Kommentar:

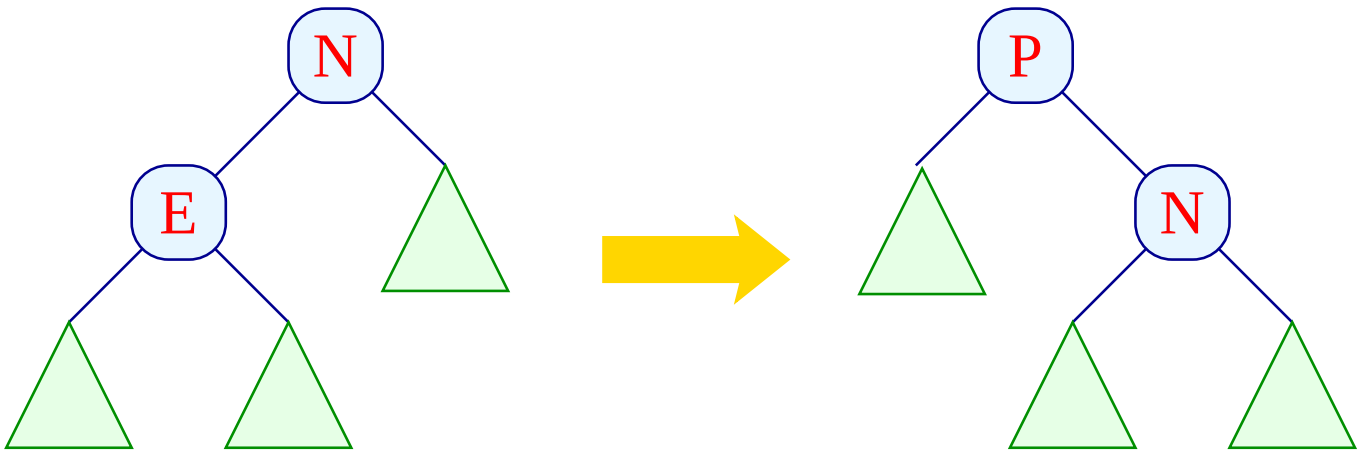
- Einfügen in den flacheren Teilbaum erhöht die Gesamttiefe nie :-)

  Gegebenenfalls werden aber beide Teilbäume gleich tief.

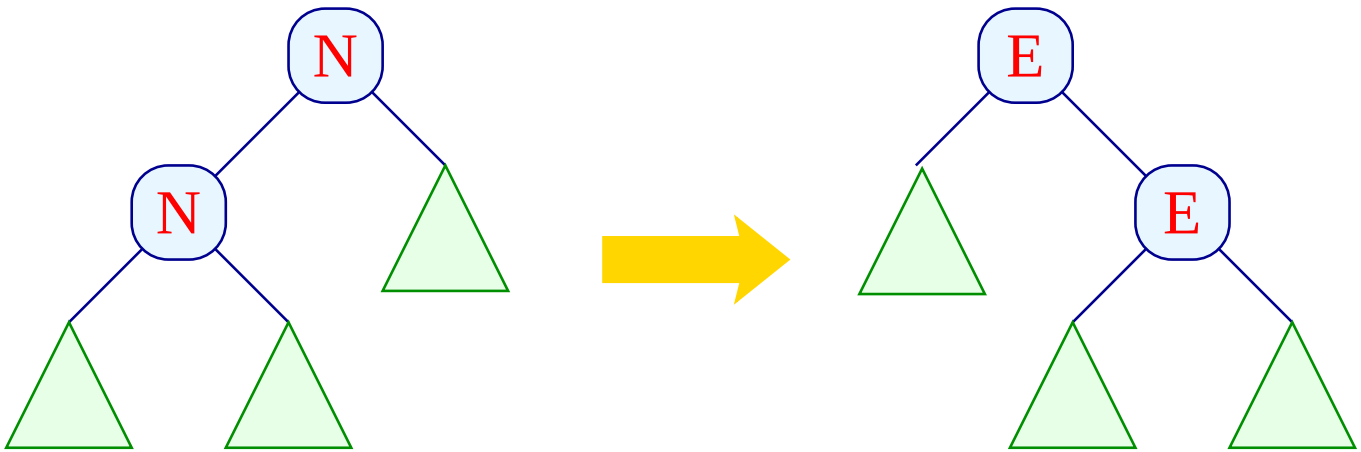
- Einfügen in den tieferen Teilbaum kann dazu führen, dass der Tiefenunterschied auf 2 anwächst :-(

  Dann rotieren wir Knoten an der Wurzel, um die Differenz auszugleichen ...

