3.5.4 Keller (Stack)

Definition: Eine lineare Liste heißt Keller oder Stapel (engl.: stack oder pushdown), wenn auf ihr genau die folgenden fünf Operationen zugelassen sind:

(1) "Empty" = Leeren der Liste.

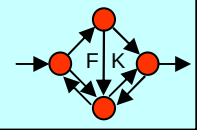
(2) "Isempty" = Abfragen auf Leerheit der Liste.

(3) "Top" = Kopieren des letzten Elements der Liste.

(4) "Push" = Hinzufügen eines Elements am Ende der Liste.

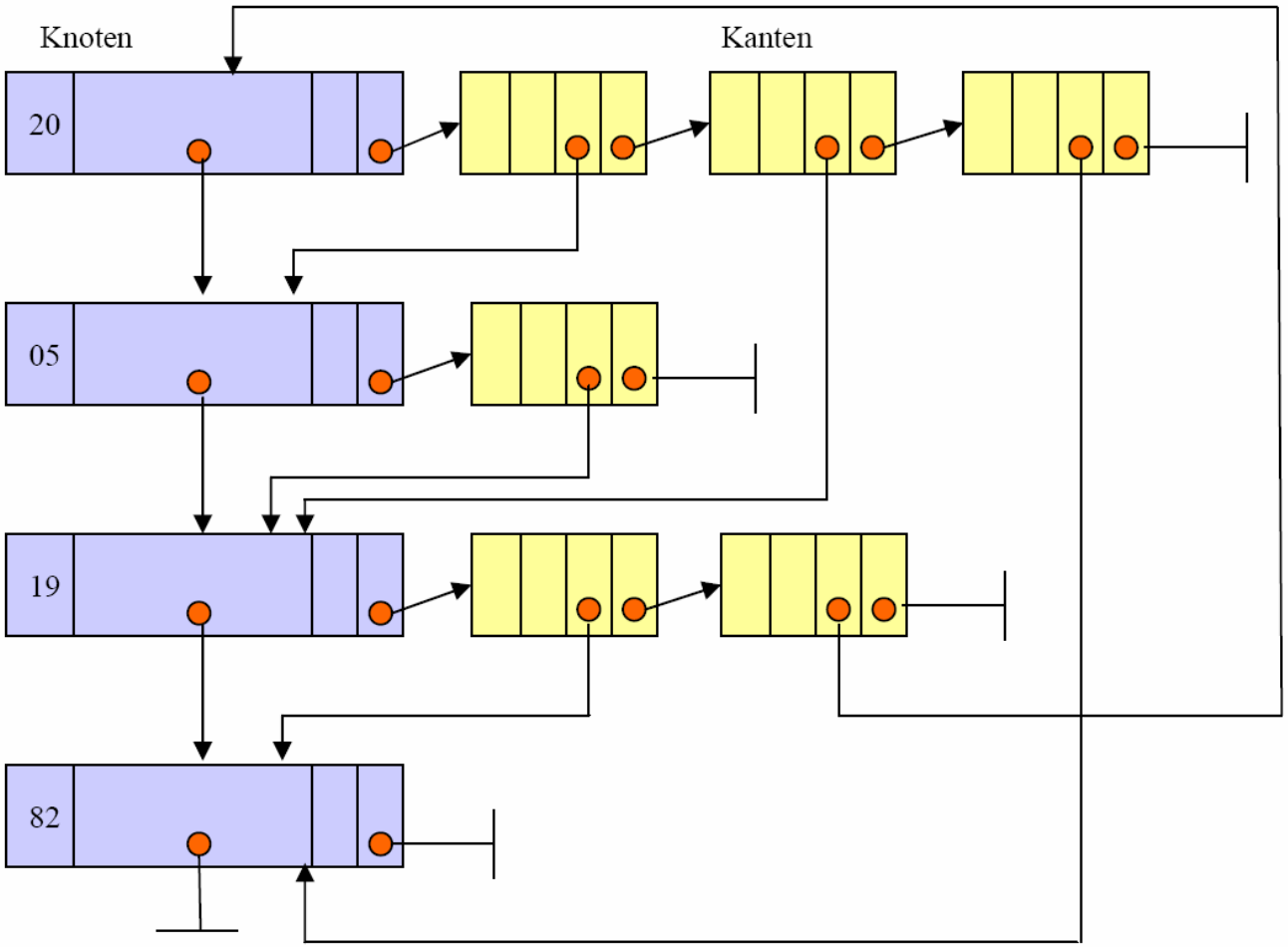
(5) "Pop" = Löschen des letzten Elements der Liste.

