Imperative Programmierung

mit dem

Java-Hamster-Modell

Autor: Dieter Lindenberg

Version 2008

Inhaltsverzeichnis

Grundbefehlssatz

Anweisungen:
```
vor();
linksUm();
nimm();
gib();
```

Blockanweisung:
```
{erster Befehl; zweiter Befehl;
.....letzter Befehl;}
```
wichtig: hinter einem Block steht kein Semikolon, weil innerhalb des Blockes hinter dem letzten Befehl schon eines steht.

Bemerkung: //

Boolesche Funktionen:
```
vornFrei()
maulLeer()
kornDa()
```

Kombinationen boolescher Funktionen:
```
Negation: !
Und:      &&
Oder:     ||
```

Prioritätenreihenfolge: **Klammern, !, &&, ||**

bedingte Anweisung:
```
if (Bedingung) Anweisung;
if (Bedingung) Anweisung;
else (Anweisung);
```

Beispiele:

a)
```
if (kornDa()) nimm;
```

b)
```
if (maulLeer()) gib(); else vor();
```

c)
```
if (vornFrei()) vor;
  else {
            linksUm();
            vor();
  }
```

d)
```
if (!kornDa() && !maulLeer()) gib(); else vor();
```

Die *while*-Schleife

```
while (Bedingung) Anweisung;
```

Beispiele:

a)
```
while (vornFrei()) vor();
```

b)
```
while (vornFrei() && !maulLeer())    {
          gib();
          vor();
   }   // Ende der while-Schleife
```

c)
```
while (vornFrei())          // Vorsicht: Endlosschleife?
       if (kornDa())     {
             nimm();
             vor();
       }   // Ende der if-Anweisung und der while-Schleife
```

Aufgaben

1. Der Hamster steht irgendwo im leeren Territorium mit beliebiger Blickrichtung. Er soll in eine beliebige Ecke laufen.
2. Der Hamster steht unten links auf der Grundlinie mit Blick nach Osten. Er soll bis zur rechten Wand laufen und dabei auf dem Boden liegende Körner aufsammeln. Auf jeder Kachel liegt höchstens 1 Korn.
3. Wie Aufgabe 2, aber auf jeder Kachel können beliebig viele Körner liegen.
4. Der Hamster steht unten links auf der Grundlinie mit Blick nach Osten. Im gesamten Territorium (5 Zeilen, beliebig viele Spalten) liegen auf beliebigen Kacheln jeweils beliebig viele Körner. Der Hamster soll alle Körner einsammeln.
5. Der Hamster steht irgendwo am linken Rand mit Blick nach Süden. Er hat Körner im Maul. Das Territorium ist leer. Der Hamster soll genau einmal am Rand des Territoriums herumlaufen.

Lösungen

```
1.void main() {
    while (vornFrei()) vor();
    linksUm();
    while (vornFrei()) vor();
  }
```

```
2.void main() {
    while (vornFrei())  {
      if (kornDa()) nimm();
      vor();
    }
    if (kornDa()) nimm();  // wichtig für die
                           // letzte Kachel
```

```
3.void main() {
    while (vornFrei())  {
      while (kornDa()) nimm();
      vor();
    }
    while (kornDa()) nimm();  // wichtig für die
  }                           // letzte Kachel
```

```
4.void main() {
    zeile();   // 1.Zeile
    linksUm();
    vor();
    linksUm();
    zeile();   // 2.Zeile
    rechtsUm();
    vor();
    rechtsUm();
    zeile();   // 3.Zeile
    linksUm();
    vor();
    linksUm();
    zeile();   // 4.Zeile
```

```
    rechtsUm();
    vor();
    rechtsUm();
    zeile();   // 5.Zeile
  }

  void zeile() {
    while (vornFrei())  {
      while (kornDa()) nimm();
      vor();
    }
    while (kornDa()) nimm();  // wichtig für die
                              // letzte Kachel
  }

  void rechtsUm()  {
    linksUm();
    linksUm();
    linksUm();
  }

5.void main() {
    gib();
    geheZurWand();   // runter
    linksUm();
    geheZurWand();   // nach rechts
    linksUm();
    geheZurWand();   // hoch
    linksUm();
    geheZurWand();   // nach links
    linksUm();
    while (!kornDa()) vor();
    nimm();
  }

  void geheZurWand() {
    while (vornFrei())  vor();
  }
```

Die *do-while*-Schleife

Beispiele:

```
a)do gib(); while (!maulLeer());

b)do  {
      gib();
      vor();
  }  while (vornFrei() && !maulLeer());
```

Aufgaben

1. Der Hamster steht unten links auf der Grundlinie mit Blick nach Osten. Im gesamten Territorium (5 Zeilen, beliebig viele Spalten) liegen irgendwo Körner herum, aber auf jeder Kachel höchstens eins. Der Hamster soll dieses Körnerbild invertieren.
2. Ersetze die folgende *while-Anweisung* durch eine *do-Anweisung*, aber so, dass in jedem möglichen Fall dasselbe bewirkt wird:
   while (kornDa()) nimm();
3. Ersetze die folgende *do-Anweisung* durch eine *while-Anweisung*, aber so, dass in jedem möglichen Fall dasselbe bewirkt wird:
   do nimm();while (kornDa())
4. Der Hamster befindet sich (mit beliebiger Blickrichtung) in einer Nische, d.h. an drei Seiten wird er von einer Mauer umgeben. Er soll einen Schritt aus dieser Nische herausgehen.
5. Der Hamster steht unten links auf der Grundlinie mit Blick nach Osten. Im gesamten, diesmal beliebig großen Territorium liegen irgendwo beliebig viele Körner herum. Der Hamster soll sie alle aufsammeln und anschließend in die Ausgangslage zurückkehren. Tip: der Hamster sollte jeweils eine Zeile säubern und immer an den Anfang dieser Zeile zurückgehen.

Lösungen

1.
```
void main() {
   invertiereZeile();  // 1.Zeile
   linksUm();
   vor();
   linksUm();
   invertiereZeile();  // 2.Zeile
   rechtsUm();
   vor();
   rechtsUm();
   invertiereZeile();  // 3.Zeile
   linksUm();
   vor();
   linksUm();
   invertiereZeile();  // 4.Zeile
   rechtsUm();
   vor();
   rechtsUm();
   invertiereZeile();  // 5.Zeile
  }

  void invertiereZeile() {
    while (vornFrei()) {
      if (kornDa()) nimm(); else gib();
      vor();
    }
    if (kornDa()) nimm(); else gib();
  }

  //  void invertiereZeile() {          Alternative
  //    do {
  //      if (kornDa()) nimm(); else gib();
  //      vor();
  //    }  while (vornFrei());
  //    if (kornDa()) nimm(); else gib();
  //  }
```

2.
```
if (kornDa())
    do nimm(); while (kornDa());
```

3.
```
nimm();
while (kornDa()) nimm();
```

4.
```
void main() {
  do linksUm(); while (!vornFrei());
  vor();
}
```

5.
```
void main() {
  linksUm();
  while (vornFrei()) {
    sammleZeile();
    vor();
  }
  sammleZeile();   // oberste Zeile
}

void sammleZeile() {
  rechtsUm();
  while (vornFrei()) {
    while (kornDa()) nimm();
    vor();
  }
  while (kornDa()) nimm();  // letzte Kachel
                            // rechts

  linksUm();
  linksUm();
  while (vornFrei()) vor();
  rechtsUm();
}
```

Boolesche Methoden

Boolsche Methoden liefern einen Wahrheitswert (*true* oder *false*).

