



Einführung in die Programmierung

Prof. Dr. Bertrand Meyer

Lektion 12: Rekursion

Definition

(nicht ganz ernst)

Eine Definition ist **rekursiv**, wenn sie
zumindest einen direkten Fall und einen
rekursiven Fall beinhaltet



```
class DOZENT feature
```

```
    erfolgreich: BOOLEAN
```

```
        -- Hat dieser Dozent die Rekursion klargemacht?
```

```
    anderer: like Current
```

```
        -- Ein Dozent, der vielleicht besser erklären kann.
```

```
erklaere_rekursion
```

```
    do
```

```
        erklare_grundlegende_idee
```

```
        if erfolgreich then
```

```
            erklare_mehr
```

```
        else
```

```
            anderer.erklare_rekursion
```

```
        end
```

```
    end
```

```
end
```





Eine Definition eines Konzepts ist **rekursiv**, falls sie selbst eine Instanz des Konzeptes beinhaltet

- Eine Definition kann mehr als eine Instanz des Konzeptes beinhalten
- *Rekursion* ist der Gebrauch einer rekursiven Definition
- Rekursion kann *direkt* oder *indirekt* sein



Eine Definition ist **rekursiv**, falls der *Definiens* das *Definiendum* beinhaltet

Die Geschichte des Universums*



*Édouard Lucas, *Récréations mathématiques*, Paris, 1883.

Dans le grand temple de Bénarès, sous le dôme qui marque le centre du monde, repose un socle de cuivre équipé de trois aiguilles verticales en diamant de 50 cm de haut.

A la création, Dieu enfila 64 plateaux en or pur sur une des aiguilles, le plus grand en bas et les autres de plus en plus petits. C'est la tour de Brahmâ.

Les moines doivent continûment déplacer les disques de manière que ceux-ci se retrouvent dans la même configuration sur une autre aiguille.

La règle de Brahmâ est simple: un seul disque à la fois et jamais un grand plateau sur un plus petit.

Arrivé à ce résultat, le monde tombera en poussière et disparaîtra.



**Übersetzung des Texts auf der vorigen Seite.*

Im Tempel von Bénarès, unterhalb des Doms, der das Zentrum der Welt kennzeichnet, stehen auf einer Kupferplatte drei diamantene Nadeln, 50 Zentimeter hoch.

Bei der Erschaffung hat Gott 64 Scheiben aus reinem Gold auf eine dieser Nadeln gesteckt, die grösste zuunterst, und darauf die restlichen, immer kleiner. Dies ist der Turm von Brahmâ.

Die Mönche müssen die Scheiben stets versetzen, bis diese in der gleichen Konfiguration auf einer anderen Nadel liegen.

Die Regel von Brahmâ ist einfach: Nur eine Scheibe pro Schritt, und niemals eine grössere Scheibe auf eine kleinere legen.

Sobald sie dieses Ziel erreichen, wird die Welt zu Staub zerfallen und für immer verschwinden.

Die Türme von Hanoi



Wieviele Schritte?



Annahme: n Scheiben ($n \geq 0$); 3 Nadeln *quelle*, *ziel*, *andere*

Die grösste Scheibe kann nur von *quelle* nach *ziel* bewegt werden, falls Letzteres leer ist; alle anderen Scheiben müssen auf *andere* liegen.

Also ergibt sich die minimale Anzahl Schritte zu:

$n - 1$ von *quelle* nach *andere* bewegen

grösste von *quelle* nach *ziel* bewegen

$$\begin{aligned} H_n &= H_{n-1} + 1 + H_{n-1} \\ &= 2 * H_{n-1} + 1 \end{aligned}$$

$n - 1$ von *andere* nach *ziel* bewegen

Da $H_1 = 1$:

$$H_n = 2^n - 1$$

Diese Logik ergibt einen Algorithmus!

hanoi (*n* : INTEGER ; *quelle*, *ziel*, *andere* : CHARACTER)
-- Verschiebe *n* Scheiben von *quelle* nach *ziel*,
-- mit *andere* als Zwischenspeicher.

require

nicht_negativ: $n \geq 0$

ungleich1: *quelle* \neq *ziel*

ungleich2: *ziel* \neq *andere*

ungleich3: *quelle* \neq *andere*

do

if $n > 0$ **then**

hanoi ($n - 1$, *quelle*, *andere*, *ziel*)

umstelle (*quelle*, *ziel*)

hanoi ($n - 1$, *andere*, *ziel*, *quelle*)

end

end

Rekursive Aufrufe



Die Türme von Hanoi



Eine mögliche Implementation für *umstelle*

umstelle (*quelle*, *ziel* : *CHARACTER*)

-- Verschieben die Spitz-Scheibe von *quelle* nach *ziel*.

require

ungleich: *quelle* /= *ziel*

-- Auf *quelle* liegt mindestens eine Scheibe,

-- die kleiner ist (wenn *ziel* nicht leer ist) als die

-- Scheibe auf dem Spitz von *ziel*

do

io.put_string (" Von ")

io.put_character (*quelle*)

io.put_string (" nach ")

io.put_character (*ziel*)

io.put_new_line

end

Die Türme von Hanoi



Ein Beispiel



Ausführung des Aufrufs

hanoi (4, 'A', 'B', 'C')

wird eine Sequenz von 15 ($2^4 - 1$) Instruktionen ausgegeben:

Von *A* nach *C*

Von *A* nach *B*

Von *C* nach *B*

Von *A* nach *C*

Von *B* nach *A*

Von *B* nach *C*

Von *A* nach *C*

Von *A* nach *B*

Von *C* nach *B*

Von *C* nach *A*

Von *B* nach *A*

Von *C* nach *B*

Von *A* nach *C*

Von *A* nach *B*

Von *C* nach *B*



- Rekursive Routine
- Rekursive Grammatik
- Rekursiv definiertes Programmierkonzept
- Rekursive Datenstrukturen
- Rekursiver Beweis



Direkte Rekursion: Der Rumpf beinhaltet einen Aufruf der Routine selbst.

Beispiel: Die Routine *hanoi* der vorangegangenen Lösung des Problems der Türme von Hanoi



Ein **Typ** ist:

- Entweder eine nicht-generische Klasse, z.B.
STATION
- Oder eine **generische Ableitung**, z.B. der Name einer Klasse, gefolgt von einer Liste von **Typen**, die **tatsächlichen generischen Parameter**, in Klammern, z.B.

LIST [STATION]

LIST [ARRAY [STATION]]

Die Bottom-Up Interpretation (von unten nach oben)



Nehmen wir vier Klassen an:

STATION (S), INTEGER (I)

-- Nicht generisch

LIST (L), ARRAY (A)

-- Generisch

Was sind die mögliche Typen?