Die Realisierung durch Ada-Programmstücke ist einfach:



Aufgabe 3 Adjazenzlisten

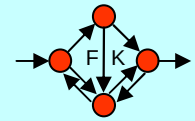
Wandeln Sie einen Graphen, der als Adjazenzmatrix gegeben ist, in eine Adjazenzliste um.



Beschreiben Sie Tiefensuche und Breitensuche in einem Graphen.

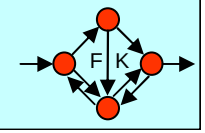
```
with Text_Io, Ada.Integer_Text_Io;
use Text_Io, Ada.Integer_Text_Io;

procedure Graph is

   type Rkante;
   type Pkante is access Rkante;
   type Rknoten;
   type Pknoten is access Rknoten;

   type Rkante is
      record
         Kanr       : Integer;
         End_Knoten : Pknoten;
         Dist       : Integer;
         Next       : Pkante;
      end record;

   type Rknoten is
      record
         Knnr  : Natural;
         Anker : Pkante;
         Next  : Pknoten;
      end record;
```

```
   Maxknoten : constant Positive := 10;
   type Adjazenzmatrix is array (1 .. Maxknoten, 1 .. Maxknoten) of Integer;
   A       : Adjazenzmatrix := (others => (others => 0));
   Aktkante : Natural       := 0;
   Anker    : Pknoten       := null;

   procedure Newknoten (
         Knnr : Natural ) is
      Newzeiger : Pknoten;
   begin
      Newzeiger:=new Rknoten'(Knnr, null, Anker);
      Anker:=Newzeiger;
   end Newknoten;

   procedure Newkante (
         Anfkn : Pknoten;
         Kanr  : Natural;
         Dist  : Integer;
         Endkn : Pknoten  ) is
      Newzeiger : Pkante;
   begin
      Newzeiger:=new Rkante'(Kanr, Endkn, Dist, Anfkn.Anker);
      Anfkn.Anker:=Newzeiger;
   end Newkante;
```



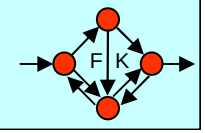
```
function Findknoten(Knnr   : Natural; Z:pKnoten) return Pknoten is
   Zeiger:pKnoten;
  begin Zeiger:=Anker;
   while Zeiger /= NULL loop
    if Zeiger.Knnr=Knnr then return Zeiger; end if;
    Zeiger:=Zeiger.Next;
   end loop;
   Put_Line("Knotennummer nicht gefunden");
   return null;
  end Findknoten;
```

```
procedure Matrix2liste is
    Pstart_Knoten,
    Pend_Knoten   : Pknoten := null;
  begin
    Anker:=null;
    for I in Adjazenzmatrix'range(1) loop
      Newknoten(I);
    end loop;

    for Start_Knoten in Adjazenzmatrix'range(1) loop
      for End_Knoten in Adjazenzmatrix'range(2) loop
        Pstart_Knoten:=Findknoten(Start_Knoten,Anker);
        Pend_Knoten:=Findknoten(End_Knoten,Anker);
        if Pstart_Knoten/=null and Pend_Knoten /= null then

Newkante(Pstart_Knoten,Aktkante,A(Start_Knoten,End_Knoten),Pend_Knoten);
        end if;
      end loop;
    end loop;
  end Matrix2liste;
```

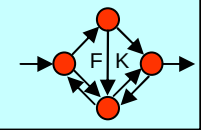


```
procedure Liste2matrix is
    procedure Knotensuche (
        Knoten_Zeiger : Pknoten ) is
      procedure Kantensuche (
          Kanten_Zeiger : Pkante ) is
      begin
        if Kanten_Zeiger /= null then

A(Knoten_Zeiger.Knnr,Kanten_Zeiger.End_Knoten.Knnr):=Kanten_Zeiger.Dist;
          Kantensuche(Kanten_Zeiger.Next);
        end if;
      end Kantensuche;
    begin
      if Knoten_Zeiger/=null then
        Kantensuche(Knoten_Zeiger.Anker);
        Knotensuche(Knoten_Zeiger.Next);
      end if;
    end Knotensuche;
  begin
    Knotensuche(Anker);
  end Liste2matrix;
```

```
procedure Init_Matrix is
  begin
    A(1,2):=1;
    A(1,4):=2;
    A(2,1):=3;
    A(3,1):=4;
    A(4,2):=5;
    A(4,5):=6;
    A(5,1):=7;
  end Init_Matrix;
```

