

Einführung in die Informatik - Teil XIX – Kellerautomaten

Inhalt:

1. Kellerautomaten
 - 1.1 Die Definition von Kellerautomaten
 - 1.2 Beispiel: geschachtelte Klammern
 - 1.3 Die Simulation des Kellerautomaten mit SIMA
 - 1.4 Zur Bedeutung von Kellerautomaten für Programmiersprachen

2. LOGO für Arme – Teil 2
 - 2.1 Änderungen an der Syntax
 - 2.2 Änderungen am Turtleparser
 - 2.3 Änderungen am Turtleinterpreter
 - 2.4 Aufgaben

Literaturhinweise:

- Vossen/Witt: Grundlagen der Theoretischen Informatik mit Anwendungen, Vieweg
- E. Modrow: Theoretische Informatik mit Delphi, www.emu-online.de
- Krüger, Guido: Handbuch der Java-Programmierung, www.javabuch.de oder Addison Wesley 2002

1. Kellerautomaten

1.1 Die Definition von Kellerautomaten

Können in einer Grammatik Nichtterminalsymbole durch Symbolfolgen ersetzt werden, die nicht den Bedingungen an reguläre Grammatiken entsprechen, und zwar ohne Rücksicht auf den Kontext, in dem das zu ersetzende Symbol auftaucht, dann spricht man von *kontextfreien Grammatiken*. Diese können von *Kellerautomaten* (push-down-Automaten, Stack-Maschinen, ...) analysiert werden¹. Standardbeispiel dafür sind geschachtelte Klammerstrukturen

- entweder so: $() \quad (()) \quad ()(()) \quad ()()()$
- oder so: $()(\quad (()) \quad ()() \quad (())((()()()()))()()$

Ein Analysator (Akzeptor, ...) für solche Strukturen muss sich „merken“ können, wie viele öffnende Klammern bisher aufgetreten und nicht durch schließende kompensiert worden sind.

Dafür muss er „zählen“ können!

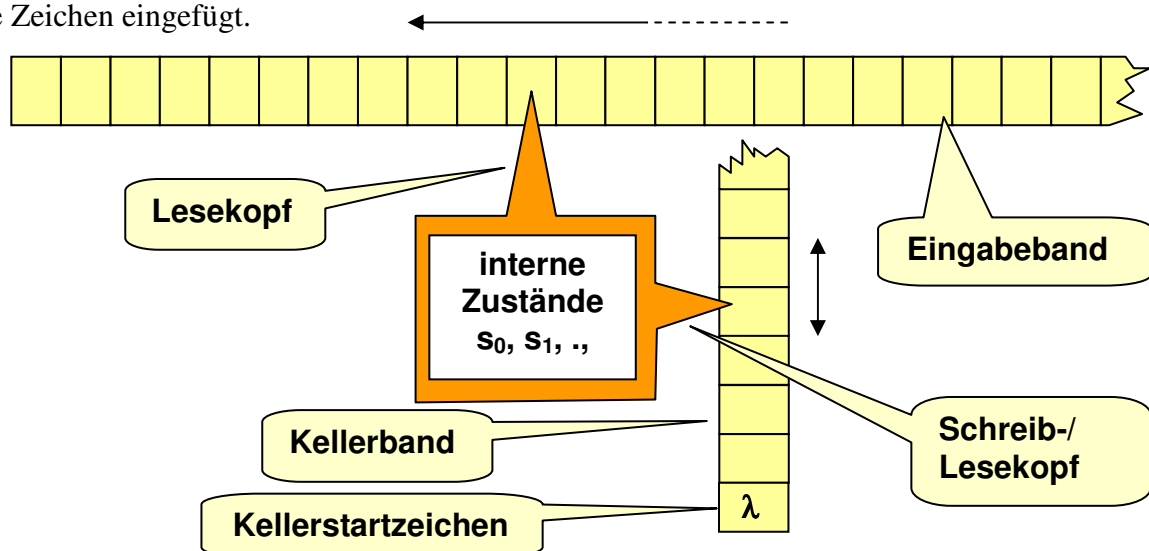
Endliche Automaten können sich nur etwas merken, indem sie den Zustand wechseln. Hat ein solcher Automat n Zustände, dann geht er spätestens beim $(n+1)$ -ten Zeichen in einen Zustand über, in dem er schon vorher gewesen ist. Es ist danach nicht mehr feststellbar, auf welchem Weg er diesen Zustand erreichte: er hat etwas „vergessen“. Ein endlicher Automat mit n Zuständen kann also nur bis n zählen.

Damit keine Missverständnisse auftauchen: Man kann zwar zu jeder gegebenen Klammerstruktur einen endlichen Akzeptor angeben. Dieser kann aber nicht *jede* Klammerstruktur analysieren, weil durch Erhöhung der Zahl der geöffneten Klammern schnell eine Klammerstruktur gefunden werden kann, die für die Analyse mehr Zustände erfordert, als der Automat hat.

Der endliche Automat muss deshalb um ein „Notizbuch“ erweitert werden, das beliebig viele Symbole speichern kann. Wird dieses „Gedächtnis“ in Form eines Stacks (*Kellers*) organisiert, dann erhält man einen Kellerautomaten.

Kellerautomat = erkennender Automat + Keller

Ein Stack arbeitet nach dem *Last-In-First-Out-Prinzip* (LIFO): Es kann immer nur das oberste Kellerzeichen gelesen werden (das dabei entfernt wird) bzw. oben auf dem Keller werden neue Zeichen eingefügt.



¹ Natürlich können Kellerautomaten auch reguläre Sprachen analysieren. Man „schießt dann nur mit Kanonen auf Spatzen“.

