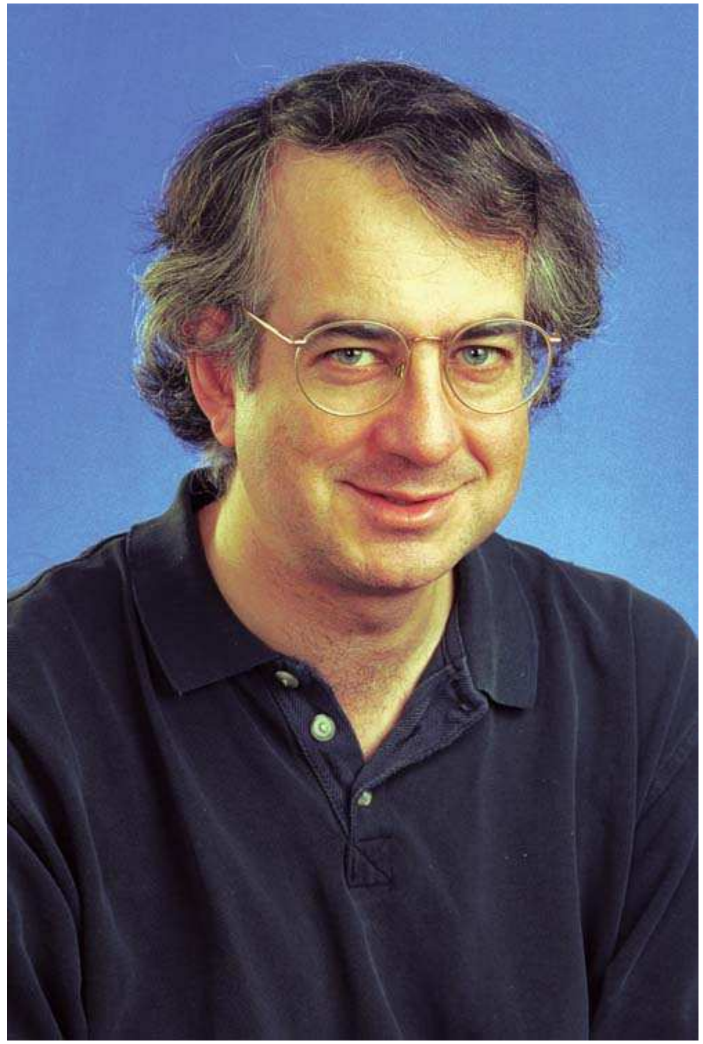


Harry Mairson, Brandeis University

Seiteneffekte

- Für ein elegantes Programmieren sind gelegentlich Variablen, deren Wert geändert werden kann, ganz nützlich :-)
- Darum erweitern wir unsere kleine Programmiersprache um Referenzen:

$$e \;::=\; \ldots \;\mid\; \textbf{ref}\; e \;\mid\; !e \;\mid\; e_1 := e_2$$

Seiteneffekte

- Für ein elegantes Programmieren sind gelegentlich Variablen, deren Wert geändert werden kann, ganz nützlich :-)
- Darum erweitern wir unsere kleine Programmiersprache um Referenzen:

$$e \;::=\; \ldots \;\mid\; \textbf{ref}\; e \;\mid\; !e \;\mid\; e_1 := e_2$$

Beispiel:

$$\begin{array}{lll}
\textbf{let}\;\; \mathsf{count} & = & \textbf{ref}\; 0; \\
\quad\;\; \mathsf{new} & = & \textbf{fn}\; () \;\Rightarrow\; \textbf{let} \\
 & & \qquad\qquad \begin{array}{lll} \mathsf{ret} & = & !\mathsf{count}; \\ _ & = & \mathsf{count} := \mathsf{ret} + 1 \end{array} \\
 & & \qquad\quad \textbf{in}\;\; \mathsf{ret} \\
\textbf{in}\;\; \mathsf{new}() + \mathsf{new}() & &
\end{array}$$

Als neuen Typ benötigen wir:

$$t \ ::= \ \ldots \ \mathbf{ref}\ t \ \ldots$$

Neue Regeln:

Ref: $$\frac{\Gamma \ \vdash \ e \ : \ t}{\Gamma \ \vdash \ (\mathbf{ref}\ e) \ : \ \mathbf{ref}\ t}$$

Deref: $$\frac{\Gamma \ \vdash \ e \ : \ \mathbf{ref}\ t}{\Gamma \ \vdash \ (!\, e) \ : \ t}$$

Assign: $$\frac{\Gamma \ \vdash \ e_1 \ : \ \mathbf{ref}\ t \qquad \Gamma \ \vdash \ e_2 \ : \ t}{\Gamma \ \vdash \ (e_1 := e_2) \ : \ ()}$$

Achtung:

Diese Regeln vertragen sich nicht mit Polymorphie !!!