Definition von *Typ*:

A. Eine nicht generische Klasse

B. *C [T]* mit generischer *C* und *Typ T*

S *I*

--

Schritt 1

L [S] *L [I]* *A [S]* *A [I]*

-- Schritt 2

L [L [S]] *L [L [I]]* *L [A [S]]* *L [A [I]]*

A [L [S]] *A [L [I]]* *A [A [S]]* *A [A [I]]*

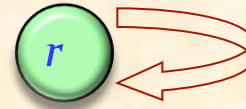
-- Schritt 3

...

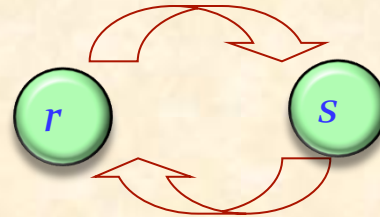
Rekursion: direkt und indirekt



Routine r ruft sich selbst auf



r ruft s auf und s ruft r auf



r_1 ruft r_2 auf ... r_n ruft r_1 auf





Instruktion ::= *Zuweisung* | **Konditional** | *Verbund* | ...

Konditional ::= **if** *Ausdruck* **then** **Instruktion**
 else **Instruktion** **end**

Die lexikographische Ordnung definieren

Sie können diesen Begriff in der Praxis verwenden:

- *ABC* kommt vor *DEF*
- *AB* kommt vor *DEF*
- *A* kommt vor *AB*
- *A* kommt vor *ABC*
- Leeres Wort kommt vor *ABC*

Die Aufgabe ist, eine Definition vorzuschlagen. Wir verlangen nicht eine konstruktive Methode (ein Algorithmus), z.B.

„Wir betrachten die ersten Buchstaben x_1 und y_1 von x und y ; wenn x_1 kleiner als y_1 ist, dann kommt x vor y ; sonst wenn x_1 gleich y_1 ist, dann... (usw.)“

sondern eine echte Definition, in der folgende Form:

*„Das Wort x kommt **„vor“** dem Wort y genau dann, wenn...“*

Die Lexikographische Ordnung definieren



Problem: Definiere den Begriff, dass das Wort w_1 **vor** dem Wort w_2 kommt, nach alphabetischer Ordnung.

Konventionen:

- Ein **Wort** ist eine Sequenz von null oder mehr Buchstaben.
- Ein **Buchstabe** ist eines der folgenden Zeichen:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

- Für alle Paare von Buchstaben ist bekannt, welcher "*kleiner*" als der andere ist. Die Ordnung ist jene der vorherigen Liste.

Die lexikographische Ordnung: Beispiele



ABC kommt vor *DEF*

AB kommt vor *DEF*

A kommt vor *AB*

A kommt vor *ABC*

Leeres Wort kommt vor *ABC*

Eine rekursive Definition



Das Wort x kommt “**vor**” dem Wort y genau dann, wenn sie eine der folgenden Bedingungen erfüllen:

- x ist leer und y ist nicht leer
- Weder x noch y sind leer, und der erste Buchstabe von x ist *kleiner* als der erste Buchstabe von y
- Weder x noch y sind leer und (beide Bedingungen):
 - Ihre ersten Buchstaben sind die Gleichen
 - Das Wort, das man erhält, wenn man den ersten Buchstaben von x weglässt, kommt **vor** dem Wort, das man durch das Weglassen des ersten Buchstabens von y erhält

Als eine rekursive Funktion ausgedrückt

```
vor alias "<" (y : STRING) : BOOLEAN
```

-- Kommt diese Kette vor *y* in lexikographischer Ordnung?

do

Result :=

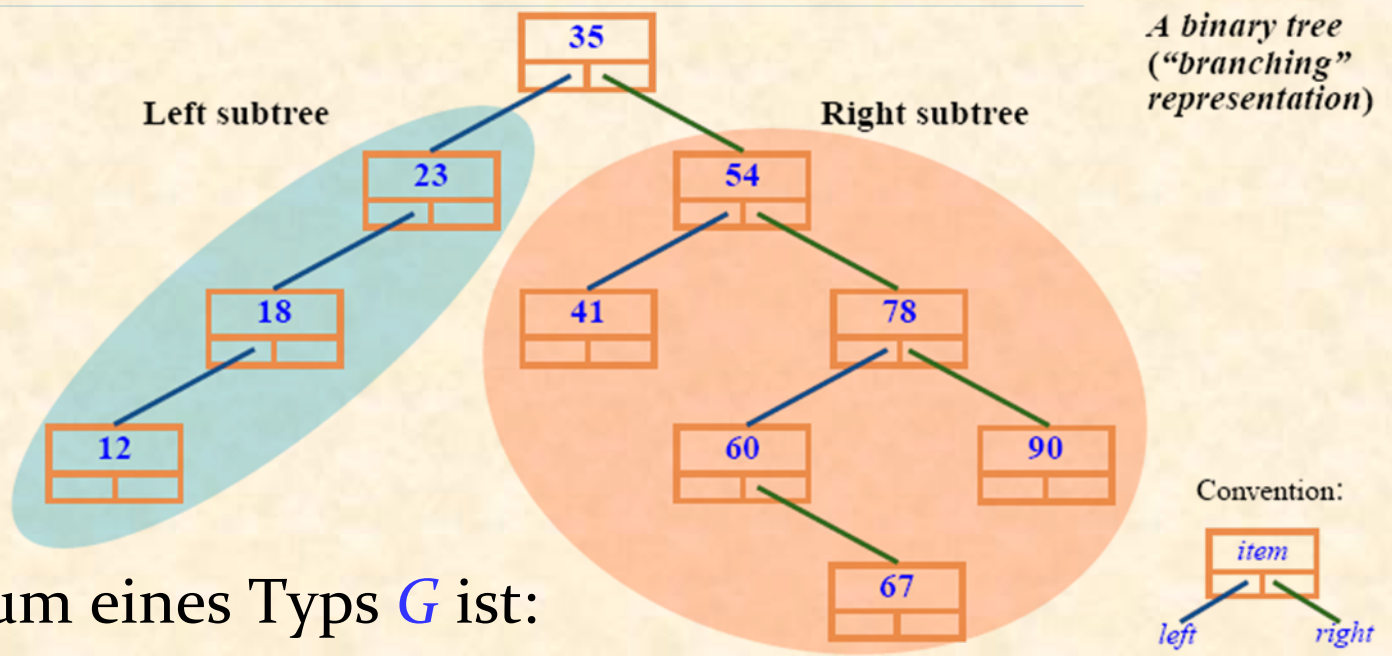
```
((is_empty and not y.is_empty) or  
((not is_empty and not y.is_empty) and then  
((first < y.first) or  
((first = y.first) and (rest .vor (y.rest))
```

```
)  
)  
)  
end
```

Erstes Zeichen (vom
Typ *STRING*)

Der Rest des Wortes
(vom Typ *STRING*)

Eine rekursive Datenstruktur



Ein binärer Baum eines Typs G ist:

