ETH Zürich
Chair of Software Engineering

Einführung in die Programierung

Prof. Dr. Bertrand Meyer

Lektion 8: Kontrollstrukturen I

In dieser (Doppel-) Lektion

- Der Begriff des Algorithmus
- Grundlegende Kontrollstrukturen: Sequenz (*sequence, compound*), Konditional (*conditional*), Schleife (*loop*)
- Bedingungsinstruktionen: Der Konditional und seine Variante
- Operationen wiederholen: Die Schleife
- Schleifen als Annäherungsstrategie: Die Schleifeninvariante
- Was braucht es, um sicherzustellen, dass eine Schleife terminiert?
- Ein Blick auf das allgemeine Problem der Programmterminierung
- Kontrollstrukturen auf einer tieferen Ebene: "Goto" und Flowcharts; siehe Argument für die „Kontrollstrukturen der strukturierten Programmierung"
- Das Entscheidungsproblem

Bitte lesen Sie Kapitel 8 von „Touch of Class"

Der Begriff des Algorithmus

Allgemeine Definition:

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

Nicht wirklich ein Algorithmus

PREPARAZIONE E TEMPI DI COTTURA
ZUBEREITUNG - PREPARATION

Versate le verdure ancora surgelate in 1 litro abbondante d'acqua fredda con 2 cucchiai 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. Er terminiert für alle Daten garantiert 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 geringer:

- 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

Kontrollstrukturen

Definition: Ein Programmkonstrukt, welches den Ablauf von Programmschritten beschreibt.

Drei fundamentale Kontrollstrukturen:

- Sequenz
- Schleife
- Konditional

Diese sind die

"Kontrollstrukturen des strukturierten Programmierens"

Kontrollstrukturen als Techniken für die Problemlösung

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."

Conditional: „Löse das Problem separat für zwei oder mehrere Teilmengen der Input-Menge."

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

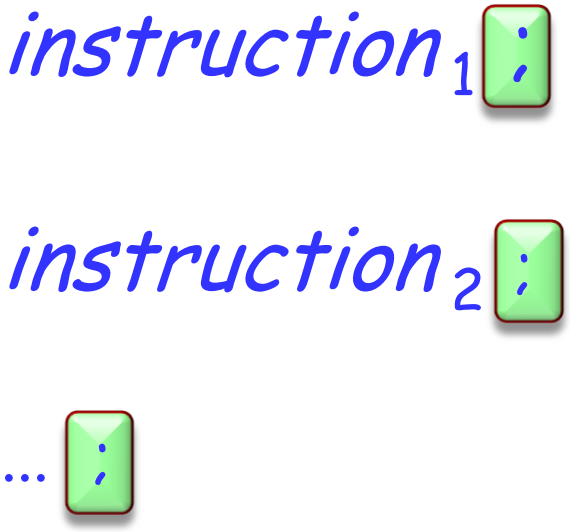
instruction $_1$

instruction $_2$

...

instruction $_n$

Das Semikolon als optionale Trennung

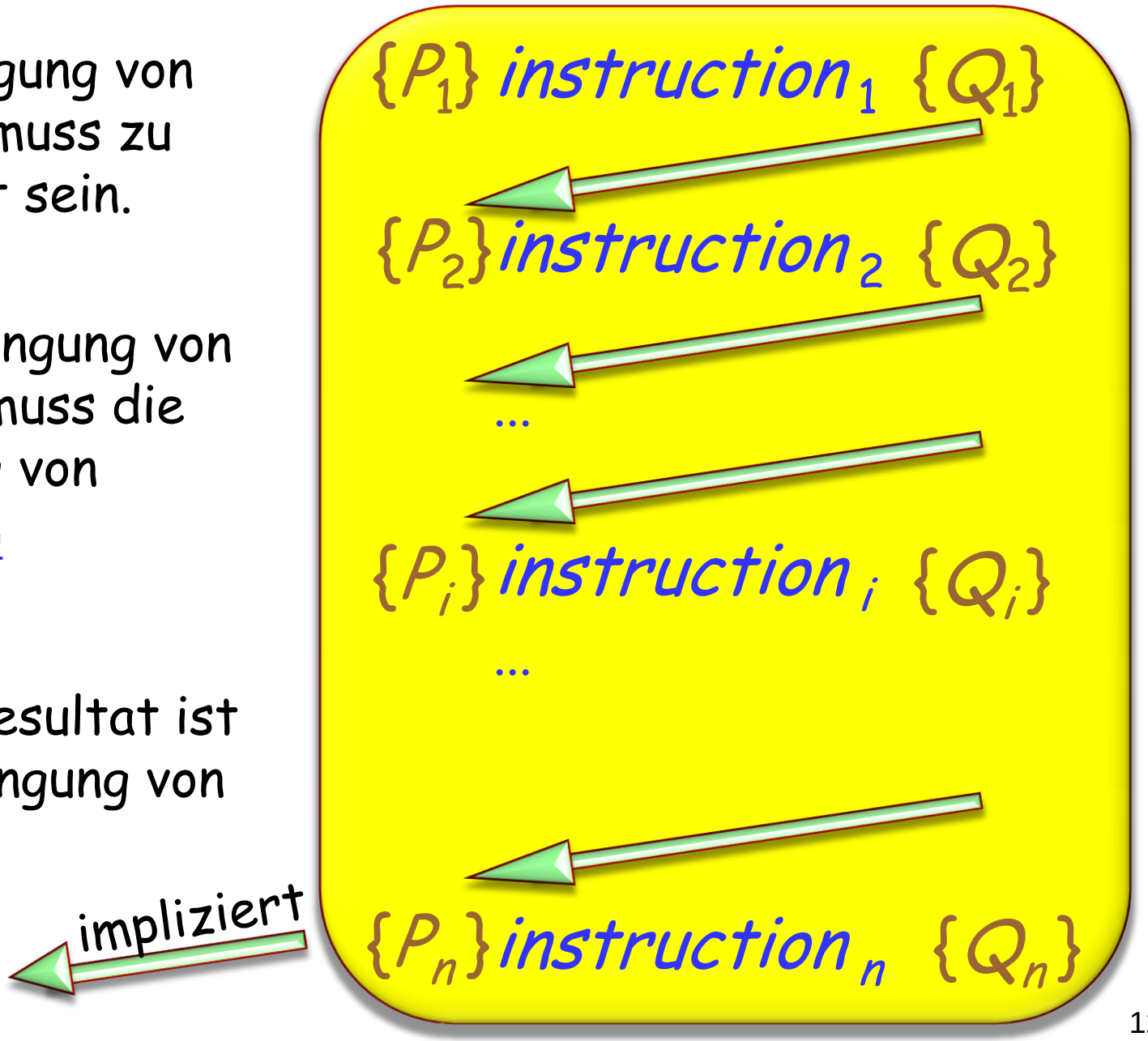


Korrektheit eines Verbunds

Die Vorbedingung von $instruction_1$ muss zu Beginn erfüllt sein.

Die Nachbedingung von $instruction_i$ muss die Vorbedingung von $instruction_{i+1}$ implizieren.

Das Schlussresultat ist die Nachbedingung von $instruction_n$



Konditional (Bedingte Aufweisung)

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

Das Maximum von zwei Zahlen ermitteln

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

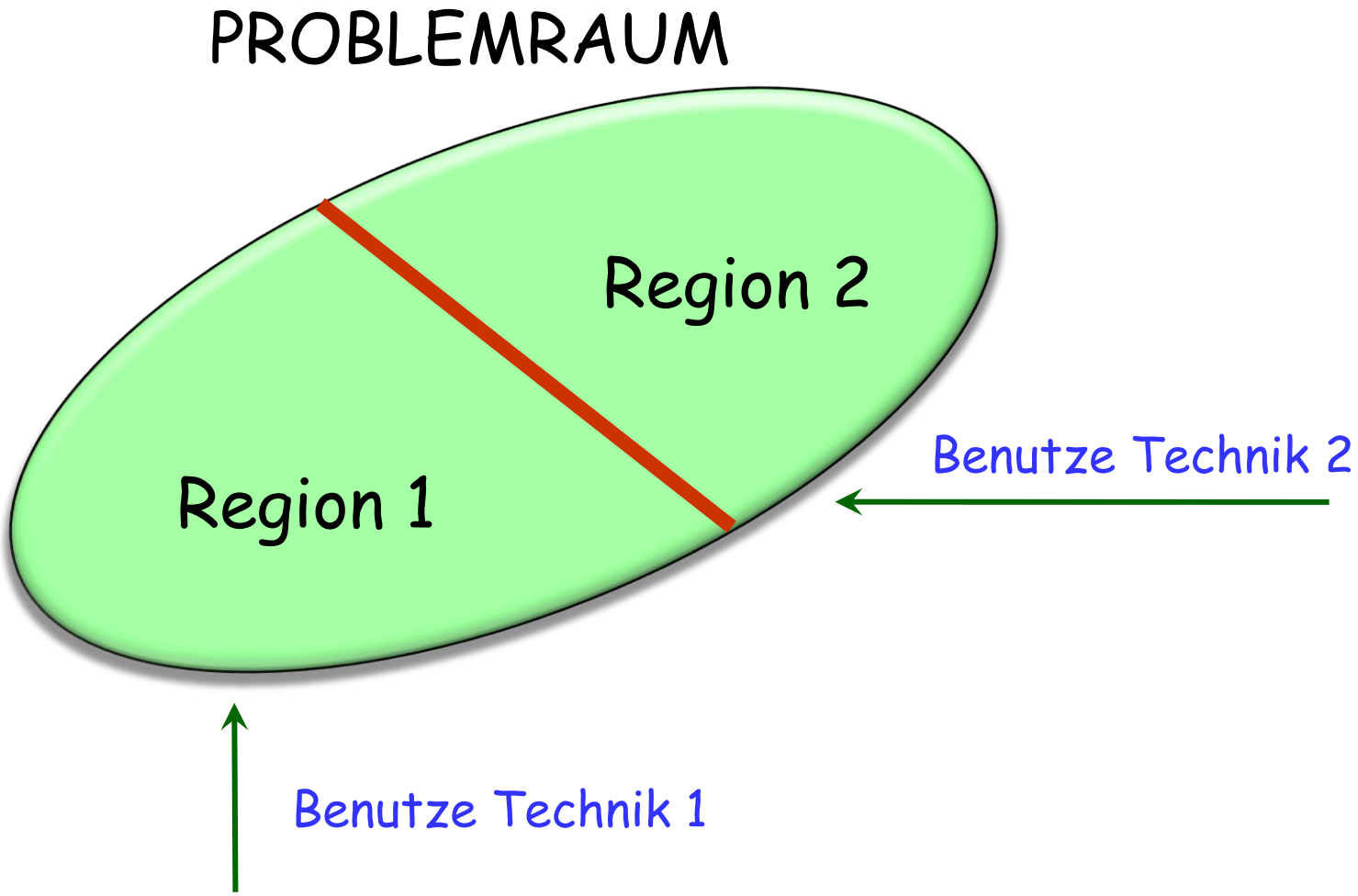
Als Feature (Abfrage)

```
greater (a, b : INTEGER): INTEGER
            -- Das Maximum von a und b.
    do
        if
            a > b
        then
            Result := a
        else
            Result := b
        end
    end
```