Ein Kellerautomat KA wird demnach beschrieben wie ein erkennender Automat (durch Eingabealphabet E , Zustandsmenge S , Überföhrungsfunktion u , Startzustand s_0 , Menge der Endzustände Σ) und ein Kelleralphabet K .

$$KA = (E, S, K, s_0, u, \Sigma)$$

Die Übergänge der Überföhrungsfunktion bestehen aus Kombinationen von Eingabezeichen e_i , Zustand s_j und oberstem Kellerzeichen k_l . Sie liefern einen Folgezustand s_m und eine Folge aus Kellerzeichen kzf , die auf das Kellerband zuröck geschrieben wird.

$$(e_i, s_j, k_l) \rightarrow (s_m, kzf)$$

Der Grund daföur, dass eine Zeichenfolge (ggf. leer) auf das Kellerband geschrieben wird, liegt darin, dass nach unserer Konvention beim Lesen des obersten Kellerzeichens dieses vom Kellerband gelöscht wird. Wird es doch weiter benötigt, dann muss es zusammen mit dem oder den zusätzlich benötigten Zeichen zuröck geschrieben werden. Das Verfahren hat wie jede Konvention Vor- und Nachteile: Wird das oberste Kellerzeichen nicht mehr benötigt, dann ist keine weitere Aktion erforderlich, benötigt man es doch, dann muss man halt schreiben.)

Der momentane Zustand des Automaten wird also beschrieben durch eine Kombination aus (Eingabezeichen, Zustand, Kellerzeichen) - und davon gibt es sehr viele! Man beschränkt sich bei der Überföhrungsfunktion deshalb auf die Angabe der „wichtigen“ Übergänge. Ist für eine Kombination kein Übergang definiert, dann bleibt der Kellerautomat stehen.

$$(Eingabezeichen, Zustand, Kellerzeichen) \rightarrow (Zustand, Kellerzeichenfolge)$$

Ein Kellerautomat akzeptiert eine Zeichenfolge, wenn

- er sich nach ihrer Verarbeitung im Endzustand befindet
- und das Kellerband dann leer ist.

Die Arbeitsweise des Kellerautomaten KA können wir im Struktogramm beschreiben:

KA in den Anfangszustand bringen ($s = s_0$), den Lesekopf über das erste Zeichen des Eingabebandes bringen, das Kellerband bis auf das Kellerstartzeichen leeren und den Schreib-/Lesekopf (SL-Kopf) über dieses Zeichen bringen.	
Eingabezeichen e (eventuell leer) und das oberste Kellerzeichen $k (= \lambda)$ lesen (damit ist das Kellerband leer)	
SOLANGE kein STOP erzeugt wurde TUE	
die Kombination (e,s,k) ist definiert	
wahr	falsch
Die (ev. leere) Kellerzeichenfolge auf das Kellerband schreiben. Der SL-Kopf steht dann über dem obersten Zeichen k der Folge.	STOP
Folgezustand s einnehmen	
Das nächste Zeichen e des Eingabebandes lesen (ev. leer).	

1.2 Beispiel: geschachtelte Klammern

Als Kellerstartzeichen wählen wir % (wie im Simulator), als Klammern [und]. Die erforderlichen Übergänge finden wir, wenn wir anhand eines Beispiels überlegen, welche Kombinationen auftreten können. Auf diese Weise finden wir die folgenden Übergänge (’ bedeutet: leeres Zeichen):

$([, s_0, \%) \rightarrow (s_0, \% [)$	erste Klammer abspeichern, Zustand bleibt
$([, s_0, [) \rightarrow (s_0, [[)$	weitere Klammern abspeichern, Zustand bleibt
$([, s_0,]) \rightarrow (s_0, ’)$	Klammer löschen, Zustand bleibt
$(’ , s_0, \%) \rightarrow (s_e, ’)$	Startzeichen löschen, Endzustand einnehmen

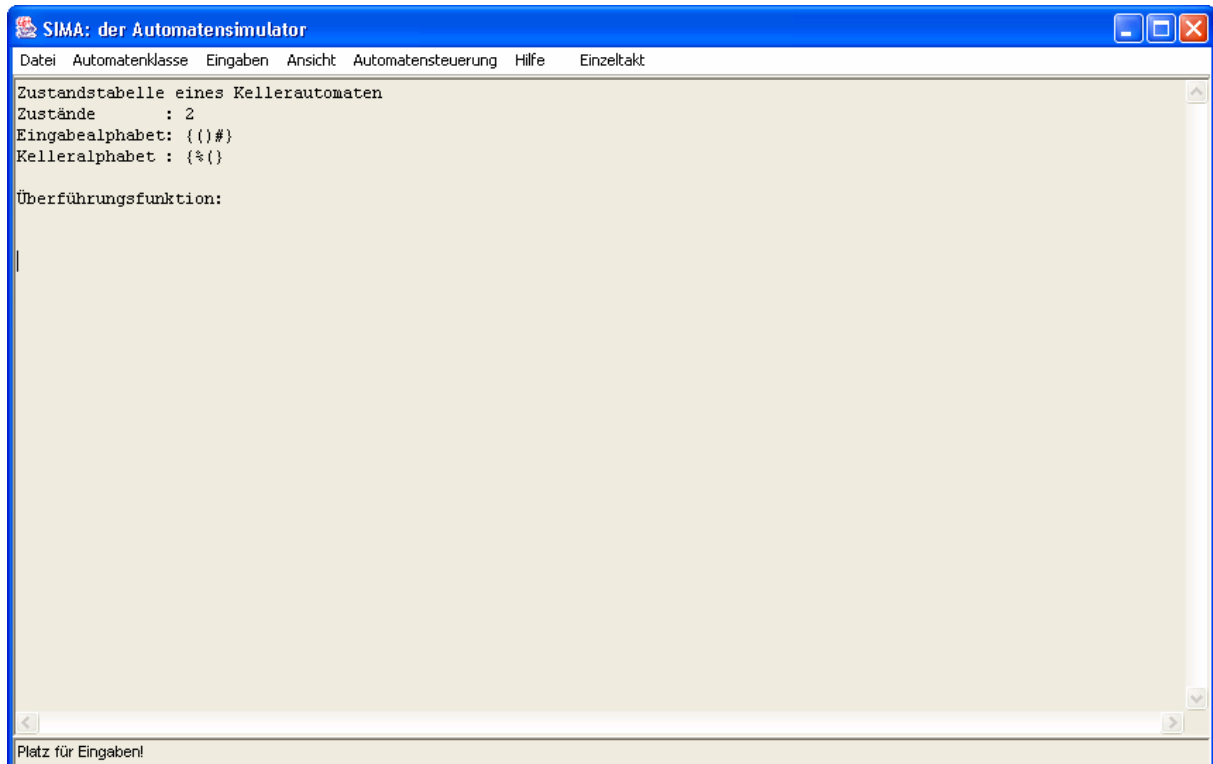
Die weiteren Größen des Kellerautomaten sind:

$$E = \{ [,], ’ \} \quad S = \{ s_0, s_e \} \quad K = \{ \%, [, ’ \} \quad \Sigma = \{ s_e \}$$

