



Einführung in die Programmierung

Prof. Dr. Bertrand Meyer

Lektion 8: Kontrollstrukturen

In dieser (Doppel-)Lektion

- Der Begriff des Algorithmus
- Grundlegende Kontrollstrukturen: Sequenz (*sequence*, *compound*), Konditional (*conditional*), Schleife (*loop*)
- Bedingungsanweisungen: Der Konditional und seine Variante
- Operationen wiederholen: Die Schleife
- Schleifen als Annäherungsstrategie: Die Schleifeninvariante
- Was braucht es, um sicherzustellen, dass eine Schleife terminiert?
- Kontrollstrukturen auf einer tieferen Ebene: “Goto” und Flussdiagramme (flowcharts); siehe Argument für die „Kontrollstrukturen der strukturierten Programmierung“
- Das **Entscheidungsproblem** und die Unentscheidbarkeit der Programm-Terminierung

Bitte lesen Sie Kapitel 8 von *Touch of Class*



Ihre Chefin gibt Ihnen den Quellcode eines C-Compiler

- Beachten Sie, dass der Compiler mehr als ein Übersetzer in Maschinencode ist. Er ist auch ein “Prüfer” (verifier)
- Er prüft viele Eigenschaften, z.B. Typ-Eigenschaften, und weist Programme ab, wenn sie diese Eigenschaften nicht erfüllen.

Ihre Chefin verlangt, dass Sie neue solche Überprüfungen hinzufügen; spezifisch will sie, dass der Compiler analysiert, ob das Programm zur Laufzeit immer terminiert

Ihre Antwort:

- 1. Ja, sehr einfach, das kann ich schaffen.
- 2. Ich kann es tun, es wird aber schwer, ich brauche mehrere Wochen oder möglicherweise ein paar Monate.
- 3. Es ist nicht möglich für C, es wäre jedoch möglich für Java.
- 4. Es ist nicht möglich für C, es wäre jedoch möglich für Eiffel.
- 5. Sorry, Chef, ich kann nicht helfen.



Allgemeine Definition:

Ein **Algorithmus** ist die Spezifikation eines Prozesses, der von einem Berechner ausgeführt wird

PREPARAZIONE E TEMPI DI COTTURA ZUBEREITUNG - PREPARATION

Versate le verdure ancora surgelate in 1 litro abbondante d'acqua fredda con 2 cucchiaini d'olio, salate e cuocete secondo i tempi indicati.

Tiefgefrorene Gemüse in einen Liter kaltes Wasser geben, 2 Esslöffel Öl und Salz hinzufügen.

Verser les légumes surgelés dans 1 litre d'eau froide, ajouter deux cuillers à soupe d'huile et du sel.



5 Eigenschaften eines Algorithmus



- 1. Definiert die Daten, auf die der Prozess angewandt wird
- 2. Jeder elementare Schritt wird aus einer Menge von genau definierten Aktionen ausgewählt
- 3. Beschreibt die Reihenfolge der Ausführung dieser Schritte
- 4. Eigenschaften 2 und 3 basieren auf genau festgelegten, für ein automatisches Gerät geeignete Konventionen
- 5. Terminiert für alle Daten nach endlich vielen Schritten

Algorithmus vs. Programm



“Algorithmus” bezeichnet allgemein einen abstrakteren Begriff, unabhängig von der Plattform, der Programmiersprache, etc.

In der Praxis ist der Unterschied jedoch eher gering:

- Algorithmen brauchen eine präzise Notation
- Programmiersprachen werden immer abstrakter

Aber:

- In Programmen sind Daten (-objekte) genauso wichtig wie Algorithmen
- Ein Programm beinhaltet typischerweise **viele** Algorithmen und Objektstrukturen

Aus was ein Algorithmus besteht



Grundlegende Schritte:

- Featureaufruf $x.f(a)$
- Zuweisung
- ...



(Eigentlich nicht
viel mehr)

Abfolge dieser grundlegenden Schritte:

KONTROLLSTRUKTUREN



Definition: Ein Programmkonstrukt, welches den Ablauf von Programmschritten beschreibt

Drei fundamentale Kontrollstrukturen:

- Sequenz
- Schleife
- Konditional

Diese sind die

“Kontrollstrukturen des *strukturierten Programmierens*”



Sequenz: „Um von **C** aus **A** zu erreichen, erreiche zuerst das Zwischenziel **B** von **A** aus, und dann **C** von **B** ausgehend“

Schleife: „Löse das Problem mithilfe von aufeinanderfolgenden Annäherungen der Input-Menge“

Konditional: „Löse das Problem separat für zwei oder mehrere Teilmengen der Input-Menge“

Die Sequenz (auch: Verbund (*Compound*))

*Instruktion*₁

*Instruktion*₂

...

*Instruktion*_n

Eiffel: Das Semikolon als optionale Trennung



*Instruktion*₁ 

*Instruktion*₂ 

... 

*Instruktion*_n

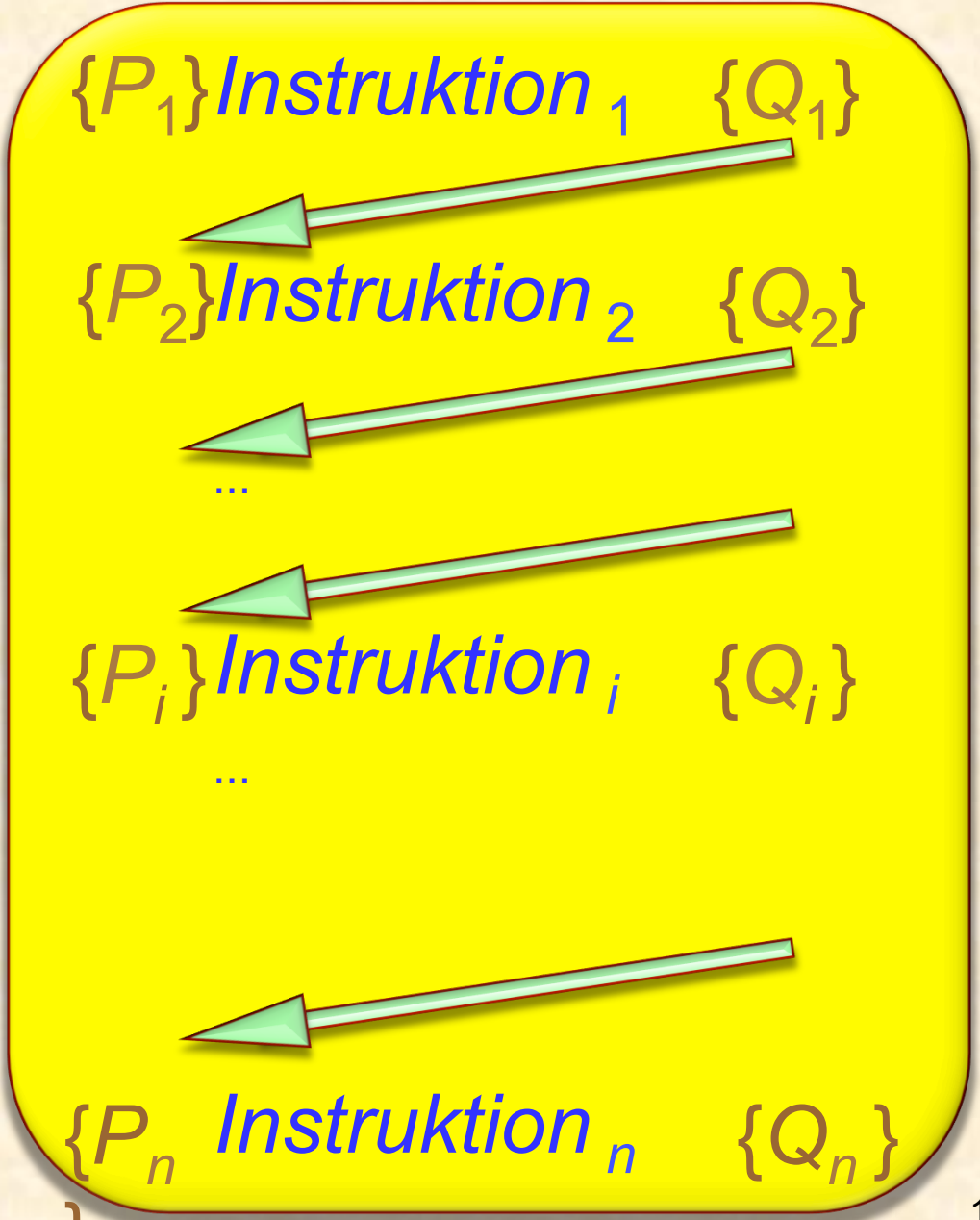
Korrektheit eines Verbunds



Die Vorbedingung von *Instruktion*₁ muss zu Beginn erfüllt sein

Die Nachbedingung von *Instruktion*_{*j*} muss die Vorbedingung von *Instruktion*_{*i+1*} implizieren

Das Schlussresultat ist die Nachbedingung von *Instruktion*_{*n*}



Konditional (Bedingte Instruktion)



```
if
    Bedingung                -- Boole'scher Ausdruck
then
    Instruktionen           -- Verbund
else
    Andere_Instruktionen    -- Verbund
end
```



```
if
    a > b
then
    max := a
else
    max := b
end
```

Als Feature (Abfrage)

In einer beliebigen Klasse:

```
greater (a, b : INTEGER): INTEGER
```

```
-- Das Maximum von a und b.
```

```
do
```

```
if
```

```
    a > b
```

```
then
```

```
    Result := a
```

```
else
```

```
    Result := b
```

```
end
```

```
end
```




$i, j, k, m, n: \text{INTEGER}$

...

$m := \text{greater}(25, 32)$

$n := \text{greater}(i + j, k)$



In einer beliebigen Klasse:

greater (a, b : INTEGER): INTEGER

-- Das Maximum von *a* und *b*.

do

if

a > b

then

Result := a

else

Result := b

end

end



In einer Klasse *DATE*:

later (*d* : *DATE*): *DATE*

-- Das Späteste von *Current* und *d*.