Typischer Aufrufe

```
i, j, k, m, n: INTEGER

...

m := max (25, 32)

n := max (i + j, k)
```

Der Konditional als Technik zur Problemlösung



Grundform

```
if Bedingung then
        Instruktionen
else
        Other_instructions
end
```

Eine Variante des Konditionals

```
if Bedingung then
   Instruktionen
end
```

Eine Variante des Konditionals

```
if Bedingung then
   Instruktionen
end
```

Ist semantisch äquivalent zu

```
if Bedingung then
      Instruktionen
else
end
```

Leere Klausel

Verschachtelung von bedingten Instruktionen

```
if Condition1 then
        Instruktionen1
else
        if Condition2 then
                Instruktionen2
        else
                if Condition3 then
                        Instruktionen3
                else
                        if Condition4 then
                                Instruktionen4
                        else
                                . . .
                        end
                end
        end
end
```

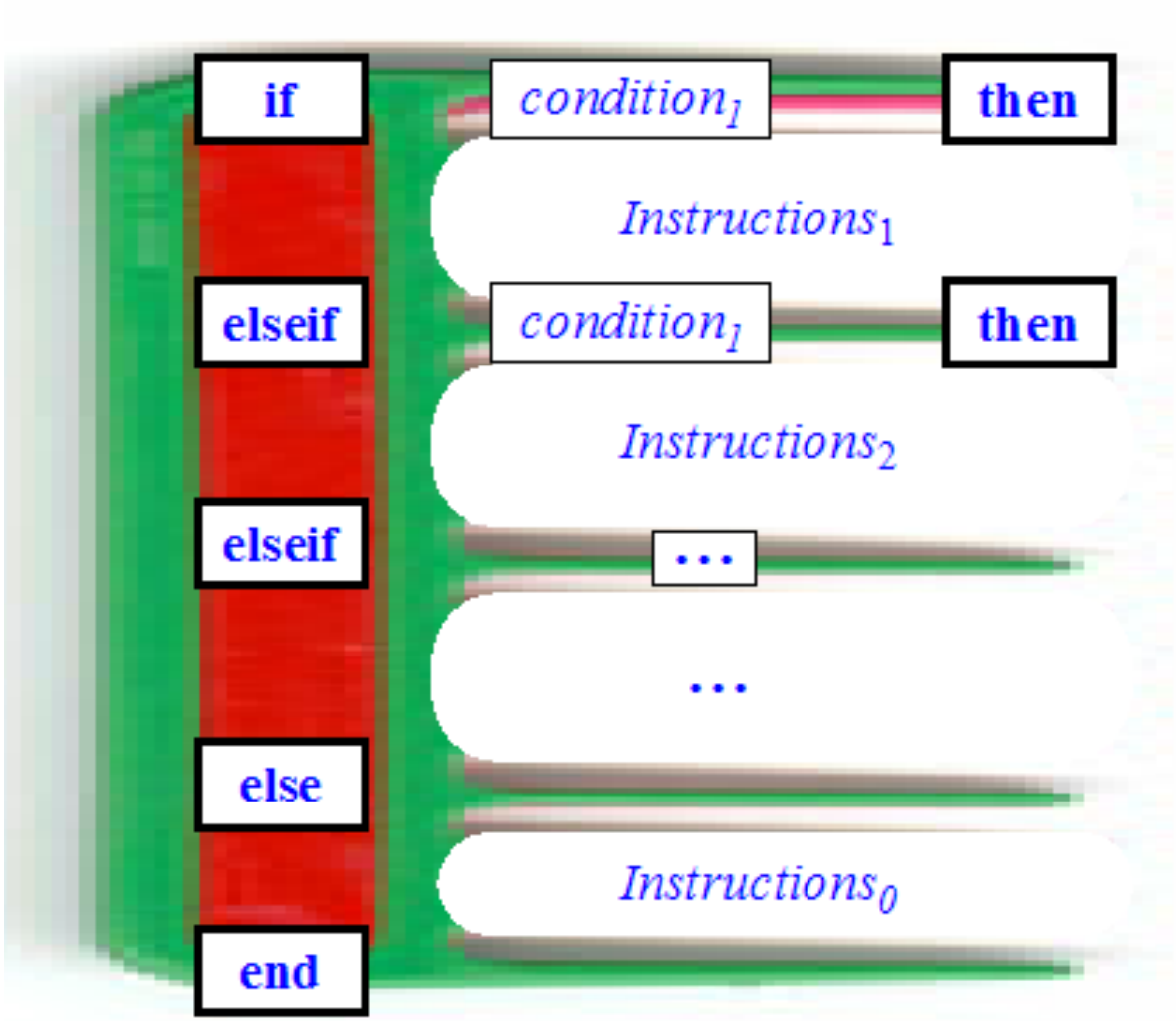
Eine verschachtelte Struktur

Eine Kamm-ähnliche Struktur

Kamm-ähnlicher Konditional

```
if Condition1 then
        Instruktionen1
elseif Bedingung2 then
        Instruktionen2
elseif Condition3 then
        Instruktionen3
elseif

        ...
else
        Instruktionen0
end
```

Eine Kamm-ähnliche Struktur



Weitere Themen zu Kontrollstrukturen

- 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 Flowcharts; siehe Argument für die „Kontrollstrukturen der strukturierten Programmierung"
- Die Unentscheidbarkeit des Entscheidungsproblems beweisen

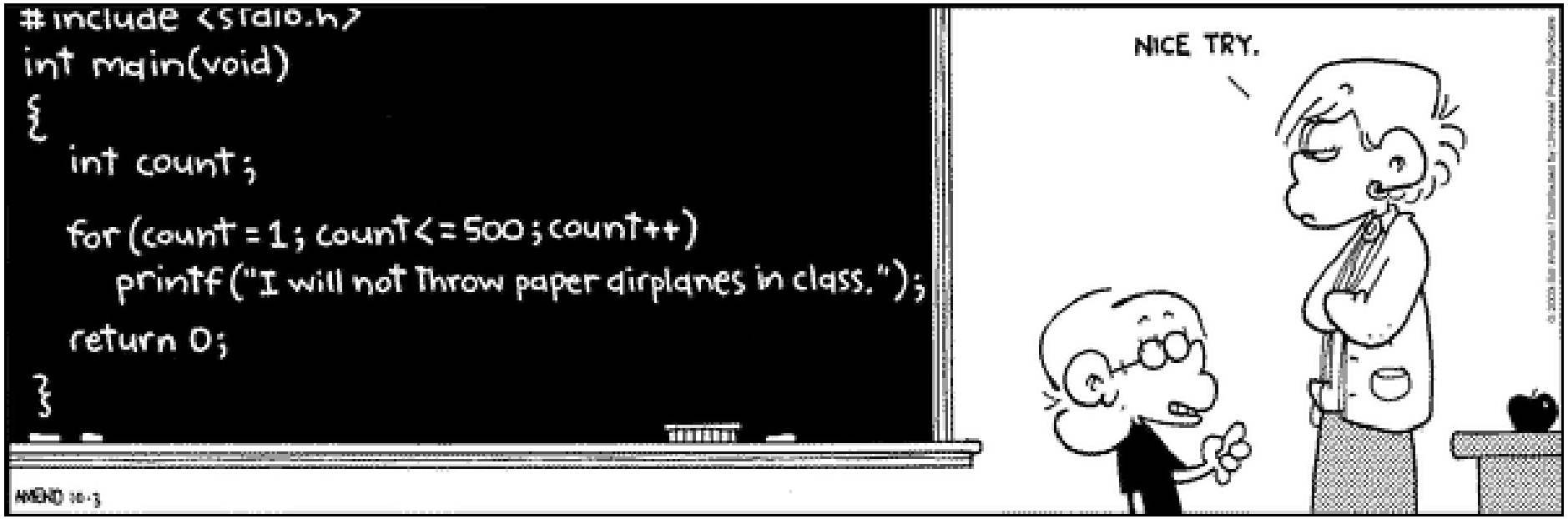
Die Schleife

```
from
        Initialization                  -- Verbund

until
        Exit_condition                  -- Boole'scher Ausdruck
loop
        Body                            -- Verbund
end
```

