Programmieren mit Python

Repetition

Interpreter und Compiler

Python ist eine interpretierte Programmiersprache. Das heisst, dass ein Python-Programm (Quellcode, Python-Skript) eingelesen und auf syntaktische Korrektheit überprüft wird. Ist dies der Fall, wird es in Maschinencode übersetzt und sofort ausgeführt.

Einen anderen Ansatz verfolgen compilierte Programmiersprachen wie Java oder C++. Auch dort wird zuerst überprüft, ob der Quellcode korrekt geschrieben ist. Falls ja, wird das Programm von einem sogenannten Compiler in Maschinenencode übersetzt und für das jeweilige Betriebssystem so aufbereitet, dass es als eigenständiges Programm aufgerufen werden kann.

Python-Code schreiben

Um Python-Programme zu schreiben, braucht es einen Editor, d. h. ein Programm, mit dem man Texte erfassen, verändern und speichern kann. Herkömmliche Textverarbeitungsprogramme sind dafür aber nicht geeignet, da sie neben dem Text auch Formatierungsinformationen abspeichern, die der Python-Interpreter nicht versteht.

Spezielle Editoren unterstützen beim Programmieren durch Syntax-Hervorhebung (Syntax-Highlighting). Dabei werden Schlüsselwörter, Variablen, Text, Zahlen und Kommentare durch Farben gekennzeichnet und erhöhen so die Lesbarkeit der Programme.

IDLE

IDLE ist eine Integrierte Entwicklungsumgebung (Integrated Development Environment, kurz IDE) und wird mit der offiziellen Version von Python mitgeliefert. Sie besteht aus dem Interpreter, einem Editor, der Python-Shell und einem Debugger (Fehlersuchprogramm), die in einer einfachen grafischen Benutzerschnittstelle zusammengefasst sind.

Der Workflow in IDLE

1. File/New File öffnet im Editor eine neue Datei. Diese sollte man sofort unter einem Namen speichern. Die Endung `.py` wird automatisch an den Dateinamen angehängt.
2. Schreibe das Programm. Gelegentlich sollte man es speichern, um bei einem Absturz Datenverlust zu vermeiden.
3. Das Programm (oder ein lauffähiger Teil) kann mit dem Befehl Run/Run-Module oder mit der F5-Taste ausgeführt werden.
4. Bei einer Fehlermeldung zeigt die letzte Fehlermeldung in der Shell die Nummer der fehlerverursachenden Zeile an. Gelegentlich befindet sich der Fehler auch in der Zeile davor. Danach ist das Programm erneut auszuführen. Wiederhole die Schritte 3 und 4 so lange, bis das Programm einwandfrei funktioniert bzw. alle Tests besteht.

Die Python-Shell

Die Python-Shell hat mehrere Funktionen:

- Zum Übungs- und Testzwecken können Programmfragmente von wenigen Zeilen in der Python-Shell eingegeben und mit der ENTER-Taste ausgeführt werden.
- Die Ausgaben der `print(...)`-Funktion werden in der Shell ausgegeben.
- Die Benutzereingaben der `input(...)`-Funktion erfolgen in der Shell.
- Fehlermeldungen werden auf der Shell ausgegeben.

Das Format von Python-Programmen

Python-Programme werden in einer Textdatei mit der Endung `.py` gespeichert.

Kommentare werden mit einem Doppelkreuz „#“ (Raute, Hashtag) eingeleitet. Text zwischen dem Kommentarzeichen und dem Zeilenende wird vom Python-Interpreter ignoriert. Mit „Triple quotes“; d. h. drei doppelten (#) oder drei einfachen (') Anführungszeichen kann ein mehrzeiliger Kommentar erstellt werden.

Als Zeichencodierung wird standardmässig UTF-8 verwendet.

Strukturierung

Für Verzweigungen, Schleifen sowie die Definition von Funktionen und Klassen müssen mehrere Anweisungen zu einem Anweisungsblock zusammengefasst werden. Viele Programmiersprachen kennzeichnen solche Blöcke durch geschweifte Klammern.

Python rückt stattdessen den zusammengehörenden Code mit einer festen Anzahl Leerzeichen ein (üblich sind vier Leerzeichen). Diese Einrückung erfolgt automatisch beim Drücken der Tabulator-Taste.

Notation

- Text in `Nichtproportionalschrift` stellt Computercode dar, der genau so eingegeben werden sollte.

```
x = 3
y = 5
print(x + y)
print("hello, world")
```

- Text in *`kursiver Nichtproportionalschrift`* ist durch benutzerdefinierte Werte zu ersetzen.

  `print(`*`Zeichenkette`*`)`

Darstellung ganzer Zahlen

Die ganzen Zahlen 741 und -741 können wie folgt dargestellt werden:

- dezimal: `741`, `-741`
- binär: `0b1011100101`, `-0b1011100101`
- oktal: `0o1345`, `-0o1345`
- hexadezimal: `0x2e5`, `-0x2e5`

Operatoren für ganze Zahlen

Operator	Beschreibung	Beispiel	Wert
-	monadisches Minus	-17	
+	Addition	23+17	
-	Subtraktion	23-17	
*	Multiplikation	4*17	
/	Division	10/4	
//	Ganzzahldivision	10//4	
%	Divisionsrest	17%6	
**	Potenzieren	-2**4	

Operatoren für ganze Zahlen

Operator	Beschreibung	Beispiel	Wert
-	monadisches Minus	-17	-17
+	Addition	23+17	
-	Subtraktion	23-17	
*	Multiplikation	4*17	
/	Division	10/4	
//	Ganzzahldivision	10//4	
%	Divisionsrest	17%6	
**	Potenzieren	-2**4	

Operatoren für ganze Zahlen

Operator	Beschreibung	Beispiel	Wert
-	monadisches Minus	-17	-17
+	Addition	23+17	40
-	Subtraktion	23-17	
*	Multiplikation	4*17	
/	Division	10/4	
//	Ganzzahldivision	10//4	
%	Divisionsrest	17%6	
**	Potenzieren	-2**4	

Operatoren für ganze Zahlen

Operator	Beschreibung	Beispiel	Wert
-	monadisches Minus	-17	-17
+	Addition	23+17	40
-	Subtraktion	23-17	6
*	Multiplikation	4*17	
/	Division	10/4	
//	Ganzzahldivision	10//4	
%	Divisionsrest	17%6	
**	Potenzieren	-2**4	

Operatoren für ganze Zahlen

Operator	Beschreibung	Beispiel	Wert
-	monadisches Minus	-17	-17
+	Addition	23+17	40
-	Subtraktion	23-17	6
*	Multiplikation	4*17	68
/	Division	10/4	
//	Ganzzahldivision	10//4	
%	Divisionsrest	17%6	
**	Potenzieren	-2**4	

Operatoren für ganze Zahlen

Operator	Beschreibung	Beispiel	Wert
-	monadisches Minus	-17	-17
+	Addition	23+17	40
-	Subtraktion	23-17	6
*	Multiplikation	4*17	68
/	Division	10/4	2.5
//	Ganzzahldivision	10//4	
%	Divisionsrest	17%6	
**	Potenzieren	-2**4	

Operatoren für ganze Zahlen

Operator	Beschreibung	Beispiel	Wert
-	monadisches Minus	-17	-17
+	Addition	23+17	40
-	Subtraktion	23-17	6
*	Multiplikation	4*17	68
/	Division	10/4	2.5
//	Ganzzahldivision	10//4	2
%	Divisionsrest	17%6	
**	Potenzieren	-2**4	

Operatoren für ganze Zahlen

Operator	Beschreibung	Beispiel	Wert
-	monadisches Minus	-17	-17
+	Addition	23+17	40
-	Subtraktion	23-17	6
*	Multiplikation	4*17	68
/	Division	10/4	2.5
//	Ganzzahldivision	10//4	2
%	Divisionsrest	17%6	5
**	Potenzieren	-2**4	

Operatoren für ganze Zahlen

Operator	Beschreibung	Beispiel	Wert
-	monadisches Minus	-17	-17
+	Addition	23+17	40
-	Subtraktion	23-17	6
*	Multiplikation	4*17	68
/	Division	10/4	2.5
//	Ganzzahldivision	10//4	2
%	Divisionsrest	17%6	5
**	Potenzieren	-2**4	-16

Funktionen für ganze Zahlen (Auswahl)

`abs(`n`)`	Liefert den Absolutwert der ganzen Zahl n zurück.
`bin(`n`)`	Wandelt die ganze Zahl n vom 10er- ins 2er-System um.
`oct(`n`)`	Wandelt die ganze Zahl n vom 10er- ins 8er-System um.
`hex(`n`)`	Wandelt die ganze Zahl n vom 10er- ins 16er-System um.
`str(`n`)`	Wandelt die ganze Zahl n in eine Zeichenkette um.

Darstellung von Gleitkommazahlen

- `0.00734`, `.00734`, `7.34e-3`, `7.34E-3`
- `-42598.0`, `-4.2598e4`, `-4.2598E4`,
- `8.0`, `8.`
- `3e4`, `3E4`

Operationen für Gleitkommazahlen

Operator	Beschreibung	Beispiel	Wert
-	monadisches Minus	`-17.4`	
+	Addition	`23.4+17.6`	
-	Subtraktion	`23.4-17.6`	
*	Multiplikation	`3.2*4.1`	
/	Division	`13.12/4.1`	
**	Potenzieren	`1.3**4`	

Operationen für Gleitkommazahlen

Operator	Beschreibung	Beispiel	Wert
-	monadisches Minus	`-17.4`	`-17.4`
+	Addition	`23.4+17.6`	
-	Subtraktion	`23.4-17.6`	
*	Multiplikation	`3.2*4.1`	
/	Division	`13.12/4.1`	
**	Potenzieren	`1.3**4`	

Operationen für Gleitkommazahlen

Operator	Beschreibung	Beispiel	Wert
-	monadisches Minus	`-17.4`	`-17.4`
+	Addition	`23.4+17.6`	`41.0`
-	Subtraktion	`23.4-17.6`	
*	Multiplikation	`3.2*4.1`	
/	Division	`13.12/4.1`	
**	Potenzieren	`1.3**4`	

Operationen für Gleitkommazahlen

Operator	Beschreibung	Beispiel	Wert
-	monadisches Minus	`-17.4`	`-17.4`
+	Addition	`23.4+17.6`	`41.0`
-	Subtraktion	`23.4-17.6`	`5.79999...`
*	Multiplikation	`3.2*4.1`	
/	Division	`13.12/4.1`	
**	Potenzieren	`1.3**4`	

Operationen für Gleitkommazahlen

Operator	Beschreibung	Beispiel	Wert
-	monadisches Minus	`-17.4`	`-17.4`
+	Addition	`23.4+17.6`	`41.0`
-	Subtraktion	`23.4-17.6`	`5.79999...`
*	Multiplikation	`3.2*4.1`	`13.12`
/	Division	`13.12/4.1`	
**	Potenzieren	`1.3**4`	

Operationen für Gleitkommazahlen

Operator	Beschreibung	Beispiel	Wert
-	monadisches Minus	`-17.4`	`-17.4`
+	Addition	`23.4+17.6`	`41.0`
-	Subtraktion	`23.4-17.6`	`5.79999...`
*	Multiplikation	`3.2*4.1`	`13.12`
/	Division	`13.12/4.1`	`3.2`
**	Potenzieren	`1.3**4`	

Operationen für Gleitkommazahlen

Operator	Beschreibung	Beispiel	Wert
-	monadisches Minus	`-17.4`	`-17.4`
+	Addition	`23.4+17.6`	`41.0`
-	Subtraktion	`23.4-17.6`	`5.79999...`
*	Multiplikation	`3.2*4.1`	`13.12`
/	Division	`13.12/4.1`	`3.2`
**	Potenzieren	`1.3**4`	`2.856100...`

Bemerkungen

- Ohne Klammern werden Operationen gleicher Stufe von links nach rechts gerechnet.
- Alle Operationen bewahren den Datentyp.
- Kommen Gleitkommazahlen und ganze Zahlen in einem arithmetischen Ausdruck vor, so ist das Ergebnis eine Gleitkommazahl.

Funktionen für Gleitkommazahlen

Für die „höheren“ mathematischen Funktion ist vorgängig mit `import math` das `math`-Modul zu laden. Hier eine Auswahl:

Ausdruck	**Wert**
`math.sqrt(`x`)`	Quadratwurzel von x
`math.sin(`x`)`	Sinuswert von x (Bogenmass)
`math.cos(`x`)`	Cosinuswert von x (Bogenmass)
`math.tan(`x`)`	Tangenswert von x (Bogenmass)
`math.pow(`x`,`y`)`	Wert der Potenz x^y
`math.exp(`x`)`	Wert der Exponentialfunktion zur Basis e.
`math.log(`x`)`	Wert der Logarithmusfunktion zur Basis e.
`math.degrees(`x`)`	Rechnet x vom Bogen- ins Gradmass um.
`math.radians(`x`)`	Rechnet x vom Grad- ins Bogenmass um.
`math.pi`	Wert der Kreiszahl $\pi = 3.141\ldots$

Darstellung von Wahrheitswerten

`True`, `False`

Operatoren für Wahrheitswerte

x	not x
True	
False	

x	y	x and y
False	False	
False	True	
True	False	
True	True	

x	y	x or y
False	False	
False	True	
True	False	
True	True	

Präzedenz (Vorrang) der Operatoren: `not` *vor* `and` *vor* `or`

Operatoren für Wahrheitswerte

x	not x
True	False
False	

x	y	x and y
False	False	
False	True	
True	False	
True	True	

x	y	x or y
False	False	
False	True	
True	False	
True	True	

Präzedenz (Vorrang) der Operatoren: `not` *vor* `and` *vor* `or`

Operatoren für Wahrheitswerte

x	not x
True	False
False	True

x	y	x and y
False	False	
False	True	
True	False	
True	True	

x	y	x or y
False	False	
False	True	
True	False	
True	True	

Präzedenz (Vorrang) der Operatoren: `not` *vor* `and` *vor* `or`

Operatoren für Wahrheitswerte

x	not x
True	False
False	True

x	y	x and y
False	False	False
False	True	
True	False	
True	True	

x	y	x or y
False	False	
False	True	
True	False	
True	True	

Präzedenz (Vorrang) der Operatoren: `not` *vor* `and` *vor* `or`

Operatoren für Wahrheitswerte

x	not x
True	False
False	True

x	y	x and y
False	False	False
False	True	False
True	False	
True	True	

x	y	x or y
False	False	
False	True	
True	False	
True	True	

Präzedenz (Vorrang) der Operatoren: `not` *vor* `and` *vor* `or`

Operatoren für Wahrheitswerte

x	not x
True	False
False	True

x	y	x and y
False	False	False
False	True	False
True	False	False
True	True	

x	y	x or y
False	False	
False	True	
True	False	
True	True	

Präzedenz (Vorrang) der Operatoren: not *vor* and *vor* or

Operatoren für Wahrheitswerte

x	not x
True	False
False	True

x	y	x and y
False	False	False
False	True	False
True	False	False
True	True	True

x	y	x or y
False	False	
False	True	
True	False	
True	True	

Präzedenz (Vorrang) der Operatoren: not *vor* and *vor* or

Operatoren für Wahrheitswerte

x	not x
True	False
False	True

x	y	x and y
False	False	False
False	True	False
True	False	False
True	True	True

x	y	x or y
False	False	False
False	True	
True	False	
True	True	

Präzedenz (Vorrang) der Operatoren: not *vor* and *vor* or

Operatoren für Wahrheitswerte

x	not x
True	False
False	True

x	y	x and y
False	False	False
False	True	False
True	False	False
True	True	True

x	y	x or y
False	False	False
False	True	True
True	False	
True	True	

Präzedenz (Vorrang) der Operatoren: not *vor* and *vor* or

Operatoren für Wahrheitswerte

x	not x
True	False
False	True

x	y	x and y
False	False	False
False	True	False
True	False	False
True	True	True

x	y	x or y
False	False	False
False	True	True
True	False	True
True	True	

Präzedenz (Vorrang) der Operatoren: not *vor* and *vor* or

Operatoren für Wahrheitswerte

x	not x
True	False
False	True

x	y	x and y
False	False	False
False	True	False
True	False	False
True	True	True

x	y	x or y
False	False	False
False	True	True
True	False	True
True	True	True

Präzedenz (Vorrang) der Operatoren: `not` *vor* `and` *vor* `or`

Beispiel 2.3.1

```
1  print(not True and False))        # =>
2  print(not (True and False))       # =>
3
4  print(True or True and False)     # =>
5  print((True or True) and False)   # =>
```

Beispiel 2.3.1

```
1  print(not True and False))        # => False
2  print(not (True and False))       # => True
3
4  print(True or True and False)     # => True
5  print((True or True) and False)   # => False
```

Vergleichsoperatoren

Operator	Bedeutung	Operator	Bedeutung
`<`	kleiner als	`<=`	kleiner gleich
`>`	grösser als	`>=`	grösser gleich
`!=`	ungleich	`==`	gleich

Die Vergleichsoperatoren können mit den logischen Operatoren kombiniert werden, haben aber eine höhere Priorität.

Python kennt für `(a<b and b<c)` die Kurzform `a<b<c`.

Beispiel 2.3.2

```
1 print(4 < 3 or 5 != 8)          # =>
2 print(5 >= 4 and not(5 == 8))   # =>
3 print(7 > 3 >= 2)               # =>
```

Beispiel 2.3.2

```
1 print(4 < 3 or 5 != 8)          # => True
2 print(5 >= 4 and not(5 == 8))   # => True
3 print(7 > 3 >= 2)               # => True
```

Ändert man den Datenyp eines Werts (z. B. von `int` zu `float` oder von `str` zu `bool`), so spricht man von einer *Typumwandlung*. Der englische Begriff dafür ist *Casting*.

Beispiel 2.4.3

```
1  print(int(True))          # =>
2  print(bool("hello"))      # =>
3  print(int(False))         # =>
4  print(str(7.5))           # =>
5  print(bool(''))           # =>
6  print(float(14))          # =>
7  print(bool(0))            # =>
8  print(int(19.3))          # =>
9  print(bool(-3.5))         # =>
```

Beispiel 2.4.3

```
1  print(int(True))          # => 1
2  print(bool("hello"))      # => True
3  print(int(False))         # => 0
4  print(str(7.5))           # => '7.5'
5  print(bool(''))           # => False
6  print(float(14))          # => 14.0
7  print(bool(0))            # => False
8  print(int(19.3))          # => 19
9  print(bool(-3.5))         # => True
```

Variablen, Objekte und Referenzen

Variablen enstehen durch den Zuweisungsoperator „=“. Die Anweisung

```
a = 39 + 3
```

löst in Python folgende Aktionen aus:

1. Der Ausdruck rechts von „=“ wird ausgewertet ($39 + 3 \rightarrow 42$).
2. Im Arbeitsspeicher wird ein Objekt mit dem Wert 42 erzeugt.
3. In einer Systemtabelle wird eine Variable mit dem Namen a erzeugt, sofern diese noch nicht existiert.
4. Es wird eine Referenz von a auf das Objekt erzeugt.

Die Funktionen `id(`*obj*`)` und `type(`*obj*`)` geben die Referenz bzw. den Typ eines Objekts *obj* zurück. Ist das Argument eine Variable *var*, so wird die Referenz und der Typ des Objekts *obj* zurückgegeben, auf das *var* zeigt.

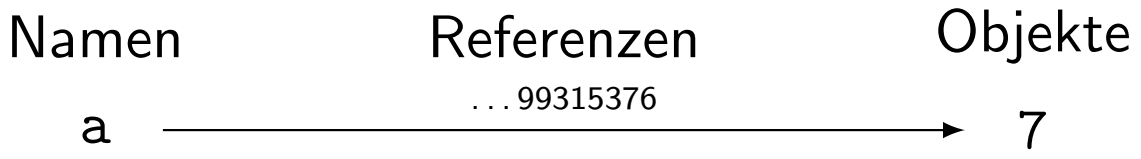
Variablen in Aktion

```
1 a = 7
2 print(type(7), id(7)) # <class 'int'> 140568399315376
3 print(type(a), id(a)) # <class 'int'> 140568399315376
4 b = a
5 print(type(b), id(b)) # <class 'int'> 140568399315376
6 a = 'hi'
7 print(type(a), id(a)) # <class 'str'> 140568398174448
8 print(type(b), id(b)) # <class 'int'> 140568399315376
```

Namen Referenzen Objekte

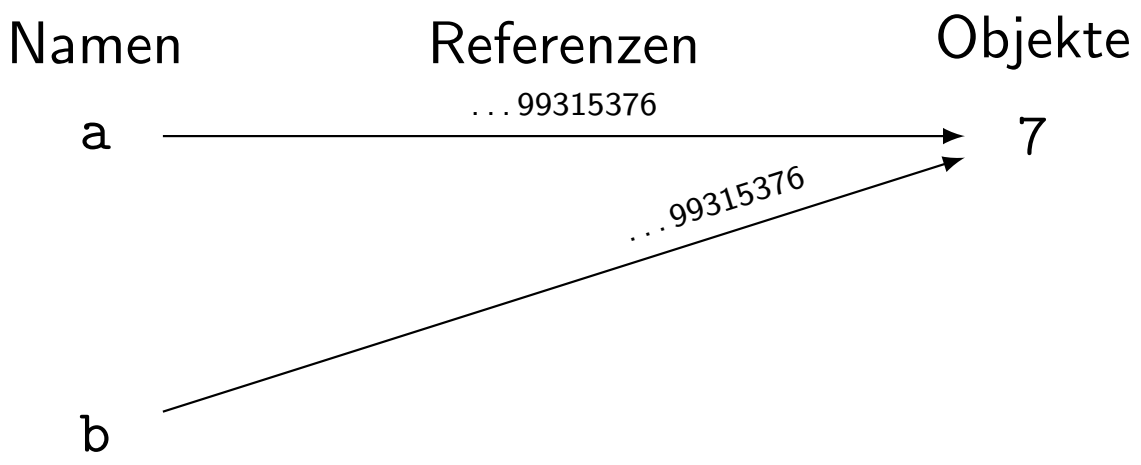
Variablen in Aktion

```
1  a = 7
2  print(type(7), id(7)) # <class 'int'> 140568399315376
3  print(type(a), id(a)) # <class 'int'> 140568399315376
4  b = a
5  print(type(b), id(b)) # <class 'int'> 140568399315376
6  a = 'hi'
7  print(type(a), id(a)) # <class 'str'> 140568398174448
8  print(type(b), id(b)) # <class 'int'> 140568399315376
```



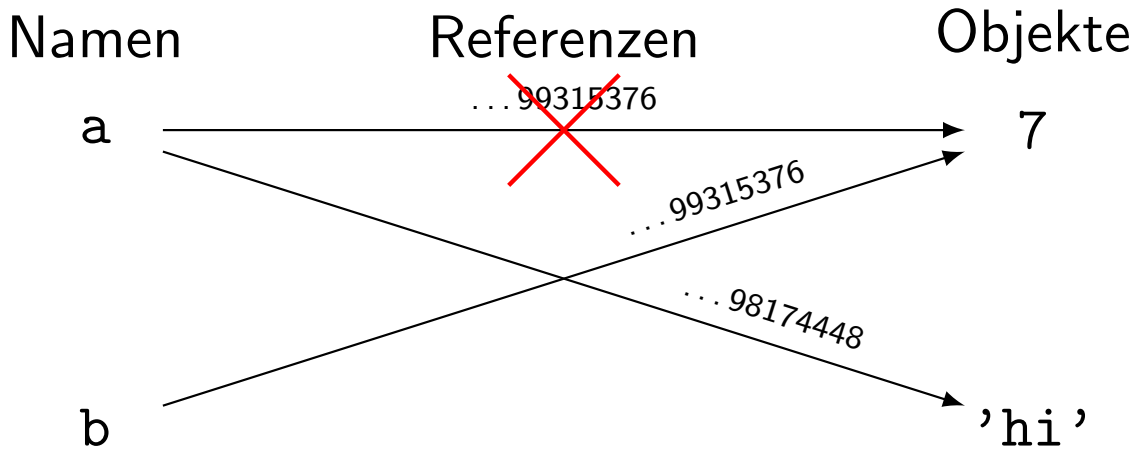
Variablen in Aktion

```
1 a = 7
2 print(type(7), id(7)) # <class 'int'> 140568399315376
3 print(type(a), id(a)) # <class 'int'> 140568399315376
4 b = a
5 print(type(b), id(b)) # <class 'int'> 140568399315376
6 a = 'hi'
7 print(type(a), id(a)) # <class 'str'> 140568398174448
8 print(type(b), id(b)) # <class 'int'> 140568399315376
```



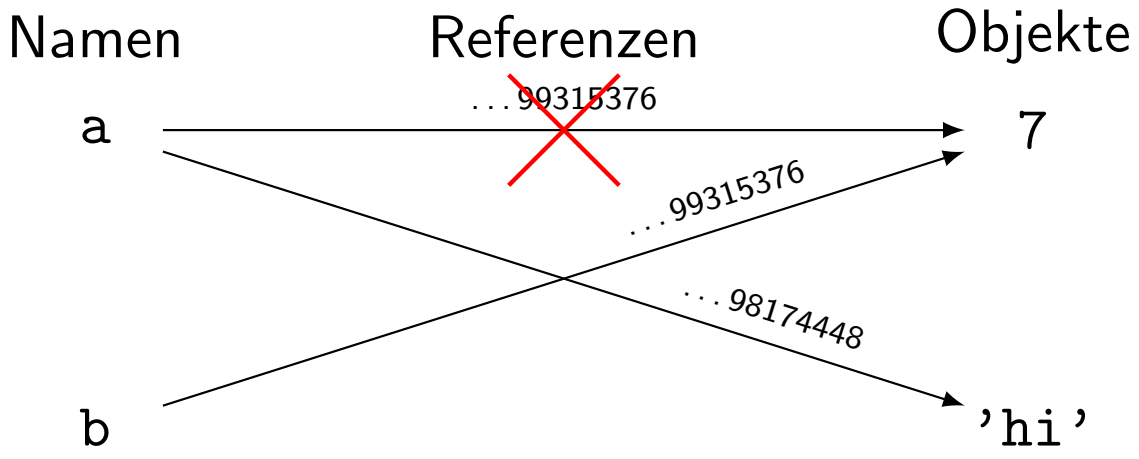
Variablen in Aktion

```
1  a = 7
2  print(type(7), id(7)) # <class 'int'> 140568399315376
3  print(type(a), id(a)) # <class 'int'> 140568399315376
4  b = a
5  print(type(b), id(b)) # <class 'int'> 140568399315376
6  a = 'hi'
7  print(type(a), id(a)) # <class 'str'> 140568398174448
8  print(type(b), id(b)) # <class 'int'> 140568399315376
```



Variablen in Aktion

```
1  a = 7
2  print(type(7), id(7)) # <class 'int'> 140568399315376
3  print(type(a), id(a)) # <class 'int'> 140568399315376
4  b = a
5  print(type(b), id(b)) # <class 'int'> 140568399315376
6  a = 'hi'
7  print(type(a), id(a)) # <class 'str'> 140568398174448
8  print(type(b), id(b)) # <class 'int'> 140568399315376
```



Die Referenzen können sich bei jeder Programmausführung ändern.

Dynamische Typisierung

Im Gegensatz zu typisierten Programmiersprachen wie C, C++ oder Java hat eine Variable in Python *keinen* Datentyp. Der Datentyp befindet sich im Objekt. Der Variable ist es „egal“, ob ihre Referenz auf ein `int`-, `float`- oder `bool`-Objekt zeigt. Daher sind Zuweisungsfolgen der Art:

```
a = 42
a = True
a = 3.14159
```

in Python möglich, auch wenn sie (im gleichen Programm) so nicht sinnvoll sind. In typisierten Programmiersprachen würde bereits die erste Zuweisung eine Fehlermeldung verursachen, da man vor der ersten Zuweisung den Typ angeben muss, den die Variable während der Laufzeit des Programms hat (und behält).

Sofortige Auswertung

Sobald in einer Anweisung eine Variable auftritt, wird sie sofort durch das Objekt ersetzt, auf das ihre Referenz verweist. Daher muss jeder Variablen vor ihrem ersten Gebrauch ein Wert zugewiesen werden. Andernfalls führt dies zu einem `NameError`.

```
1 a = 3
2 print(a, id(a)) # 3 ...1417904
3 a = a + 5
4 print(a, id(a)) # 8 ...1418064
5 a = a // 2
6 print(a, id(a)) # 4 ...1417936
```

Da jede neue Zuweisung ein neues Objekt erzeugt, ändern sich die Referenzen der Variable a laufend.

Regeln für die Bildung von Bezeichnern (Namen)

- Bezeichner beginnen mit einem Buchstaben oder einem Unterstrich. Danach können weitere Buchstaben, Unterstriche oder Ziffern folgen. Dabei wird Gross- und Kleinschreibung unterschieden.
- Diese Python-Schlüsselworte dürfen nicht als Bezeichner verwendet werden:

`False`	`await`	`else`	`import`	`pass`
`None`	`break`	`except`	`in`	`raise`
`True`	`class`	`finally`	`is`	`return`
`and`	`continue`	`for`	`lambda`	`try`
`as`	`def`	`from`	`nonlocal`	`while`
`assert`	`del`	`global`	`not`	`with`
`async`	`elif`	`if`	`or`	`yield`

Du musst nur die im Unterricht behandelten Schlüsselwörter kennen.

Beispiel 3.1

Ist der Ausdruck syntaktisch korrekt?

```
Klasse5a = 20          #
4you = 2               #
True = 3               #
_status = 'ok'         #
6 = x                  #
false = 4              #
gewinn-2024 = 12345 #
sehrLangerName = 8  #
max_wert = 100         #
```

Beispiel 3.1

Ist der Ausdruck syntaktisch korrekt?

```
1  Klasse5a = 20          # korrekt
2  4you = 2               # falsch (Ziffer am Anfang)
3  True = 3               # falsch (Schlüsselwort)
4  _status = 'ok'         # korrekt
5  6 = x                  # falsch (Variable links!)
6  false = 4              # korrekt (aber nicht empfehlenswert)
7  gewinn-2024 = 12345 # falsch (Sonderzeichen)
8  sehrLangerName = 8  # korrekt
9  max_wert = 100         # korrekt
```

Beispiel 3.2

```
1  x = 2 * 4
2  print(x)    # =>
3  y = 5 + x
4  print(y)    # =>
5  x = x - 1
6  print(x)    # =>
```

Beispiel 3.2

```
1  x = 2 * 4
2  print(x)    # => 8
3  y = 5 + x
4  print(y)    # => 13
5  x = x - 1
6  print(x)    # => 7
```

Erweiterte Zuweisungen

Wird einer Variablen der Wert einer Operation zugewiesen, in der die Variable als Operand vorkommt, ist eine erweiterte Zuweisung (augmented assignment) platzsparender und meist auch effizienter.

Operator	Beispiel	gleichwertiger Ausdruck
`+=`	`x += 7`	`x = x + 7`
`-=`	`x -= 7`	`x = x - 7`
`*=`	`x *= 7`	`x = x * 7`
`/=`	`x /= 7`	`x = x / 7`
`**=`	`x **= 7`	`x = x ** 7`
`//=`	`x //= 7`	`x = x // 7`
`%=`	`x %= 7`	`x = x % 7`

Beispiel 3.3

```
1  x = 5
2  x *= 10
3  print(x)    # =>
4  x -= 1
5  print(x)    # =>
6  x //= 8
7  print(x)    # =>
```

Beispiel 3.3

```
1  x = 5
2  x *= 10
3  print(x)    # => 50
4  x -= 1
5  print(x)    # => 49
6  x //= 8
7  print(x)    # => 6
```

Mehrfachzuweisungen

In Python können mehreren Variablen „gleichzeitig“ und unabhängig voneinander Werte zugewiesen werden. Die Mehrfachzuweisung

```
a, b = 5, 7
```

entspricht den zwei einzelnen Zuweisungen

```
a = 5
b = 7
```

Mehrfachzuweisungen sollen Programme besser lesbar machen und sind auch mit mehr als zwei Variablen möglich, sofern jedem Bezeichner links ein Wert rechts entspricht.

Beispiel 3.4

```
1 a, b, c = 4, 2, 1
2 print(b) # =>
3 a, b, c = 2*c, 3*a, -b
4 print(b) # =>
```

Beispiel 3.4

```
1  a, b, c = 4, 2, 1
2  print(b) # => 2
3  a, b, c = 2*c, 3*a, -b
4  print(b) # => 12
```

Bedingte Anweisung

```
if bedingung:
    codeblock
...
```

Der *codeblock* besteht aus mindestens einer (um 4 Blanks oder mit Tab) eingerückten Anweisung. Er wird nur dann ausgeführt, wenn *bedingung* den Wert `True` hat. In jedem Fall wird das Programm an der ersten nicht mehr eingerückten Zeile fortgesetzt.

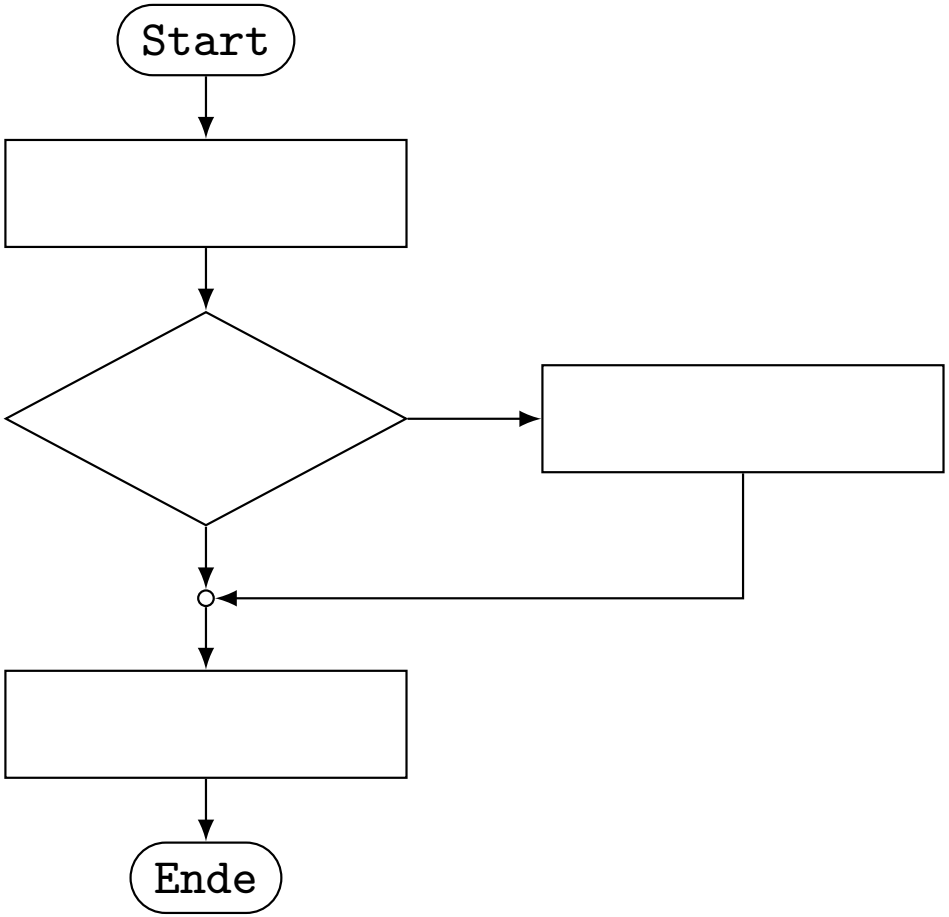
Beispiel 4.1

```
1  x = 3
2  if x > 2:
3      x = x + 1
4  x = x + 10
5  print(x)        # Ausgabe:
```

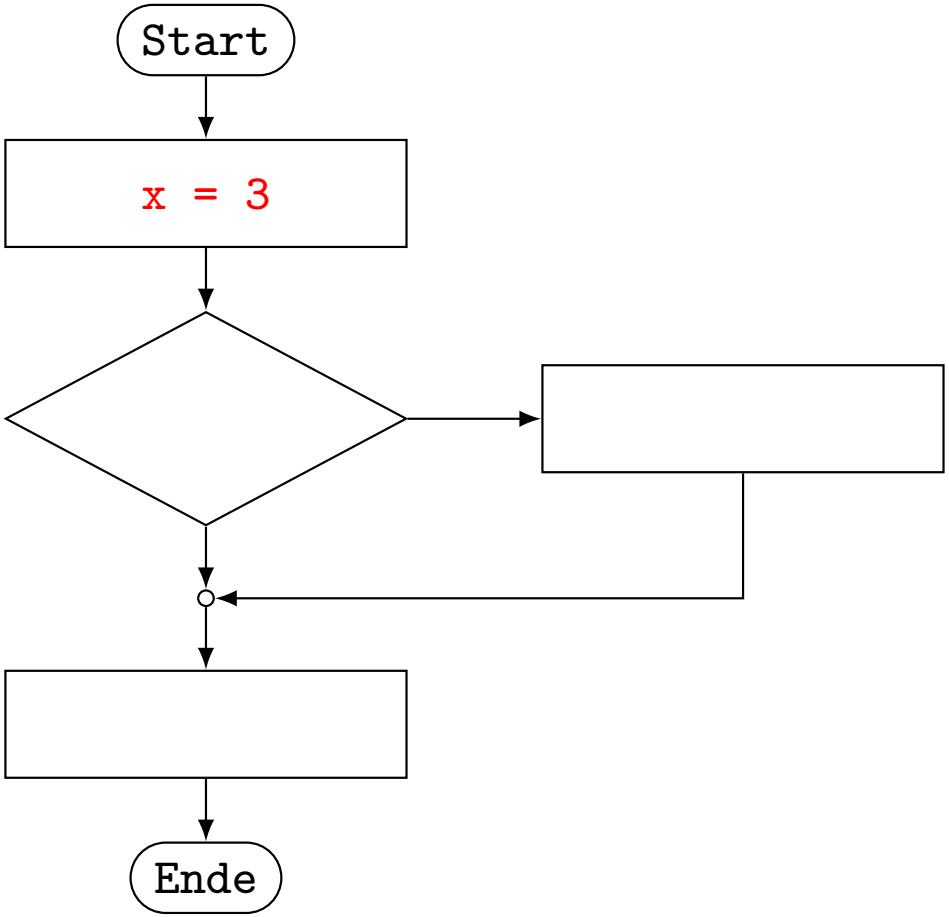
Beispiel 4.1

```
1  x = 3
2  if x > 2:
3      x = x + 1
4  x = x + 10
5  print(x)        # Ausgabe: 14
```