do

if

d > **Current**

then

Result := *d*

else

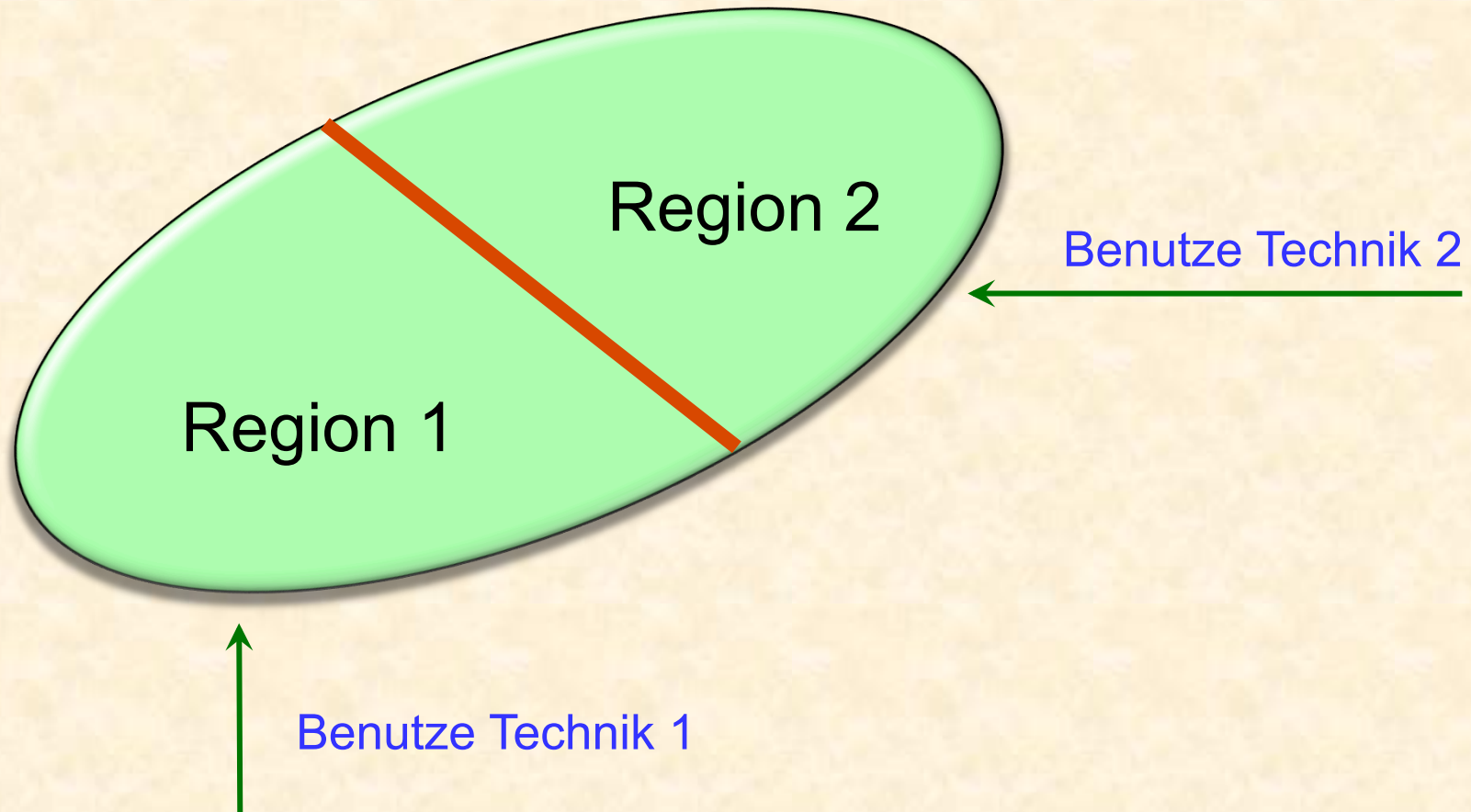
Result := **Current**

end

end



PROBLEMRAUM





if *Bedingung* **then**

Instruktionen

else

andere_Instruktionen

end



```
if Bedingung then  
    Instruktionen  
end
```

Ist semantisch äquivalent zu

```
if Bedingung then  
    Instruktionen
```

```
else
```

```
end
```



Leere Klausel



later (*d* : *DATE*): *DATE*

-- Das Späteste von *Current* und *d*.

do

if *d* > **Current** **then**

Result := *d*

else

Result := **Current**

end

end



later (*d* : *DATE*): *DATE*

-- Das Späteste von *Current* und *d*.

do

Result := Current

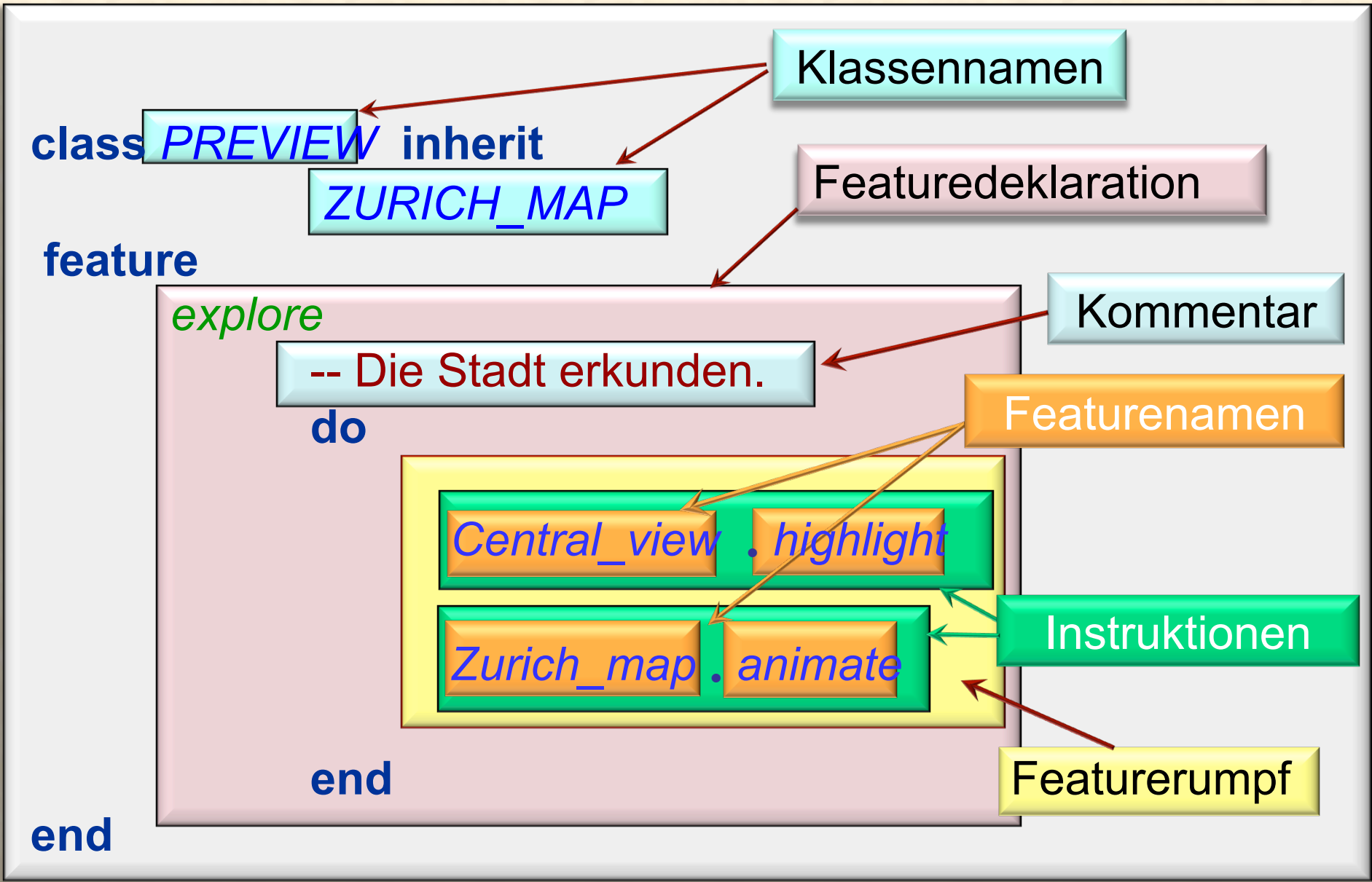
if *d* > **Current** **then**

Result := *d*

end

end

Erinnerung: Verschachtelung* (von Lektion 3)



*Engl: *Embedding*



```
if Bedingung1 then  
    Instruktionen1  
else  
    if Bedingung2 then  
        Instruktionen2  
    else  
        if Bedingung3 then  
            Instruktionen3  
        else  
            if Bedingung4 then  
                Instruktionen4  
            else  
                ...  
            end  
        end  
    end  
end
```

Eine verschachtelte Struktur



Eine Kamm-ähnliche Struktur



Verschachtelung von bedingten Instruktionen

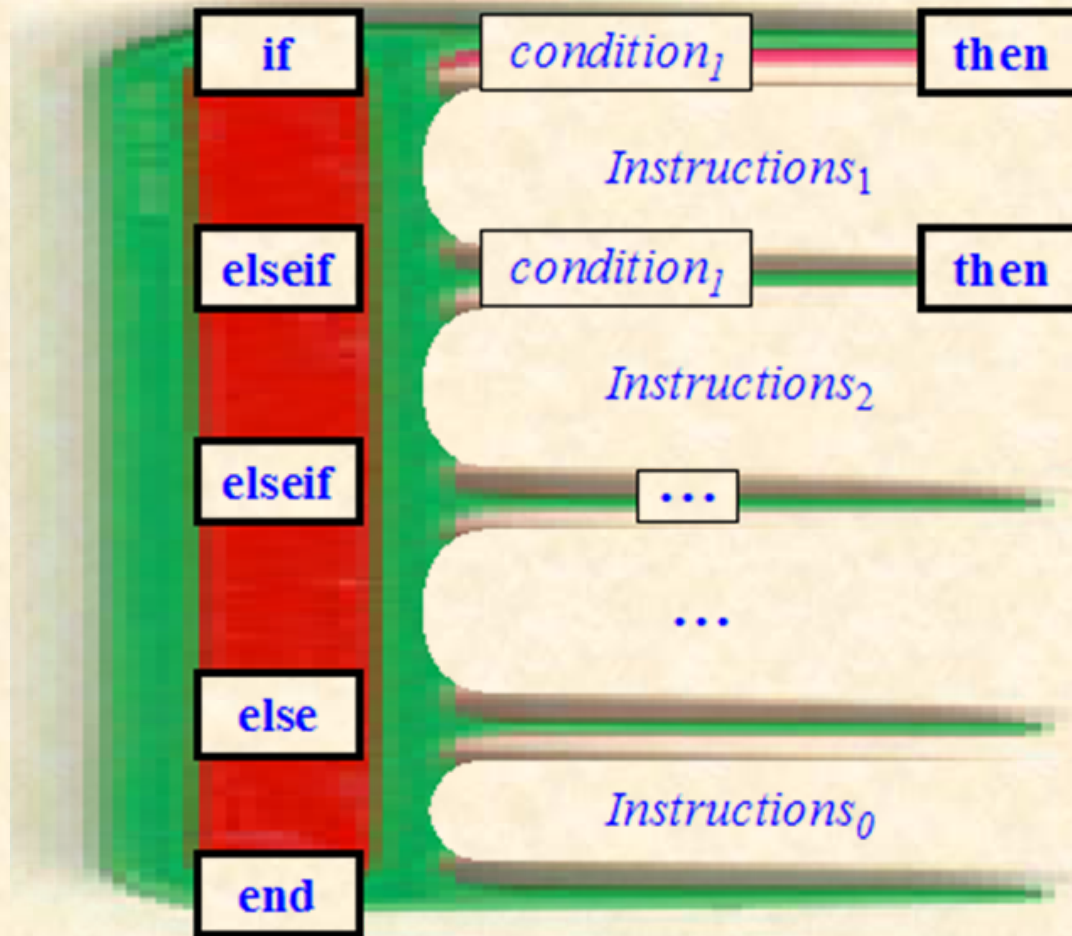


```
if Bedingung1 then  
    Instruktionen1  
else  
    if Bedingung2 then  
        Instruktionen2  
    else  
        if Bedingung3 then  
            Instruktionen3  
        else  
            if Bedingung4 then  
                Instruktionen4  
            else  
                ...  
            end  
        end  
    end  
end
```



```
if Bedingung1 then  
    Instruktionen1  
elseif Bedingung2 then  
    Instruktionen2  
elseif Bedingung3 then  
    Instruktionen3  
elseif  
    ...  
else  
    Instruktionen0  
end
```

Eine Kamm-ähnliche Struktur



Auch in Eiffel: «Inspect» (Multi-branch)



inspect

eingabe

CHARACTER
oder INTEGER

when “E” then

Instruktionen₁

when “K” then

Instruktionen₂

...