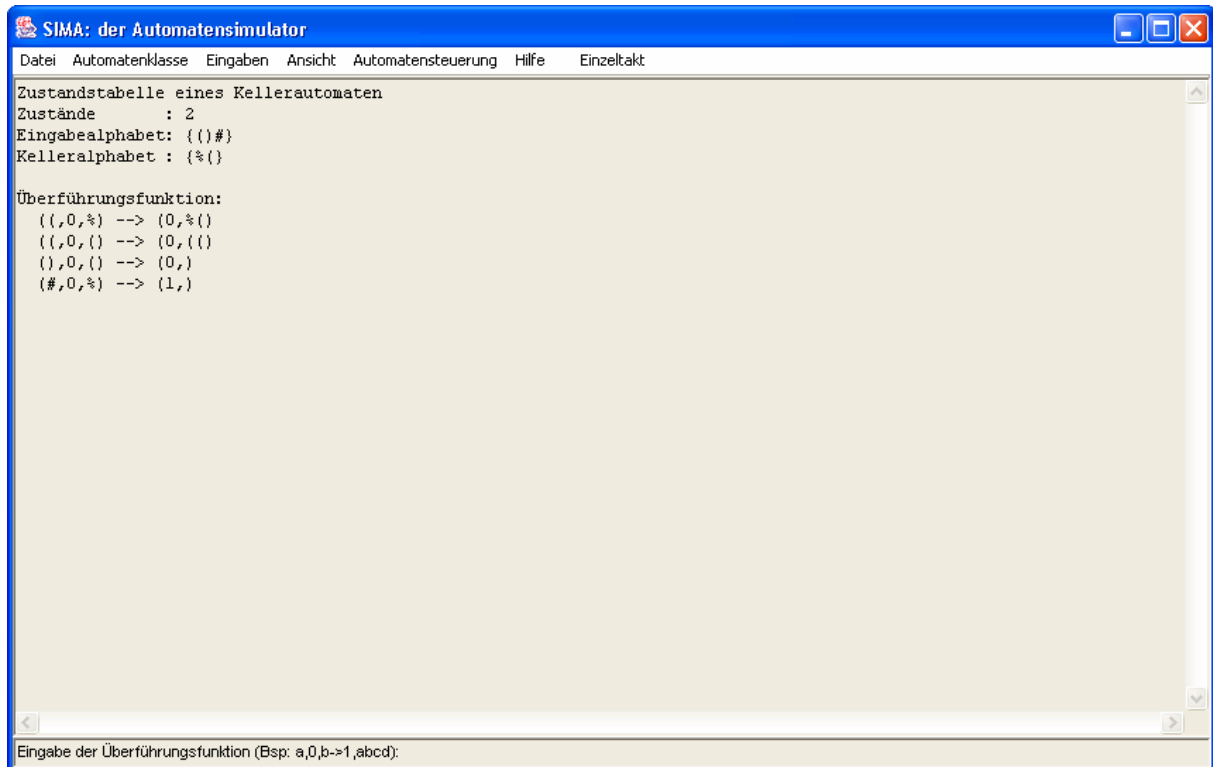
Anfangszustand: s_0

1.3 Die Simulation des Kellerautomaten mit SIMA

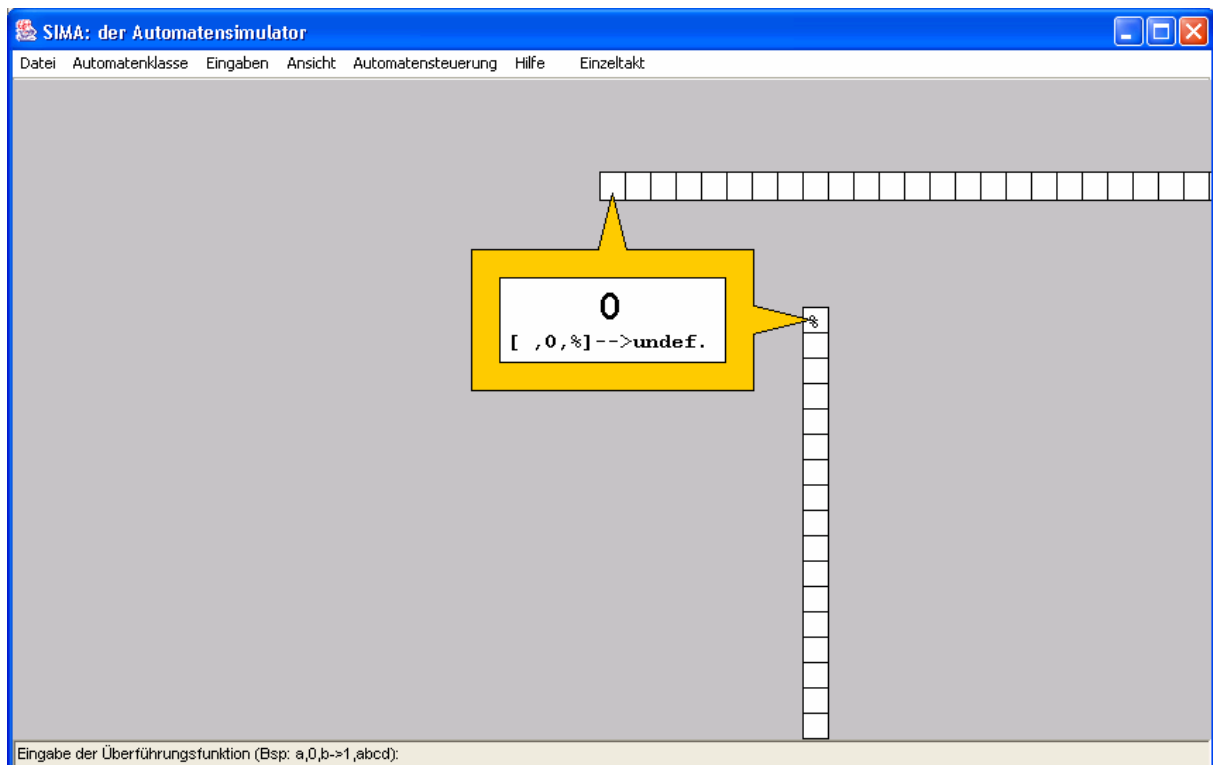
Wir stellen im Automatenimulator SIMA den Automatentyp „Kellerautomat“ ein und wählen 2 Zustände. Als Eingabealphabet nehmen wir die beiden Klammern und ein beliebiges Endezeichen, in diesem Fall „#“ (ein „leeres“ ASCII-Zeichen gibt es ja nicht.) Als Ausgabealphabet wählen wir die öffnende Klammer und das Kellerstartzeichen. Damit erhalten wir die folgende Ausgabe:



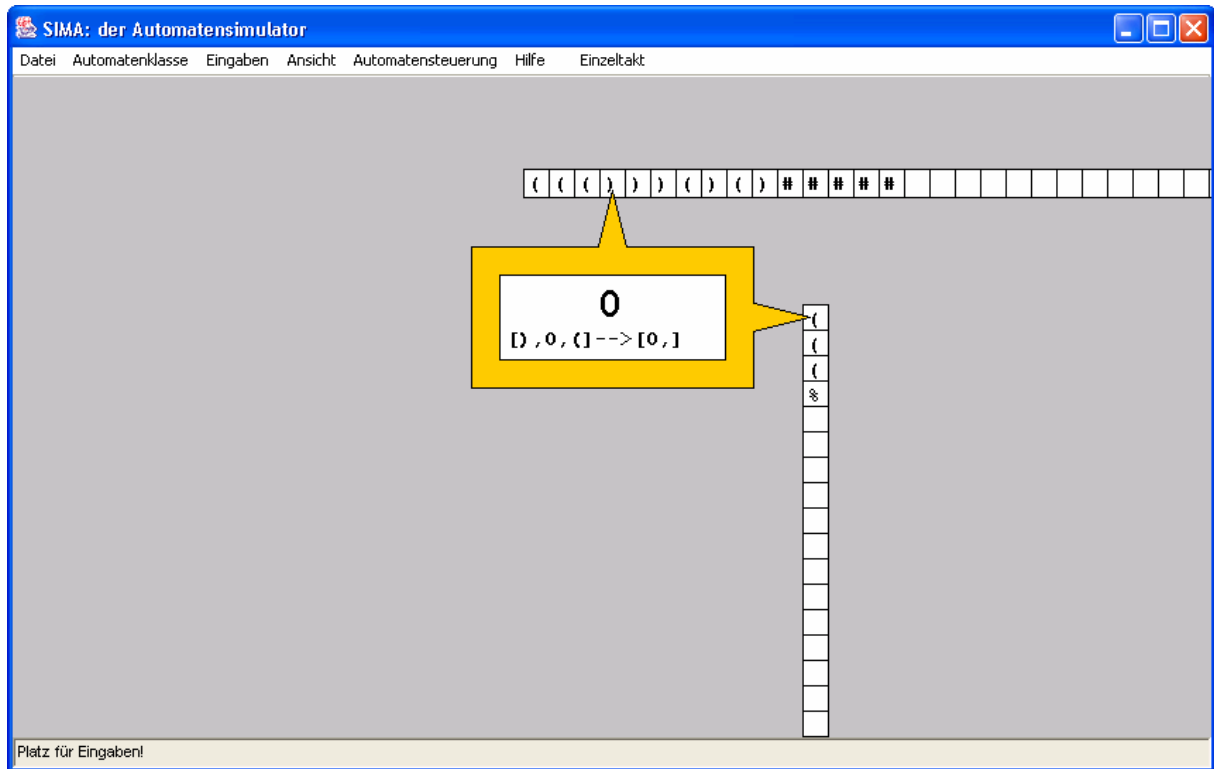
Jetzt kommt die Überföhrungsfunktion. Die wird auch in der Eingabezeile eingegeben, und zwar nur für die „interessanten“ Übergänge. (Fehlt ein Übergang, dann bleibt die Maschine ja einfach stehen.)