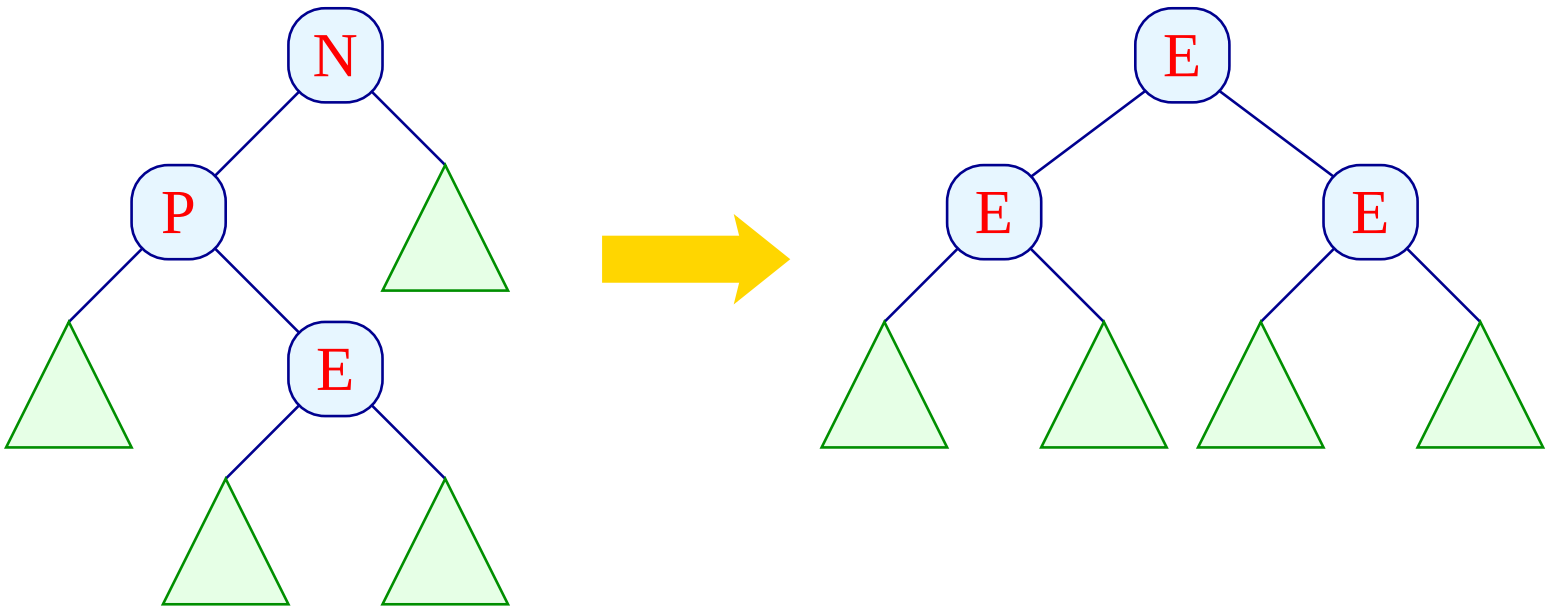
`rotateRight:`



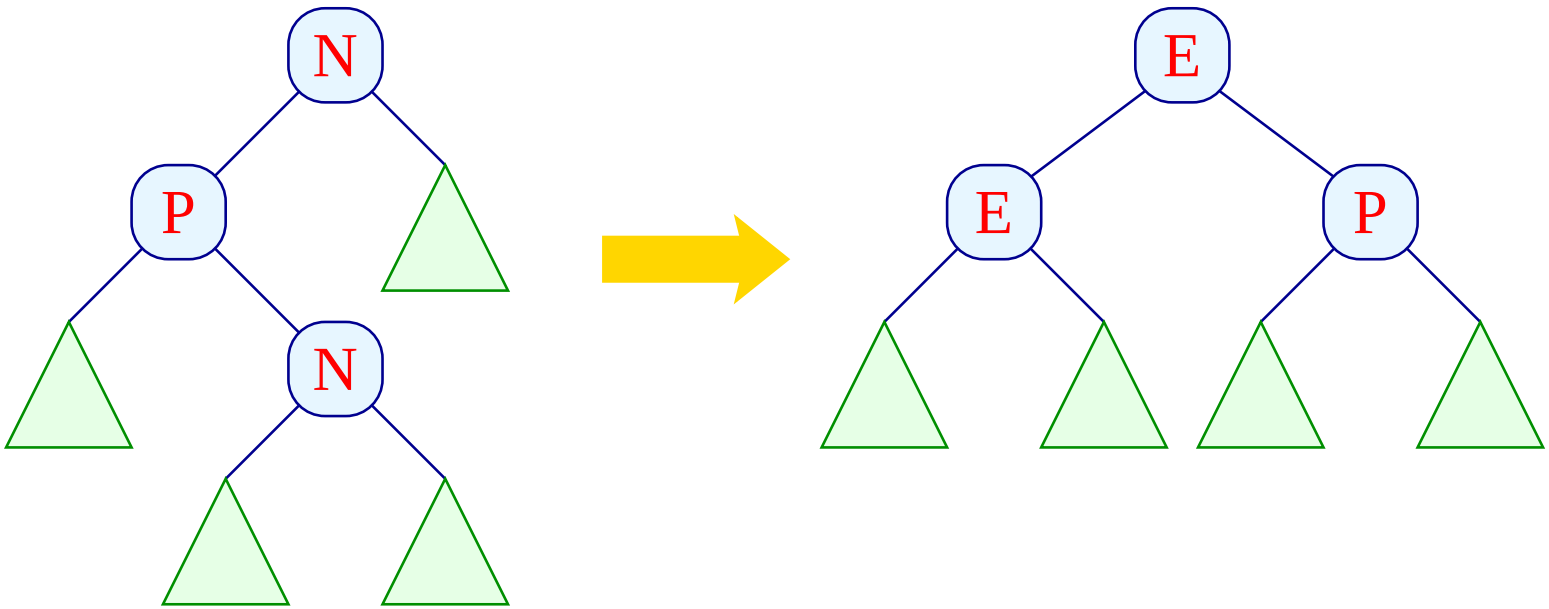
`rotateRight:`



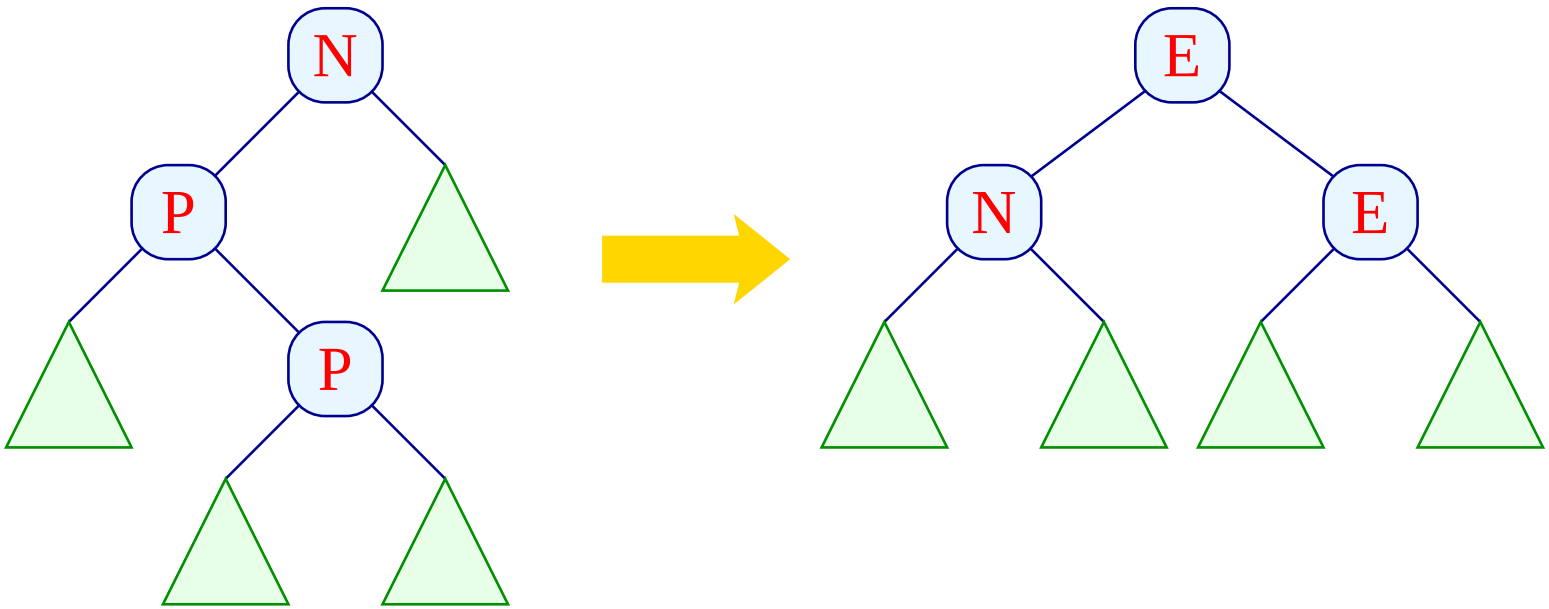
`rotateRight:`



`rotateRight:`



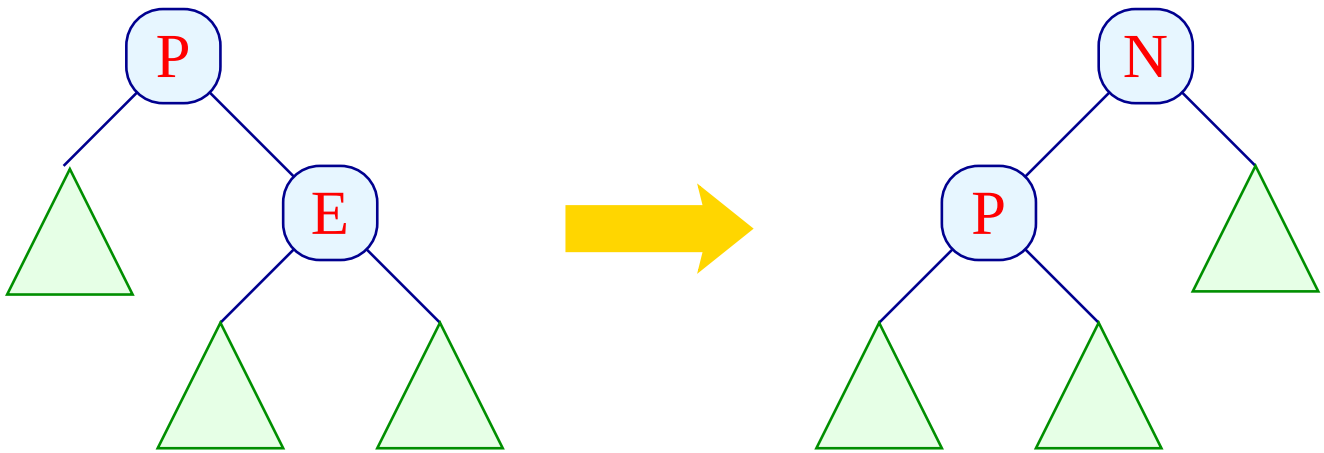
`rotateRight:`



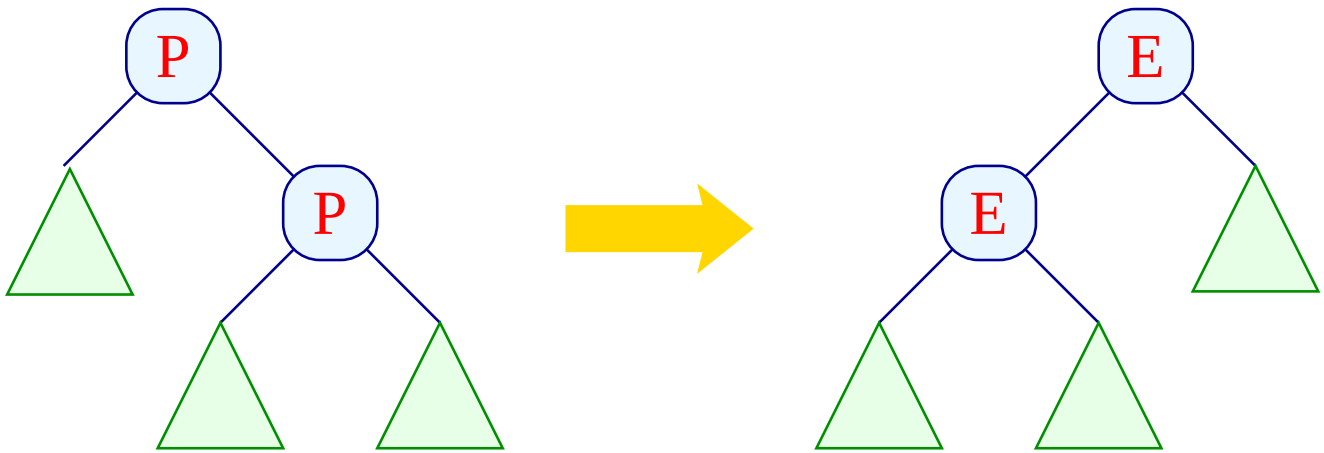
```
let rotateRight (left, y, right) = match left
    with Eq (l1,y1,r1) -> (Pos (l1, y1, Neg (r1,y,right)), false)
      |  Neg (l1,y1,r1) -> (Eq (l1, y1, Eq (r1,y,right)), true)
      |  Pos (l1, y1, Eq (l2,y2,r2)) ->
             (Eq (Eq (l1,y1,l2), y2,  Eq (r2,y,right)), true)
      |  Pos (l1, y1, Neg (l2,y2,r2)) ->
             (Eq (Eq (l1,y1,l2), y2, Pos (r2,y,right)), true)
      |  Pos (l1, y1, Pos (l2,y2,r2)) ->
             (Eq (Neg (l1,y1,l2), y2, Eq (r2,y,right)), true)
```

- Das zusätzliche Bit gibt diesmal an, ob der Baum nach der Rotation in der Tiefe abnimmt ...
- Das ist nur dann nicht der Fall, wenn der tiefere Teilbaum von der Form `Eq (...)` ist — was hier nie vorkommt :-)