Die volle Form der Schleife

```
from
        Initialization                  -- Verbund
invariant
        Invariant_expression            -- Boole'scher Ausdruck
variant
        Variant_expression              -- Integer-Ausdruck
until
        Exit_condition                  -- Boole'scher Ausdruck
loop
        Body                            -- Verbund (Schleifenrumpf)
end
```

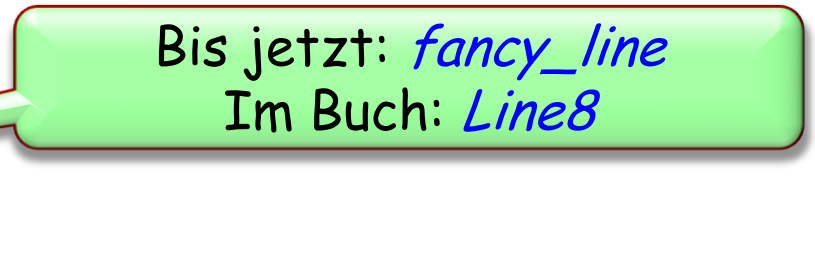
Eine andere Schleifensyntax



Die volle Form der Schleife

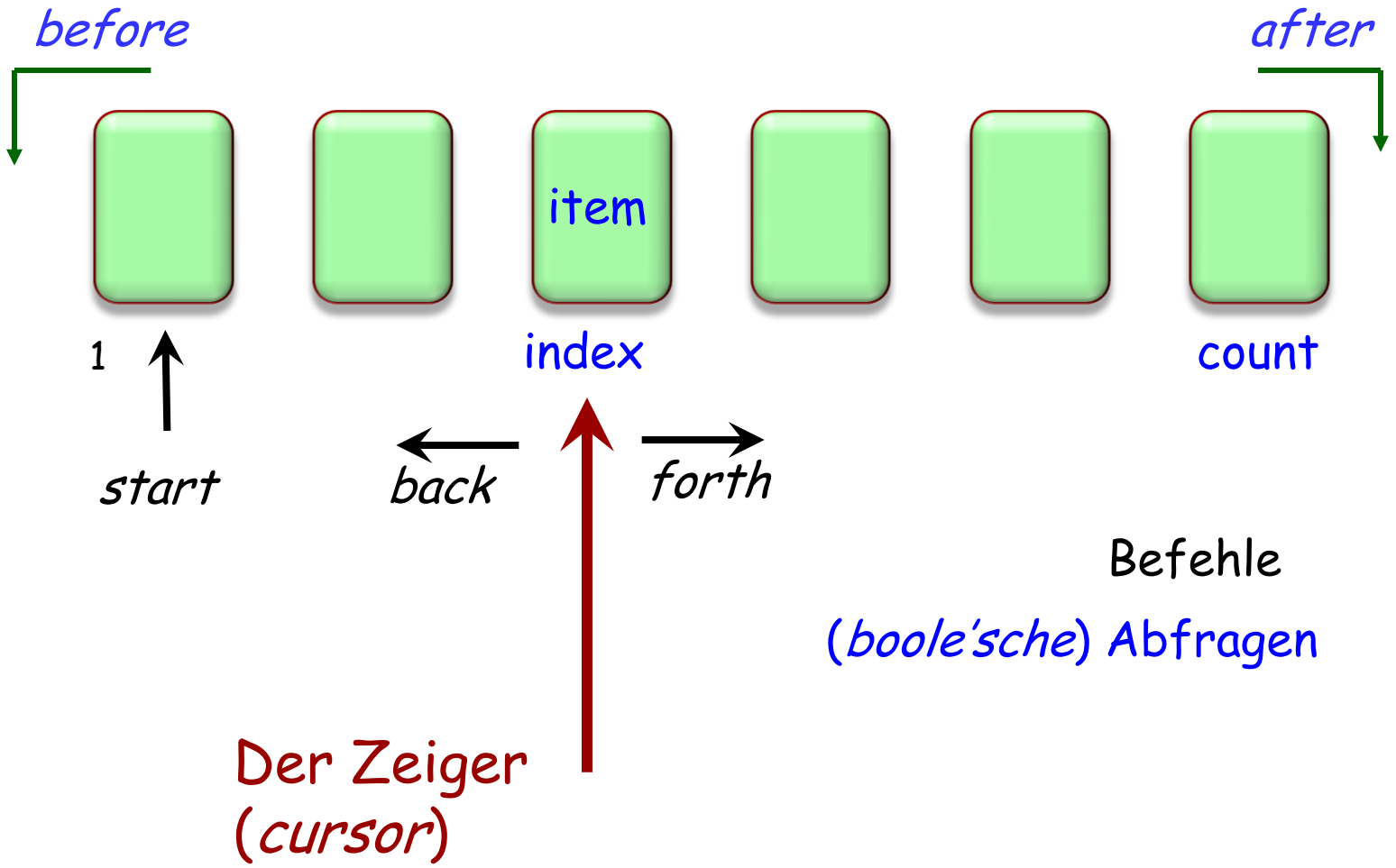
```
from
        Initialization                  -- Verbund
invariant
        Invariant_expression            -- Boole'scher Ausdruck
variant
        Variant_expression              -- Integer-Ausdruck
until
        Exit_condition                  -- Boole'scher Ausdruck
loop
        Body                            -- Verbund
end
```

Über Stationen einer Linie iterieren („loopen“)

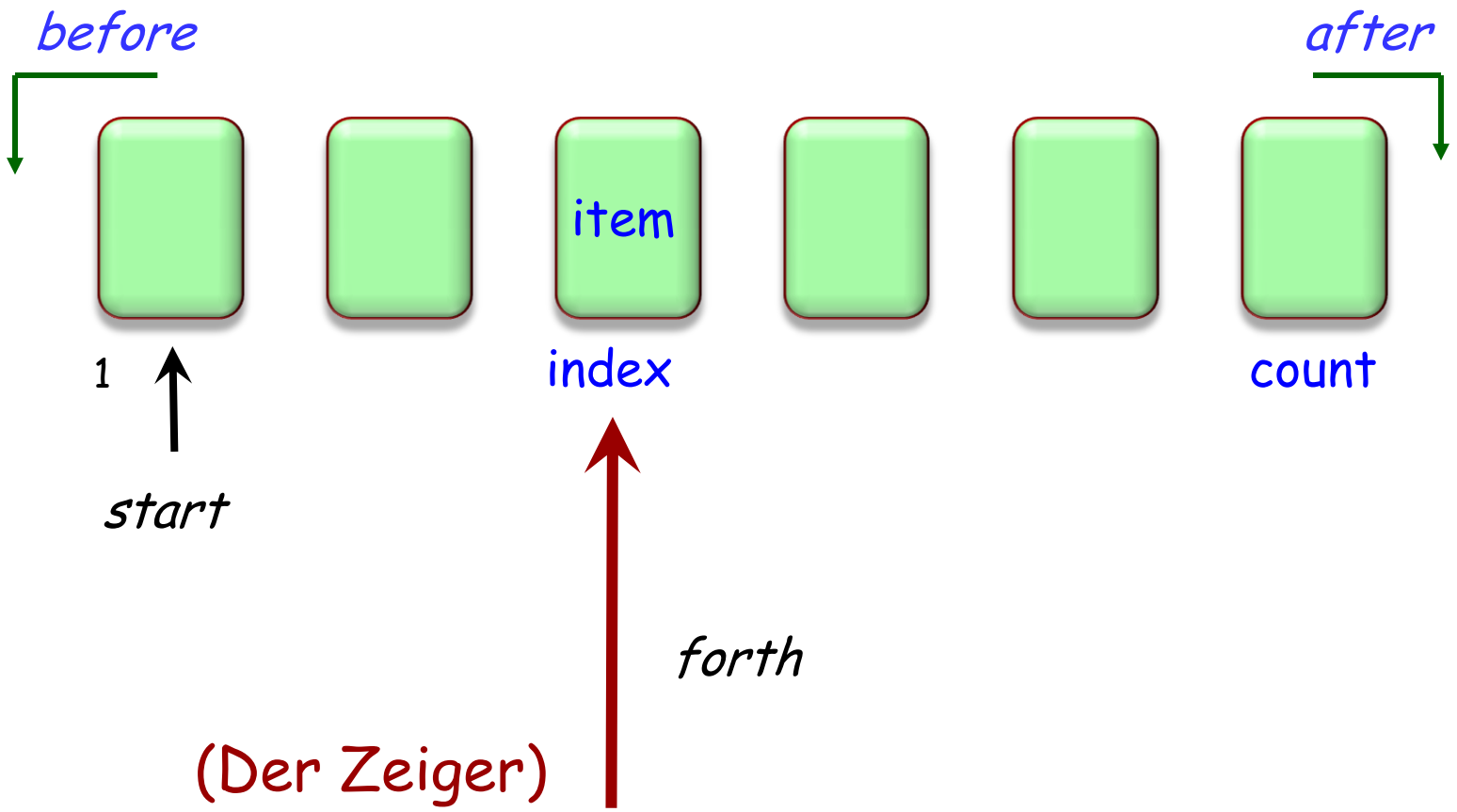



```
from
        fancy.start
until
        fancy.after
loop
        -- "Tu was mit fancy.item"
        fancy.forth
end
```

