A06: 24P


```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Aufgabenblatt 06 - SE3-LP WiSe 16/17
%
% Finn-Lasse Jörgensen 6700628 4joergen@informatik.uni-hamburg.de
% Fabian Behrendt 6534523 fabian.behrendt95@gmail.com
% Daniel Klotzsche 6535732 daniel_klotzsche@hotmail.de
%
% Wir sind bereit folgende Aufgaben zu präsentieren:
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

A1: 12P

```
%%% A1 %%%
%% Aufgabe 1.1

% Implementation mit rekursivem Abstieg.
goldener_schnitt_abstieg(0, 1).
goldener_schnitt_abstieg(Schritt, Result) :-
    Schritt > 0,
    Naechster_Schritt is Schritt - 1,
    goldener_schnitt_abstieg(Naechster_Schritt, Naechster_Schritt_Result),
    Result is (1 / Naechster_Schritt_Result + 1).

% ?- goldener_schnitt_abstieg(10, Result).
%    Result = 1.617977528089887

% Implementation mit rekursivem Aufstieg.
goldener_schnitt_schritt(Akk, 0, Akk).
goldener_schnitt_schritt(Akk, StepsLeft, Result) :-
    StepsLeft > 0,
    Zwischen_Ergebnis is ((1 / Akk) + 1),
    StepsLeftNow is StepsLeft - 1,
    goldener_schnitt_schritt(Zwischen_Ergebnis, StepsLeftNow, Result).

goldener_schnitt_aufstieg(Steps, Result) :-
    goldener_schnitt_schritt(1, Steps, Result).

% ?- goldener_schnitt_aufstieg(10, Result).
%    Result = 1.6179775280898876

% goldener_schnitt_aufstieg ist endrekursiv

%% Aufgabe 1.2

% ?- time(goldener_schnitt_abstieg(100000, Result)).
% 300,000 inferences, 3.359 CPU in 3.365 seconds (100% CPU, 89302 Lips)
% Result = 1.618033988749895

% ?- time(goldener_schnitt_aufstieg(100000, Result)).
% 300,001 inferences, 0.031 CPU in 0.034 seconds (93% CPU, 9600032 Lips)
% Result = 1.618033988749895 .

% ?- time(goldener_schnitt_abstieg(1000000, Result)).
% Prolog liefert kein Ergebnis und reagiert nicht mehr auf Eingaben. Hypothese:
 Out of Stack.

% ?- time(goldener_schnitt_aufstieg(1000000, Result)).
% 3,000,001 inferences, 0.312 CPU in 0.321 seconds (97% CPU, 9600003 Lips)
% Result = 1.618033988749895

% ?- time(goldener_schnitt_abstieg(10000000, Result)).
% Prolog liefert kein Ergebnis und reagiert nicht mehr auf Eingaben. Hypothese:
 Out of Stack.

% ?- time(goldener_schnitt_aufstieg(10000000, Result)).
```



```
% 30,000,002 inferences, 2.813 CPU in 2.855 seconds (99% CPU, 10666667 Lips)
% Result = 1.618033988749895

% Die nicht-endrekursive Variante läuft ab einer Schrittzahl von 1000000 wahrsc
heinlich 'out of stack',
% während die Implementation mit Endrekursion effizienter arbeitet, selbst mit
größerer Rekursionstiefe.

%% Aufgabe 1.3

%% fibonacci(+Rekursionstiefe,?Fibonacci-Zahl)
fibonacci(0,1).
fibonacci(1,2).
fibonacci(N,F) :-
 N > 1,
 N1 is N - 1, N2 is N - 2,
 fibonacci(N1,F1), fibonacci(N2,F2),
 F is F1 + F2.

