. . . im Beispiel:

$$
\begin{array}{l}
\textbf{class}\ \mathsf{Eq}\ \textbf{where} \\
\quad (=) = \textbf{fn}\ \beta \Rightarrow\ \ \textbf{case}\ \beta\ \textbf{of} \\
\qquad\qquad\qquad\quad \mathsf{list}\,\alpha\ \ \rightarrow\ \ (=)_{\mathsf{list}}\ \alpha \\
\qquad\qquad\qquad\quad \mid \ldots \\
\textbf{inst}\ \beta \in \mathsf{Eq} \Rightarrow \textbf{list}\ \beta \in \mathsf{Eq} \\
\textbf{where}\ (=)_{\mathsf{list}} = \ \ \textbf{fn}\,\beta \Rightarrow \textbf{letrec}\ f = \textbf{fn}\,l_1\ \Rightarrow \textbf{fn}\,l_2 \Rightarrow \textbf{case}\ l_1\ \textbf{of} \\
\qquad\qquad\qquad\quad [] \rightarrow \textbf{case}\ l_2\ \textbf{of}\ [] \rightarrow \textbf{true} \mid \_ \rightarrow \textbf{false} \\
\qquad\qquad\qquad \mid\ x : xs \rightarrow\ \ \textbf{case}\ l_2\ \textbf{of}\ [] \rightarrow \textbf{false} \\
\qquad\qquad\qquad\qquad\qquad\quad \mid y : ys \rightarrow \textbf{if}\ (=)\ \beta\ x\ y\ \textbf{then}\ f\ xs\ ys\ \textbf{else false} \\
\qquad\qquad\qquad \textbf{in}\ \ f
\end{array}
$$

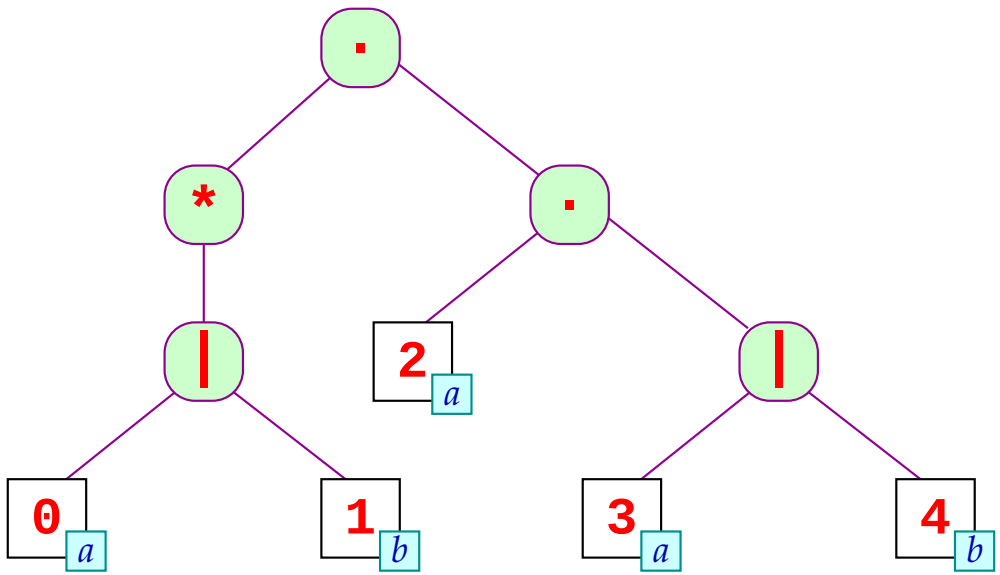
Schlussbemerkung

- Haskell bietet neben Typ-Klassen auch noch Typ-Konstruktor-Klassen.
- Diese sind entscheidend zur generischen Behandlung von Monaden.
- Mit Monaden lassen sich rein funktional theoretisch sauber Ein- und Ausgabe sowie Seiteneffekte modellieren.
- Der formale Aufwand ist jedoch enorm ...
- ... und disqualifiziert Haskell damit als Programmiersprache für Anfänger :-(