else

Instruktionen₀

end



- Schleifen und ihre Invarianten
- Was braucht es, um sicherzustellen, dass eine Schleife terminiert?
- Ein Blick auf das allgemeine Problem der Schleifen- und Programmterminierung
- Kontrollstrukturen auf einer tieferen Ebene: “Goto” und Flussdiagramme (flowcharts); siehe Argument für die „Kontrollstrukturen der strukturierten Programmierung“
- Die Unentscheidbarkeit des Entscheidungsproblems beweisen



from

Initialisierung

-- Verbund

until

Abbruchbedingung

-- Boole'scher Ausdruck

loop

Rumpf

-- Verbund

end

Die volle Form der Schleife



from

Initialisierung

-- Verbund

invariant

invarianter Ausdruck

-- Boole'scher Ausdruck

variant

varianter Ausdruck

-- Integer-Ausdruck

until

Abbruchbedingung

-- Boole'scher Ausdruck

loop

Rumpf

-- Verbund (Schleifenrumpf)

end

Eine andere Schleifensyntax

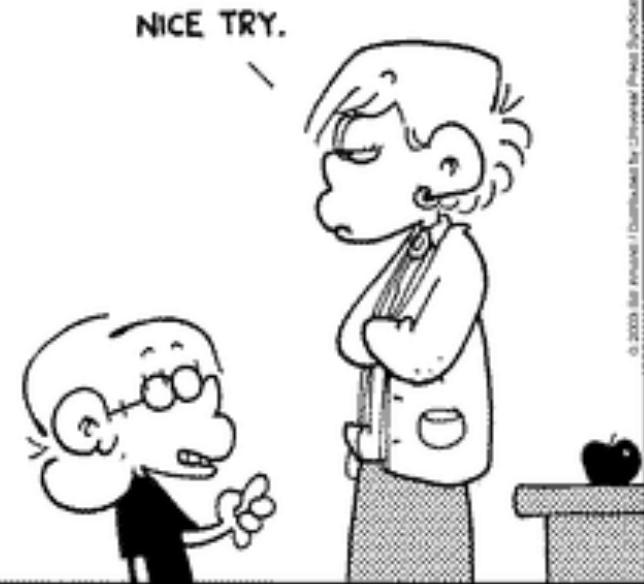


```
#include <stdio.h>
int main(void)
{
    int count;

    for (count = 1; count <= 500; count++)
        printf("I will not throw paper airplanes in class.");

    return 0;
}
```

AVENUE 10.3



© 2003 by author / Distributed by Chronicle Press Books

Schleifenformen (in verschiedenen Sprachen)

from **-- Eiffel**
Initialiesirung
until
Exit_Bedingung
loop
Rumpf
end

E

W

while *Bedingung* **do**
Rumpf
end

across **-- Eiffel**
Struktur as var
loop
Rumpf -- Auf var
end

A

R

repeat
Rumpf
until
Exit_Bedingung
end

for *i : a..b* **do**
Rumpf
end

F

C

for (*Initialiesirung; Bedingung; Instruktion*) **do**
Rumpf
end

In Eiffel (volle Form)



from

Initialisierung

-- Verbund

invariant

invarianter Ausdruck

-- Boole'scher

Ausdruck

variant

varianter Ausdruck

-- Integer-Ausdruck

until

Abbruchbedingung

-- Boole'scher Ausdruck

loop

Rumpf

-- Verbund (Schleifenrumpf)

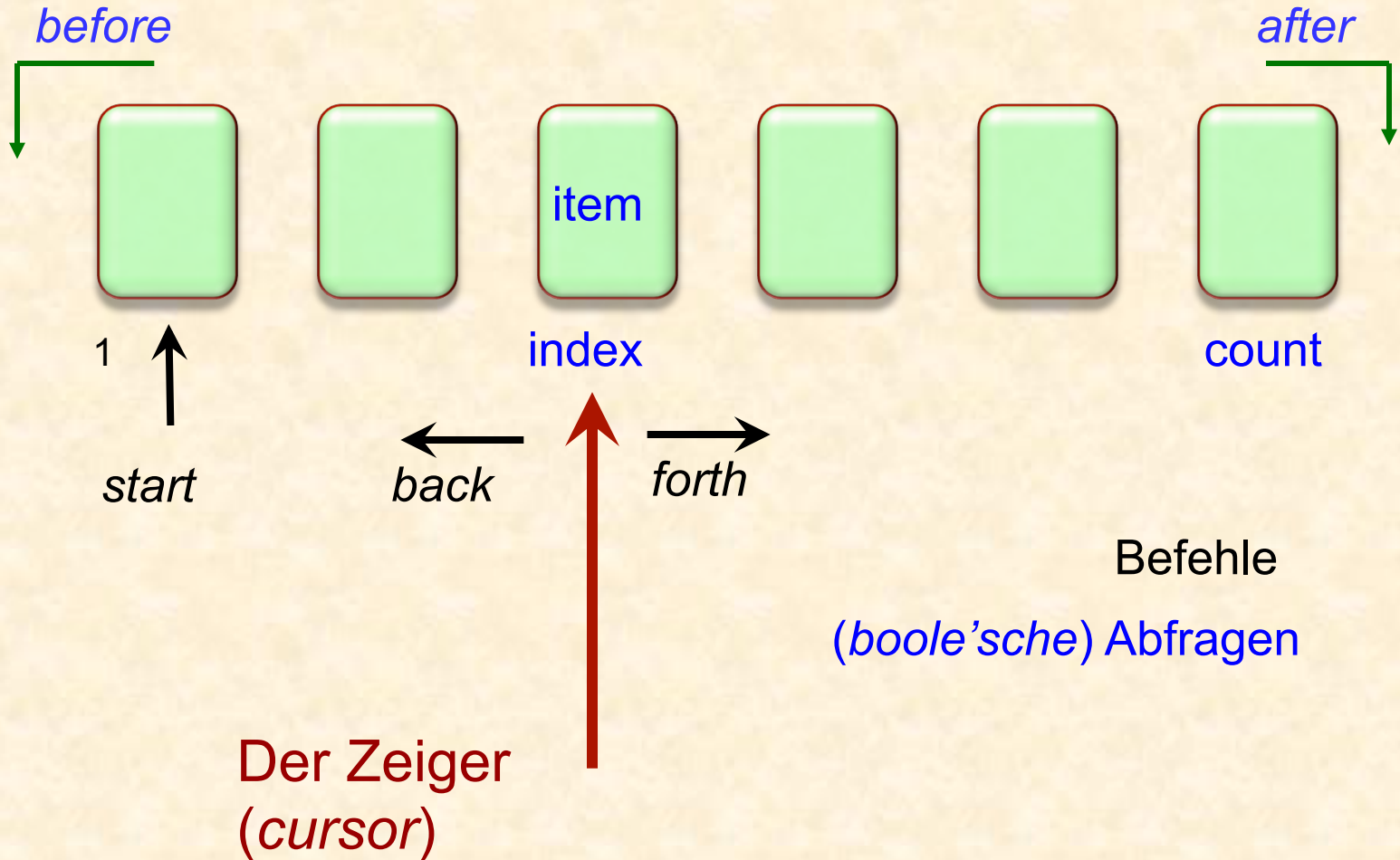
end

Über Stationen einer Linie iterieren („loopen“)

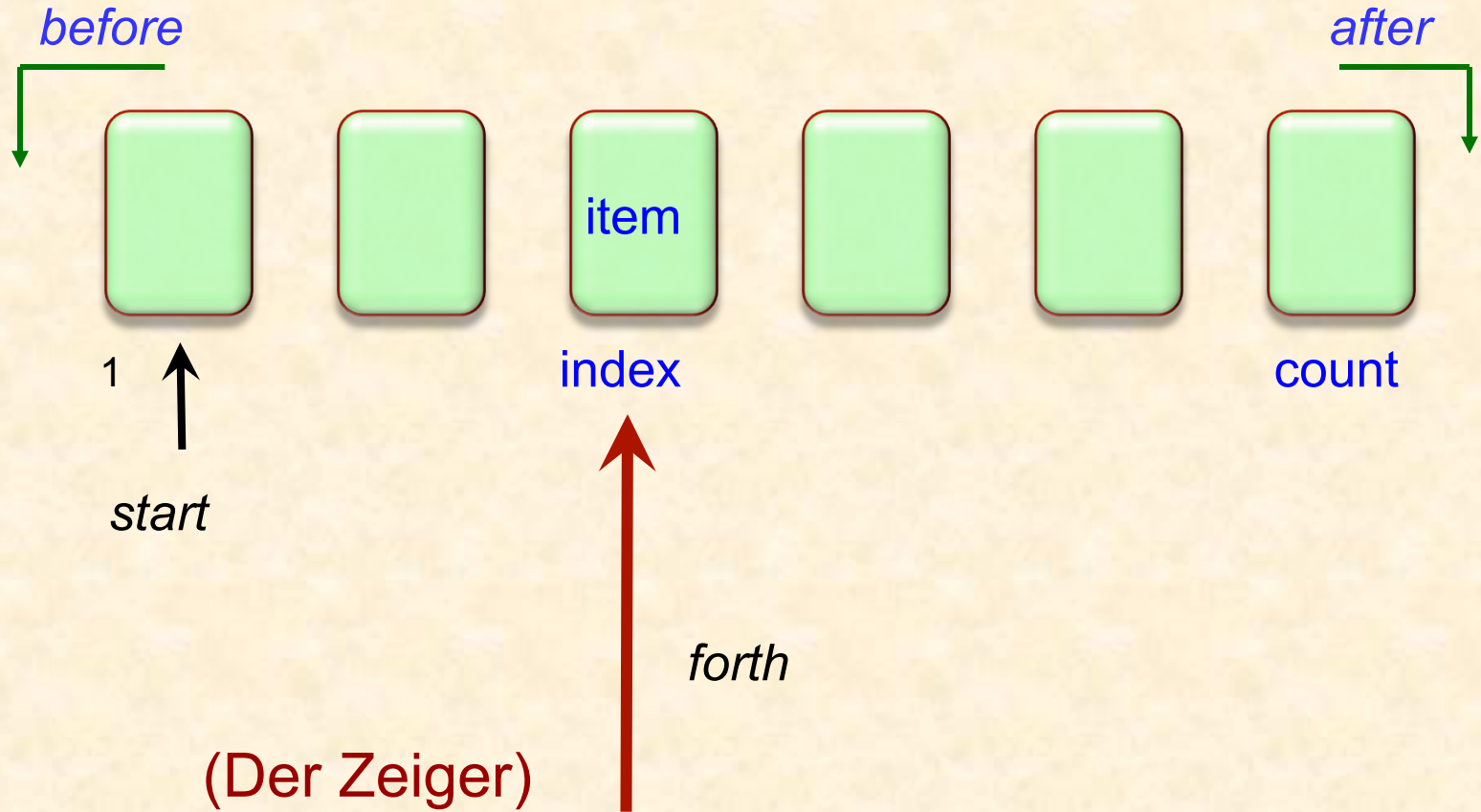


```
from      Line8.start  
until    Line8.after  
loop  
    -- “Tu was mit Line8.item.”  
    Line8.forth  
end
```