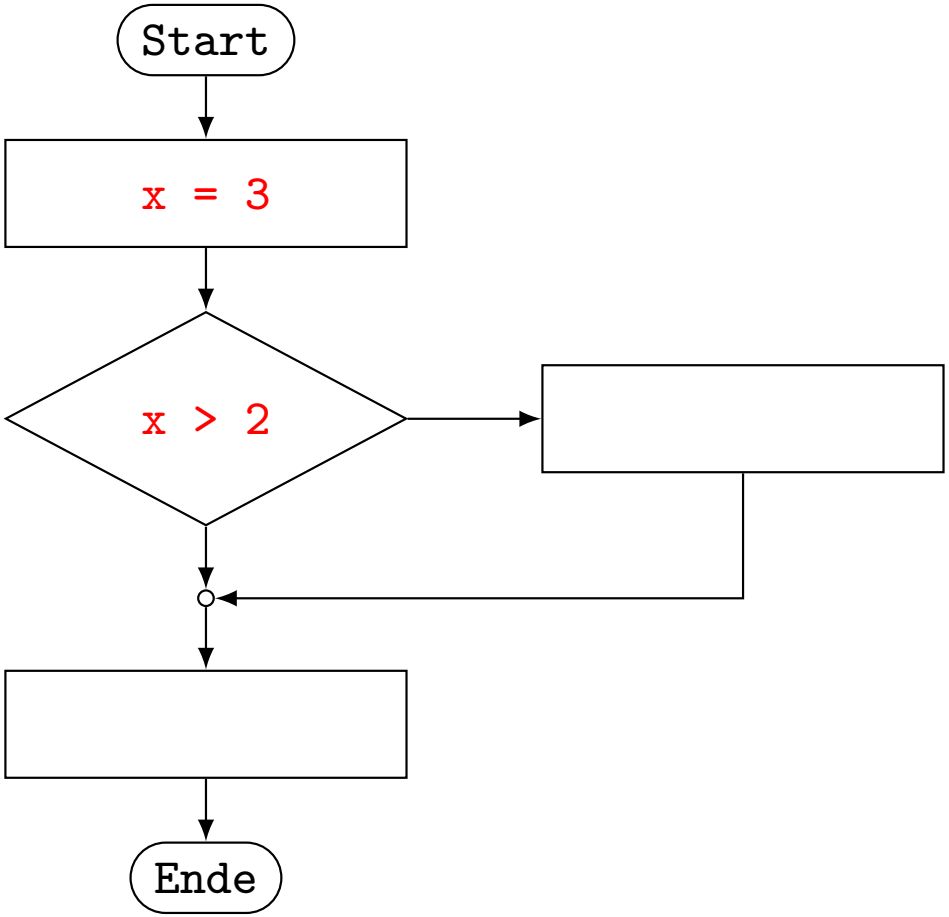
Darstellung als Flussdiagramm



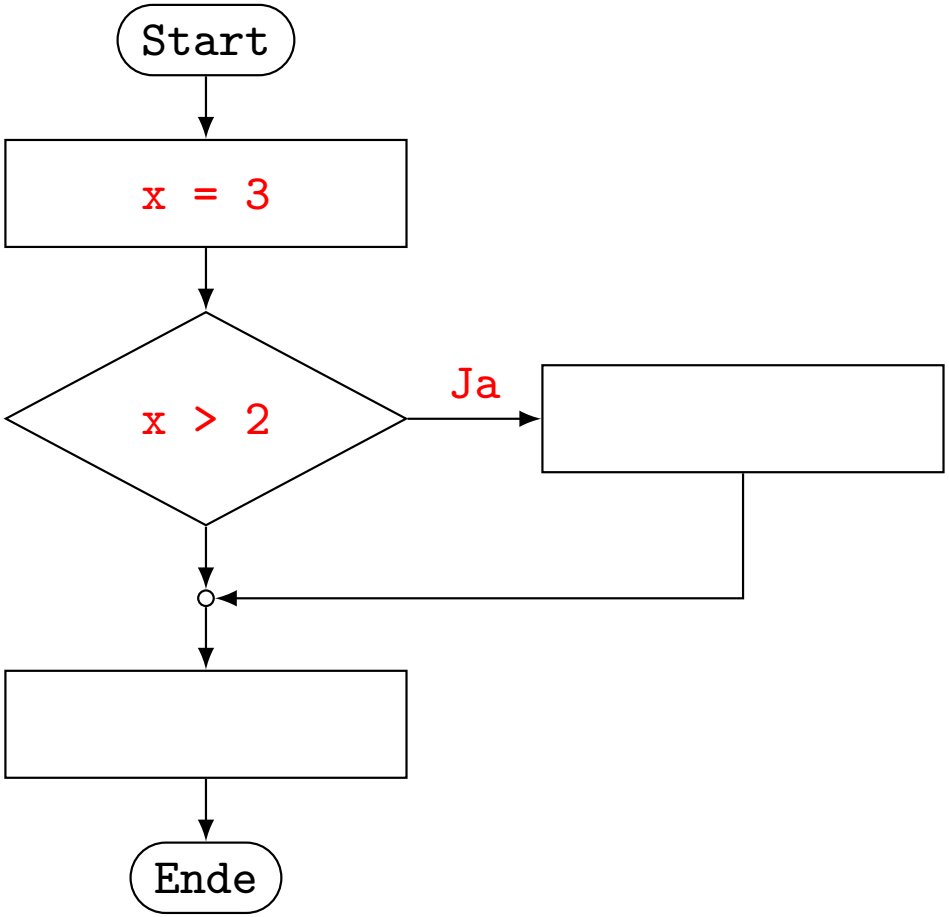
Darstellung als Flussdiagramm



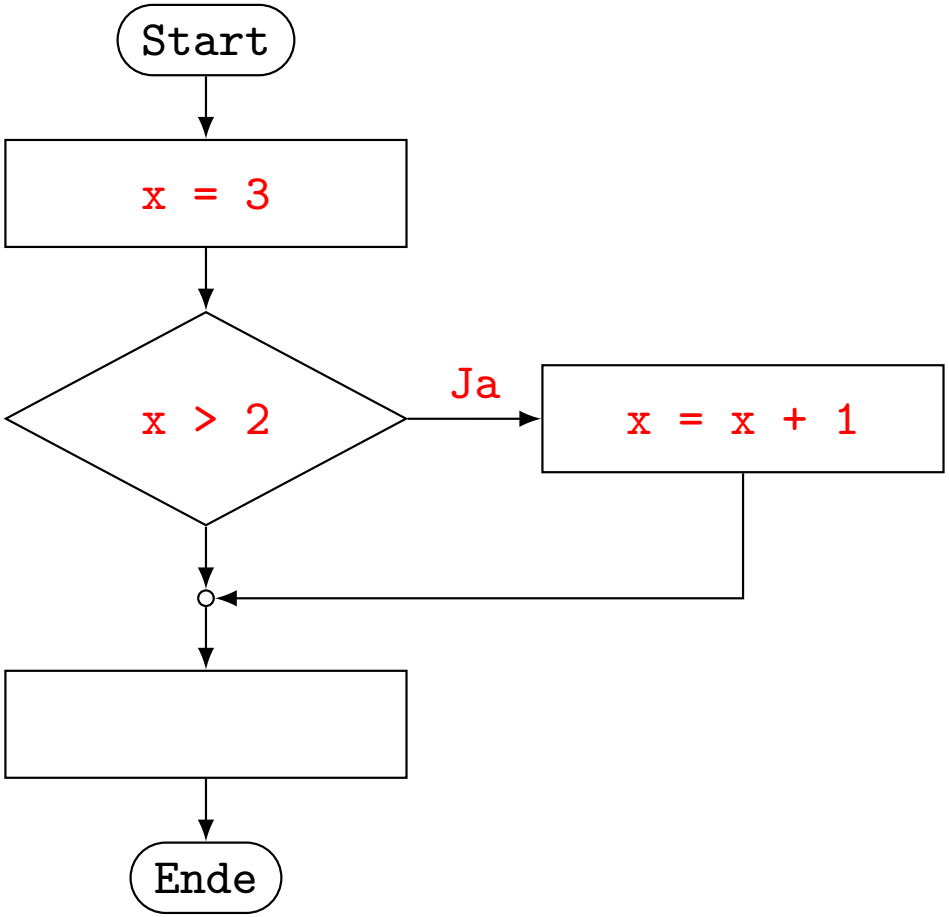
Darstellung als Flussdiagramm



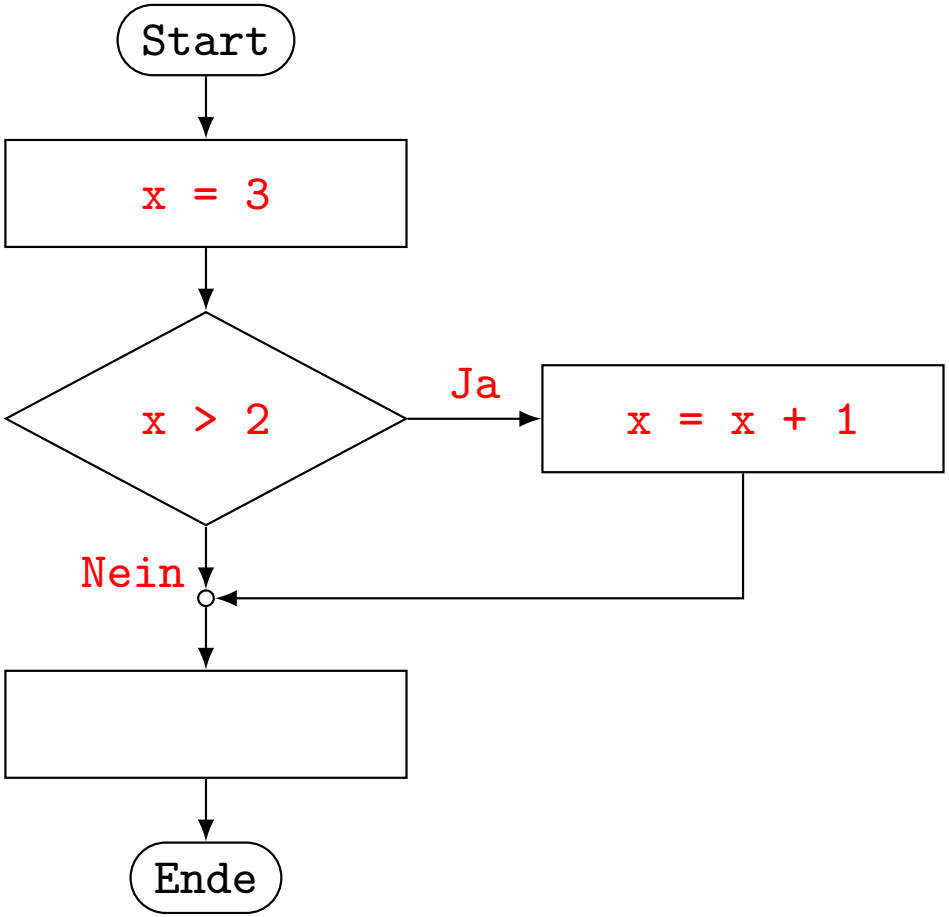
Darstellung als Flussdiagramm



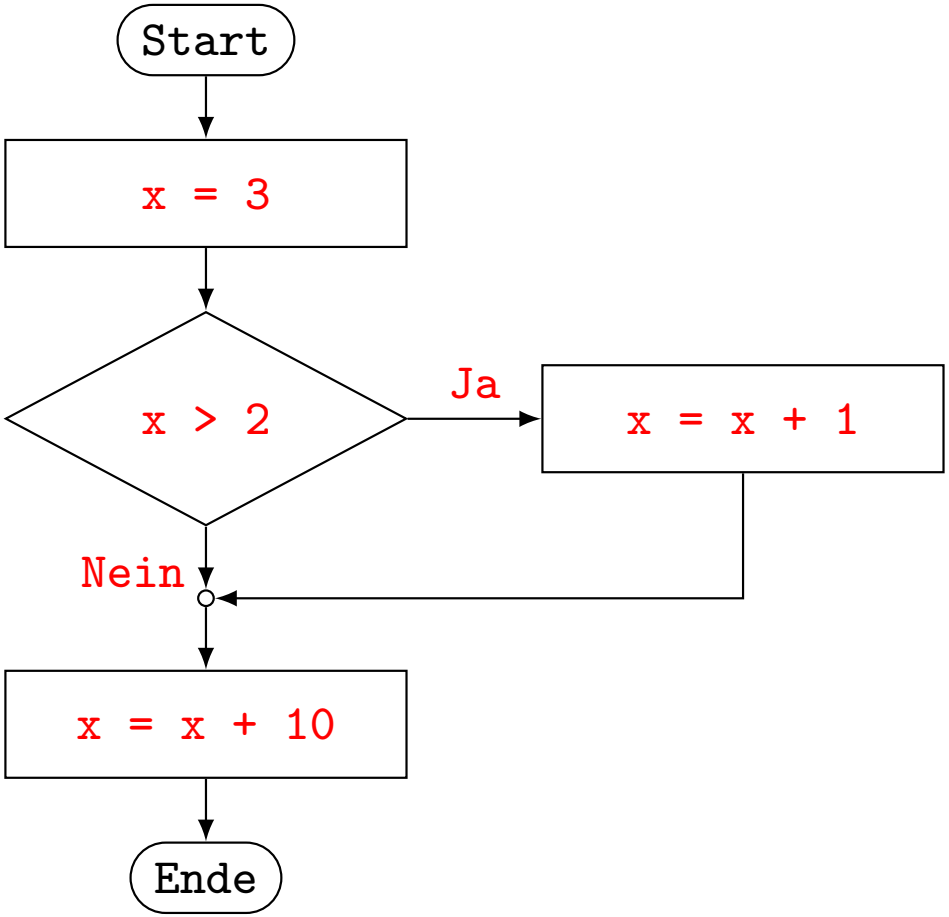
Darstellung als Flussdiagramm



Darstellung als Flussdiagramm



Darstellung als Flussdiagramm



Einfache Verzweigung

```
if bedingung:
    codeblock_1
else:
    codeblock_2
...
```

Wenn *bedingung* den Wert `True` hat wird *codeblock_1* ausgeführt, *codeblock_2* übersprungen und das Programm danach fortgesetzt. Andernfalls wird *codeblock_1* übersprungen, *codeblock_2* ausgeführt und das Programm danach fortgesetzt.

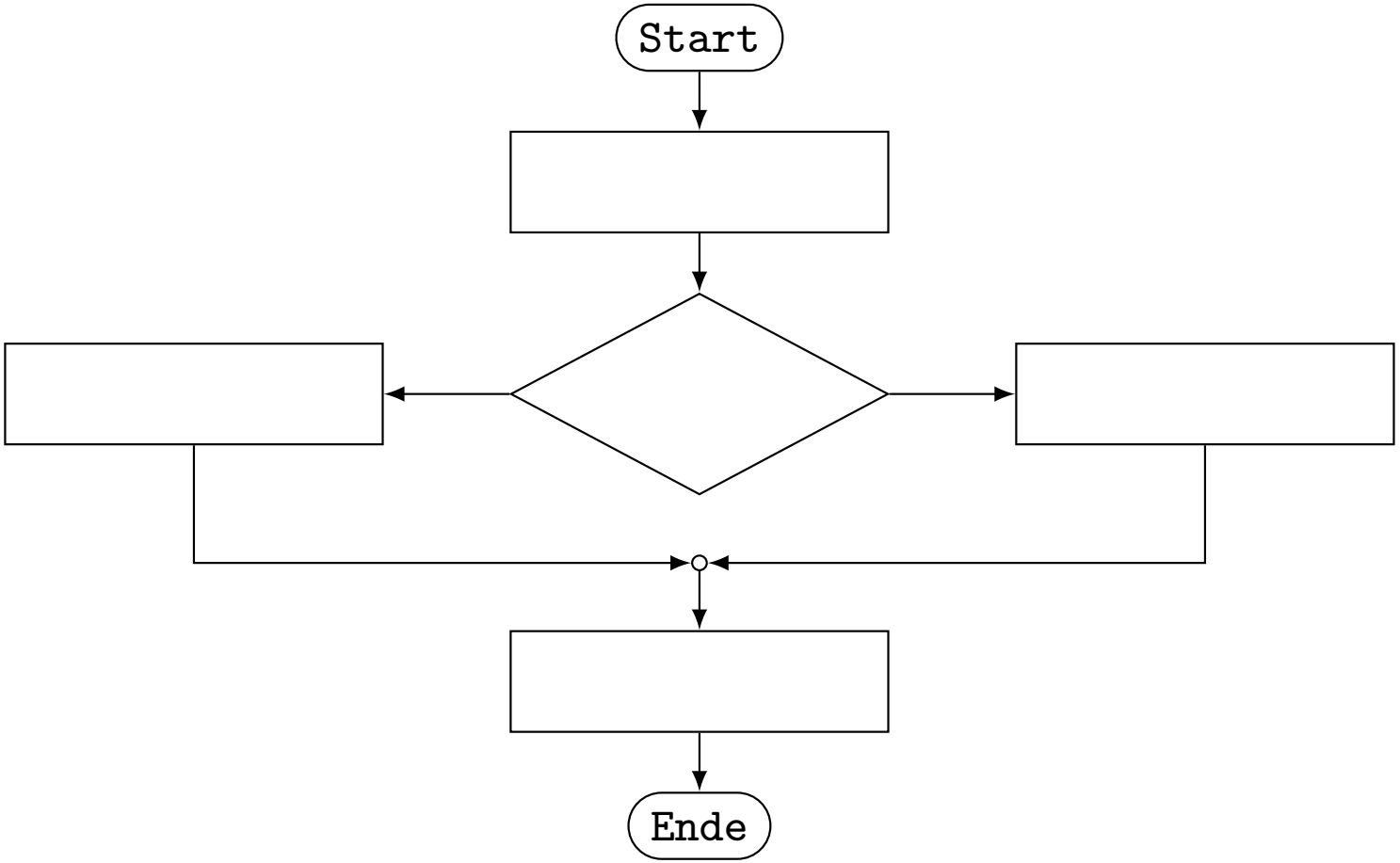
Beispiel 4.2

```
1  x = 3
2  if x > 4:
3      x = x + 1
4  else:
5      x = x - 1
6  x = x + 10
7  print(x)        # Ausgabe:
```

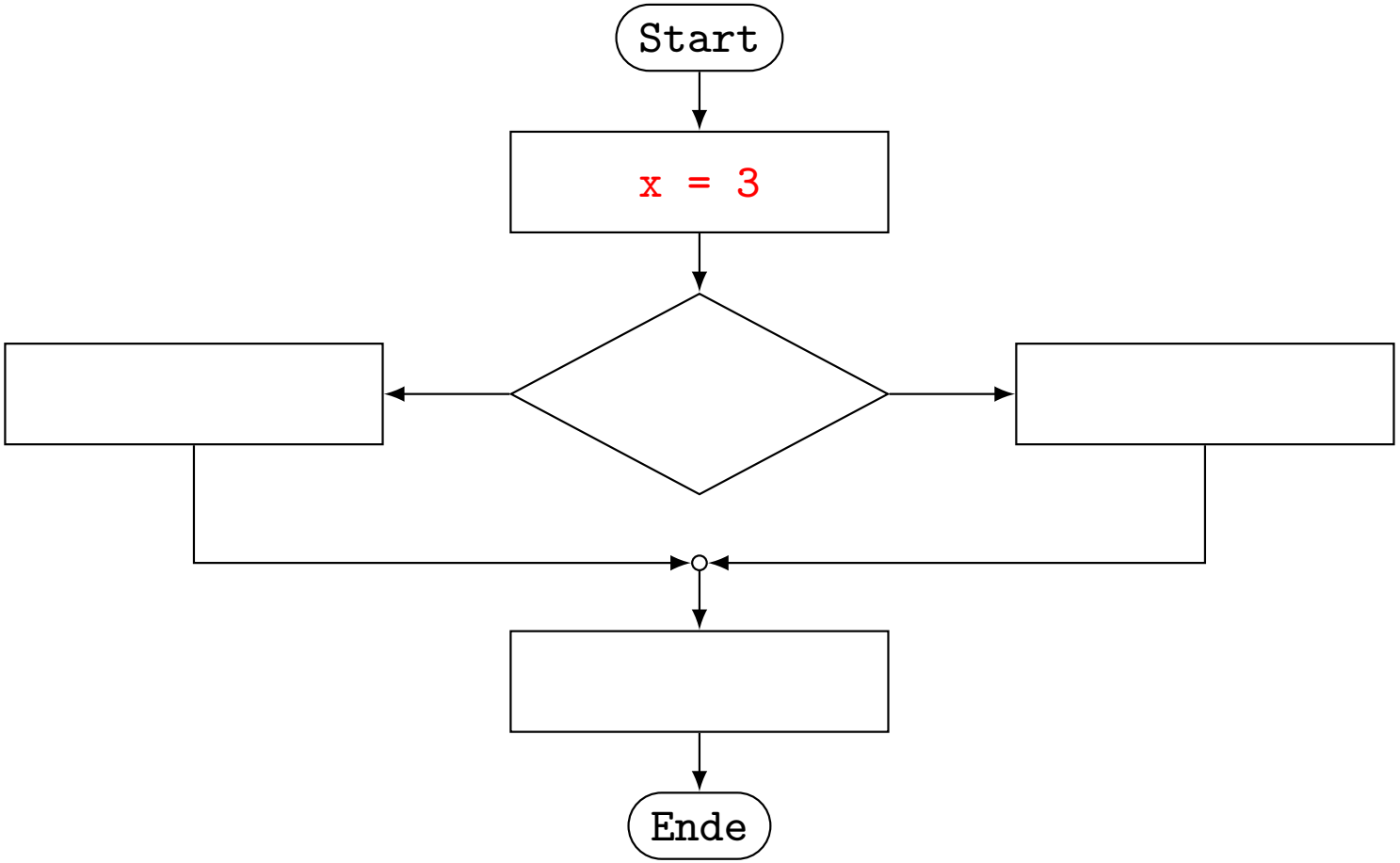
Beispiel 4.2

```
1  x = 3
2  if x > 4:
3      x = x + 1
4  else:
5      x = x - 1
6  x = x + 10
7  print(x)      # Ausgabe: 12
```

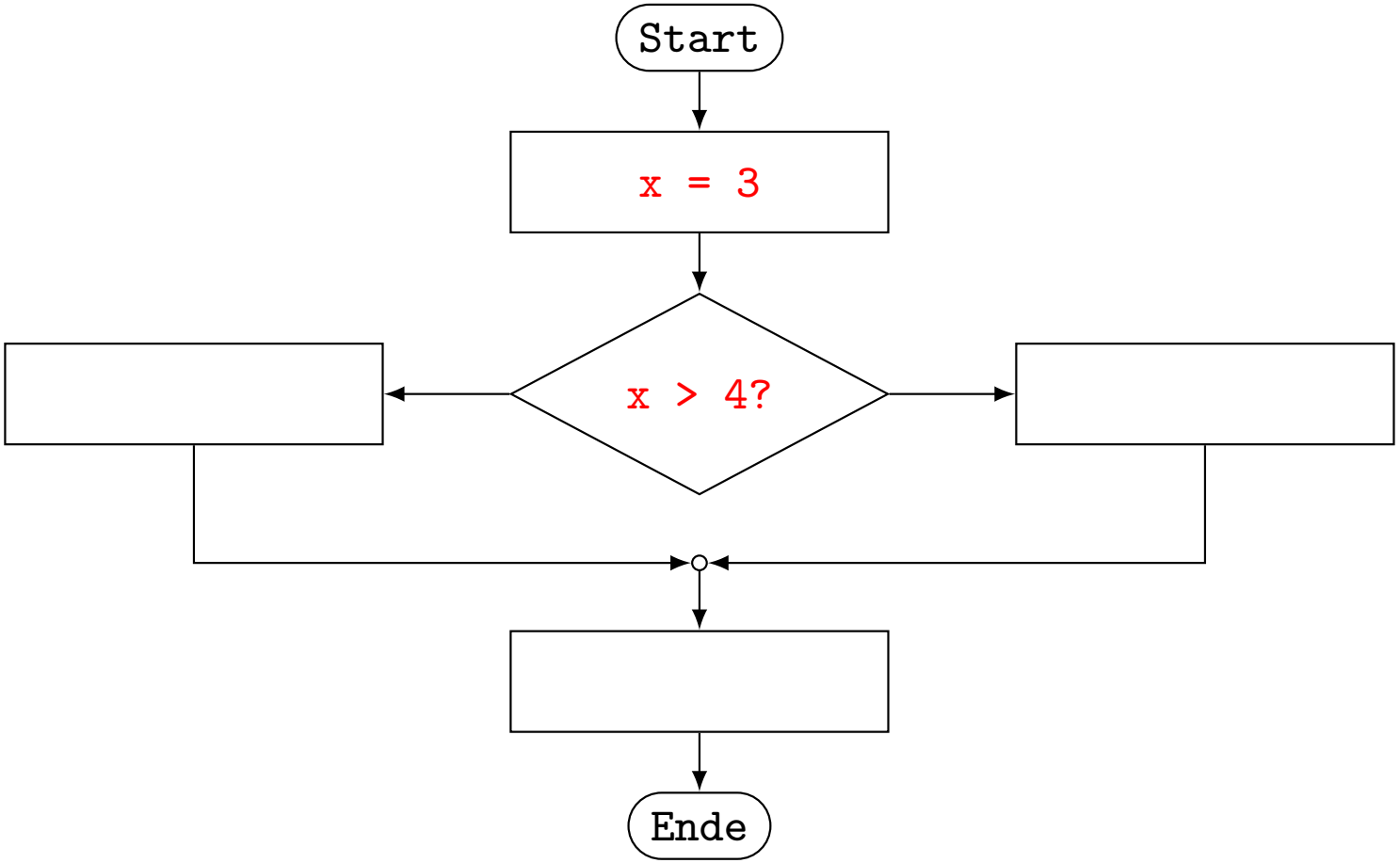
Darstellung als Flussdiagramm



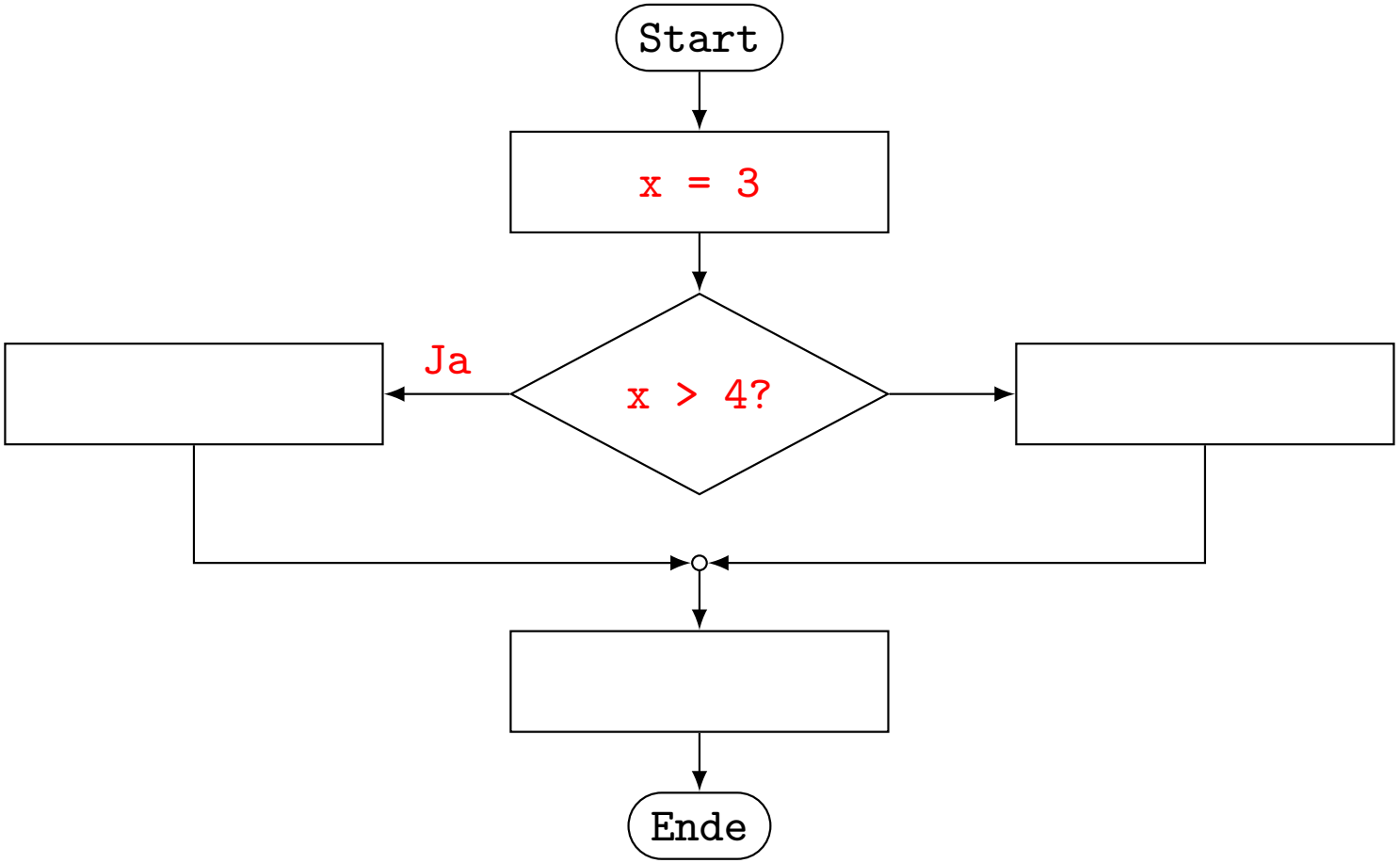
Darstellung als Flussdiagramm



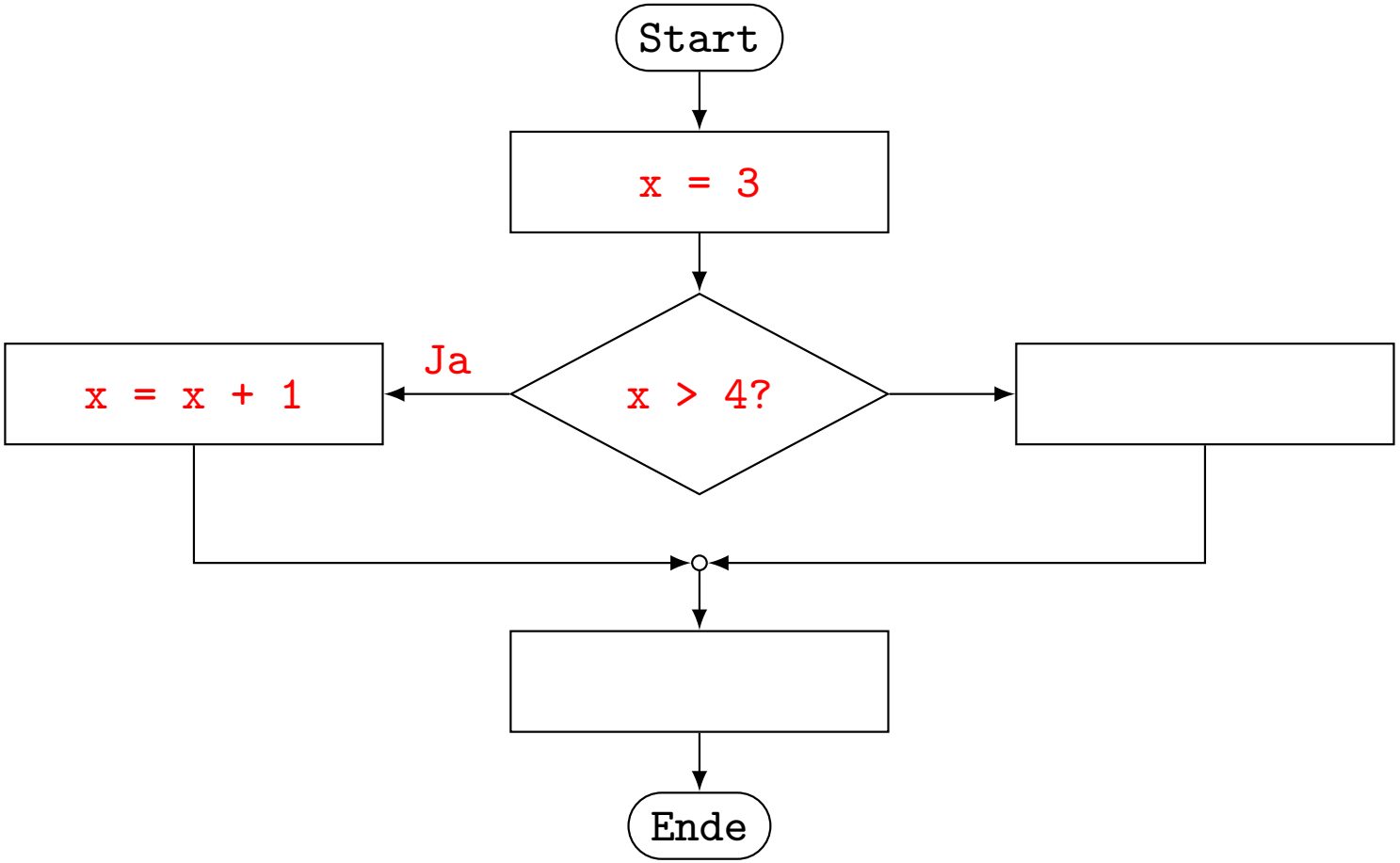
Darstellung als Flussdiagramm



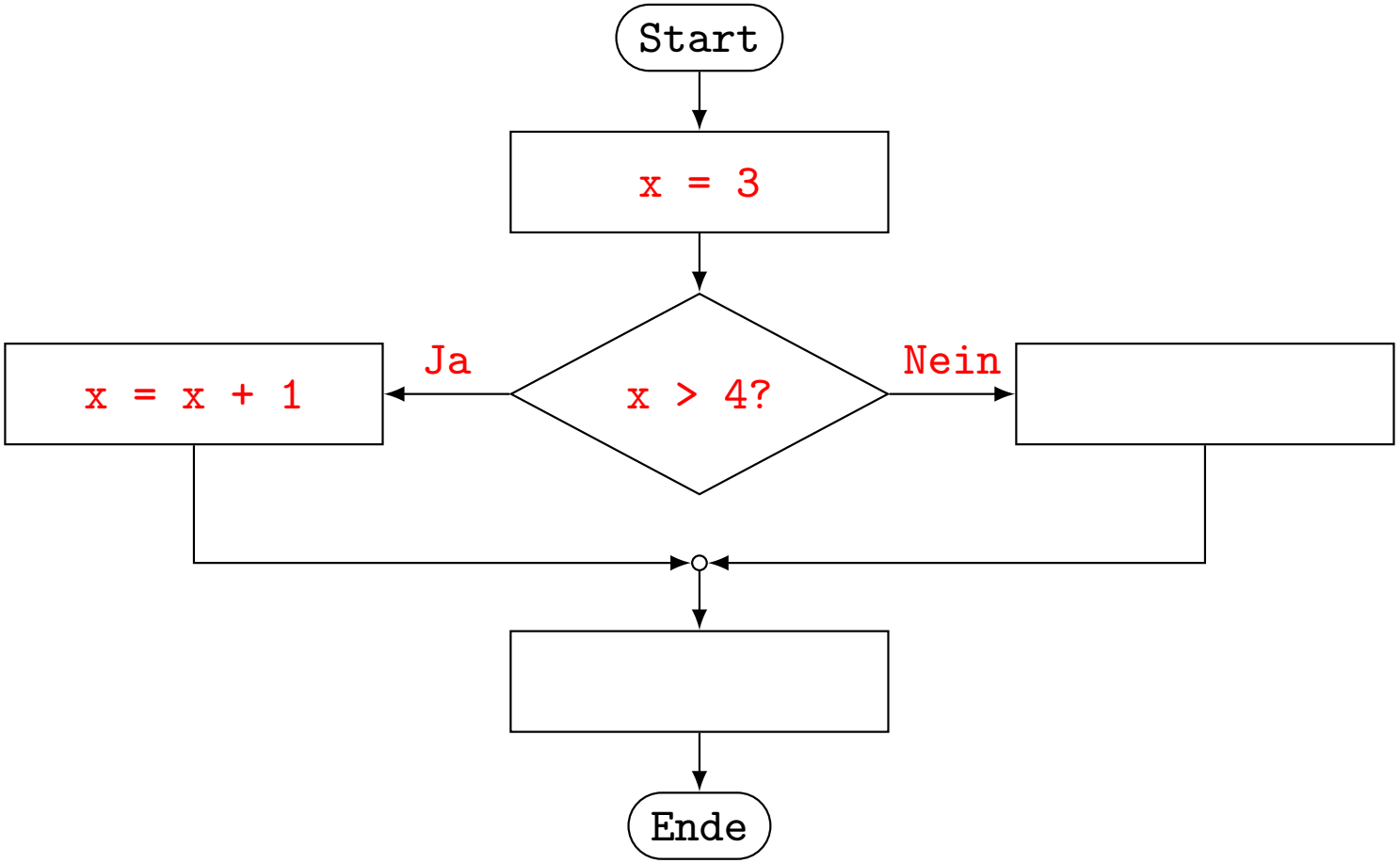
Darstellung als Flussdiagramm



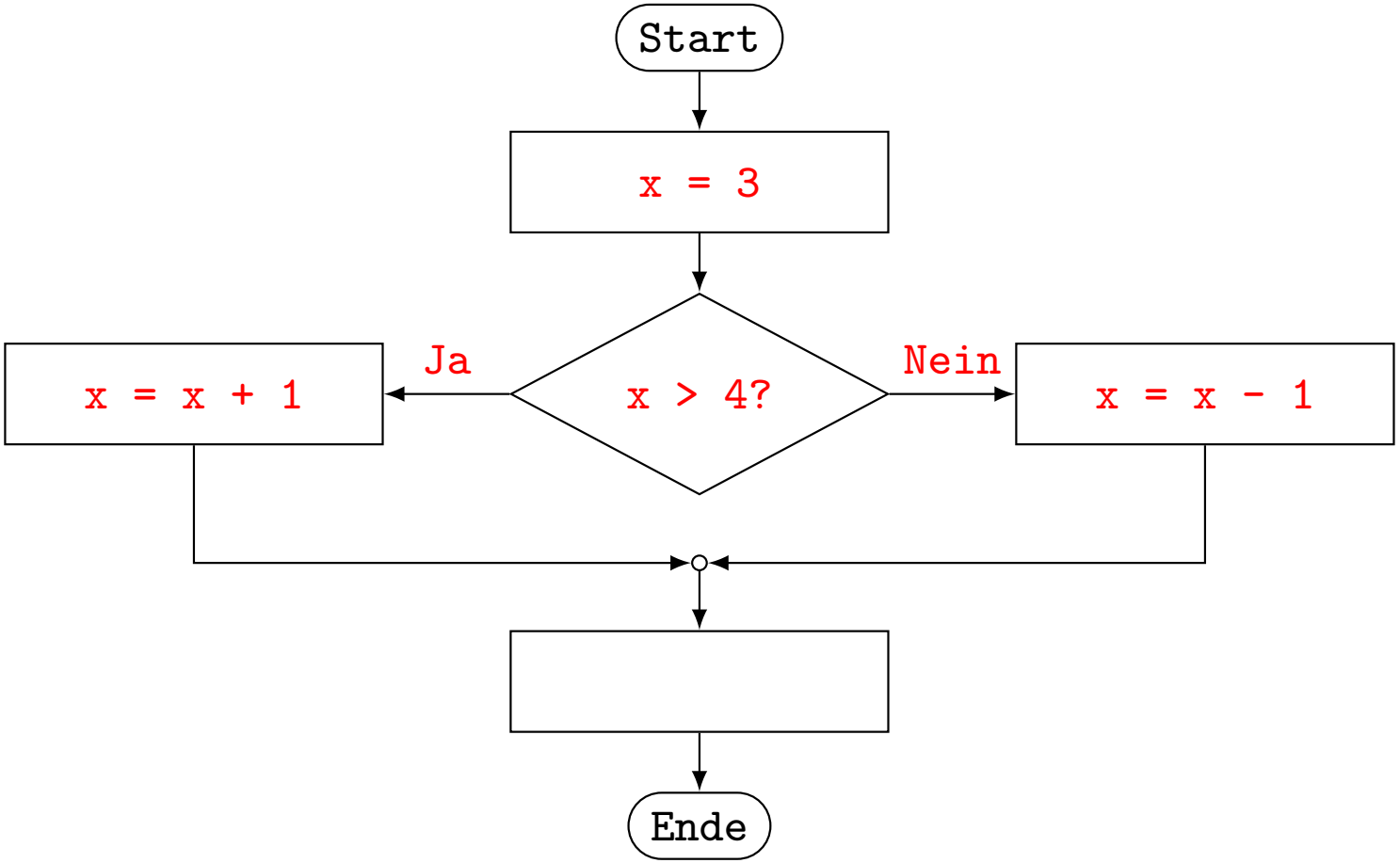
Darstellung als Flussdiagramm



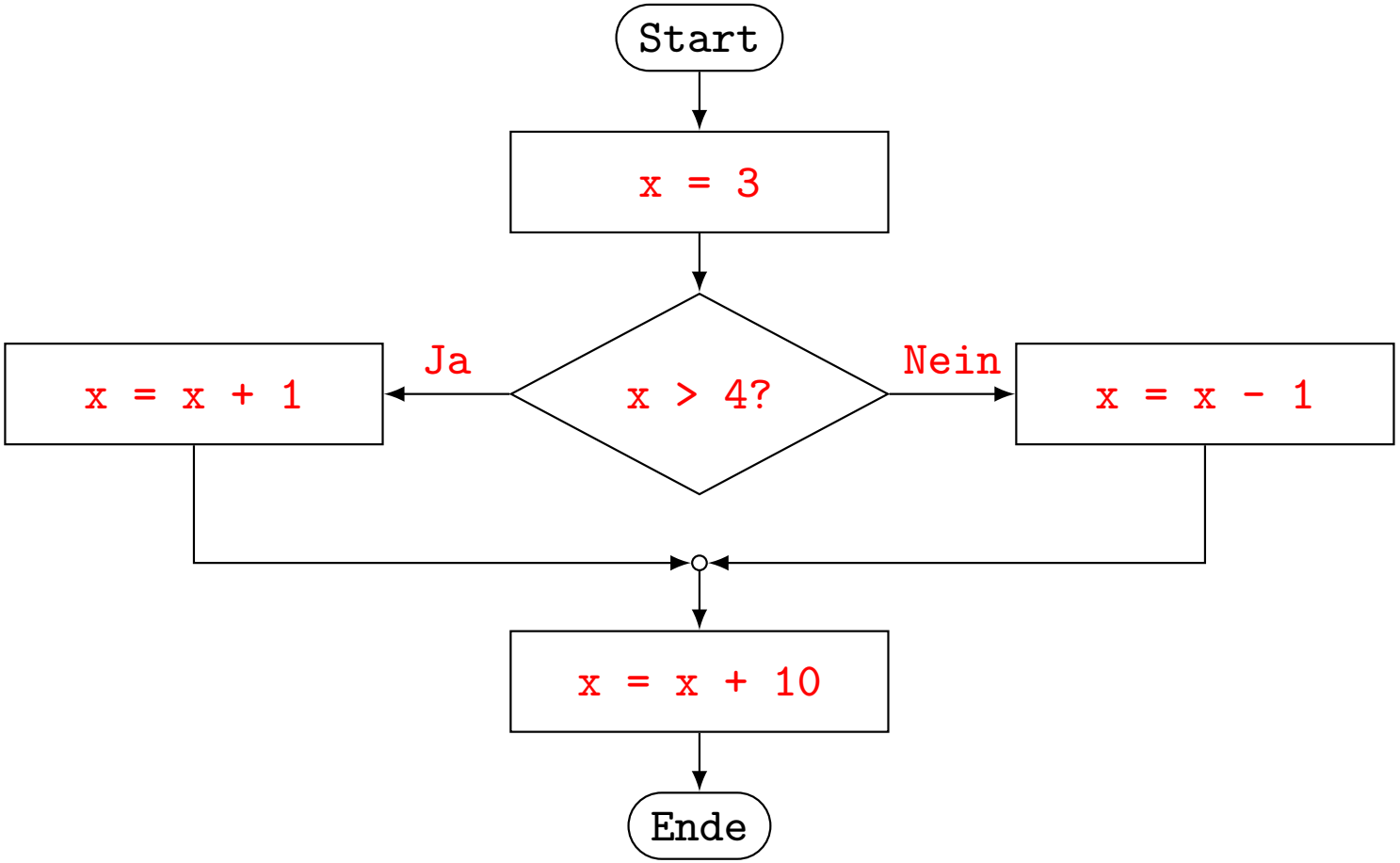
Darstellung als Flussdiagramm



Darstellung als Flussdiagramm



Darstellung als Flussdiagramm



Mehrfachverzweigung

```
if bedingung_1:
    codeblock_1
elif bedingung_2:
    codeblock_2
elif bedingung_3:
    codeblock_3
...
    ....
else:
    codeblock_n
...
```