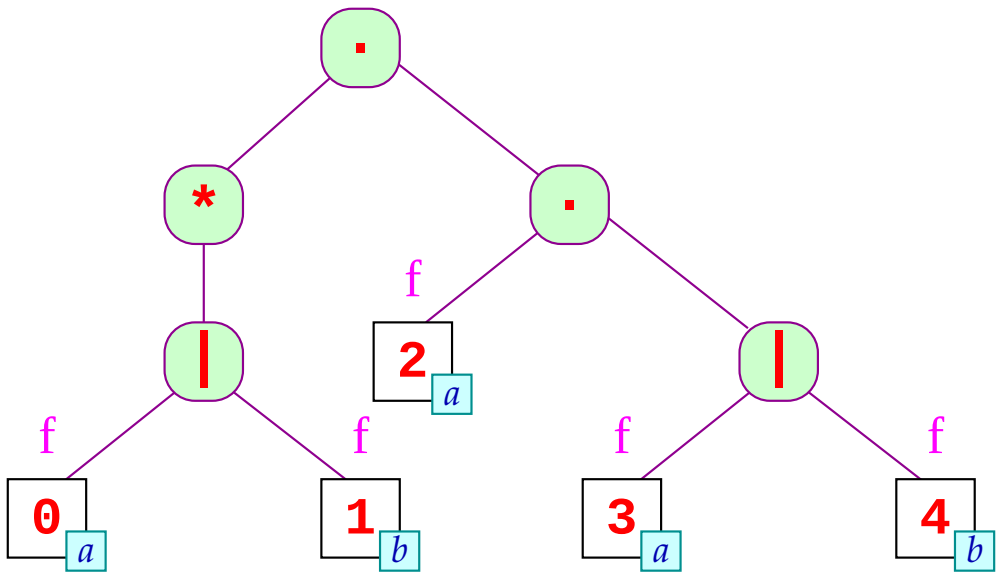
3.4 Attributierte Grammatiken

- Viele Berechnugen der semantischen Analyse wie während der Code-Generierung arbeiten auf dem Syntaxbaum.
- An jedem Knoten greifen sie auf bereits berechnete Informationen zu und berechnen daraus neue Informationen :-)
- Was lokal zu tun ist, hängt nur von der Sorte des Knotens ab !!!
- Damit die zu lesenden Werte an jedem Knoten bei jedem Lesen bereits vorliegen, müssen die Knoten des Syntaxbaums in einer bestimmten Reihenfolge durchlaufen werden ...

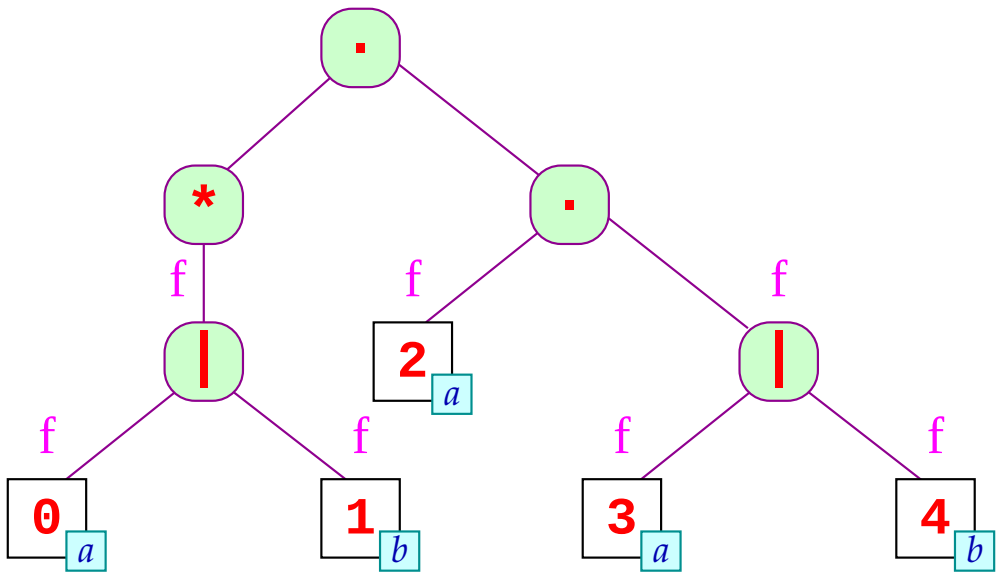
Beispiel: Berechnung des Prädikats empty$[r]$



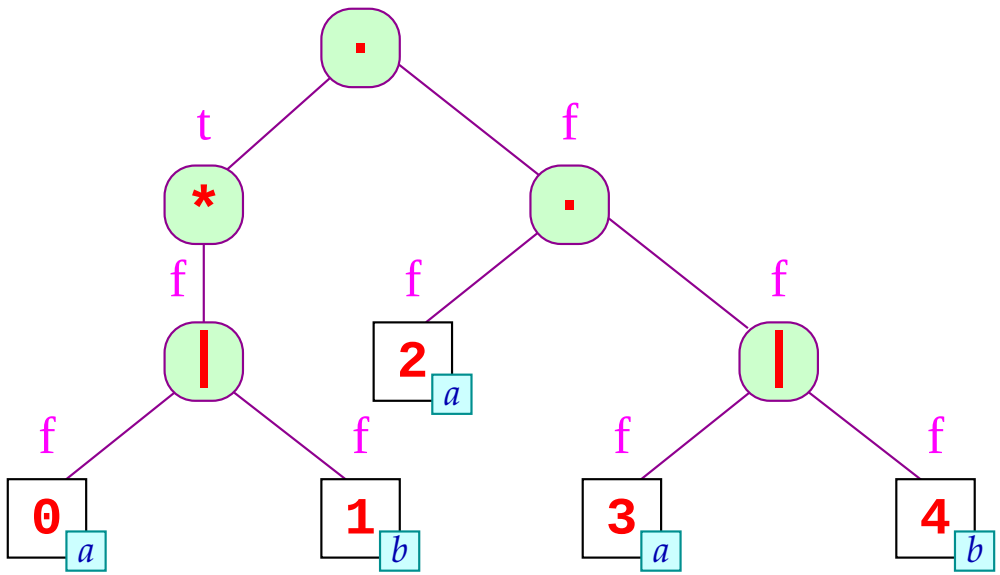
Beispiel: Berechnung des Prädikats empty$[r]$