Beispiel:

```
boolean linksFrei()   {
            linksUm();
            if (vornFrei())  {
               rechtsUm();
               return true;
            }  else   {
                  rechtsUm();
                  return false;
            }
}
```

Benutzt werden kann diese Methode etwa folgendermaßen:

```
if linksFrei()  {
   linksUm();
   vor();
}
```

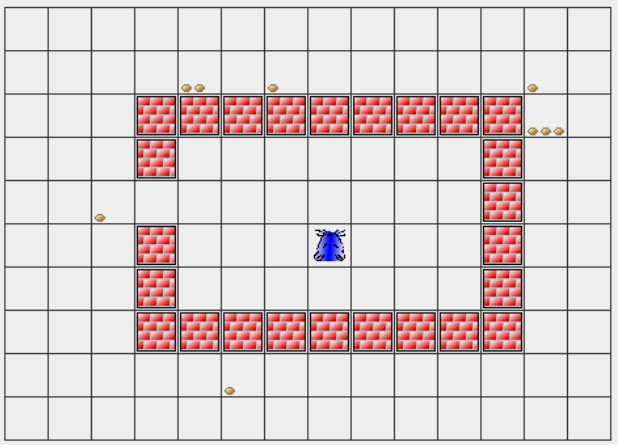
Die Ausführung der *return-Anweisung* führt zur unmittelbaren Beendigung der Funktion. Dabei wird der entsprechende Wahrheitswert als sog. *Funktionswert* zurückgegeben.

Boolsche *return-Anweisungen* dürfen nur innerhalb Boolscher Methoden verwendet werden!

In jedem möglichen Weg durch die Boolsche Methode muss eine *return-Anweisung* stehen, weil es ansonsten kein Funktionsergebnis geben könnte.

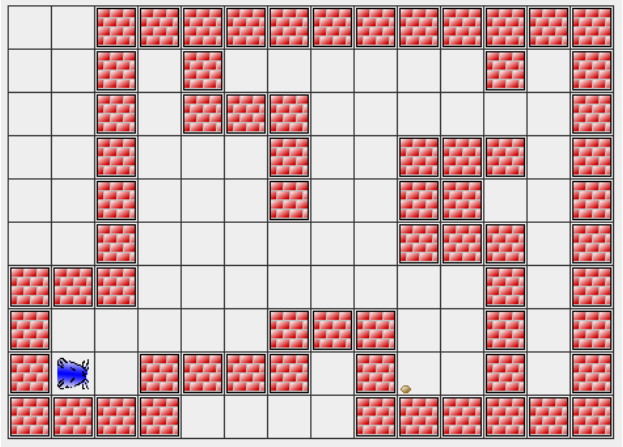
Der Programmierer sollte dringend darauf achten, dass eine Boolsche Funktion nur einen Wahrheitswert liefert und ansonsten den Zustand des Territoriums und des Hamsters nicht ändert. Falls also z.B. der Hamster zu Beginn der Funktion seine Blickrichtung ändert oder einen Schritt vorwärts macht, so muss er dies unbedingt vor Ende der Methode wieder rückgängig machen.

Aufgaben

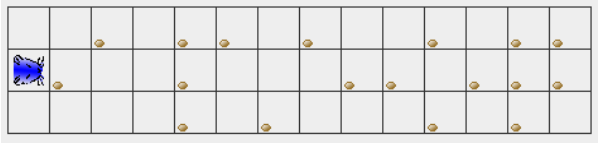


1. Der Hamster befindet sich in seinem Bau an beliebiger Stelle mit beliebiger Blickrichtung. Allerdings schaut er nicht direkt in die Richtung des Ausgangs. Um seinen Bau herum liegen Körner. Der Hamster soll sie alle auflesen und in der linken unteren Ecke seines Baues ablegen.

2. Der Hamster befindet sich, wie dargestellt, unten links mit beliebiger Blickrichtung in einem beliebigen Labyrinth, welches allerdings keine „Inseln“ enthält. Er soll immer links an der Wand entlang laufen, bis er das Korn findet.



3. Der Hamster soll solange vorwärts gehen, bis er sich im sog. Körnerparadies befindet. Dies ist ein Ort, an dem sich links, vor, rechts und hinter dem Hamster jeweils ein Korn befindet. Voraussetzung: das Territorium enthält mindestens ein Körnerparadies.



Lösungen

Aufgabe 1

```
void main() {
  hinausGehen(); // Hamster geht einen Schritt hinaus
  if (kornDa()) nimm();
  linksUm();
  vor();
  aussenWandEntlang();   // nach Süden
  aussenWandEntlang();   // nach Osten
  aussenWandEntlang();   // nach Norden
  aussenWandEntlang();   // nach Westen
  aussenWandEntlang();   // nach Süden
  vor();                 // hinein
  rechtsUm();
  while (vornFrei()) vor();
  while (!maulLeer()) gib();
}


void hinausGehen() {
 while (vornFrei()) vor();
 while (!vornFrei()) rechtsUm();
 while (!linksFrei()) einSchrittAnWandEntlang();
 linksUm();
 vor();
 vor();
}


void einSchrittAnWandEntlang()  {
  if (vornFrei()) vor();
  else rechtsUm();
}


boolean linksFrei()  {
  linksUm();
  if (vornFrei()) {
    rechtsUm();
    return true;
  }
```

```
  else {
    rechtsUm();
    return false;
  }
}


void rechtsUm() {
  linksUm();
  linksUm();
  linksUm();
}


void aussenWandEntlang()  {
  if (kornDa()) nimm();
  while (!linksFrei()) schrittAnAussenWandEntlang();
  linksUm();
  vor();
}


void schrittAnAussenWandEntlang()  {
  vor();
  if (kornDa()) nimm();
}
```