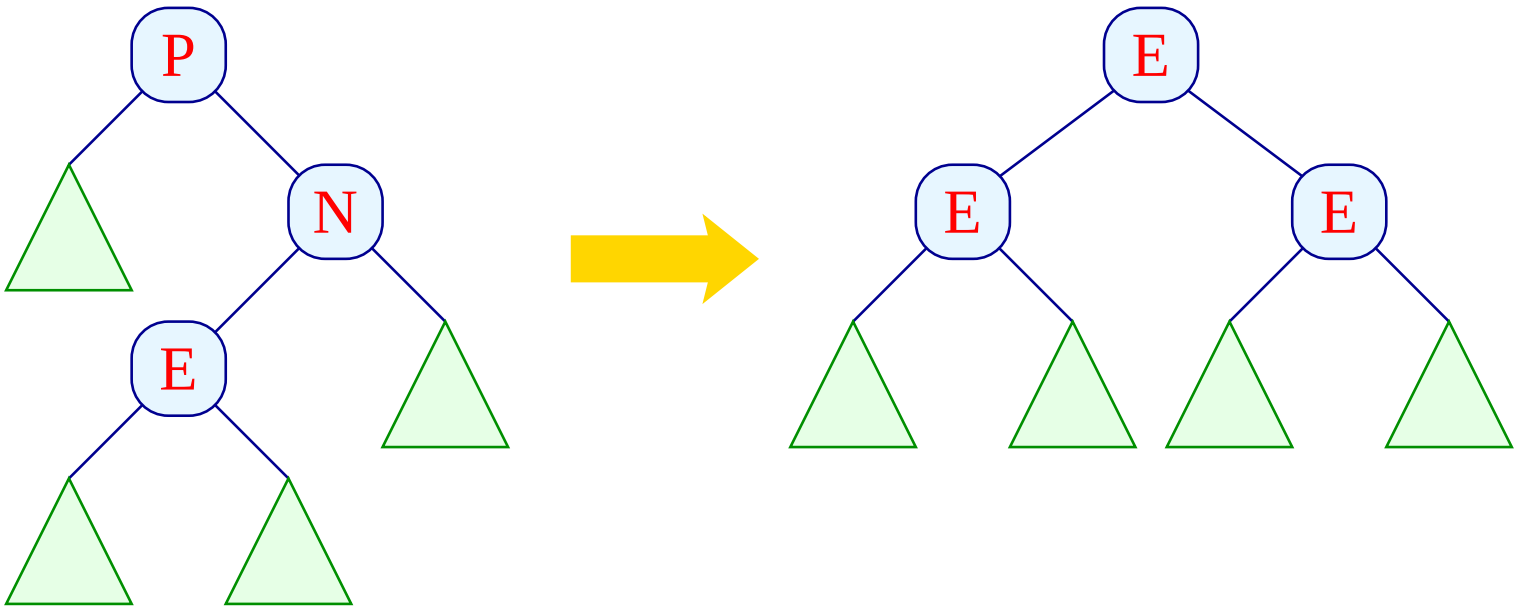
`rotateLeft:`



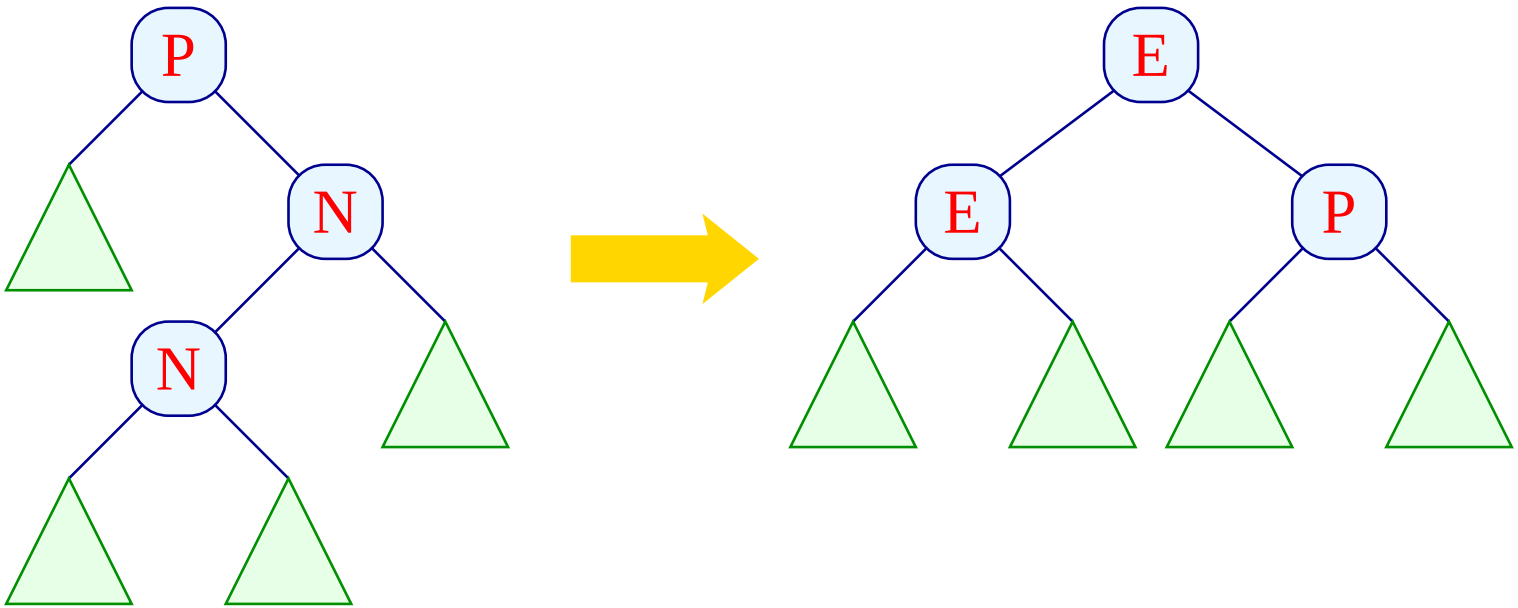
`rotateLeft:`



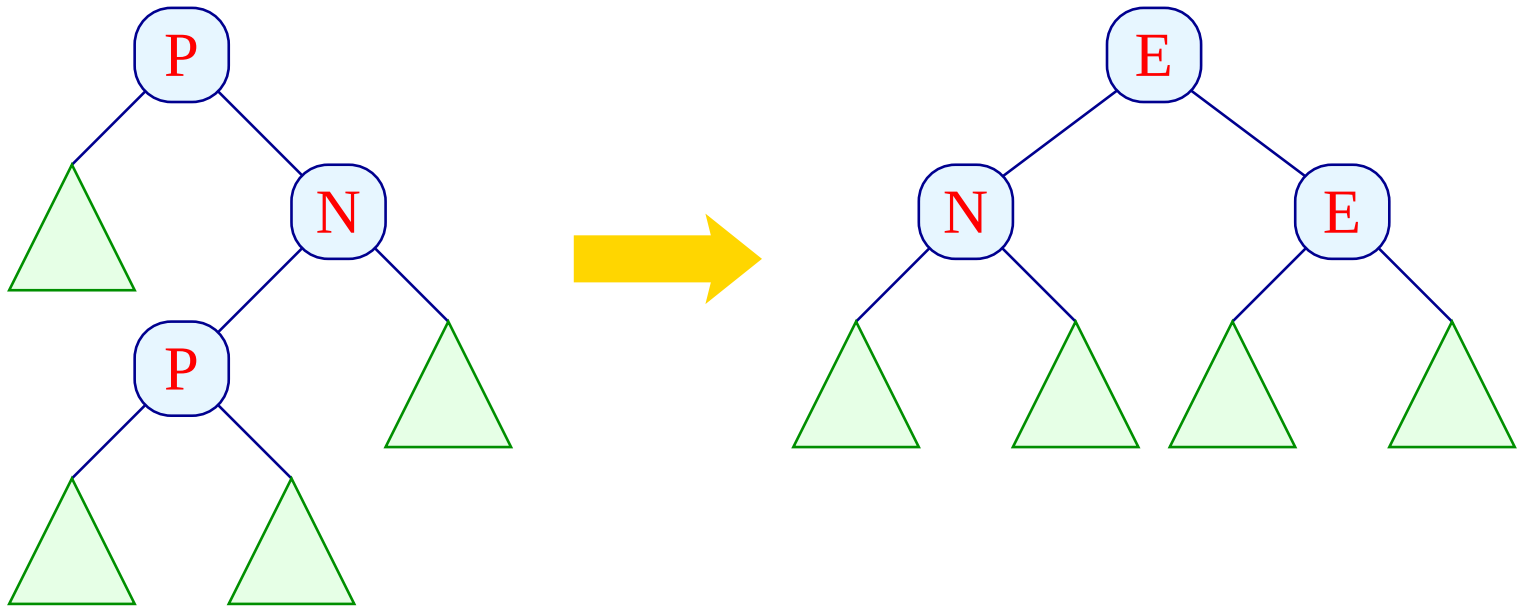
`rotateLeft:`



`rotateLeft:`



`rotateLeft:`



```
let rotateLeft (left, y, right) = match right
    with Eq (l1,y1,r1) -> (Neg (Pos (left,y,l1), y1, r1), false)
    |   Pos (l1,y1,r1) -> (Eq  (Eq  (left,y,l1), y1, r1), true)
    |   Neg (Eq (l1,y1,r1), y2 ,r2) ->
                (Eq (Eq (left,y,l1),y1,  Eq (r1,y2,r2)), true)
    |   Neg (Neg (l1,y1,r1), y2 ,r2) ->
                (Eq (Eq (left,y,l1),y1, Pos (r1,y2,r2)), true)
    |   Neg (Pos (l1,y1,r1), y2 ,r2) ->
                (Eq (Neg (left,y,l1),y1, Eq (r1,y2,r2)), true)
```