Auf eine Liste anwendbare Operationen



Auf eine Liste anwendbare Operationen



Über Stationen einer Linie iterieren („loopen“)

```
from
        fancy.start
until
        fancy.after
loop
        -- “Tu was mit fancy.item”


        fancy.forth
end
```

Die Stationsnamen anzeigen

```
from
        fancy.start
until
        fancy.after
loop
                -- Den Namen der aktuellen Station anzeigen:
        Console.show (fancy.item)

        fancy.forth
end
```

Ein anderes Beispiel (von „Example 7“)

```
from
        Line8.start
invariant ... variant ...  until Line8.after loop
        if Line8.item.is_railway_connection then
                show_big_red_spot (Line8.item.location)
        elseif Line8.item.is_exchange then
                show_blinking_spot (Line8.item.location)
        else
                show_spot (Line8.item.location)
        end
        Line8.forth
end
```

Das „Maximum“ der Stationsnamen berechnen

```
from
        fancy.start ; Result := ""
until
        fancy.after
loop
        Result := greater (Result, fancy.item.name)

        fancy.forth
end
```

Das „Maximum“ der Stationsnamen berechnen

```
from
        fancy.start ;  Result := " "
until
        fancy.after
loop
        Result := greater (Result, fancy.item .name)

        fancy.forth
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 einer Feature

```
highest_name: STRING
                    -- Alphabetisch grösste Stationsname der Linie
        do
            from
                    fancy.start ;  Result := ""
            until
                    fancy.after
            loop
                    Result := greater (Result, fancy.item.name)

                    fancy.forth
            end
        end
```

Schleifenformen (in verschiedenen Sprachen)

```
from                    -- Eiffel
        Instruktionen
until
        Bedingung
loop
        Instruktionen
end
```

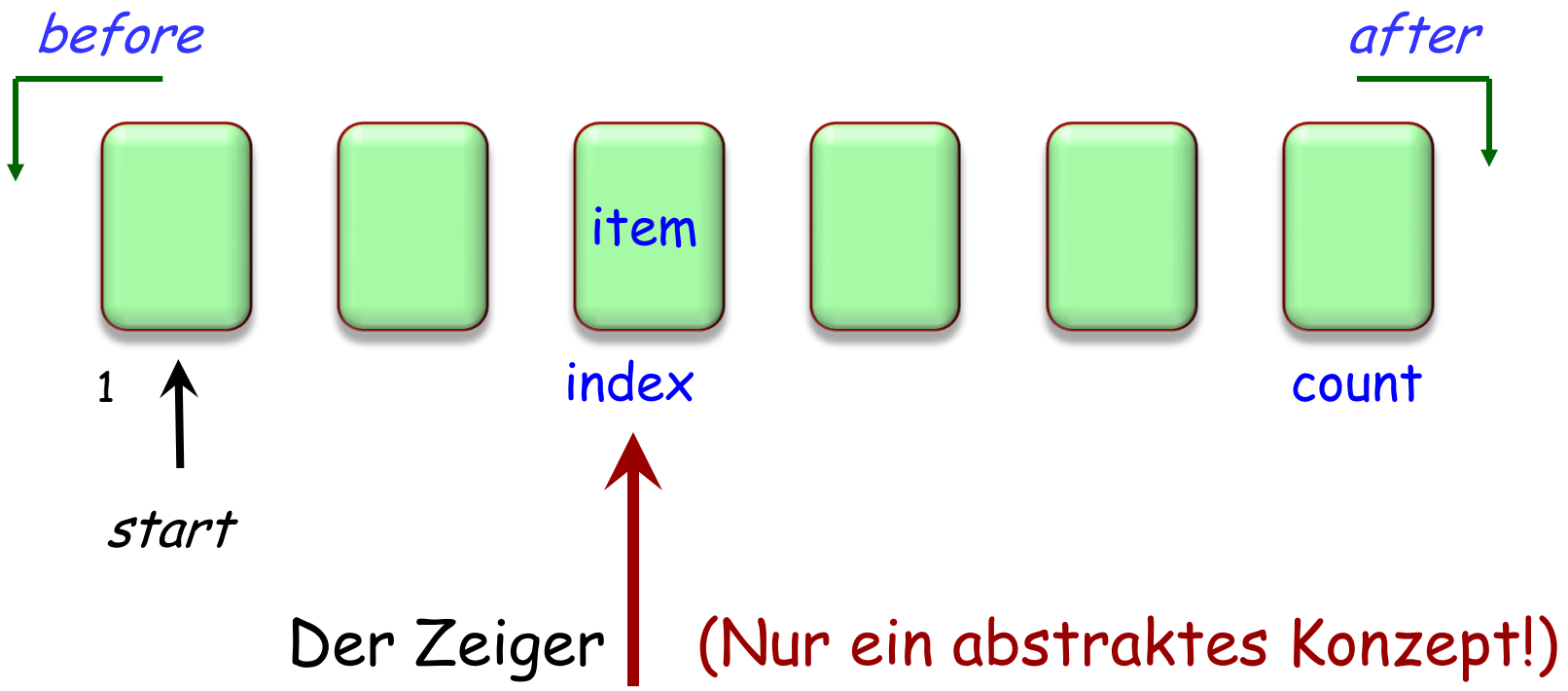
```
while Bedingung do
        Instruktionen
end
```

```
repeat
        Instruktionen
until
        Bedingung
end
```

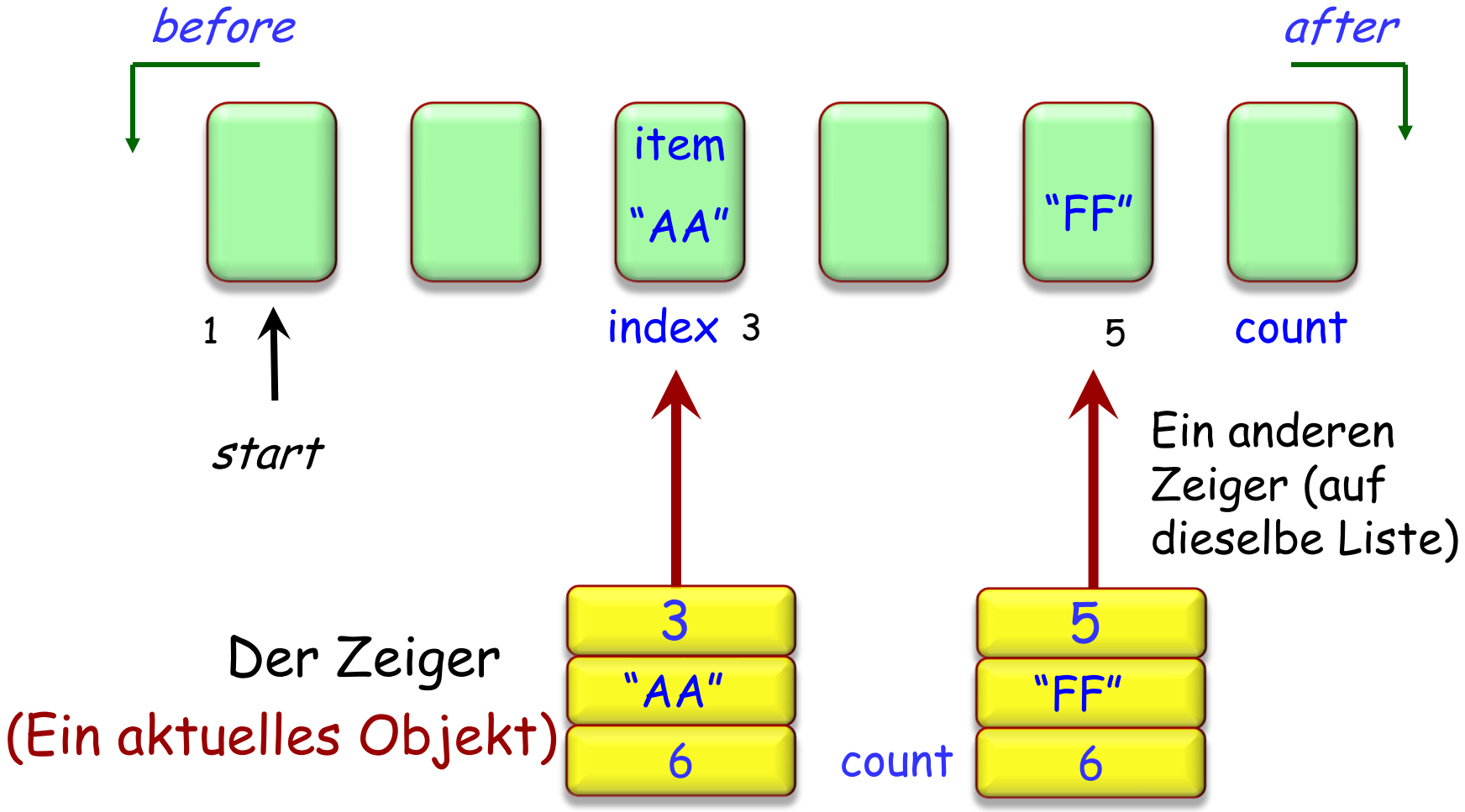
```
for i : a..b do
        Instruktionen
end
```