Beispiel:

$$\begin{array}{lcll} \textbf{let } y & = & \textbf{ref}\,[\,]; & \\ \quad \_ & = & y & := \; 1 : (!\,y); \\ \quad \_ & = & y & := \; \textbf{true} : (!\,y) \\ \textbf{in } 1 & & & \end{array}$$

Achtung:

Diese Regeln vertragen sich nicht mit Polymorphie !!!

Beispiel:

$$\begin{array}{lllll} \textbf{let } y & = & \textbf{ref}\,[\,]; & & \\ \quad\_ & = & y & := & 1 : (!\,y); \\ \quad\_ & = & y & := & \textbf{true} : (!\,y) \\ \textbf{in } 1 & & & & \end{array}$$

Für y erhalten wir den Typ: $\forall\,\alpha.\,\textbf{ref}\,(\mathsf{list}\ \alpha)$

$\Longrightarrow$ Die Typ-Inferenz liefert keinen Fehler

$\Longrightarrow$ Zur Laufzeit entsteht eine Liste mit **int** und **bool** :-(

Ausweg: Die Value-Restriction

- Generalisiere nur solche Typen, die Werte repräsentieren, d.h. keine Verweise auf Speicherstellen enthalten :-)
- Die Menge der Value-Typen lässt sich einfach beschreiben:

$$v \;::=\; \mathbf{bool} \mid \mathbf{int} \mid \mathsf{list}\; v \mid (v_1, \ldots, v_m) \mid t \to t$$

Ausweg: Die Value-Restriction

- Generalisiere nur solche Typen, die Werte repräsentieren, d.h. keine Verweise auf Speicherstellen enthalten :-)
- Die Menge der Value-Typen lässt sich einfach beschreiben:

$$v \ ::= \ \mathbf{bool} \ | \ \mathbf{int} \ | \ \mathsf{list}\ v \ | \ (v_1, \ldots, v_m) \ | \ t \to t$$

... im Beispiel:

Der Typ: $\mathbf{ref}\ (\mathsf{list}\ \alpha)$ ist kein Value-Typ.

Darum darf er nicht generalisiert werden $\Longrightarrow$ Problem gelöst :-)

Matthias Felleisen, Northeastern University

Schlussbemerkung:

- Polymorphie ist ein sehr nützliches Hilfsmittel bei der Programmierung :-)
- In Form von Templates hält es in Java 1.5 Einzug.
- In der Programmiersprache Haskell hat man Polymorphie in Richtung bedingter Polymorphie weiter entwickelt ...

Schlussbemerkung:

- Polymorphie ist ein sehr nützliches Hilfsmittel bei der Programmierung :-)
- In Form von Templates hält es in Java 1.5 Einzug.
- In der Programmiersprache Haskell hat man Polymorphie in Richtung bedingter Polymorphie weiter entwickelt ...

Beispiel:

$$
\begin{array}{llll}
\textbf{fun}\ \ \textsf{member}\ x\ \textsf{list} & = & \textbf{case}\ \ \textsf{list} & \\
& \textbf{of} & [\,] & \rightarrow\ \ \textbf{false} \\
& \mid & h::t & \rightarrow\ \ \textbf{if}\ \ x = h\ \ \textbf{then}\ \ \textbf{true} \\
& & & \ \ \textbf{else}\ \ \textsf{member}\ x\ t
\end{array}
$$

Schlussbemerkung:

- Polymorphie ist ein sehr nützliches Hilfsmittel bei der Programmierung :-)
- In Form von Templates hält es in Java 1.5 Einzug.
- In der Programmiersprache Haskell hat man Polymorphie in Richtung bedingter Polymorphie weiter entwickelt ...

Beispiel:

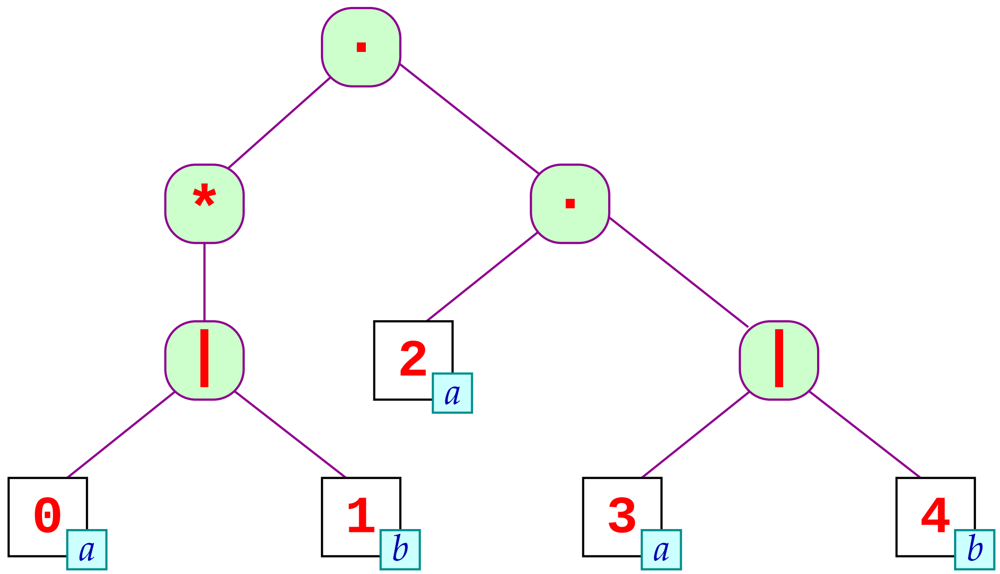
$$
\begin{array}{lllll}
\textbf{fun}\ \ \mathsf{member}\ x\ \mathsf{list} & = & \textbf{case}\ \ \mathsf{list} & & \\
 & & \textbf{of}\ \ [\,] & \to & \textbf{false} \\
 & & \mid\ \ h{::}t & \to & \textbf{if}\ \ x = h\ \ \textbf{then}\ \ \textbf{true} \\
 & & & & \textbf{else}\ \ \mathsf{member}\ x\ t
\end{array}
$$