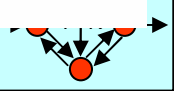


```
procedure Put (
        A : Adjazenzmatrix ) is
   begin
      New_Line;
      for I in Adjazenzmatrix'range(1) loop
         for J in Adjazenzmatrix'range(2) loop
            Put(A(I,J),2);
         end loop;
         New_Line;
      end loop;
      New_Line;
   end Put;

begin
   Put(A);
   Init_Matrix;
   Put(A);
   Matrix2liste;
   Put(A);
   A := (others => (others => 0));
   Put(A);
   Liste2matrix;
   Put(A);
end Graph;
```

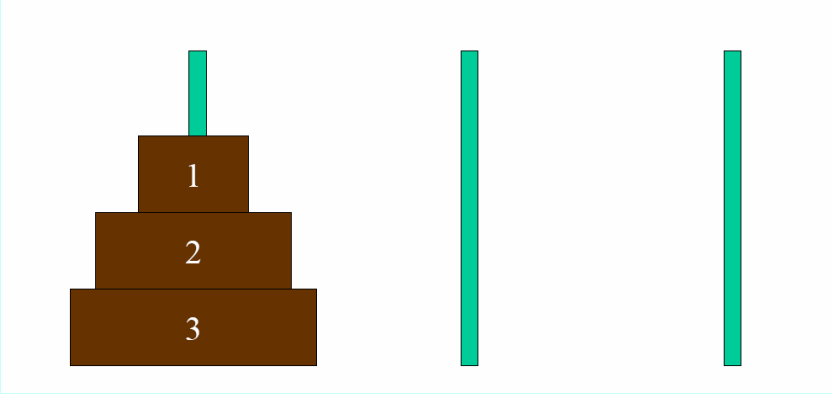


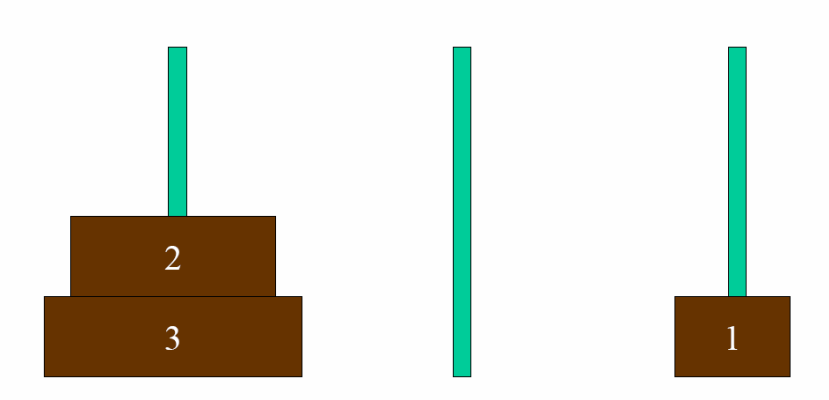
Aufgabe 1: (Rekursion) Die Türme von Hanoi