- Entweder leer
- Oder ein *Knoten*, bestehend aus drei disjunkten Teilen:
 - Ein Wert des Typs G : Die Wurzel
 - Ein **binärer** des Typs G , der **linke Teilbaum***
 - Ein **binärer** des Typs G , der **rechte Teilbaum**
Baum

*Engl.: *Subtree*



Theorem: Mit jedem Knoten jedes binären Baumes können wir einen binären Baum assoziieren, sodass sie eins zu eins übereinstimmen

Beweis:

- Falls der Baum leer ist: trivial
- Andernfalls:
 - Assoziiere die Wurzel mit dem gesamten Baum
 - Jeder andere Knoten n ist entweder im linken oder im rechten Teilbaum; assoziiere n mit dem in diesem Teilbaum **assozierten** Baum

Konsequenz: Wir können über linke und rechte Teilbäume eines **Knoten** reden

Skelett einer Klasse eines Binärbaums



```
class BINARY_TREE [G]
```

```
feature
```

```
  item : G
```

```
  left : BINARY_TREE [G]
```

```
  right : BINARY_TREE [G]
```

... Andere Anfrage, Einfüge- und Löschbefehle ...

```
end
```


Rekursive Routine auf einer rekursiven Datenstruktur



count : INTEGER

-- Anzahl Knoten.

do

Result := 1

if *left* /= Void then

Result := Result + *left.count*

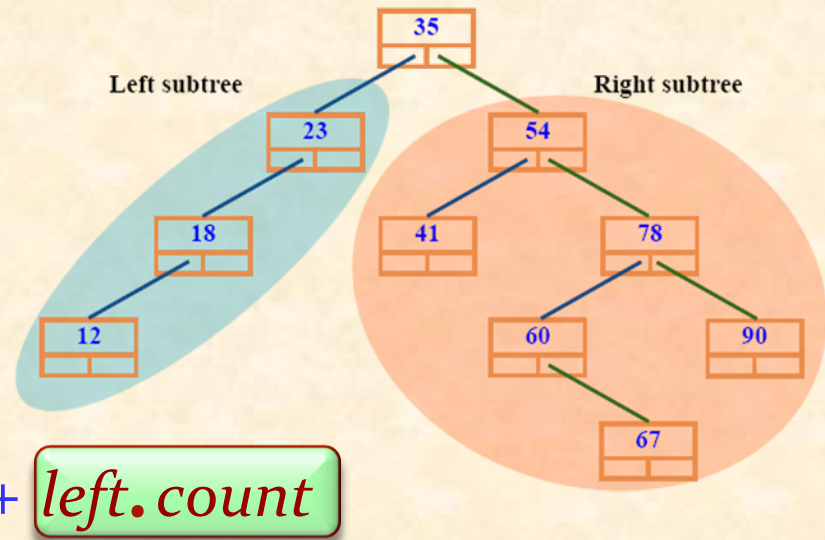
end

if *right* /= Void then

Result := Result + *right.count*

end

end



Traversieren eines Binärbaums



print_all

-- Alle Knotenwerte ausgeben.

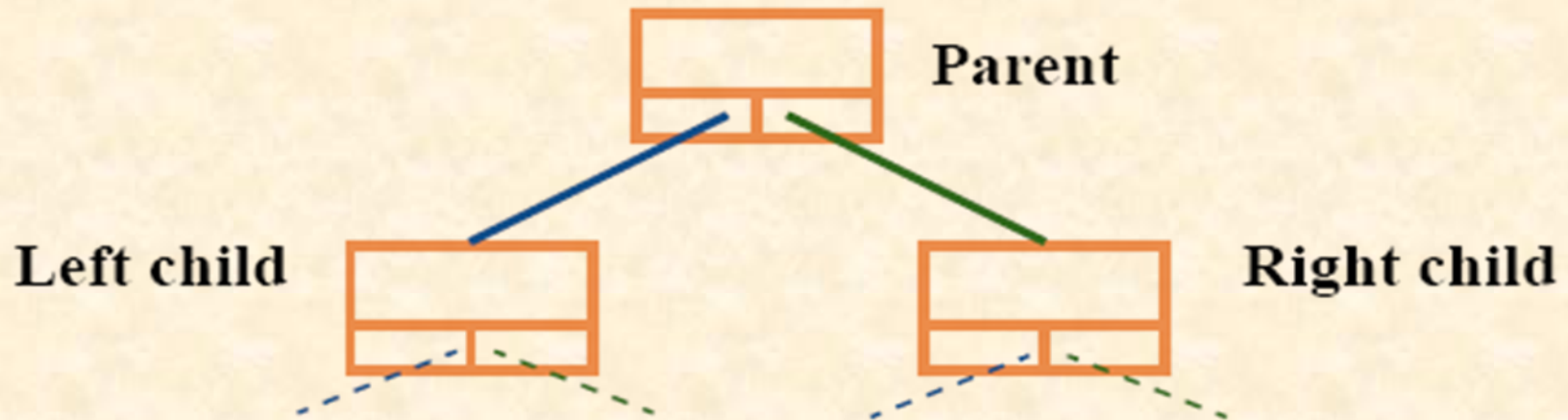
do

if *left* /= Void **then** *left.print_all* **end**

print (*item*)

if *right* /= Void **then** *right.print_all* **end**

end



Theorem: nur ein Vater(-knoten) (parent)

Jeder Knoten in einem binären Baum hat genau einen Vater, ausser die Wurzel, die hat keinen

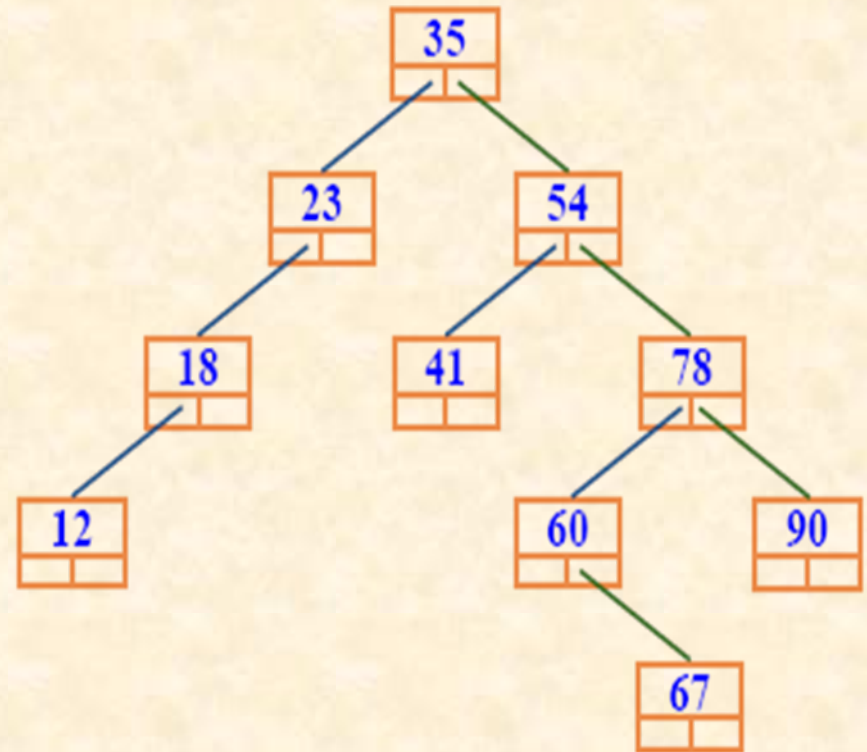
Mehr Eigenschaften und Terminologie



Ein Knoten eines Binärbaumes kann ...