Auf eine Liste anwendbare Operationen



Auf eine Liste anwendbare Operationen



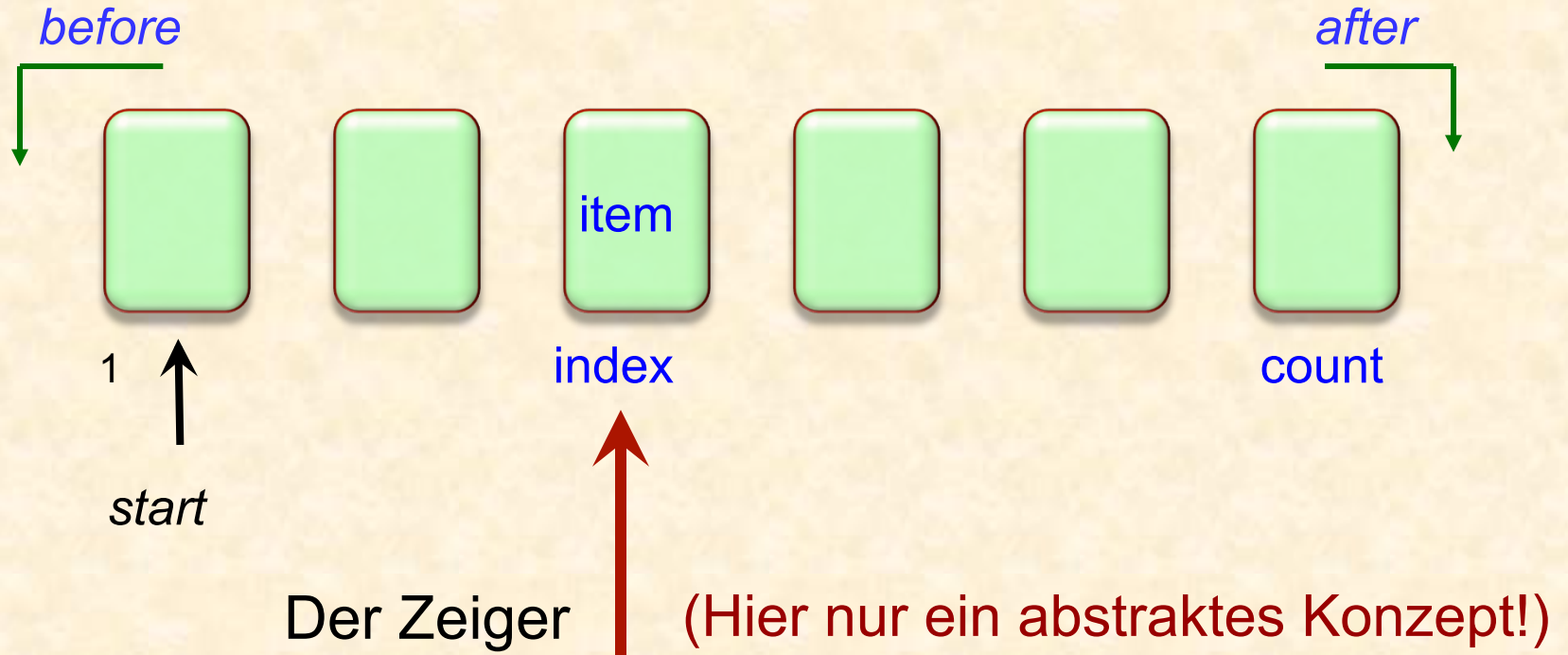
Das Problem mit internen Zeigern

```
has_duplicates: BOOLEAN  
  -- Hat Linie 8 Duplikate?  
local  
  s: STATION  
do  
  from  
    Line8.start  
  until  
    Line8.after or Result  
  loop  
    s := Line8.item  
    Line8.forth  
    -- Überprüfen, ob s nochmals in der Linie vorkommt:  
    Line8.search_forth (s)  
    Result := not Line8.after  
  end  
end
```

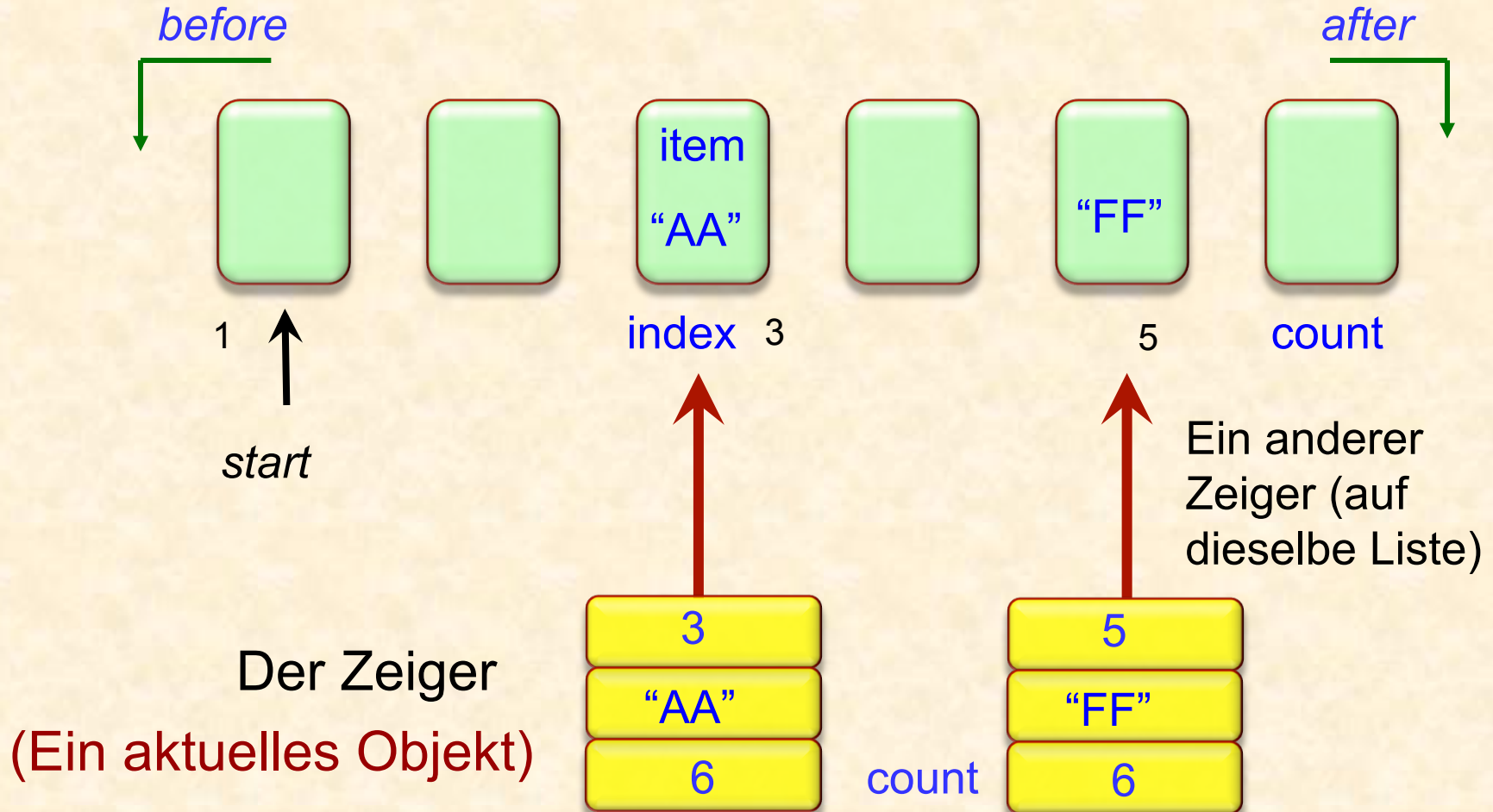
Die Zeigerposition muss immer gespeichert und wiederhergestellt werden

search verändert den Zeiger ebenfalls!

Listen mit internem Zeiger



Listen mit externem Zeiger



Über Stationen einer Linie iterieren (1)



Mit internem Zeiger (Erinnerung):

```
from      Line8.start  
until    Line8.after  
loop  
    -- "Tu was mit Line8.item."  
    Line8.forth  
end
```

Über Stationen einer Linie iterieren (2)

Mit externem Zeiger:

local

Die Zeiger-Variable

c : **like** *Line8.new_cursor*

Derselbe Typ wie
Line8.new_cursor

do

from

c := Line8.new_cursor

Ein neu erzeugter
Zeiger, der auf das
erste Element zeigt

until

c.after

loop

-- "Tu was mit *c.item*"

c.forth

end

end

Über Stationen einer Linie iterieren (3)



across

Line8 as *c*

loop

-- "Tu was mit *c.item*"

end

Die gleiche Wirkung wie (2), aber kürzer!

Die Stationsnamen anzeigen

across

Line8 as c

loop

-- Den Namen der aktuellen Station anzeigen.

console.output (c.item)

end

Ein anderes Beispiel



-- Alle Anschluss-Stationen der Linie 8 anzeigen:

across

Line8 as c

loop

if *c.item.is_exchange* **then**

console.output (c.item)

Zurich_map.station_view (c.item.name).highlight

end

end

Das „Maximum“ der Stationsnamen berechnen



from

Result := ""

across

Line8 as *c*

loop

Result := *greater* (Result, *c.item.name*)

end

Das (alphabetische) Maximum zweier Zeichenketten berechnen, z.B.

greater ("ABC ", "AD ") = "AD"

„Maximum“ zweier Zeichenketten



greater (*a*, *b* : *STRING*): *STRING*

-- Das Maximum von *a* und *b*.

do

if

a > *b*

then

Result := *a*

else

Result := *b*

end

end

In einem Feature



```
highest_name: STRING
```

```
-- Alphabetisch grösster Stationsname der Linie.
```

```
do
```

```
  from
```

```
    Result := ""
```

```
  across
```

```
    Line8 as c
```

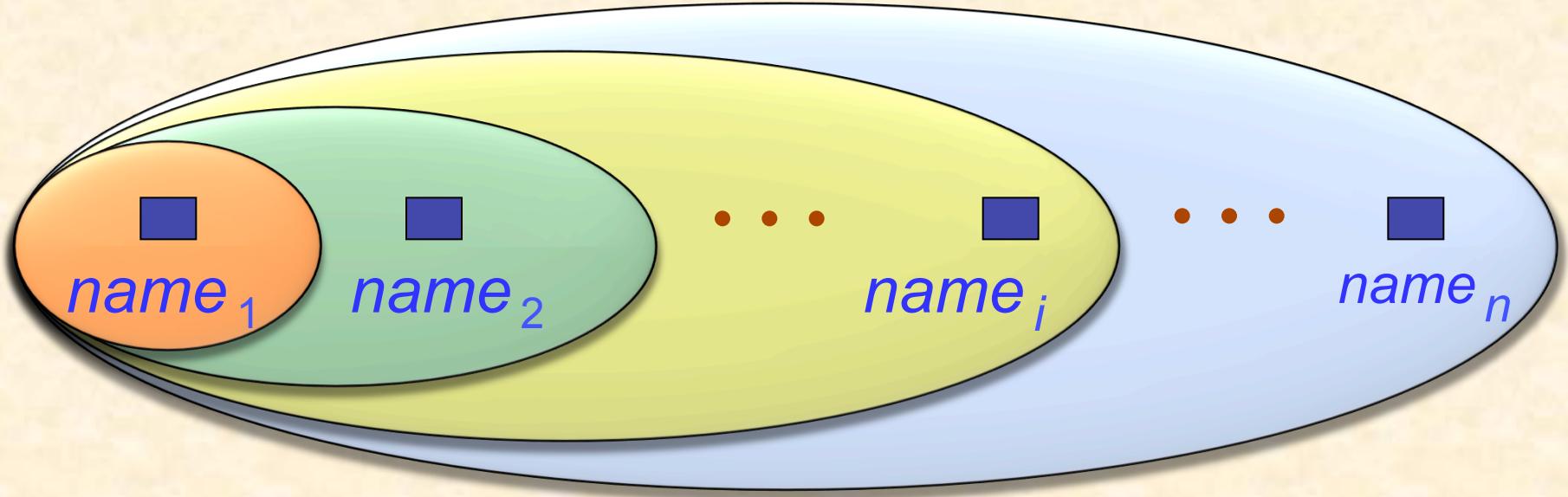
```
  loop
```

```
    Result := greater (Result, c.item.name)
```

```
  end
```

```
end
```