Hat eine Verzweigung zusätzlich `elif`-Klauseln, wird nur der Codeblock nach der ersten wahren Bedingung ausgeführt. Ist keine der Bedingungen wahr, so wird der Codeblock nach `else` (ohne eine Bedingung) ausgeführt. In jedem Fall wird das Programm danach fortgesetzt.

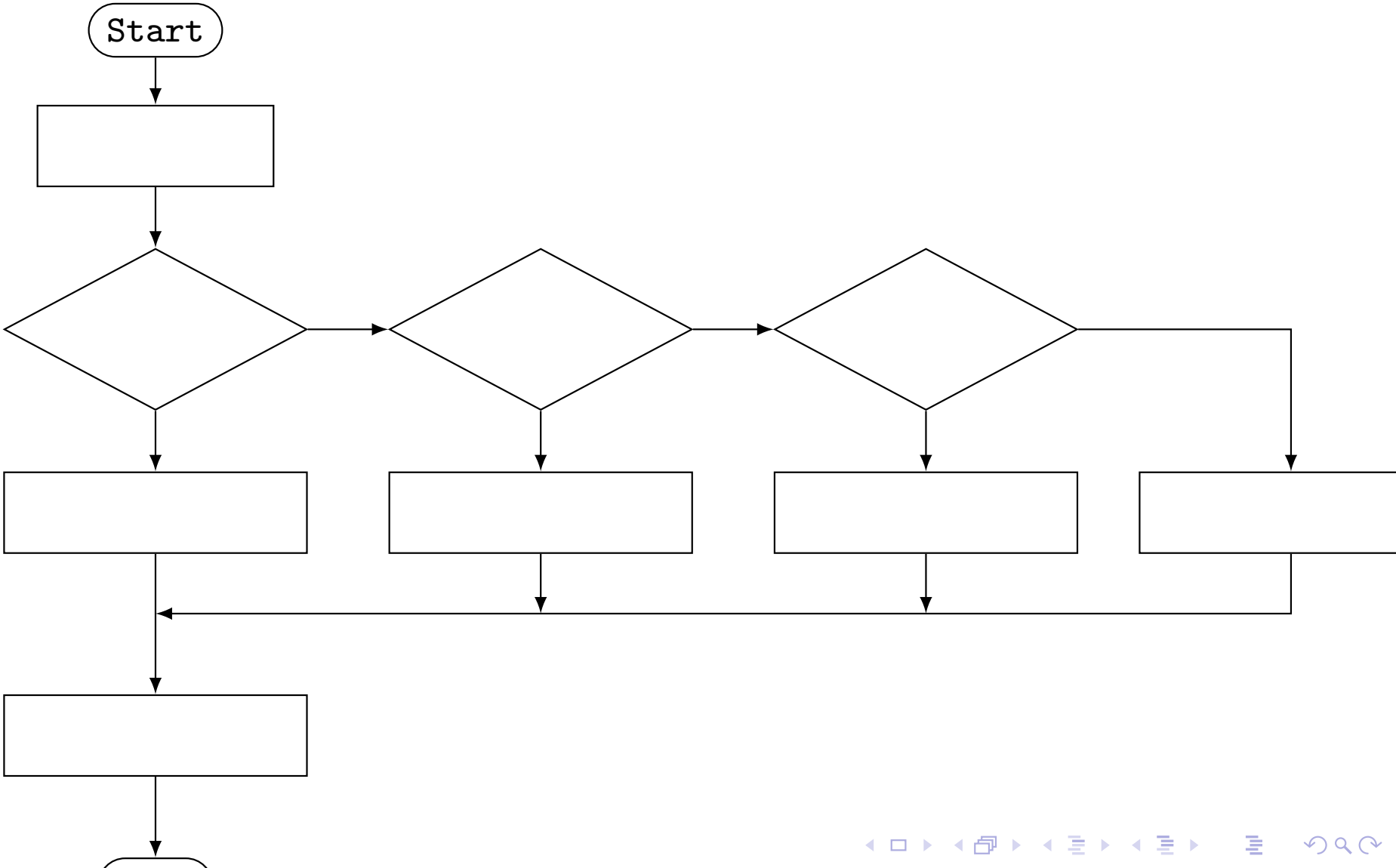
Beispiel 4.3

```
1  x = 7
2  if x < 4:
3      x = x + 1
4  elif x < 5:
5      x = x + 2
6  elif x < 6:
7      x = x + 3
8  else:
9      x = x + 4
10 x = x + 10
11 print(x)          # Ausgabe:
```

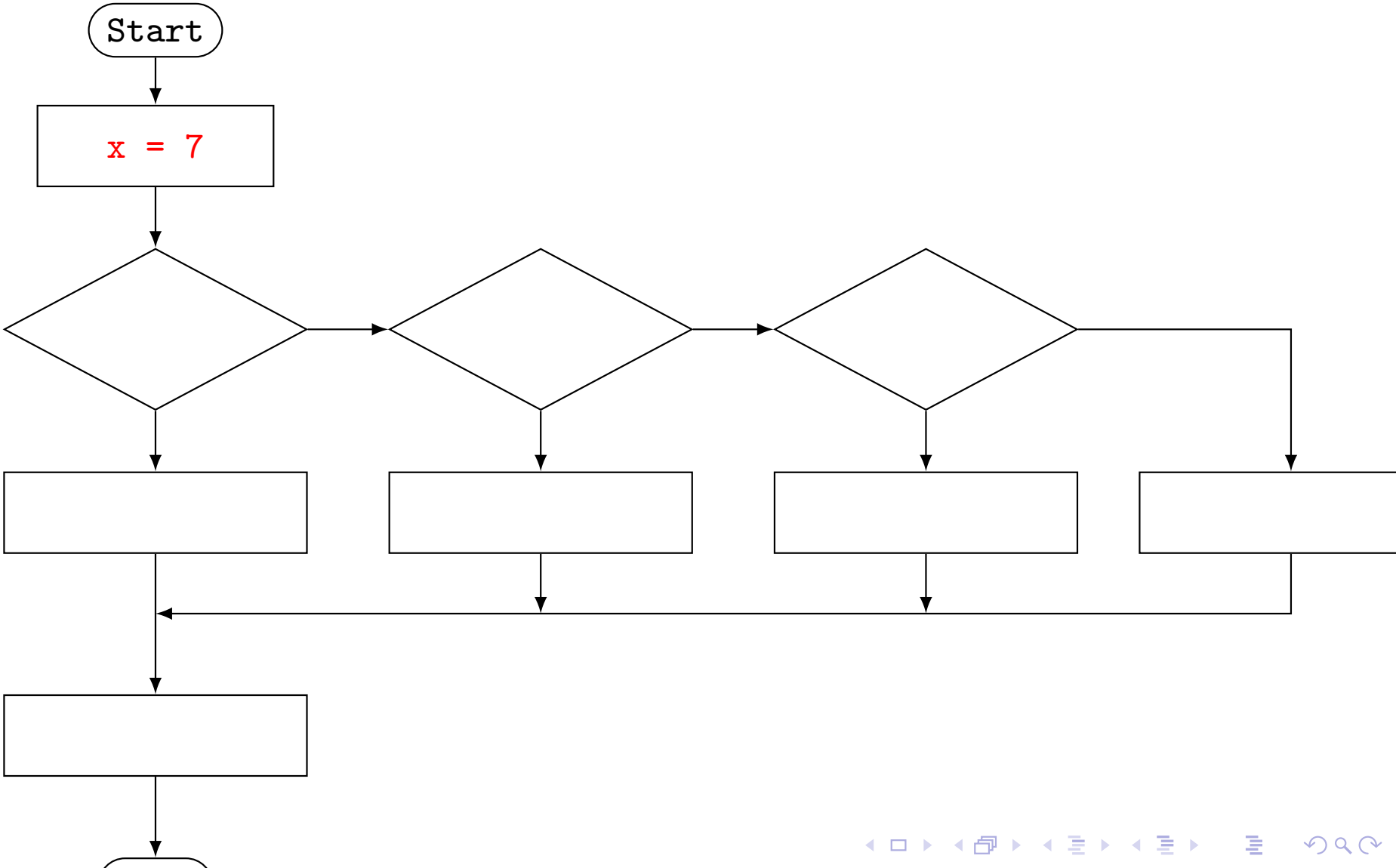
Beispiel 4.3

```
1  x = 7
2  if x < 4:
3      x = x + 1
4  elif x < 5:
5      x = x + 2
6  elif x < 6:
7      x = x + 3
8  else:
9      x = x + 4
10 x = x + 10
11 print(x)          # Ausgabe: 21
```