- Ein linkes und ein rechtes Kind
- Nur ein linkes Kind
- Nur ein rechtes Kind
- Kein Kind

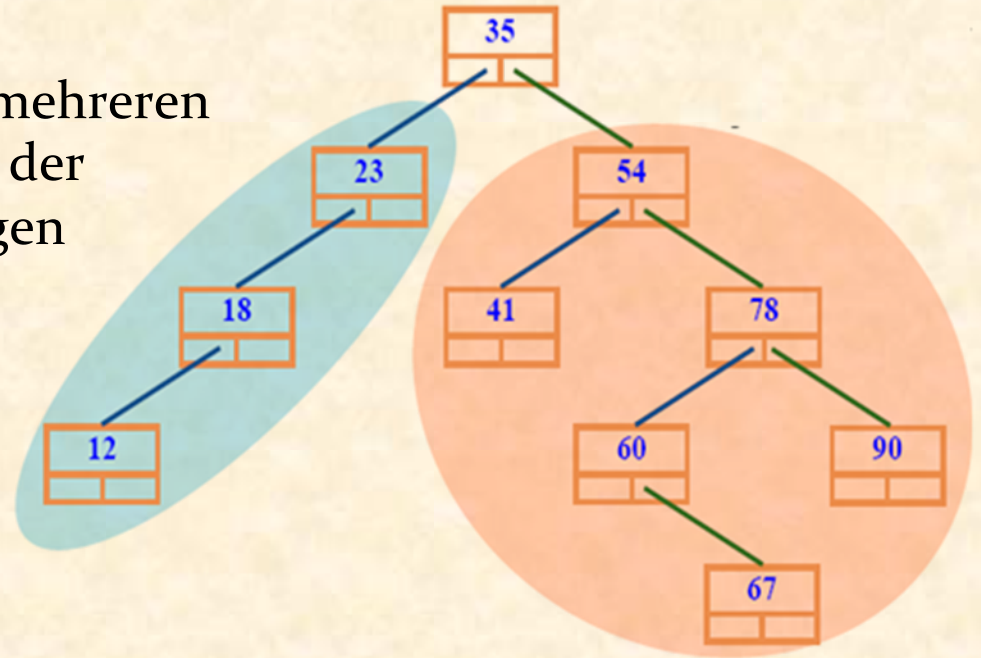
... haben



Mehr Eigenschaften und Terminologie

Aufwärtspfad (*upward path*):

- Eine Sequenz von einem oder mehreren Knoten, wobei jeder Knoten in der Sequenz der Vater des vorherigen ist (falls vorhanden)
- Analog: Abwärtspfad (*downward path*)



Theorem: Wurzelpfad

- Von jedem Knoten in einem Binärbaum gibt es einen einzigen Aufwärtspfad zu der Wurzel

Theorem: Abwärtspfad

- Für jeden Knoten in einem Binärbaum gibt es einen einzigen Abwärtspfad, der die Wurzel mit dem Knoten durch aufeinanderfolgende Anwendungen von *left* und *right* verbindet

Die Höhe eines Binärbaums

Die maximale Anzahl Knoten auf einem Abwärtspfad von der Wurzel zu einem Blatt

height : INTEGER

- Maximale Anzahl Knoten
- auf einem Abwärtspfad.

local

lh, rh : INTEGER

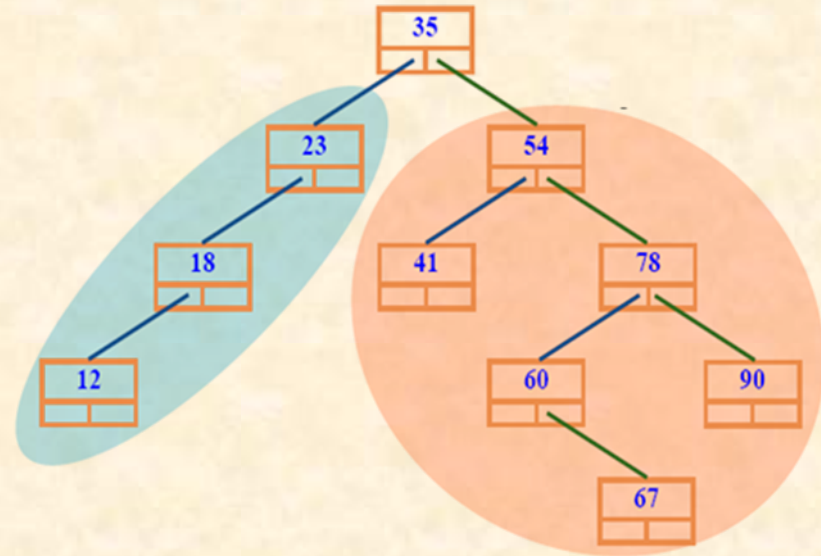
do

if *left* /= Void then *lh* := *left.height* end

if *right* /= Void then *rh* := *right.height* end

Result := 1 + *lh.max* (*rh*)

end



Operationen auf binären Bäumen



```
add_left (x : G)  
    -- Erzeuge linkes Kind mit Wert x.  
    require  
        no_left_child_behind: left = Void  
    do  
        create left.make (x)  
    end
```

add_right (*x* : *G*) ...gleiches Muster...

```
make (x : G)  
    -- Initialisiere mit Wert x.  
    do  
        item := x  
    ensure  
        set: item = x  
    end
```

