```
> restart:
> # kernelopts(assertlevel=2);
```

Aufgabe 3

Iteration versus Rekursion (mit option remember)

Schreiben Sie zwei Prozeduren zur effizienten Berechnung des n-ten Fibonacci-Polynoms (Leonardo Fibonacci). Die Fibonacci-Polynome sind durch die Rekursion

$F_0 = 0, F_1 = 1, F_n = x\,F_{n-1} + F_{n-2}$

definiert. Programmieren Sie die Rekursion

a) mittels "for loop" und "double assignment" (Doppelzuweisung, allgemein Mehrfachzuweisung)

Hinweis: Expandieren Sie die Polynome in jedem Rekursionsschritt (expand), sowohl in a) als auch in b).

```
> # fib_it := proc(n::integer, polynom::boolean := true)::integer;
   fib_it := proc(n::nonnegint, polynom::boolean := true)     #
  Besser:   n muss >= 0 sein.
                                                               #
  Rückgabe ist ein Polynom! ;
                              # Returntype wird nur beachtet,
  wenn
                              # kernelopts(assertlevel=2) gesetzt
  ist, ansonsten ignoriert.
    local f_n_minus_one, f_n_minus_two, f_n, i, x;
    description "Fibonacci iterativ. Wenn polynom auf true steht,
  so wird ein Fibonacci-Polynom ausgewertet, ansonsten die
  Fibonacci-Zahl.";
            # Es soll kein Polynom ausgewertet werden sondern ein
  Polynom generiert werden!

    if n < 0 then return(FAIL) end if;          # if-Abfragen sind
  zeitintensiv;
                                                # kann im
  Prozedurkopf überprüft werden, mit ::nonnegint;

    f_n_minus_one, f_n_minus_two, f_n := 0, 1, 1;

    if n = 0 then
```

```
    return f_n_minus_one
  elif n = 1 then
    return f_n_minus_two
  end if;

  if polynom then x:=x; else x:=1; end if;    # if-Abfrage kann
vermieden werden,
                                              # wenn x kann im
Prozedurkopf bestimmt wird.
                                              #    x::{symbol,1}
oder
                                              #    x::symbol:=1
                                              # So können auch
Polynome mit beliebigem Varibalennamen generiert werden.

  for i from 2 by 1 to n do
    f_n_minus_two := f_n_minus_one;
    f_n_minus_one := f_n;
    f_n := expand(x * f_n_minus_one + f_n_minus_two);  #
Doppelzuweisung noch effizienter. siehe fib_it1.
  end do;

  f_n

end proc;
```

fib_it := **proc**(*n*::*nonnegint*, *polynom*::*boolean* := *true*) **(1)**

local *f_n_minus_one*, *f_n_minus_two*, *f_n*, *i*, *x*;

description

"Fibonacci iterativ. Wenn polynom auf true steht, so wird ein Fibonacci-Polynom ausgewertet, ansonsten die Fibonacci-Zahl.";

if $n < 0$ **then return** *FAIL* **end if**;

f_n_minus_one, *f_n_minus_two*, *f_n* := 0, 1, 1;

if $n = 0$ **then return** *f_n_minus_one* **elif** $n = 1$ **then return** *f_n_minus_two* **end if**;

if *polynom* **then** *x* := *x* **else** *x* := 1 **end if**;

for *i* **from** 2 **to** *n* **do**

f_n_minus_two := *f_n_minus_one*;

f_n_minus_one := *f_n*;

f_n := *expand*(*x* * *f_n_minus_one* + *f_n_minus_two*)

```
    end do;
    f_n
end proc
```

```
> seq(fib_it(i),i=0..5);      # Eine Prozedur besser zum Test
  einmal aufrufen.
  seq(fib_it(i,1),i=0..5);
  seq(fib_it(i,true),i=0..5);
  seq(fib_it(i,false),i=0..5);
```

$$0, 1, x, x^2 + 1, x^3 + 2x, x^4 + 3x^2 + 1$$

$$0, 1, x, x^2 + 1, x^3 + 2x, x^4 + 3x^2 + 1$$

$$0, 1, x, x^2 + 1, x^3 + 2x, x^4 + 3x^2 + 1$$

$$0, 1, 1, 2, 3, 5 \tag{2}$$

```
> combinat:-fibonacci(5,x);    # Probe.
  combinat:-fibonacci(5);
```