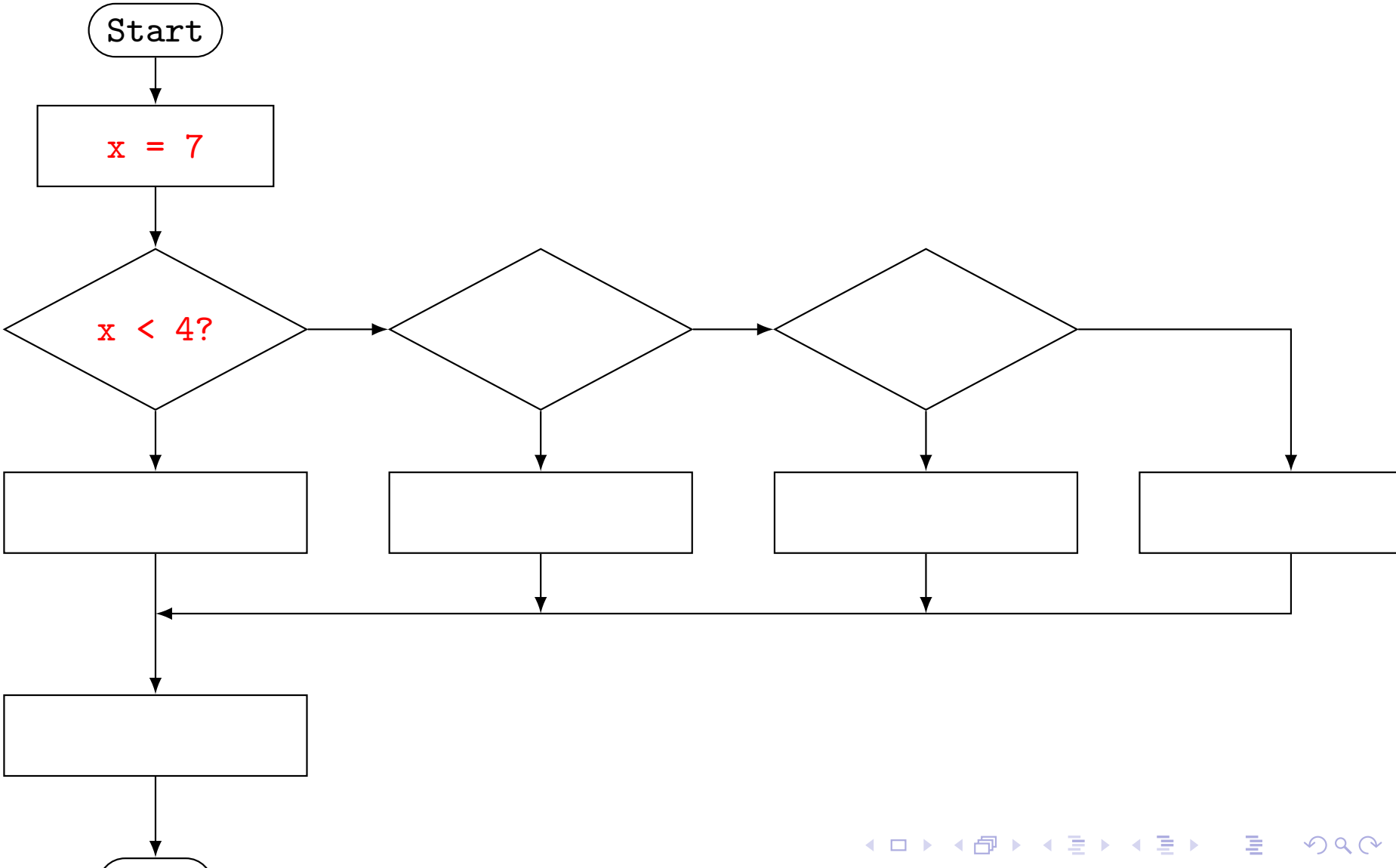
Darstellung als Flussdiagramm



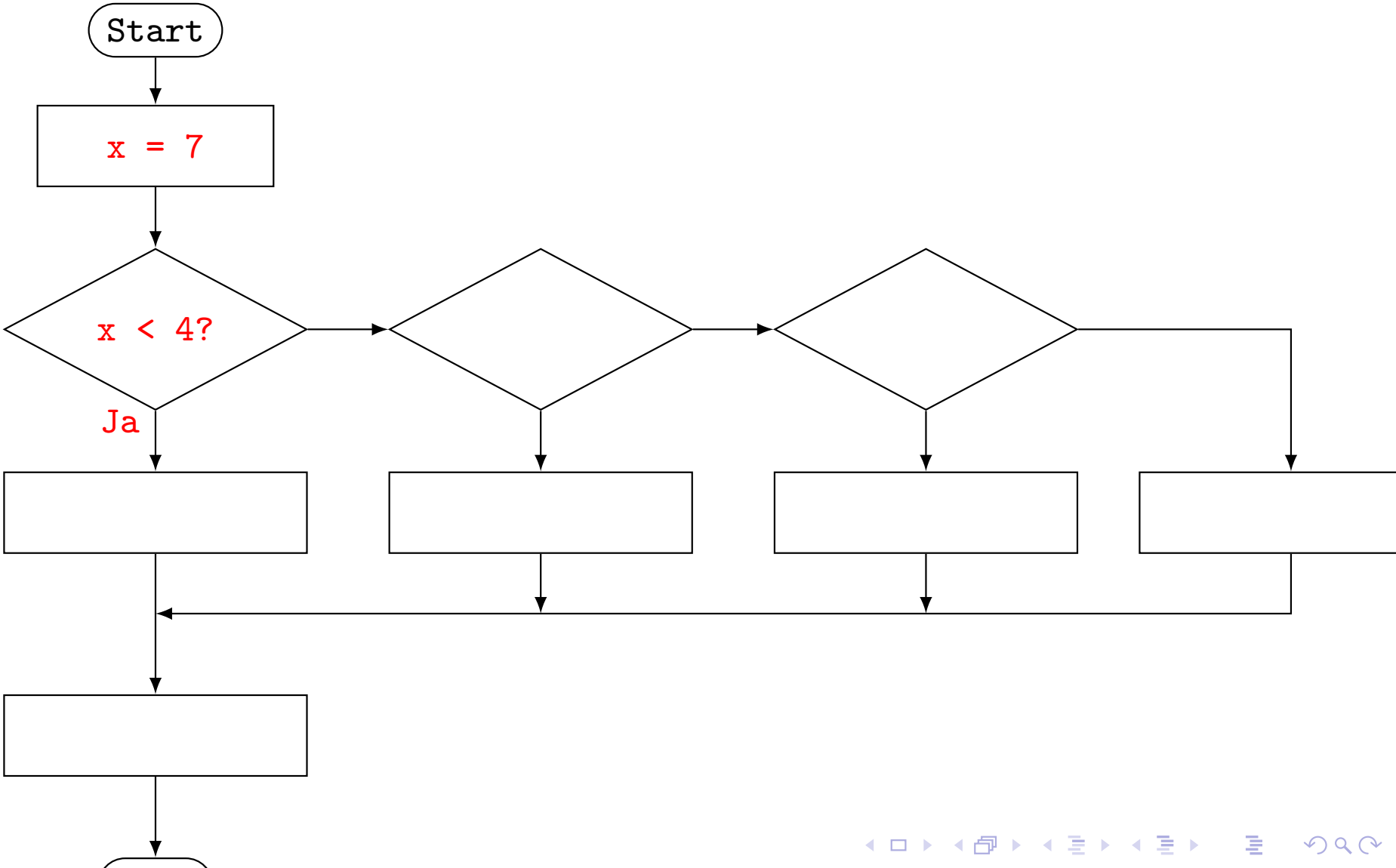
Darstellung als Flussdiagramm



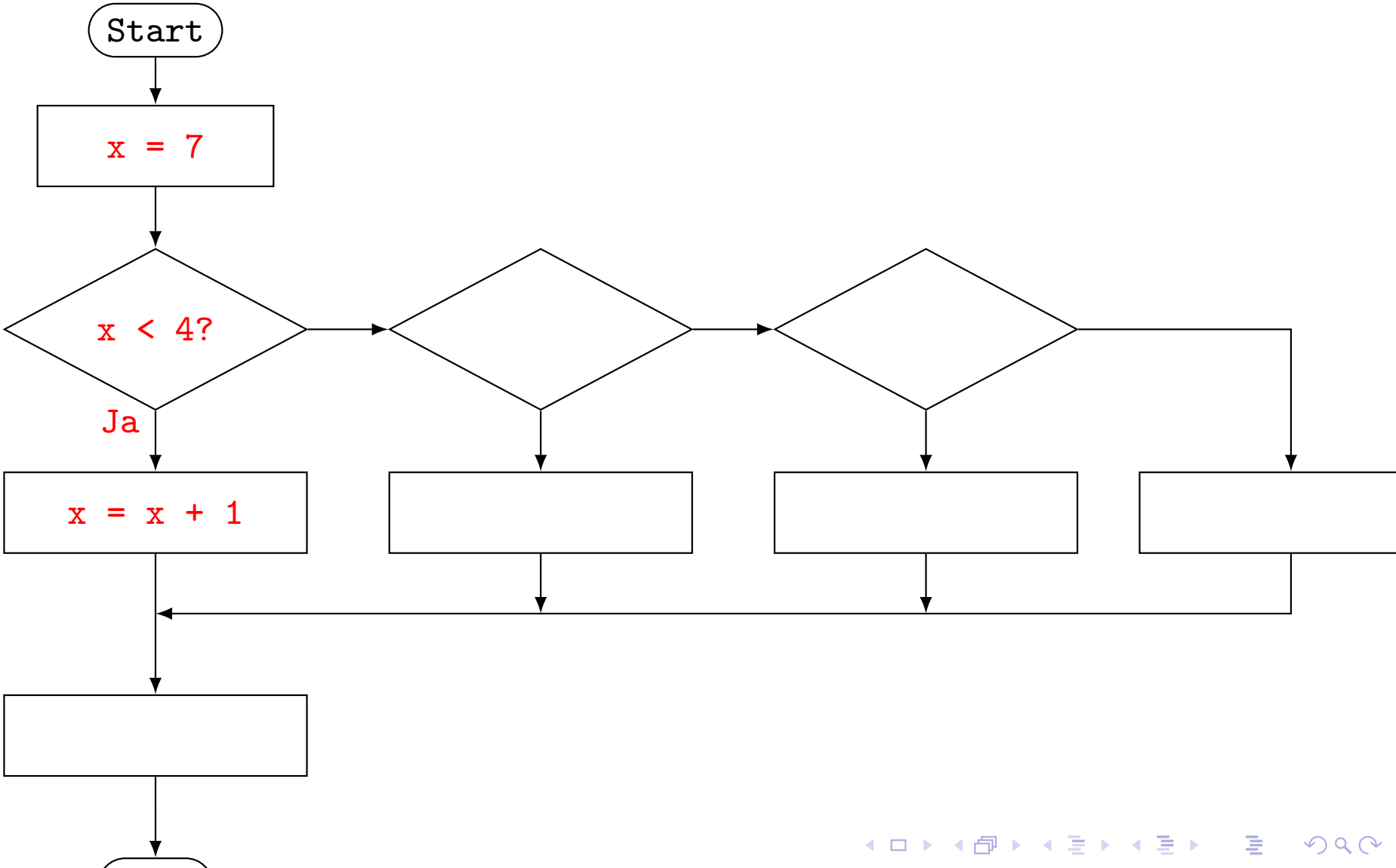
Darstellung als Flussdiagramm



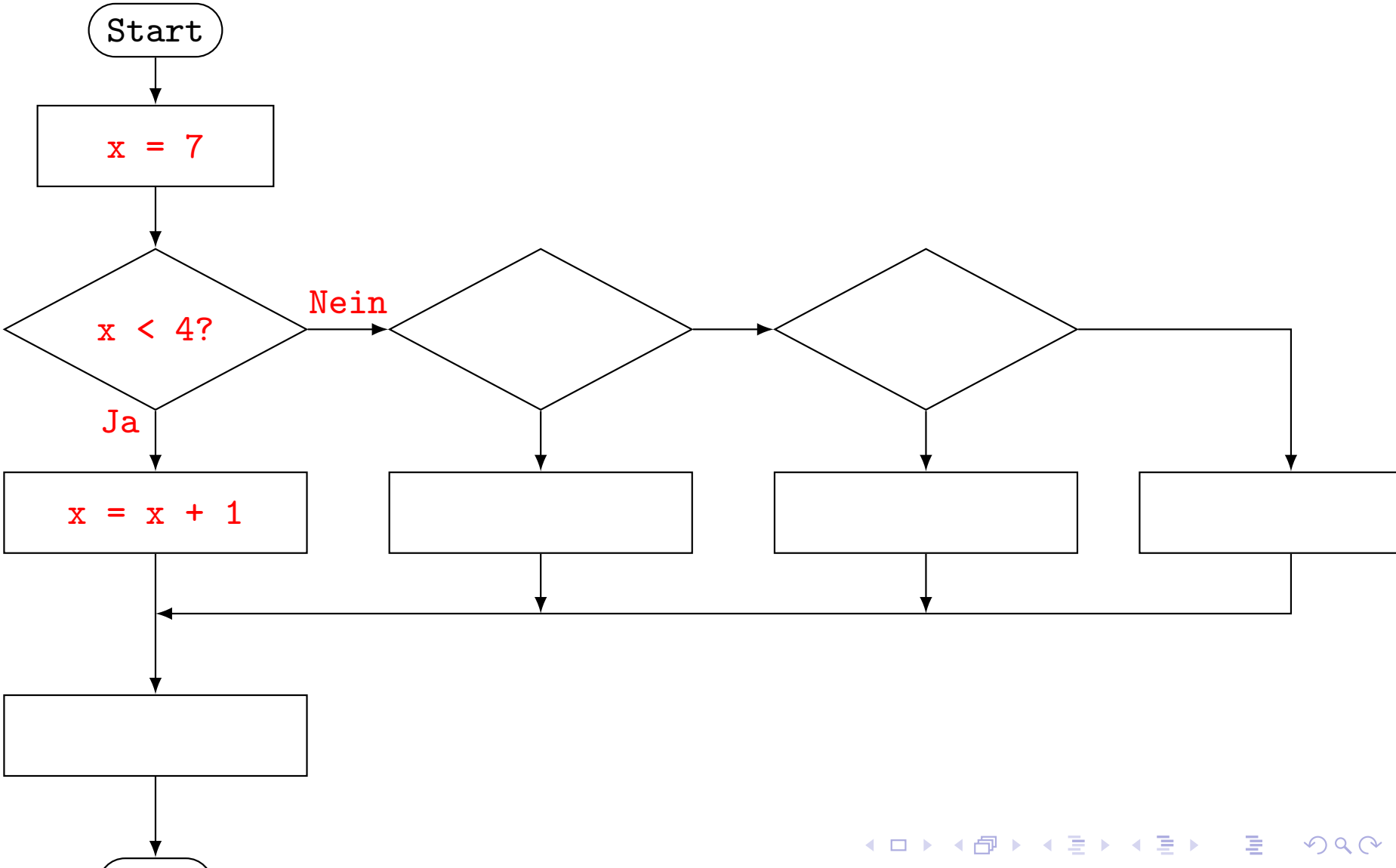
Darstellung als Flussdiagramm



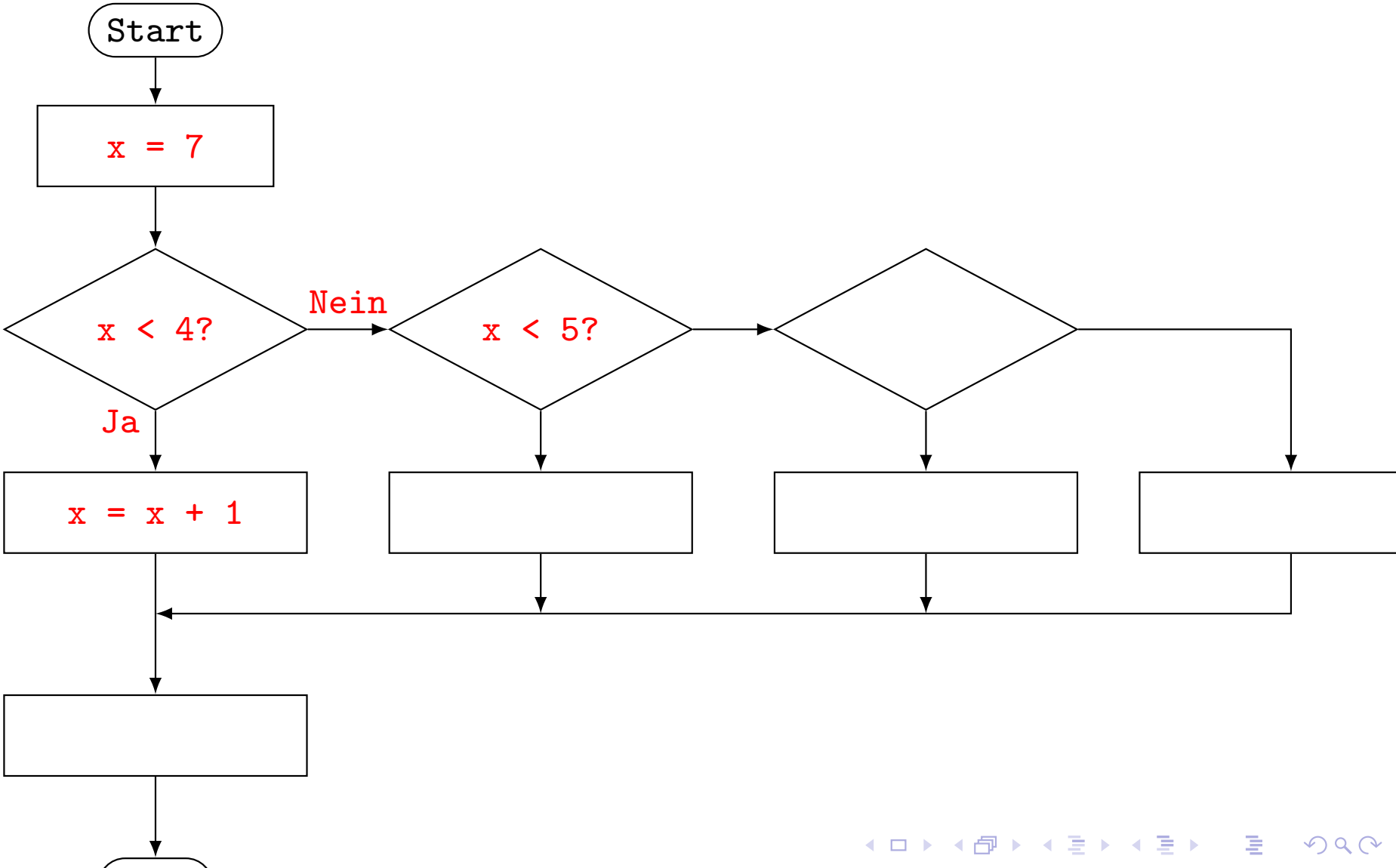
Darstellung als Flussdiagramm



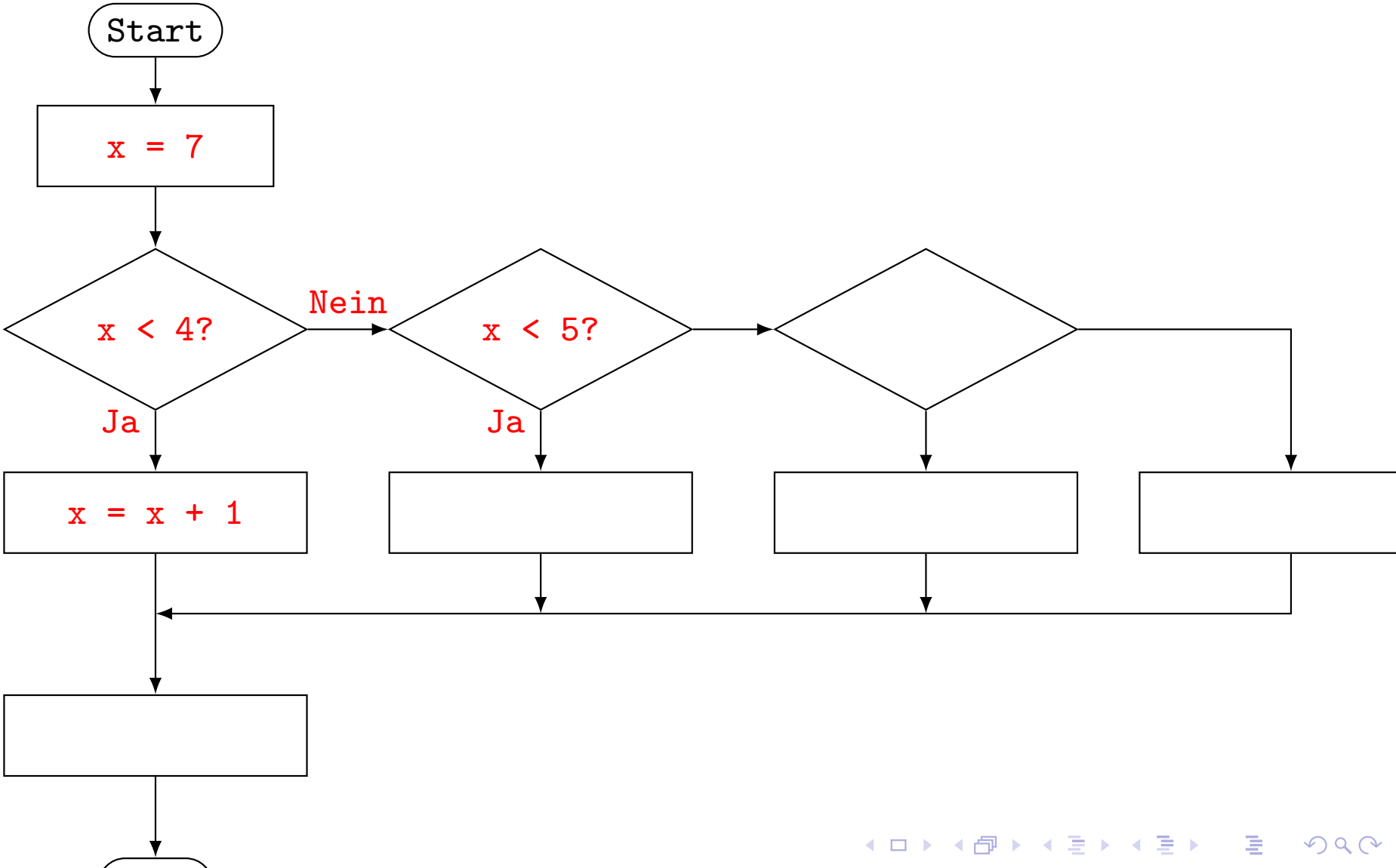
Darstellung als Flussdiagramm



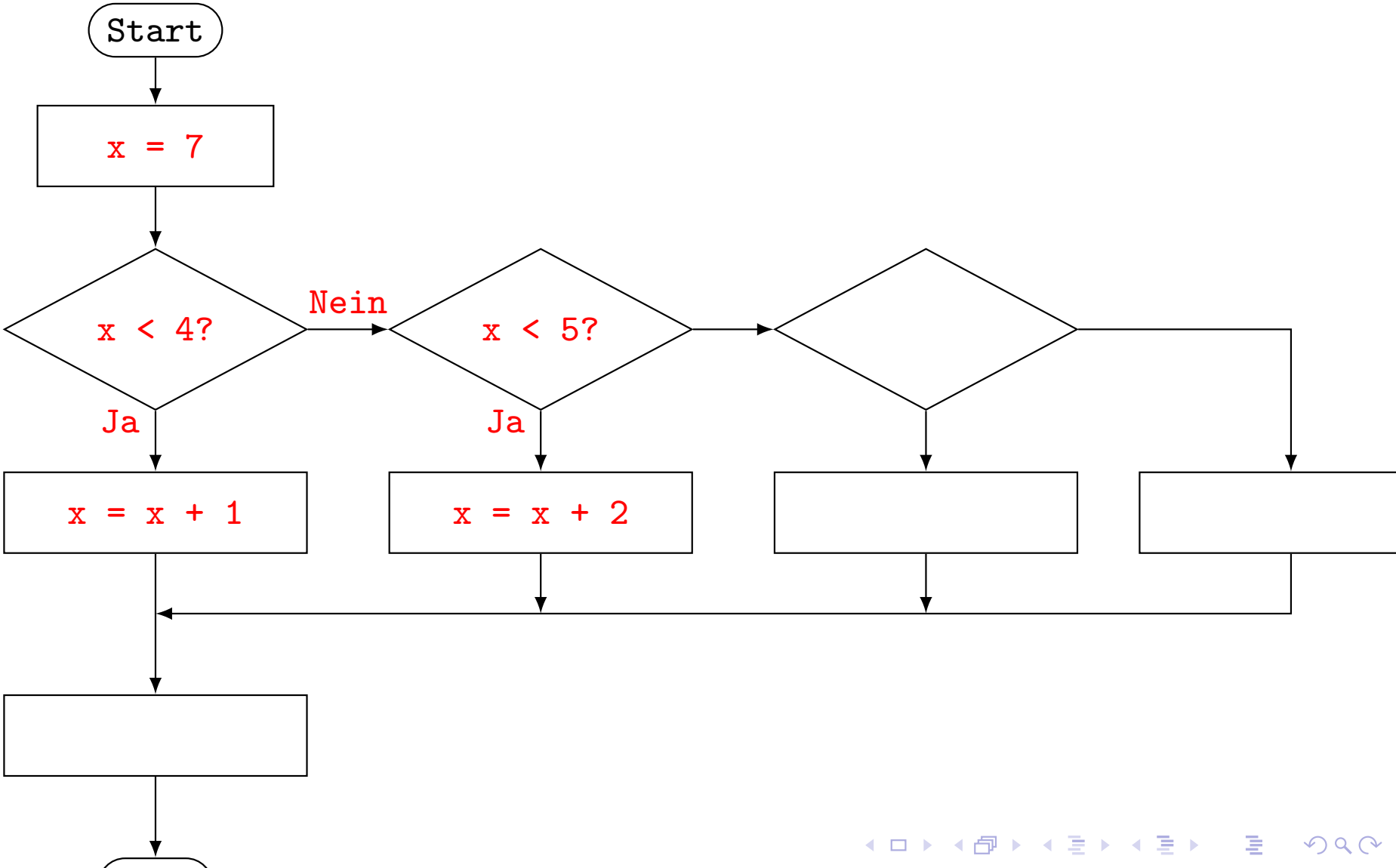
Darstellung als Flussdiagramm



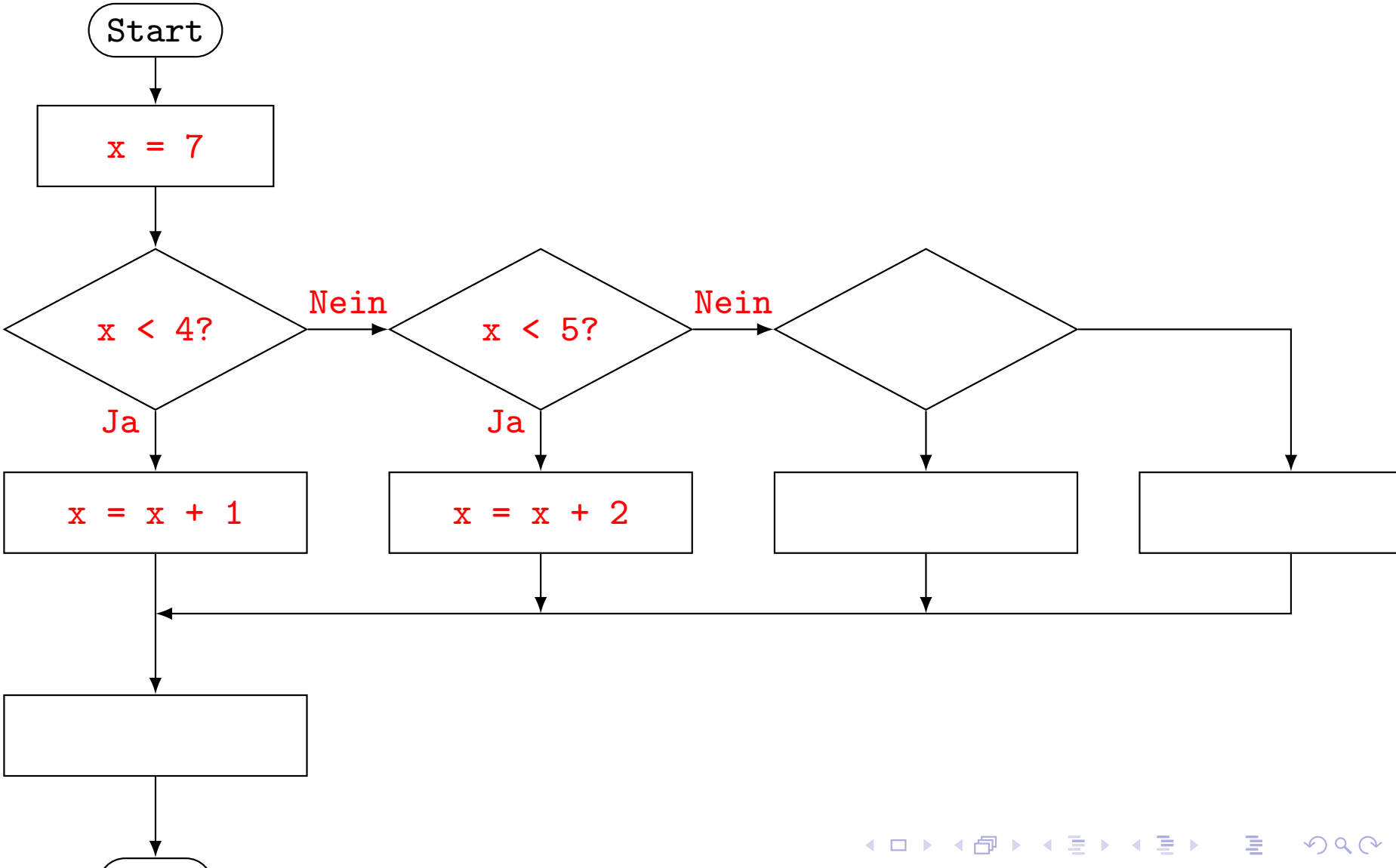
Darstellung als Flussdiagramm



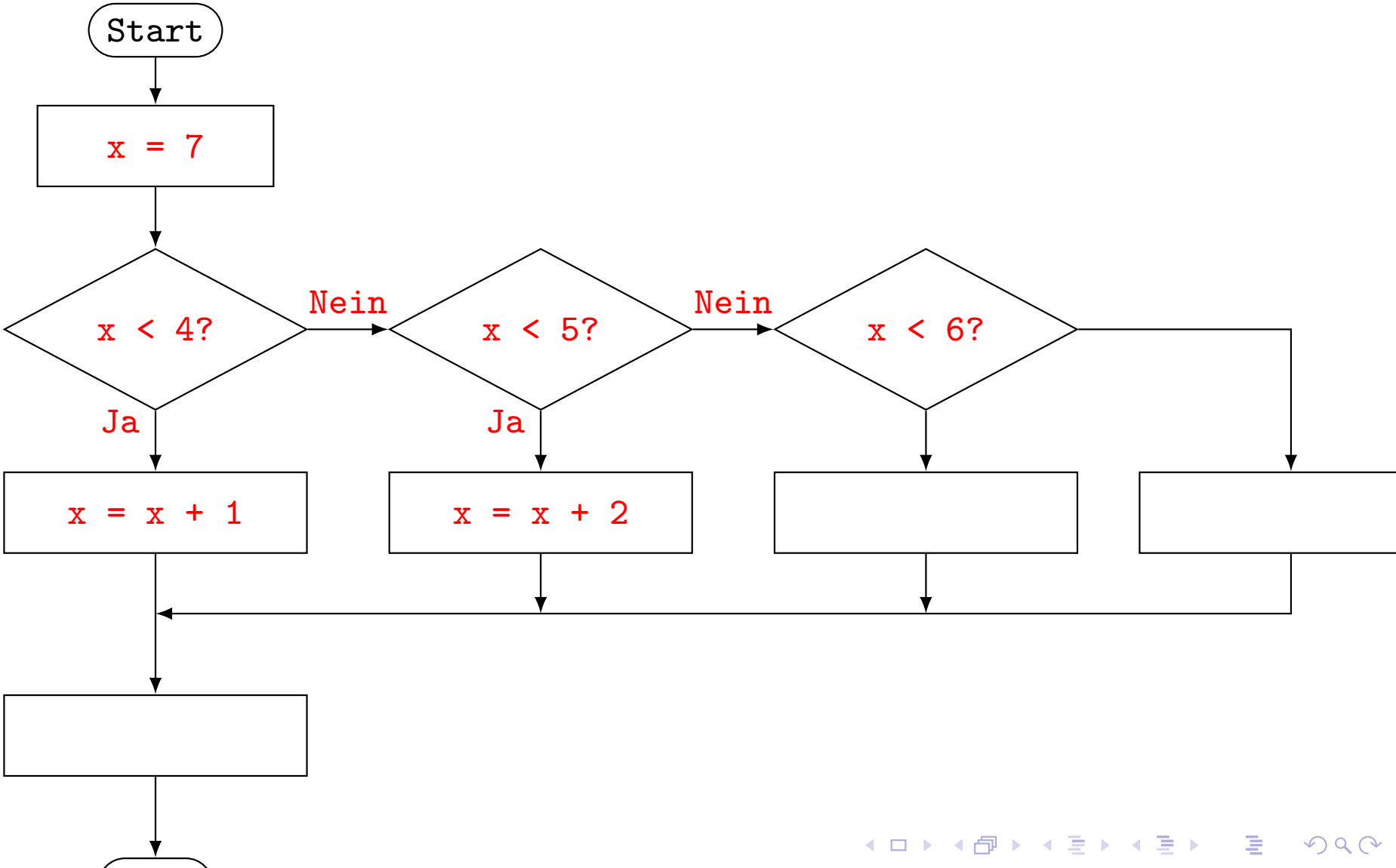
Darstellung als Flussdiagramm



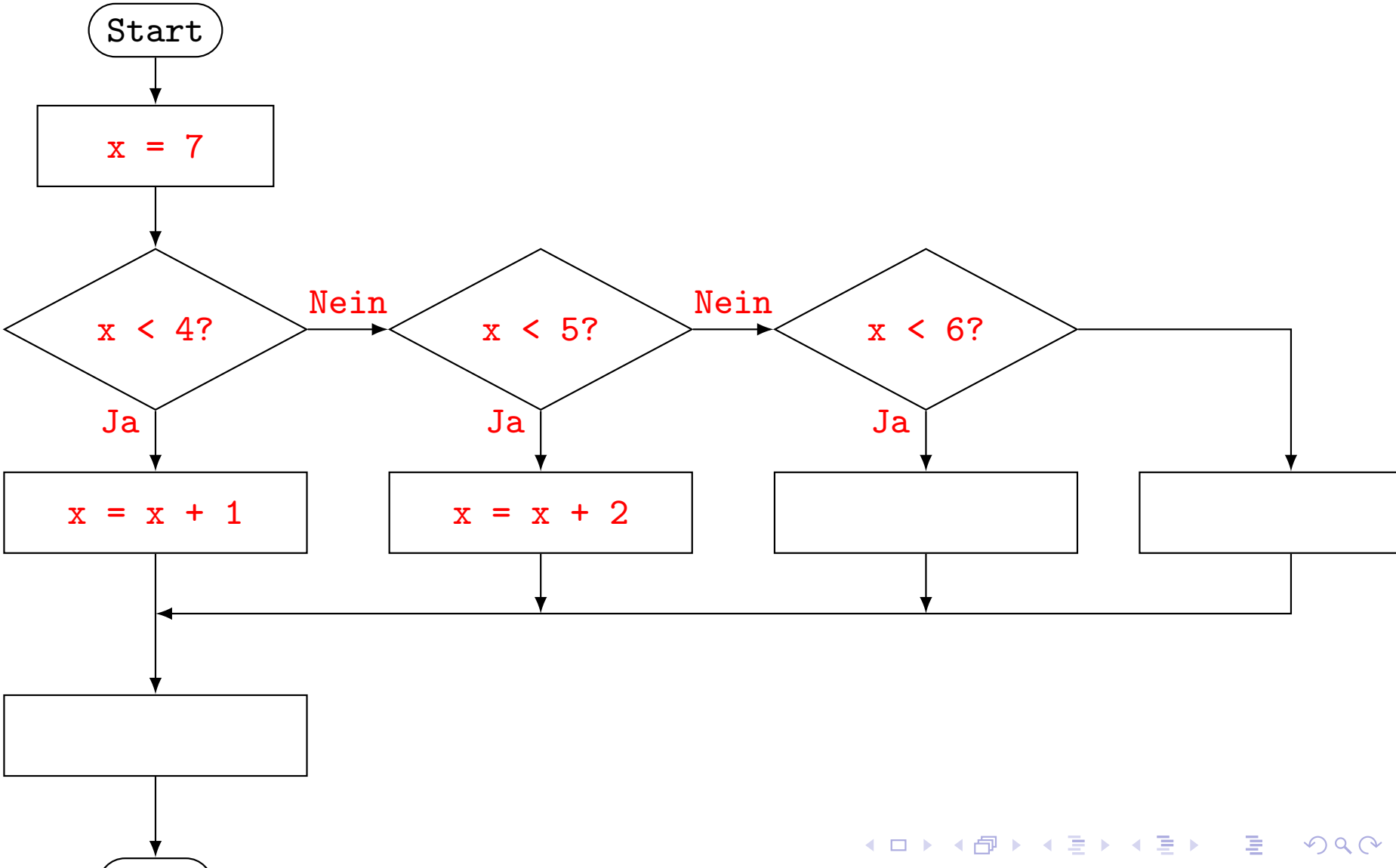
Darstellung als Flussdiagramm



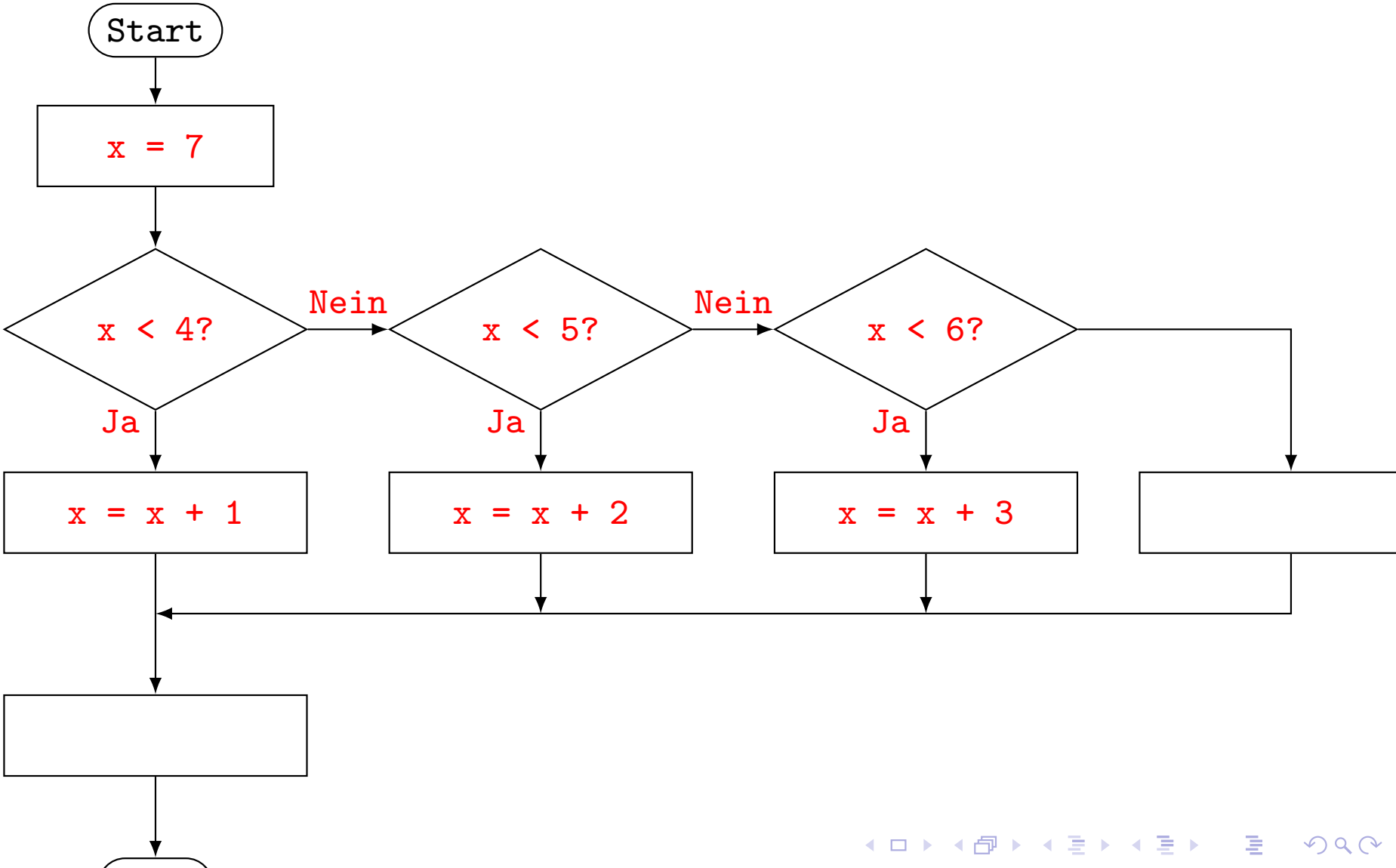
Darstellung als Flussdiagramm



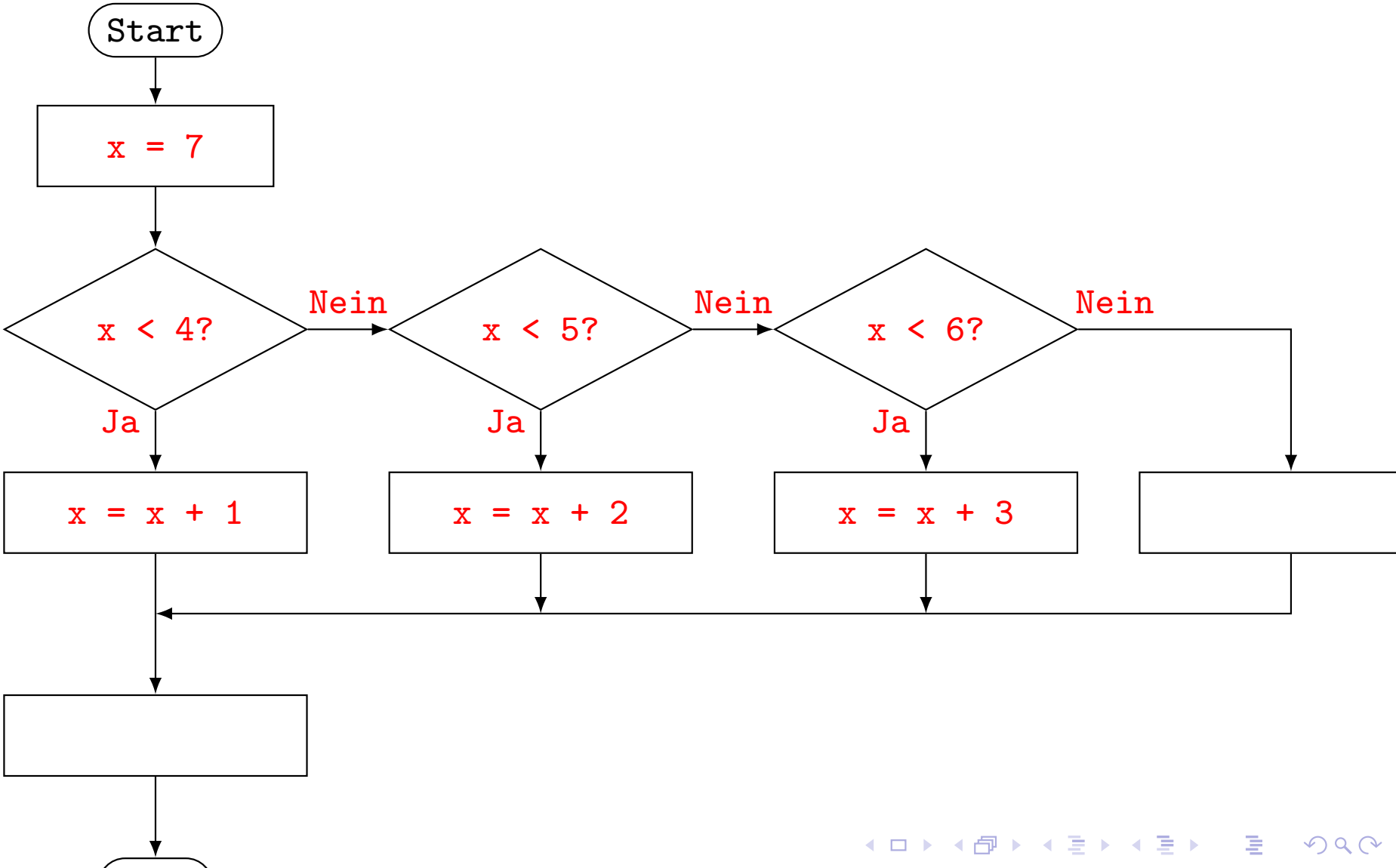
Darstellung als Flussdiagramm



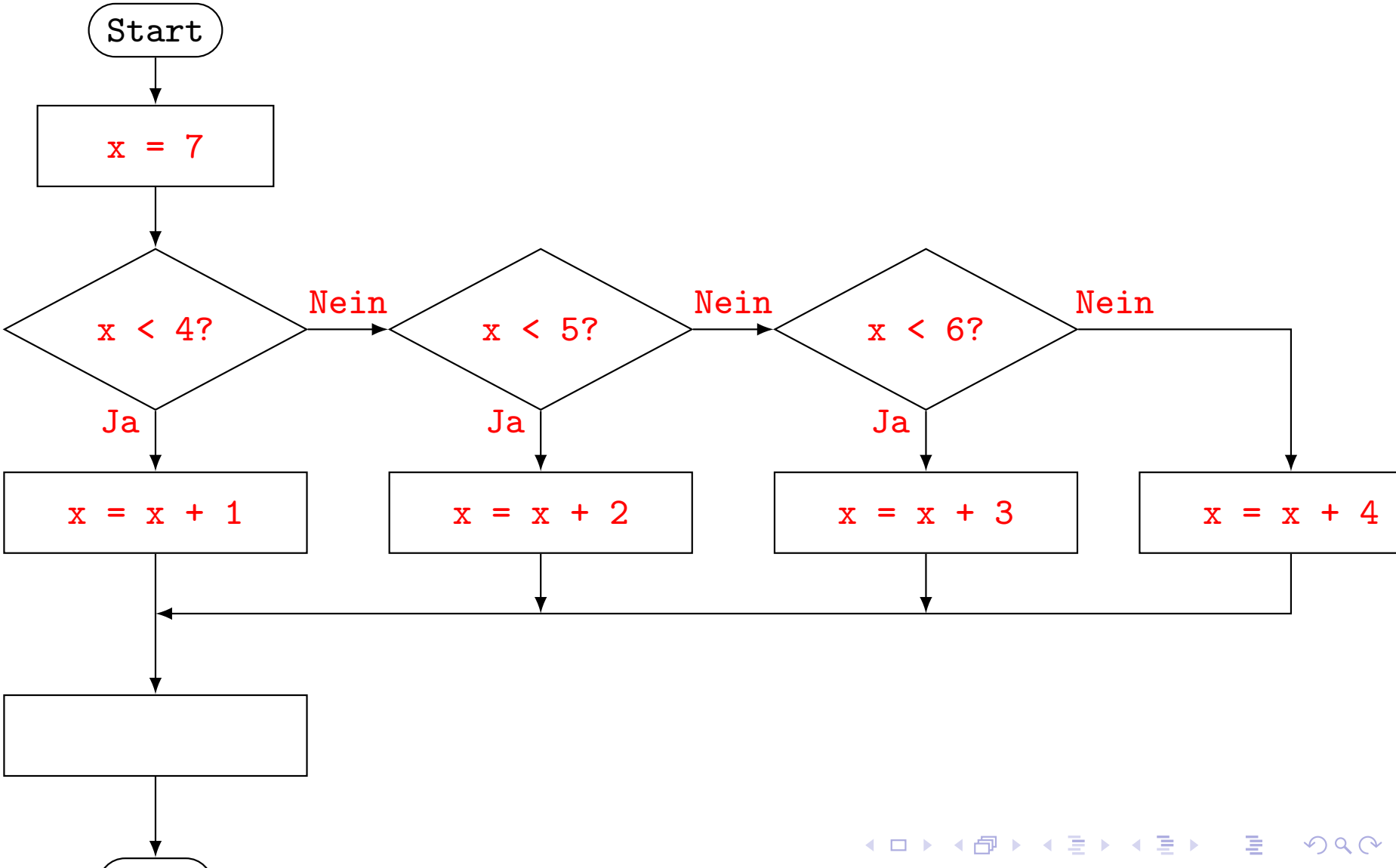
Darstellung als Flussdiagramm



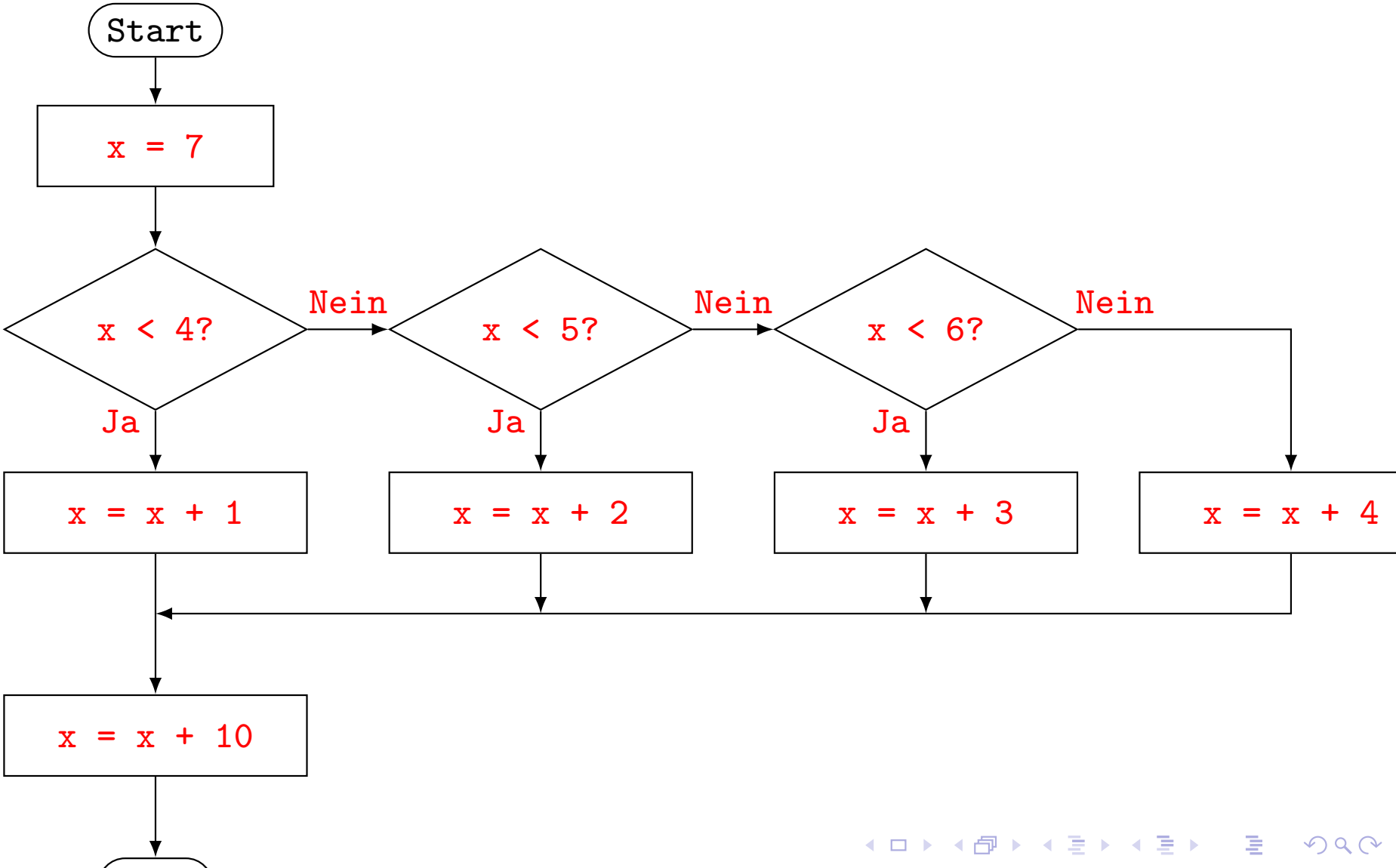
Darstellung als Flussdiagramm



Darstellung als Flussdiagramm



Darstellung als Flussdiagramm



Bemerkungen

- Bedingte Anweisungen und Verzweigungen können auch verschachtelt werden.
- Wie das letzte Beispiel zeigt, sind Flussdiagramme bei komplexeren Verzweigungsstrukturen nicht mehr besonders anschaulich.

Die zählergesteuerte `for`-Schleife

```
for var in range(start, stop[, step]):
    codeblock
...
```

var ist ein frei wählbarer Bezeichner, *start*, *stop* und *step* sind ganze Zahlen.

Ist die Schrittweite *step* positiv, so wird der Variablen *var* vor jedem neuen Schleifendurchlauf der Wert $start + k * step$ mit $k = 0, 1, 2, \ldots$ zugewiesen, sofern die Bedingung $start + k * step < stop$ erfüllt ist. Mit diesem Wert von *var* wird jeweils der *codeblock* ausgeführt. Für die Schrittweite $step = 1$ ist die Angabe von *step* optional. Sobald die oben genannte Bedingung nicht mehr erfüllt ist, wird das Programm nach dem *codeblock* fortgesetzt.

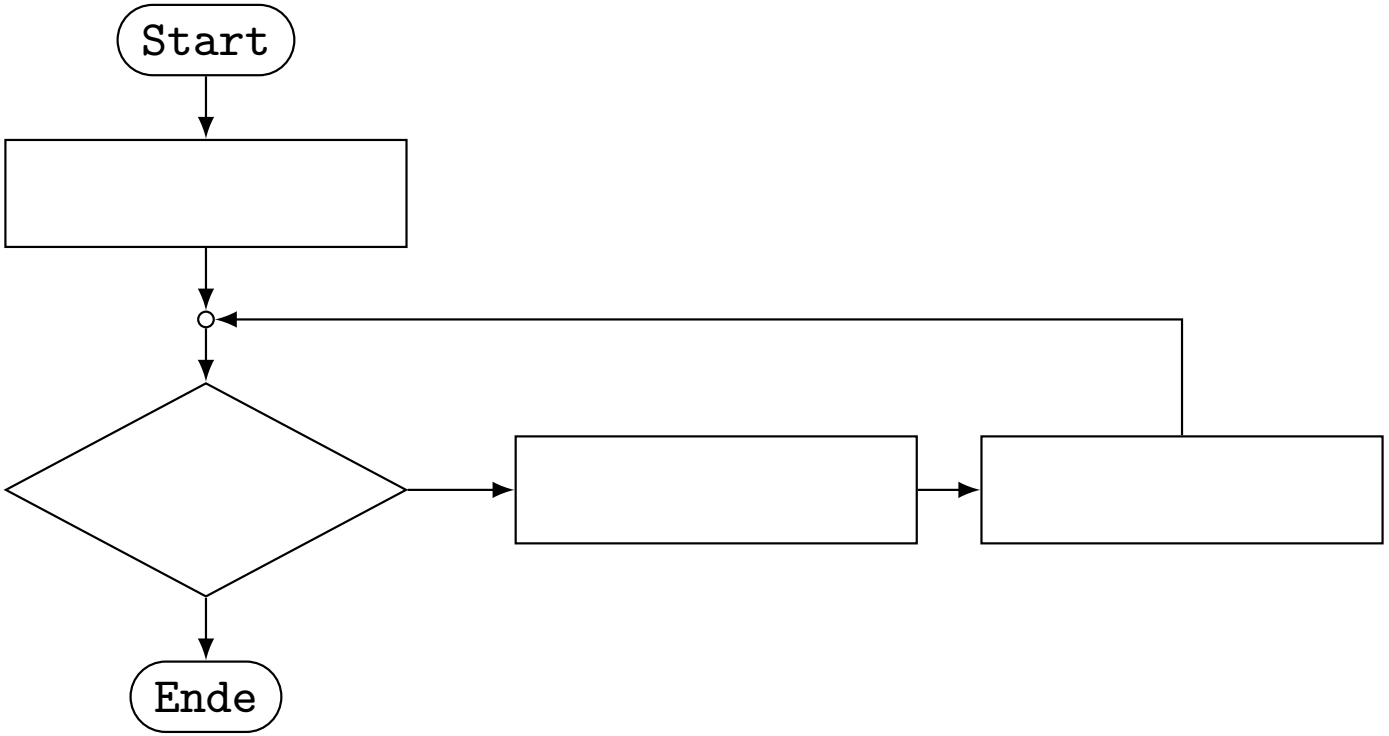
Beispiel 5.1

```
1  for i in range(3, 11, 2):   # Ausgaben:
2      print(i**2)             #
3                              #
4                              #
```

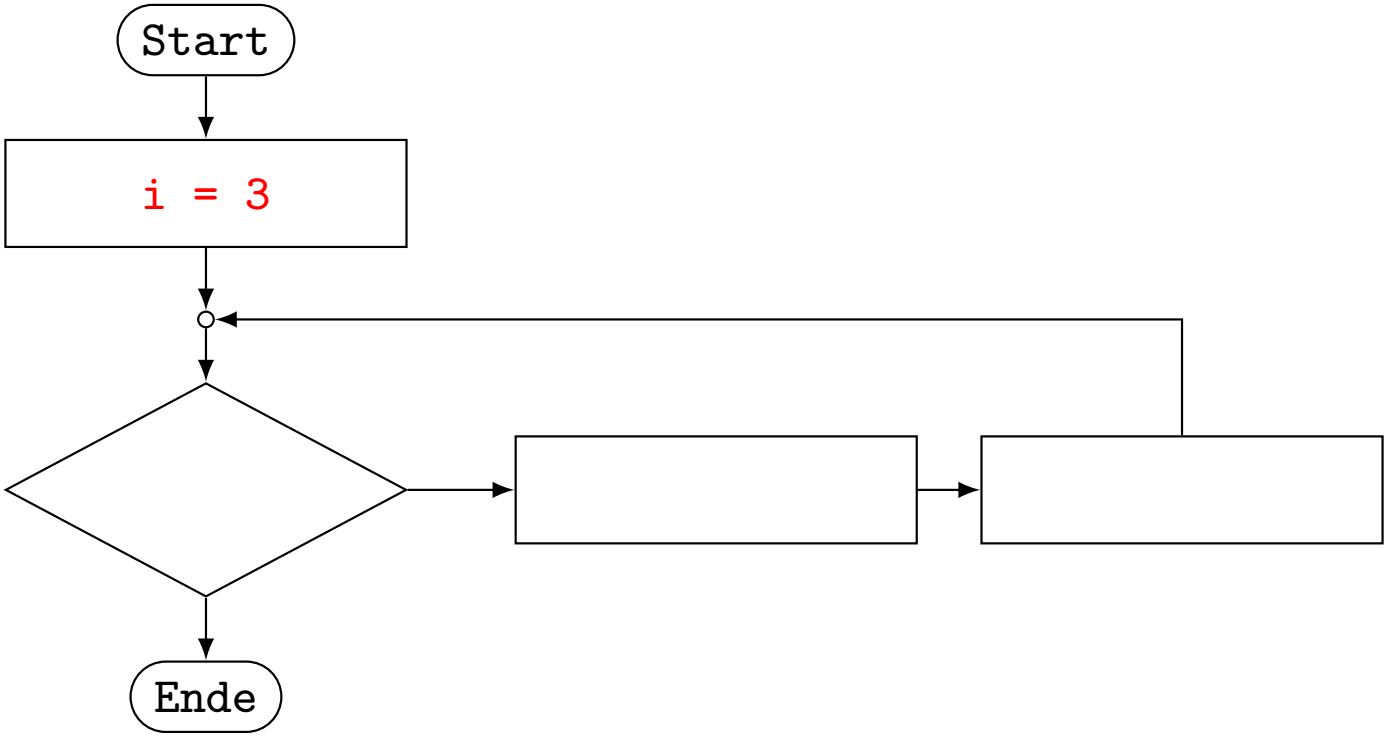
Beispiel 5.1

```
1  for i in range(3, 11, 2):   # Ausgaben:  9
2      print(i**2)             #          25
3                              #          49
4                              #          81
```