Traversieren eines Binärbaums

Symmetrische Reihenfolge (inorder):

traversiere linken Teilbaum

besuche die Wurzel

traversiere rechten Teilbaum

Hauptreihenfolge (preorder):

besuche die Wurzel

traversiere links

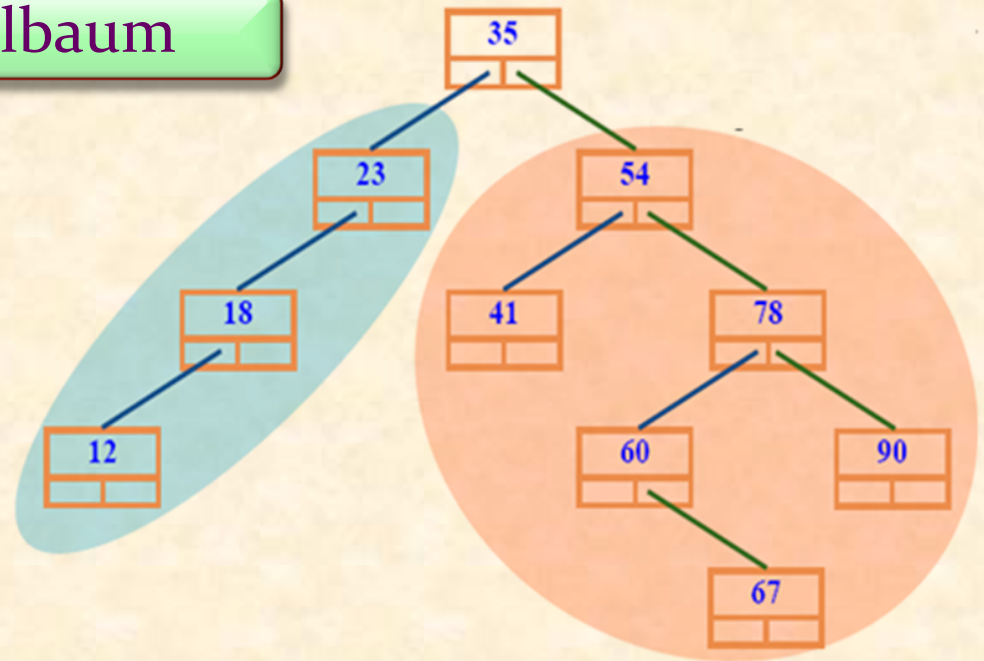
traversiere rechts

Nebenreihenfolge (postorder):

traversiere links

traversiere rechts

besuche die Wurzel



Binärer Suchbaum

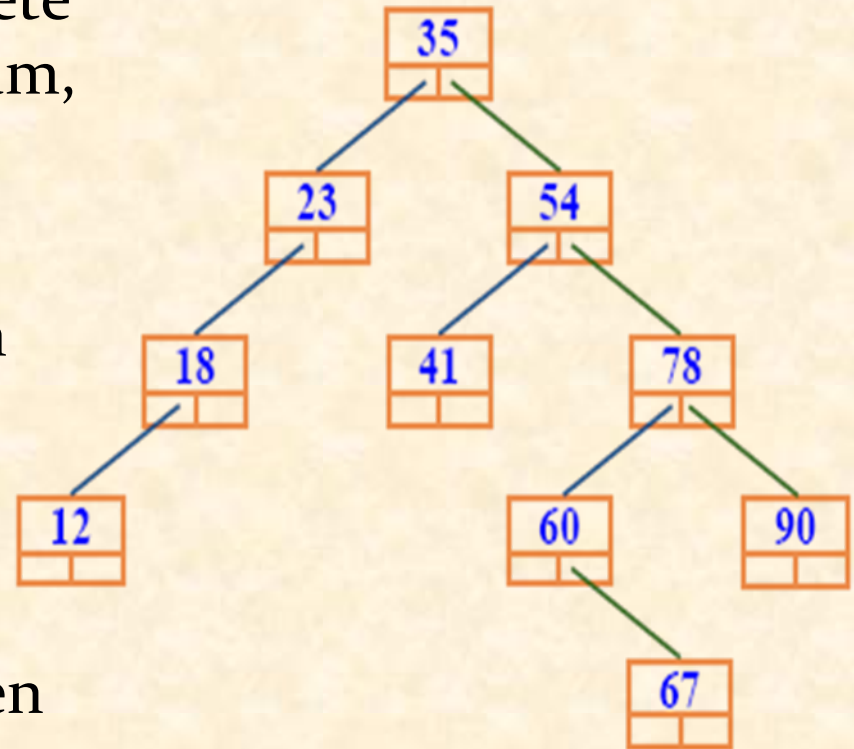
Ein Binärbaum über eine geordnete Menge G ist ein binärer Suchbaum, falls für jeden Knoten n gilt:

- Für jeden Knoten x des linken Teilbaums von n :

$$x.item < n.item$$

- Für jeden Knoten x des rechten Teilbaums von n :

$$x.item \geq n.item$$



Elemente geordnet ausgeben

```
class BINARY_SEARCH_TREE [G ...] feature
```

```
  item : G
```

```
  left, right : BINARY_SEARCH_TREE [G]
```

```
  print_sorted
```

```
    -- Elemente geordnet ausgeben.
```

```
  do
```

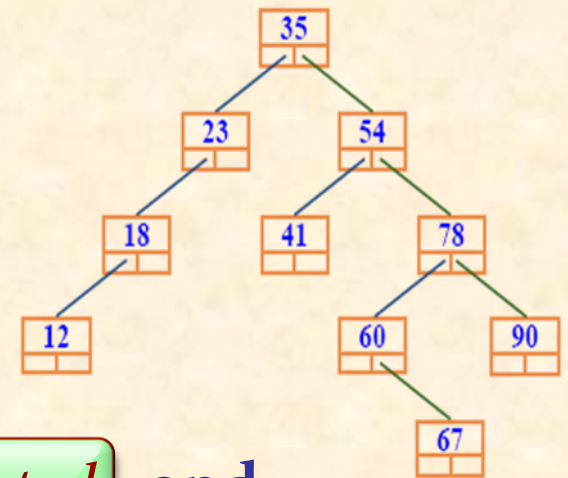
```
    if left /= Void then left.print_sorted end
```

```
    print (item)
```

```
    if right /= Void then right.print_sorted end
```

```
  end
```

```
end
```



Suchen in einem Binärbaum

```
class BINARY_SEARCH_TREE [G ...] feature  
  item : G
```

```
  left, right : BINARY_SEARCH_TREE [G]
```

```
  has (x : G) : BOOLEAN
```

```
    -- Kommt x in einem Knoten vor?
```

```
  require
```

```
    argument_exists: x /= Void
```

```
  do
```

```
    if x = item then
```

```
      Result := True
```

```
    elseif x < item and left /= Void then
```

```
      Result := left.has (x)
```

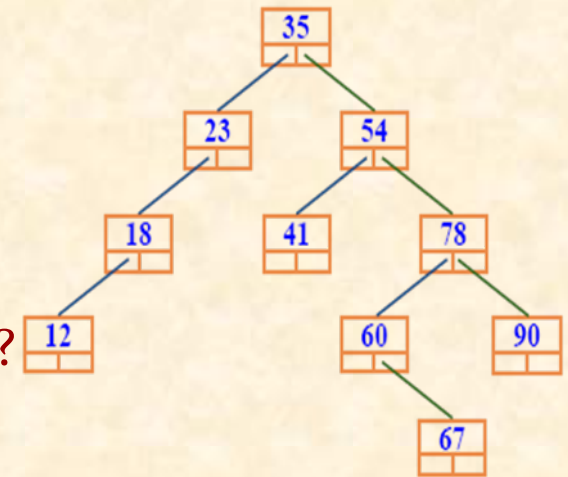
```
    elseif x > item and right /= Void then
```

```
      Result := right.has (x)
```

```
    end
```

```
  end
```

```
end
```



Einfügen in einen Binärbaum



Versuchen Sie dies als Übung!