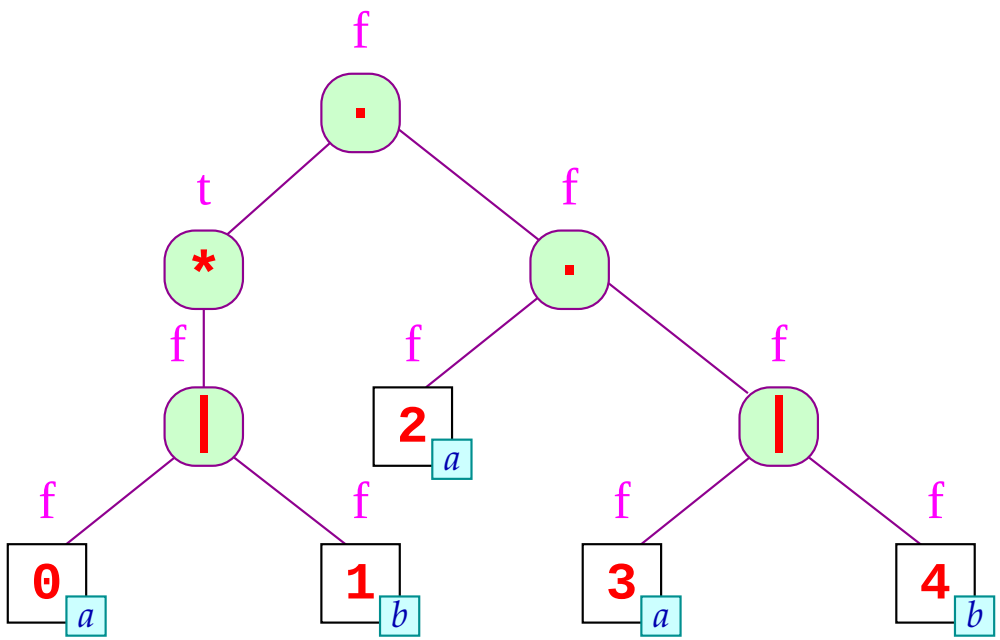
Beispiel: Berechnung des Prädikats empty$[r]$



Beispiel: Berechnung des Prädikats empty$[r]$



Beispiel: Berechnung des Prädikats empty$[r]$



Idee zur Implementierung:

- Für jeden Knoten führen wir ein Attribut empty ein.
- Die Attribute werden in einer DFS post-order Traversierung berechnet:
  - → An einem Blatt lässt sich der Wert des Attributs unmittelbar ermitteln ;-)
  - → Das Attribut an einem inneren Knoten hängt darum nur von den Attributen der Nachfolger ab :-)
- Wie das Attribut lokal zu berechnen ist, ergibt sich aus dem Typ des Knotens ...

Für Blätter $r \equiv \boxed{i \mid x}$ ist $\mathsf{empty}[r] = (x \equiv \epsilon)$.

Andernfalls:

$$
\begin{aligned}
\mathsf{empty}[r_1 \mid r_2] &= \mathsf{empty}[r_1] \vee \mathsf{empty}[r_2] \\
\mathsf{empty}[r_1 \cdot r_2] &= \mathsf{empty}[r_1] \wedge \mathsf{empty}[r_2] \\
\mathsf{empty}[r_1^*] &= t \\
\mathsf{empty}[r_1?] &= t
\end{aligned}
$$

Diskussion:

- Wir benötigen einen einfachen und flexiblen Mechanismus, mit dem wir über die Attribute an einem Knoten und seinen Nachfolgern reden können.
- Der Einfachkeit geben wir ihnen einen fortlaufenden Index:

  $\mathsf{empty}[0]$: das Attribut des aktuellen Knotens

  $\mathsf{empty}[i]$: das Attribut des i-ten Sohns $(i > 0)$

Diskussion:

- Wir benötigen einen einfachen und flexiblen Mechanismus, mit dem wir über die Attribute an einem Knoten und seinen Nachfolgern reden können.
- Der Einfachkeit geben wir ihnen einen fortlaufenden Index:

  $\mathsf{empty}[0]$: das Attribut des aktuellen Knotens

  $\mathsf{empty}[i]$: das Attribut des i-ten Sohns $(i > 0)$

... im Beispiel:

$$
\begin{array}{lcll}
\boxed{x} & : & \mathsf{empty}[0] & := (x \equiv \epsilon) \\
\boxed{|} & : & \mathsf{empty}[0] & := \mathsf{empty}[1] \vee \mathsf{empty}[2] \\
\boxed{\cdot} & : & \mathsf{empty}[0] & := \mathsf{empty}[1] \wedge \mathsf{empty}[2] \\
\boxed{*} & : & \mathsf{empty}[0] & := t \\
\boxed{?} & : & \mathsf{empty}[0] & := t
\end{array}
$$

Diskussion:

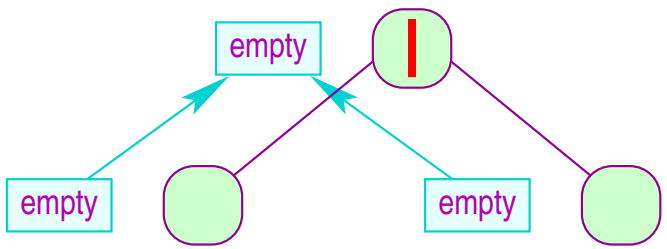
- Die lokalen Berechnungen der Attributwerte müssen zu einem globalen Algorithmus zusammen gesetzt werden :-)
- Dazu benötigen wir:

  (1) eine Besuchsreihenfolge der Knoten des Baums;

  (2) lokale Berechnungsreihenfolgen ...

- Die Auswertungsstrategie sollte aber mit den Attribut-Abhängigkeiten kompatibel sein :-)

... im Beispiel:



Achtung:

- Zur Ermittlung einer Auswertungsstrategie reicht es nicht, sich die lokalen Attribut-Abhängigkeiten anzusehen.
- Es kommt auch darauf an, wie sie sich global zu einem Abhängigkeitsgraphen zusammen setzen !!!
- Im Beispiel sind die Abhängigkeiten stets von den Attributen der Söhne zu den Attributen des Vaters gerichtet.

  $\Longrightarrow$ Postorder-DFS-Traversierung
- Die Variablen-Abhängigkeiten können aber auch komplizierter sein ...

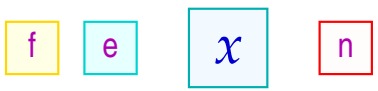
Beispiel: Simultane Berechnung von empty, first, next :

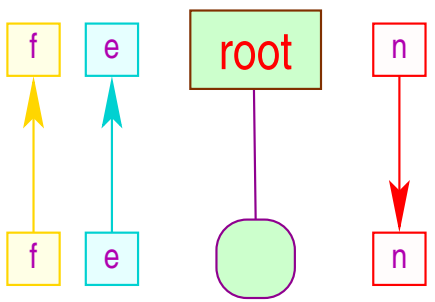
$\boxed{x}$:

$$\begin{aligned} \mathsf{empty}[0] &:= (x \equiv \epsilon) \\ \mathsf{first}[0] &:= \{x \mid x \neq \epsilon\} \\ & \quad // \quad \text{(keine Gleichung für next !!!)} \end{aligned}$$

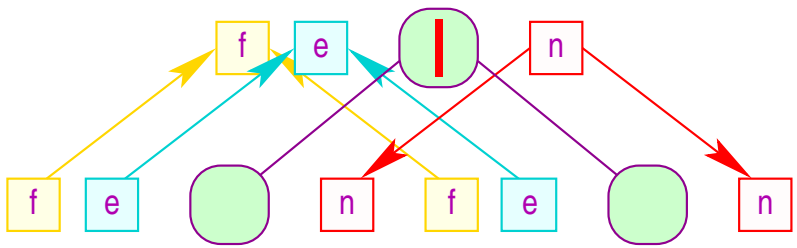
$\boxed{\text{root:}}$:

$$\begin{aligned} \mathsf{empty}[0] &:= \mathsf{empty}[1] \\ \mathsf{first}[0] &:= \mathsf{first}[1] \\ \mathsf{next}[0] &:= \emptyset \\ \mathsf{next}[1] &:= \mathsf{next}[0] \end{aligned}$$