Schleifen als Annäherungsstrategie



Result = $name_1 = \text{Max}(\text{names}_{1..1})$

Result = $\text{Max}(\text{names}_{1..2})$

Teil

Result = $\text{Max}(\text{names}_{1..i})$

Result = $\text{Max}(\text{names}_{1..n})$

Schleifenrumpf:

```
i := i + 1  
Result := greater  
          (Result, c.item.name)
```

Nachbedingung?

highest_name: STRING

-- Alphabetisch grösster Stationsname der Linie.

do

from

Result := ""

across

Line8 as c

loop

Result := greater (Result, c.item.name)

end

ensure

Result /= Void

*-- **Result** ist der alphabetisch grösste Stationsname*

-- der Linie

end



Das „Maximum“ der Stationsnamen berechnen



from

c := Line8.new_cursor ; **Result** := ""

until

c.after

loop

Result := *greater* (**Result**, *c.item.name*)

c.forth

ensure

-- **Result** ist der alphabetisch grösste Stationsname der Linie

end

Die Schleifeninvariante



from

c := Line8.new_cursor ; Result := ""

until

c.after

invariant

c.index >= 1

c.index <= Line8.count+ 1

-- **Result** ist der alphabetisch grösste Name aller
-- bisherigen Stationen

loop

Result := *greater (Result, c.item.name)*

c.forth

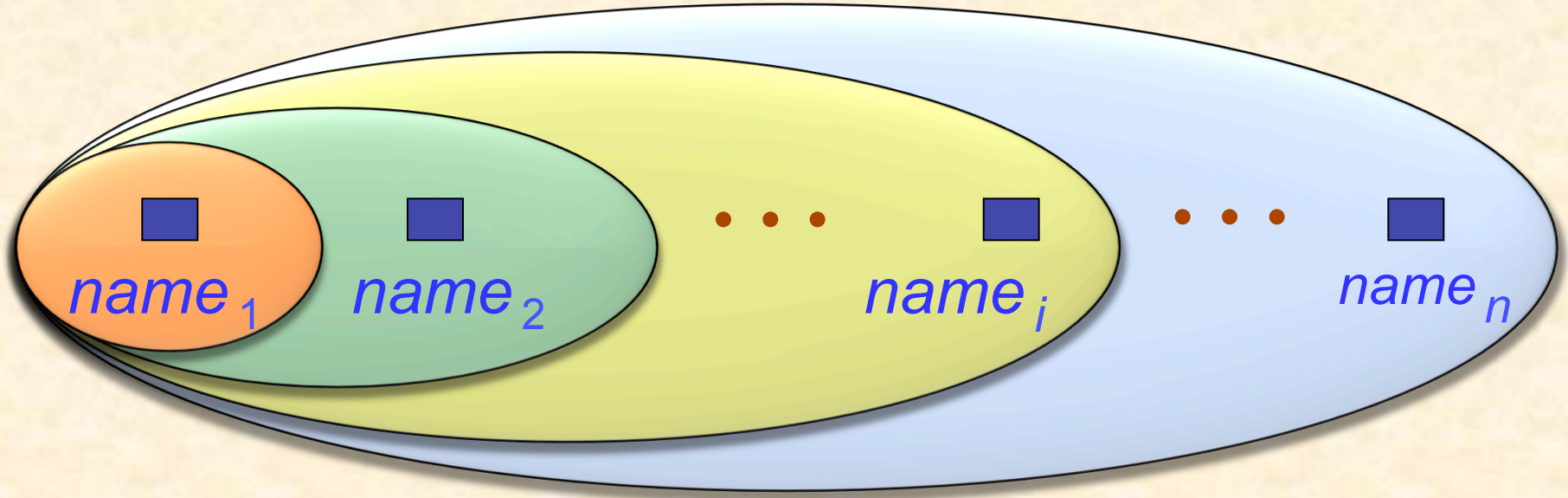
ensure

-- **Result** ist der alphabetisch grösste Stationsname der Linie

end

Fehlt etwas?

Schleifen als Annäherungsstrategie



Result = $name_1 = \text{Max}(\text{names}_{1..1})$

Result = $\text{Max}(\text{names}_{1..2})$

Result = $\text{Max}(\text{names}_{1..i})$

Schleifeninvariante

Schleifenrumpf:

```
 $i := i + 1$   
Result := greater  
          (Result,  $c.item.name$ )
```

Result = $\text{Max}(\text{names}_{1..n})$

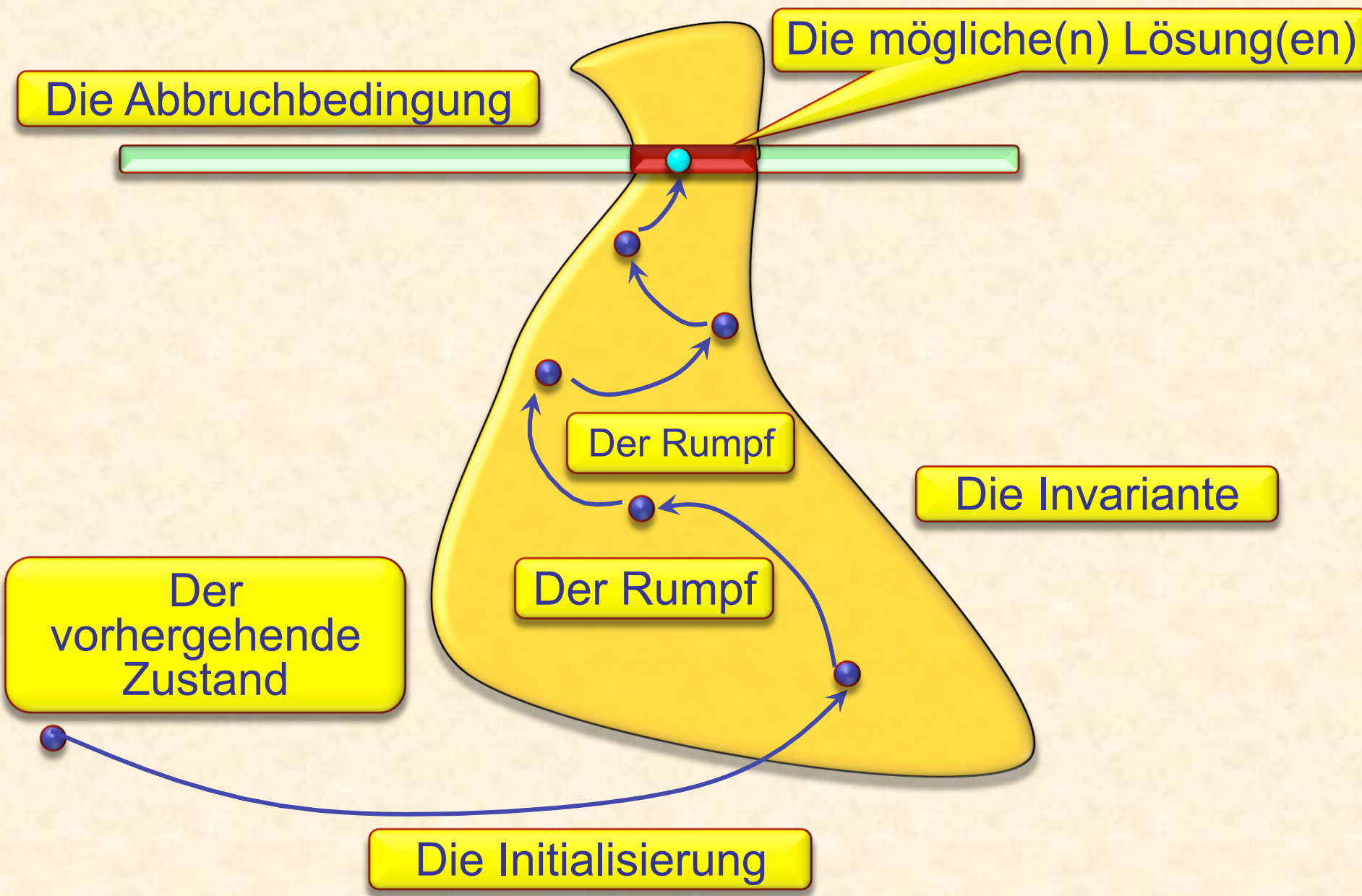


(Nicht zu verwechseln mit der Klasseninvariante)

Eine Eigenschaft, die:

- Nach der Initialisierung (**from**-Klausel) erfüllt ist
- Von jedem Schleifendurchlauf (**loop**-Klausel), bei der die Abbruchbedingung (**until**-Klausel) *nicht* erfüllt ist, eingehalten wird
- Wenn die Abbruchbedingung erfüllt ist, das gewünschte Ergebnis sicherstellt

Die Schleifeninvariante



Die Schleifeninvariante

from

```
c := Line8.new_cursor ; Result := ""
```

until

```
c.after
```

invariant

```
c.index >= 1
```

```
c.index <= Line8.count
```

```
-- Result ist der alphabetisch grösste Name aller  
-- bisherigen Stationen
```

loop

```
Result := greater (Result, c.item.name)
```

```
c.forth
```

ensure

```
-- Result ist der alphabetisch grösste Stationsname der Linie
```

end

Result = Max ($names_{1..i}$)

Die Schleifeninvariante

from

c := Line8.new_cursor; **Result** := ""

until

c.after

invariant

c.index >= 1

c.index <= *Line8.count*

-- **Falls es bisherige Stationen gibt**, ist

-- **Result** der alphabetisch grösste ihrer Namen.

loop

Result := *greater* (**Result**, *c.item.name*)

c.forth

ensure

-- **Result** ist der alphabetisch grösste Stationsname, falls es

-- eine Station gibt.

end

Result = Max (*names*_{1..i})



Der Effekt einer Schleife



from

```
c := Line8.new_cursor ; Result := ""
```

Invariante nach der Initialisierung erfüllt.

invariant

```
c.index >= 1  
c.index <= Line8.count + 1
```

-- **Result** ist der grösste der bisherigen Namen.

until

```
c.after
```

Abbruchbedingung am Ende erfüllt

Invariante nach jedem Durchlauf erfüllt.

loop

```
Result := greater (Result, c.item.name)
```

end

```
c.forth
```

Am Schluss: Invariante **and** Abbruchbedingung

- Alle Stationen besucht (*c.after*)
- **Result** ist der "grösste" Stationsname

Quiz: Finde die Invariante



```
xxxx (a, b : INTEGER): INTEGER
  -- ?????????????????????????????????????????????????????????
  require
    a > 0 ; b > 0
  local
    m, n : INTEGER
  do
    from
      m := a ; n := b
    invariant
      -- "?????????"
    variant
      -- ??????????
    until
      m = n
    loop
      if m > n then
        m := m - n
      else
        n := n - m
      end
    end
  end
  Result := m
end
```

Quiz: Finde die Invariante

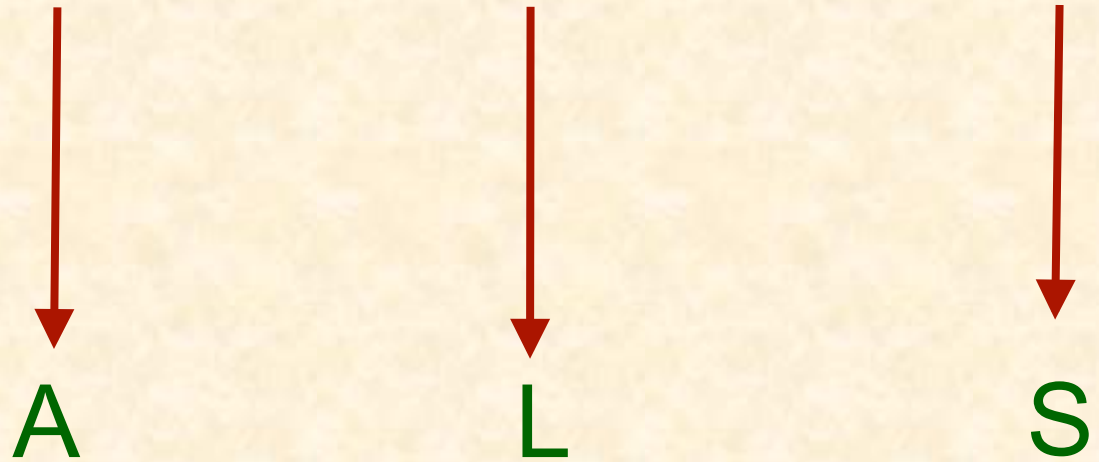


```
euclid (a, b: INTEGER): INTEGER
  -- Grösster gemeinsamer Teiler von a und b.
  require
    a > 0 ; b > 0
  local
    m, n : INTEGER
  do
    from
      m := a ; n := b
    invariant
      -- "?????????"
    variant
      -- ??????????
    until
      m = n
    loop
      if m > n then
        m := m - n
      else
        n := n - m
      end
    end
  end
  Result := m
end
```


Levenshtein-Distanz



Von "Beethoven" nach "Beatles"?