```
for (Instruktion; Bedingung; Instruktion) do
        Instruktionen
end
```

Listen mit internem Zeiger



Externe Zeiger sind auch möglich



Mit einem externem Zeiger programmieren

```
c: LINKED_LIST_CURSOR
...

create c.make (fancy)
```

Wie vorher, mit c statt *fancy*

```
from
        c.start ;  Result := ""
until
        c.after
loop
        Result := greater (Result, c.item.name)
        c.forth
end
```

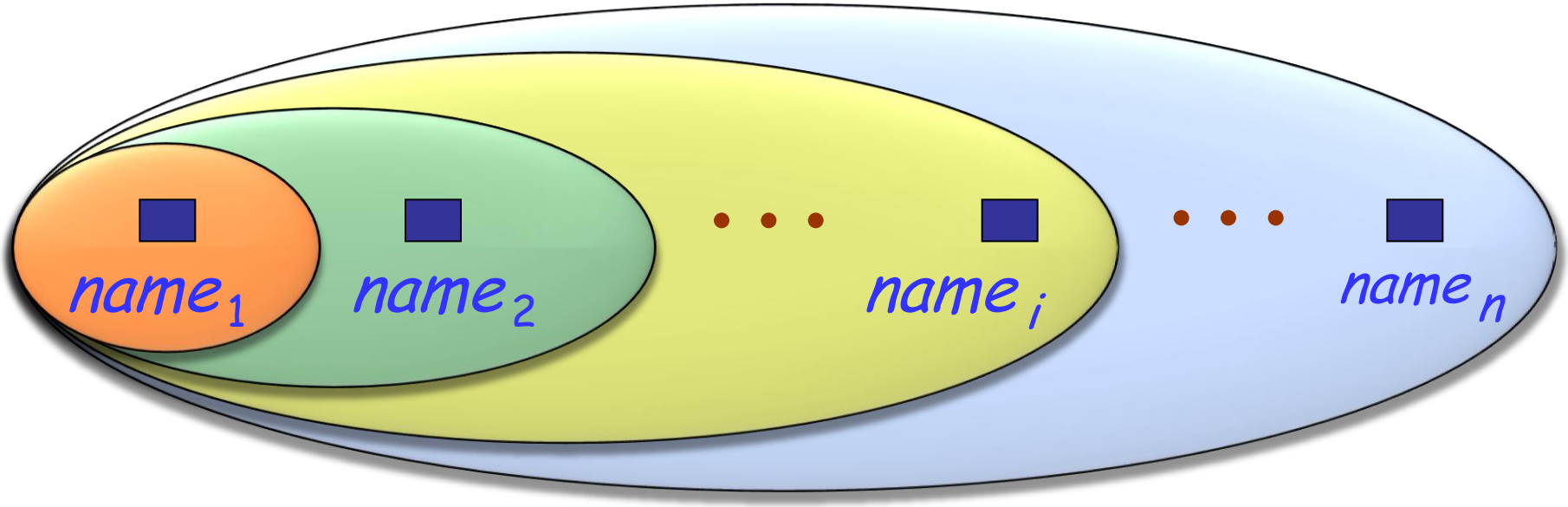
Eine andere Eiffel Schleifenform

```
across meine_liste as c loop
    "Operation auf c.item"
end
```

Diese Notation is eine Abkürzung für

```
from c.start until c.after loop
    "Operation auf c.item"
    c.forth
end
```

Schleifen als Annäherungsstrategie



Result = *name*$_1$ = Max (*names*$_{1..1}$)

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

Teil

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

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

Schleifenrumpf:

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

Nachbedingung?

```
highest_name: STRING
                    -- Alphabetisch grösste Stationsname der Linie
        do
                from
                        fancy.start ; Result := ""
                until
                        fancy.after
                loop
                        Result := greater (Result, fancy.item.name)

                        fancy.forth
                end
        ensure
                Result /= Void and then not Result.empty
                -- Result ist der Alphabetisch grösste Stationsname
                -- der Linie

        end
```

Das „Maximum“ der Stationsnamen berechnen

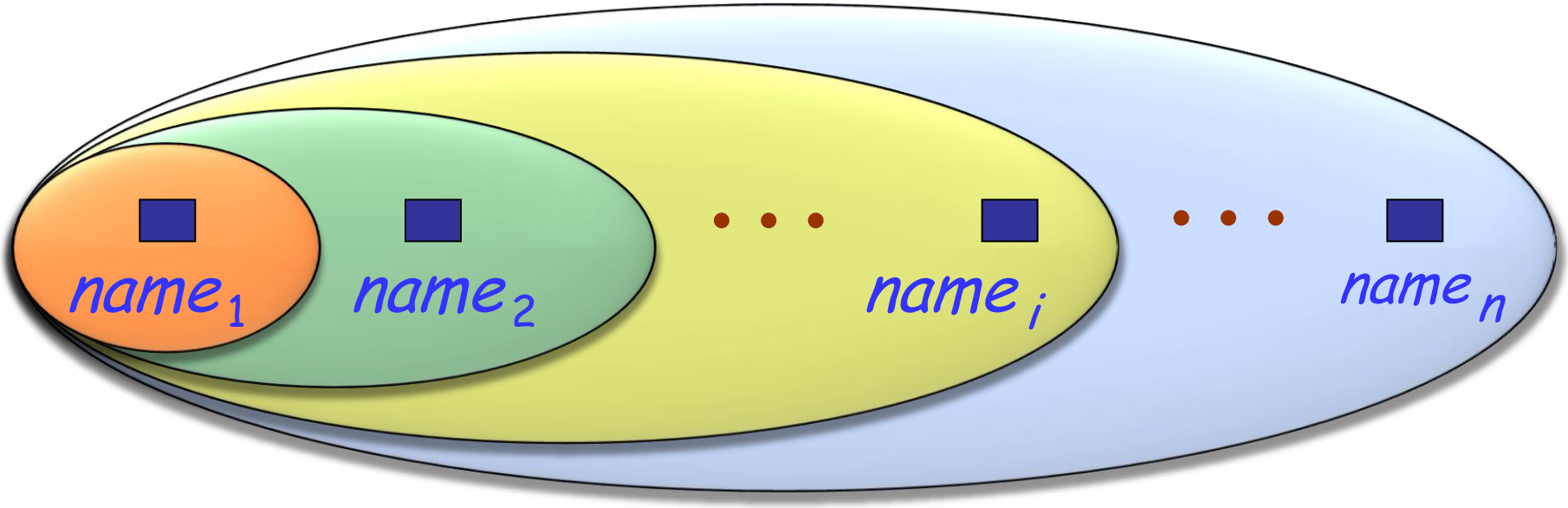
```
from
        fancy.start ;  Result := ""




until
        fancy.after
loop
        Result := greater (Result, fancy.item.name)

        fancy.forth
ensure
        -- Result ist der Alphabetisch grösste Stationsname der Linie
end
```

Die Schleifeninvariante

Fehlt etwas?

```
from
        fancy.start ; Result := ""
invariant
        fancy.index >= 1
        fancy.index <= fancy.count + 1
        -- Result ist der alphabetisch grösste Name aller
        -- bisherigen Stationen
until
        fancy.after
loop
        Result := greater (Result, fancy.item.name)

        fancy.forth
ensure
        -- Result ist der alphabetisch grösste Stationsname
end
```

Schleifen als Annäherungsstrategie



Result = *name* $_1$ = Max (*names* $_{1..1}$)

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

Schleifenrumpf:

i := *i* + 1
Result := *greater*
(**Result** , *fancy.item.name*)

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

Schleifeninvariante

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

Die Schleifeninvariante

(Nicht zu verwechseln mit der Klasseninvariante)

Eine Eigenschaft, die:

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

Die Schleifeninvariante

```
from
        fancy.start ; Result := ""
invariant
        fancy.index >= 1
        fancy.index <= fancy.count + 1
        -- Result ist der alphabetisch grösste Name
        -- aller bisherigen Stationen
until
        fancy.after
loop
        Result := greater (Result, fancy.item.name)

        fancy.forth
ensure
        -- Result ist der alphabetisch grösste Stationsname
end
```

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

Eine bessere Schleifeninvariante

```
from
        fancy.start ;  Result := ""
invariant
        index >= 1
        index <= count + 1
        -- Falls es bisherige Stationen gibt, ist
        -- Result der alphabetisch Grösste ihrer Namen
until
        fancy.after
loop
        Result := greater (Result, fancy.item.name)

        fancy.forth
ensure
        -- Result ist der alphabetisch grösste Stationsname, falls es
        -- eine Station gibt.
end
```

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

Der Effekt einer Schleife