Aufgabe 2

```
void main() {
  drehInLaufrichtung();
  while (!kornDa()) einSchrittAnDerLinkenWandEntlang();
}


void einSchrittAnDerLinkenWandEntlang()  {
  if (!linksFrei() && vornFrei()) vor();
  else  {
    if (!linksFrei() && !vornFrei()) rechtsUm();
    else {
       linksUm();
       vor();
    }
  }
}


void drehInLaufrichtung()  {
  while (!(!linksFrei() && vornFrei())) linksUm();
}


boolean linksFrei()  {
  linksUm();
  if (vornFrei())  {
    rechtsUm();
    return true;
  }
  else {
    rechtsUm();
    return false;
  }
}


void rechtsUm()  {
  linksUm();
  linksUm();
  linksUm();
}
```

Aufgabe 3

```
void main() {
  do vor(); while (!paradiesDa());
}


boolean paradiesDa()  {
  return  kornVornDa() && kornLinksDa()  &&
                  kornRechtsDa()  && kornHintenDa();
}


boolean kornVornDa()  {
  vor();
  if (kornDa())  {
     linksUm();
     linksUm();
     vor();
     linksUm();
     linksUm();
     return true;
  }
  else {
     linksUm();
     linksUm();
     vor();
     linksUm();
     linksUm();
     return false;
 }
}


boolean kornLinksDa()  {
  linksUm();
  vor();
  if (kornDa())  {
     linksUm();
     linksUm();
     vor();
     linksUm();
     return true;
  }
```

```
  else {
     linksUm();
     linksUm();
     vor();
     linksUm();
     return false;
 }
}


boolean kornRechtsDa()  {
  linksUm();
  linksUm();
  linksUm();
  vor();
  if (kornDa())  {
     linksUm();
     linksUm();
     vor();
     linksUm();
     linksUm();
     linksUm();
     return true;
  }
  else {
     linksUm();
     linksUm();
     vor();
     linksUm();
     linksUm();
     linksUm();
     return false;
 }
}


boolean kornHintenDa()  {
  linksUm();
  linksUm();
  vor();
  if (kornDa())  {
     linksUm();
     linksUm();
     vor();
```

```
      return true;
   }
   else {
      linksUm();
      linksUm();
      vor();
      return false;
 }
}
```

Lösung 11.7.5 Aufgabe 5

```
void main() {
  gehBisWandOderAbgrund();
  while (!rechtsFrei())  {
    stufe();
    gehBisWandOderAbgrund();
  };
  linksUm();
  linksUm();
  vor();
}


void gehBisWandOderAbgrund()  {
  while(!(!vornFrei() || rechtsFrei())) vor();
}


boolean rechtsFrei()  {
  linksUm();
  linksUm();
  linksUm();
  if (vornFrei())  {
    linksUm();
    return true;
  }
  else  {
    linksUm();
```

```
    return false;
  }
}


void stufe()  {
  linksUm();
  do vor(); while (!rechtsFrei());
  linksUm();
  linksUm();
  linksUm();
  vor();
}
```

Lösung 11.7.13 Aufgabe 13

```
void main() {
  while (kornDa()) folgeDerSpurEinenSchritt();
}


void folgeDerSpurEinenSchritt()  {
  nimm();
  if (kornVornDa()) vor();
  else  if (kornRechtsDa()) rechts();
        else  if (kornLinksDa())  links();
}


boolean kornVornDa()  {
  if (!vornFrei()) return false;
  else {
     vor();
     if (kornDa()) {
       linksUm();
       linksUm();
       vor();
       linksUm();
       linksUm();
       return true;
```

```
      }
      else  {
        linksUm();
        linksUm();
        vor();
        linksUm();
        linksUm();
        return false;
      }
  }
}


boolean kornRechtsDa()  {
  linksUm();
  linksUm();
  linksUm();
  if (!vornFrei())  {
    linksUm();
    return false;
  }
  else {
     vor();
     if (kornDa()) {
       linksUm();
       linksUm();
       vor();
       linksUm();
       linksUm();
       linksUm();
       return true;
     }
     else  {
      linksUm();
      linksUm();
      vor();
      linksUm();
      linksUm();
      linksUm();
      return false;
    }
  }
}
```

```
boolean kornLinksDa()  {
  linksUm();
  if (!vornFrei())  {
    linksUm();
    linksUm();
    linksUm();
    return false;
  }
  else {
     vor();
     if (kornDa()) {
       linksUm();
       linksUm();
       vor();
       linksUm();
       return true;
     }
    else  {
      linksUm();
      linksUm();
      vor();
      linksUm();
      return false;
    }
  }
}


void rechts()  {
  linksUm();
  linksUm();
  linksUm();
  vor();
}


void links()  {
  linksUm();
  vor();
}
```

Boolesche Variablen

- Variablen müssen Namen zugewiesen werden, damit sie im Programm angesprochen werden können. Außerdem muss dem Rechner der Datentyp der Variablen mitgeteilt werden, damit er weiß, wie viel Speicherplatz reserviert werden muss, und in welcher Form die Werte gespeichert werden sollen. Diese Angaben bezeichnet man als *Deklaration*. Aufgrund dieser *Deklaration* wird ein Bereich des Hauptspeichers des Rechners für sie reserviert, in dem der aktuelle Wert der Variablen abgelegt werden kann.
- Variablen sollten (möglichst schon bei ihrer Deklaration) initialisiert werden, d.h. ihnen sollte ein Wert zugewiesen werden, damit sie jederzeit einen definierten Wert enthalten. Diesen Vorgang bezeichnet man als Variablen-*Definition*.

Beispiele für boolesche Variablendefinitionen:

```
- boolean gefunden = true;
- boolean fertig = false;
- boolean test = kornDa() && !maulLeer() && !fertig
- boolean test1 = true, test2 = vornFrei(),
          test3 = gefunden || fertig;
- boolean b1, b2 = true, b3;
```

Fehlt bei einer boolschen Variablendeklaration der Initialisierungswert, so wird automatisch der Anfangswert *false* zugewiesen! In dem letzten Beispiel erhalten also die Variablen b1 und b3 automatisch den Anfangswert *false*.

Die schon öfter benutzte boolesche Methode linksFrei() lässt sich nun folgendermaßen vereinfachen:

```
boolean linksFrei() {
  linksUm();
  boolean istFrei = vornFrei();
  rechtsUm();
  return istFrei;
}
```

Beispiel: Falls auf dem Weg des Hamsters zur Wand ein Korn liegen sollte, soll er sich direkt vor der Wand einmal um sich selbst drehen.

```
void main() {
      if (kornWarDa()) {
        linksUm(); linksUm(); linksUm(); linksUm();
      }
}

boolean kornWarDa() {
  boolean gefunden = false;
  while (vornFrei()) {
    if (kornDa()) {
      gefunden = true;
    }
    vor();
  }
  if (kornDa()) {  // letzte Kachel vor der Wand
    gefunden = true;
  }
  return gefunden;
}
```