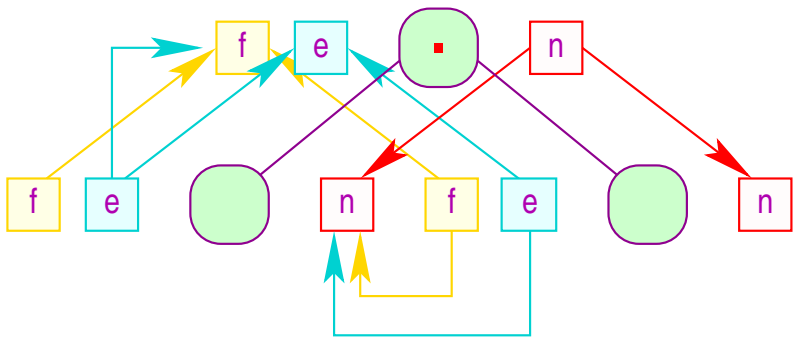




$$
\begin{array}{lcll}
\boxed{|} : & \mathsf{empty}[0] & := & \mathsf{empty}[1] \vee \mathsf{empty}[2] \\
 & \mathsf{first}[0] & := & \mathsf{first}[1] \cup \mathsf{first}[2] \\
 & \mathsf{next}[1] & := & \mathsf{next}[0] \\
 & \mathsf{next}[2] & := & \mathsf{next}[0]
\end{array}
$$



$$
\begin{array}{lcll}
\boxed{\cdot} & : & \mathsf{empty}[0] & := & \mathsf{empty}[1] \wedge \mathsf{empty}[2] \\
 & & \mathsf{first}[0] & := & \mathsf{first}[1] \cup (\mathsf{empty}[1])\,?\,\mathsf{first}[2] : \emptyset \\
 & & \mathsf{next}[1] & := & \mathsf{first}[2] \cup (\mathsf{empty}[2])\,?\,\mathsf{next}[0] \\
 & & \mathsf{next}[2] & := & \mathsf{next}[0]
\end{array}
$$



$$
\begin{array}{lcll}
\boxed{*} & : & \mathsf{empty}[0] & := & t \\
 & & \mathsf{first}[0] & := & \mathsf{first}[1] \\
 & & \mathsf{next}[1] & := & \mathsf{first}[1] \cup \mathsf{next}[0] \\
 & & & & \\
\boxed{?} & : & \mathsf{empty}[0] & := & t \\
 & & \mathsf{first}[0] & := & \mathsf{first}[1] \\
 & & \mathsf{next}[1] & := & \mathsf{next}[0]
\end{array}
$$

Problem:

- Eine Auswertungsstrategie kann es nur dann geben, wenn die Variablen-Abhängigkeiten in jedem attributierten Baum azyklisch sind !!!
- Es ist DEXPTIME-vollständig, herauszufinden, ob keine zyklischen Variablenabhängigkeiten vorkommen können :-(

Problem:

- Eine Auswertungsstrategie kann es nur dann geben, wenn die Variablen-Abhängigkeiten in jedem attributierten Baum azyklisch sind !!!
- Es ist DEXPTIME-vollständig, herauszufinden, ob keine zyklischen Variablenabhängigkeiten vorkommen können :-(

Ideen:

(1) Die Benutzerin soll die Strategie spezifizieren ;-)

(2) Bestimme die Strategie dynamisch ;-}

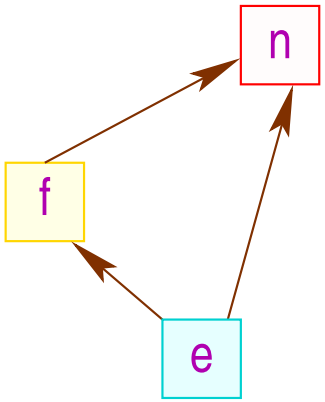
(3) Betrachte Teilklassen ...

Stark azyklische Attributierung:

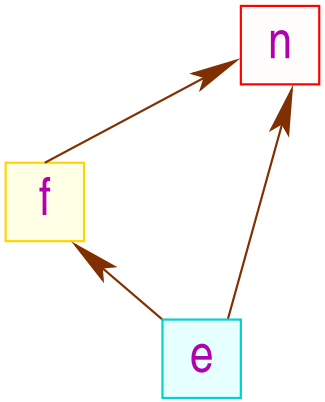
Berechne eine partielle Ordnung auf den Attributen eines Knotens, die kompatibel mit den lokalen Attribut-Abhängigkeiten ist:

- Wir starten mit der trivialen Ordung $\sqsubseteq = =$:-)
- Die aktuelle Ordnung setzen wir an den Sohn-Knoten in die lokalen Abhängigkeitsgraphen ein.
- Ergibt sich ein Kreis, geben wir auf :-))
- Andernfalls fügen wir alle Beziehungen $a \sqsubseteq b$ hinzu, für die es jetzt einen Pfad von $a[0]$ nach $b[0]$ gibt.
- Lässt sich $\sqsubseteq$ nicht mehr vergrößern, hören wir auf ...