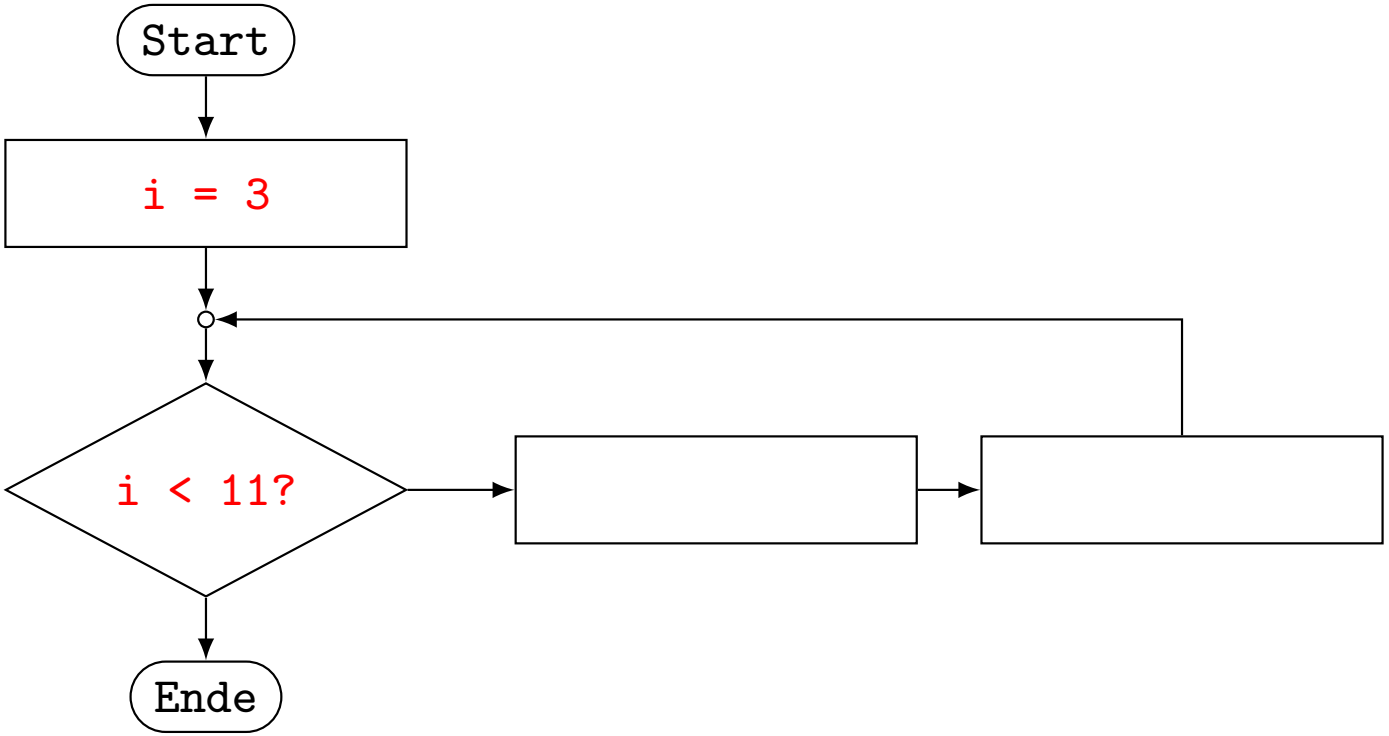
Darstellung als Flussdiagramm



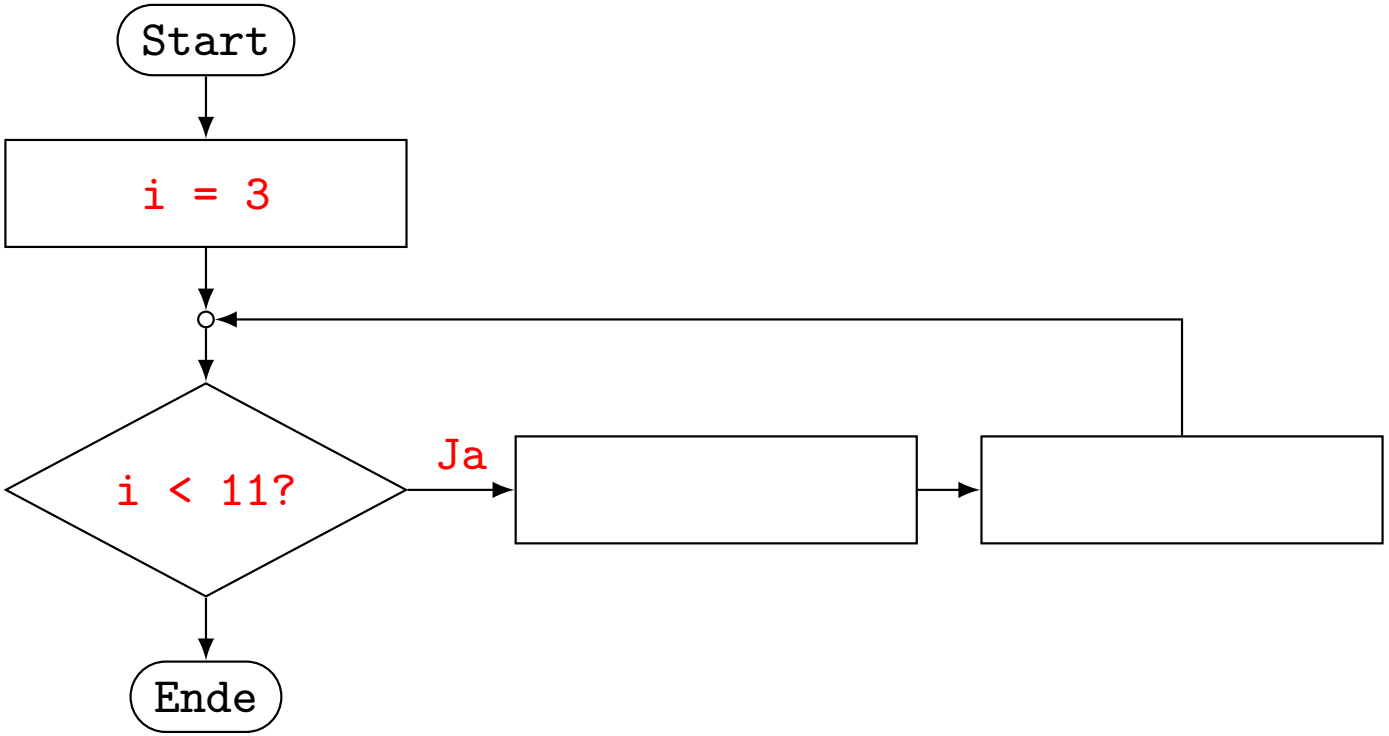
Darstellung als Flussdiagramm



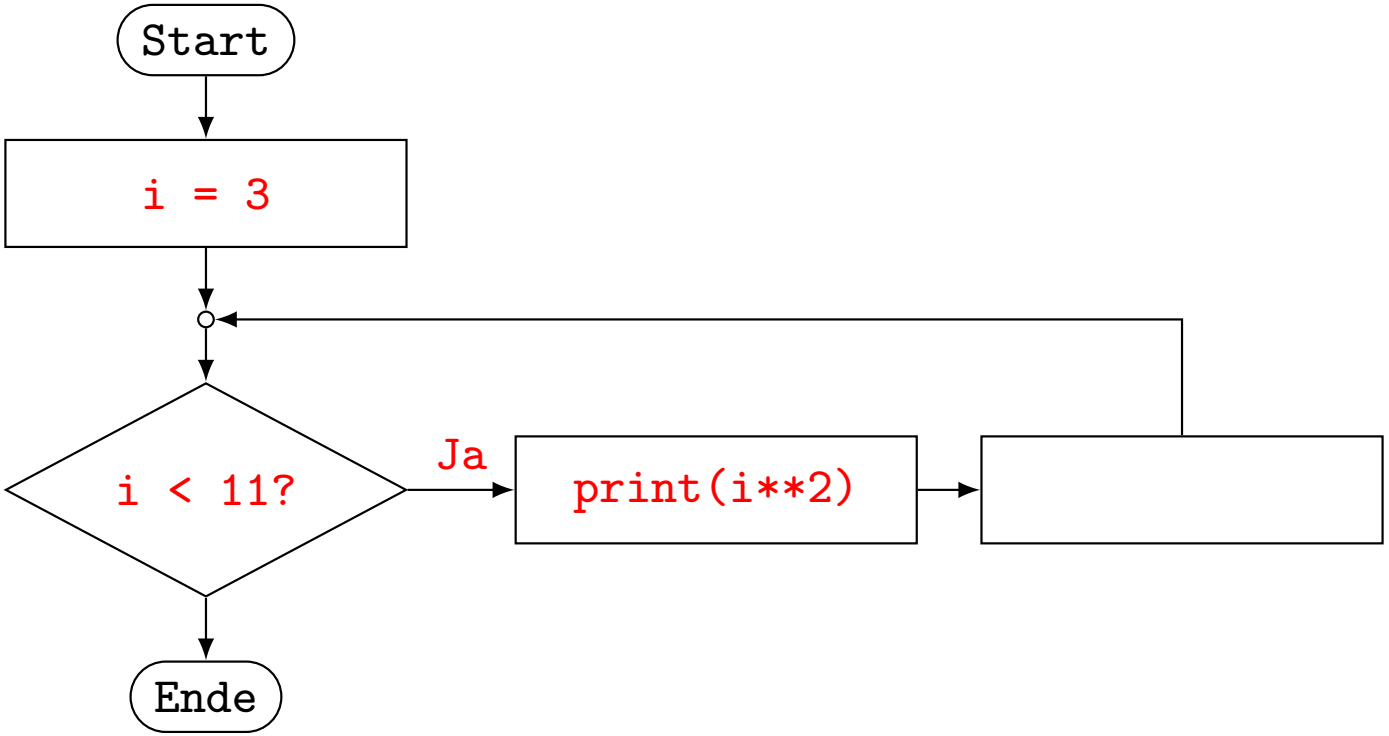
Darstellung als Flussdiagramm



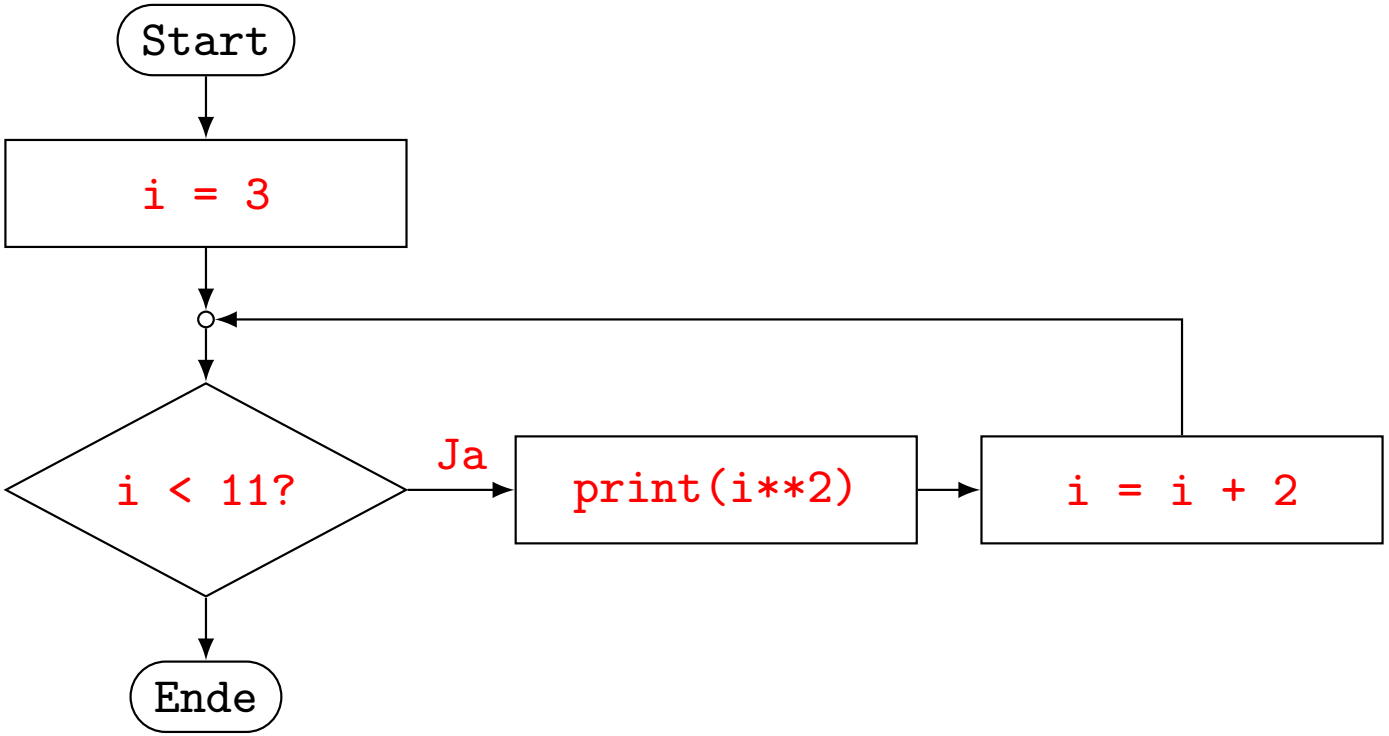
Darstellung als Flussdiagramm



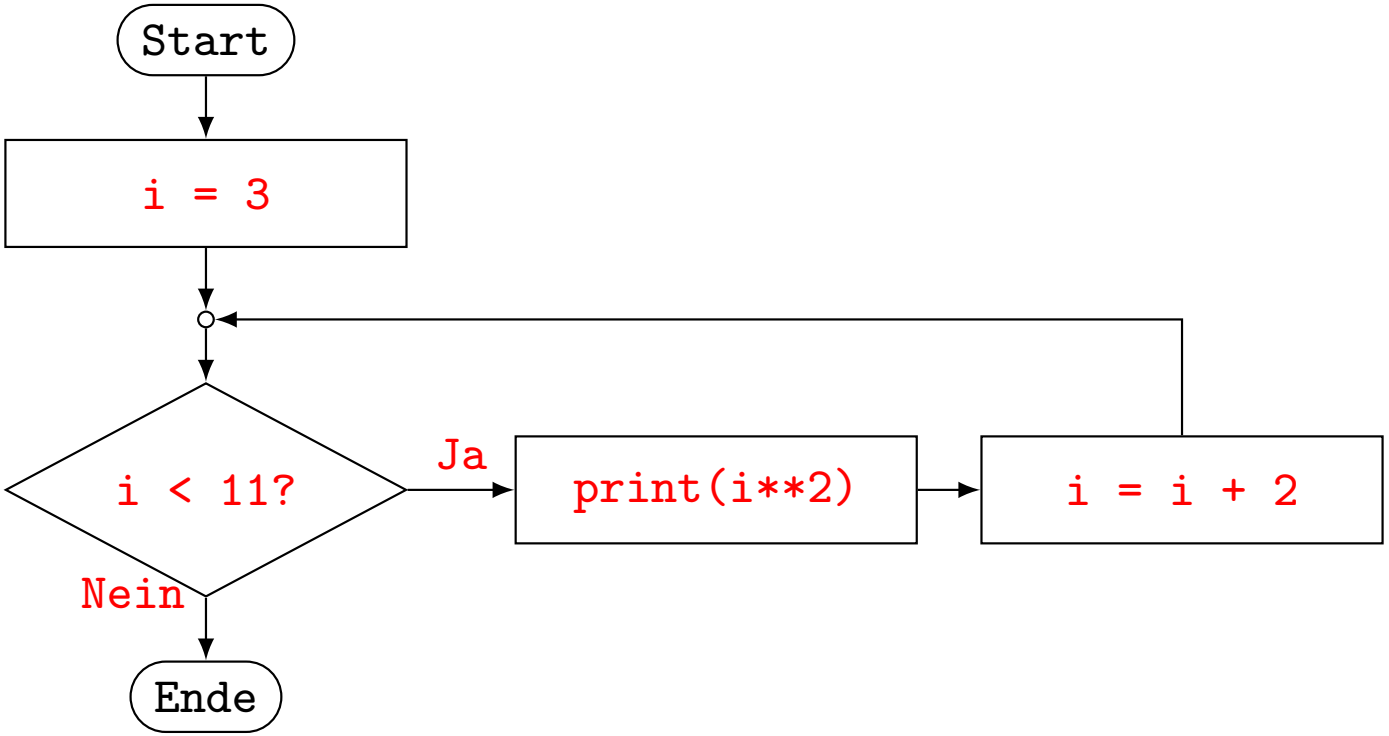
Darstellung als Flussdiagramm



Darstellung als Flussdiagramm



Darstellung als Flussdiagramm



Die `while`-Schleife

```
while bedingung:
    codeblock
...
```

Bei der `while`-Schleife wird der *`codeblock`* so lange ausgeführt, wie der Wert von *`bedingung`* wahr ist. Sobald *`bedingung`* falsch ist, wird das Programm nach dem *`codeblock`* fortgesetzt.

Hier müssen wir uns selber darum kümmern, dass beim Erreichen des gewünschten Zustands die Bedingung falsch wird, damit die Schleife terminiert. Andernfalls entsteht eine Endlosschleife, die in der Shell mit der Tastenkombination `Ctrl-C` unterbrochen werden muss.

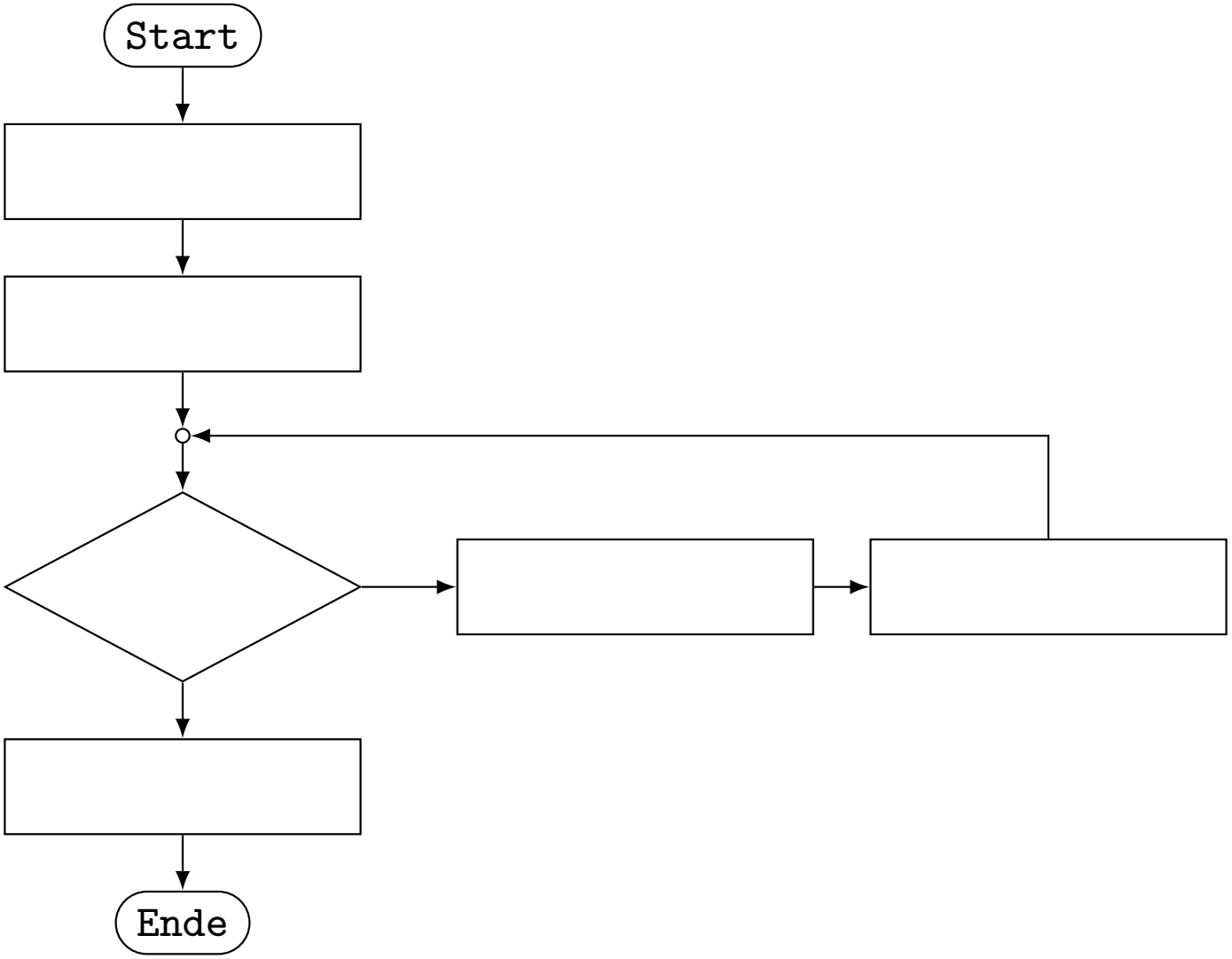
Beispiel 5.2

```
1  s = 0              #    s |  k  | s < 10
2  k = 1              # -----+-----+--------
3  while s < 10:      #      |     |
4      s += k         #      |     |
5      k += 2         #      |     |
6  print(s)           #      |     |
7                     #      |     |
8                     #
9                     # Ausgabe:
```

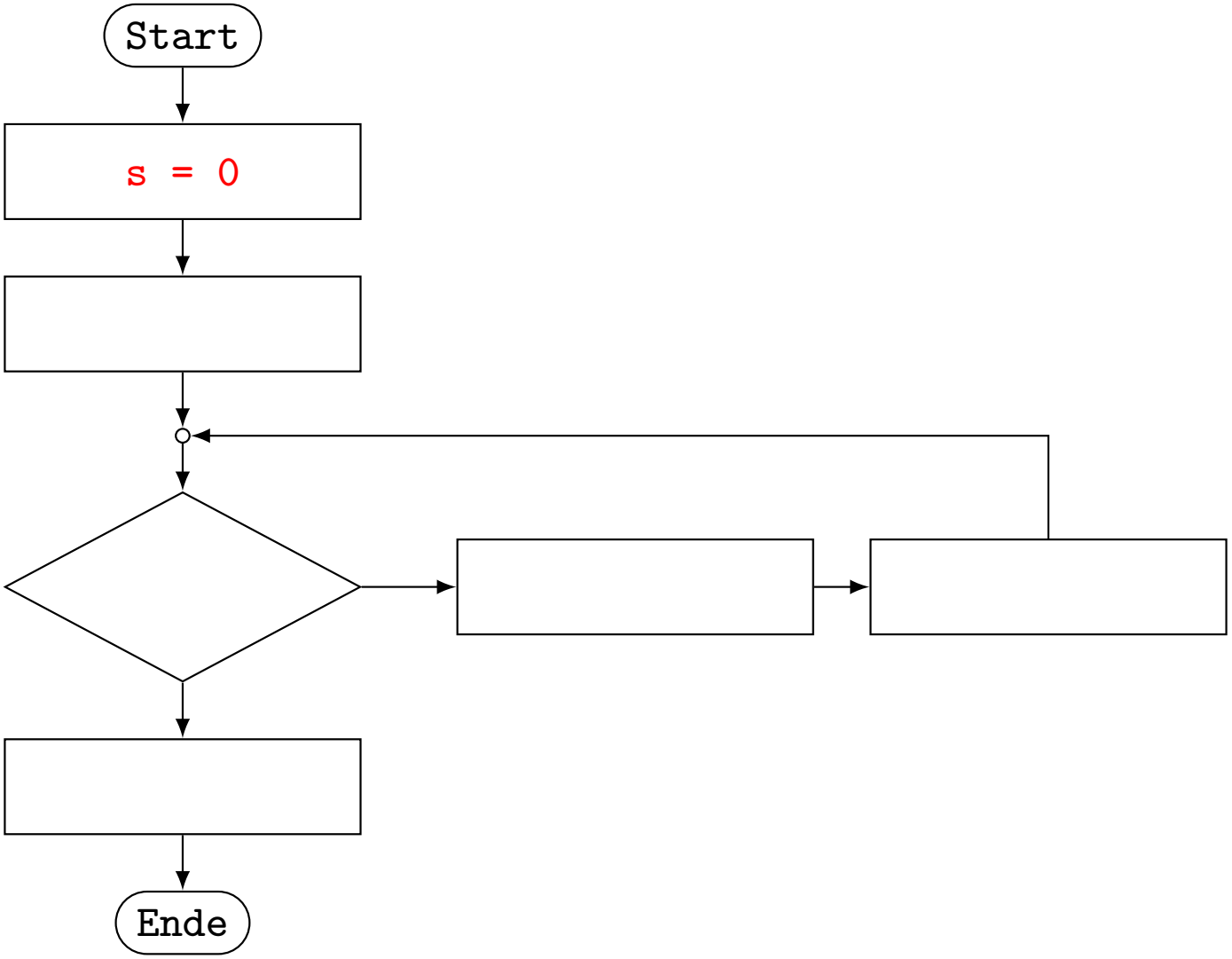
Beispiel 5.2

```
1  s = 0            #    s |  k  | s < 10
2  k = 1            # -----+-----+--------
3  while s < 10:    #    0 |  1  | True
4      s += k       #    1 |  3  | True
5      k += 2       #    4 |  5  | True
6  print(s)         #    9 |  7  | True
7                   #   16 |  9  | False
8                   #
9                   # Ausgabe: 16
```