member hat den Typ: $\alpha' \to \mathsf{list}\ \alpha' \to \textbf{bool}$ für jedes α' mit Gleichheit !!

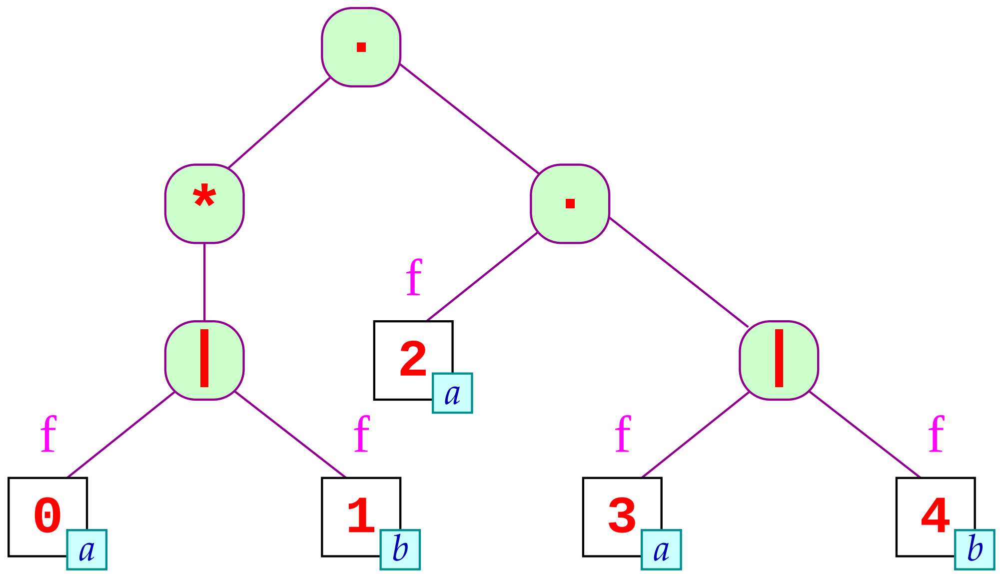
3.4 Attributierte Grammatiken

- Viele Berechnungen 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 ...

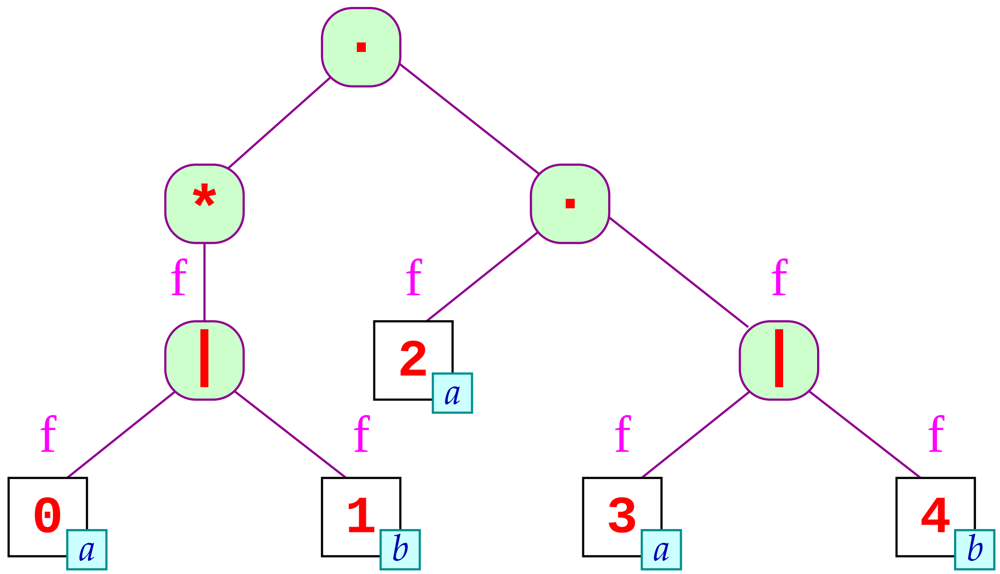
Beispiel: Berechnung des Prädikats $\text{empty}[r]$