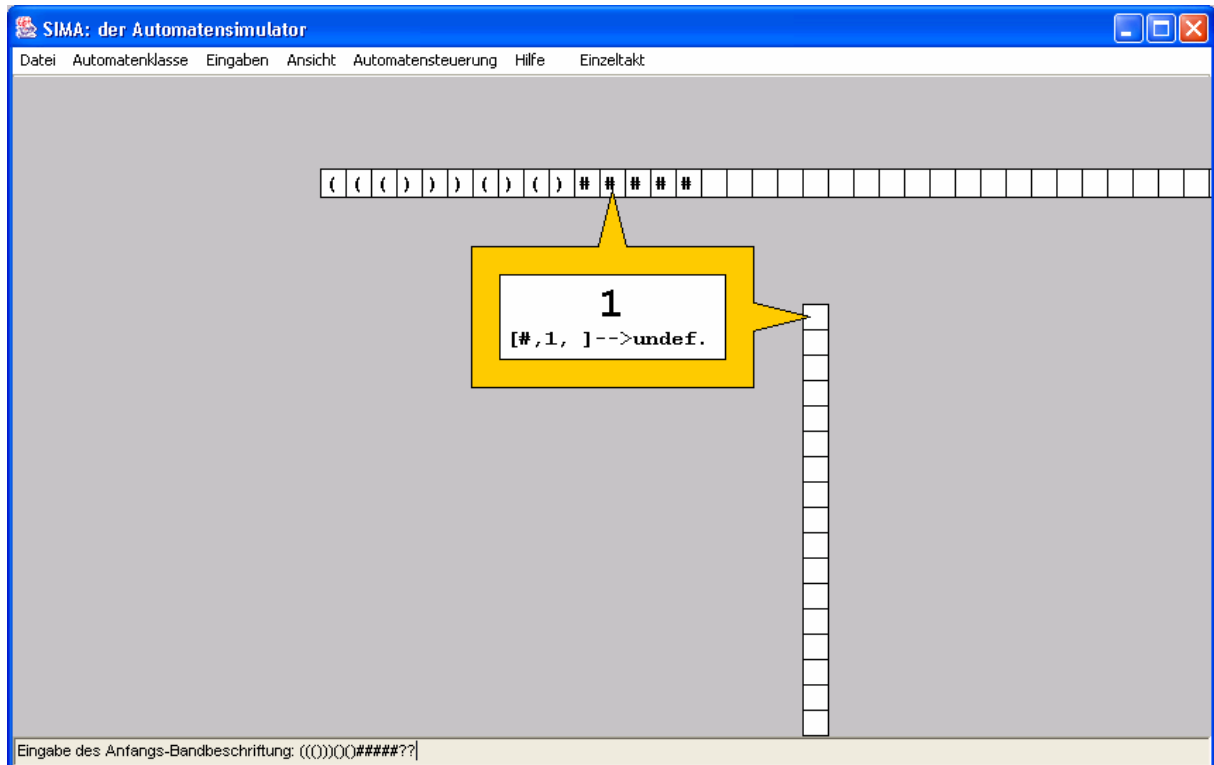
Danach kann der Automat arbeiten, Wir wählen im Menü „Ansicht“ den Punkt „Simulator“ und sehen die Maschine in ihrer ganzen Pracht:



Wir geben eine Anfangsbeschriftung ein (und vergessen dabei das Endezeichen „#“ nicht!) und lassen die Maschine ihre Werk verrichten – entweder im Einzeltaktmodus oder fortlaufend im Dauertakt ...



... bis sie fertig ist!



1.4 Zur Bedeutung von Kellerautomaten für Programmiersprachen

Geschachtelte Strukturen gehören zu den Grundkonstrukten von Programmiersprachen. Seien es geschachtelte Schleifen, geschachtelte Alternativen, geschachtelte Blöcke, ...

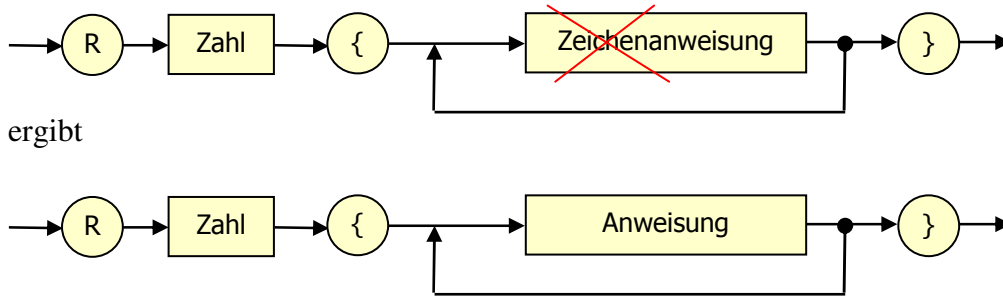
```
do                if (...)          {
{                {                    ...
...              ...                  {
do              if (...)            ...
{                {                    }
...              ...                  ...
}                }                    }
while(...)      else
...              {
}                ...
while (...)     }
                ...
                }
                else ...
```

Auch die Aufrufstrukturen von Methodenaufrufen, die Benutzung rekursiver Datenstrukturen usw. kann nur erfolgen, wenn ein **Zählmechanismus** vorhanden ist, der die korrekte Analyse bzw. Abarbeitung in einer korrekten Reihenfolge sicherstellt. In allen modernen Programmiersprachen werden deshalb Stapel (stacks) verwendet, die geschachtelte Strukturen verwalten. Der für die Funktion blockorientierter Sprachen wesentliche Stack ist aus dem zweiten Semester (Referenztypen, ...) ja schon gut bekannt. Da z. B. Java selbst diesen Stack verwaltet, ist die Möglichkeit, rekursive Prozesse zu verwirklichen, dieser Sprache immanent. Moderne Laufzeitsysteme enthalten also Keller in vielfältiger Form. **Sie sind Kellerautomaten.** Im Interpreter von „Logo für Arme“ wird dieses durch die rekursiven Aufrufe ausgenutzt. Obwohl der Parser dieser Sprache geschachtelte Strukturen verhindert, könnten diese vom Interpreter problemlos ausgeführt werden.

2. LOGO für Arme – Version 2

2.1 Änderungen an der Syntax

Wir wollen nur eine kleine, aber folgenreiche Veränderung vornehmen: Innerhalb der Schleife sollen nicht nur Zeichenanweisungen, sondern Anweisungen ausgeführt werden können – ggf. also weitere Schleifen.