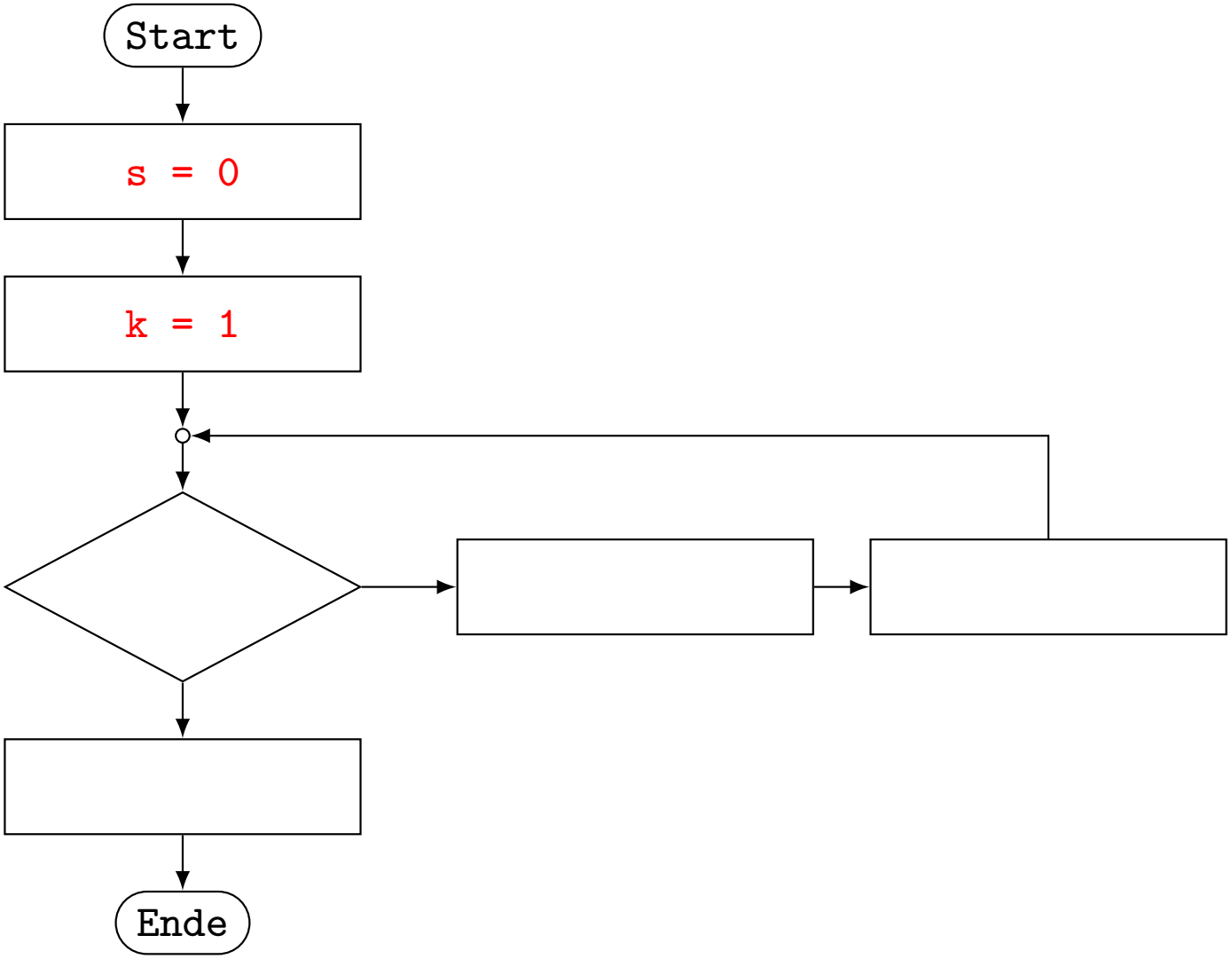
Darstellung als Flussdiagramm



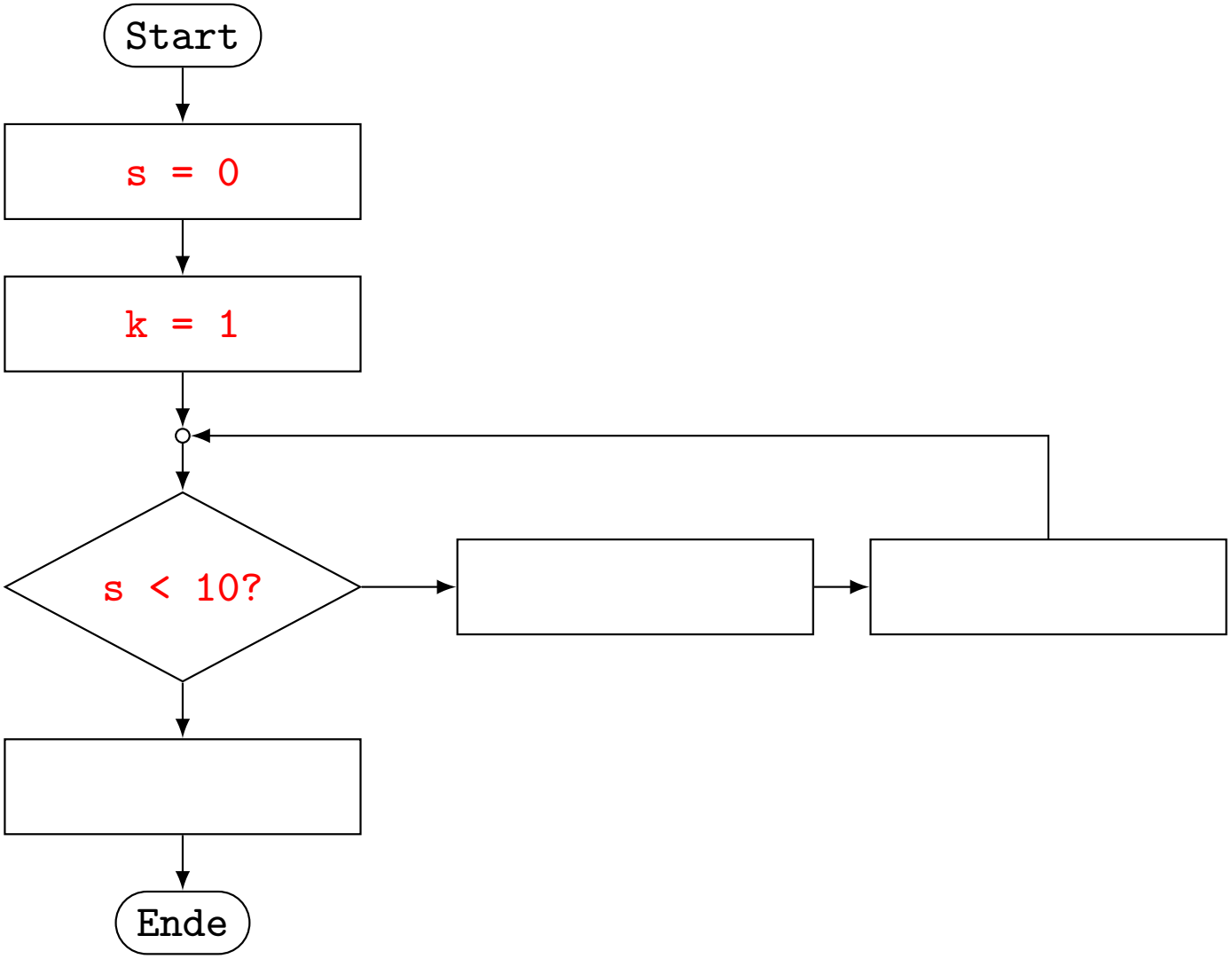
Darstellung als Flussdiagramm



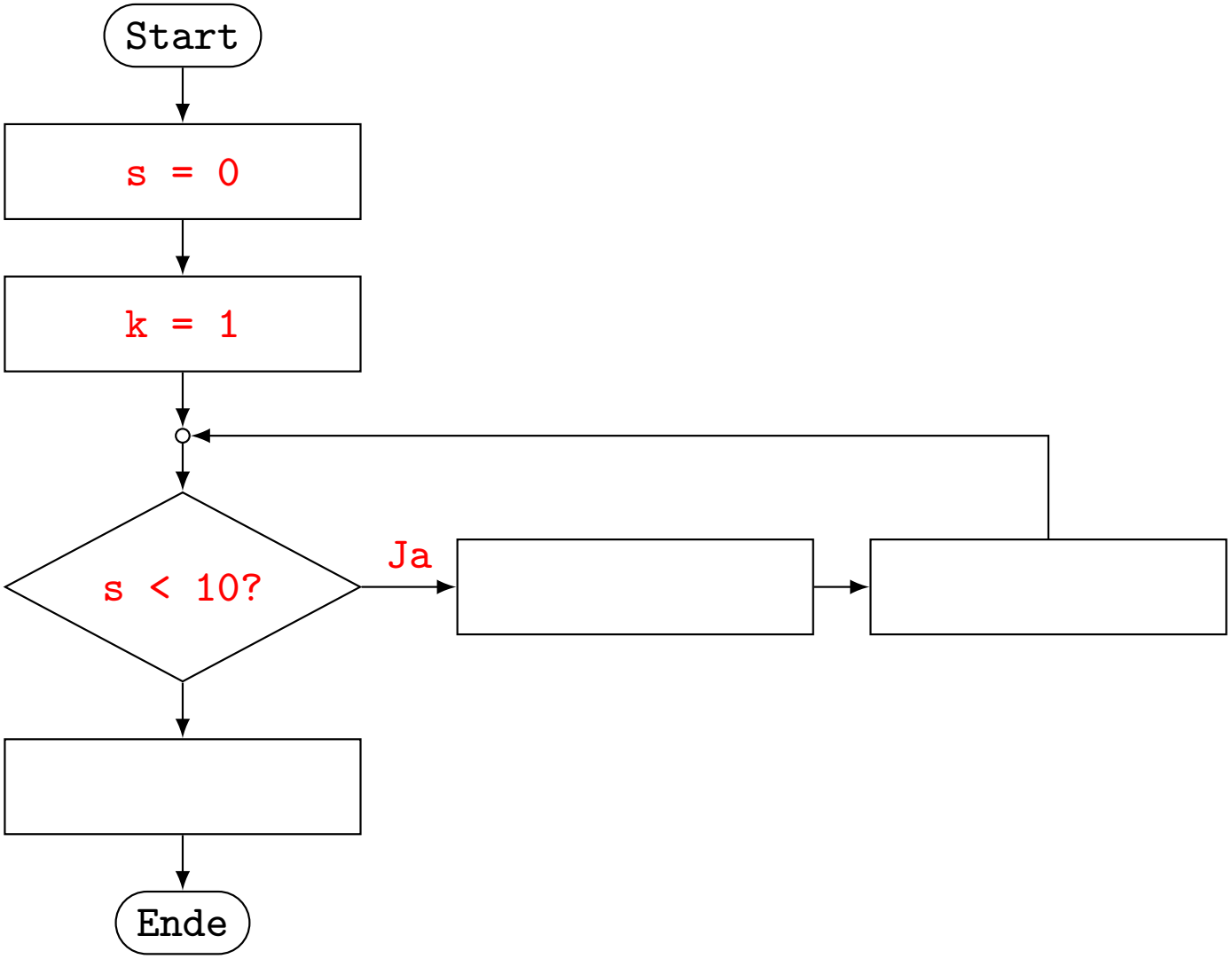
Darstellung als Flussdiagramm



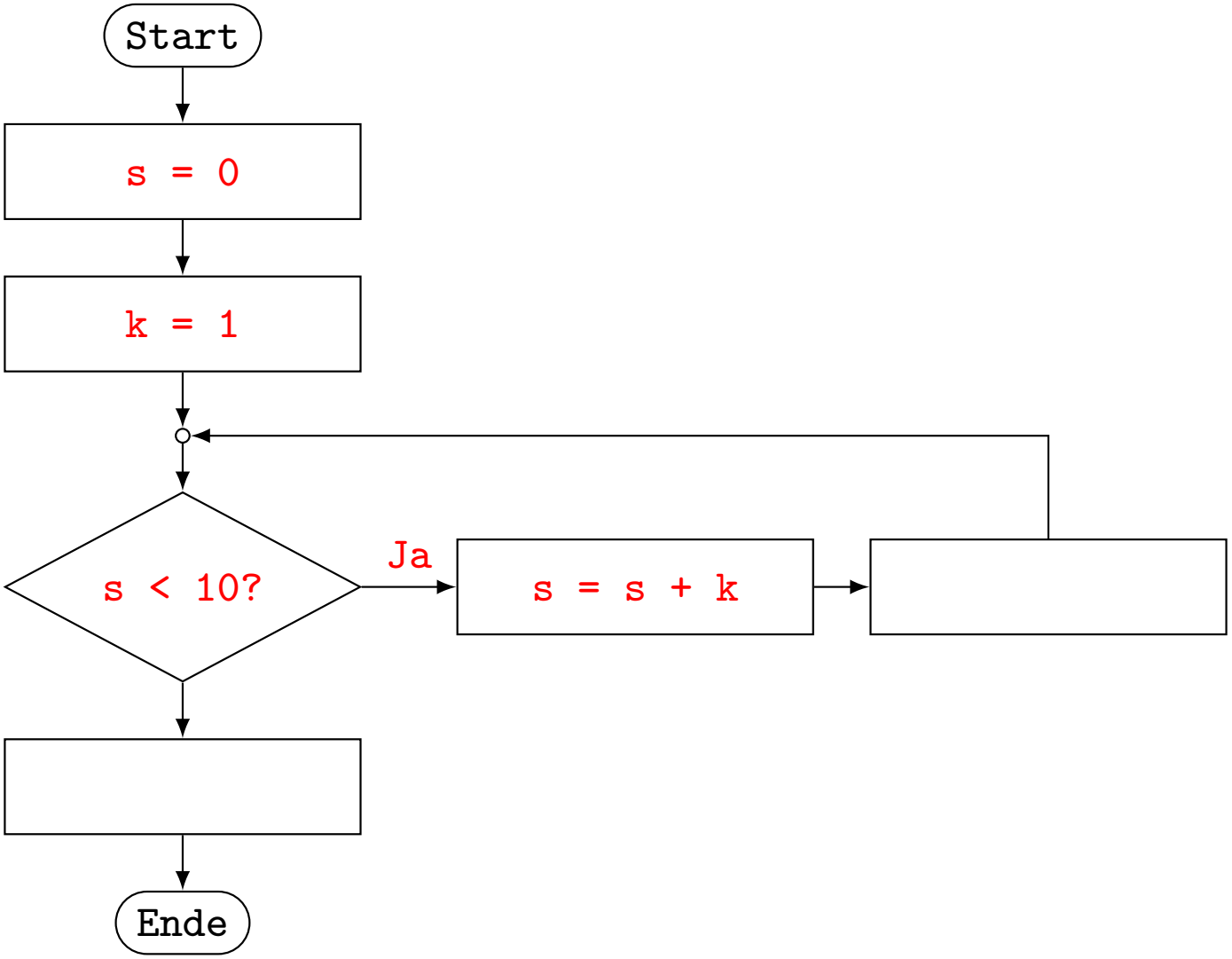
Darstellung als Flussdiagramm



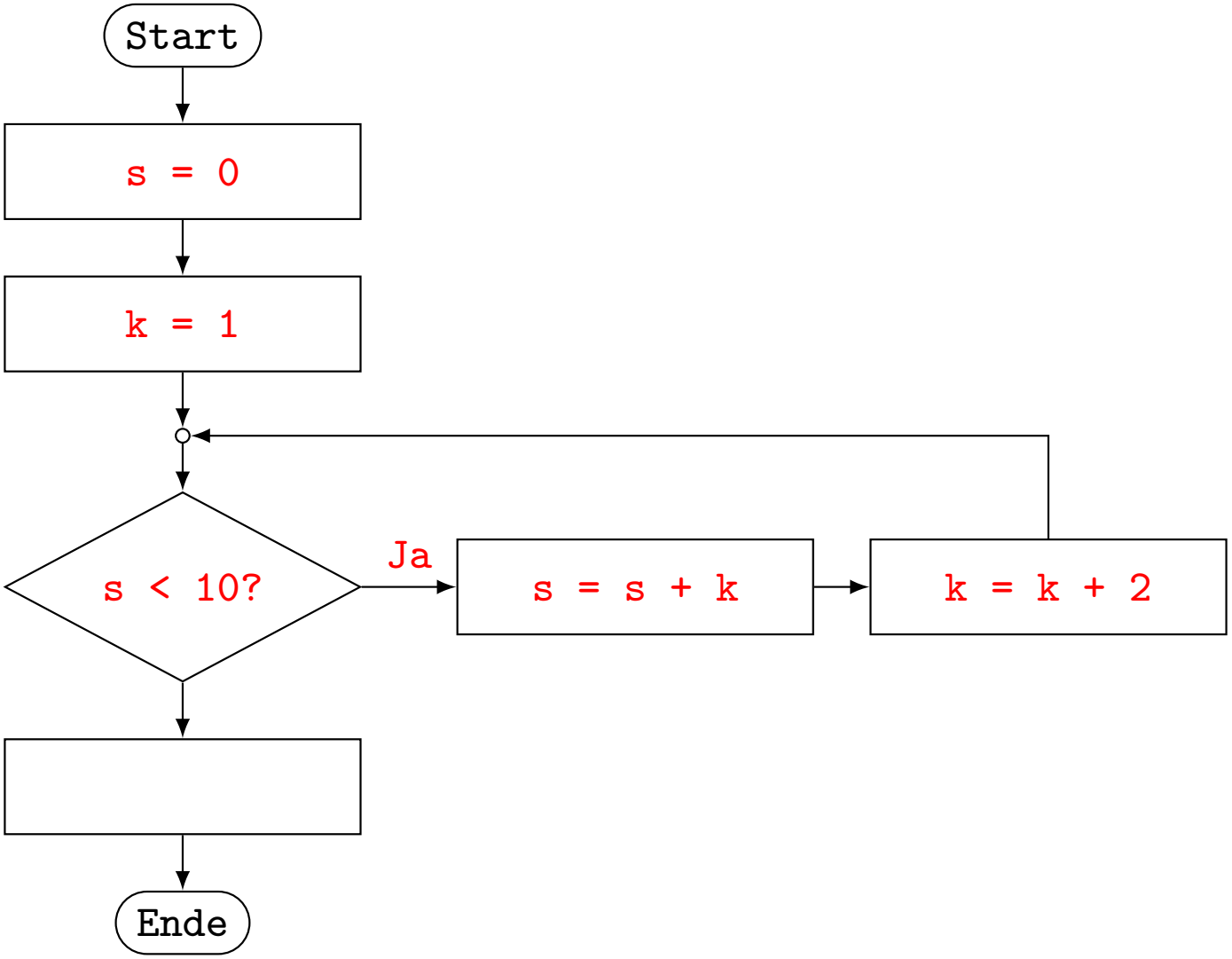
Darstellung als Flussdiagramm



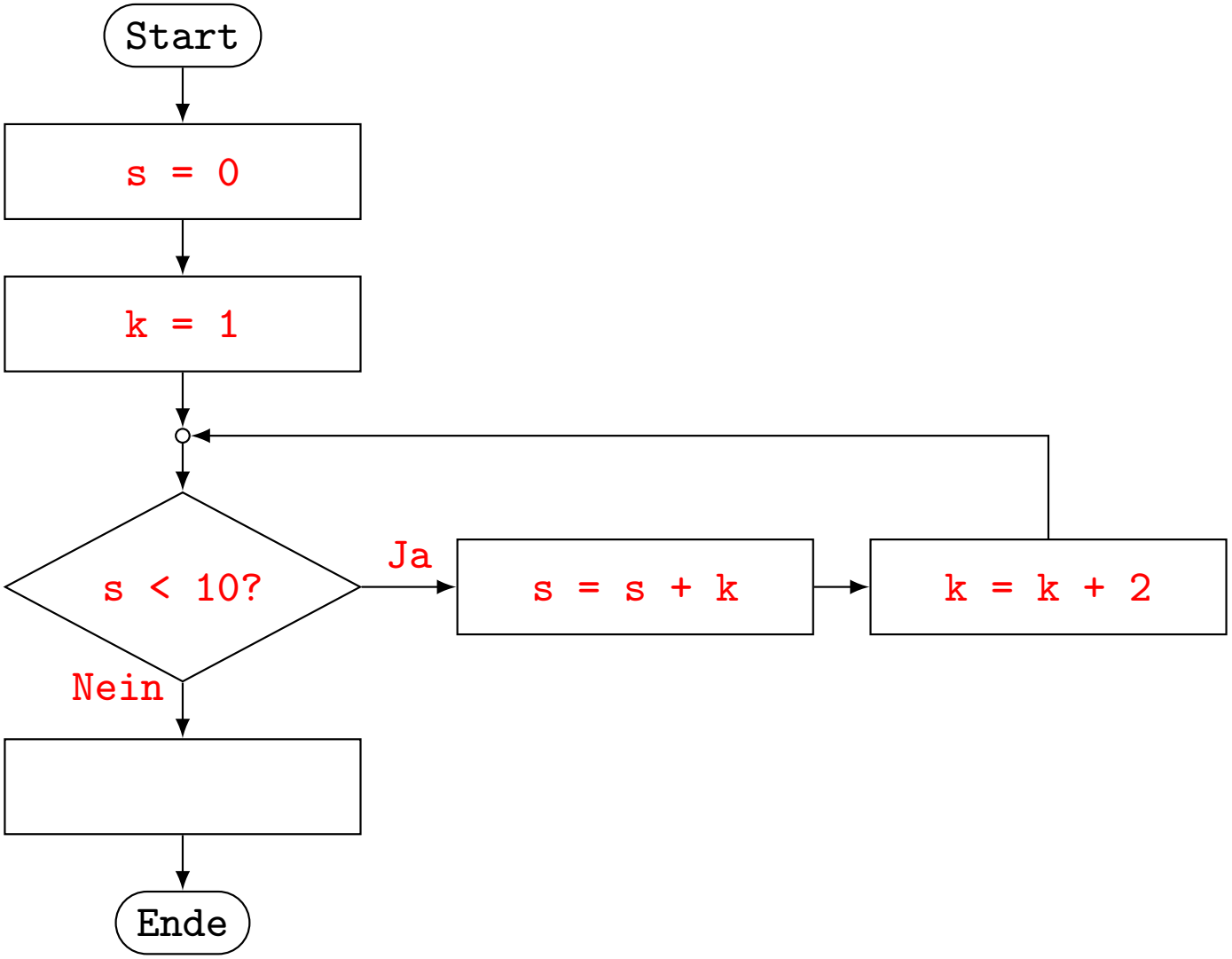
Darstellung als Flussdiagramm



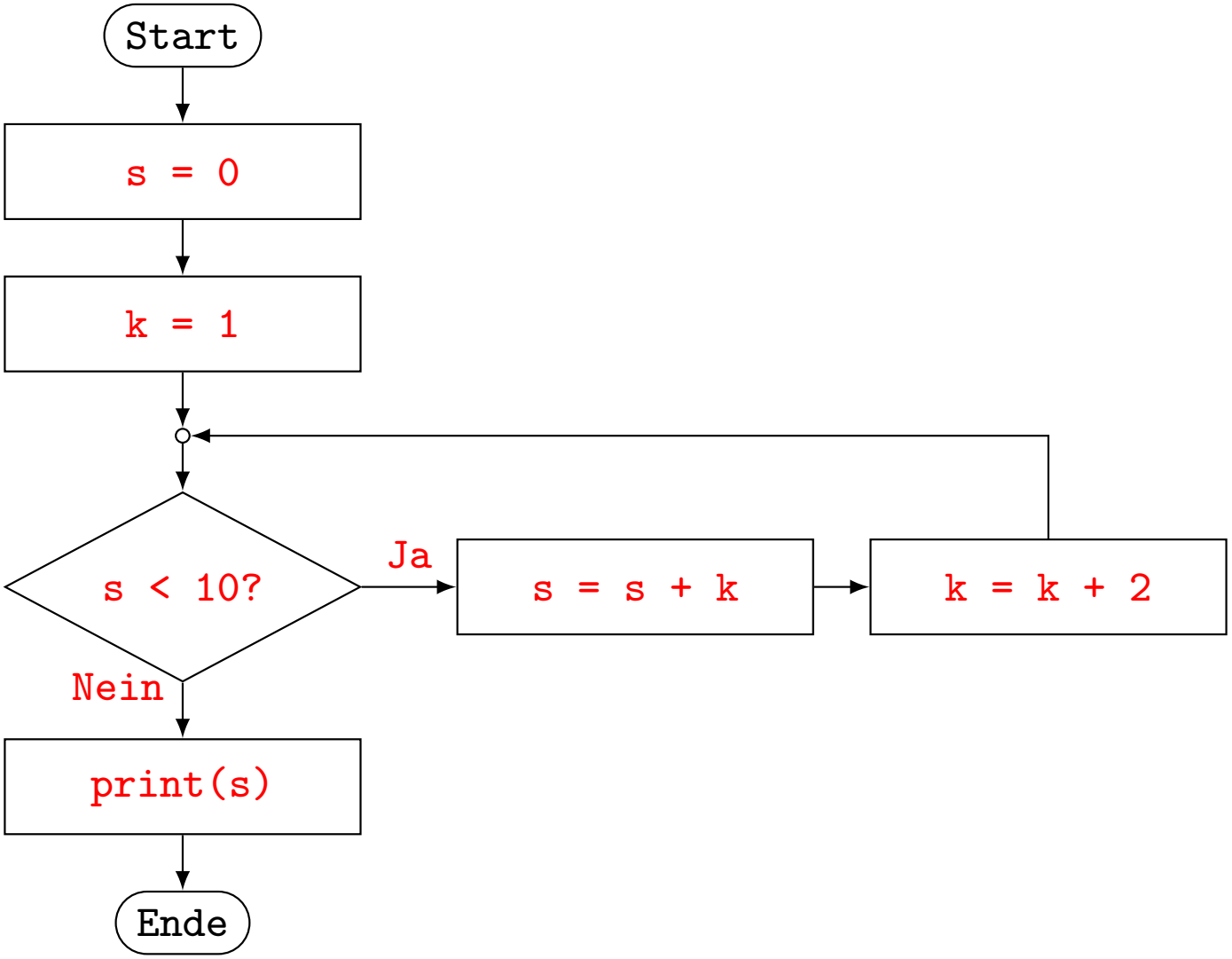
Darstellung als Flussdiagramm



Darstellung als Flussdiagramm



Darstellung als Flussdiagramm



Schleifen mit `break` frühzeitig abbrechen

Trifft Python bei der Ausführung des Codeblocks einer `for`- oder `while`-Schleife auf das Schlüsselwort `break`, so bricht es die Ausführung der Schleife ab und setzt das Programm nach der Schleife fort.

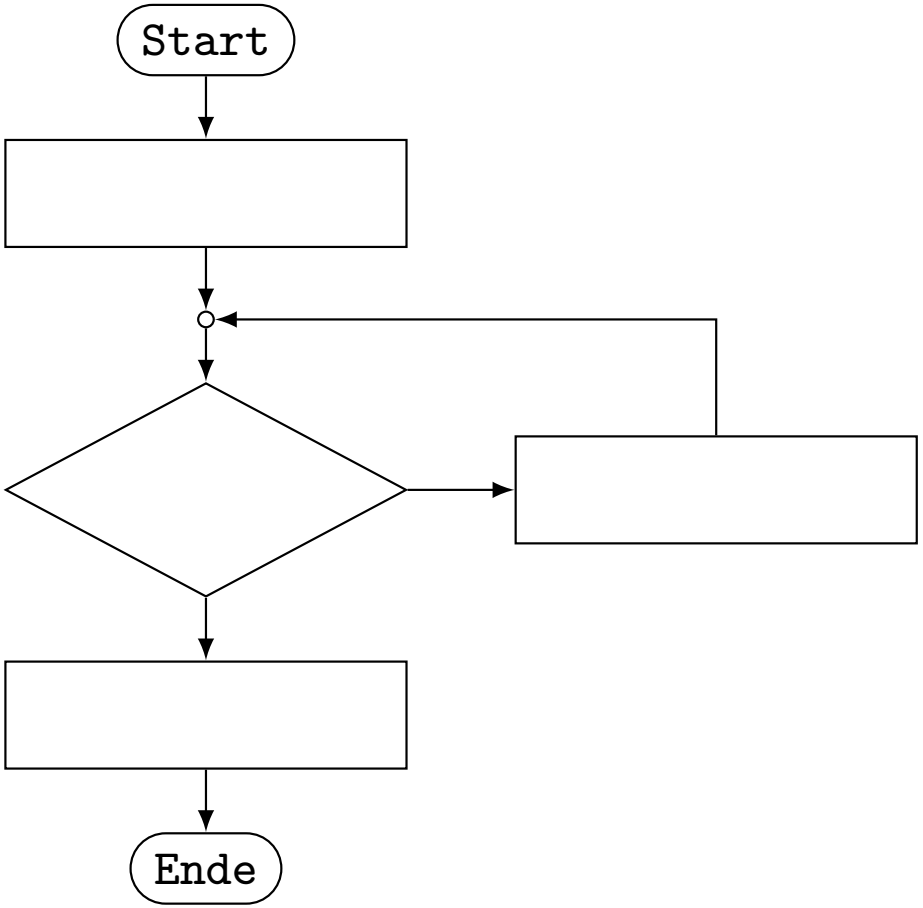
Beispiel 5.3

```
1  p = 4
2  while True:
3      if p > 50:
4          break
5      else:
6          p *= 2
7
8  print(p) # Ausgabe:
```

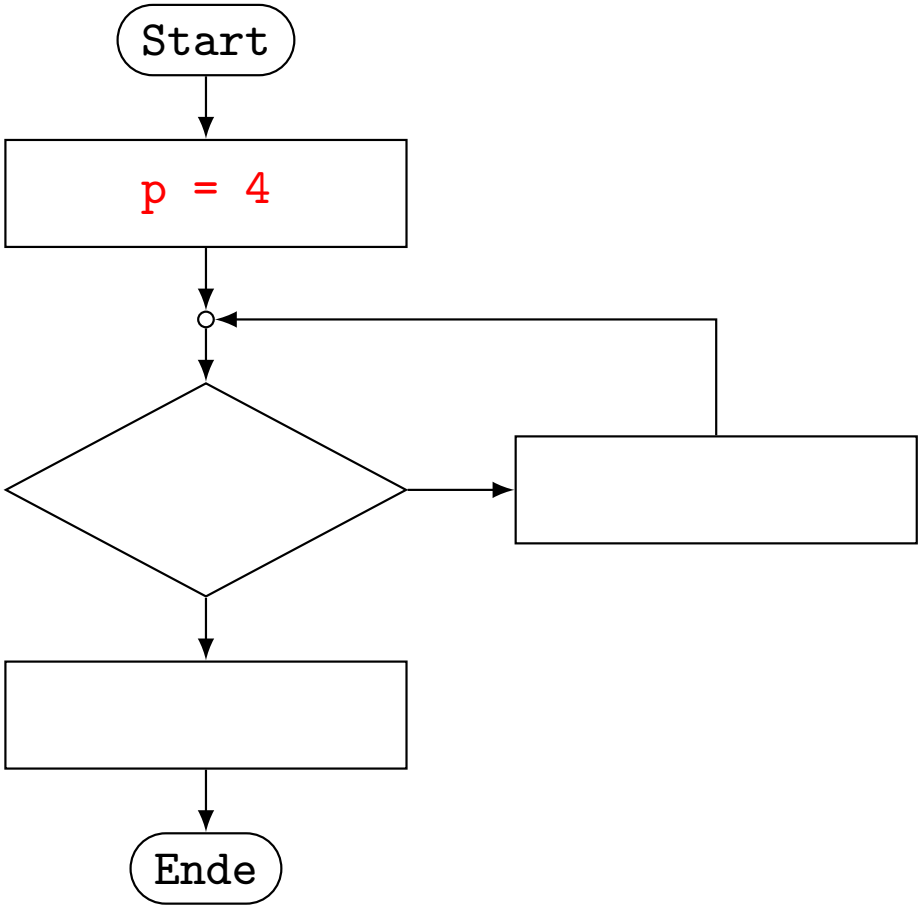
Beispiel 5.3

```
1  p = 4
2  while True:
3      if p > 50:
4          break
5      else:
6          p *= 2
7
8  print(p) # Ausgabe: 64
```

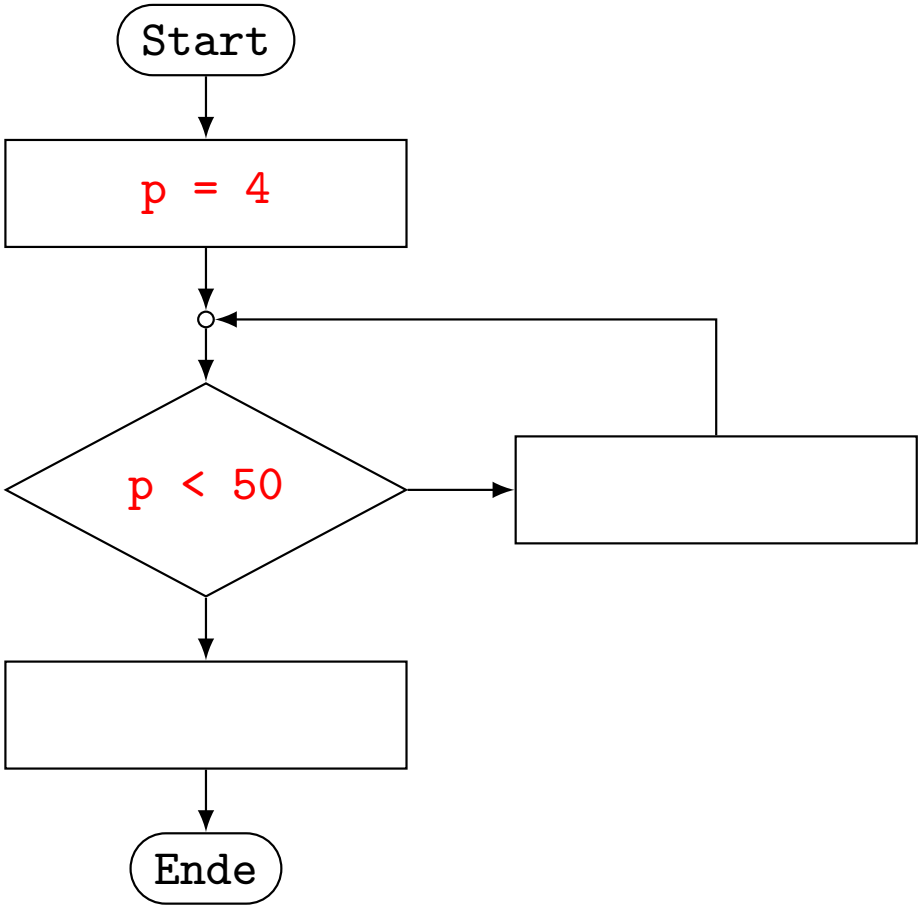
Darstellung als Flussdiagramm



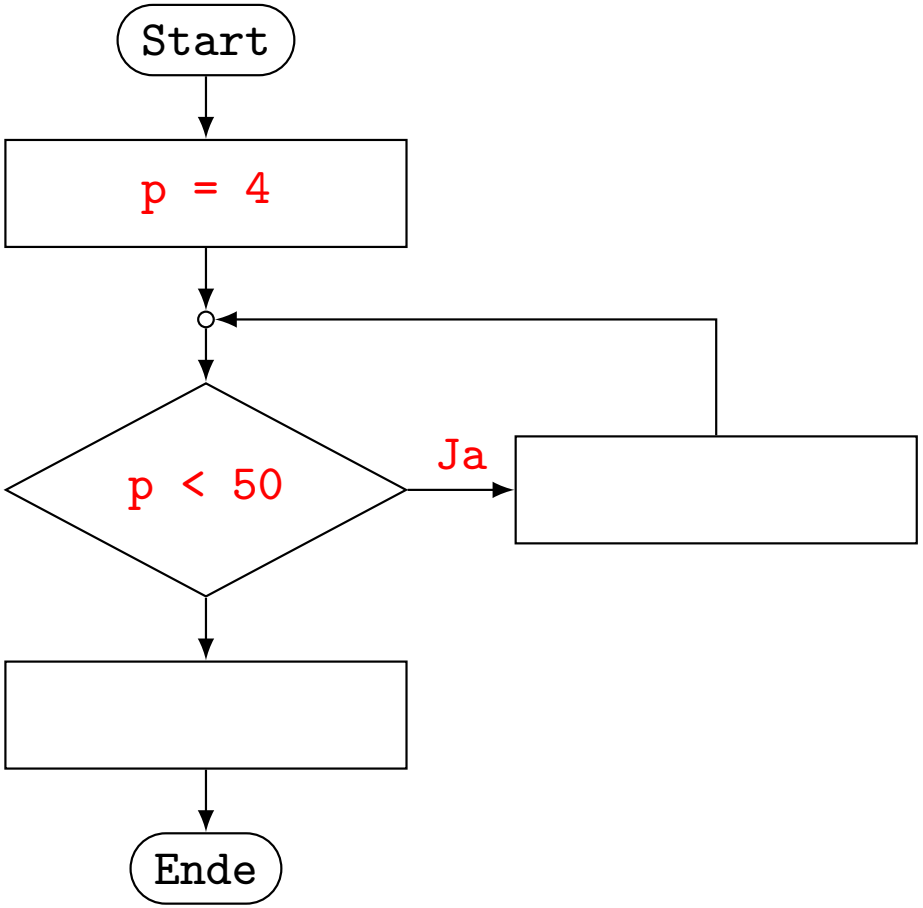
Darstellung als Flussdiagramm



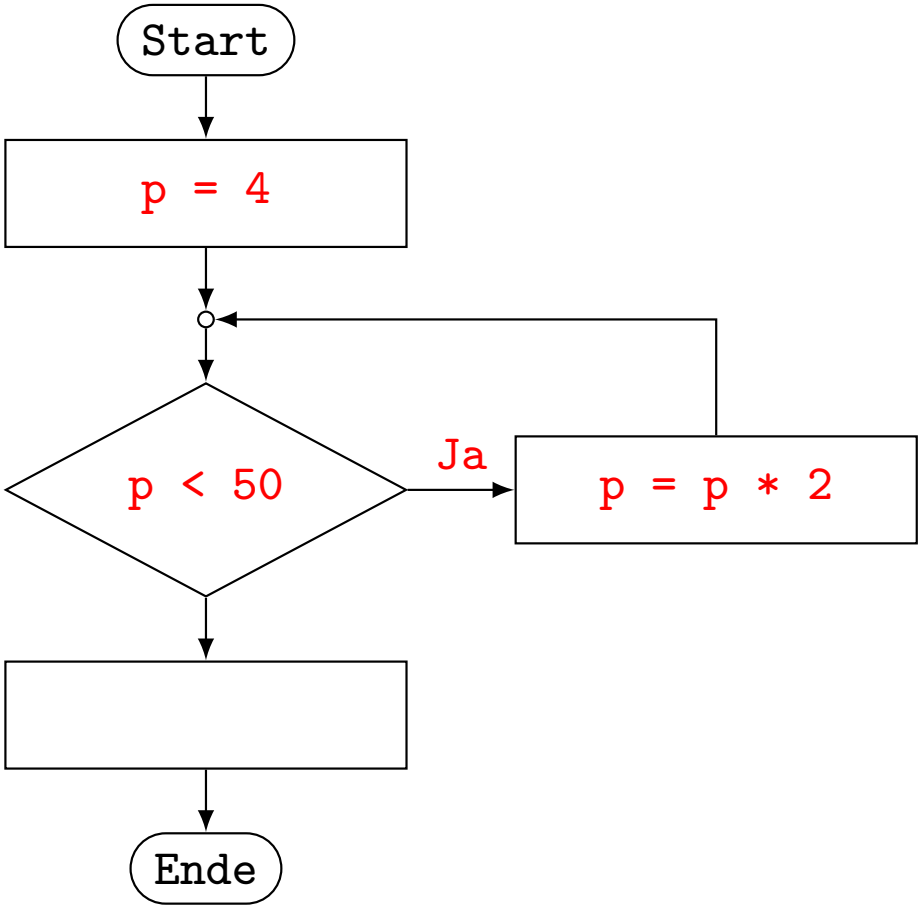
Darstellung als Flussdiagramm



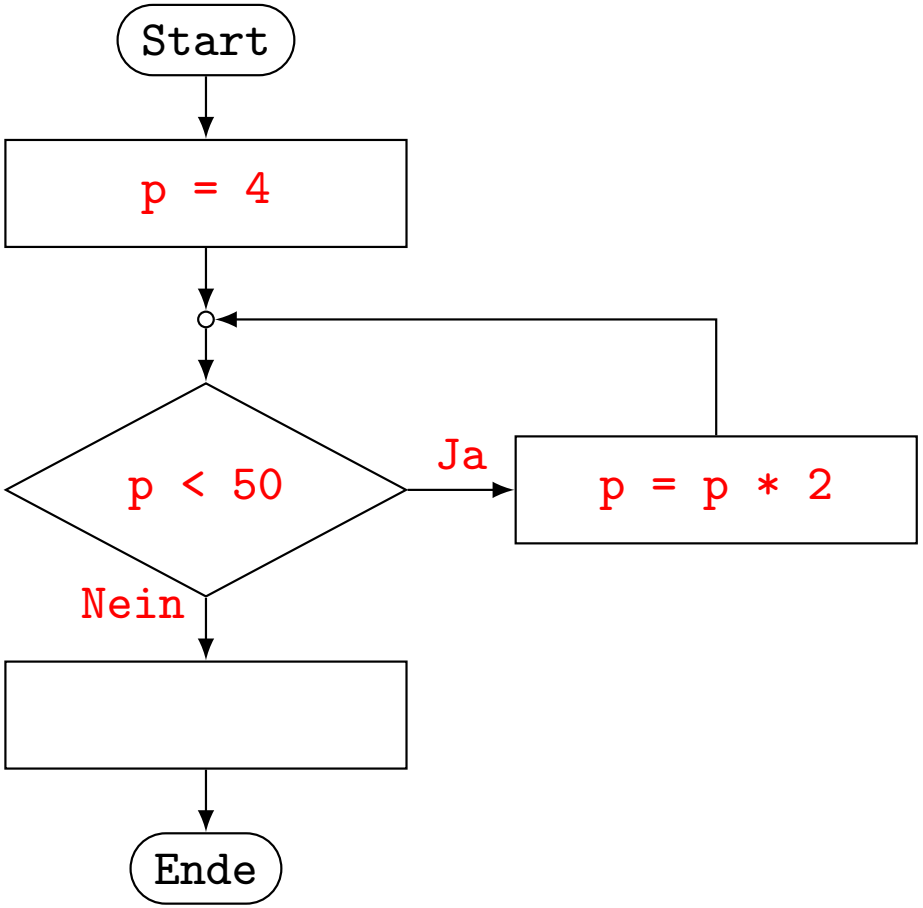
Darstellung als Flussdiagramm



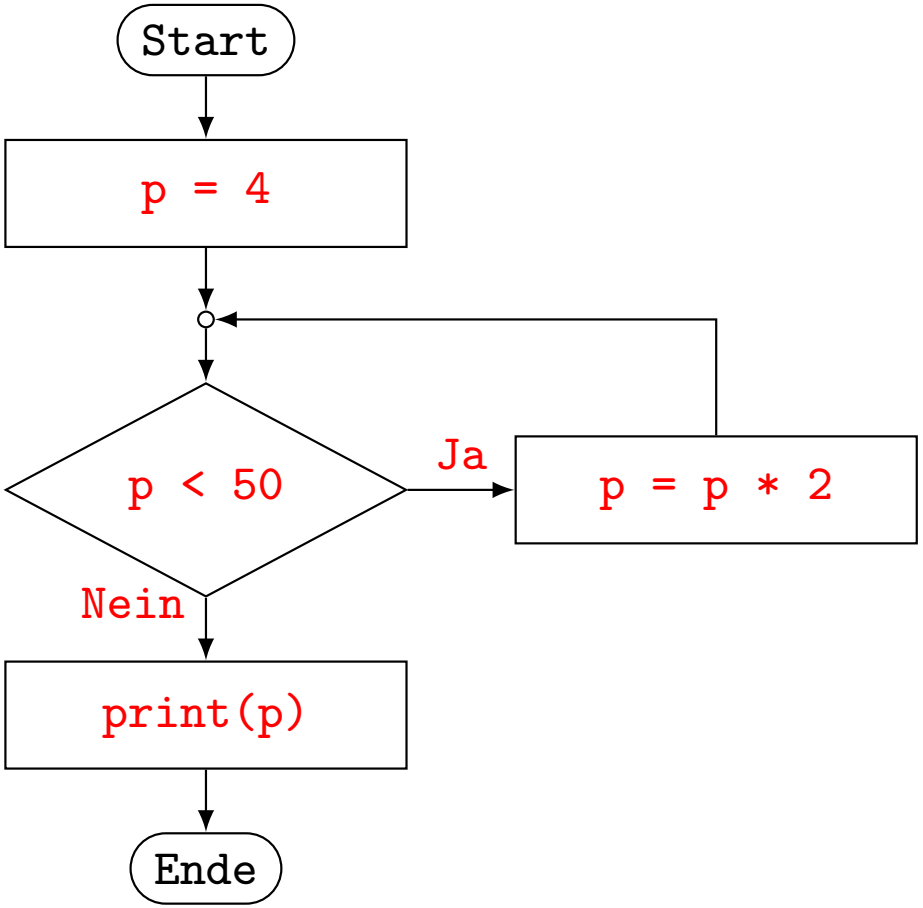
Darstellung als Flussdiagramm



Darstellung als Flussdiagramm



Darstellung als Flussdiagramm



Schleifendurchläufe mit `continue` überspringen

Trifft Python bei der Ausführung des Codeblocks einer `for`- oder `while`-Schleife auf das Schlüsselwort `continue`, so bricht es die Ausführung der Schleife ab und springt zur Bedingung im Schleifenkopf, um dort eventuell den nächsten Schleifendurchlauf zu starten.

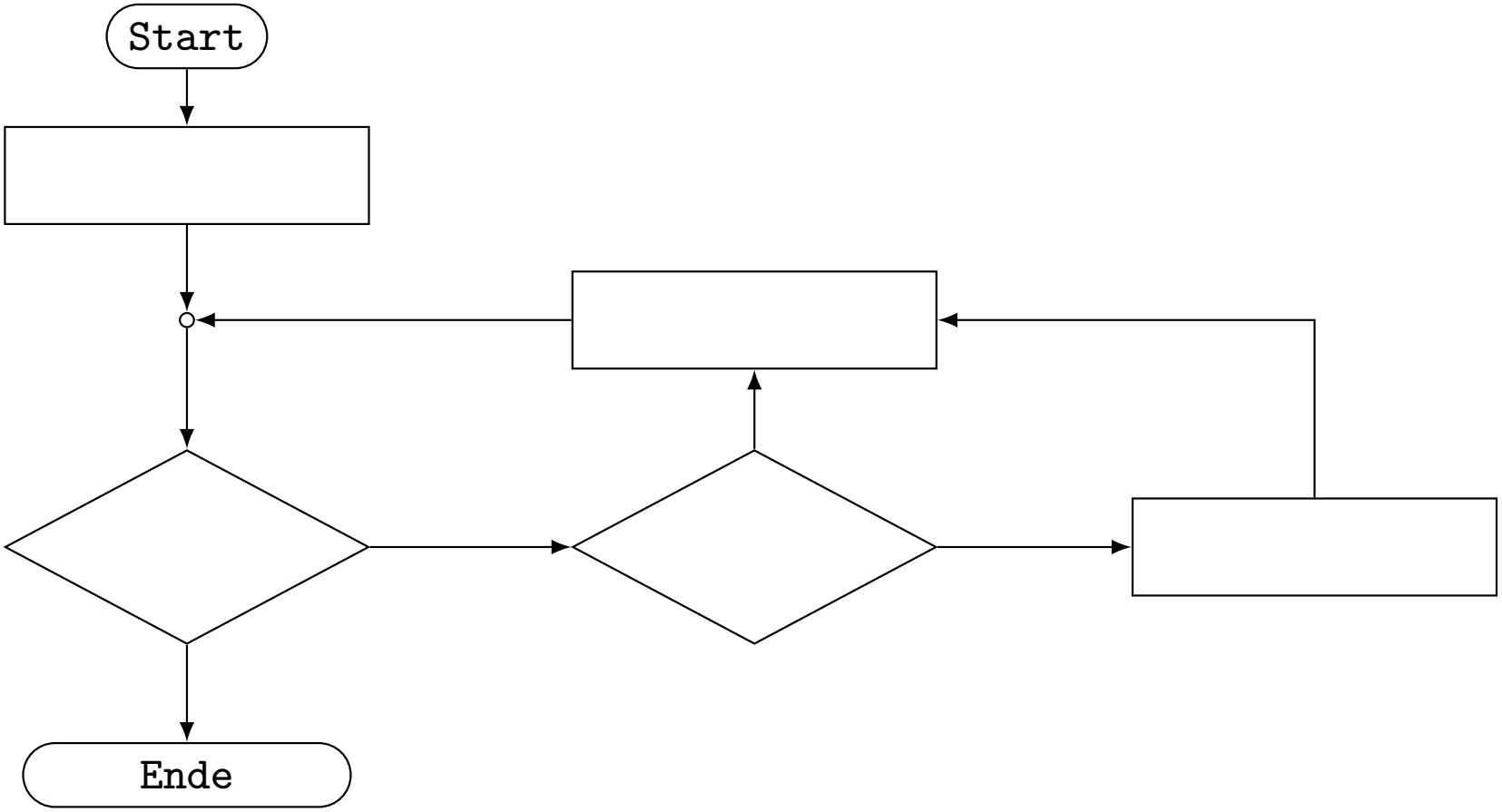
Beispiel 5.4

```
1  for k in range(1, 8):
2      if k % 3 == 0:
3          continue
4      print(k) # Ausgabe(n):
```

Beispiel 5.4

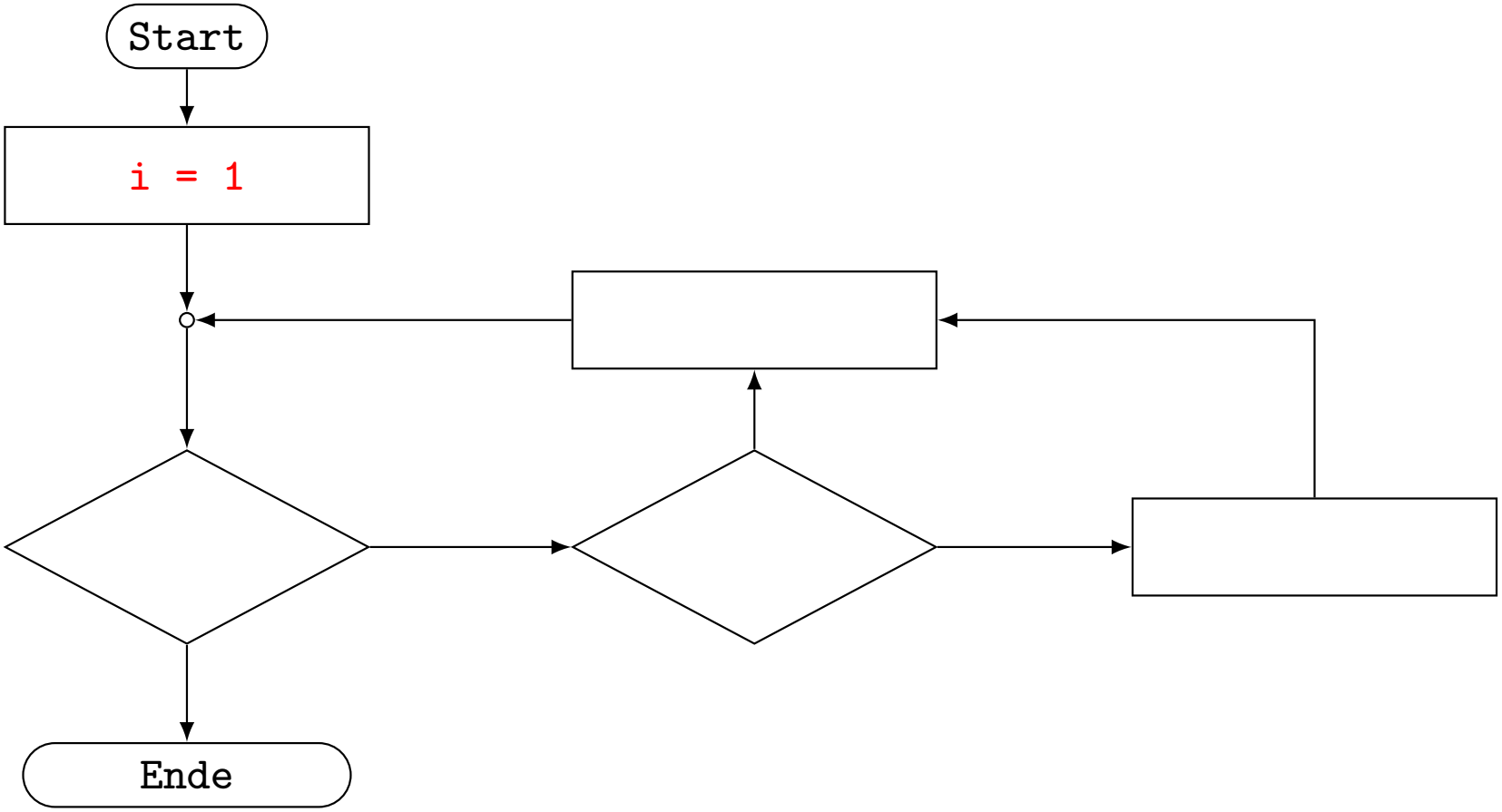
```
1  for k in range(1, 8):
2      if k % 3 == 0:
3          continue
4      print(k) # Ausgabe(n): 1, 2, 4, 5, 7
```

Darstellung als Flussdiagramm



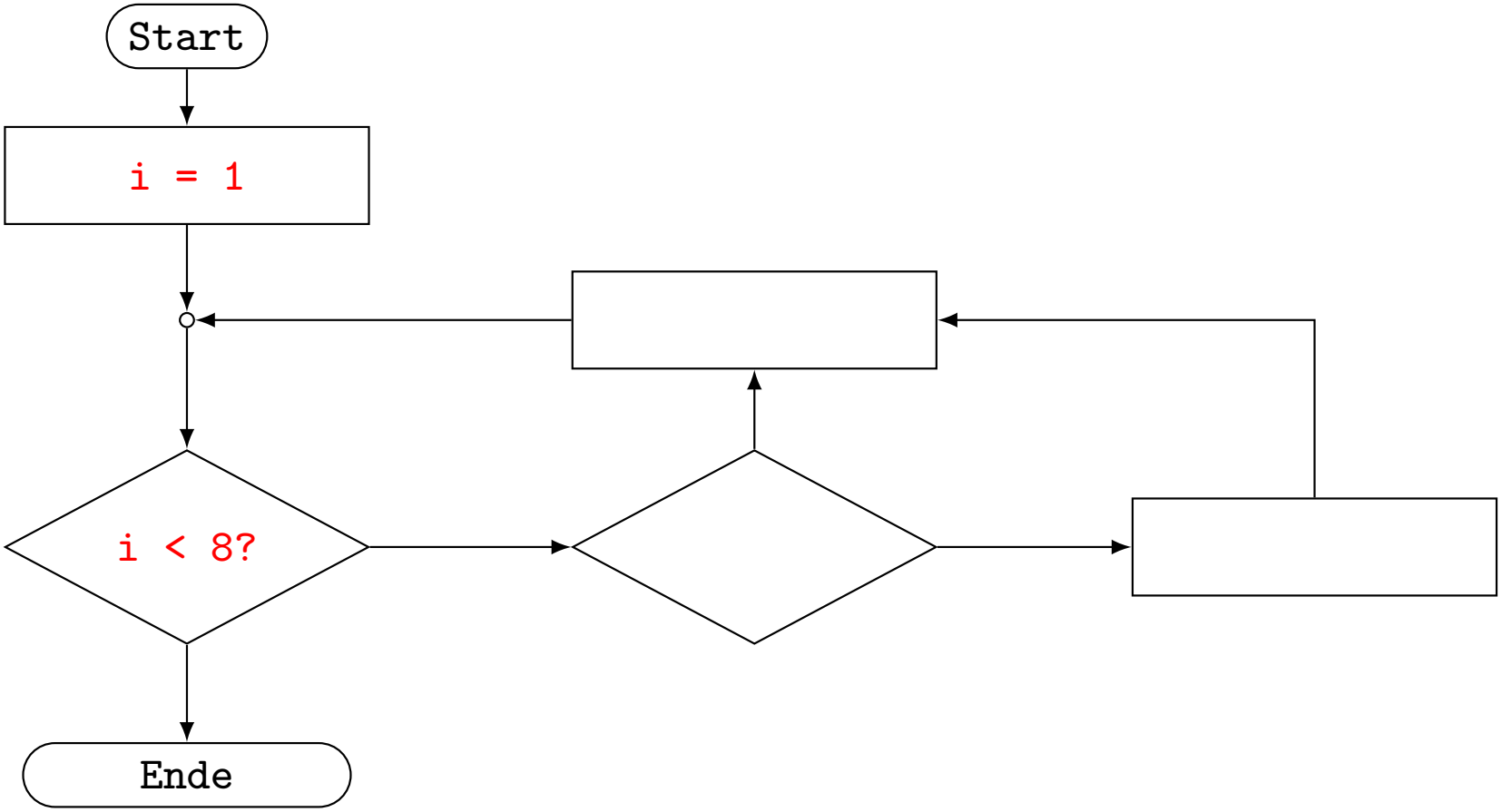
Bemerkung: Es gibt keine speziellen graphischen Elemente, um Schleifen in Flussdiagrammen darzustellen. Dafür wird die Verarbeitung des Programms im Prozessor genauer abgebildet.