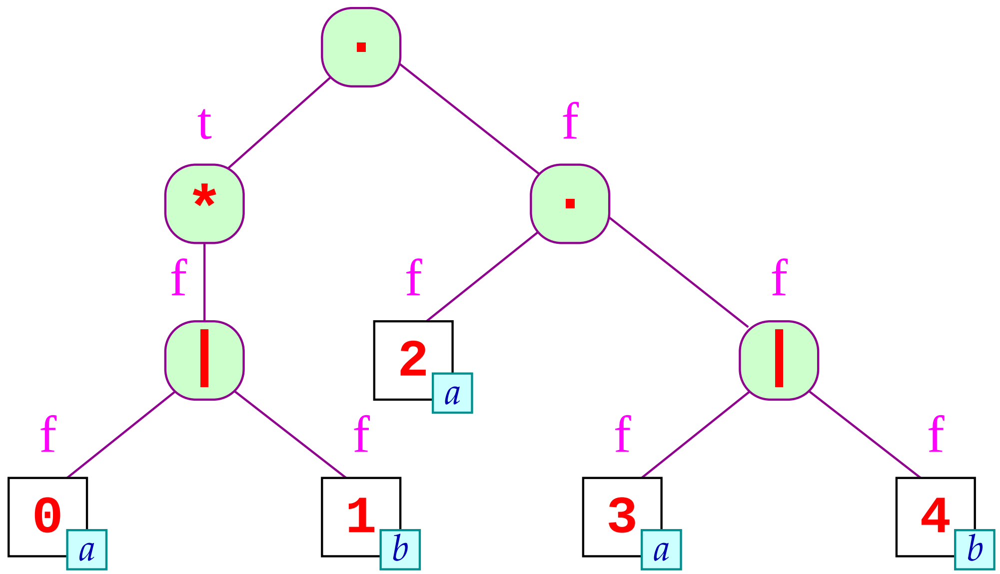
Beispiel: Berechnung des Prädikats empty$[r]$



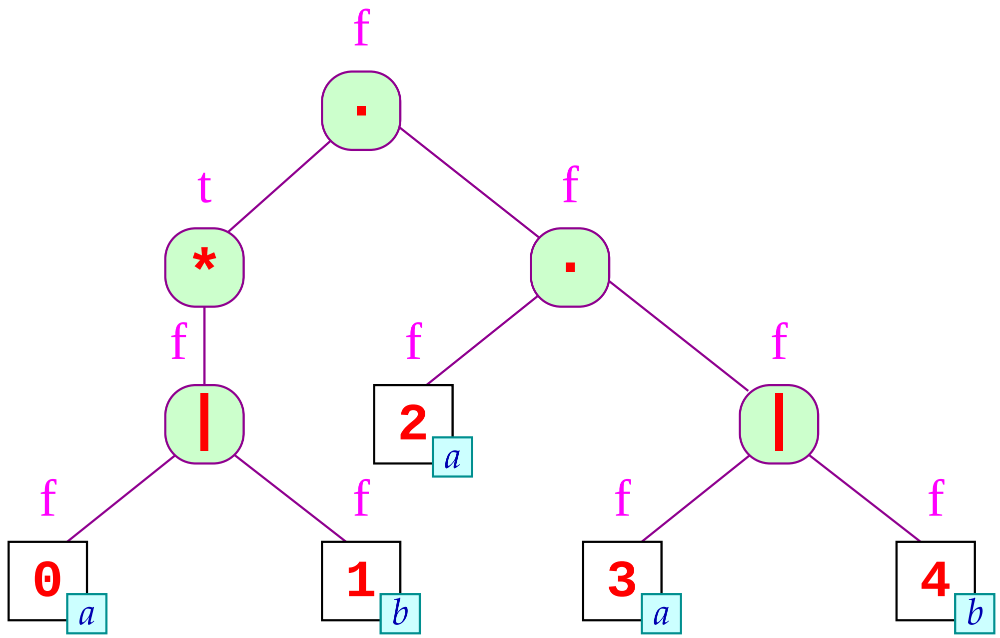
Beispiel: Berechnung des Prädikats $\mathsf{empty}[r]$



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 $\text{empty}[r] = (x \equiv \epsilon)$.

Andernfalls:

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

Diskussion:

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

empty$[0]$:	das Attribut des Vater-Knotens
empty$[i]$:	das Attribut des i-ten Sohns $\quad (i > 0)$

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:

$\text{empty}[0]$:	das Attribut des Vater-Knotens
$\text{empty}[i]$:	das Attribut des i-ten Sohns $\quad (i > 0)$

... im Beispiel:

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

Diskussion:

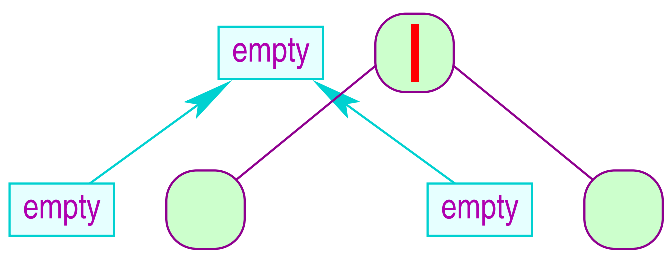
- Die lokalen Berechnungen der Attributwerte müssen zu einem globalen Algorithmus zusammen gesetzt werden :-)
- Dazu benötigen wir:

  (1) eine Besuchsreihenfolge der Knoten des Baums;

  (2) lokale Berechnungsreihenfolgen ...