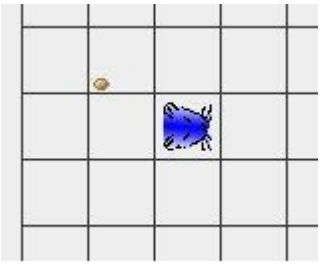
Beispiel: Falls der Hamster eine gerade Anzahl Körner im Maul hat, soll er einen Schritt vorgehen.

```
void main() {
   if (geradeAnzahl()) vor();
}

boolean geradeAnzahl() {
  boolean gerade = true;

  while (!maulLeer()) {
    gib();
    gerade = !gerade;
  }
  while (kornDa()) nimm();
  return gerade;
}
```

Beispiel: Der Hamster soll feststellen, ob auf einem seiner Nachbarfelder ein Korn liegt oder nicht.



```
void main() {
  if (test()) {vor(); vor(); vor();}
}

boolean test() {
  boolean kornVorhanden = false;

  vor();
  kornVorhanden = kornDa();
  linksUm();
  vor(); // jetzt oben rechts
  kornVorhanden = kornVorhanden || kornDa();
  linksUm();
  vor();
  kornVorhanden = kornVorhanden || kornDa();
  vor();// jetzt oben links
  kornVorhanden = kornVorhanden || kornDa();
  linksUm();
  vor();
  kornVorhanden = kornVorhanden || kornDa();
  vor();// jetzt unten links
  kornVorhanden = kornVorhanden || kornDa();
  linksUm();
  vor();
  kornVorhanden = kornVorhanden || kornDa();
  vor();// jetzt unten rechts
  kornVorhanden = kornVorhanden || kornDa();
  linksUm();
  vor();
  linksUm();
  vor();
  linksUm();
  linksUm();

  return kornVorhanden;
}
```

Gültigkeitsbereich einer booleschen Variablen

Sog. ***lokale*** Variablen werden innerhalb eines Blockes deklariert. Ein Block wird markiert durch eine öffnende und eine zugehörige, schließende geschweifte Klammer.
Der Gültigkeitsbereich einer lokalen Variablen erstreckt sich von der der Variablendeklaration folgenden Anweisung bis zum Ende des Blockes und umschließt alle inneren Blöcke.
Eine lokale Variable kann also z.B. auch nur innerhalb einer *if*-Anweisung gültig sein.
Bemerkungen:

- Angenommen, innerhalb eines Blockes wird eine Variable *v1* deklariert und innerhalb desselben Blockes wird eine Prozedur aufgerufen. Dann gilt die Variable *v1* natürlich nicht innerhalb der aufgerufenen Prozedur!
- wenn man innerhalb des Anweisungsblockes einer *while*-Schleife eine Variable deklariert, so handelt es sich bei jedem Blockdurchlauf um eine andere, neue Variable, die nur denselben Namen hat wie die vorherige! Die vorherige Variable wurde mit der schließenden Klammer *}* gelöscht.

Sog. ***globale*** Variablen werden im Deklarationssteil eines Programms festgelegt, also neben den Prozeduren und Funktionen.
Globale Variablen gelten im gesamten Programm!

Man sollte die Definitionen aller Variablen möglichst am Anfang (entweder des Blockes oder des gesamten Programmes) platzieren. Das erhöht die Übersichtlichkeit der Programme.

Alle Variablen werden in der Reihenfolge initialisiert, in der sie deklariert werden. Man kann also in der Initialisierung einer Variablen nicht auf Werte einer anderen Variablen zurückgreifen, die erst weiter unten deklariert wird.
Beispiele:

```
boolean b1 = true;
boolean b2 = !b3;   // Fehler
boolean b3 = !b1;   // erlaubt
```

Aufgaben

1. Der Hamster steht in der linken, unteren Ecke seines Territoriums mit Blick nach Osten. Nur wenn sich auf seiner Kachel eine ungerade Anzahl von Körnern befindet, soll er (ohne die Körner) zur nächsten Wand laufen. Ansonsten soll er sich einmal um sich selbst drehen.
2. Der Hamster steht links unten auf der Grundlinie mit Blick nach Osten. Er soll bis zur Wand laufen und dabei alle Körner, die er unterwegs findet, einsammeln. Nur wenn er eine gerade Anzahl an Körnern eingesammelt hat, soll er alle Körner, die er im Maul hat, an der Mauer ablegen.
3. Der Hamster soll testen, ob er eingemauert ist (also an vier Seiten direkt von einer Mauer umgeben) oder ob er sich nur in einer engen Nische befindet (nur an drei Seiten befindet sich eine Mauer). Im letzteren Fall soll er einen Schritt aus der Nische herausgehen.
4. Der Hamster steht unten links in der Ecke mit Blick nach Norden. Er soll untersuchen, ob irgendwo in seinem Territorium ein Korn liegt. Er soll das Korn, falls eines irgendwo herumliegt, liegenlassen. Anschließend kehrt er zurück in die Ecke unten links. Das Territorium enthält keine Mauern. Nur wenn der Hamster ein Korn gesehen hat, soll er sich zum Schluss einmal um sich selbst drehen.
5. Der Hamster steht links unten auf der Grundlinie mit Blick nach Osten. Er hat keine Körner im Maul. Auf der gesamten Grundlinie liegen auf jeder Kachel beliebig viele Körner. Der Hamster soll dafür sorgen, dass auf jeder Kachel eine gerade Anzahl Körner liegen (eventuell Null), indem er abwechselnd gegebenenfalls entweder ein Korn aufnimmt oder abgibt.

Integer-Variablen

Eine ganze Zahl vom Datentyp ***int*** benötigt 4 Bytes Speicherplatz. Man kann also 2^{32} unterschiedliche Zahlen darstellen. Als Wertebereich werden alle ganzen Zahlen von $-2^{31} = -2\,147\,483\,648$ bis $2^{31} - 1 = 2\,147\,483\,647$ gewählt.

Beispiele:
```
int n = 36;
int zahl = -450;
int i, k, l = 7, p;
int max = -2*(30-4);
int min = -max -1;
```

Fehlt der Initialisierungswert, so erhält die *int*-Variable automatisch den Anfangswert 0.

Operatoren: Es existieren insgesamt fünf arithmetische Operatoren.

Addition +
Subtraktion −
Multiplikation *
Ganzzahldivision / *dabei werden Nachkommastellen ignoriert.*
Beispiel: 7 / 2 = 3 und −9 / 2 = −4
Modulo % *liefert den Rest bei einer Ganzzahldivision.*
Beispiel: 16 % 3 = 1

Es gilt die übliche Prioritätenregel ***Punkt-vor-Strichrechnung***.

Natürlich stürzt das Hamsterprogramm ab, wenn man versucht, durch 0 zu dividieren. Das wäre das gleiche, als wenn man den Hamster gegen eine Wand rennen lässt.

Probleme gibt es bei einer Zahlenbereichsüberschreitung.
Beispiel: 2 147 483 647 + 10 kann Java nicht richtig ausrechnen. Leider erfolgt hier keine Fehlermeldung sondern Java rechnet im negativen Zahlenbereich weiter.