Darstellung als Flussdiagramm



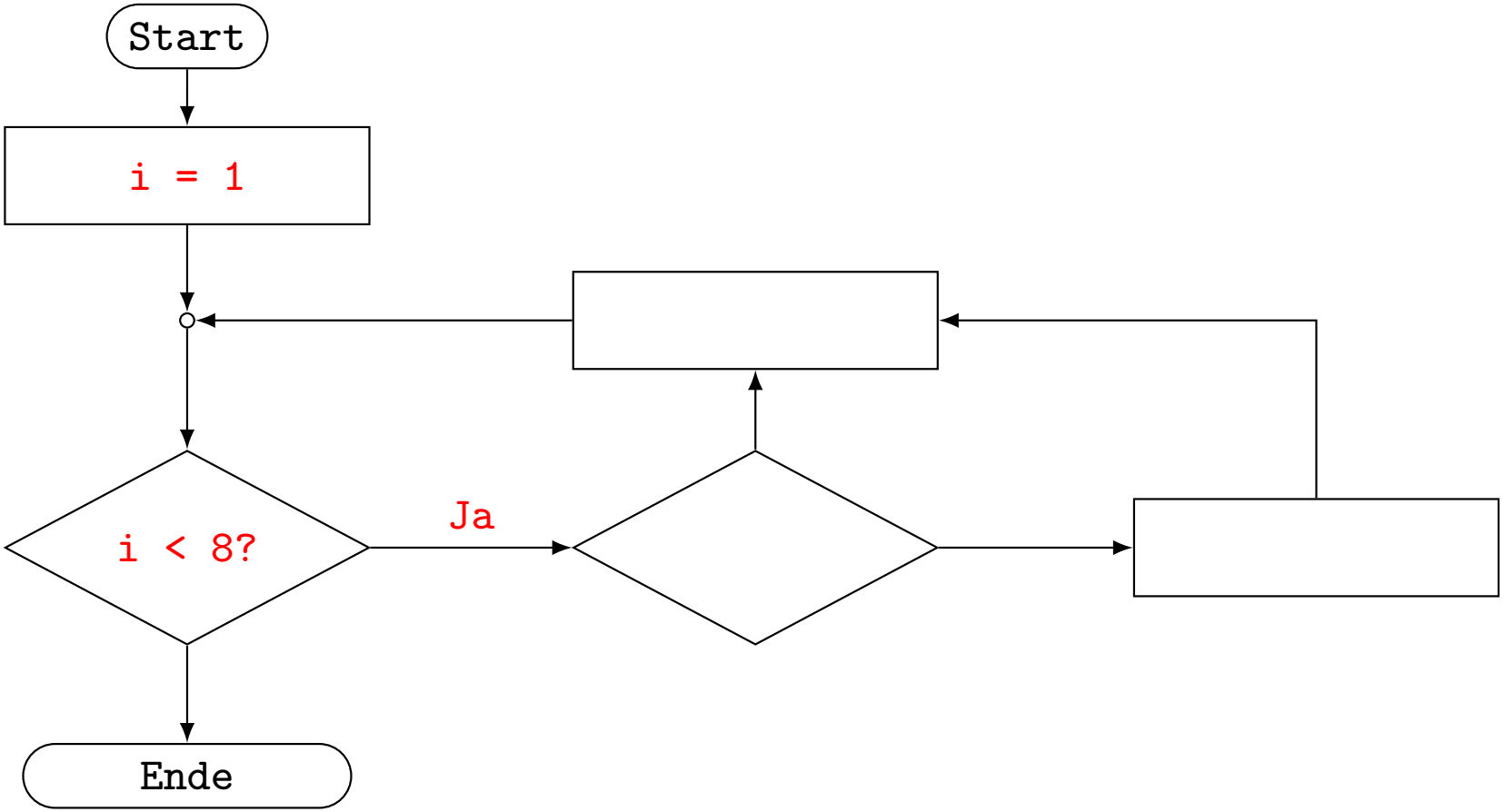
Bemerkung: Es gibt keine speziellen graphischen Elemente, um Schleifen in Flussdiagrammen darzustellen. Dafür wird die Verarbeitung des Programms im Prozessor genauer abgebildet.

Darstellung als Flussdiagramm



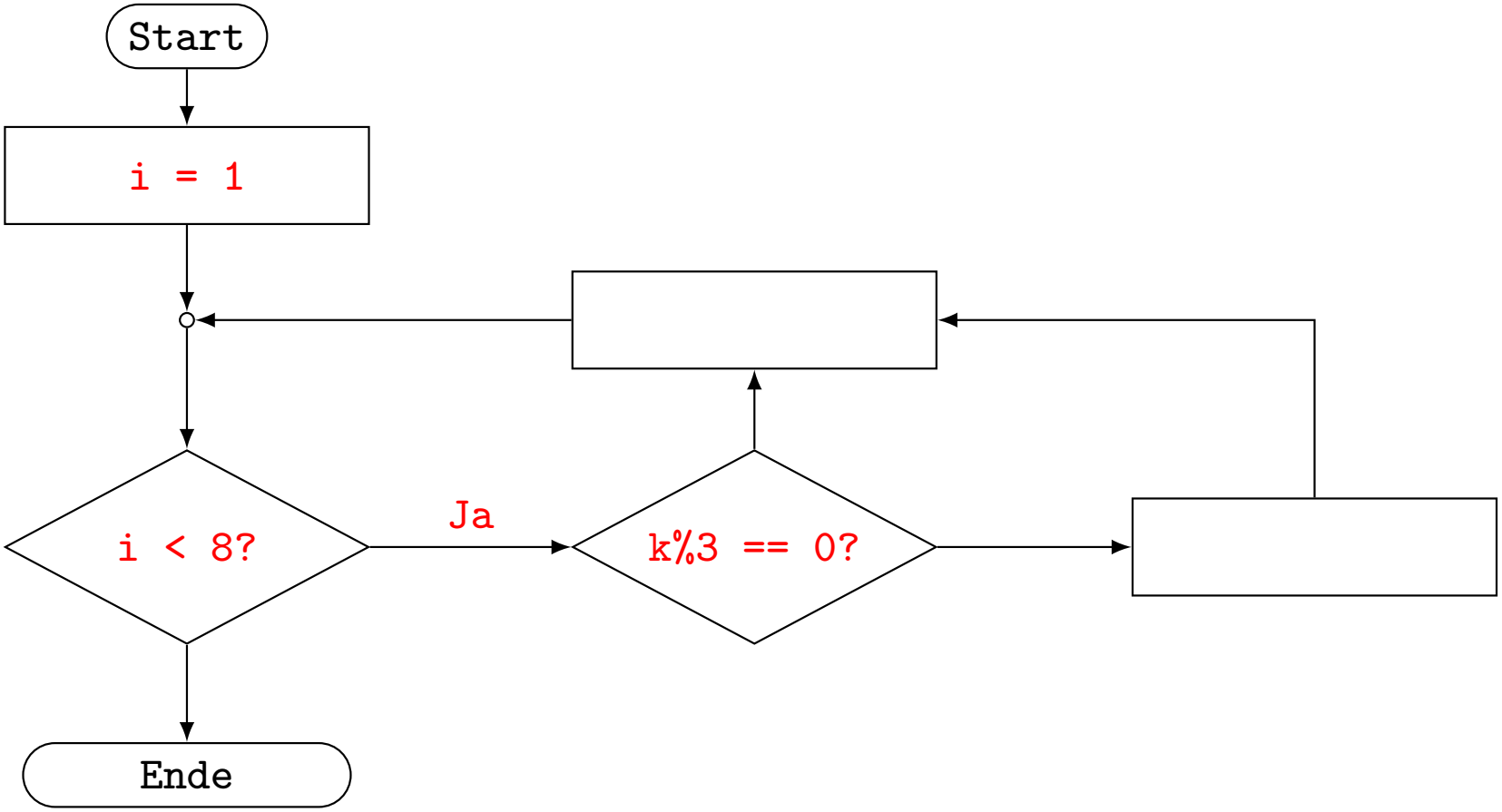
Bemerkung: Es gibt keine speziellen graphischen Elemente, um Schleifen in Flussdiagrammen darzustellen. Dafür wird die Verarbeitung des Programms im Prozessor genauer abgebildet.

Darstellung als Flussdiagramm



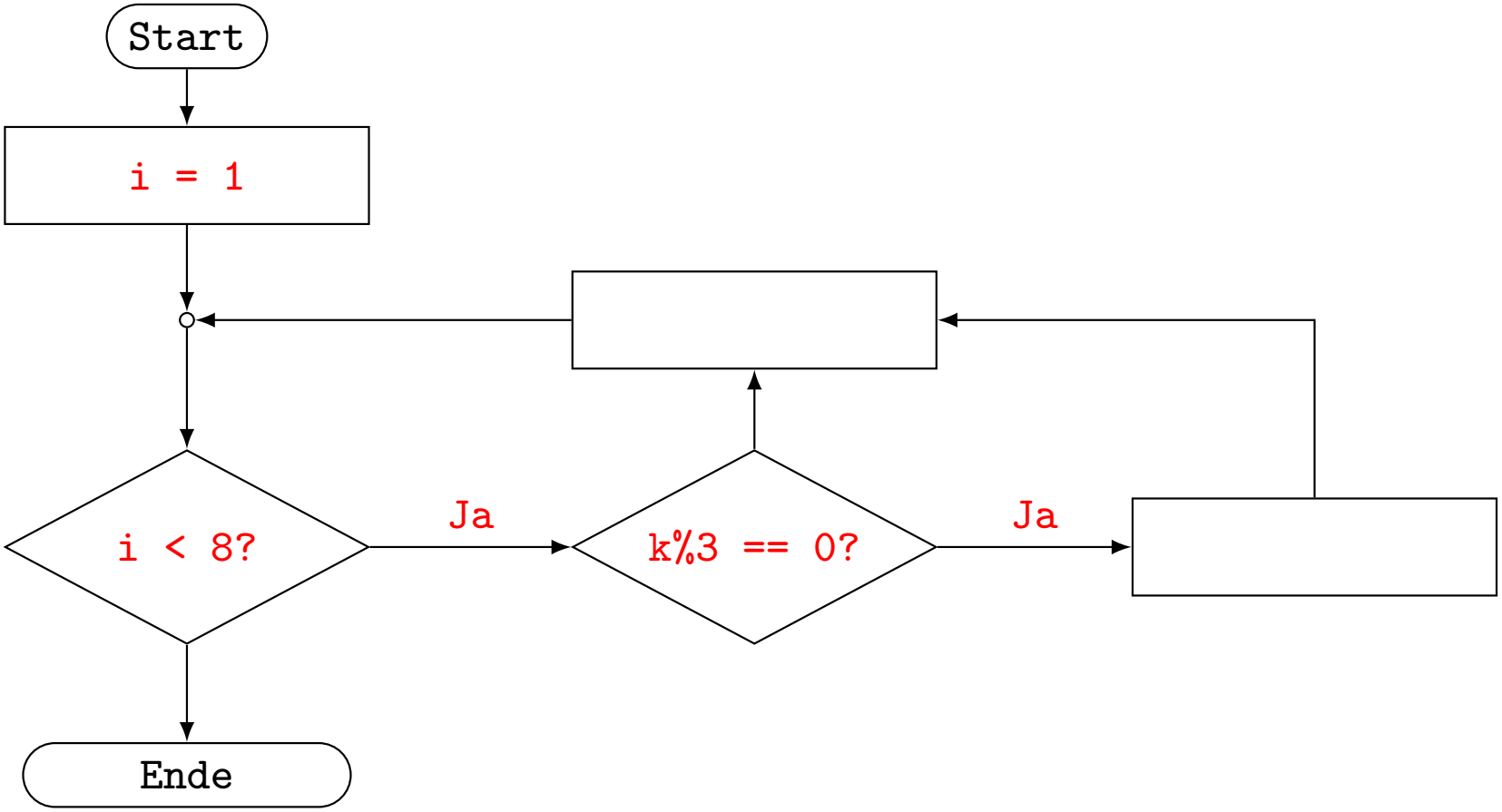
Bemerkung: Es gibt keine speziellen graphischen Elemente, um Schleifen in Flussdiagrammen darzustellen. Dafür wird die Verarbeitung des Programms im Prozessor genauer abgebildet.

Darstellung als Flussdiagramm



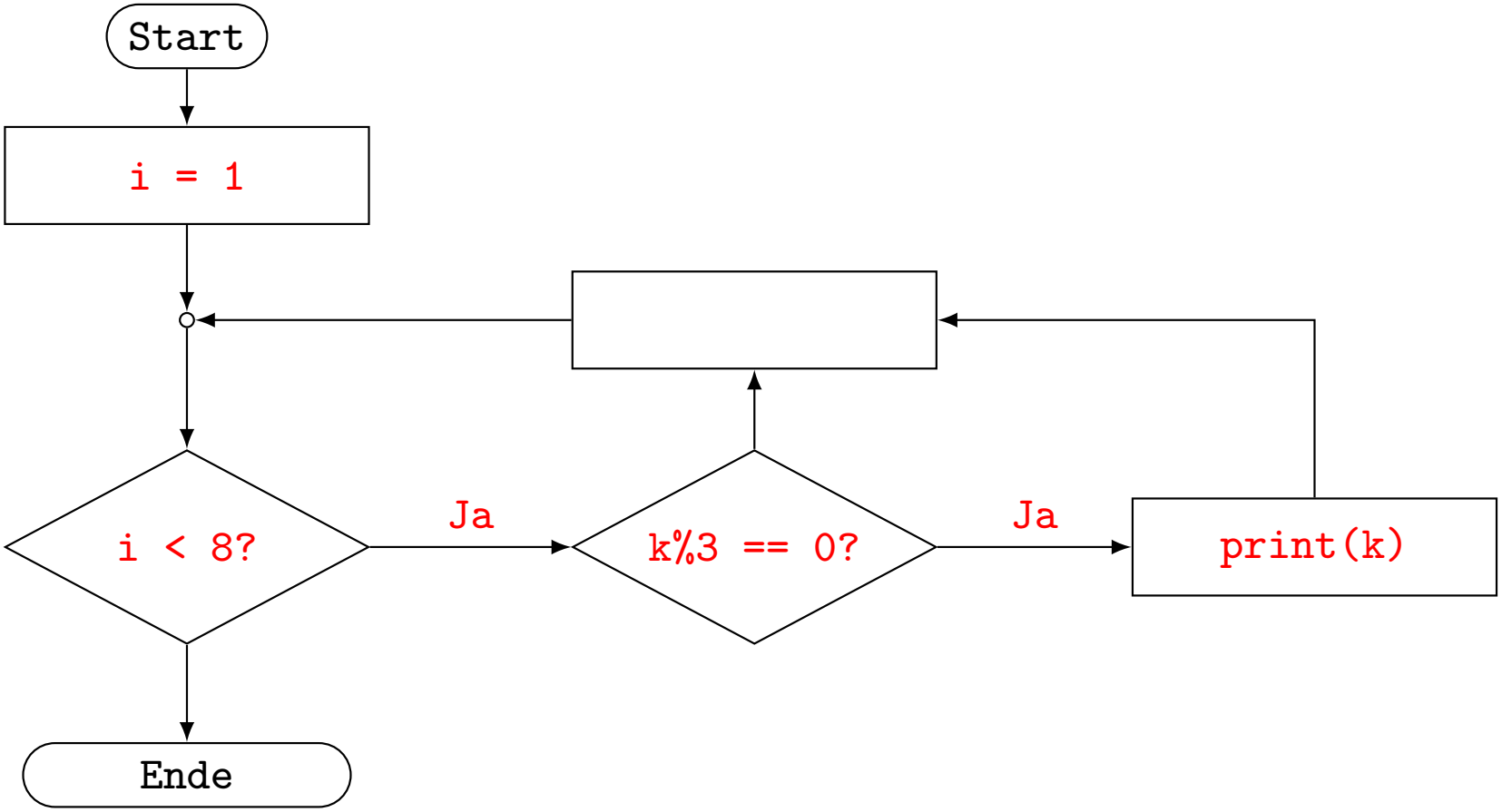
Bemerkung: Es gibt keine speziellen graphischen Elemente, um Schleifen in Flussdiagrammen darzustellen. Dafür wird die Verarbeitung des Programms im Prozessor genauer abgebildet.

Darstellung als Flussdiagramm



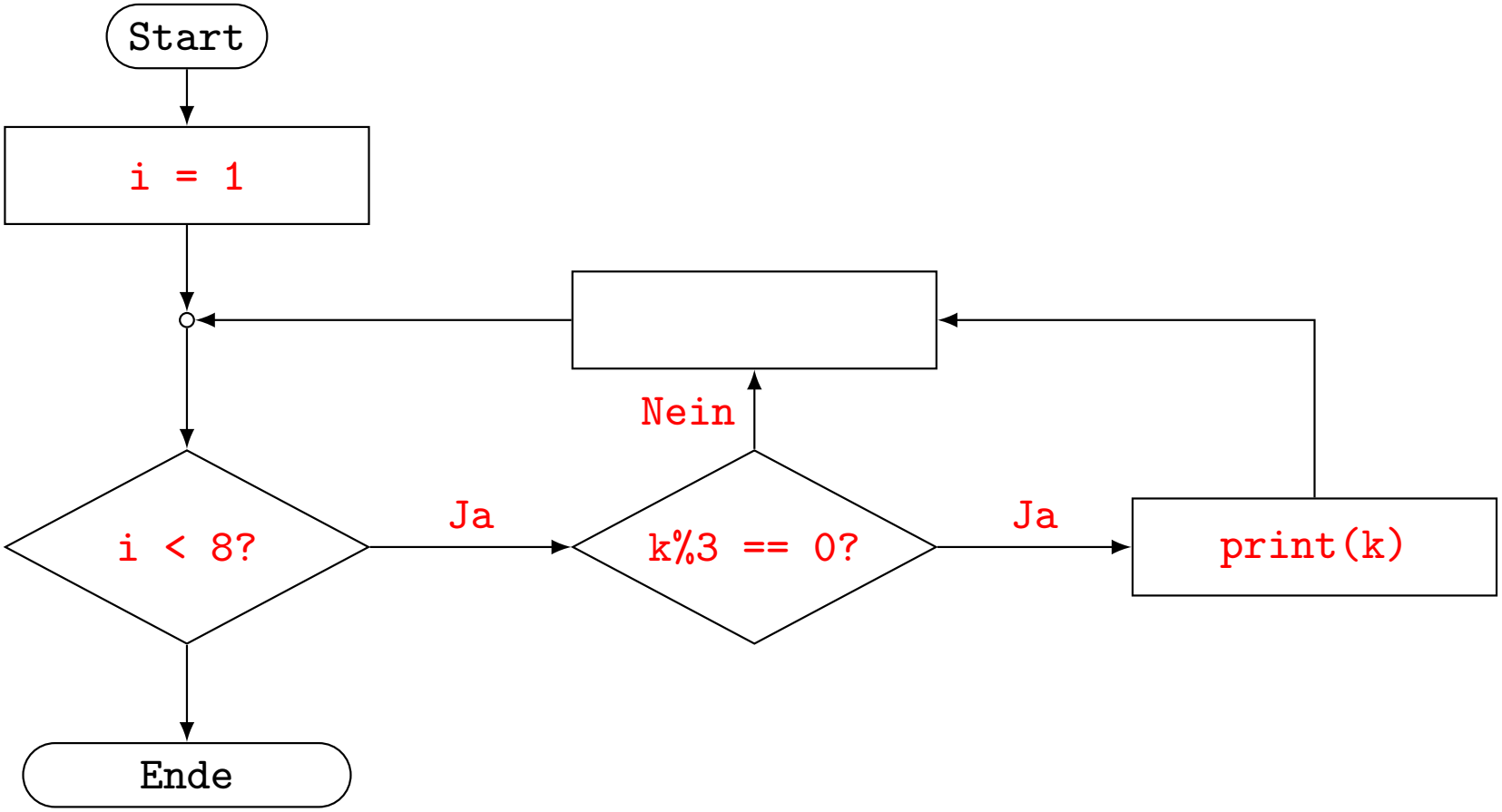
Bemerkung: Es gibt keine speziellen graphischen Elemente, um Schleifen in Flussdiagrammen darzustellen. Dafür wird die Verarbeitung des Programms im Prozessor genauer abgebildet.

Darstellung als Flussdiagramm



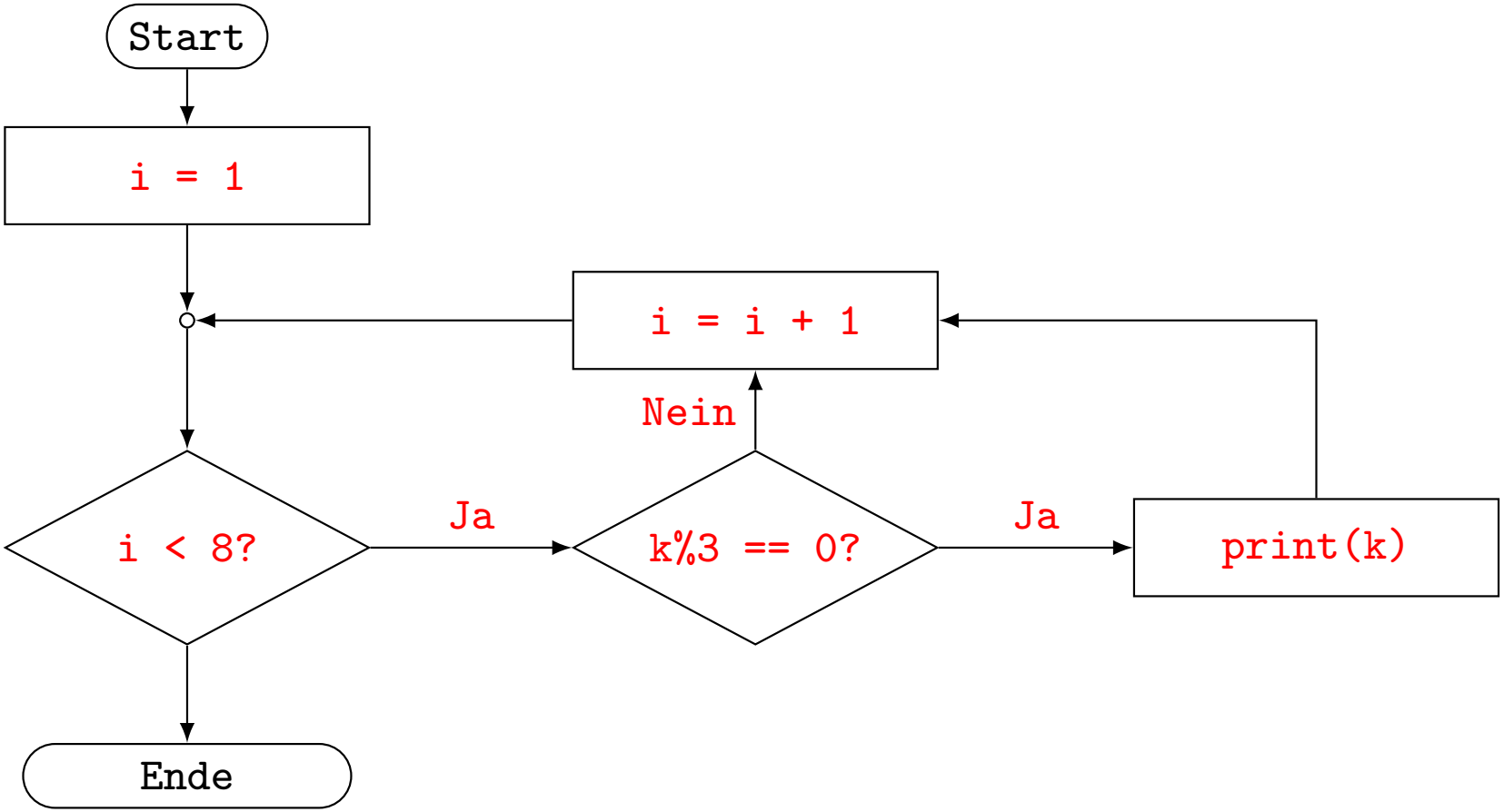
Bemerkung: Es gibt keine speziellen graphischen Elemente, um Schleifen in Flussdiagrammen darzustellen. Dafür wird die Verarbeitung des Programms im Prozessor genauer abgebildet.

Darstellung als Flussdiagramm



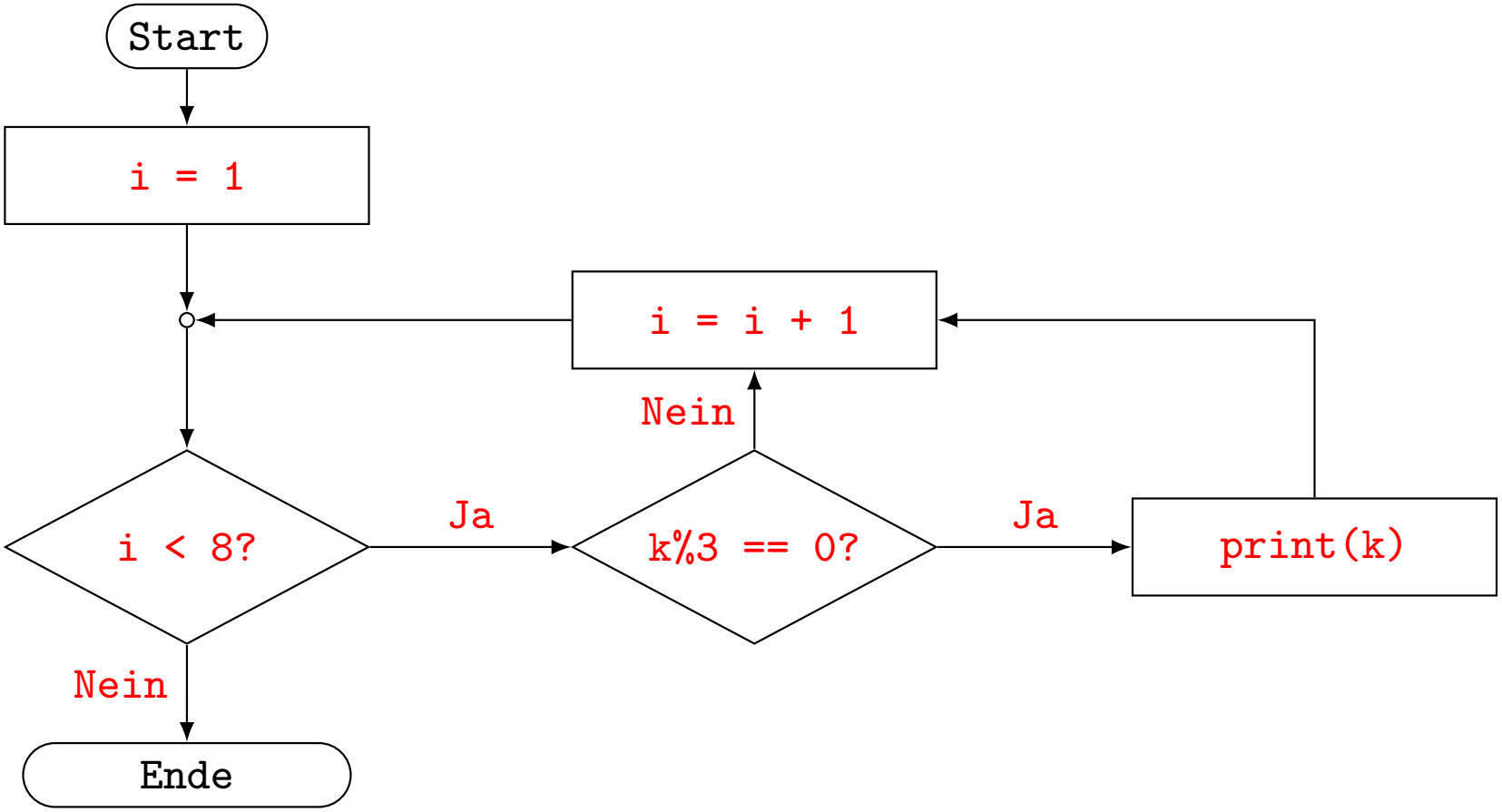
Bemerkung: Es gibt keine speziellen graphischen Elemente, um Schleifen in Flussdiagrammen darzustellen. Dafür wird die Verarbeitung des Programms im Prozessor genauer abgebildet.

Darstellung als Flussdiagramm



Bemerkung: Es gibt keine speziellen graphischen Elemente, um Schleifen in Flussdiagrammen darzustellen. Dafür wird die Verarbeitung des Programms im Prozessor genauer abgebildet.

Darstellung als Flussdiagramm



Bemerkung: Es gibt keine speziellen graphischen Elemente, um Schleifen in Flussdiagrammen darzustellen. Dafür wird die Verarbeitung des Programms im Prozessor genauer abgebildet.

Darstellung von Listen

Eine Liste wird durch eine kommaseparierte Folge von Werten beliebigen Datentyps definiert, die von einem Paar eckiger Klammern `[...]` eingeschlossen ist. Es ist auch möglich, eine Liste ohne Elemente zu definieren; dann spricht man von *der leeren Liste*.

Die Elemente einer Liste werden durch ihren Index, der bei Null beginnt, referenziert. Negative Indizes bedeuten, dass die Position vom Ende der Liste gezählt wird. So hat z. B. der Ausdruck `[5, -2, 8, 3][-1]` den Wert 3.

Ein Index, der grösser oder gleich der Anzahl Elemente in der Liste ist, verursacht zur Laufzeit einen `IndexError`.

Beispiel 6.1

```
1  L = [25, True, -7, 'abc', 1.41421]
2  print(L[2])  # =>
3  print(L[-2]) # =>
4  print(L[5])  # =>
```

Beispiel 6.1

```
1 L = [25, True, -7, 'abc', 1.41421]
2 print(L[2])  # => -7
3 print(L[-2]) # => 'abc'
4 print(L[5])  # => IndexError
```

Slices

Der Slice-Operator gibt in der einfachsten Form `L[i:j]` die Liste zurück, welche die Elemente von `L` an den Positionen `i`, `i+1`, ..., `j-1` enthält.

`[3, 8, 5, 0, 7, 1][1:4]` ⇒ `[8, 5, 0]`

Spezialfälle:

- `[3, 8, 5, 0, 7, 1][:4]` ⇒ `[3, 8, 5, 0]`
- `[3, 8, 5, 0, 7, 1][4:]` ⇒ `[7, 1]`
- `[3, 8, 5, 0, 7, 1][:]` ⇒ `[3, 8, 5, 0, 7, 1]`
- `[3, 8, 5, 0, 7, 1][2:100]` ⇒ `[5,0, 7, 1]`
- `[3, 8, 5, 0, 7, 1][3:3]` ⇒ `[]`
- `[3, 8, 5, 0, 7, 1][4:2]` ⇒ `[]`
- `[3, 8, 5, 0, 7, 1][::-1]` ⇒ `[1, 7, 0, 5, 8, 3]`
- `[3, 8, 5, 0, 7, 1][1::2]` ⇒ `[8, 0, 1]`

Beispiel 6.2

```
1 L = [3, 8, 1, 7, 5, 4, 2]
2 print(L[2:5]) # =>
3 print(L[3:])  # =>
4 print(L[:3])  # =>
5 print(L[:])   # =>
```

Beispiel 6.2

```
1 L = [3, 8, 1, 7, 5, 4, 2]
2 print(L[2:5]) # => [1, 7, 5]
3 print(L[3:])  # => [7, 5, 4, 2]
4 print(L[:3])  # => [3, 8, 1]
5 print(L[:])   # => [3, 8, 1, 7, 5, 4, 2]
```

Funktionen für Listen

Im Folgenden sei `L` eine Python-Liste.

Funktion	**Wert**
`len(L)`	Anzahl der Elemente von `L`
`sorted(L)`	die Liste der aufsteigend sortierten Elementen von `L`
`max(L)`	das grösste Element von `L`
`min(L)`	das kleinstes Element von `L`
`sum(L)`	Summe aller Elemente einer Liste `L` von Zahlen

Beispiel 6.3

```
1  L = [5, 3, 8, 2]
2  print(len(L))      # =>
3  print(sorted(L))   # =>
4  print(max(L))      # =>
5  print(sum(L))      # =>
```

Beispiel 6.3

```
1  L = [5, 3, 8, 2]
2  print(len(L))      # => 4
3  print(sorted(L))   # => [2, 3, 5, 8]
4  print(max(L))      # => 8
5  print(sum(L))      # => 18
```

Operatoren für Listen

Im Folgenden seien `A` und `B` Python-Listen sowie `n` eine natürliche Zahl.

Operator	**Resultat**
`A + B`	Liste aus den Elementen von `A` und `B`
`n * A`	Liste aus `n`-facher Repetition der Elemente von `A`

Beispiel 6.4

```
1  A = [1, 4]
2  B = [3, 2, 5]
3  print(A + B)   # =>
4  print(3*A)     # =>
```

Beispiel 6.4

```
1  A = [1, 4]
2  B = [3, 2, 5]
3  print(A + B)  # => [1, 4, 3, 2, 5]
4  print(3*A)    # => [1, 4, 1, 4, 1, 4]
```

Methoden für Listen

Im Folgenden sind `L` und `M` Python-Listen, `i` ein gültiger Index sowie `e` ein beliebiges Objekt.

Methode	**Beschreibung**
`L.pop()`	Entfernt `L[-1]` und liefert es als Wert zurück.
`L.pop(i)`	Entfernt `L[i]` und liefert es als Wert zurück.
`L.append(e)`	Fügt `e` am Ende von `L` an.
`L.insert(i,e)`	Fügt `e` an der Position `i` ein.
`L.index(e)`	Index des ersten Auftretens von `e`.
`L.reverse()`	Kehrt die Reihenfolge der Elemente *in place* um.
`L.remove(e)`	Entfernt erstes Vorkommen von `e`.
`L.sort()`	Sortiert `L` *in place*.
`L.extend(M)`	Erweitert `L` um die Elemente der Liste `M` .

Beispiel 6.5

```
1  L = [3, -4, 5, 2, 7]
2  x = L.pop()
3  print(x)              # =>
4  print(L)              # =>
5  y = L.pop(0)
6  print(y)              # =>
7  print(L)              # =>
8  L.append(8)
9  print(L)              # =>
10 L.insert(3, 1)
11 print(L)              # =>
12 L.extend([3, 7])
13 print(L)              # =>
```

Beispiel 6.5

```
1  L = [3, -4, 5, 2, 7]
2  x = L.pop()
3  print(x)              # => 7
4  print(L)              # => [3, -4, 5, 2]
5  y = L.pop(0)
6  print(y)              # => 3
7  print(L)              # => [-4, 5, 2]
8  L.append(8)
9  print(L)              # => [-4, 5, 2, 8]
10 L.insert(3, 1)
11 print(L)              # => [-4, 5, 2, 1, 8]
12 L.extend([3, 7])
13 print(L)              # => [-4, 5, 2, 1, 8, 3, 7]
```

Beispiel 6.6

```
1  L = [8, 5, 2, -3, 4, 5]
2  L.reverse()
3  print(L)              # =>
4  L.remove(5)
5  print(L)              # =>
6  L.sort()
7  print(L)              # =>
8  print(L.index(5))     # =>
```

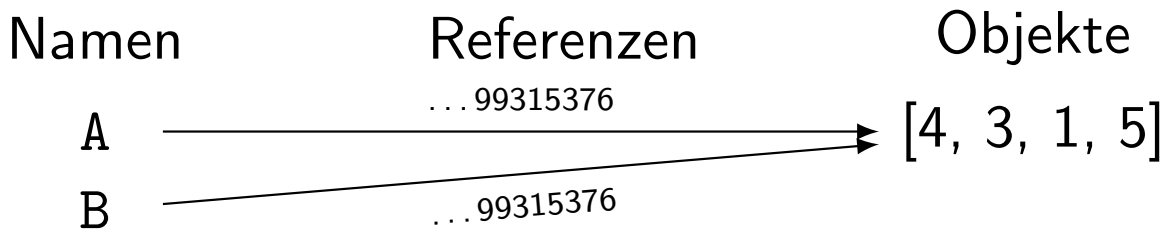
Beispiel 6.6

```
1  L = [8, 5, 2, -3, 4, 5]
2  L.reverse()
3  print(L)              # => [5, 4, -3, 2, 5, 8]
4  L.remove(5)
5  print(L)              # => [4, -3, 2, 5, 8]
6  L.sort()
7  print(L)              # => [-3, 2, 4, 5, 8]
8  print(L.index(5))     # => 3
```

Listen sind veränderliche Objekte

Da Listen *veränderliche Objekte* sind, können Zuweisungen zu unerwünschten Resultaten führen.

```
1 A = [4, 3, 1, 5]
2 B = A
3 B[0] = 9
4 print(A) # => [9, 3, 1, 5]
```



Da `A` und `B` dasselbe Objekt referenzieren, sind Änderungen an einer der Variablen immer auch bei der anderen sichtbar.