Wieso binäre Suchbäume?



Lineare Strukturen: Einfügen, Suchen und Löschen:

$$O(n)$$

Binärer Suchbaum: durchschnittlich für Einfügen, Suchen und Löschen: $O(\log(n))$

Aber: Worst case: $O(n)$! (Denkübung: Wann?)

- Verbesserungen: Rot-Schwarz-Bäume, AVL-Bäume

Messungen der Komplexität: best case, average case, worst case



Eine nützliche rekursive Definition sollte folgendes sicherstellen:

- **R₁** Es gibt mindestens einen nicht-rekursiven Zweig
- **R₂** Jeder rekursive Zweig tritt in einem vom Originalkontext verschiedenen Kontext auf
- **R₃** Für jeden rekursiven Zweig gilt, dass der Kontextwechsel (**R₂**) ihn näher zu einem nicht-rekursiven Zweig bringt (**R₁**)

“Hanoi” ist wohlgeformt

hanoi (*n* : INTEGER ; *quelle*, *ziel*, *andere* : CHARACTER)

-- Verschiebe *n* Scheiben von *quelle* nach *ziel*,
-- mit *andere* als Zwischenspeicher.

require

nicht_negativ: $n \geq 0$

ungleich1: *quelle* \neq *ziel*

ungleich2: *ziel* \neq *andere*

ungleich3: *quelle* \neq *andere*

do

if $n > 0$ **then**

hanoi ($n - 1$, *quelle*, *andere*, *ziel*)

umstelle (*quelle*, *ziel*)

hanoi ($n - 1$, *andere*, *ziel*, *quelle*)

end

end

Was wir bisher gesehen haben:



Eine Definition ist rekursiv, falls sie den eigenen Begriff auf ein kleineres Ziel anwendet

Was alles rekursiv sein kann: Eine Routine, die Definition eines Konzeptes...

Immer noch einiges unklar: Besteht nicht die Gefahr einer zyklischen Definition?



Eine rekursive Routine sollte eine Rekursionsvariante benutzen, eine ganze Zahl, die mit jedem Aufruf verbunden wird, so dass:

- Die Variante ist immer ≥ 0 (Aus der Vorbedingung)
- Falls die Variante zu Beginn einer Routinenausführung den Wert v hat, gilt für den Wert v' für alle rekursiven Aufrufe

$$0 \leq v' < v$$

Hanoi: Wie lautet die Variante?



hanoi (*n* : INTEGER ; *quelle*, *ziel*, *andere* : CHARACTER)
-- Verschiebe *n* Scheiben von *quelle* nach *ziel*,
-- mit *andere* als Zwischenspeicher.

require

...

do

if *n* > 0 then

hanoi (*n* - 1, *quelle*, *andere*, *ziel*)

umstelle (*quelle*, *ziel*)

hanoi (*n* - 1, *andere*, *ziel*, *quelle*)

end

end

Suchbäume ausgeben: Was ist die Variante?

```
class BINARY_SEARCH_TREE [G ...] feature
```

```
  item : G
```

```
  left, right : BINARY_SEARCH_TREE [G]
```

```
  print_sorted
```

```
    -- Elemente geordnet ausgeben.
```

```
  do
```

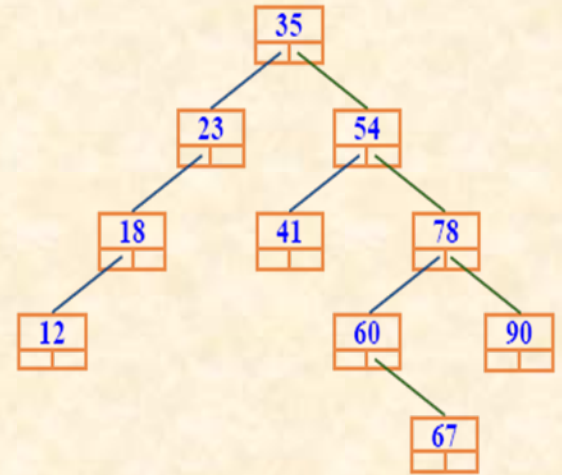
```
    if left /= Void then left.print_sorted end
```

```
    print (item)
```

```
    if right /= Void then right.print_sorted end
```

```
  end
```

```
end
```



Verträge für rekursive Routinen

hanoi (n : INTEGER; *quelle*, *ziel*, *andere* : CHARACTER)

-- Verschiebe n Scheiben von *quelle* nach *ziel*,

-- mit *andere* als Zwischenspeicher.

-- **variant:** n

-- **invariant:** Die Scheiben auf jeder Nadel sind

-- abnehmend in ihrer Grösse

require

...

do

if $n > 0$ **then**

hanoi ($n - 1$, *quelle*, *andere*, *ziel*)

umstelle (*quelle*, *ziel*)

hanoi ($n - 1$, *andere*, *ziel*, *quelle*)

end

end

McCarthy's 91-Funktion



$M(n) =$

- $n - 10$ falls $n > 100$
- $M(M(n + 11))$ falls $n \leq 100$

Noch eine Funktion



$bizarr(n) =$

➤ 1

falls $n = 1$

➤ $bizarr(n / 2)$

falls n gerade ist

➤ $bizarr((3 * n + 1) / 2)$

falls $n > 1$ und n ungerade ist



$$fib(1) = 0$$

$$fib(2) = 1$$

$$fib(n) = fib(n-2) + fib(n-1) \quad \text{for } n > 2$$

Fakultät (*factorial*)



