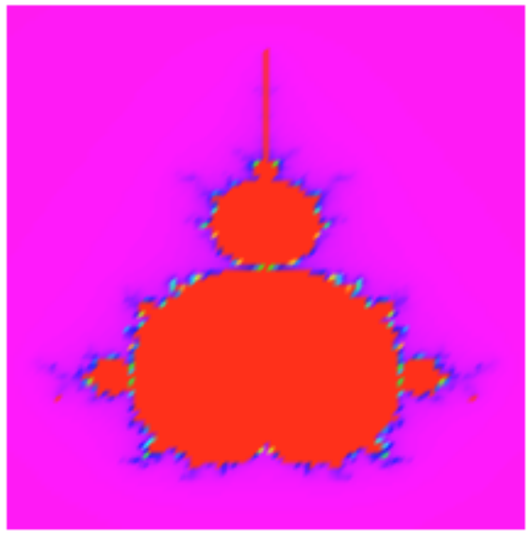
```
> with(plots):
Warning, the name changecoords has been redefined
> plot3d(mandelbrot_xy, -2.25..0.75, -1.25..1.25,
  numpoints=10000);
```



Programmbeispiel: Mandelbrotmenge (#5)

Schritt 4: 3D-Plot als Farbinformation (anstatt perspektivisch) darstellen

```
> plot3d(mandelbrot_xy, -2.25..0.75, -1.25..1.25,
  numpoints=10000, style=patchnogrid, shading=zhue,
  orientation=[0,0]);
```



Zoomen Sie in das Bild hinein, indem Sie auch die x,y-Intervalle
-0.7432..-0.7440, 0.1314..0.1322 bzw. -0.7425..-0.7445, 0.1304..0.1324 ausprobieren.