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

... im Beispiel:



Achtung:

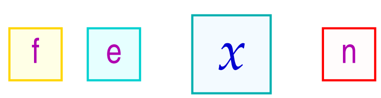
- Zur Ermittlung einer Auswertungsstrategie reicht es nicht, sich die lokalen Attribut-Abhängigkeiten anzusehen.
- Es kommt auch darauf an, wie sie sich global zu einem Abhängigkeitsgraphen zusammen setzen !!!
- Im Beispiel sind die Abhängigkeiten stets von den Attributen der Söhne zu den Attributen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

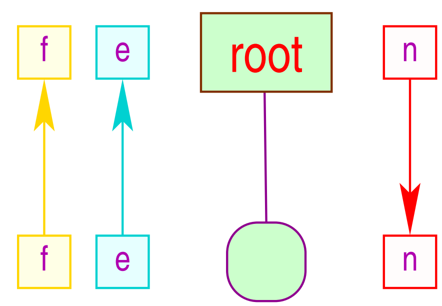
  $\Longrightarrow$ Postorder-DFS-Traversierung
- Die Variablen-Abhängigkeiten können aber auch komplizierter sein ...

Beispiel: Simultane Berechnung von empty, first, next :

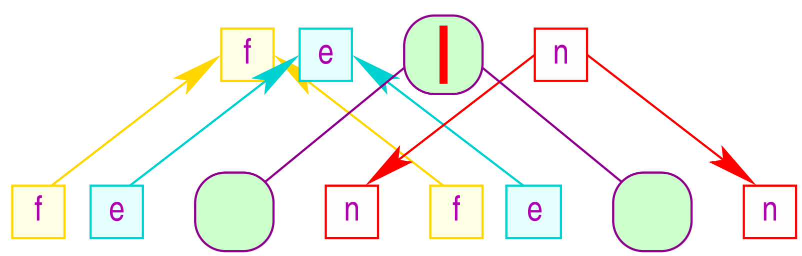
$$\boxed{x} \quad : \quad \begin{aligned} \text{empty}[0] &:= (x \equiv \epsilon) \\ \text{first}[0] &:= \{x \mid x \neq \epsilon\} \\ & // \quad \text{(keine Gleichung für next !!!)} \end{aligned}$$

$$\boxed{\text{root:}} \quad : \quad \begin{aligned} \text{empty}[0] &:= \text{empty}[1] \\ \text{first}[0] &:= \text{first}[1] \\ \text{next}[0] &:= \emptyset \\ \text{next}[1] &:= \text{next}[0] \end{aligned}$$

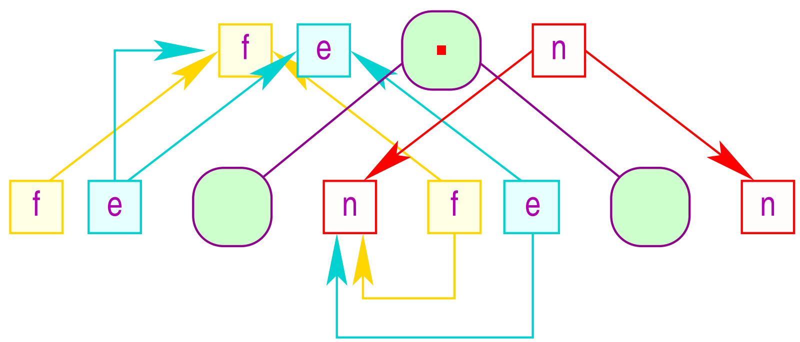




$$
\boxed{\;|\;} \quad : \quad
\begin{array}{lcl}
\mathsf{empty}[0] & := & \mathsf{empty}[1] \vee \mathsf{empty}[2] \\
\mathsf{first}[0] & := & \mathsf{first}[1] \cup \mathsf{first}[2] \\
\mathsf{next}[1] & := & \mathsf{next}[0] \\
\mathsf{next}[2] & := & \mathsf{next}[0]
\end{array}
$$