% Die Lösung ist sehr ineffizient, da in jedem Schritt alle vorherigen Fibonacc
i Zahlen berechnet werden müssen.
% Wenn man sich die Vorgänger merkt und sie nicht immer neu berechnen muss, wür
de man sich viele rechenschritte sparen.

fibonacci_endr(0, 1).
fibonacci_endr(1, 2).
fibonacci_endr(N, F) :-
        fibonacci_endr_schritt(1, N, F).

fibonacci_endr_schritt(_, F1, N, N, F1).
fibonacci_endr_schritt(F0, F1, Akk, N, F) :-
        F2 is F0 + F1,
        Akk_2 is Akk + 1,
        fibonacci_endr_schritt(F1, F2, Akk_2, N, F).

% ?- time(fibonacci(10, R)).
% 353 inferences, 0.000 CPU in 0.000 seconds (?% CPU, Infinite Lips)
% R = 144 .

% ?- time(fibonacci_endr(10, R)).
% 21 inferences, 0.000 CPU in 0.000 seconds (?% CPU, Infinite Lips)
% R = 144

%% Aufgabe 1.4

goldener_schnitt_fib(N, Result) :-
  N_2 is N - 1,
  fibonacci_endr(N, F1),
  fibonacci_endr(N_2, F2),
  Result is F1 / F2.
```

A2: 0P

```
%%% A2 %%%
%% 1)
% Gibt alle natuerlichen Zahlen aus.
% Wenn eine natuerliche Zahl als Resultat uebergeben wird, muss nach dem true m
it . abgebrochen werden.
% nat_zahl(?Resultat)
nat_zahl(Resultat) :-
    nat_zahl_helper(0, Resultat).
nat_zahl_helper(Resultat, Resultat).
nat_zahl_helper(Zwischenresultat, Resultat) :-
    Zwischenresultat2 is Zwischenresultat + 1,
    nat_zahl_helper(Zwischenresultat2, Resultat).

%% 2)
```

```
% nat_zahl(?Resultat, +Grenze)
nat_zahl(Resultat, Grenze) :-
    nat_zahl_helper(0, Resultat, Grenze).
nat_zahl_helper(Resultat, Resultat, _).
nat_zahl_helper(Zwischenresultat, Resultat, Grenze) :-
    Zwischenresultat < Grenze,
    Zwischenresultat2 is Zwischenresultat + 1,
    nat_zahl_helper(Zwischenresultat2, Resultat, Grenze).
```

%% 3) %es sollte goldener Schnitt als Strom definiert werden

```
% goldener_schnitt_incr(+Rekursionen, -R)
goldener_schnitt_incr(Rekursionen, R) :-
    nat_zahl(Resultat, Rekursionen),
    goldener_schnitt_absteiger(Resultat, R).

/*
Es werden ungefähr 5-6 Schritte benötigt bevor die
Darstellungsgenauigkeit nicht mehr gesteigert werden kann.
*/
```

A3: 12P
Doku?-1P

```
%%% A3 %%%
%% Aufgabe 3.1

% Definiert ein Prädikat als Typtest für Binärbäume.
%
% binaryTree(+X)
binaryTree(X) :-
  atom(X).

binaryTree(s(X, Y)) :-
  binaryTree(X),
  binaryTree(Y).

% Tests %
% ?- binaryTree(a).
% true.

% ?- binaryTree(s(a)).
% false.

% ?- binaryTree(s(a, b)).
% true.

% ?- binaryTree(s(s(a, b), c)).
% true.

% ?- binaryTree(s(a, s(b, c))).
% true.

% ?- binaryTree(s(a, 1)).
% false.

%% Aufgabe 3.2
% Das Prädikat lokaleTiefe(Binärbaum, Tiefe) berechnet für einen Binärbaum
```



```
% die lokalen Einbettungstiefen auf allen Pfaden vom Spitzenknoten zu den
% einzelnen Blattknoten.
%
% lokaleTiefe(+Binärbaum, -Tiefe)
lokaleTiefe(X, 0) :-
  atom(X).

lokaleTiefe(s(X, Y), Tiefe) :-
  binaryTree(s(X, Y)),
  lokaleTiefe(X, TiefeX),
  Tiefe is TiefeX + 1.

lokaleTiefe(s(X, Y), Tiefe) :-
  binaryTree(s(X, Y)),
  lokaleTiefe(Y, TiefeY),
  Tiefe is TiefeY + 1.