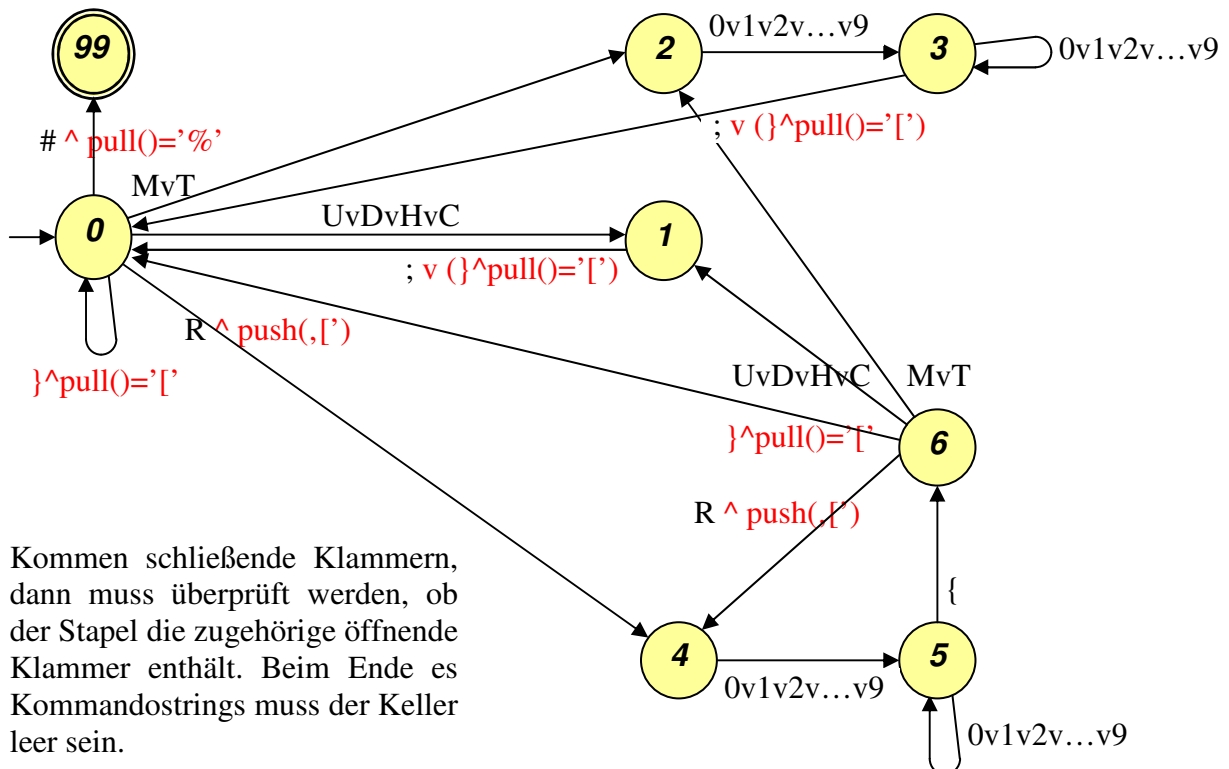


Damit lassen wir geschachtelte Strukturen zu, die von einem Kellerautomaten kontrolliert werden müssen.

2.2 Änderungen am Turtleparser

Am Eingabealphabet ändert sich gar nichts: $E = \{U, D, H, C, M, T, R, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, \{, \}, ;, \#, \}$

Interessant wird es erst, wenn innerhalb einer Schleife ein neuer Schleifenbefehl auftaucht. Es wird aber auch einfacher: Weil jetzt ein Kellerautomat die Schachtelung kontrolliert, brauchen wir innerhalb des Parsers nicht mehr zu unterscheiden, ob wir uns innerhalb oder außerhalb einer Schleife befinden. Einige doppelt auftretende Strukturen können damit entfallen. Die Bezeichnung „ $\} \wedge pull() = '['$ “ ist so zu verstehen: Der Übergang erfolgt, wenn bei Eingabe von „ $\}$ “ oben auf dem Stack eine „ $[$ “ liegt. Da unsere Schleifen mit einem „ R “ beginnen, kellern wir bei diesem Zeichen eine öffnende Klammer ab.



Kommen schließende Klammern, dann muss überprüft werden, ob der Stapel die zugehörige öffnende Klammer enthält. Beim Ende es Kommandostrings muss der Keller leer sein.

Der Turtleparser benötigt einen Keller, der von einer eigenen Klasse *Stack* verwaltet werden soll. Diese stellt die Methoden *push()* und *pull()* zur Verfügung. Als Keller benutzt sie einen *String*.

```
public class Stack
{
    String stack = "%";

    public void push(String s) {
        stack = s + stack; //Zeichenkette vorne anhängen
    }

    public char pull() {
        if(stack.length()>0)
        {
            char h = stack.charAt(0); //erstes Zeichen entfernen
            stack = stack.substring(1);
            return h;
        }
        else return '?'; //bei Katastrophen
    }
}
```

Der Parser selbst muss so geändert werden, dass er an den angegebenen Stellen den Kellerautomaten aufruft.

```
class Turtleparser
{
    int s;
    Stack stapel = new Stack();

    public void Turtleparser()
    {
        s = 0;
    }

    private int u(int s, char c)
    {
        switch(s)
        {
            case 0:
            {
                if(c=='#')
                {
                    if(stapel.pull()=='%') return 99;
                    else return 26;
                }
                else if((c=='U') || (c=='D') || (c=='H') || (c=='C')) return 1;
                else if((c=='M') || (c=='T')) return 2;
                else if(c=='R')
                {
                    stapel.push("[");
                    return 4;
                }
                else if(c=='}')
                {
                    if(stapel.pull()=='[') return 0;
                    else return 26;
                }
                else return 20;
            }
        }
    }
}
```

```
case 1:{
  if(c==';') return 0;
  else
  {
    if(c=='}')
    {
      if(stapel.pull()=='[') return 0;
      else return 26;
    }
    else return 21;
  }
}
case 2:{
  if((c=='0')||(c=='1')||(c=='2')||(c=='3')||(c=='4')||(c=='5')||
    (c=='6')||(c=='7')||(c=='8')||(c=='9')) return 3;
  else return 22;
}
case 3:{
  if((c=='0')||(c=='1')||(c=='2')||(c=='3')||(c=='4')||(c=='5')||
    (c=='6')||(c=='7')||(c=='8')||(c=='9')) return 3;
  else if(c==';') return 0;
  else
  {
    if(c=='}')
    {
      if(stapel.pull()=='[') return 0;
      else return 26;
    }
    else return 23;
  }
}
case 4:{
  if((c=='0')||(c=='1')||(c=='2')||(c=='3')||(c=='4')||(c=='5')||
    (c=='6')||(c=='7')||(c=='8')||(c=='9')) return 5;
  else return 22;
}
case 5:{
  if((c=='0')||(c=='1')||(c=='2')||(c=='3')||(c=='4')||(c=='5')||
    (c=='6')||(c=='7')||(c=='8')||(c=='9')) return 5;
  else if(c=='{') return 6;
  else return 24;
}
case 6:{
  if((c=='U')||(c=='D')||(c=='H')||(c=='C')) return 1;
  else if((c=='M')||(c=='T')) return 2;
  else if(c=='R')
  {
    stapel.push("[");
    return 4;
  }
  else if(c=='}')
  {
    if(stapel.pull()=='[') return 0;
    else return 26;
  }
  else return 25;
}
default: return s;
}
}
```