```
from
        fancy.start ; Result := ""
invariant
        index >= 1
        index <= count + 1
        -- Result ist der grösste der bisherigen Namen
until
        fancy.after
loop
        Result := greater (Result, fancy.item.name)

        fancy.forth
end
```

Invariante nach der Initialisierung erfüllt.

Invariant nach jedem Durchlauf erfüllt.

Ausstiegsbedingung am Ende erfüllt

Am Schluss: Invariante **and** Ausstiegsbedingung

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

Quiz: Finde die Invariante

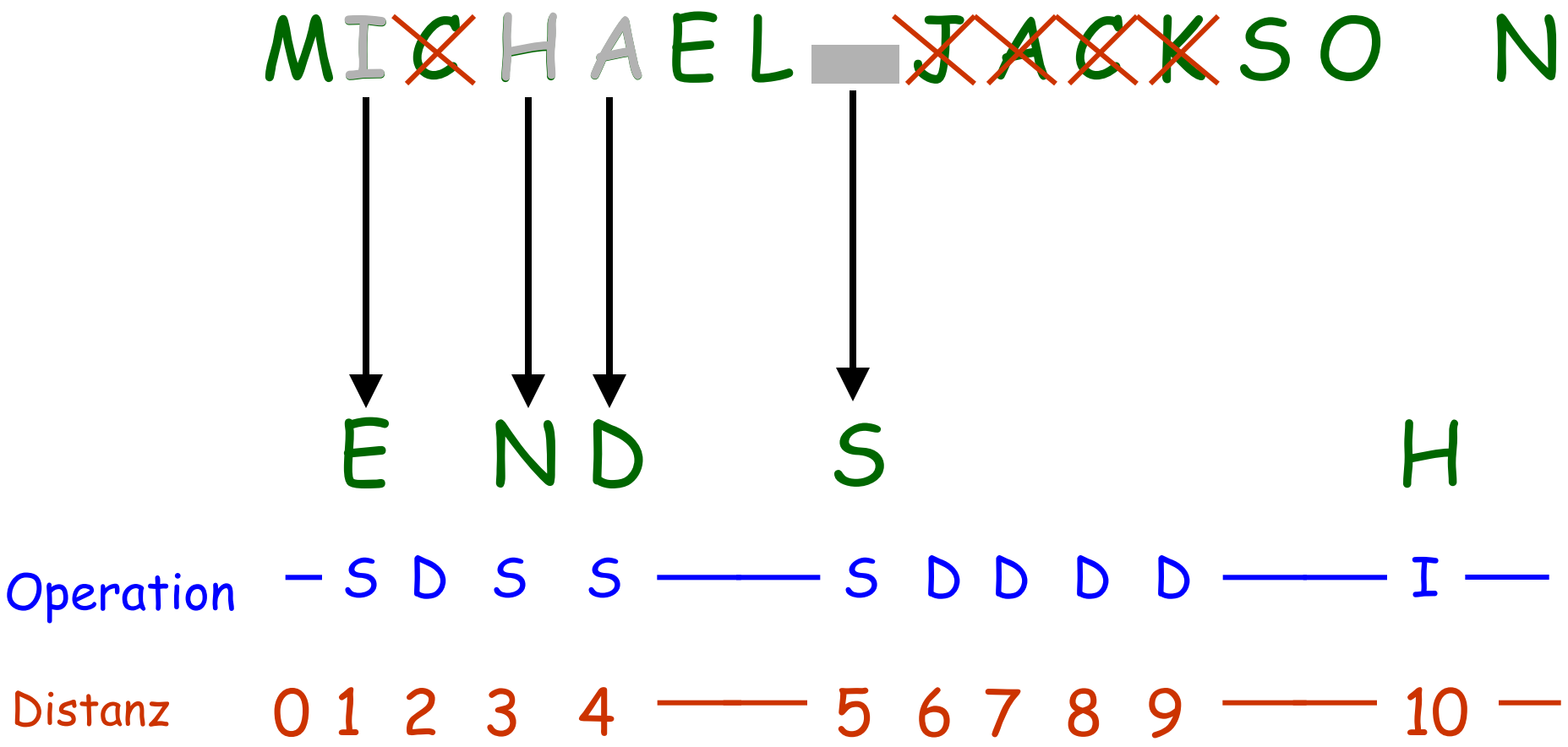
```
xxxx (a, b : INTEGER): INTEGER
        -- ????????????????????????????????
    require
        a > 0 ; b > 0
    local
        m, n : INTEGER
    do
         from
                m := a ; n := b
        invariant
                -- "????????"
        variant
                ????????
        until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

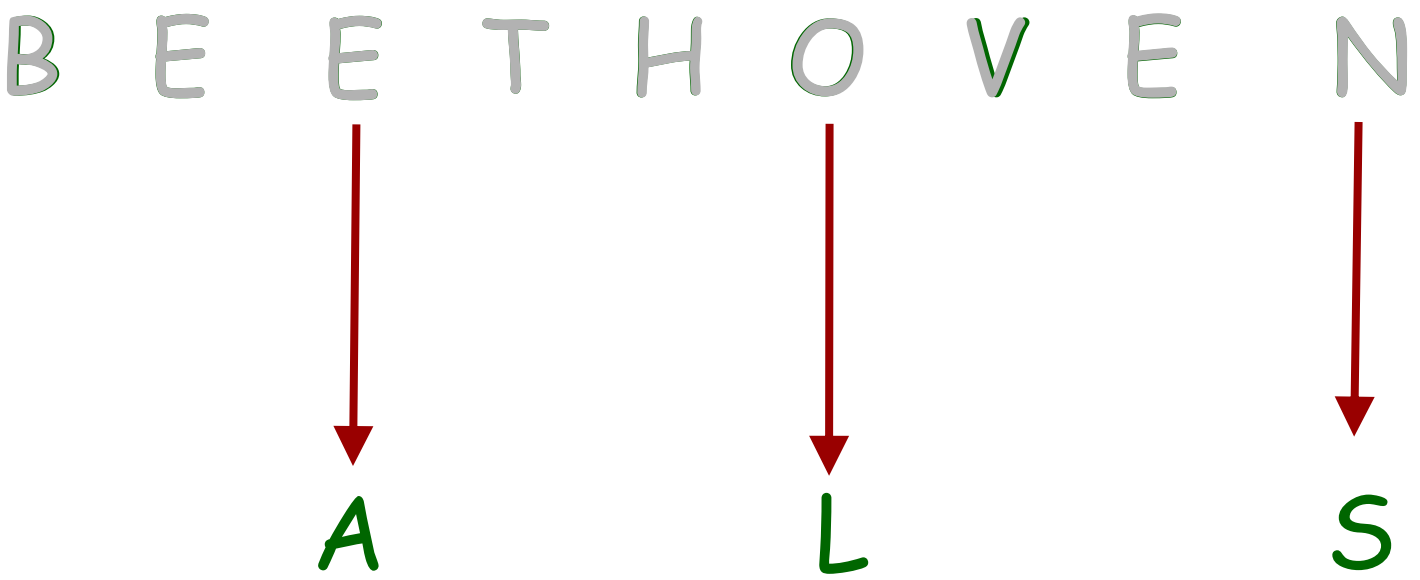
Ein weiteres Beispiel

Von "Michael Jackson" nach "Mendelssohn"



Levenshtein-Distanz

Von "Beethoven" nach "Beatles"