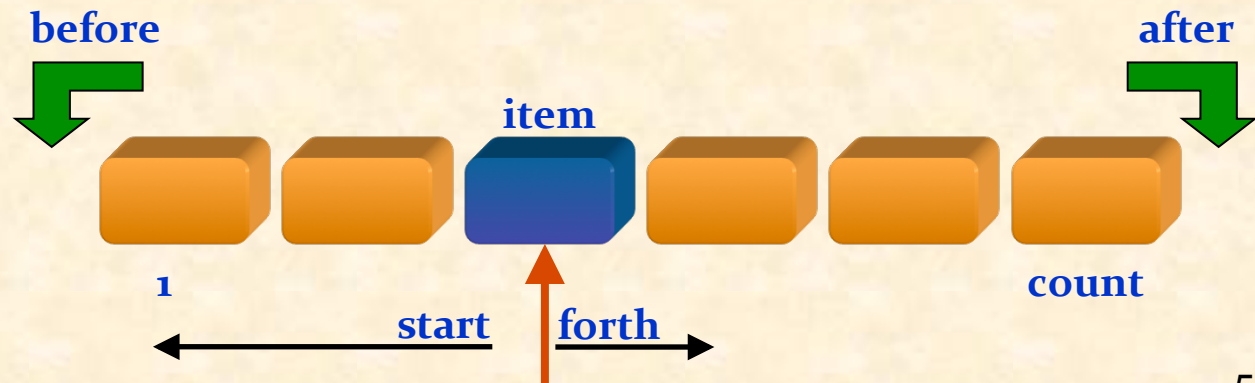
$$0! = 1$$

$$n! = n * (n - 1)! \quad \text{for } n > 0$$

Die rekursive Definition ist nur für Demonstrationszwecke interessant; in der Praxis werden Schleifen (oder Tabellen) benutzt

Unser ursprüngliches Schleifenbeispiel

```
highest_name : STRING
    -- Alphabetisch grösster Stationsname von Line8.
    local
        c : ITERATION_CURSOR [STATION]
    do
        from
            c := Line8.new_cursor; Result := ""
        until
            c.after
        loop
            Result := greater (Result, c.item.name)
        c.forth
    end
end
```



Eine rekursive Alternative



highest_name : *STRING*

-- Alphabetisch grösster Stationsname von *Line8*.

require

not *Line8.is_empty*

do

Result := *highest_from_cursor* (*Line8.new_cursor*)

end

Hilfsfunktion für Rekursion

```
highest_from_cursor (c: ITERATION_CURSOR [STATION]): STRING
```

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

```
-- von der Cursorposition c ausgehend.
```

```
require
```

```
c /= Void and not c.off
```

```
do
```

```
Result := c.item.name
```

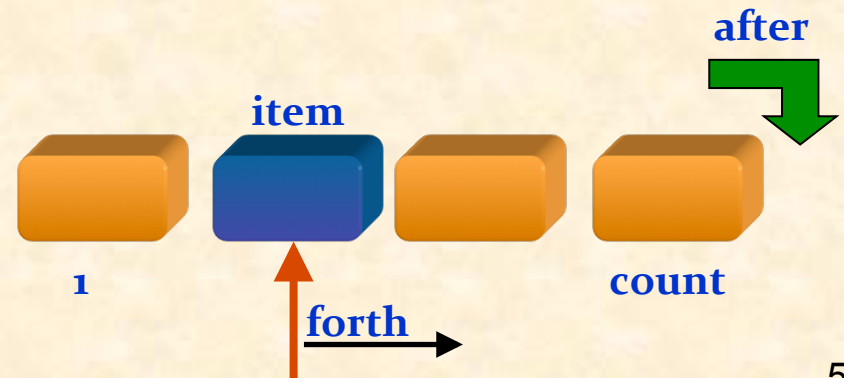
```
c.forth
```

```
if not c.after then
```

```
Result := greater (Result, highest_from_cursor (c))
```

```
end
```

```
end
```



Eine Schleifenversion mit Argumenten



```
maximum (a : ARRAY [STRING]): STRING  
    -- Alphabetisch grösstes Element in a.  
require  
    a.count >= 1  
local  
    i : INTEGER  
do  
    from  
        i := a.lower + 1; Result := a.item (a.lower)  
    invariant  
        i > a.lower; i <= a.upper + 1  
        -- Result ist das Maximum der Elemente von a [a.lower .. i - 1]  
    until  
        i > a.upper  
    loop  
        if a.item (i) > Result then Result := a.item (i) end  
        i := i + 1  
    end  
end
```

Rekursive Version



```
maxrec (a : ARRAY [STRING]): STRING
```

```
-- Alphabetisch grösstes Element in a.
```

```
require
```

```
a.count >= 1
```

```
do
```

```
Result := max_from (a, a.lower)
```

```
end
```

```
max_from (a : ARRAY [STRING]; i : INTEGER): STRING
```

```
-- Alphabetisch grösstes Element in a, von Index i ausgehend.
```

```
require
```

```
i >= a.lower; i <= a.upper
```

```
do
```

```
Result := a.item (i)
```

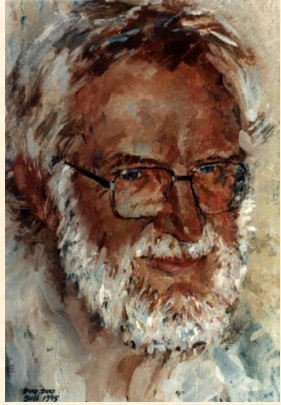
```
if i < a.upper then
```

```
Result := greater (Result, max_from (a, i + 1))
```

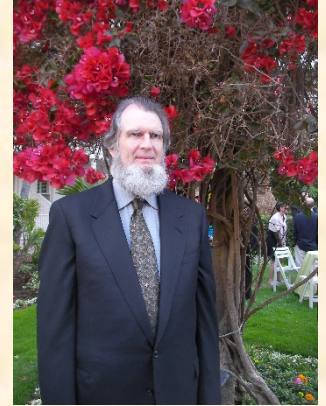
```
end
```

```
end
```

Jim Horning (details in *Touch of Class*)



In the summer of 1961 I attended a lecture in Los Angeles by a little-known Danish computer scientist. His name was Peter Naur and his topic was the new language Algol 60. In the question period, the man next to me stood up. “It seems to me that there is an error in one of your slides.”



Peter was puzzled, “No, I don’t think so. Which slide?”

“The one that shows a routine calling itself. That’s impossible to implement.”

Peter was even more puzzled: “But we have implemented the whole language, and run all the examples through our compiler.”

The man sat down, still muttering to himself, “Impossible! Impossible!”. I suspect that much of the audience agreed with him.

At the time it was fairly common practice to allocate statically the memory for a routine’s code, its local variables and its return address. The call stack had been independently invented at least twice in Europe, under different names, but was still not widely understood in America.

$r(n)$

do

... Einige Instruktionen...

$n := n'$
goto *start_of_r*

$r(n')$

-- z.B. $r(n - 1)$

... Mehr Instruktionen ...

end

Vielleicht
brauchen wir n !