... im Beispiel:



... im Beispiel:



Diskussion:

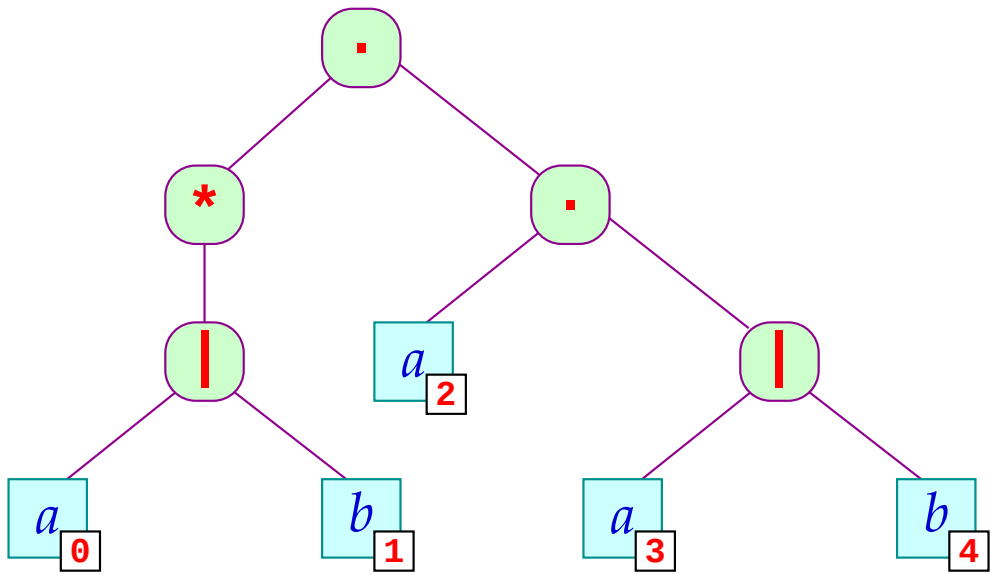
- Die Berechnung der partiellen Ordnung $\sqsubseteq$ ist eine Fixpunkt-Berechnung :-)
- Die partielle Ordnung können wir in eine lineare Ordnung einbetten ...
- Die lineare Ordnung gibt uns an, in welcher Reihenfolge die Attribute berechnet werden müssen :-)
- Die lokalen Abhängigkeitsgraphen zusammen mit der linearen Ordnung erlauben die Berechnung einer Strategie ...

Mögliche Strategien:

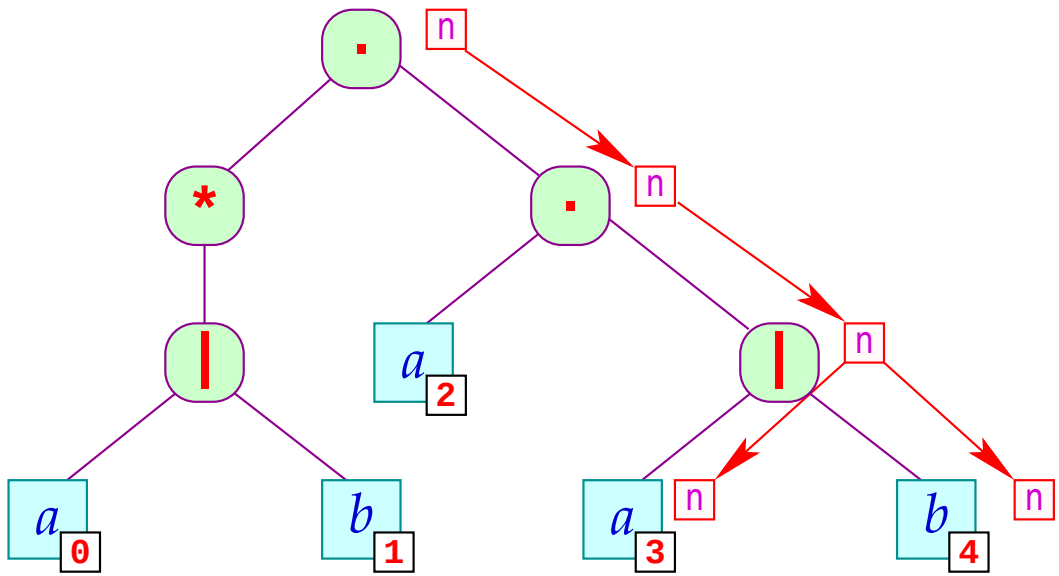
(1) Bedarfsgetriebene Auswertung:

- Beginne mit der Berechnung eines Attributs.
- Sind die Argument-Attribute noch nicht berechnet, berechne rekursiv deren Werte :-)
- Besuche die Knoten des Baum nach Bedarf...