Alternative Zuweisungsoperatoren: **i++ ⇔ i = i+1** *Inkrement-Operator*
i-- ⇔ i = i-1 *Dekrement-Operator*

Vergleichsausdrücke:

```
x == y   Gleichheitssoperator
x != y   Ungleichheitsoperator
x < y
x <= y
x > y
x >= y
```

Beispiel: `if (x != 0) {y = y/x ;}`

Vergleichsoperatoren haben eine niedrigere Priorität als arithmetische Operatoren. Deshalb benötigt man im folgenden Beispiel keine zusätzlichen Klammern: `if (x == 5*y+3) {…}`

Im folgenden Hamster-Programm dreht sich der Hamster genau viermal linksum:

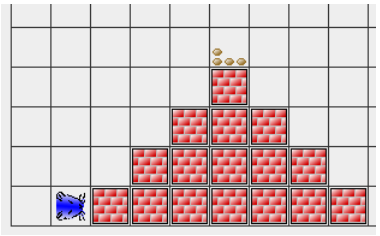
```
void main() {
   int anzahl = 0;
   while (anzahl < 4) {
      linksUm();
      anzahl++;
   }
}
```

Beispiel: Der Hamster steht irgendwo in seinem Territorium. Er soll geradeaus bis zur Wand laufen und anschließend an den Ausgangsort zurückkehren.

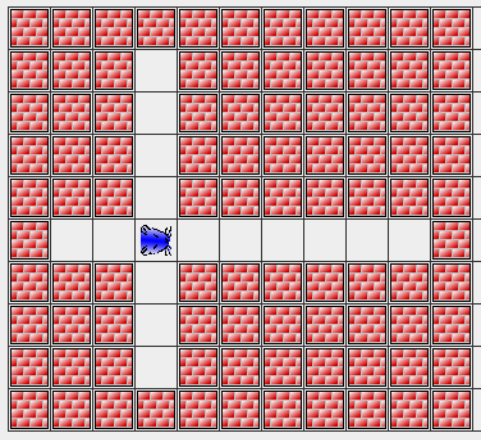
```
void main ()  {
   int schritte = 0;
   while (vornFrei())  {
      vor();
      schritte++;
   }
   linksUm();
   linksUm();
   while (schritte > 0)  {
      vor();
      schritte--;
   }
}
```

Aufgaben

1. Der Hamster steht unten links in der Ecke mit Blick nach Osten. Er soll einmal sein gesamtes Territorium am Rand umrunden. Löse diese Aufgabe mit Hilfe einer Zählvariablen so: der Hamster soll 4 mal Folgendes machen: er geht bis zur nächsten Wand und dreht sich dann einmal links herum.
2. Mithilfe des Modulo-Operators lässt sich einfach feststellen, ob eine Zahl gerade ist oder nicht.
   Der Hamster steht unten links in der Ecke mit Blick nach Osten. Er soll bis zur rechten Wand laufen. Falls die Anzahl der Kacheln auf der Grundlinie gerade ist, soll er sich 10 mal um sich selbst drehen.
3. Der Hamster hat eine bestimmte (aber unbekannte) Anzahl Körner im Maul. Außerdem befinden sich auf der Kachel, auf der er gerade steht, ebenfalls einige Körner. Falls er mehr Körner im Maul hat als sich Körner auf seiner Kachel befinden, soll er einen Schritt vorgehen (vorne ist frei). Auf der Kachel sollen anschließend wieder genauso viele Körner liegen wie vorher auch.
4. Der Hamster steht unten links auf der Grundlinie mit Blick nach Osten und hat eine Anzahl von Körnern im Maul, welche ein Vielfaches der Zahl 4 ist. Im gesamten Territorium befinden sich keine weiteren Körner. Der Hamster soll nun alle seine Körner auf die vier Ecken aufteilen, d.h. in jeder der vier Ecken sollen anschließend gleich viele Körner liegen.
5. Der Hamster befindet sich unten links auf der Grundlinie mit Blick nach Osten. Er hat bereits einige (aber eine unbekannte Anzahl) Körner im Maul. Auf der Grundlinie liegen auf zufälligen Kacheln zufällig viele Körner. Der Hamster soll sie alle aufsammeln. Nachdem er anschließend rechts angekommen ist, soll er, falls er mehr als 20 Körner im Maul hat, diese alle in der rechten Ecke ablegen.
6. Der Hamster steht unten direkt vor einer beliebig hohen Treppe. Er soll ganz oben auf der Treppe genau so viele Körner ablegen wie die Treppe Stufen hat.



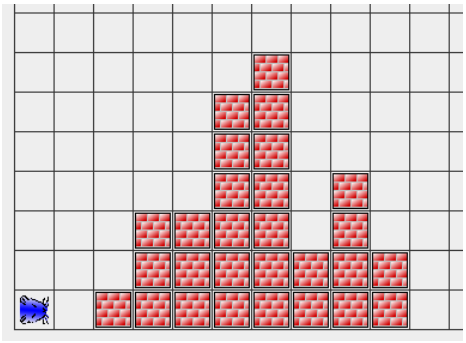
7. Der Hamster steht auf der Grundlinie unten links mit Blick nach Osten. Er hat eine ihm unbekannte Zahl Körner im Maul. Diese Zahl ist (im Dezimalsystem) dreistellig. Er soll feststellen, wie viele Körner er im Maul hat, und dann auf die ersten drei Kacheln unten links genau die Anzahl Körner legen, die der entsprechenden Stelle der Zahl entspricht. Hat er zum Beispiel 537 Körner im Maul, so sollte er auf die linke Kachel 5, auf die mittlere 3 und auf die dritte Kachel 7 Körner legen. Der Hamster steht anschließend auf der vierten Kachel.
8. Der Hamster steht unten links in der Ecke mit Blick nach Osten. Er hat genügend Körner im Maul. Er soll die Grundlinie seines Territoriums mit Körnern markieren. Auf der ersten Kachel soll ein Korn liegen, auf der zweiten zwei Körner, auf der dritten drei Körner usw.
9. Der Hamster steht unten links in der Ecke mit Blick nach Osten. Er hat genügend Körner im Maul. Er soll die Grundlinie seines Territoriums mit Körnern markieren. Auf der ersten Kachel soll ein Korn liegen, auf der zweiten zwei Körner, auf der dritten vier Körner, auf der vierten acht Körner usw.
10. Das Territorium zeigt, wie der Hamster an einer Wegkreuzung steht. Vor, rechts, hinter und links von ihm befinden sich verschieden lange Gänge (unbekannter Länge). Der Hamster soll zum Ende des längsten Ganges gehen und dort stehen bleiben.



11. Der Hamster steht unten links in der Ecke mit Blick nach Osten. Auf den einzelnen Kacheln der Grundlinie können beliebig viele Körner liegen. Er soll die Kachel mit den meisten Körnern suchen und diese fressen.