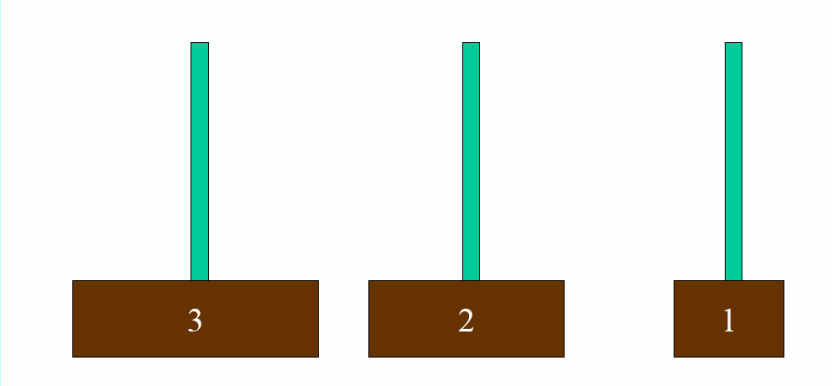
Die Legende, welche die Türme von Hanoi begleitet, besagt, dass unter der Herrschaft von Kaiser Fo Hi in Benares ein Tempel stand – mit einem Dom, der die Mitte der Welt markierte. Als die Welt erschaffen wurde, stellte Gott dort 3 diamantene Stäbe auf, die eine Elle lang und so dick wie 3 Körper einer Biene waren. Danach legte Gott 64 goldene Scheiben auf den ersten der Stäbe. Die Scheiben wurden immer kleiner, so dass sie zusammen eine Art Kegel bildeten.
Er trug den Mönchen auf, die goldenen Scheiben auf den dritten Stab zu transportieren. Der zweite Stab darf als "Zwischenlager" benutzt werden. Es darf aber niemals eine größere auf einer kleineren Scheibe zu liegen kommen und immer nur eine Scheibe auf einmal bewegt werden. In diesem Dom bewegen die Mönche seither goldene Scheiben zwischen diamantenen Stäben. Man sagt, dass das Ende der Welt kommt, wenn die Mönche mit ihrer Aufgabe fertig sind.
a) Geben Sie eine Lösung für n = 3 Scheiben an. Wie viele Züge benötigen Sie für die Lösung mindestens?
b) Überlegen Sie sich eine Formel, mit der Sie für jede beliebige Anzahl n von Scheiben bestimmen können, wie viele Züge mindestens notwendig sind, um den Stapel den Regeln entsprechend zu versetzen.
c) Überlegen Sie sich einen rekursiven Algorithmus, der das Problem löst. Es genügt dabei, wenn ihr Programm bzw. Ihr Pseudocode für jeden Schritt den Ausgangs- und den Zielstapel der zu bewegenden Platte ausgibt.

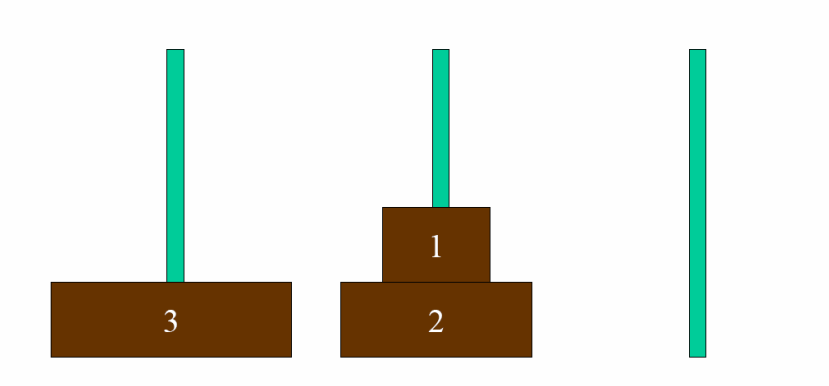


Beispiel: Die Türme von Hanoi mit 3 Scheiben (1. Teil)