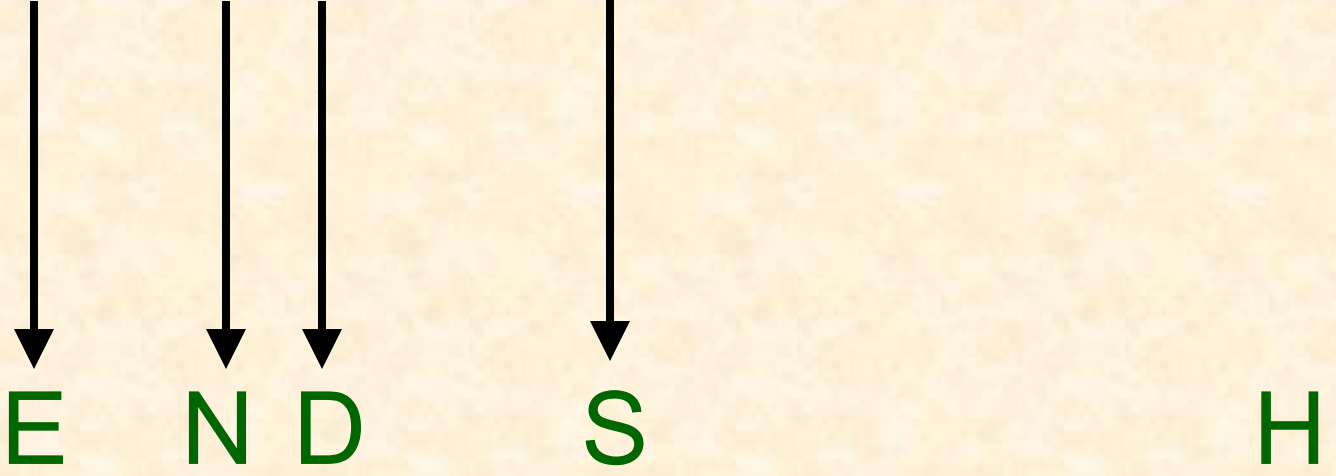
Operration	—	—	R	—	D	R	D	—	R
Distanz	0	0	1	1	2	3	4	4	5

Ein weiteres Beispiel



Von "Michael Jackson" nach "Mendelssohn"

M I ~~C~~ H A E L █ ~~J~~ ~~A~~ ~~C~~ ~~K~~ S O N



Operation

- R D R S ——— R D D D D ——— I —

Distanz

0 1 2 3 4 ——— 5 6 7 8 9 ——— 10 —

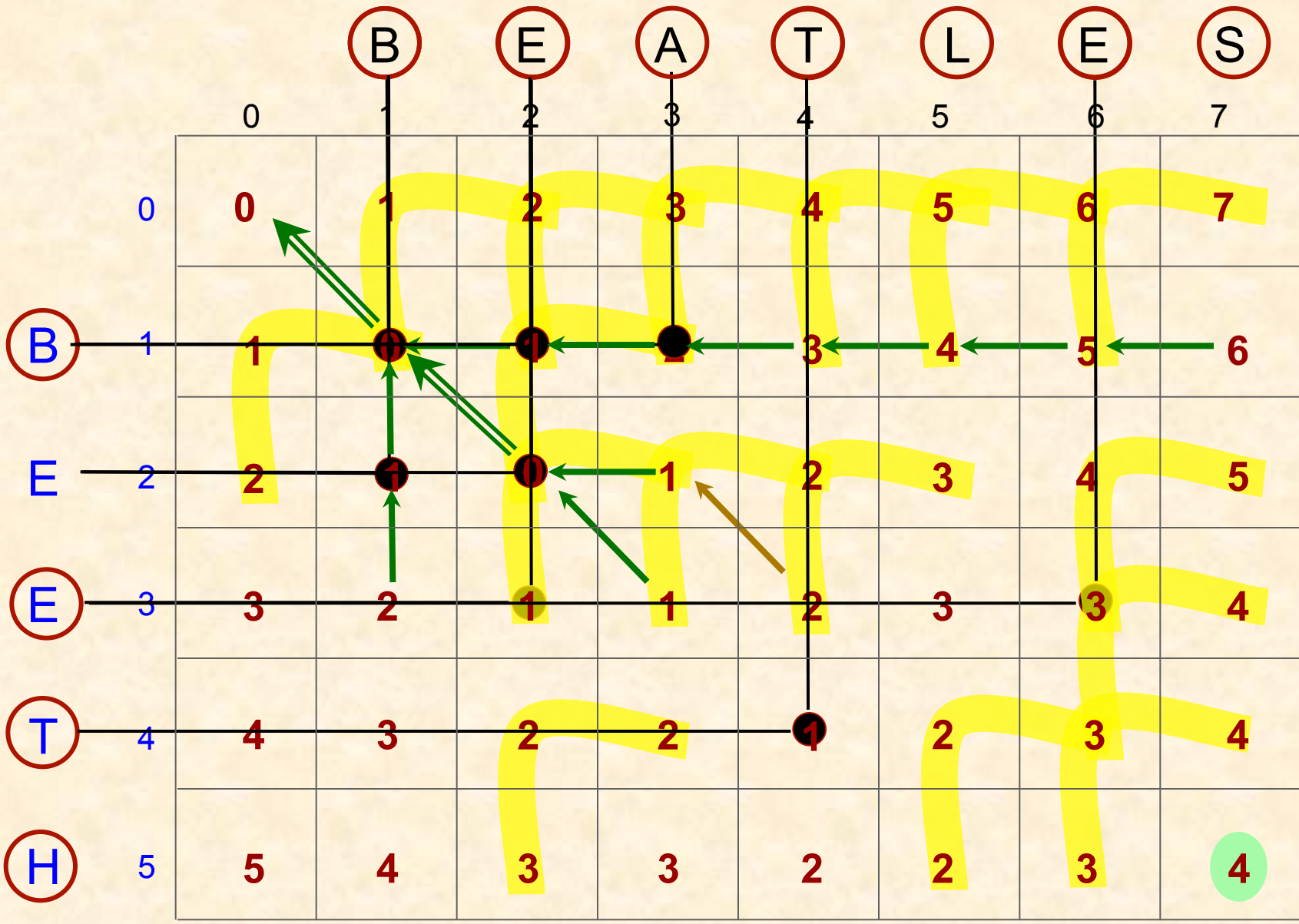


Auch als „Editierdistanz“ (*edit distance*) bekannt

Zweck: Die kleinste Menge von Grundoperationen

- **Einfügung** (I - insertion)
- **Löschung** (D - deletion)
- **Ersetzung** (R - replacement)

bestimmen, so dass aus einer Zeichenkette eine andere wird



Der Levenshtein-Distanz-Algorithmus



distance (source, target: STRING): INTEGER

-- Minimale Anzahl Operationen, um *source* in *target*
-- umzuwandeln.

local

dist: ARRAY_2 [INTEGER]

i, j, del, ins, subst: INTEGER

do

create *dist.make (source.count, target.count)*

from *i := 0 until i > source.count loop*

dist [i, 0] := i ; i := i + 1

end

from *j := 0 until j > target.count loop*

dist [0, j] := j ; j := j + 1

end

-- (Fortsetzung folgt)

Der Levenshtein-Distanz-Algorithmus



```
from  $i := 1$  until  $i > source.count$  loop  
  from  $j := 1$  until  $j > target.count$  invariant
```

???

```
loop
```

```
  if  $source[i] = target[j]$  then
```

```
     $dist[i, j] := dist[i-1, j-1]$ 
```

```
  else
```

```
     $deletion := dist[i-1, j]$ 
```

```
     $insertion := dist[i, j-1]$ 
```

```
     $substitution := dist[i-1, j-1]$ 
```

```
     $dist[i, j] := minimum(deletion, insertion, substitution) + 1$ 
```

```
  end
```

```
   $j := j + 1$ 
```

```
end
```

```
   $i := i + 1$ 
```

```
end
```

```
Result :=  $dist(source.count, target.count)$ 
```

```
end
```

Wie wissen wir, ob eine Schleife terminiert?

from

c := *Line8.new_cursor* ; **Result** := ""

invariant

c.index >= 1

c.index <= *Line8.count* + 1

-- **Result** ist der grösste der bisherigen Namen.

until

c.after

loop

Result := *greater* (**Result**, *c.item.name*)

c.forth

end





Ein Integer-Ausdruck, der

- Nach der Initialisierung (**from**) nicht-negativ sein muss
- Sich bei jeder Ausführung des Schleifenrumpfs (**loop**), bei der die Abbruchbedingung *nicht* erfüllt ist, um mindestens eins **verringern**, aber trotzdem nicht-negativ bleiben muss

Die Variante in unserem Beispiel



from

c := Line8.new_cursor ; **Result := ""**

invariant

c.index >= 1

c.index <= Line8.count + 1

-- **Result** ist der grösste der bisherigen Namen.

until

c.after

loop

Result := greater (Result, c.item.name)

c.forth

variant

Line8.count – c.index +

end 1



Ihre Chefin gibt Ihnen den Quellcode eines C-Compiler

- Beachten Sie, dass der Compiler mehr als ein Übersetzer in Maschinencode ist. Er ist auch ein “Prüfer” (verifier)
- Er prüft viele Eigenschaften, z.B. Typ-Eigenschaften, und weist Programme ab, wenn sie diese Eigenschaften nicht erfüllen.

Ihre Chefin verlangt, dass Sie neue solche Überprüfungen hinzufügen; spezifisch will sie, dass der Compiler analysiert, ob das Programm zur Laufzeit immer terminiert

Ihre Antwort:




- 1. Ja, sehr einfach, das kann ich schaffen.
- 2. Ich kann es tun, es wird aber schwer, ich brauche mehrere Wochen oder möglicherweise ein paar Monate.
- 3. Es ist nicht möglich für C, es wäre jedoch möglich für Java.
- 4. Es ist nicht möglich für C, es wäre jedoch möglich für Eiffel.
- 5. Sorry, Chef, ich kann nicht helfen.

Das allgemeine Entscheidungsproblem



Kann EiffelStudio herausfinden, ob Ihr Programm terminieren wird?

Leider nein 

Auch kein anderes Programm kann dies für irgendeine realistische Programmiersprache herausfinden!   



(“Halting Problem”, Alan Turing, 1936)

Es ist **nicht** möglich, eine effektive Prozedur zu schreiben, die herausfindet, ob ein beliebiges Programm mit beliebigem Input terminieren wird

(Oder, im Speziellen, ob ein beliebiges Programm ohne Input terminiert)



Nehmen Sie an, wir haben ein Feature

```
terminiert (datei: STRING): BOOLEAN
```

```
-- Ist es der Fall, dass das in datei
```

```
-- gespeicherte Programm terminiert ?
```

```
require
```

```
enthält_programm (datei)
```

```
do
```

```
... Ihr Algorithmus ...
```

```
end
```

Dann...