Operation	–	–	R	–	D	R	D	–	R
Distanz	0	0	1	1	2	3	4	4	5

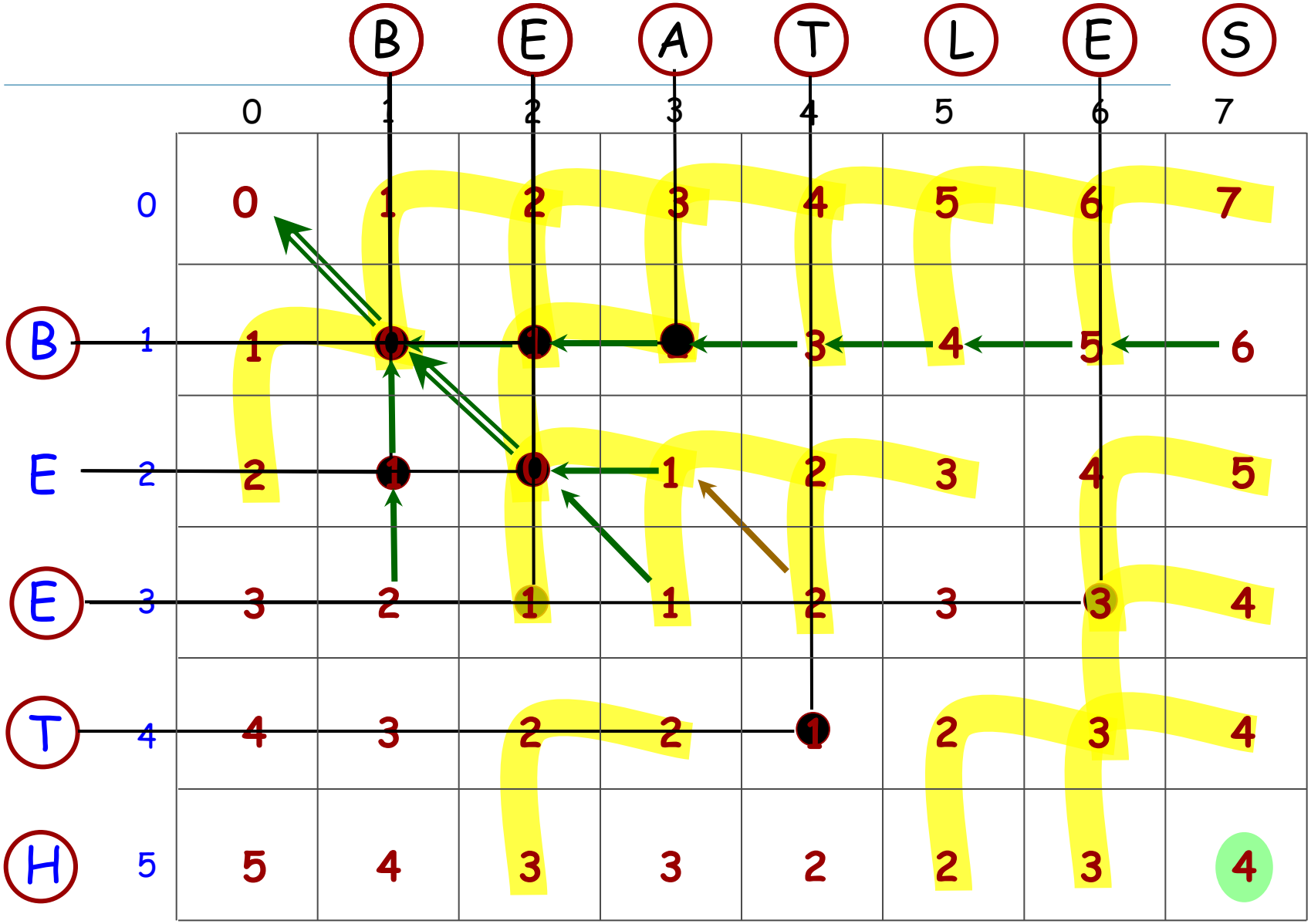
Levenshtein-Distanz

Auch als „Editierdistanz“ bekannt Bekant

Zweck: Die kleinste Menge von Grundoperationen

- Einfügung (insertion)
- Löschung (deletion)
- Ersetzung (replacement)

bestimmen, so dass aus einer Zeichenkette eine andere wird.


B E A T L E S
0 1 2 3 4 5 6 7
B
E
E
T
H
0 1 2 3 4 5 6 7
1 0 1 2 3 4 5 6
2 1 0 1 2 3 4 5
3 2 1 1 2 3 3 4
4 3 2 2 1 2 3 4
5 4 3 3 2 2 3 4

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
        -- (Weitergeführt)
```

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
        fancy.start ;  Result := ""
invariant
        index >= 1
        index <= count + 1
        -- Falls es bisherige Stationen gibt, ist
        -- Result der alphabetisch Grösste ihrer Namen
until
        fancy.after
loop
        Result := greater (Result, fancy.item.name)

        fancy.forth
end
```

Wie wissen wir, ob eine Schleife terminiert?

```
from
        fancy.start ;  Result := ""
invariant
        index >= 1
        index <= count + 1
        -- Falls es bisherige Stationen gibt, ist
        -- Result der alphabetisch Grösste ihrer Namen
until
        fancy.after
loop
        Result := greater (Result, fancy.item.name)

        fancy.forth
end
```

Die Schleifenvariante

Ein Integer-Ausdruck, der

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

Die Variante in unserem Beispiel

```
from
        fancy.start ;  Result := ""
invariant
        index >= 1
        index <= count + 1
        -- Falls es bisherige Stationen gibt, ist
        -- Result der alphabetisch Grösste ihrer Namen
variant
        fancy.count - fancy.index + 1
until
        fancy.after
loop
        Result := greater (Result, fancy.item.name)

        fancy.forth
end
```

Das allgemeine Entscheidungsproblem

Kann EiffelStudio herausfinden, ob Ihr Programm terminieren wird?

Leider nein ☹

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

Das Entscheidungsproblem und Unentscheidbarkeit

("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.)

Das Entscheidungsproblem in Eiffel

Nehmen Sie an, wir haben ein Feature

```
terminates (my_program: STRING): BOOLEAN
            -- Terminiert my_program ?
        do
            ... Your algorithm here ...
        end
```

terminates_if_not

Das Entscheidungsproblem in der Praxis

Manche Programme terminieren in gewissen Fällen nicht.

Das ist ein Bug!

Ihre Programme sollten in allen Fällen terminieren!

Benutzen Sie Varianten!

Kontrollstrukturen auf Maschinenebene

Nicht-konditionaler Zweig:

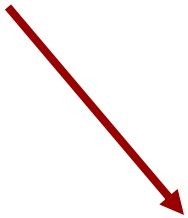
BR *label*

Konditionaler Zweig, z.B.:

BEQ *loc_a loc_b label*

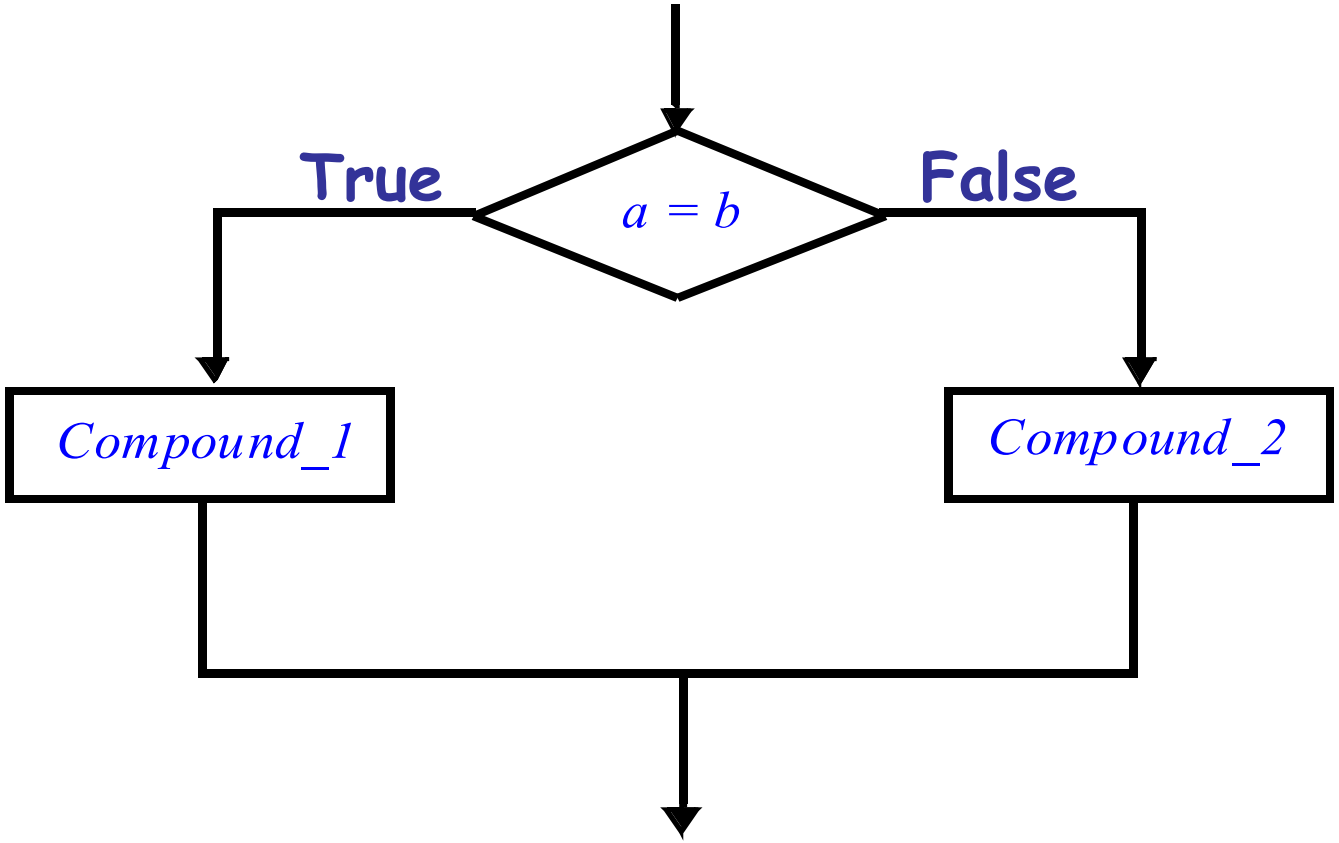
Das Äquivalent zu if-then-else

if *a = b* **then** *Compound_1* **else** *Compound_2* **end**



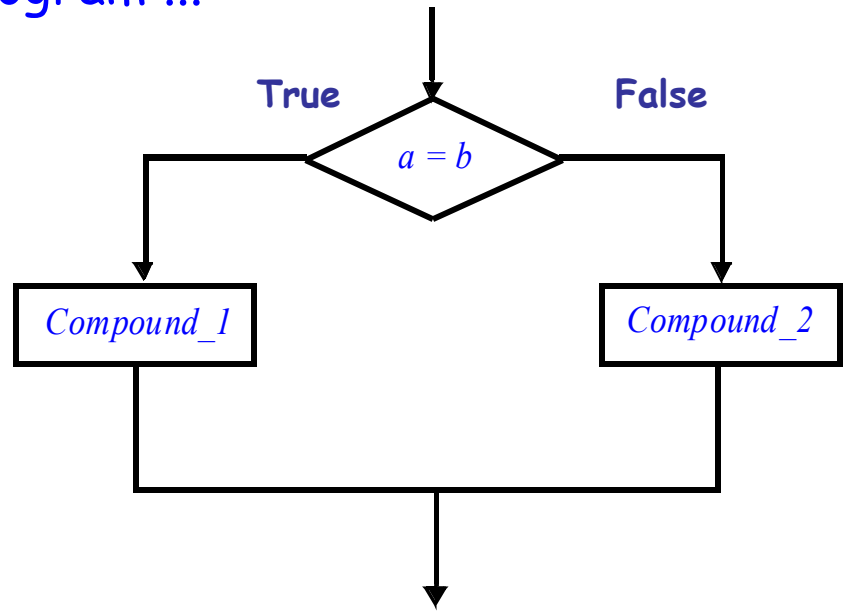
```
        BEQ loc_a loc_b 111

101     ... Code für Compound_2 ...

        BR 125

111     ... Code für Compound_1 ...

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

Flowcharts



In Programmiersprachen: Goto

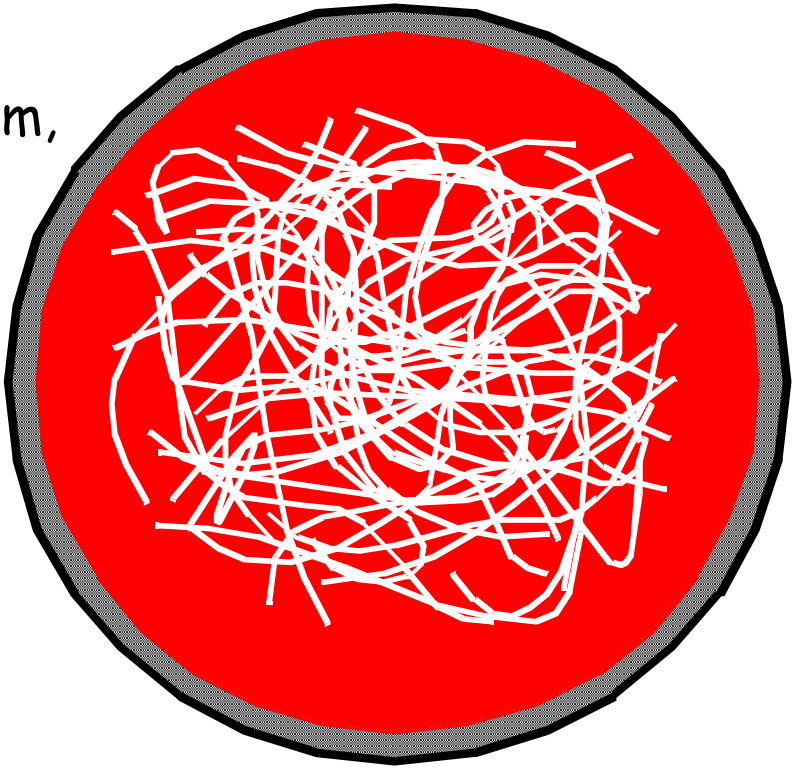
```
                test condition goto else_part
                Compound_1
                goto continue
else_part :     Compound_2
continue :      ... Continuation of program ...
```



“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, in dem man Sequenzen und Schleifen benutzt.



Beispiel zur Transformation in *Touch of Class*

Goto heute

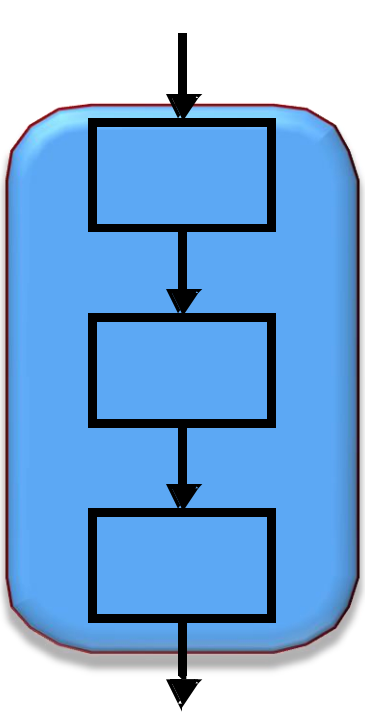
Fast allgemein verschrien.

Immer noch in einigen Programmiersprachen vorhanden.

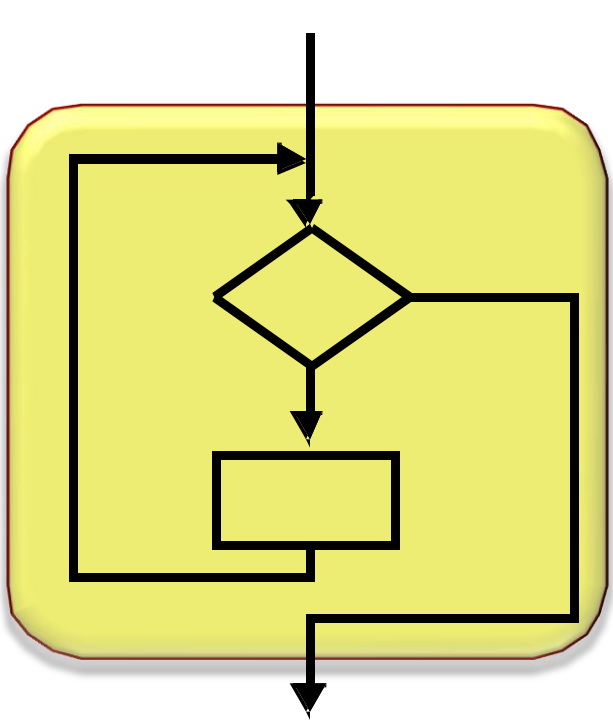
Es versteckt sich auch unter anderen Namen, z.B. **break**

```
loop
        ...
        if c then break end
        ...
end
```

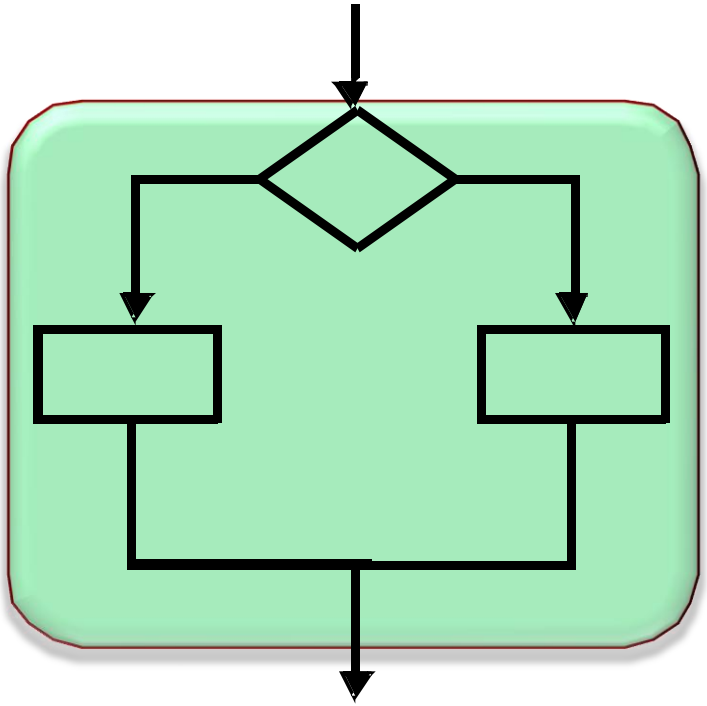
Ein Eingang, ein Ausgang



(Verbund)



(Schleife)



(Konditional)

Quiz: Finde die Invariante!

```
xxx (a, b: INTEGER): INTEGER is
        -- ??????????????????????????????????
    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 is
        -- 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
        Result := m
    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
                -- (Weitergeführt)
```

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)
```

Paradoxa in der Logik

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?

Das Paradox von Grelling

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

Das Lügner-Paradox

(Sehr Alt!)

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

Das Entscheidungsproblem und Unentscheidbarkeit

("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.)

Das Entscheidungsproblem in Eiffel

Nehmen Sie an, wir haben ein Feature

```
terminates (my_program: STRING): BOOLEAN
                -- Terminiert my_program ?
        do
                ... Your algorithm here ...
        end
```

Dann...

Wurzelprozedur des Systems:

```
what_do_you_think
        -- Terminate nur, falls nein.
  do
        from
        until
                not terminates ("/usr/home/turing.e")
        loop
        end
  end
```

Speichern Sie diesen Programmtext in /usr/home/turing.e

Das Entscheidungsproblem in der Praxis

Manche Programme terminieren in gewissen Fällen nicht.

Das ist ein Bug!

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

Was wir gesehen haben

- Der Begriff des Algorithmus
  - Grundlegende Eigenschaften
  - Unterschied zu Programm

- Der Begriff der Kontrollstruktur
- Korrektheit einer Instruktion
- Kontrollstruktur: Sequenz
- Kontrollstruktur: Konditional
- Verschachtelung, und wie sich diese vermeiden lässt