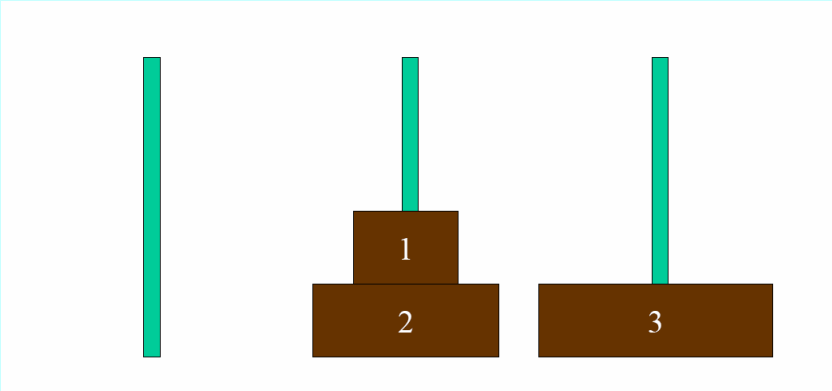


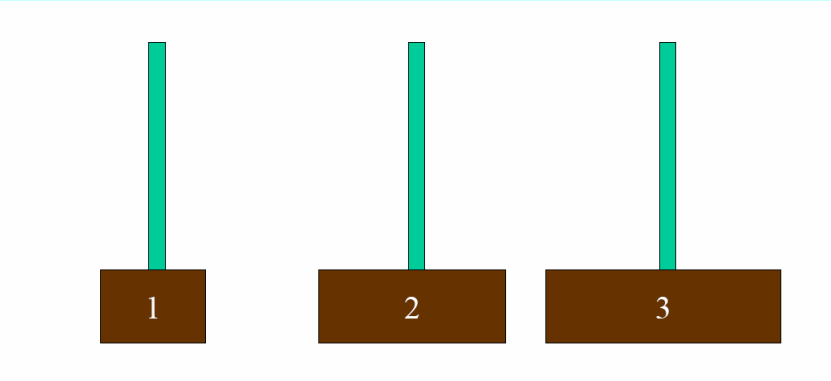


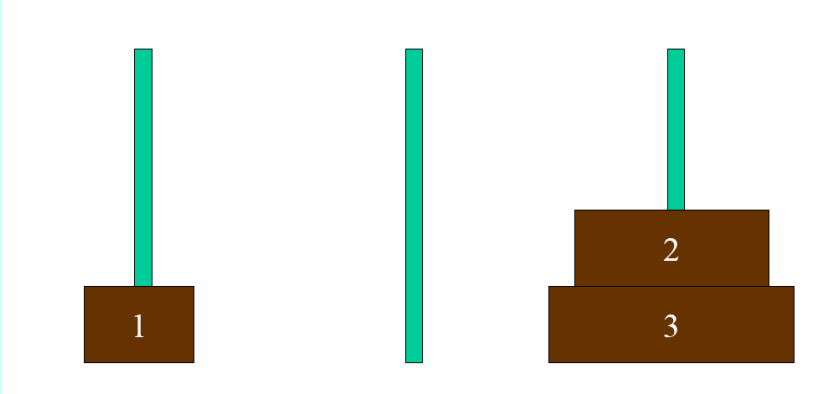


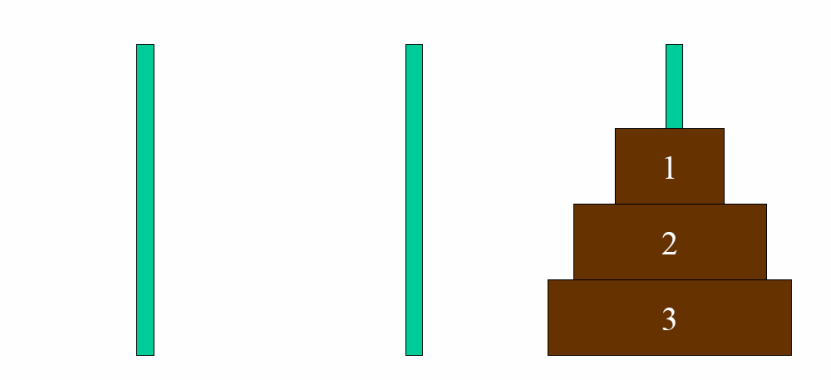


Beispiel: Die Türme von Hanoi mit 3 Scheiben (2. Teil)





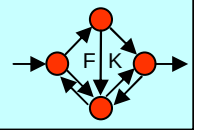




Rekursion:

1.) Bewege einen Turm der Höhe n-1 von A nach Z (über E)
2.) Bewege die letzte (oberste) Scheibe von A nach E
3.) Bewege einen Turm der Höhe n-1 von Z nach E (über A)

```
procedure Lege_oberste_Scheibe(A,E:Stab);
begin
  Welt[E]:=Welt[E]+Welt[A][length(Welt[A])];
  Welt[A]:=copy(Welt[A],1,length(Welt[A])-1);
  Anz:=Anz+1;
end;

procedure bewegeTurm(A, Z, E: Stab; h: integer); (* von über nach *)
begin
  if h > 0 then begin
    bewegeTurm(A, E, Z, h-1);
    Lege_oberste_Scheibe(A, E);
    bewegeTurm(Z, A, E, h-1);
  end;
end;
```



Aufgabe 2: (Rekursion) Summe für *expN(x,n)*

Gegeben sei die Exponentialreihe, die auf der Menge *IR* wie folgt definiert ist:

$$exp(x) = \sum_{n=0}^{\infty} \frac{x^n}{n!} \qquad \text{mit } x \in \mathbb{R}$$

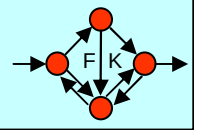
Implementieren Sie eine rekursive function *expN(x,n)*, die die Exponentialreihe fü die Eingabe *x* bis zum *n*-ten Glied berechnet.