% Tests %
% ?- lokaleTiefe(a, Tiefe).
% Tiefe = 0.

% ?- lokaleTiefe(s(a, b), Tiefe).
% Tiefe = 1 ;
% Tiefe = 1.

% ?- lokaleTiefe(s(s(a, s(b, c)), s(d, e)), Tiefe).
% Tiefe = 2 ;
% Tiefe = 3 ;
% Tiefe = 3 ;
% Tiefe = 2 ;
% Tiefe = 2.

%% Aufgabe 3.3
% Das nichtrekursive Prädikat maxTiefe(Binärbaum, Tiefe) ermittelt für einen
% Binärbaum die maximale Einbettungstiefe.
%
% maxTiefe(+Binärbaum, -Tiefe)
maxTiefe(Baum, Tiefe) :-
  findall(LokaleTiefe, lokaleTiefe(Baum, LokaleTiefe), Tiefenliste),
  max_list(Tiefenliste, Tiefe).

% Tests %
% ?- maxTiefe(a, Tiefe).
% Tiefe = 0.

% ?- maxTiefe(s(a, b), Tiefe).
% Tiefe = 1.

% ?- maxTiefe(s(s(a, b), c), Tiefe).
% Tiefe = 2.

% ?- maxTiefe(s(a, s(b,c)), Tiefe).
% Tiefe = 2.

%% Aufgabe 3.4
% Das rekursive Prädikat maxTiefeAlt(Binärbaum, Tiefe) ermittelt für einen
% Binärbaum die maximale Einbettungstiefe.
%
% maxTiefeAlt(+Binärbaum, -Tiefe)
maxTiefeAlt(X, 0) :-
  atom(X).

maxTiefeAlt(s(X, Y), Tiefe) :-
  maxTiefeAlt(X, TiefeX),
  maxTiefeAlt(Y, TiefeY),
```

```
  Tiefe is 1 + maxTiefeX, TiefeY).

% Tests %
% ?- maxTiefeAlt(a, Tiefe).
% Tiefe = 0.

% ?- maxTiefeAlt(s(a, b), Tiefe).
% Tiefe = 1.

% ?- maxTiefeAlt(s(s(a, b), c), Tiefe).
% Tiefe = 2.

% ?- maxTiefeAlt(s(a, s(b,c)), Tiefe).
% Tiefe = 2.

%% Aufgabe 3.5
% Das Prädikat minTiefeAlt(Binärbaum, Tiefe) ermittelt für einen
% Binärbaum die minimale Einbettungstiefe.
%
% minTiefeAlt(+Binärbaum, -Tiefe)
minTiefeAlt(X, 0) :-
  atom(X).

minTiefeAlt(s(X, Y), Tiefe) :-
  minTiefeAlt(X, TiefeX),
  minTiefeAlt(Y, TiefeY),
  Tiefe is 1 + minTiefeX, TiefeY).

% Das Prädikat balanciert(Binärbaum) überprüft, ob ein Binärbaum balanciert
% ist, d.h. die lokalen Einbettungstiefen dürfen sich maximal um den Wert
% eins unterscheiden.
%
% balanciert(+Binärbaum)
balanciert(Baum) :-
  maxTiefeAlt(Baum, MaxTiefe),
  minTiefeAlt(Baum, MinTiefe),
  MaxTiefe = MinTiefe.

balanciert(Baum) :-
  maxTiefeAlt(Baum, MaxTiefe),
  minTiefeAlt(Baum, MinTiefe),
  MaxTiefe =:= MinTiefe + 1.

% Tests %
% ?- balanciert(s(a, s(b, c))).
% true.

% ?- balanciert(s(a, s(b, s(c, d)))).
% false.
```