$$x^4 + 3x^2 + 1$$

$$5 \tag{3}$$

Kompakte Prozedur:

```
> fib_it1:=proc(n::nonnegint,x::symbol:=1)
      local s0,s1,i;
      s0,s1:=0,1;
      for i from 1 to n do
          s0,s1:=s1,expand(x*s1+s0);
      end do;
      s0;
  end proc:
> seq(fib_it1(i,y),i=0..5);
```

$$0, 1, y, y^2 + 1, y^3 + 2y, y^4 + 3y^2 + 1 \tag{4}$$

b) mittels rekursiver Prozedur mit "option remember".

```
> fib_rek := proc(n::integer, polynom::boolean := true)::integer;
  # Siehe oben: Returntype wird nur beachtet, wenn  ...
    option remember;
    description "Fibonacci rekursiv. Wenn polynom auf true steht,
  so wird ein Fibonacci-Polynom ausgewertet, ansonsten die
  Fibonacci-Zahl.";
```

```
    if polynom then
      if n < 2 then n else expand(x * fib_rek(n-1) + fib_rek(n-2))
  end if;
    else
      if n < 2 then n else fib_rek(n-1, false) + fib_rek(n-2,
  false) end if;
    end if;
  end proc;
```

fib_rek := **proc**(*n*::*integer*, *polynom*::*boolean* := *true*)::*integer*; **(5)**

option *remember*;

description

"Fibonacci rekursiv. Wenn polynom auf true steht, so wird ein Fibonacci-Polynom ausgewertet, ansonsten die Fibonacci-Zahl.";

if *polynom* **then**

if $n < 2$ **then** n **else** $expand(x * fib\_rek(n-1) + fib\_rek(n-2))$ **end if**

else

if $n < 2$ **then** n **else** $fib\_rek(n-1, false) + fib\_rek(n-2, false)$ **end if**

end if

end proc

```
> seq(fib_rek(i,y),i=0..5);
  seq(fib_rek(i,true),i=0..5);
  seq(fib_rek(i,false),i=0..5);
```

$$0, 1, x, x^2 + 1, x^3 + 2x, x^4 + 3x^2 + 1$$

$$0, 1, x, x^2 + 1, x^3 + 2x, x^4 + 3x^2 + 1$$

$$0, 1, 1, 2, 3, 5 \quad (6)$$

Verbesserte, kompakte Prozedur:

```
> fib_rek1:=proc(n::nonnegint,x::symbol:=1)
      option remember;
      if n < 2 then n
      else
         expand(x*fib_rek1(n-1,x)+fib_rek1(n-2,x))
      end if
  end proc:
```

c) Vergleichen Sie die CPU-Zeiten für die Berechnungsmethoden in a) und b) bei $n = 3000$. Erklären Sie das beobachtete Laufzeitverhalten.

```
> n := 3000;
  forget(fib_rek);                # Löschen der remember table;
  time(fib_it(n));
  time(fib_rek(n));
  n := 5000;
  forget(fib_rek1);
  time(fib_it1(n,x));
  time(fib_rek1(n,y));
```

$$n := 3000$$

$$2.265$$

$$5.484$$

$$n := 5000$$

$$19.281$$

$$29.125 \tag{7}$$

Antwort: Die Laufzeitunterschiede verhalten sich ähnlich wie in anderen Programmiersprachen, da die rekursive Funktion meistens die einfachere und auch die mit weniger Aufwand programmierbare ist. Allerdings wird der Stack durch die vielen rekursiven Aufrufe vollgeschrieben und muss dann in einer Art Baummodell abgearbeitet werden. Dies ist extrem langsam und daher ist die iterative Variante, bei der nur meistens nur der letzte Eintrag gespeichert werden muss viel schneller, als wenn immer bis auf die Abbruchbedingung zurückgegangen werden muss.

Deine Antwort ist die richtig.

Die größer werdende "remember table" bei Algorithmus b) verursacht zunehmend Speicherverwaltungszeiten, die in Algorithmus a) vermieden werden.

d) Für $x = 1$ definiert die obige Rekursion die Fibonacci-Zahlen. Modifizieren Sie die Parameterlisten der Prozeduren in a) und b) so, dass sowohl Fibonacci-Polynome als auch Fibonacci-Zahlen berechnet werden können.

```
> for i from 0 by 1 to 5 do
    fib_it(i, false);
    fib_rek(i, false);
  end do;
```

$$0$$

$$0$$

$$1$$

$$1$$

$$1$$

$$1$$

$$2$$

2

3

3

5

5 **(8)**