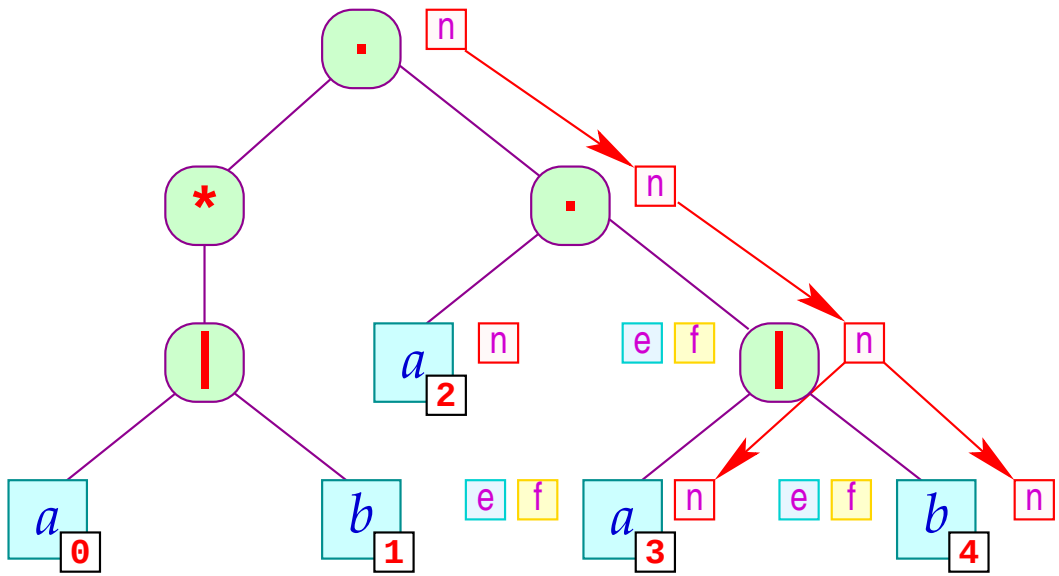
Beispiel, bedarfsgetrieben:



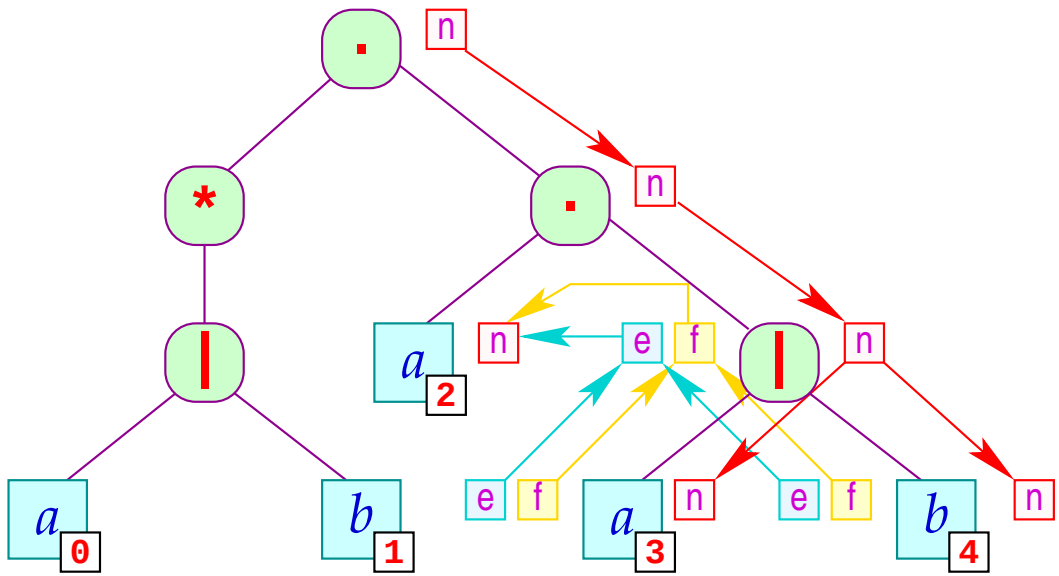
Beispiel, bedarfsgetrieben:



Beispiel, bedarfsgetrieben:



Beispiel, bedarfsgetrieben:



Diskussion:

- Die Reihenfolge hängt i.a. vom zu attributierenden Baum ab.
- Der Algorithmus muss sich merken, welche Attribute er bereits berechnete :-(
- Der Algorithmus besucht manche Knoten unnötig oft.
- Der Algorithmus ist nicht-lokal :-((

Mögliche Strategien (Forts.):

(2) Auswertung in Pässen:

- Minimiere die Anzahl der Besuche an jedem Knoten.
- Organisiere die Auswertung in Durchläufe durch den Baum.
- Berechne für jeden Pass eine Besuchsstrategie für die Knoten zusammen mit einer lokalen Strategie für jeden Knoten-Typ ...