Nach einem Aufruf müssen die vorherigen Werte der Argumente und andere Kontextinformationen wiederhergestellt werden

Einen Stapel (*stack*) benutzen

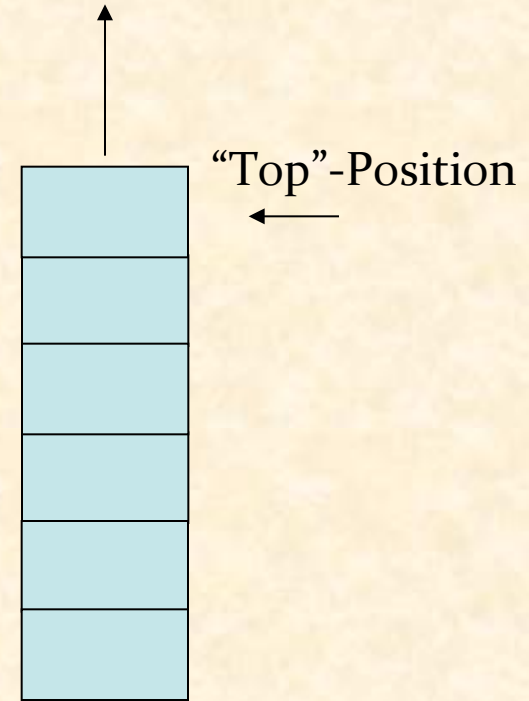


Abfragen:

- Ist der Stapel leer? *is_empty*
- Oberstes Element (falls vorhanden):
top *item*

Befehle:

- Ein Element zuoberst einfügen: *push* *put*
- Oberstes Element entfernen (falls vorhanden): *pop* *remove*



Vor einem Aufruf: Ein Frame wird auf den Stapel “gepusht”. Es beinhaltet die Werte von lokalen Variablen, Argumenten und Rückgabeinformationen

Nach einem Aufruf: Das Frame wird vom Stapel abgerufen und die Werte wiederhergestellt

Rekursionseliminierung



$r(n)$

do

start:

... Einige Instruktionen ...

-- $r(n')$

Push Frame

$n := n'$

goto start

after:

... Mehr Instruktionen ...

end

if *Stapel nicht leer* **then**

Pop Frame

goto after

end



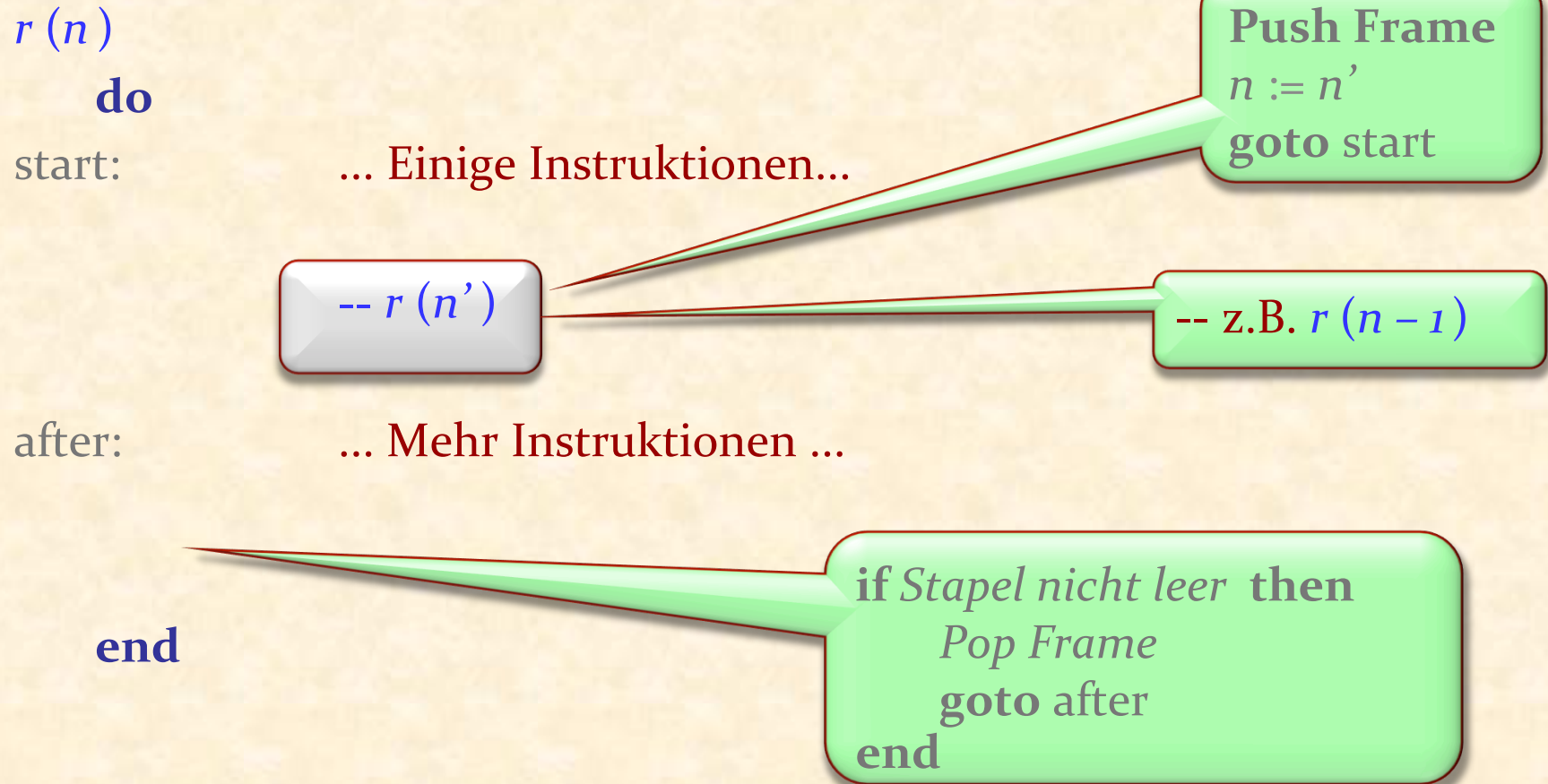
Rekursive Aufrufe bringen (in einer Standardimplementierung ohne Optimierungen) zur Laufzeit eine Performanceeinbusse mit sich: Der Stapel der konservierten Werte muss aufrecht erhalten werden

Verschiedene Optimierungen sind möglich

Manchmal kann ein rekursives Schema durch eine Schleife ersetzt werden. Dies nennt man **Rekursionseliminierung**

“**Endrecursion**” (Die letzte Instruktion einer Routine ist ein rekursiver Aufruf) kann meistens eliminiert werden

Die Stapel-Nutzung minimieren



Man muss simple Transformationen nicht auf dem Stapel speichern oder von dort abrufen. Z.B. $n := n - 1$, und umgekehrt $n := n + 1$



Anwendbar, falls Sie eine Lösung eines Problems aus Lösungen für kleinere Teilprobleme zusammensetzen können



- Definition des Begriffs der Rekursion
- Viele rekursive Routinen
- Rekursive Datenstrukturen
- Rekursive Beweise
- Die Anatomie eines rekursiven Algorithmus: Die Türme von Hanoi
- Was eine rekursive Definition „wohlgeformt“ macht
- Binärbäume
- Binäre Suchbäume
- Anwendungen von Rekursion
- Grundlagen von Rekursionsimplementierung