Aufgabe 3 (Quersumme rekursiv)

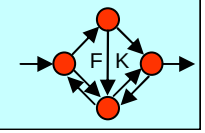
Implementieren Sie eine rekursive function *Quer*, die die Quersumme einer natürlichen Zahl berechnet.

Aufgabe 4 (Ackermann-Funktion)

Die Funktion *A(m, n)*ist durch folgende Rekursionen für natürliche Zahlen m und n definiert:
(i) $A(0, n) = n + 1$
(ii) $A(m, 0) = A(m-1, 1)$, *für* $m > 0$
(iii) $A(m, n) = A(m-1, A(m, n-1))$, *für* $m, n > 0$
Die Funktion *A* heißt Ackermann-Funktion. Implementieren Sie eine rekursive function *A*, die *A(m, n)* berechnet.



```
 x:=1; n:=6;
  Erg:=eReihe(1,1,0);


function eReihe(px,fak:double;i:integer):double;
begin
  if i<=n then eReihe:=px/fak+eReihe(px*x,fak*(i+1),i+1)
          else eReihe:=0;
end;
```



Aufgabe 4 (Ackermann-Funktion)

Die Funktion *A(m, n)*ist durch folgende Rekursionen für natürliche Zahlen m und n definiert:

(i) $A(0, n) = n + 1$

(ii) $A(m, 0) = A(m-1, 1)$, *für* $m > 0$

(iii) $A(m, n) = A(m-1, A(m, n-1))$, *für* $m, n > 0$

Die Funktion *A* heißt Ackermann-Funktion. Implementieren Sie eine rekursive function *A*, die *A(m, n)* berechnet.

Aufgabe 6 (Dezimal-/Dualkonvertierung)

Programmieren Sie eine **rekursive** Funktion, die als Parameter eine positive, ganze Zahl annimmt und einen String mit der Binärdarstellung dieser Zahl zurückliefert.

Beispiel: Parameter: 14 Ergebnis: “1110“
Parameter: 22 Ergebnis: “10110“

Aufgabe 5 (Buchstabenersetzung)

Es soll ein Wortspiel realisiert werden. Gegeben sind zwei Wörter gleicher Länge (frei wählbar durch den Nutzer). Gesucht werden weitere Wörter, die durch Austauschen eines Buchstabens das eine Wort in das andere überführen.

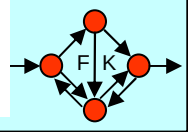
Beispiel: BEIN → BERN → BERG → WERG

Es soll ein neues Wort durch Austauschen des i-ten Buchstaben zusammengebaut werden. Beispiel:

Wort1=BEIN, Wort2=WERG 1 Bst. ersetzt: Zwischenergebnis = WEIN
Wort1=BEIN, Wort2=WERG 2 Bst. ersetzt: Zwischenergebnis = BEIN
Wort1=BEIN, Wort2=WERG 3 Bst. ersetzt: Zwischenergebnis = BERN
Wort1=BEIN, Wort2=WERG 4 Bst. ersetzt: Zwischenergebnis = BEIG (kein Wort)

Bereits implementiert sei eine Funktion IsWort(s:String) return boolean, die entscheidet, ob ein String s ein Wort ist oder nicht. Sie können das Prinzip der Rekursion verwenden.

Zusatzaufgabe: Es sollen auch Wörter gesucht werden, die einen „Umweg“ bedeuten, d.h. Buchstaben verwandt werden, die in keinem der beiden Wörter vorhanden sind.
Beispiel: BEIN → BERN → BERT → BELT (R ist nicht in beiden enthalten!)



Grammatik die arithmetische Ausdrücke berechnet

$E \rightarrow E + T$

$E \rightarrow E - T$

$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow T / F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow -F$

$F \rightarrow$ Zahl