- `rotateLeft` ist analog zu `rotateRight` — nur mit den Rollen von `Pos` und `Neg` vertauscht.
- Wieder wird fast immer die Tiefe verringert :-)

Diskussion:

- Einfügen benötigt höchstens soviele Aufrufe von `insert` wie der Baum tief ist.
- Nach Rückkehr aus dem Aufruf für einen Teilbaum müssen maximal drei Knoten umorganisiert werden.
- Der Gesamtaufwand ist darum proportional zu $\log(n)$:-)
- Im allgemeinen sind wir aber nicht an dem Zusatz-Bit bei jedem Aufruf interessiert. Deshalb definieren wir:

```
let insert x tree = let (tree,_) = insert x tree
                    in   tree
```

Extraktion des Minimums:

- Das Minimum steht am linksten inneren Knoten.
- Dieses finden wir mithilfe eines rekursiven Besuchens des jeweils linken Teilbaums :-)

  Den linksten Knoten haben wir gefunden, wenn der linke Teilbaum `Null` ist :-))
- Entfernen eines Blatts könnte die Tiefe verringern und damit die AVL-Eigenschaft zerstören.
- Nach jedem Aufruf müssen wir darum den Baum lokal reparieren ...

```
let rec extract_min avl = match avl
    with Null                  -> (None, Null, false)
       | Eq (Null,y,right) -> (Some y, right, true)
       | Eq (left,y,right) -> let (first,left,dec) = extract_min left
                  in if dec then (first, Pos (left,y,right), false)
                     else        (first,  Eq (left,y,right), false)
       | Neg (left,y,right) -> let (first,left,dec) = extract_min left
                   in if dec then (first,  Eq (left,y,right), true)
                      else        (first, Neg (left,y,right), false)
       | Pos (Null,y,right) -> (Some y, right, true)
       | Pos (left,y,right) -> let (first,left,dec) = extract_min left
                in if dec then let (avl,b) = rotateLeft (left,y,right)
                               in (first,avl,b)
                   else        (first, Pos (left,y,right), false)
```

Diskussion:

- Rotierung ist nur erforderlich, wenn aus einem Baum der Form `Pos (...)` extrahiert wird und sich die Tiefe des linken Teilbaums verringert :-)
- Insgesamt ist die Anzahl der rekursiven Aufrufe beschränkt durch die Tiefe. Bei jedem Aufruf werden maximal drei Knoten umgeordnet.
- Der Gesamtaufwand ist darum proportional $\log(n)$:-)
- Analog konstruiert man Funktionen, die das Maximum bzw. das letzte Element aus einem Intervall extrahieren ...

5 Praktische Features in Ocaml

- Ausnahmen
- Ein- und Ausgabe als Seiteneffekte
- Sequenzen

5.1 Ausnahmen (Exceptions)

Bei einem Laufzeit-Fehler, z.B. Division durch Null, erzeugt das Ocaml-System eine exception (Ausnahme):

```
# 1 / 0;;
Exception: Division_by_zero.
# List.tl (List.tl [1]);;
Exception: Failure "tl".
# Char.chr 300;;
Exception: Invalid_argument "Char.chr".
```

Hier werden die Ausnahmen `Division_by_zero`, `Failure ¨tl¨` bzw. `Invalid_argument ¨Char.chr¨` erzeugt.

Ein anderer Grund für eine Ausnahme ist ein unvollständiger Match:

```
# match 1+1 with 0 -> "null";;
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
1
Exception: Match_failure ("", 2, -9).
```

In diesem Fall wird die Exception `Match_failure ("", 2, -9)` erzeugt :-)

Vordefinierte Konstruktoren für Exceptions:

`Division_by_zero`	Division durch Null
`Invalid_argument of string`	falsche Benutzung
`Failure of string`	allgemeiner Fehler
`Match_failure of string * int * int`	unvollständiger Match
`Not_found`	nicht gefunden :-)
`Out_of_memory`	Speicher voll
`End_of_file`	Datei zu Ende
`Exit`	für die Benutzerin ...

Eine Exception ist ein First Class Citizen, d.h. ein Wert eines Datentyps `exn` ...

```
# Division_by_zero;;
- : exn = Division_by_zero
# Failure "Kompletter Quatsch!";;
- : exn = Failure "Kompletter Quatsch!"
```

Eigene Exceptions werden definiert, indem der Datentyp `exn` erweitert wird ...

```
# exception Hell;;
exception Hell
# Hell;;
- : exn = Hell
```

```
# Division_by_zero;;
- : exn = Division_by_zero
# Failure "Kompletter Quatsch!";;
- : exn = Failure "Kompletter Quatsch!"
```

Eigene Exceptions werden definiert, indem der Datentyp `exn` erweitert wird ...

```
# exception Hell of string;;
exception Hell of string
# Hell "damn!";;
- : exn = Hell "damn!"
```