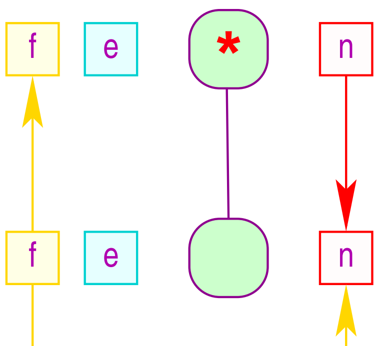


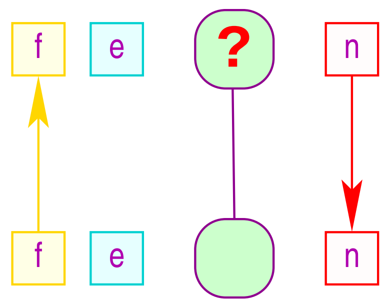
$$
\begin{aligned}
\boxed{\cdot} \quad : \qquad \mathsf{empty}[0] &:= \mathsf{empty}[1] \wedge \mathsf{empty}[2] \\
\mathsf{first}[0] &:= \mathsf{first}[1] \cup (\mathsf{empty}[1])\ ?\ \mathsf{first}[2] : \emptyset \\
\mathsf{next}[1] &:= \mathsf{first}[2] \cup (\mathsf{empty}[2])\ ?\ \mathsf{next}[0]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



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

$$\boxed{?} \quad : \quad \begin{aligned} \mathsf{empty}[0] &:= t \\ \mathsf{first}[0] &:= \mathsf{first}[1] \\ \mathsf{next}[1] &:= \mathsf{next}[0] \end{aligned}$$





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) In der Praxis wirds schon nicht so schlimm kommen ;-}

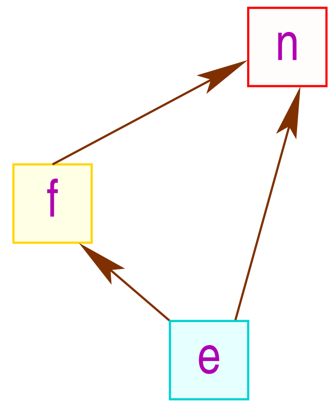
(3) Betrachte Teilklassen ...

Stark azyklische Attributierung:

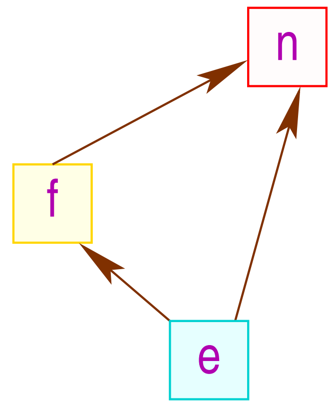
Berechne eine partielle Ordnung auf den Attributen eines Knotens, die kompatibel mit den lokalen Attribut-Abhängigkeiten ist:

- 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:

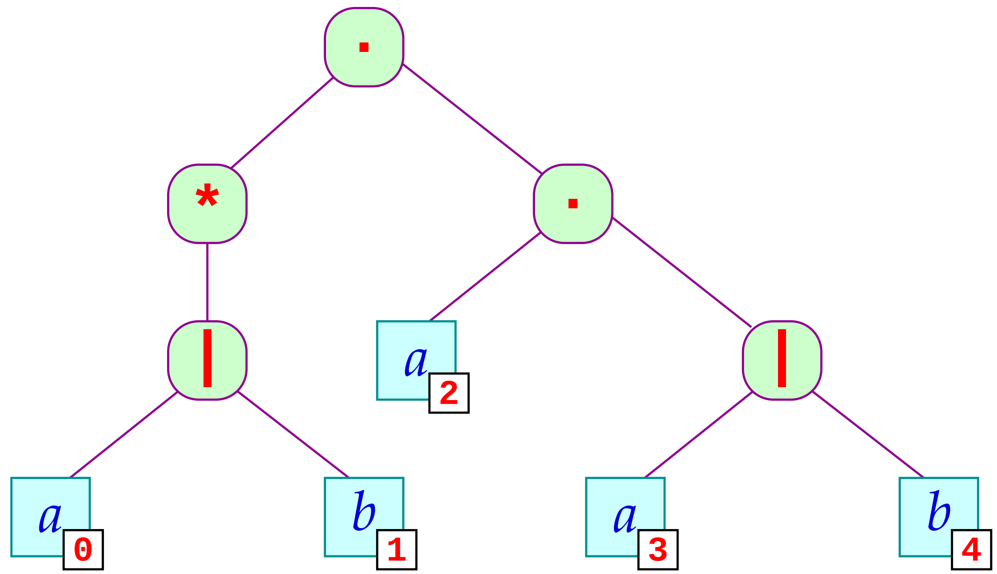
- Die Berechnung der partiellen Ordnung $\sqsubseteq$ ist eine Fixpunkt-Berechnung :-)
- Die partielle Ordnung können wir in eine lineare Ordnung einbetten ...
- Die lineare Ordnung gibt uns an, in welcher Reihenfolge die Attribute berechnet werden müssen :-)
- Die lokalen Abhängigkeitsgraphen zusammen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

Mögliche Strategien:

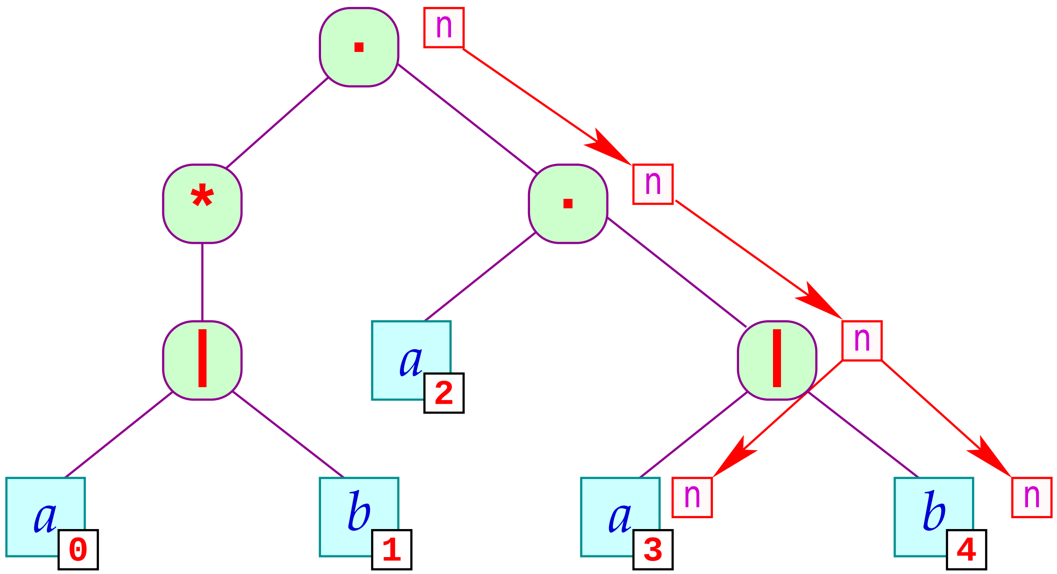
(1) **Bedarfsgetriebene Auswertung:**

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

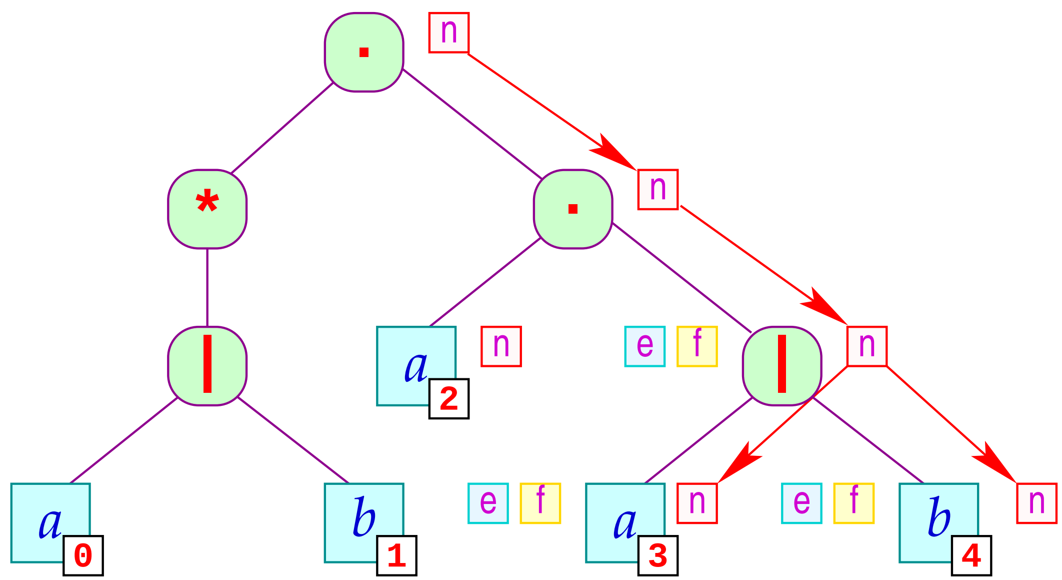
Beispiel, bedarfsgetrieben:



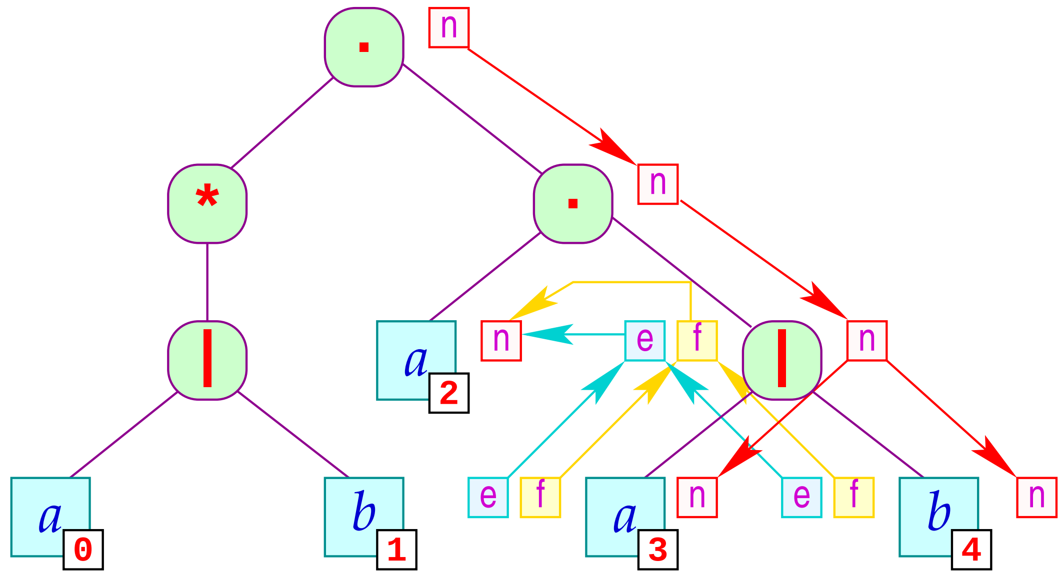
Beispiel, bedarfsgetrieben:



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:

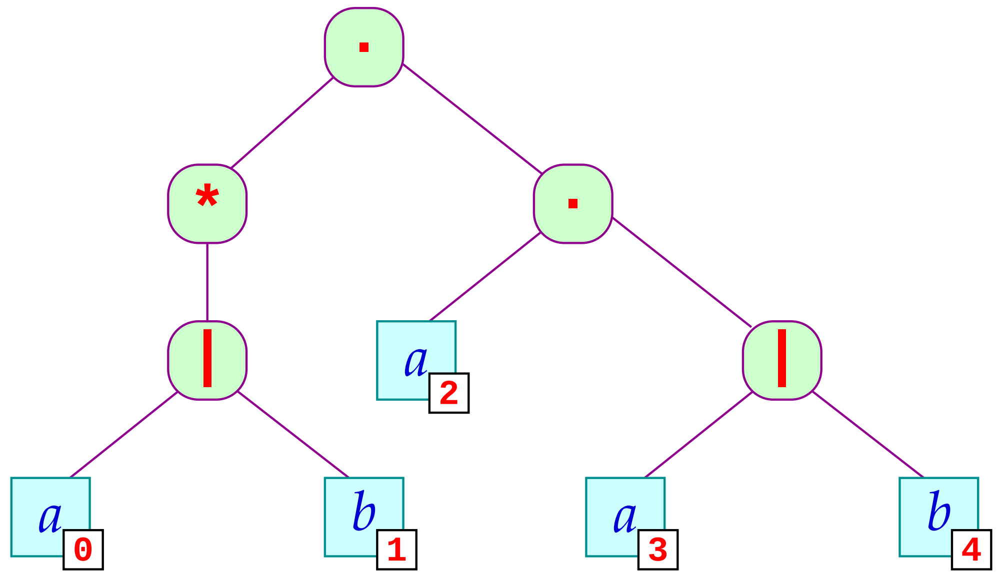
- Das minimale Attribut in der Anordnung für stark azyklische Attributierungen lässt sich stets in einem Pass berechnen :-)
- Man braucht folglich für stark azyklische Attributierungen maximal so viele Pässe, wie es Attribute gibt :-))
- Hat man einen Baum-Durchlauf zur Berechnung einiger Attribute, 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:

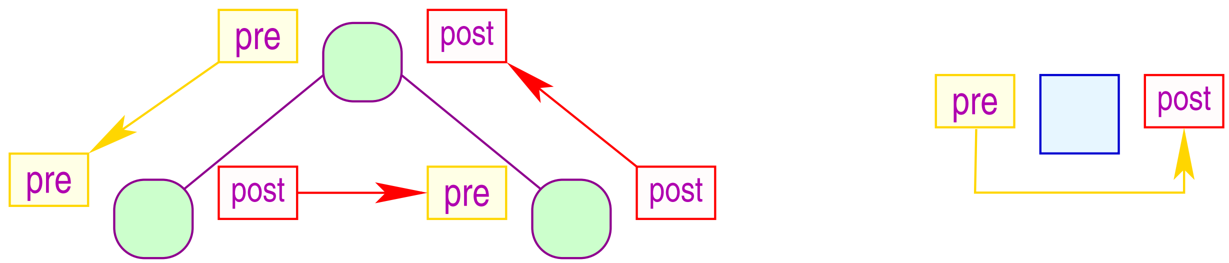


Idee:

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

$$
\begin{array}{llll}
\text{Root:} & \mathsf{pre}[0] & := & 0 \\
& \mathsf{pre}[1] & := & \mathsf{pre}[0] \\
& \mathsf{post}[0] & := & \mathsf{post}[1] \\
\\
\text{Node:} & \mathsf{pre}[1] & := & \mathsf{pre}[0] \\
& \mathsf{pre}[2] & := & \mathsf{post}[1] \\
& \mathsf{post}[0] & := & \mathsf{post}[2] \\
\\
\text{Leaf:} & \mathsf{post}[0] & := & \mathsf{pre}[0] + 1
\end{array}
$$

... 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:

  $\longrightarrow$ Attribute für jedes Nichtterminal;

  $\longrightarrow$ lokale Attribut-Gleichungen.

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