Wurzelprozedur des Systems:

```
terminiert_wenn_nicht  
    -- Terminiert nur, falls nein.  
do  
    from  
    until  
        not terminiert ("/usr/home/turing.e")  
    loop  
    end  
end
```

Dann: wir speichern diesen Programmtext in </usr/home/turing.e>



Das Paradox von Russel:

- Manche Mengen sind Element von sich selber; die Menge aller unendlichen Mengen ist z.B. selbst unendlich
- Manche Mengen sind nicht Element von sich selbst; die Menge aller endlichen Mengen ist z.B. nicht endlich
- Betrachten sie die Menge aller Mengen, die sich nicht selbst enthalten

Das Barbier-Paradox (Russel, leicht abgeändert)

- In Zürich gibt es einen Barbier, der alle Männer rasiert, die sich nicht selbst rasieren
- Wer rasiert den Barbier?



In der deutschen Sprache ist ein Adjektiv

- “**autologisch**”, falls es sich selbst beschreibt (z.B. “deutsch” oder „mehrsilbig“)
- “**heterologisch**” sonst

Was ist nun mit “**heterologisch**”?

Eine andere Form:

Die erste Aussage auf dieser Folie, die in rot erscheint, ist falsch



(Sehr alt!)

- Epaminondas sagt, dass alle Kreter Lügner sind
- Epaminondas ist ein Kreter



Manche Programme terminieren in gewissen Fällen nicht

Das ist ein Bug!

- Ihre Programme sollten in jedem Fall terminieren!
- Benutzen Sie Varianten!

Das generelle Unentscheidbarkeit-Theorem verhindert nicht, die Terminierung eines **spezifischen** Programms zu beweisen



Nicht-konditionaler Zweig:

BR *label*

Konditionaler Zweig, z.B.:

BEQ *loc_a loc_b label*

Das Äquivalent zu if-then-else



```
if a = b then Verbund_1 else Verbund_2 end
```



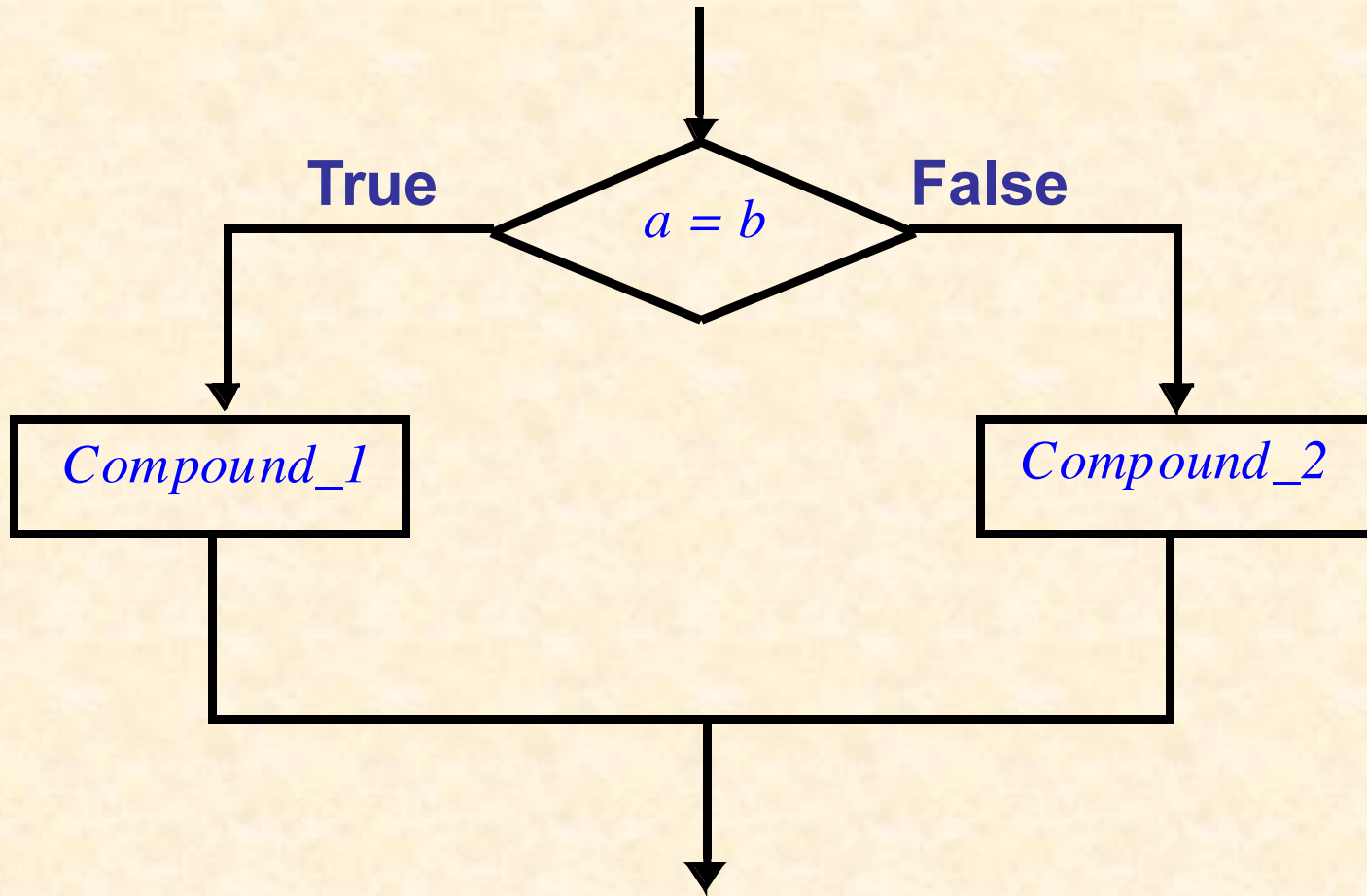
```
BEQ loc_a loc_b 111
```

```
101 ... Code für Verbund_2 ...
```

```
BR 125
```

```
111 ... Code für Verbund_1 ...
```

```
125 ... Code für Rest des Programms ...
```





test *Bedingung* **goto** *else_part*

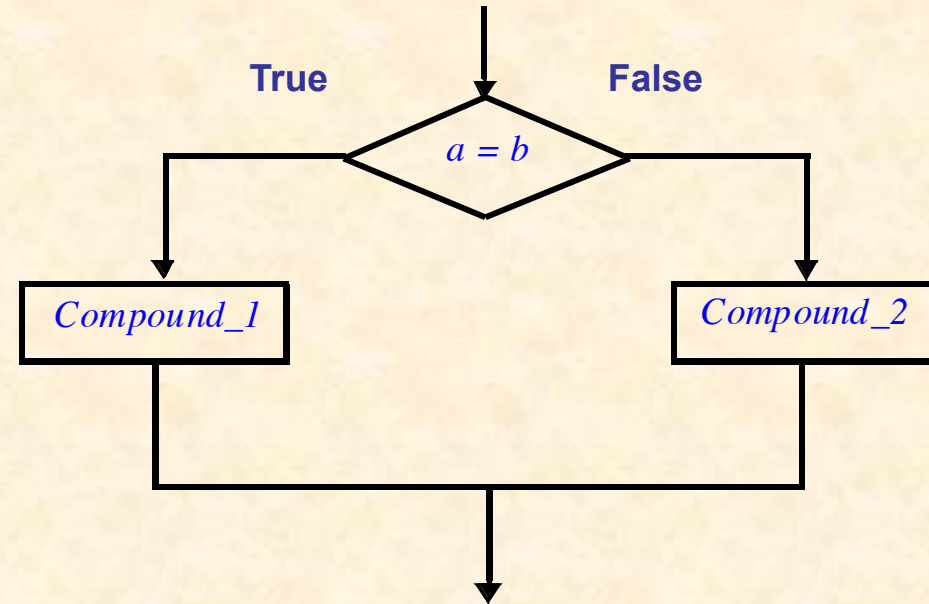
Verbund_1

goto *continue*

else_part :

Verbund_2

continue : ... Fortsetzung des Programms...



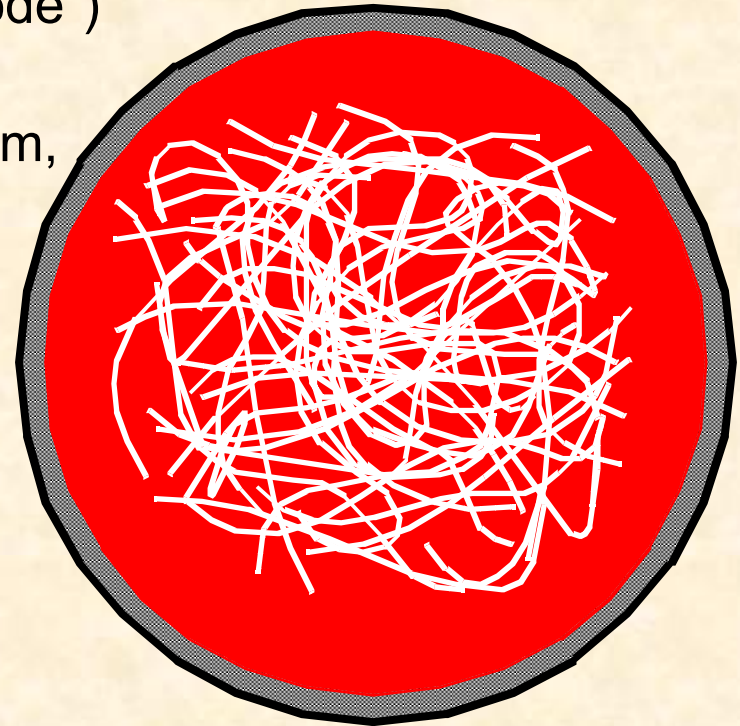
“Goto considered harmful”

Dijkstra, 1968

Willkürliche Goto-Instruktionen führen zu unübersichtlichen, schwer zu wartenden Programmen (“spaghetti code”)

Böhm-Jacopini-Theorem: Jedes Programm, das mit **goto**-Instruktionen und Konditionalen geschrieben werden kann, kann auch ohne **goto**‘s geschrieben werden, indem man Sequenzen und Schleifen benutzt.

Beispiel zur Transformation
in *Touch of Class*





Fast allgemein verschrien

Immer noch in einigen Programmiersprachen vorhanden.