Ist dies unerwünscht, muss man eine „echte“ Kopie der Liste (z. B. mit einer Slice `B = A[:]`) erzwingen.

Die listengesteuerte for-Schleife

```
1 for var in liste:
2     codeblock
3 ...
```

Die Elemente der Liste *liste* werden in der Reihenfolge ihres Auftretens der Variablen *var* zugewiesen und damit jeweils der *codeblock* ausgeführt.

Beispiel 6.7

```
1  s = 0
2  for x in [2, -5, 3, -7, 9]:
3      if x > 0:
4          s += x
5  print(s)            # Ausgabe:
```

Beispiel 6.7

```
1  s = 0
2  for x in [2, -5, 3, -7, 9]:
3      if x > 0:
4          s += x
5  print(s)            # Ausgabe: 14
```

List Comprehensions

Mit List Comprehensions („Listenumfassungen“) können Listen ohne explizite Schleifen effizient erzeugt und verarbeitet werden.

Beispiel 6.8

```
1  # Listen erzeugen:
2  A = [i**2 for i in range(1, 5)]
3  print(A) # => [1, 4, 9, 16]
4
5  # Listen "filtern":
6  B = [3, -1, 7, -2, 5]
7  C = [x for x in B if x > 0]
8  print(C) # => [3, 7, 5]
9  D = [1 if x > 0 else 0 for x in B]
10 print(D) # => [1, 0, 1, 0, 1]
```

Beispiel 6.9

```
1  A = [3, 2, 9, 8, 4]
2  B = [x + 10 for x in A]
3  print(B)  # =>
4
5  C = [x for x in A if x % 2 == 0]
6  print(C)  # =>
7
8  D = [1 if x < 5 else 0 for x in A]
9  print(D)  # =>
```

Beispiel 6.9

```
1  A = [3, 2, 9, 8, 4]
2  B = [x + 10 for x in A]
3  print(B)  # => [13, 12, 19, 18, 14]
4
5  C = [x for x in A if x % 2 == 0]
6  print(C)  # => [2, 8, 4]
7
8  D = [1 if x < 5 else 0 for x in A]
9  print(D)  # => [1, 1, 0, 0, 1]
```

Was sind Funktionen?

Eine Funktion ist die Definition eines Codeblocks, der zu einem spätereren Zeitpunkt mit einem Funktionsnamen und mit Parametern aufgerufen werden kann.

Funktionen ...

- erlauben die Wiederverwendung von Code,
- zerlegen Programme in kleinere logische Einheiten, so dass sie lesbarer werden,
- erleichtern das Testen, Ändern und Warten von Programmen.

Die def-Anweisung

```
def fname(p1, p2, ...):
    codeblock
...
```

Funktionen werden mit dem Schlüsselwort `def` definiert. Es folgt ein gültiger Bezeichner (*fname*) auf den unmittelbar ein Paar runder Klammern folgt. In diesen Klammern kann eine durch Kommas getrennte Folge von Parametern stehen. Parameter sind Namen, die als Platzhalter im *codeblock* eingesetzt werden und denen später beim Aufruf der Funktion in der Reihenfolge der Parameter konkrete Objekte zugewiesen werden. Der *codeblock* enthält die Anweisungen, die beim Aufruf der Funktion ausgeführt werden.

Ähnlich wie bei einer Zuweisung wird mit `def` einem Namen ein Funktionsobjekt zugewiesen und eine Referenz erstellt.

Rückgabewerte

```
def fname(p1, p2, ...):
    ...
    return wert
```

Steht im Funktionsrumpf eine Anweisung der Form `return` *wert* so beendet Python die Abarbeitung allfälliger weiterer Codezeilen und setzt den Rückgabwert *wert* an die Stelle des Funktionsaufrufs. Fehlt eine `return`-Anweisung, so gibt die Funktion den Wert `None` zurück.

Eine Funktion kann nur einen Rückgabewert haben. Diese Einschränkung lässt sich jedoch umgehen, indem man einen zusammengesetzten Datentyp (z. B. eine Liste) als Wert zurückgibt.

Beispiel 7.1

```
1 def f(a, b):
2     return 2*a + b
3
4 print(f(7, 1) + 4) # => 15 + 4 = 19
```

In den Zeilen 1 und 2 wird eine Funktion mit dem Namen `f` definiert, die zum doppelten Wert des ersten Parameters `a` den Wert des zweiten Parameters `b` addiert und das Resultat an den aufrufenden Code zurückgibt.

In Zeile 4 wird die Funktion `f` aufgerufen und den Parametern `a` und `b` in dieser Reihenfolge die Werte 7 und 1 zugewiesen. Im Funktionsrumpf wird damit $2 \cdot 7 + 1 = 15$ berechnet. Die `return`-Anweisung sorgt dafür, dass dieser Wert an die Stelle von `f(7, 1)` gesetzt wird. Daher gibt die `print`-Anweisung 19 aus.

Schlüsselwort-Argumente

Wenn bei einem Funktionsaufruf jedem Parameter („Schlüsselwort“) mit dem Zuweisungsoperator „=“ der gewünschte Wert zugewiesen wird, spielt die sonst positionsabhängige Reihenfolge der Parameter keine Rolle mehr, da die Zuordnung damit eindeutig wird. Positionale Argumente und Schlüsselwort-Argumente können auch gemischt werden, wenn die positionsabhängigen Argumente (in der richtigen Reihenfolge) *vor* den Schlüsselwort-Argumenten stehen.

Beispiel 7.2

```
1  def f(a, b, c):
2      return 100*a + 10*b + c
3
4  print(f(b=3, c=7, a=5)) # =>
5  print(f(4, c=1, b=2))   # =>
```

Beispiel 7.2

```
1  def f(a, b, c):
2      return 100*a + 10*b + c
3
4  print(f(b=3, c=7, a=5)) # => 537
5  print(f(4, c=1, b=2))   # => 421
```

Vorgabewerte

Bei der Definition einer Funktion können den Parametern vorgegebene Werte zugewiesen werden. Werden beim Aufruf der Funktion diesen Parametern keine Objekte zugewiesen, setzt Python die in der Funktionsdefinition angegebenen Vorgabewerte ein, sofern Python erkennen kann, welche der Parameter durch die Vorgaben ersetzt werden sollen. Daher müssen die Parameter mit den Vorgabewerten ganz rechts stehen

Beispiel 7.3

```
1  def f(a, b=3, c=4):
2      return a + b + c
3
4  print(f(1, 2, 3))    # =>
5  print(f(1, 2))       # =>
6  print(f(1))          # =>
7  print(f(b=5, a=7))   # =>
8  print(f(0, c=2))     # =>
```

Beispiel 7.3

```
1  def f(a, b=3, c=4):
2      return a + b + c
3
4  print(f(1, 2, 3))    # => 1 + 2 + 3 = 6
5  print(f(1, 2))       # => 1 + 2 + 4 = 7
6  print(f(1))          # => 1 + 3 + 4 = 8
7  print(f(b=5, a=7))   # => 7 + 5 + 4 = 16
8  print(f(0, c=2))     # => 0 + 3 + 2 = 5
```

Gültigkeitsbereich von Variablen

Wenn wir Variablen verwenden, dann sucht, erzeugt oder ändert Python die Namen an einem Ort, der Namensraum (namespace) genannt wird. Der Ort, an dem einer Variablen erstmals ein Wert zugewiesen wird, bestimmt den speziellen Namensraum, auch Gültigkeitsbereich (scope) genannt, in dem sich die Variable befindet. Genauer: Namen, denen innerhalb einer Funktionsdefinition ein Wert zugewiesen wird, ...

- sind nur innerhalb dieser Funktionsdefinition sichtbar. Es ist nicht möglich von ausserhalb der Funktion auf diese Namen zuzugreifen.
- kollidieren nicht mit identischen Namen ausserhalb der Funktionsdefinition.

Beispiel 7.4

```
1  x = 99
2
3  def meine_funktion(y):
4      z = x + y
5      return z
6
7  print(meine_funktion(1)) # =>
```

- Die Namen `x` und `meine_funktion` befinden sich im globalen Gültigkeitsbereich.
- Die Namen `y` und `z` liegen im lokalen Gültigkeitsbereich.
- Die Variable `x` ist überall sichtbar.
- Die Variablen `y` und `z` sind nur innerhalb von `meine_funktion` sichtbar.

Beispiel 7.4

```
1  x = 99
2
3  def meine_funktion(y):
4      z = x + y
5      return z
6
7  print(meine_funktion(1)) # => 100
```

- Die Namen `x` und `meine_funktion` befinden sich im globalen Gültigkeitsbereich.
- Die Namen `y` und `z` liegen im lokalen Gültigkeitsbereich.
- Die Variable `x` ist überall sichtbar.
- Die Variablen `y` und `z` sind nur innerhalb von `meine_funktion` sichtbar.

Unveränderliche und veränderliche Datentypen

Auch wenn Parameter und lokale Variablen einer Funktion unabhängig von gleichnamigen globalen Variablen sind, kann es in bestimmten Fällen dazu kommen, dass eine Funktion globale Variablen verändert, was meist unerwünscht ist. Dies hängt davon ab, ob das an die Funktion übergebene Objekt veränderlich (mutable) oder unveränderlich (immutable) ist und ob den lokalen Variablen im Funktionsinnern neue Objekte zugewiesen werden.

unveränderliche Datentypen: `int`, `float`, `str`, `bool`

veränderliche Datentypen: `list`

Die Aufzählung ist unvollständig und deckt nur die bisher behandelten Datentypen ab.

- Wird einer lokalen Variable ein *unveränderliches Objekt* einer globalen Variable zugewiesen, so entsteht automatisch ein neues (unabhängiges) lokales Objekt. Der Wert der globalen Variable bleibt deshalb unverändert.
- Wird einer lokalen Variable *ein veränderliches Objekt* einer globalen Variable zugewiesen und verändert die Funktion die lokale Variable *ohne eine Neuzuweisung* (*in place*), so wird auch der Wert der globalen Variable verändert.
- Wird einer lokalen Variable *ein veränderliches Objekt* einer globalen Variablen zugewiesen, und verändert die Funktion die lokale Variabe *mit einer Neuzuweisung*, so entsteht ein neues (unabhängiges) lokales Objekt. Der Wert der globalen Variable bleibt deshalb unverändert.

Beispiel 7.5

```
1  a = 1          # immutable
2  b = [3, 4]     # mutable
3  c = [6, 7]     # mutable
4
5  def try_to_modify(x, y, z):
6      x = x + 1
7      y.append(5)
8      z = 2 * z
9      print(x, y, z)
10
11 try_to_modify(a, b, c)  # =>
12
13 print(a, b, c)          # =>
```

Beispiel 7.5

```
1  a = 1          # immutable
2  b = [3, 4]     # mutable
3  c = [6, 7]     # mutable
4
5  def try_to_modify(x, y, z):
6      x = 2 * x
7      y.append(5)
8      z = 2 * z
9      print(x, y, z)
10
11 try_to_modify(a, b, c)  # => 2 [3, 4, 5] [6, 7, 6, 7]
12
13 print(a, b, c)          # => 1 [3, 4, 5] [6, 7]
```

Rekursion

Manchmal ist es sinnvoll, anstelle einer Schleife eine Funktion zu schreiben, die sich selber (rekursiv) aufruft. Dann muss innerhalb der Funktion eine Bedingung definiert sein, die für den Abbruch der Rekursion sorgt (Base Case) und zu ihrer Auflösung führt.

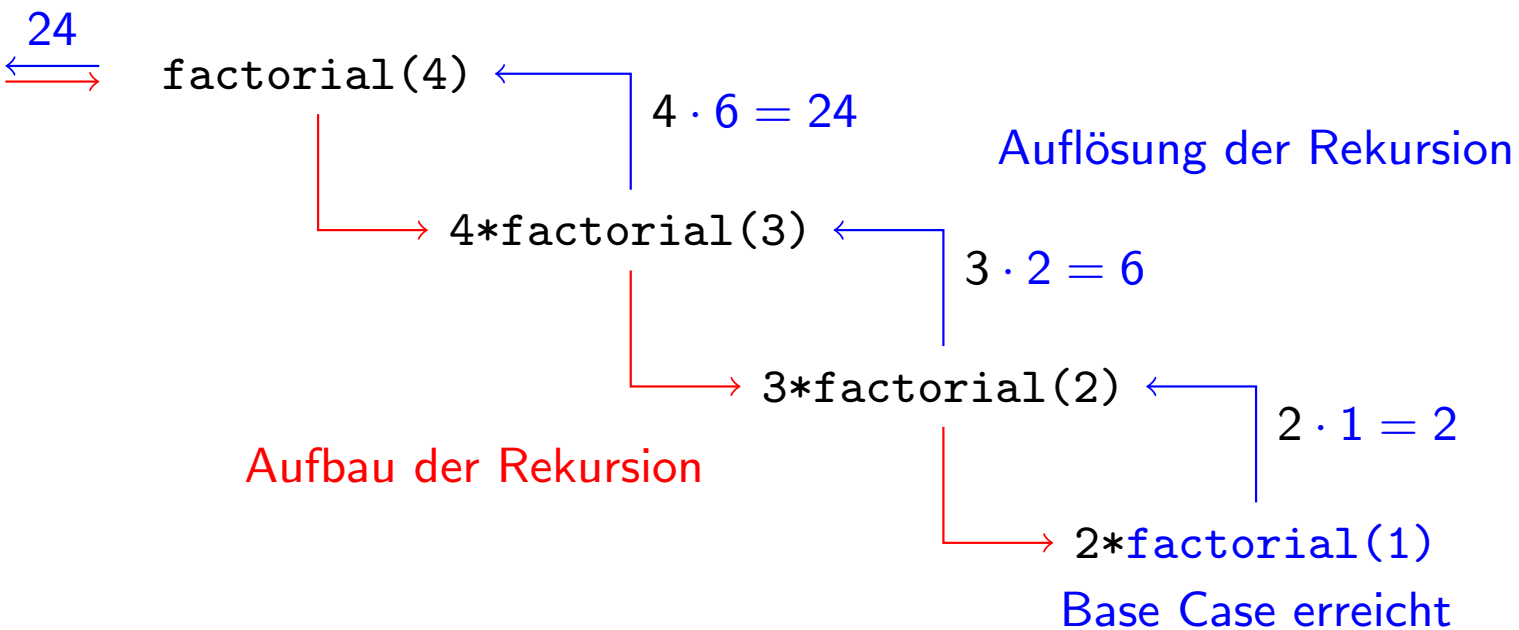
Da bei jedem Funktionsaufruf ein neuer Namensraum angelegt wird, kostet die rekursive Lösung einer Aufgabe (im Gegensatz zur iterativen Lösung mit Schleifen) zusätzlichen Arbeitsspeicher. Deshalb wird Rekursion dann angewendet, sich die Problemgrösse bei jedem Funktionsaufruf um einen Faktor verkleinert (z. B. halbiert), so dass nur relativ wenige Aufrufe der Funktion nötig sind.

Der Vorteil der Rekursion gegenüber der Iteration besteht darin, dass sich einige algorithmische Probleme damit einfacher lösen lassen.

Beispiel 7.6

```
1  def factorial(n): # Rekursive Berechnung von n! (Fakultät)
2      if n == 1:    # Base Case
3          return 1
4      else:         # Löse das nächstkleinere Problem ...
5          return n*factorial(n-1)
```

Rekursionsschema für `factorial(4)`:



Beispiel 7.7

Zum Vergleich die iterative Lösung der Fakultätsberechnung:

```
1 def factorial(n): # Iterative Berechnung von n! (Fakultät)
2     f = 1
3     for k in range(2, n+1):
4         f = k * f
5     return f
```

Man muss hier einräumen, dass auch die iterative Lösung nicht besonders schwierig zu verstehen bzw. zu programmieren ist.

Beispiel 7.8

```
1  def f(n):
2      if n < 3:
3          return 10
4      else:
5          return n + 2 * f(n-2)
6
7  print(f(6)) # =>
8              #
9              #
10             #
11             #
```

Beispiel 7.8

```
1  def f(n):
2      if n < 3:
3          return 10
4      else:
5          return n + 2 * f(n-2)
6
7  print(f(6)) # => 6 + 2 * f(4)              [Aufbau]
8              #  = 6 + 2 * (4 + 2 * f(2)) [Aufbau]
9              #  = 6 + 2 * (4 + 2 * 10)   [Base Case]
10             #  = 6 + 2 * 24             [Auflösung]
11             #  = 54                     [Auflösung]
```

Darstellung von Zeichenketten

Unter einer Zeichenkette (*string*) kann man sich eine Liste aus einzelnen Zeichen vorstellen. Wir werden gleich sehen, dass viele Funktionen, Operatoren und Methoden für Zeichenketten eine ähnliche Semantik haben wie die von Listen.

In Python werden Zeichenketten durch ein Paar von

- einfachen Anführungszeichen (`'...'`),
- doppelten Anführungszeichen (`"..."`),
- Triple Quotes (`'''...'''` oder `"""..."""`)

definiert. Während sich einfache und doppelte Anführungszeichen für kürzere Strings eignen, eignen sich Triple Quotes für mehrzeilige Texte oder Kommentare.

Indizes und Slices

Auf einzelne Zeichen oder Zeichenbereiche innerhalb eines Strings können wir wie bei Listen mit der Index- und der Slice-Syntax zugreifen.

Beispiel 8.1

```
1  s = 'Sahara'
2  print(s[0])      # =>
3  print(s[-2])     # =>
4  print(s[1:4])    # =>
5  print(s[::-1])   # =>
```

Beispiel 8.1

```
1  s = 'Sahara'
2  print(s[0])      # => 'S'
3  print(s[-2])     # => 'r'
4  print(s[1:4])    # => 'aha'
5  print(s[::-1])   # => 'arahaS' (praktisch!)
```

Escape-Sequenzen

Bestimmte Zeichen erhalten eine andere Semantik, wenn ihnen ein Backslash vorangestellt wird. Dies wird Maskieren genannt.

Escape-Sequenz	Bedeutung
`\n`	Zeilenschaltung (*newline*)
`\t`	horizontaler Tabulator
`\"`	Anführungszeichen
`\'`	Hochkomma (Apostroph)
...	...

Einfache Anführungszeichen können ohne Maskierung in einem String verwendet werden, der von doppelten Anführungszeichen umschlossen ist. Auch das Umgekehrte ist möglich.

```
1 print('Er hat nur "Hallo!" gesagt.') # => Er hat nur
    "Hallo!" gesagt.
```

Funktionen für Zeichenketten

Es seien *s* und *c* Strings, wobei *c* aus einem einzelnen Zeichen besteht.

Funktion	**Wert**
`len(`*s*`)`	Anzahl der Zeichen von *s*
`list(`*s*`)`	Liste der einzelnen Zeichen von String *s*
`int(`*s*`)`	die von *s* dargestellte ganze Zahl (wenn möglich)
`float(`*s*`)`	die von *s* dargestellte Gleitkommazahl (wenn möglich)
`ord(`*c*`)`	Unicode-Nummer des Zeichens *c* (dezimal)

Beispiel 8.2

```
1 print(len('Hello World!')) # =>
2 print(list('UHU'))         # =>
3 print(int('22') + 3)       # =>
4 print(float('22') + 3)     # =>
5 print(ord('A'))            # =>
```

Die Nummern der ersten 128 Unicode-Zeichen können in der ASCII-Tabelle am Ende dieses Kapitels nachgeschlagen werden.

Beispiel 8.2

```
1 print(len('Hello World!')) # => 12
2 print(list('UHU'))         # => ['U','H','U']
3 print(int('22') + 3)       # => 25
4 print(float('22') + 3)     # => 25.0
5 print(ord('A'))            # => 65
```

Die Nummern der ersten 128 Unicode-Zeichen können in der ASCII-Tabelle am Ende dieses Kapitels nachgeschlagen werden.

Operatoren für Zeichenketten

Es seien s, t Zeichenketten sowie n eine natürliche Zahl.

Operator	**Resultat**
s + t	Verkettung (*concatenation*) von s und t
s * n	String aus n Kopien von s

Beispiel 8.3

```
1  print('Bon' + 'bon') # =>
2  print('la' * 4)      # =>
3  print('hallo' * 0)   # =>
```

Beispiel 8.3

```
1 print('Bon' + 'bon') # => 'Bonbon'
2 print('la' * 4)      # => 'lalalala'
3 print('hallo' * 0)   # => ''
```

Methoden für Zeichenketten (Auswahl)

Im Folgenden sind s, t, u Python-Zeichenketten und L eine Liste aus Zeichenketten.

Methode	**Wert**
`s.find(t)`	Startindex von t, wenn t in s vorkommt; sonst -1
`s.replace(t,u)`	Ersetzt String t durch String u in s
`s.join(L)`	String, der die Elemente von L durch s verbindet.
`s.lower()`	String s mit Grossbuchstaben $\rightarrow$ Kleinbuchstaben
`s.upper()`	String s mit Kleinbuchstaben $\rightarrow$ Grossbuchstaben
`s.strip()`	String s ohne Whitspaces links und rechts
`s.split(t)`	Liste der Teilstrings von s, getrennt an den Stellen von t
`s.format(...)`	Fügt die Argumente bei {0}, {1}, ... in s ein.
`s.count(t)`	Zählt, wie oft der String t im String s vorkommt.

Beispiel 8.4

```
1 print('Abenteuer'.find('ente'))     # =>
2 print('Turm'.replace('T', 'W'))     # =>
3 print(' & '.join(['Anna', 'Ben'])) # =>
4 print('CH'.lower())                 # =>
5 print('ch'.upper())                 # =>
6 print('   test  \n'.strip())        # =>
7 print('20-06-2020'.split('-'))      # =>
8 print('P({1}|{0})'.format(2,5))     # =>
9 print('Mississippi'.count('s'))     # =>
```

Beispiel 8.4

```
1 print('Abenteuer'.find('ente'))     # => 2
2 print('Turm'.replace('T', 'W'))     # => Wurm
3 print(' & '.join(['Anna', 'Ben'])) # => Anna & Ben
4 print('CH'.lower())                 # => ch
5 print('ch'.upper())                 # => CH
6 print('   test  \n'.strip())        # => test
7 print('20-06-2020'.split('-'))      # => ['20','06','2020']
8 print('P({1}|{0})'.format(2,5))     # => P(5|2)
9 print('Mississippi'.count('s'))     # => 4
```

Strings sind unveränderlich

Im Gegensatz zu Listen können Strings nicht „in place“ verändert werden; d. h. es können keine einzelnen Zeichen im String-Objekt „ausgewechselt“ werden. Stattdessen erzeugen die Operationen, die scheinbar solche Manipulationen durchführen, jeweils eine neues String-Objekt.

Beispiel 8.5

```
1  s = 'Haus'
2  print(s, id(s))              # => Haus 139968772628976
3  s = s.replace('H', 'M')
4  print(s, id(s))              # => Maus 139968772676208
5  s[0] = 'R'                   # => ... TypeError
```

Unicode

Der Unicode-Standard hat den Zweck, jedem Schriftzeichen auf dieser Welt eine eindeutige Nummer zuzuordnen und definiert Codierungen, welche diese Nummern in einen Binärcode umwandeln (*Unicode Transformation Code, UTF*).

Die ASCII-Tabelle

Aus Kompatibilitätsgründen wurde der seit 1968 verwendete und weit verbreitete *American Standard Code for Information Interchange* (ASCII) in das Unicode-System integriert. Die Zeichen mit den Nummern 0–31 und 127 stellen Steuerzeichen dar.

DEC	.0	.1	.2	.3	.4	.5	.6	.7	.8	.9
00.	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT
01.	LF	VT	FF	CR	SO	SI	DLE	DC1	DC2	DC3
02.	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS
03.	RS	US	SP	!	"	#	$	%	&	’
04.	(	)	*	+	,	-	.	/	0	1
05.	2	3	4	5	6	7	8	9	:	;
06.	<	=	>	?	@	A	B	C	D	E
07.	F	G	H	I	J	K	L	M	N	O
08.	P	Q	R	S	T	U	V	W	X	Y
09.	Z	[	\	]	^	_	‘	a	b	c
10.	d	e	f	g	h	i	j	k	l	m
11.	n	o	p	q	r	s	t	u	v	w
12.	x	y	z	{	|	}	~	DEL		

Ausgabe in der Shell

`print(`*wert_1*`, `*wert_2*`, ..., sep=' ', end='\n')`

Gibt die durch Kommas getrennten Werte auf der Standardausgabe (Shell) aus.

Ohne Angabe der Parameter `sep=' '` und `end='\n'` werden die Voreinstellungen (ein Leerzeichen, eine Zeilenschaltung) verwendet.

Beispiel 9.1

```
1  print(4, 7, 9, 8, sep='+') # => 4+7+9+8
2
3  print(1, 2, 3, sep='\n')   # => 1
4                             #    2
5                             #    3
6
7  print('a', end='***')      # => a***b***c***
8  print('b', end='***')
9  print('c', end='***\n')
```

Eingaben von der Shell

var = `input(`*string*`)`

Zeigt auf der Shell die Zeichenkette *string* an und wartet, bis der Benutzer eine Eingabe gemacht und mit der ENTER-Taste abgeschlossen hat. Die Eingabe wird als String zusammen mit der Zeilenschaltung (von der Enter-Taste) der Variablen *var* zugewiesen.

Falls es sich bei der Eingabe um eine Zahl handelt, mit der gerechnet werden soll, muss sie, enteweder mit `int(..)` oder mit `float(..)` in eine ganze Zahl oder in eine Gleitkommazahl umgewandelt werden.

Beispiel 9.2

```
1  a = float(input('1. Zahl: ')) # Eingabe: 7
2  b = float(input('2. Zahl: ')) # Eingabe: 2
3  print(a + b)                  # => 9.0
```

Ausgabe in eine Datei

1. Einen *Dateideskriptor* für den Schreibzugriff öffnen:

   fd = open(*dateiname*, mode='w') ['w' steht für *write*]

2. die Ausgabe(n) in den Deskriptor schreiben:

   fd.write(*ausgabe_1*)
   fd.write(*ausgabe_2*)
   ...

3. Den Deskriptor wieder schliessen:

   fd.close()

fd	ein gültiger Bezeichner (oft fd oder fh)
dateiname	eine Zeichenkette mit dem Dateinamen
ausgabe_i	eine Ausgabe in Form einer Zeichenkette

Bemerkungen

- `open(`*`dateiname`*`, ...)` überschreibt eine existierende mit dem Namen *`dateiname`* ohne Rückfrage.
- *`fd`*`.write(...)` fügt nicht automatisch eine Zeilenschaltung an das Ende des Strings an.

Beispiel 9.3

```
fd = open('myfile.txt', mode='w')

fd.write('{0}\n'.format(1234))
fd.write('{0}\n'.format(4321))
fd.write('{0}\n'.format(2222))

fd.close()
```

Inhalt der Datei `myfile.txt`:

```
1234
4321
2222
```

Lesen aus einer Datei

1. Einen *Dateideskriptor* für den Lesezugriff öffnen:

   fd = open(*dateiname*, mode='r') ['r' steht für *read*]

2. die Ausgabe zeilenweise aus dem Deskriptor lesen und in *codeblock* verarbeiten

   ```
   for line in fd:
       codeblock
   ```

3. Den Deskriptor wieder schliessen:

   fd.close()

fd	ein Bezeichner (meist fd oder fh)
dateiname	eine Zeichenkette mit dem Dateinamen
line	ein Bezeicher für die aktuell gelesene Zeile

Beispiel 9.4 (siehe `myfile.txt` aus Beispiel 9.3)

```
1  fd = open('myfile.txt', mode='r')
2
3  s = 0
4
5  for line in fd:
6      try: # Falls möglich, ...
7          s += int(line) # mache 'line' ganzzahlig
8      except: # Geht das nicht ...
9          pass # überspringe die Zeile.
10
11 fd.close()
12
13 print(s) # =>
```

Beispiel 9.4 (siehe `myfile.txt` aus Beispiel 9.3)

```
1  fd = open('myfile.txt', mode='r')
2
3  s = 0
4
5  for line in fd:
6      try: # Falls möglich, ...
7          s += int(line) # mache 'line' ganzzahlig.
8      except: # Geht das nicht ...
9          pass # überspringe die Zeile.
10
11 fd.close()
12
13 print(s) # => 7777
```

Überblick

In Python ist ein Dictionary eine Datenstruktur, die Werte mit Hilfe (unveränderlicher) Schlüssel speichert. In Dictionaries können Werte anhand ihres Schlüssels effizient gefunden werden [$O(1)$ amortisiert]. Dafür verliert man die Ordnungstruktur der Elemente, wie man sie bei Listen kennt.

Erzeugen von Dictionaries

Eine Dictionary besteht aus Menge von Schlüssel-Wert-Paaren, wobei die Schlüssel von unveränderlichem Typ sein müsssen. Ds folgende Beispiel zeigt die Syntax zur direkten Erzeugung eines Dictionaries.

```
1 table = {'red': 'rot', 'green': 'grün'}
```

Mit `dict()` oder {} wird ein leeres Dictionary erzeugt.

Zugriff auf die Elemente

Ist *key* ein Schlüssel im Dictionary, *mydict*, so liefert uns *mydict*[*key*] den zum Schlüssel gehörenden Wert.

```
1 print(table[red]) # =>  'rot'
```

Weitere Schlüsssel-Wert-Paare hinzufügen/ändern

```
1  table['blue'] = 'blau'   # erzeugt einen neuen Eintrag
2  table['green'] = 'gruen' # ersetzt den alten Wert
```

Dictionaries durchlaufen

Dictionaries können auf verschiedene Weise mit einer `for`-Schleife durchlaufen werden.

```
1  mydict = {'c':4, 'a':5, 'b':1}
2
3  for key in mydict.keys():
4      print(key) # => c, a, b [stimmt zufällig]
5
6  for key in sorted(mydict.keys()):
7      print(key) # => a, b, c
8
9  for val in mydict.values():
10     print(val) # => 4, 5, 1
11
12 for key, value in mydict.items():
13     print(key, '->', value) # => c -> 4, a -> 5, b -> 1
```

Achtung: Die Reihenfolge, in der die Elemente ausgegeben

Schlüssel-Wert-Paare entfernen

Entweder mit `del` *dict*[*key*] oder *dict*.pop(*key*).

```
table = {'red': 'rot', 'green': 'grün'}
del table['green']
table.pop('red')
print(table) # -> {}
```

Testen, ob ein Schlüssel im Dictionary existiert

Dafür können wie bei Listen die Operatoren `in` und `not in` verwendet werden.

```
1 D = {'a':3, 'b':5, 7:'u', 'm':'k'}
2 print('b' in D) # True
3 print('u' in D) # False ('u' ist Wert und nicht Schlüssel)
```

Grösse eines Dictionaries

Die Anzahl der Schlüssel-Wert-Paare wird analog zu Listen mit `len(`*dict*`)` bestimmt.

```
1 D = {(1,2):55, (3,4):817, (5,7):27} # schwach besetzte
      Matrix
2 print(len(D)) # => 3
```

Dictionary Comprehensions

Analog zu den Listen kann man auch Dictionaries „von innen heraus“ erzeugen. Die Syntax ist intuitiv, sofern man nicht komplexere Aufgaben damit bewältigen will.

```
1  D1 = {x:0 for x in 'abcd'}
2  print(D1) # => {'a': 0, 'b': 0, 'c': 0, 'd': 0}
3
4  D2 = {x:y for (x,y) in zip(['a', 'b', 'c'], [1, 2, 3])}
5  print(D2) # => {'a': 1, 'b': 2, 'c': 3}
```

`zip()` bildet aus n Listen mit jeweils m Elementen jeweils m Listen mit jeweils n Elementen.

Eine Klasse ist ein massgeschneiderter Datentyp und gleichzeitig ein Bauplan für Objekte dieses Datentyps. In Python definiert man eine Klasse mit dem Schlüsselwort `class`:

```
class MyClass:
    <klassenrumpf>
```

Der Klassenname wird üblicherweise mit einem Grossbuchstaben begonnen. Der Klassenrumpf enthält Variablenzuweisungen und Funktionsdefinitionen; er darf aber auch nur aus einer `pass`-Anweisung bestehen.

Nachdem eine Klasse so definiert wurde, können Instanzen (Objekte) dieser Klasse erzeugt werden, indem man den Klassennamen wie eine Funktion aufruft.

```
x = MyClass():
```

Einer Instanz kann man mit der Punkt-Schreibweise individuelle Variablen (Attribute, Eigenschaften, Instanzvariablen) zuordnen. Mit der gleichen Notation kann auch auf die Werte dieser Variablen zugegriffen werden.

```python
class Kreis:
    pass

a = Kreis()
a.radius = 5

b = Kreis()
b.radius = 9

print(a) # <__main__.Kreis object at 0xb71c5c4c>
print(b) # <__main__.Kreis object at 0xb71c5ccc>

print(a.radius) # => 5
print(b.radius) # => 9
```

Jede Instanz verfügt also über einen separaten Speicherort, den die `print`-Funktion anzeigt.

Statt aber jeder Instanz manuell Variablen und Werte zuzuweisen, kann man der Klassendefinition eine Initialisierungsfunktion hinzufügen, um diese Arbeit automatisch auszuführen.

Im Jargon der objektorientierten Programmierung spricht man aber nicht mehr von Funktionen sonder von Methoden, wenn eine Funktion zu einem Objekt gehört.

Daher spricht man von einer Initialisierungsmethode bzw. einem Konstruktor.

```python
class Kreis:

    def __init__(self, r):
        self.radius = r

a = Kreis(5)
b = Kreis(9)

print(a.radius) # => 5
print(b.radius) # => 9
```

Nach Konvention ist `self` immer das erste Argument eines Konstruktors. Weitere Parameter (im Beispiel: `r`) können folgen, um Instanzvariablen mit Anfangswerten zu versehen.

Eine Methode ist eine Funktion, die zu einer bestimmten Klasse gehört. Neben der speziellen Methode `__init__(...)`, die bei jeder neu erzeugten Instanz als erstes ausgeführt wird, lassen sich auch eigene Methoden definieren.

```
1  class Kreis:
2
3      def __init__(self, r):
4          self.radius = r
5
6      def flaeche(self):
7          return 3.141 * self.radius**2
8
9  c = Kreis(10)
10
11 print(c.flaeche()) # => 314.1
```

Soll eine Methode auf die jeweilige Instanz angewendet werden, so steht der erste Parameter für das Objekt selbst. Üblicherweise wird dieser Parameter `self` genannt. Damit kann man innerhalb der Klassendefinition auf alle Eigenschaften und Methoden des Objekts zugreifen.

Wie bei den Instanzvariablen, werden Methoden auf Instanzen angewendet, indem man sie mit der Punkt-Notation an den Instanznamen bindet.

Soll eine Variable für die gesamte Klasse (unabhängig von einer Instanz) denselben Wert haben, so muss sie ausserhalb der `__init__`-Methode definiert werden.

```
1  class Kreis:
2
3      pi = 3.141
4
5      def __init__(self, r):
6          self.radius = r
7
8      def flaeche(self):
9          return Kreis.pi * self.radius**2
10
11 print(Kreis.pi)
```

Um auf den Wert einer Klassenvariable zuzugreifen, muss ihr jeweils der Klassenname vorangestellt werden.

Etwas unschön ist der Umstand, dass Python bei einer Instanzvariablen auf eine gleichnamigen Klassenvariable zugreift, wenn die Instanzvariablen nicht definiert ist:

```
1  class Kreis:
2
3      pi = 3.141
4
5      def __init__(self, r):
6          self.radius = r
7
8      def flaeche(self):
9          return Kreis.pi * self.radius**2
10
11 c = Kreis(10)
12 print(c.pi) # => 3.141
```

Soll eine Methode unabhängig von den Instanzen, d. h. für die ganze Klasse gelten, so muss der erste Parameter `self` weggelassen werden. Die Klassenmethode wird dafür mit dem Klassen- statt dem Objektnamen (und den jeweiligen Parametern) aufgerufen.

```python
import random

class Kreis:

    # Kreis mit Zufallsradius
    def zufall():
        r = random.randint(1,20)
        return Kreis(r)

    def __init__(self, r):
        self.radius = r

a = Kreis.zufall()
print(a.radius) # => (pseudo-)zufaelliger Radius
```

Das Konzept der Vererbung ermöglicht es, Variablen und Methoden von Klassen in anderen Klassen wiederzuverwenden. Das folgende Beispiel zeigt den Fall, bei dem ein Objekt ein Spezialfall eines allgemeineren Objekts ist

```python
class Rechteck:

    def __init__(self, a, b):
        self.a = a
        self.b = b

    def flaeche(self):
        return self.a * self.b

class Quadrat(Rechteck):

    def __init__(self, a):
        super().__init__(a, a)

q = Quadrat(3)
print(q.flaeche()) # => 9
```

Da ein Quadrat eine spezielles Rechteck mit zwei gleich langen Seiten ist, muss dafür kein neuer Code geschrieben werden. Stattdessen wird mit der Methode `super()` auf den Konstruktor der Elternklasse zugegriffen. Da es sich in dieser Zeile um die Anwendung und nicht um die Definition des Konstruktors handelt, darf hier kein `self` stehen.

Neben dem Konstruktor gibt es noch andere spezielle Methoden, um den Code besser lesbar zu machen. Beispielsweise um die Objekte mit den bestehenden Funktionen und Operatoren zu verarbeiten.

```
1  class Vektor:
2
3      def __init__(self, x, y):
4          self.x = x
5          self.y = y
6
7      def __str__(self):
8          return '({0.x}, {0.y})'.format(self)
9
10     def __add__(self, other):
11         return Vektor(self.x+other.x, self.y+other.y)
12
13     def __eq__(self, other):
14         return (self.x==other.x and self.y==other.y)
```

```
1  a = Vektor(2,1)
2  b = Vektor(3,-4)
3
4  print(a)       # => (2, 1)
5  print(a+b)     # => (5, -3)
6  print(a==b)    # => False
```

Klasse: Bauplan für Objekte (abstrakter Datentyp)

Instanz: Objekt, das zur Laufzeit des Programms erzeugt wird.

Konstruktor: Methode, mit der (in Python) ein Objekt (Instanz) initialisiert wird.

Instanzvariable/Attribut: Variable, deren Wert von Instanz zu Instanz verschieden sein kann.

Instanzmethode: Funktion, die auf eine Instanz angewendet wird.

Klassenvariable: Variable, deren Wert unabhängig von einer Instanz ist.

Klassenmethode: Funktion, die unabhängig von einer Instanz ist.

Vererbung: Ein Konzept, das darin besteht, neue Klassen (Sub- oder Kindklasse) aus vorhandenen Klassen (Super- oder Elternklasse) abzuleiten. Die von der Superklasse geerbten Eigenschaften und Methoden können beibehalten oder überschrieben (abgeändert) werden.

Überladen von Operatoren Ein Mechanismus, der es erlaubt, den gleichen Operator (den gleichen Funktionsnamen) in unterschiedlichen Kontexten zu verwenden. Beispielsweise verwendent Python den +-Operator für das Addieren von ganzen Zahlen und das Verketten von Strings.