12. Der Hamster steht unten links in der Ecke mit Blick nach Osten. Er hat eine beliebige Anzahl an Körnern im Maul. Auf den einzelnen Kacheln der Grundlinie können ebenfalls jeweils beliebig viele Körner liegen. Die Aufgabe des Hamsters besteht darin, eine Kachel zu suchen, auf der genauso viele Körner liegen, wie er im Maul hat. Auf einer solchen Kachel soll er stehen bleiben.
    Es soll vorausgesetzt werden, dass es auf der Grundlinie mindestens eine solche Kachel gibt.

13. Der Hamster steht unten links in der Ecke mit Blick nach Osten. Er hat eine beliebige Anzahl an Körnern im Maul. Auf den einzelnen Kacheln der Grundlinie können ebenfalls jeweils beliebig viele Körner liegen. Der Hamster soll dafür sorgen, dass auf jeder dieser Kacheln eine durch 3 teilbare Anzahl an Körnern liegt.
    Dabei soll er folgendermaßen vorgehen: Wenn er eine Kachel mit einer nicht durch 3 teilbaren Anzahl an Körnern entdeckt, soll er, falls er noch genügend Körner im Maul hat, so viele Körner ablegen, dass anschließend die Körneranzahl der Kachel durch 3 teilbar ist.
    Andernfalls soll er soviele Körner aufnehmen, dass anschließend die Körnerzahl auf der Kachel durch 3 teilbar ist.

14. *(schwierig und umfangreich!)*
    
    Der Hamster klettert über ein Gebirge, bis er sich wieder auf der Ausgangshöhe befindet. Er gibt allen Kacheln Koordinaten. Er startet im Punkt (0; 0). Er merkt sich die Koordinaten des höchsten Punktes. Er hat das Gebirge überquert, wenn er wieder auf der Höhe 0 ist. Nach der Überquerung kehrt er zum höchsten Punkt des Gebirges zurück.

Die *for*-Schleife

Die *for*-Schleife wird immer dann eingesetzt, wenn man vorher schon weiß, wie oft genau die Schleife durchgeführt werden soll.

Im folgenden Beispiel geht der Hamster genau 5 Schritte vor:

```
for (int a=1; a <=5; a++)  {
    vor();
}
```

Im nächsten Beispiel wird die Summe der ersten 10 natürlichen Zahlen berechnet:

```
int summe = 0;
for (int b=10; b>=0; b--)  {
     summe = summe + b;
}
```

Im nächsten Beispiel wird das Produkt aller ungeraden Zahlen zwischen 10 und 20 berechnet:

```
int ergebnis = 1;
for (int c=11; c<=20; c=c+2)  {
     ergebnis = ergebnis * c;
}
```

Allgemein sieht die for-Schleife folgendermaßen aus:

```
for (Initialisierung; Laufbedingung; Schrittweite)    {
    auszuführende Befehle;
}
```

Aufgaben

1. Der Hamster soll genau 30 Körner abgeben.

2. Der Hamster soll sich 12 mal links drehen.

3. Der Hamster soll sich zu der Kachel bewegen, die sich 5 Schritte diagonal rechts oben von ihm befindet.

4. Der Hamster soll 6 Schritte vorgehen. Löse diese Aufgabe
a) mit einer while-Schleife,
b) mit einer do-Schleife,
c) mit einer for-Schleife!

5. Der Hamster soll genau 7 Schritte vorgehen und nach jedem Schritt jedesmal 10 Körner abgeben. Hinweis: man kann *for-Schleifen* auch ineinander schachteln.

6. Der Hamster soll ein Quadrat der Kantenlänge 8 ablaufen.

7. Der Hamster soll sein Territorium 3 mal am Rand umrunden.

8. Der Hamster soll die Summe aller natürlichen Zahlen von 1 bis 100 berechnen. Das richtige Ergebnis ist übrigens 5050. Falls der Hamster richtig gerechnet hat, soll er sich vor Freude 3 mal um sich selbst drehen, ansonsten soll er zur Buße alle seine Körner, die er im Maul hat, abgeben.

Lösungen

```
5.   void main() {
     for (int a=1; a<=7; a++)  {
        vor();
        for (int b=1; b<=10; b++)  {
          gib();
        }
     }
   }
```

```
6.   void main() {
       for (int a=1; a<=4; a++)  {
         for (int b=1; b<=7; b++)  vor();
         linksUm();
       }
     }
```

```
7.   void main() {
        for (int a=1; a<=12; a++)  {
           while (vornFrei())  vor();
           linksUm();
        }
     }
```

```
8.   void main() {
       int summe = 0;
       for (int n=1; n<=100; n++)  {
          summe = summe + n;
       }
       if (summe == 5050)  {
          for (int i=1; i<=12; i++)  linksUm();
       }
       else while (!maulLeer()) gib();
}
```

Integer-Methoden

Integer-Methoden liefern eine ganze Zahl.

Beispiel:

```
int anzahlKoernerImMaul()   {
    int hilf, anzahl = 0;
    while (!maulLeer())  {
      gib();
      anzahl++;
    }
    hilf = anzahl;
    while (hilf > 0)  {
      nimm();
      hilf--;
    }
    return anzahl;
}
```

Benutzt werden kann diese Funktion etwa folgendermaßen:

```
   if (anzahlKoernerImMaul() > 1)  {
      gib();
      gib();
   }
```

Am häufigsten wird eine Funktion allerdings in Form einer Zuweisung benutzt:

```
   int hilf;
   hilf = anzahlKoernerImMaul();
   ……..
```

Die Ausführung der *return-Anweisung* führt zur unmittelbaren Beendigung der Funktion. Dabei wird der entsprechende Zahlenwert als sog. *Funktionswert* zurückgegeben.
In jedem möglichen Weg durch die Funktion muss eine *return-Anweisung* stehen, weil es ansonsten kein Funktionsergebnis geben könnte.
Der Programmierer sollte dringend darauf achten, dass eine Funktion nicht den Zustand des Territoriums und des Hamsters ändert.

Aufgaben

1. Der Hamster hat genügend Körner im Maul und soll genau so viele Körner auf seinem Platz ablegen, wie es Schritte bis zur Mauer sind. Verwende dafür eine Funktion ***int AnzahlSchritteBisMauer()*** !

2. Der Hamster steht direkt vor einer regelmäßigen Treppe unbekannter Höhe. Auf der Kachel, auf der er sich befindet, liegen genügend Körner. Der Hamster soll zuerst die Anzahl der Stufen ermitteln und anschließend auf jeder Stufe ein Korn ablegen. Er soll aber nur die genau passende Anzahl Körner mitnehmen. Verwende eine Funktion namens ***int ermittleStufenanzahl()*** !