Es versteckt sich auch unter anderen Namen, z.B. **break**

```
loop
```

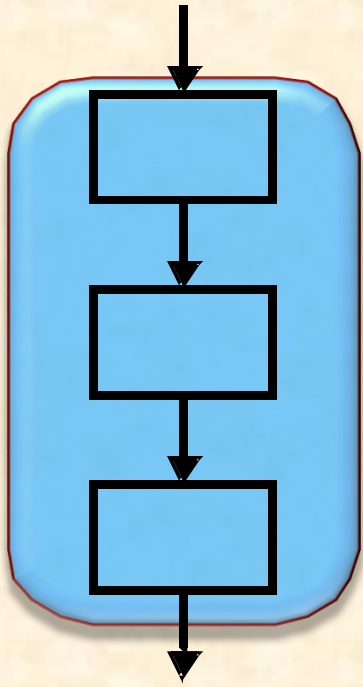
```
...
```

```
if c then break end
```

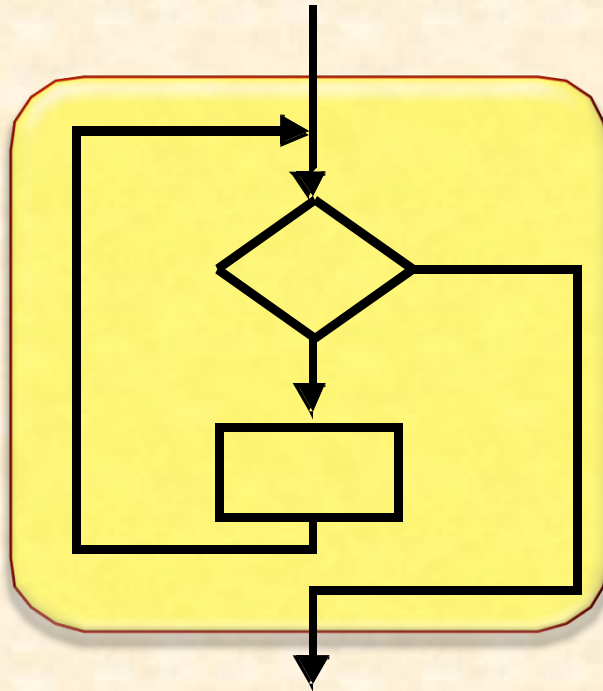
```
...
```

```
end
```

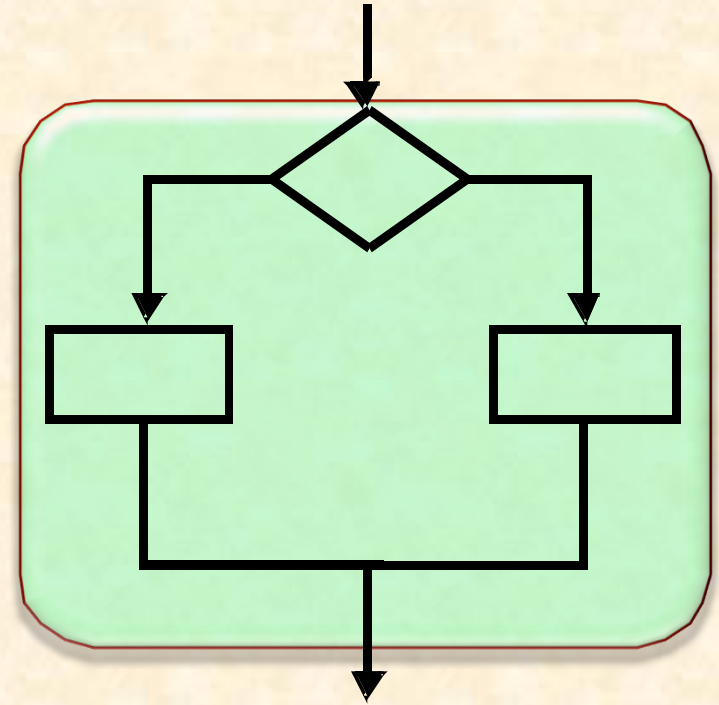

Ein Eingang, ein Ausgang



(Verbund)



(Schleife)



(Konditional)

Quiz: Finde die Invariante!

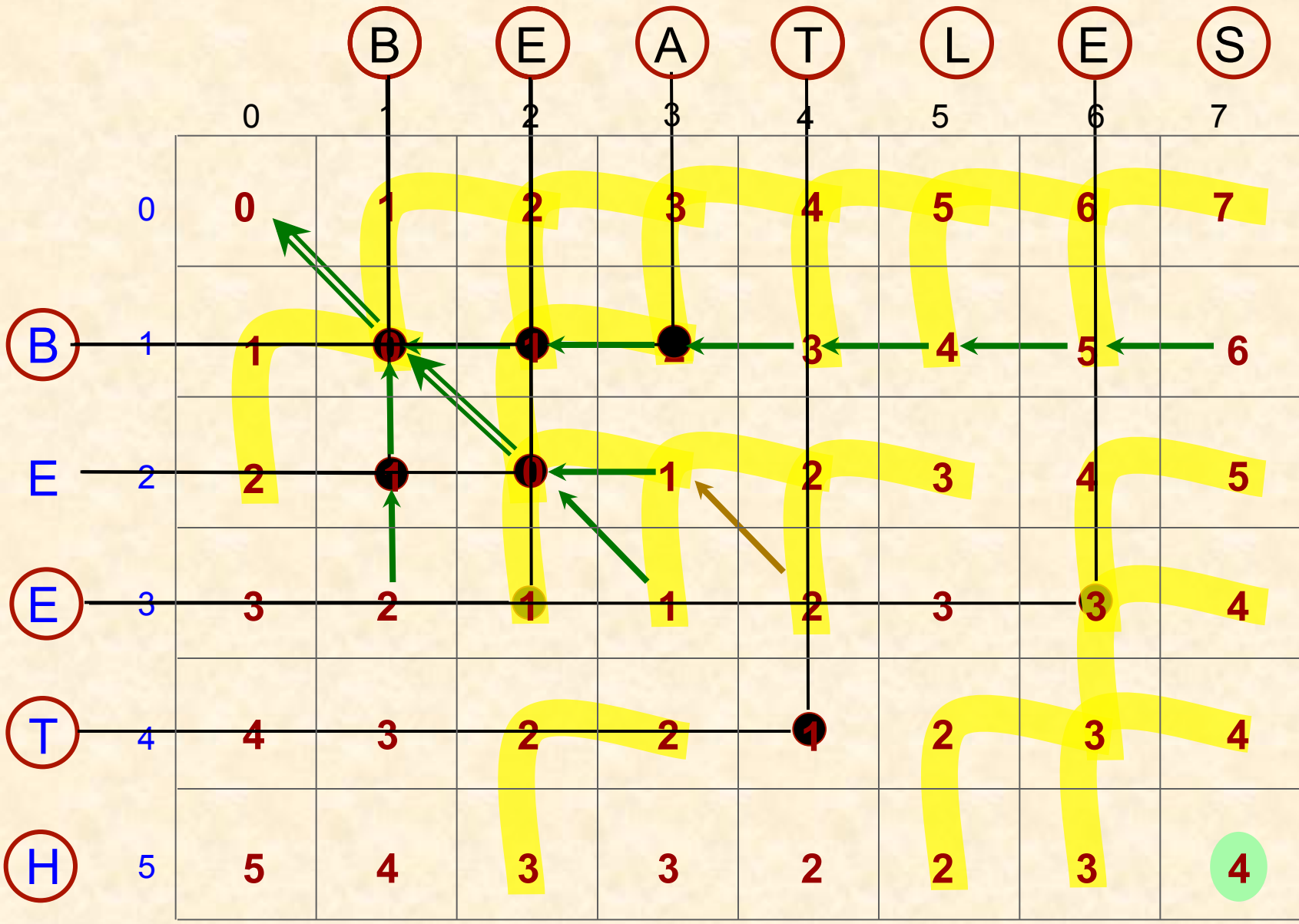


```
xxx (a, b: INTEGER): INTEGER
-- ?????????????????????????????????????????
require
  a > 0 ; b > 0
local
  m, n : INTEGER
do
  from
    m := a ; n := b
  invariant
    -- "?????????"
  variant
    -- ??????????
  until
    m = n
  loop
    if m > n then
      m := m - n
    else
      n := n - m
    end
  end
  Result := m
end
```

Quiz: Finde die Invariante!



```
euclid (a, b: INTEGER): INTEGER
  -- Grösster gemeinsamer Teiler von a und b.
  require
    a > 0 ; b > 0
  local
    m, n : INTEGER
  do
    from
      m := a ; n := b
    invariant
      -- "?????????"
    variant
      -- ??????????
    until
      m = n
    loop
      if m > n then
        m := m - n
      else
        n := n - m
      end
    end
  end
  Result := m
end
```



Der Levenshtein-Distanz-Algorithmus



distance (source, target: STRING): INTEGER

-- Minimale Anzahl Operationen, um *source* in *target*
-- umzuwandeln.

local

dist: ARRAY_2 [INTEGER]

i, j, del, ins, subst: INTEGER

do

create *dist.make (source.count, target.count)*

from *i := 0 until i > source.count loop*

dist [i, 0] := i ; i := i + 1

end

from *j := 0 until j > target.count loop*

dist [0, j] := j ; j := j + 1

end

-- (Fortsetzung folgt)

Der Levenshtein-Distanz-Algorithmus



```
from  $i := 1$  until  $i > source.count$  loop  
  from  $j := 1$  until  $j > target.count$  invariant
```

???

```
loop
```

```
  if  $source[i] = target[j]$  then
```

```
     $dist[i, j] := dist[i-1, j-1]$ 
```

```
  else
```

```
     $deletion := dist[i-1, j]$ 
```

```
     $insertion := dist[i, j-1]$ 
```

```
     $substitution := dist[i-1, j-1]$ 
```

```
     $dist[i, j] := minimum(deletion, insertion, substitution) + 1$ 
```

```
  end
```

```
   $j := j + 1$ 
```

```
end
```

```
   $i := i + 1$ 
```

```
end
```

```
Result :=  $dist(source.count, target.count)$ 
```

```
end
```

Levenshtein, fortgesetzt



```
from i := 1 until i > source.count loop
  from j := 1 until j > target.count invariant
    -- Für alle  $p : 1 .. i$ ,  $q : 1 .. j-1$ , können wir  $source [1 .. p]$ 
    -- in  $target [1 .. q]$  umwandeln mit  $dist [p, q]$  Operationen.
  loop
    if source [i] = target [j] then
      new := dist [i-1, j-1]
    else
      deletion := dist [i-1, j]
      insertion := dist [i, j-1]
      substitution := dist [i-1, j-1]
      new := deletion.min (insertion.min (substitution)) + 1
    end
    dist [i, j] := new
    j := j + 1
  end
  i := i + 1
end
Result := dist (source.count, target.count)
```

Der Levenshtein-Distanz-Algorithmus



distance (source, target: STRING): INTEGER

-- Minimale Anzahl Operationen, um *source* in *target*
-- umzuwandeln.

local

dist: ARRAY_2 [INTEGER]

i, j, new, deletion, insertion, substitution : INTEGER

do

create *dist.make (source.count, target.count)*

from *i := 0 until i > source.count loop*

dist [i, 0] := i ; i := i + 1

end

from *j := 0 until j > target.count loop*

dist [0, j] := j ; j := j + 1

end

-- (Fortsetzung folgt)

Levenshtein, fortgesetzt



```
from  $i := 1$  until  $i > source.count$  loop
  from  $j := 1$  until  $j > target.count$  invariant
    -- Für alle  $p : 1 .. i, q : 1 .. j-1$ , können wir  $source [1 .. p]$ 
    -- in  $target [1 .. q]$  umwandeln mit  $dist [p, q]$  Operationen.
  loop
    if  $source [i] = target [j]$  then
       $new := dist [i-1, j-1]$ 
    else
       $deletion := dist [i-1, j]$ 
       $insertion := dist [i, j-1]$ 
       $substitution := dist [i-1, j-1]$ 
       $new := deletion.min (insertion.min (substitution)) + 1$ 
    end
     $dist [i, j] := new$ 
     $j := j + 1$ 
  end
   $i := i + 1$ 
end
Result :=  $dist (source.count, target.count)$ 
```



- Der Begriff des Algorithmus
 - Grundlegende Eigenschaften
 - Unterschied zu einem Programm
- Der Begriff der Kontrollstruktur
- Korrektheit einer Instruktion
- Kontrollstrukturen der Strukturierten Programmierung:
 - Sequenz
 - Konditional
 - Kontrollstruktur: Konditional
- Verschachtelung, und wie sich diese vermeiden lässt
- Korrektheit von Schleifen: Invariante & Variante
- Das Entscheidungsproblem und Unentscheidbarkeit