Achtung:

- Das minimale Attribut in der Anordnung für stark azyklische Attributierungen lässt sich stets in einem Pass berechnen :-)
- Man braucht folglich für stark azyklische Attributierungen maximal so viele Pässe, wie es Attribute gibt :-))
- Hat man einen Baum-Durchlauf zur Berechnung einiger Attribute, kann man überprüfen, ob er geeignet ist, gleichzeitig weitere Attribute auszuwerten $\Longrightarrow$ Optimierungsproblem

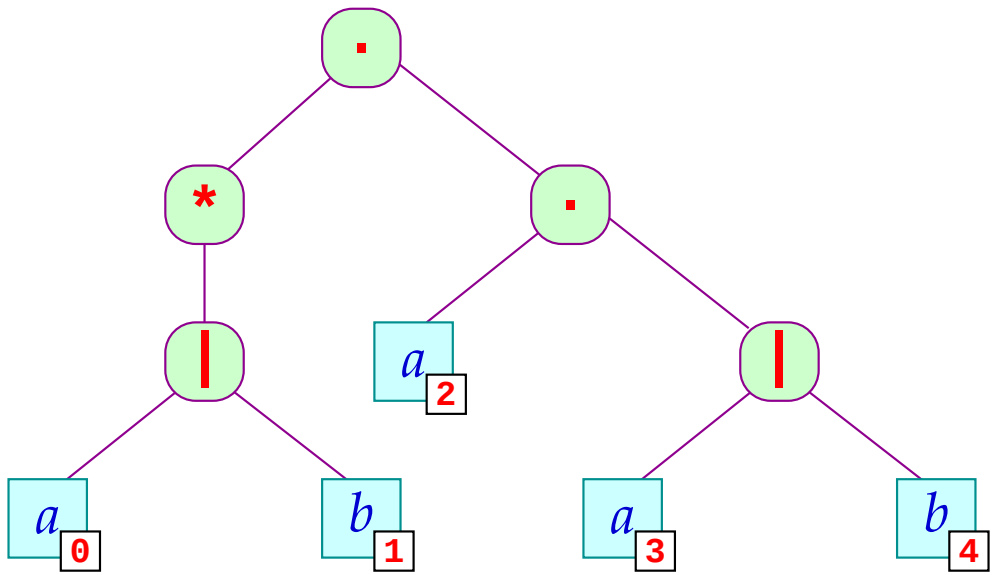
... im Beispiel:

empty und first lassen sich gemeinsam berechnen.

next muss in einem weiteren Pass berechnet werden :-(

Weiteres Beispiel:

Nummerierung der Blätter eines Baums:

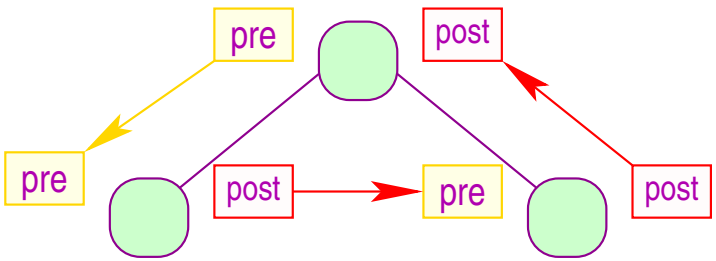


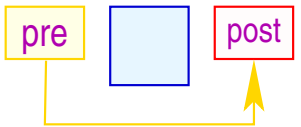
Idee:

- Führe Hilfsattribute pre und post ein !
- Mit pre reichen wir einen Zählerstand nach unten
- Mit post reichen wir einen Zählerstand wieder nach oben ...

Root:	$\mathsf{pre}[0]$	$:=$	0
	$\mathsf{pre}[1]$	$:=$	$\mathsf{pre}[0]$
	$\mathsf{post}[0]$	$:=$	$\mathsf{post}[1]$
Node:	$\mathsf{pre}[1]$	$:=$	$\mathsf{pre}[0]$
	$\mathsf{pre}[2]$	$:=$	$\mathsf{post}[1]$
	$\mathsf{post}[0]$	$:=$	$\mathsf{post}[2]$
Leaf:	$\mathsf{post}[0]$	$:=$	$\mathsf{pre}[0] + 1$

... die lokalen Attribut-Abhängigkeiten:





- Die Attributierung ist offenbar stark azyklisch :-)
- Man kann alle Attribute in einem Links-Rechts-Durchlauf auswerten :-))
- So etwas nennen wir L-Attributierung.
- L-Attributierung liegt auch unseren Query-Tools zur Suche in XML-Dokumenten zugrunde $\Longrightarrow$ fxgrep

Praktische Erweiterungen:

- Symboltabellen, Typ-Überprüfung / Inferenz und (einfache) Codegenerierung können durch Attributierung berechnet werden :-)
- In diesen Anwendungen werden stets Syntaxbäume annotiert.
- Die Knoten-Beschriftungen entsprechen den Regeln einer kontextfreien Grammatik :-)
- Knotenbeschriftungen können in Sorten eingeteilt werden — entsprechend den Nichtterminalen auf der linken Seite ...
- Unterschiedliche Nichtterminale benötigen evt. unterschiedliche Mengen von Attributen.
- Eine attributierte Grammatik ist eine CFG erweitert um:

  → Attribute für jedes Nichtterminal;

  → lokale Attribut-Gleichungen.

- Damit können die syntaktische, Teile der semantischen Analyse wie der Codeerzeugung generiert werden :-)