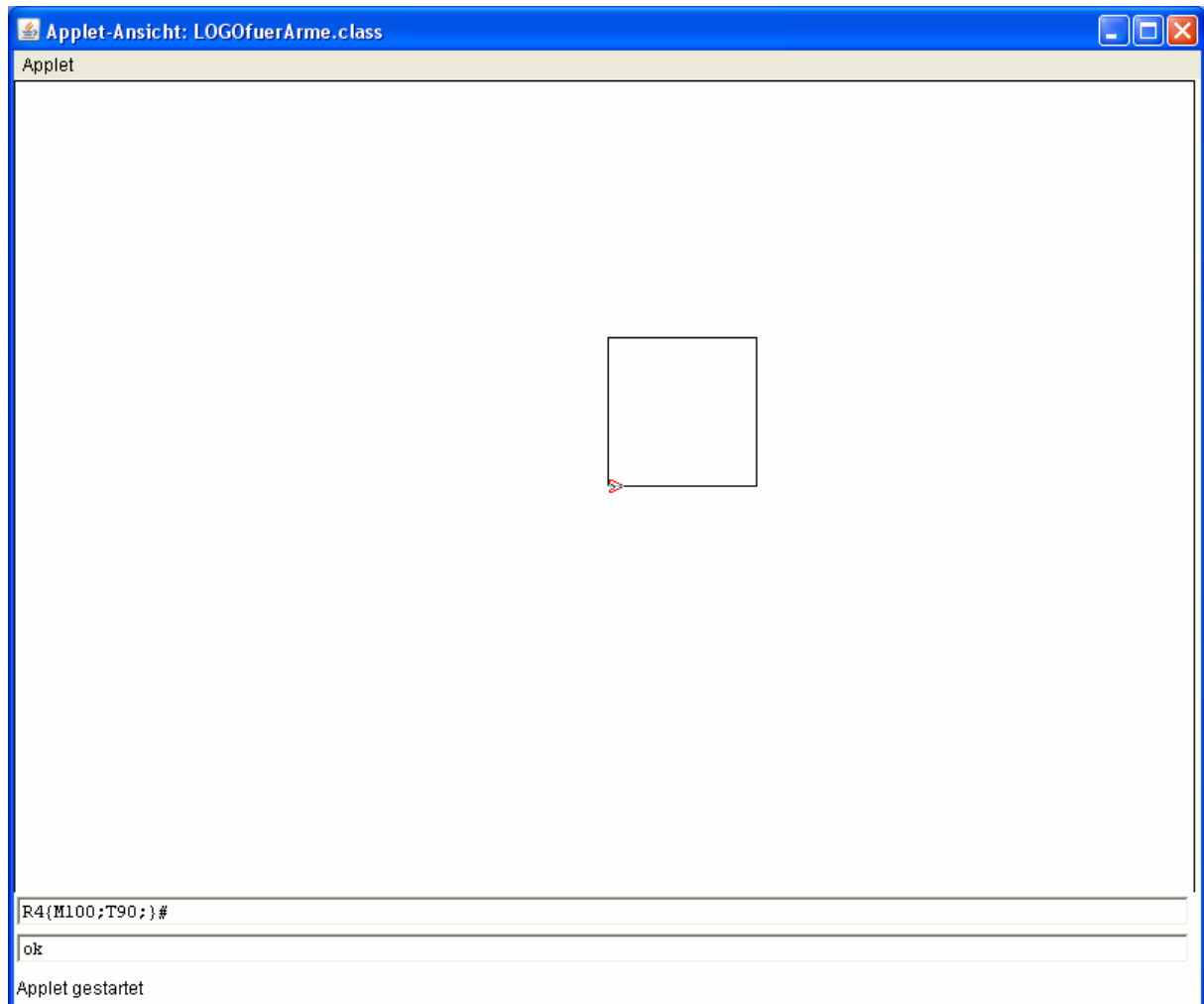
2.3 Änderungen am Turtleinterpreter

Im Interpreter muss nur innerhalb der Schleife der zu dieser Schleife gehörende Befehlsstring zusammengesucht werden. Da dieser weitere Schleifen enthalten kann, dürfen wir nicht einfach die nächste schließende Klammer suchen, sondern wir müssen die richtige finden. Das geschieht, indem wir die Klammern zählen.