Ausnahmebehandlung:

Wie in Java können Exceptions ausgelöst und behandelt werden:

```
# let teile (n,m) = try Some (n / m)
      with Division_by_zero -> None;;

# teile (10,3);;
- : int option = Some 3
# teile (10,0);;
- : int option = None
```

So kann man z.B. die `member`-Funktion neu definieren:

```
let rec member x l = try if x = List.hd l then true
                         else member x (List.tl l)
    with Failure _ -> false

# member 2 [1;2;3];;
- : bool = true
# member 4 [1;2;3];;
- : bool = false
```

Das Schlüsselwort `with` leitet ein Pattern Matching auf dem Ausnahme-Datentyp `exn` ein:

```
try <exp>
with <pat1> -> <exp1> | ...  | <patN> -> <expN>
```

$\Longrightarrow$ Man kann mehrere Exceptions gleichzeitig abfangen :-)

Der Programmierer kann selbst Exceptions auslösen.
Das geht mit dem Schlüsselwort `raise` ...

```
# 1 + (2/0);;
Exception: Division_by_zero.
# 1 + raise Division_by_zero;;
Exception: Division_by_zero.
```

Eine Exception ist ein Fehlerwert, der jeden Ausdruck ersetzen kann.

Bei Behandlung wird sie durch einen anderen Ausdruck (vom richtigen Typ) ersetzt — oder durch eine andere Exception ;-)

Exception Handling kann nach jedem beliebigen Teilausdruck, auch geschachtelt, stattfinden:

```
# let f (x,y) = x / (y-1);;
# let g (x,y) = try let n = try f (x,y)
                            with Division_by_zero ->
                                 raise (Failure "Division by zero")
                       in string_of_int (n*n)
                  with Failure str -> "Error: "^str;;

# g (6,1);;
- : string = "Error: Division by zero"
# g (6,3);;
- : string = "9"
```

5.2 Textuelle Ein- und Ausgabe

- Lesen aus der Eingabe und Schreiben auf die Ausgabe sprengt den rein funktionalen Rahmen !
- Diese Operationen werden darum mit Hilfe von Seiteneffekten realisiert, d.h. mit Hilfe von Funktionen, deren Rückgabewert uninteressant ist (etwa `unit`).
- Während der Ausführung wird dann aber die entsprechende Aktion ausgeführt

  $\Longrightarrow$ nun kommt es genau auf die Reihenfolge der Auswertung an !!!

- Selbstverständlich kann man in Ocaml auf den Standard-Output schreiben:

```
# print_string "Hello World!\n";;
Hello World!
- : unit = ()
```

- Analog gibt es eine Funktion: `read_line : unit -> string`

...

```
# read_line ();;
Hello World!
- : "Hello World!"
```

Um aus einer Datei zu lesen, muss man diese zum Lesen öffnen ...

```
# let infile = open_in "test";;
val infile : in_channel = <abstr>
# input_line infile;;
- : "Die einzige Zeile der Datei ...";;
# input_line infile;;
Exception: End_of_file
```

Gibt es keine weitere Zeile, wird die Exception `End_of_file` geworfen :-)

Benötigt man einen Kanal nicht mehr, sollte man ihn geregelt schließen ...

```
# close_in infile;;
- : unit = ()
```

Weitere nützliche Funktionen:

```
stdin                : in_channel
input_char           : in_channel -> char
in_channel_length : in_channel -> int
input : in_channel -> string -> int -> int -> int
```

- `in_channel_length` liefert die Gesamtlänge der Datei.
- `input chan buf p n` liest aus einem Kanal `chan` `n` Zeichen und schreibt sie ab Position `p` in den String `buf` :-)

Die Ausgabe in Dateien erfolgt ganz analog ...

```
# let outfile = open_out "test";;
val outfile : out_channel = <abstr>
# output_string outfile "Hello ";;
- : unit = ()
# output_string outfile "World!\n";;
- : unit = ()
...
```

Die einzeln geschriebenen Wörter sind mit Sicherheit in der Datei erst zu finden, wenn der Kanal geregelt geschlossen wurde ...

```
# close_out outfile;;
- : unit = ()
```

5.3 Sequenzen

Bei Seiteneffekten kommt es auf die Reihenfolge an :-)

Mehrere solche Aktionen kann man mit dem Sequenz-Operator ; hintereinander ausführen:

```
# print_string "Hello";
  print_string " ";
  print_string "world!\n";;
Hello world!
- : unit = ()
```

Oft möchte man viele Strings ausgeben !

Hat man etwa eine Liste von Strings, hilft das Listenfunktional `List.iter`: weiter:

```
#  let rec iter f = function
       []    -> ()
     | x::[] -> f x
     | x::xs -> f x; iter f xs;;

val iter : ('a -> unit) -> 'a list -> unit = <fun>
```