3. Der Hamster soll untersuchen, ob direkt vor ihm oder direkt hinter ihm mehr Körner liegen. Danach soll er auf diejenige Kachel gehen, welche mehr Körner enthält, und all diese Körner auffressen. Verwende einen Funktion namens ***int ermittleKoerneranzahl()*** !

4. Der Hamster steht irgendwo auf der Grundlinie mit Blick nach Osten in einem rechteckigen, ansonsten leeren Territorium. Der Hamster soll die Größe, d.h. Länge und Breite, des Raumes ermitteln und anschließend wieder seine anfängliche Startposition einnehmen.
   Er hat eine bestimmte Anzahl Körner im Maul. Falls diese ausreicht, soll er auf jeder Kachel am Rand des Territoriums ein Korn ablegen. Anschließend steht er wieder in seiner Startposition.

Methoden mit Parametern

```
void  geheVor(int n)  {
   while ((n > 0) && vornFrei())   {
      vor();
      n--;
   }
}
```

Der Parameter *n* ist für diese Prozedur praktisch eine lokale Variable. Ihren Initialisierungswert erhält sie beim Aufruf der Prozedur.

```
void  main()  {
   int zahl1 = 3, zahl2 = 4;
   geheVor(zahl1);
   geheVor(5);
   geheVor(zahl1 + zahl2);
   geheVor(7-zahl1);
}
```

Beim Aufruf obiger Methode wird ein Wert übergeben. Deshalb nennt man diese Übergabe bzw. diesen Aufruf auch ***call-by-value***. Wichtig ist folgendes: Beim ersten Methodenaufruf im obigen Programm erhält der Parameter *n* den Wert der Variablen *zahl1* aus dem Hauptprogramm. Innerhalb der Methode wird der Wert von *n* dauernd geändert. Dies hat jedoch keinen Einfluß auf die Variable *zahl1* des Hauptprogrammes!

Im nächsten Beispiel wird die Fakultät einer Zahl berechnet:

```
void  main()  {
   int zahl = fak(4);
   while (zahl>0) {
     gib();
     zahl--;
   }
}


int fak(int n)  {
  int ergebnis = 1;
  while (n>0)  {
    ergebnis = ergebnis * n;
    n--;
  }
  return ergebnis;
}
```

Es ist auch möglich, mehrere Parameter zu übergeben. Dabei ist natürlich die Reihenfolge der Übergabewerte wichtig.

```
void main() {
  goAndGive(4,2);
}


void goAndGive(int a, int b)  {
   while (a>0) {
     vor();
     a--;
   }
   while (b>0)  {
     gib();
     b--;
   }
}
```

Aufgaben

1. Der Hamster steht auf der Grundlinie unten links mit Blick nach Osten. Er soll ein Quadrat der Kantenlänge *n* ablaufen. Dies soll er mehrmals mit verschieden großen Quadraten machen.

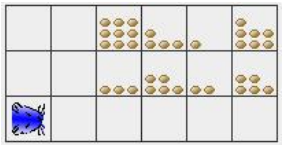
2. Der Hamster steht auf der Grundlinie unten links mit Blick nach Osten. Er soll auf den sieben vor ihm liegenden Feldern folgende Anzahlen von Körnern ablegen: 1, 2, 4, 8, 16, 32, 64. Schreibe dafür eine Hilfsfunktion, welche die Potenz 2^n berechnet, wobei n als Parameter übergeben werden soll!

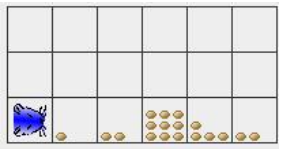
3. Der Hamster soll die Summe der ersten n natürlichen Zahlen berechnen und eine dementsprechende Körnerzahl auf seinem Platz ablegen. Benutze dafür eine Funktion *int summe(int n)* !

4. Der Hamster steht auf der Grundlinie unten links mit Blick nach Osten. Er soll in dem vor ihm liegenden rechteckigen Feld der Breite 3 und der Höhe 5 durch Körnerablegen eine Ziffer zeichnen (im Bild die Ziffer 3). Benutze dafür die Funktion *void zeichneZiffer(int n)* !

5. Der Hamster steht auf der Grundlinie unten links mit Blick nach Osten. Er soll direkt vor ihm ein Rechteck der Länge *l* und der Breite *b* aus Körnern hinlegen (also liegen anschließend *l·b* Körner da). Der Hamster hat genügend Körner im Maul. Löse diese Aufgabe mithilfe einer Funktion (mit Parametern)!

6. Im Territorium stehen oben rechts zwei vierstellige, durch Körner kodierte Dezimalzahlen (im Bild: 9 417 und 3 525). Der Hamster soll die eventuell fünfstellige Summe in die unterste Zeile schreiben. Benutze die Integer-Methoden *int summenziffer(int s1, int s2, int ue)* und *int uebertrag(int s1, int s2, int ue)* !





7. Der Hamster überprüft, ob die Anzahl der Körner, die er im Maul hat, eine Primzahl ist. Falls ja, legt er sie alle ab, falls nein, behält er sie im Maul. Benutze dazu eine Funktion *boolean istPrim(int n)* !

8. Der Hamster steht auf der Grundlinie unten links mit Blick nach Osten. Er soll auf jeder Kachel der Grundlinie eine der nächsten Primzahl entsprechende Körneranzahl ablegen. Es liegen also anschließend 2, 3, 5, 7, 11, 13, 17 usw. Körner auf den Kacheln.

Lösungen

Aufgabe 1

```
void main() {
  quadratLauf(4);
  quadratLauf(7);
}


void quadratLauf(int n) {
  for (int a=1; a<=n; a++)  {
     streckenLauf(n);
     linksUm();
  }
}


void streckenLauf(int n)  {
  while (n>0) {
    vor();
    n--;
  }
}
```

Aufgabe 6

```
int ue = 0;

void main() {
  geheZurWand();
  linksUm();
  rechne();
}


void geheZurWand()  {
  while (vornFrei()) vor();
}


void rechne()  {
  for (int i=1; i<=5; i++)  {
     bearbeiteStelle;
  }
}


void bearbeiteStelle()  {
  geheZurWand();
  linksUm(); linksUm();
  int s1 = anzahlKoernerDa();
  vor();
  int s2 = anzahlKoernerDa();
  vor();
  int summe = summenZiffer(s1, s2, ue);
  ue = uebertrag(s1, s2, ue);
  legAb(summe);
  linksUm(); linksUm(); linksUm();
  vor();
  linksUm(); linksUm(); linksUm();
}
```

```
int anzahlKoernerDa()  {
  int hilf = 0;
  while (kornDa())  {
    nimm();
    hilf++;
  }
  return hilf;
}


void legAb(int n)  {
  while (n>0)  {
    gib();
    n--;
  }
}


int summenZiffer(int s1, int s2, int ue)  {
   return (s1+s2+ue) % 10;
}


int uebertrag(int s1, int s2, int ue)  {
 return (s1+s2+ue) / 10;
}
```

Aufgabe 7

```
void main() {
  if (istPrim(13)) {
    while (!maulLeer()) gib();
  }
}
```

```
boolean istPrim(int n)  {
  boolean ergebnis = true;
  for (int i=2; i < n/2; i++)  {
     if (n%i == 0) {
        ergebnis = false;
     }
  }
  return ergebnis;
}
```

Rekursion

Funktionen, die sich selbst aufrufen, nennt man ***rekursive Funktionen***. Beispiel:

```
void main() {
  geheBisZurWand();
}

void geheBisZurWand()  {
  if (vornFrei())  {
    vor();
    geheBisZurWand();
  }
}
```

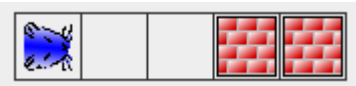
Sehr wichtig ist, dass der rekursive Aufruf nur unter einer Bedingung stattfindet, die irgendwann nicht mehr erfüllt ist. Ansonsten gäbe es eine sog. Endlosrekursion und der Rechner würde abstürzen! Der Grund dafür liegt darin, dass für jede neue Funktion Speicherplatz im Rechner belegt wird. Dieser Speicherplatz wird erst wieder freigegeben, wenn die Funktion beendet wird. Der zur Verfügung stehende Speicherplatz des Rechners ist aber nur endlich groß. Also darf die sog. ***Rekursionstiefe*** (das ist die Anzahl der geschachtelten Funktionen) nicht allzu groß sein!

Den großen Vorteil rekursiver Funktionen erkennt man am nachfolgenden Beispiel. Man beachte, dass der Hamster vor und nach jedem rekursiven Aufruf jeweils einen Schritt vorwärts geht. Das sorgt dafür, dass er nach Beendigung des Programms wieder an seinem Ausgangsort steht.

```
void main() {
  geheBisZurWandUndZurueck();
}

void geheBisZurWandUndZurueck()  {
  if (vornFrei())  {
    vor();
    geheBisZurWandUndZurueck();
    vor();
  }
  else {
    linksUm();
    linksUm();
  }
}
```

Angenommen, der Hamster steht auf der dritten Kachel vor einer Wand. Er kann also noch zwei Schritte bis zur Wand machen:



Die folgende Skizze erklärt für diesen Fall die Wirkungsweise der Rekursion. Jedes Rechteck steht für einen Aufruf der Funktion.

```
vor();
    vor();
        linksUm();
        linksUm();
    vor()
vor();
```

Oft werden auch rekursive Funktionen mit Parametern benötigt. Zum Beispiel kann man die Fakultät einer Zahl (n! = 1·2·3·4·…·n) auch folgendermaßen rekursiv definieren: 1! = 1 und für n>1 gilt n! = n· (n-1)!
Aus dieser Definition folgt z.B., dass 4! = 4·3! = 4·3·2! = 4·3·2·1! = 4·3·2·1 = 24
Diese rekursive Definition lässt sich auch für eine völlig analoge, rekursive Programmierung nutzen:

```
void main() {
  int anzahl = fakultaet(4);
  for (int i=1; i<= anzahl; i++)  gib();
}

int fakultaet(int n) {
  if (n == 1) return 1;
  else return n*fakultaet(n-1);
}
```

Aufgaben

Löse alle Aufgaben mit rekursiven Funktionen!

1. Der Hamster soll alle Körner auf seinem Platz aufnehmen.

2. Der Hamster soll alle Körner, die er im Maul hat, ablegen.

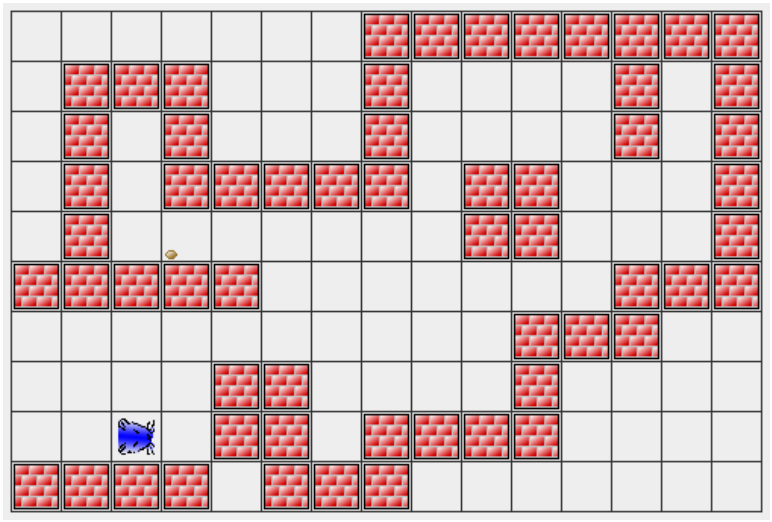
3. Der Hamster steht direkt vor einer regelmäßig geformten Treppe unbekannter Höhe. Er soll sie
   a) nur hoch laufen.
   b) hoch laufen und wieder zurück laufen.
   c) hoch laufen und auf der anderen Seite hinunterlaufen.

4. Schreibe eine Prozedur *void vorRek(int n)* , welche den Hamster (falls möglich) n Schritte vorgehen lässt.

5. Schreibe eine Funktion, welche nicht das Produkt der ersten *n* natürlichen Zahlen (also die Fakultät) sondern die Summe der ersten *n* Zahlen berechnet! Zur Kontrolle soll der Hamster eine entsprechende Körnerzahl ablegen.

6. In der höheren Mathematik ist die sog. *Fibonacci-Folge* sehr berühmt. Sie wird rekursiv definiert:
   fib(1) = 1,
   fib(2) = 2,
   Für alle n>2 gilt: fib(n) = fib(n-1) + fib(n-2)
   Die ersten 8 Fibonacci-Zahlen lauten also: 1, 2, 3, 5, 8, 13, 21, 34, ….
   Programmiere eine entsprechende Funktion und kontrolliere es wie in Aufgabe 5!
   Interessant an dieser Programmierung ist auch, dass die Funktion sich selber zwei mal (mit kleineren Parametern) aufruft.

7. (sehr schwierig !)
   Der Hamster sucht im Labyrinth einen Schatz (dargestellt durch ein Korn). Er soll diesen Schatz finden, aufnehmen und wieder zurück an seinen Ausgangsort gehen.



   Die Lösungsidee ist Folgende:
   Der Hamster macht (rekursiv) immer einen Schritt rechts an der Wand entlang. Dies macht er solange, bis er auf den Schatz stößt. Er nimmt den Schatz auf, dreht sich um und macht jetzt für jeden Schritt, den er vorher gemacht hat, einen Schritt links an der Wand entlang.