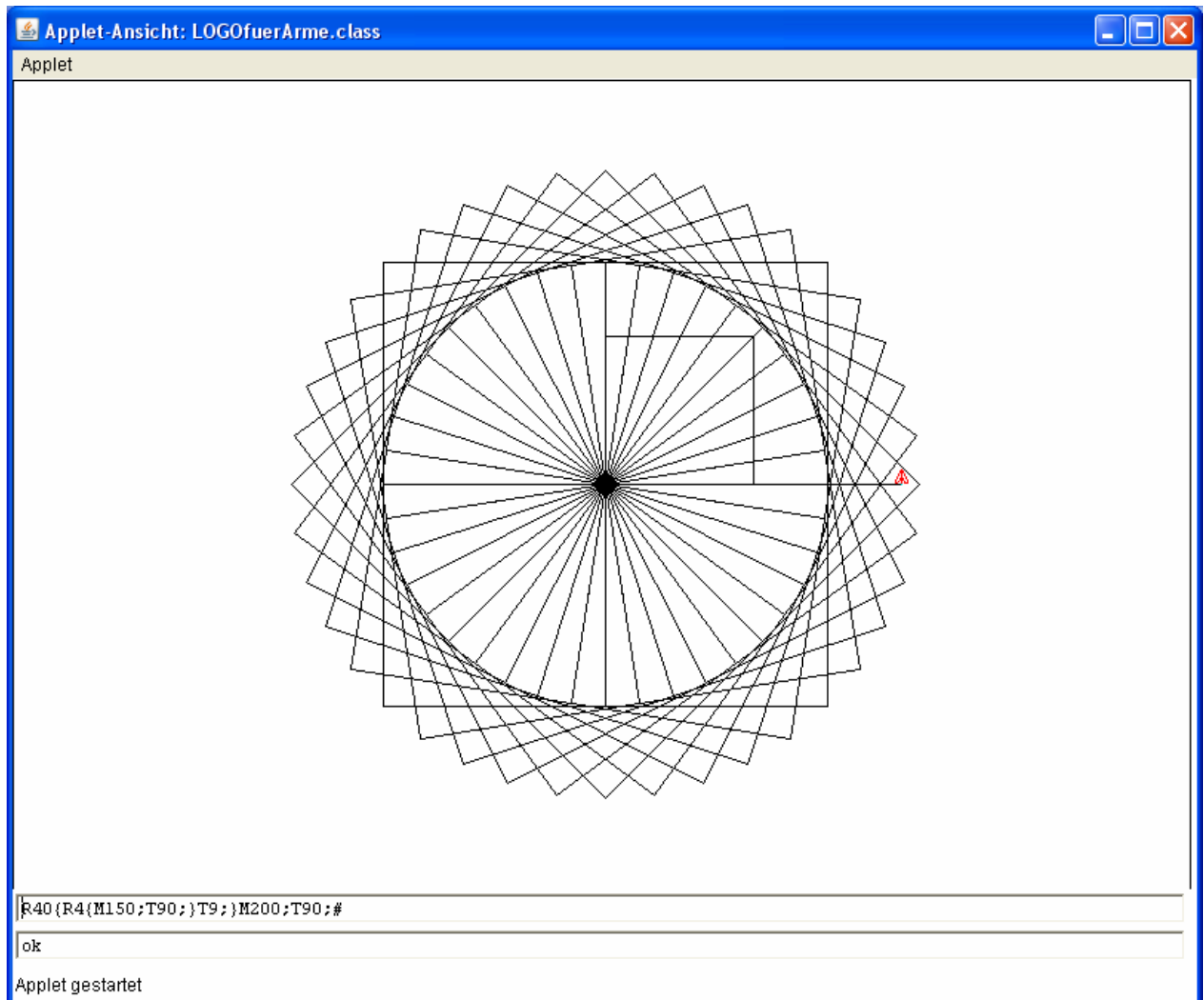
```
...
else if(c == 'R')
{
    z = 0; //Anzahl der Wiederholungen feststellen
    do
    {
        c = s.charAt(0);
        s = s.substring(1);
        z = 10*z + (int)c - (int)'0';
    }
    while((c >= '0') && (s.charAt(0) <= '9'));
    s = s.substring(1);
    String h = "";
    int offeneKlammern = 1; //offene Klammern zählen
    boolean ok = false;
    while(!ok)
    {
        if(s.charAt(0) == '{') //das war noch eine öffnende Klammer
        {
            offeneKlammern++;
            h = h + s.charAt(0);
        }
        else if(s.charAt(0) == '}')
        {
            if(offeneKlammern > 1) //rückwärts zählen ...
            {
                offeneKlammern--;
                h = h + s.charAt(0);
            }
            else ok = true; //... oder fertig
        }
        else h = h + s.charAt(0);
        s = s.substring(1);
    }
    for(int i=0; i<z; i++)
        fuehreAus(h, t); //Klammerkörper interpretieren
}
...
```

Am restlichen Programm sind keinerlei Änderungen erforderlich.

Jetzt können wir also nicht nur Rechtecke zeichnen, ...



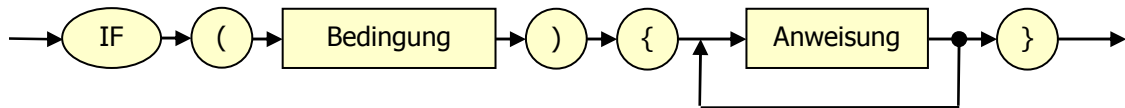
... sondern auch die üblichen Figuren aus verdrehten Rechtecken erzeugen.



2.4 Aufgaben

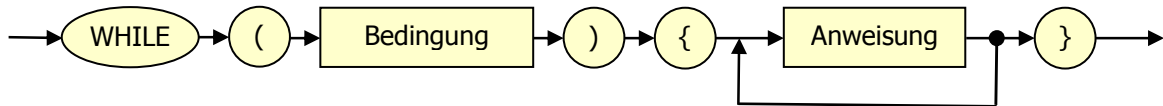
1. Führen Sie eine Alternative ein, die weitere Alternativen enthalten kann:

In Abhängigkeit von der Farbe des Pixels am Ort der Turtle oder der Turtleposition sollen unterschiedliche Zeichenbefehlsfolgen ausgeführt werden können. Reduzieren Sie die Syntax geeignet und implementieren Sie den Befehl. Natürlich muss auch die Bedingung noch spezifiziert werden.



4. Auf entsprechende Art sollen zwei Schleifenarten eingeführt werden: Die Turtle soll Zeichenbefehle ausführen, *solange* (WHILE) bzw. *bis* (DO) die Turtle sich über Pixeln einer anzugebenden Farbe befindet der bestimmte Positionen erreicht hat.

- a: Führen Sie die While-Kontrollanweisung ein:



- b: Führen Sie die DO-Kontrollanweisung ein:

