## Beispiel:

$$
\begin{array}{lllll}
\textbf{let} & \textsf{apply} & = & \textbf{fn}\ f \Rightarrow \textbf{fn}\ x \Rightarrow f\ x; \\
& \textsf{inc} & = & \textbf{fn}\ y \Rightarrow y + 1; \\
& \textsf{single} & = & \textbf{fn}\ y \Rightarrow y : [\,] \\
\textbf{in} & \multicolumn{4}{l}{\textsf{apply single}\ (\textsf{apply inc}\ 1)} \\
\textbf{end} & & & 
\end{array}
$$

## Beispiel:

$$\begin{array}{lll}
\textbf{let}\ \ \mathsf{apply} & = & \textbf{fn}\ f \Rightarrow \textbf{fn}\ x \Rightarrow f\ x; \\
\quad\ \ \mathsf{inc} & = & \textbf{fn}\ y \Rightarrow y+1; \\
\quad\ \ \mathsf{single} & = & \textbf{fn}\ y \Rightarrow y : [\,] \\
\textbf{in}\ \ \mathsf{apply\ single}\ (\mathsf{apply\ inc}\ 1) & & \\
\textbf{end} & &
\end{array}$$

Wir finden:

$$\begin{array}{lll}
\alpha[\mathsf{apply}] & = & (\alpha \to \beta) \to \alpha \to \beta \\
\alpha[\mathsf{inc}] & = & \textbf{int} \to \textbf{int} \\
\alpha[\mathsf{single}] & = & \gamma \to \mathsf{list}\ \gamma
\end{array}$$

- Durch die Anwendung: $\mathsf{apply\ single}$ erhalten wir:

$$\begin{array}{lcl}
\alpha & = & \gamma \\
\beta & = & \mathsf{list}\ \gamma \\
\alpha[\mathsf{apply}] & = & (\gamma \to \mathsf{list}\ \gamma) \to \gamma \to \mathsf{list}\ \gamma
\end{array}$$

- Durch die Anwendung: $\mathsf{apply\ inc}$ erhalten wir:

$$\begin{array}{lcl}
\alpha & = & \mathbf{int} \\
\beta & = & \mathbf{int} \\
\alpha[\mathsf{apply}] & = & (\mathbf{int} \to \mathbf{int}) \to \mathbf{int} \to \mathbf{int}
\end{array}$$

$\Longrightarrow$ Typ-Fehler ???

**Idee 1:** Kopiere jede Definition für jede Benutzung ...

... im Beispiel:

$$
\begin{array}{lllllll}
\textbf{let} & \textsf{apply} & = & \textbf{fn}\ f & \Rightarrow & \textbf{fn}\ x \ \Rightarrow & f\ x; \\
 & \textsf{inc} & = & \textbf{fn}\ y & \Rightarrow & y+1; & \\
 & \textsf{single} & = & \textbf{fn}\ y & \Rightarrow & y : [\,] & \\
\multicolumn{7}{l}{\textbf{in}\ \ (\textbf{fn}\ f \ \Rightarrow\ \textbf{fn}\ x \ \Rightarrow\ f\ x)\ \textsf{single}} \\
\multicolumn{7}{l}{\quad ((\textbf{fn}\ f \ \Rightarrow\ \textbf{fn}\ x \ \Rightarrow\ f\ x)\ \textsf{inc}\ 1)} \\
\multicolumn{7}{l}{\textbf{end}}
\end{array}
$$

## Idee 1: Kopiere jede Definition für jede Benutzung ...

## ... im Beispiel:

$$
\begin{array}{llllllll}
\textbf{let} & \textsf{apply} & = & \textbf{fn}\ f & \Rightarrow & \textbf{fn}\ x & \Rightarrow & f\ x; \\
 & \textsf{inc} & = & \textbf{fn}\ y & \Rightarrow & y+1; & & \\
 & \textsf{single} & = & \textbf{fn}\ y & \Rightarrow & y : [\,] & & \\
\textbf{in} & \multicolumn{7}{l}{(\textbf{fn}\ f \Rightarrow \textbf{fn}\ x \Rightarrow f\ x)\ \textsf{single}} \\
 & \multicolumn{7}{l}{((\textbf{fn}\ f \Rightarrow \textbf{fn}\ x \Rightarrow f\ x)\ \textsf{inc}\ 1)} \\
\textbf{end} & & & & & & &
\end{array}
$$

+ Die beiden Teilausdrücke $(\textbf{fn}\ f \Rightarrow \textbf{fn}\ x \Rightarrow f\ x)$ erhalten jeweils einen eigenen Typ mit unabhängigen Typ-Variablen :-)
+ Das expandierte Programm ist typbar :-))
- Das expandierte Programm kann seeehr groß werden :-(
- Typ-Checking ist nicht mehr modular :-((

## Idee 2: Kopiere die Typen für jede Benutzung ...

- Wir erweitern Typen zu Typ-Schemata:

$$
\begin{array}{lll}
t & ::= & \alpha \mid \mathbf{bool} \mid \mathbf{int} \mid (t_1, \ldots, t_m) \mid \mathsf{list}\ t \mid t_1 \to t_2 \\
\sigma & ::= & t \mid \forall\, \alpha_1, \ldots, \alpha_k.t
\end{array}
$$

- Achtung: Der Operator $\forall$ erscheint nur auf dem Top-Level !!!
- Typ-Schemata werden für **let**-definierte Variablen eingeführt.
- Bei deren Benutzung wird der Typ im Schema mit frischen Typ-Variablen instantiiert ...

# Neue Regeln:

$$\text{Inst:}\quad \frac{\Gamma(x) = \forall\,\alpha_1,\ldots,\alpha_k.t}{\Gamma \;\vdash\; x \;:\; t[t_1/\alpha_1,\ldots,t_k/\alpha_k]} \qquad (t_1,\ldots,t_k \quad \text{beliebig})$$

$$\text{Let:}\quad \frac{\begin{array}{lllll} \Gamma_0 & \vdash & e_1 & : & t_1 \qquad \Gamma_1 = \Gamma_0 \oplus \{x_1 \mapsto \mathsf{close}\; t_1\; \Gamma_0\} \\ & \ldots & & & \qquad\qquad \ldots \\ \Gamma_{m-1} & \vdash & e_m & : & t_m \qquad \Gamma_m = \Gamma_{m-1} \oplus \{x_m \mapsto \mathsf{close}\; t_m\; \Gamma_{m-1}\} \\ \Gamma_m & \vdash & e_0 & : & t_0 \end{array}}{\Gamma_0 \;\vdash\; (\mathbf{let}\; x_1 = e_1; \ldots; x_m = e_m\; \mathbf{in}\; e_0) \;:\; t_0}$$

Der Aufruf $\mathsf{close}\ t\ \Gamma$ macht alle Typ-Variablen in $t$ generisch (d.h. instantiierbar), die nicht auch in $\Gamma$ vorkommen ...

$$\begin{array}{l} \textbf{fun}\ \ \mathsf{close}\ t\ \Gamma \ \ = \ \ \textbf{let} \\ \qquad\qquad \textbf{val}\ \ \alpha_1, \ldots, \alpha_k = \mathsf{free}\,(t) \setminus \mathsf{free}\,(\Gamma) \\ \qquad\qquad \textbf{in}\ \ \forall\, \alpha_1, \ldots, \alpha_k.\, t \\ \qquad\qquad \textbf{end} \end{array}$$

Eine Instantiierung mit frischen Typ-Variablen leistet die Funktion:

$$\begin{array}{l} \textbf{fun}\ \ \mathsf{inst}\ \sigma \ \ = \ \ \textbf{let} \\ \qquad\qquad \textbf{val}\ \ \forall\, \alpha_1, \ldots, \alpha_k.\, t = \sigma \\ \qquad\qquad \textbf{val}\ \ \beta_1 = \mathsf{new}() \quad \ldots \quad \textbf{val}\ \ \beta_k = \mathsf{new}() \\ \qquad\qquad \textbf{in}\ \ t[\beta_1/\alpha_1, \ldots, \beta_k/\alpha_k] \\ \qquad\qquad \textbf{end} \end{array}$$

# Der Algorithmus $\mathcal{W}$ (erweitert):

$$
\begin{array}{lll}
 & \ldots & \\
\mid & x & \to \quad \mathsf{inst}\ (\Gamma(x)) \\
\mid & (\mathbf{let}\ x_1 = e_1; \ldots; x_m = e_m\ \mathbf{in}\ e_0) & \\
 & & \to \quad \mathbf{let}\ \ \mathbf{val}\ \ (t_1, \theta) = \mathcal{W}\ e_1\ (\Gamma, \theta) \\
 & & \qquad\qquad \mathbf{val}\ \ \sigma_1 = \mathsf{close}\ (\theta\ t_1)\ (\theta\ \Gamma) \\
 & & \qquad\qquad \mathbf{val}\ \ \Gamma = \Gamma \oplus \{x_1 \mapsto \sigma_1\} \\
 & & \qquad\qquad\qquad\quad \ldots \\
 & & \qquad\qquad \mathbf{val}\ \ (t_m, \theta) = \mathcal{W}\ e_m\ (\Gamma, \theta) \\
 & & \qquad\qquad \mathbf{val}\ \ \sigma_m = \mathsf{close}\ (\theta\ t_m)\ (\theta\ \Gamma) \\
 & & \qquad\qquad \mathbf{val}\ \ \Gamma = \Gamma \oplus \{x_m \mapsto \sigma_m\} \\
 & & \qquad\qquad \mathbf{val}\ \ (t_0, \theta) = \mathcal{W}\ e_0\ (\Gamma, \theta) \\
 & & \qquad\ \ \mathbf{in}\ \ (t_0, \theta) \\
 & & \qquad\ \ \mathbf{end}
\end{array}
$$

## Beispiel:

$$
\begin{array}{lll}
\textbf{let } \mathsf{apply} & = & \textbf{fn } f \Rightarrow \textbf{fn } x \Rightarrow f\, x; \\
\quad \mathsf{inc} & = & \textbf{fn } y \Rightarrow y + 1; \\
\quad \mathsf{single} & = & \textbf{fn } y \Rightarrow y : [\,] \\
\multicolumn{3}{l}{\textbf{in } \mathsf{apply\ single\ (apply\ inc\ } 1)} \\
\textbf{end} & &
\end{array}
$$

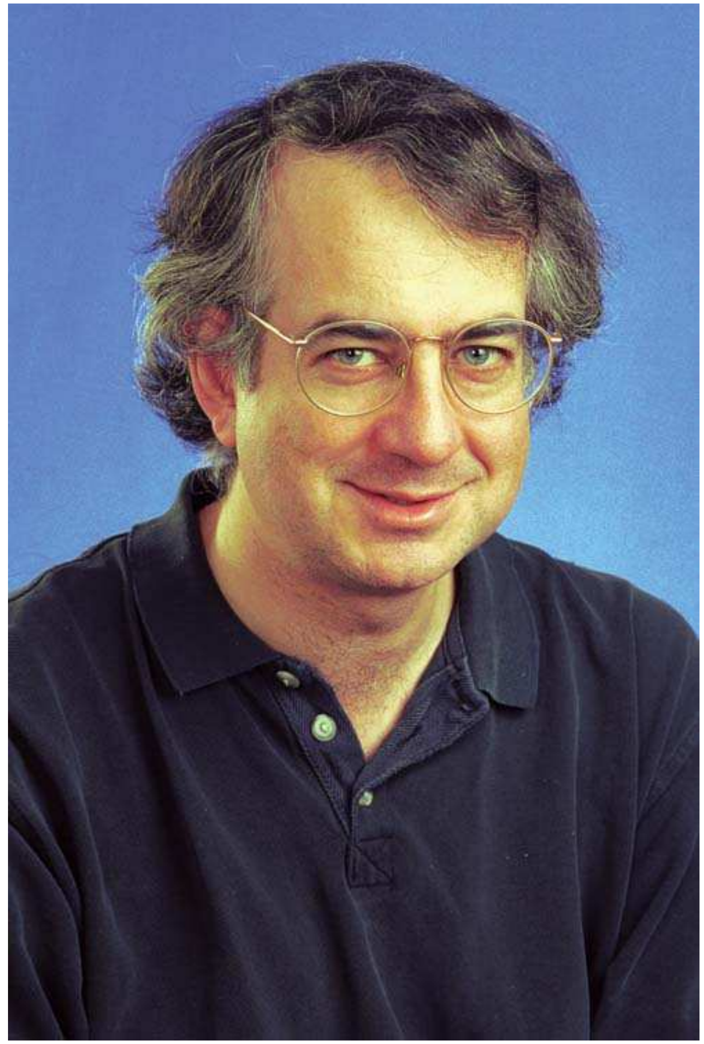
Wir finden:

$$
\begin{array}{lll}
\alpha[\mathsf{apply}] & = & \forall\, \alpha, \beta.\, (\alpha \to \beta) \to \alpha \to \beta \\
\alpha[\mathsf{inc}] & = & \textbf{int} \to \textbf{int} \\
\alpha[\mathsf{single}] & = & \forall\, \gamma.\, \gamma \to \mathsf{list}\ \gamma
\end{array}
$$

## Bemerkungen:

- Der erweiterte Algorithmus berechnet nach wie vor allgemeinste Typen :-)
- Instantiierung von Typ-Schemata bei jeder Benutzung ermöglicht polymorphe Funktionen sowie modulare Typ-Inferenz :-))
- Die Möglichkeit der Instantiierung erlaubt die Codierung von DEXPTIME-schwierigen Problemen in die Typ-Inferenz ??

  ... ein in der Praxis eher marginales Problem :-)
- Die Einführung von Typ-Schemata ist nur für nicht-rekursive Definitionen möglich: die Ermittlung eines allgemeinsten Typ-Schemas für rekursive Definitionen ist nicht berechenbar !!!



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:

$$e \ ::= \ \ldots \ | \ \mathbf{ref}\ e \ | \ !e \ | \ e_1 := e_2$$

## Beispiel:

$$\begin{array}{llll}
\mathbf{let} & \mathsf{count} & = & \mathbf{ref}\ 0; \\
 & \mathsf{new} & = & \mathbf{fn}\ () \Rightarrow \mathbf{let} \\
 & & & \quad \mathsf{ret} = !\mathsf{count}; \\
 & & & \quad _ = \mathsf{count} := \mathsf{ret} + 1 \\
 & & & \quad \mathbf{in}\ \mathsf{ret} \\
\mathbf{in} & \multicolumn{3}{l}{\mathsf{new}() + \mathsf{new}()}
\end{array}$$

Als neuen Typ benötigen wir:

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

# 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:

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

# Achtung:

Diese Regeln vertragen sich nicht mit Polymorphie !!!

# Beispiel:

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

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:

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

## 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:

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

## ... 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:

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

## 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:

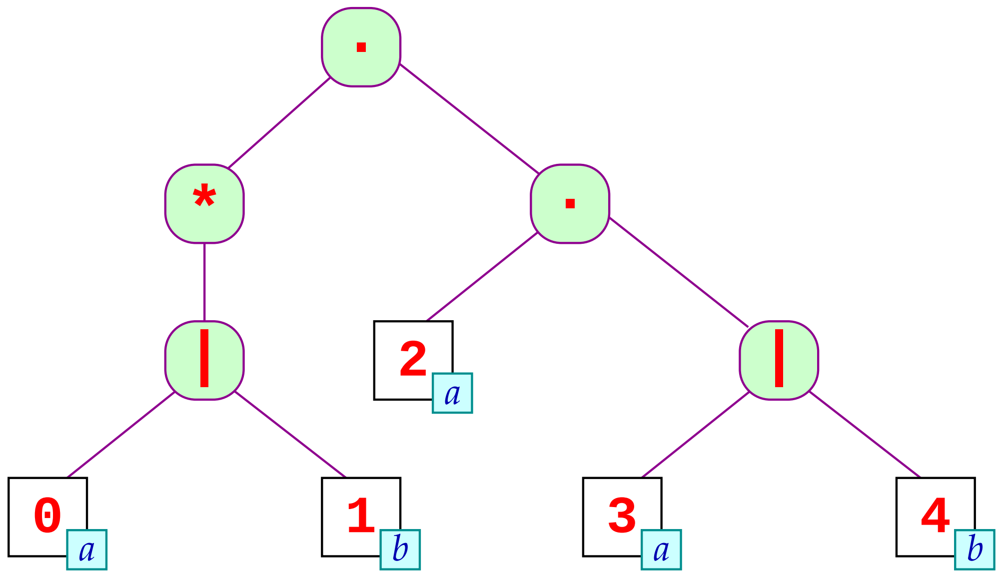
$$
\begin{array}{llll}
\textbf{fun}\ \ \mathsf{member}\ x\ \mathsf{list} & = & \textbf{case}\ \ \mathsf{list} & \\
& & \textbf{of}\quad [\,] & \to\ \ \textbf{false} \\
& & \mid\quad h{::}t & \to\ \ \textbf{if}\ \ x = h\ \ \textbf{then}\ \ \textbf{true} \\
& & & \phantom{\to}\ \ \textbf{else}\ \ \mathsf{member}\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

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

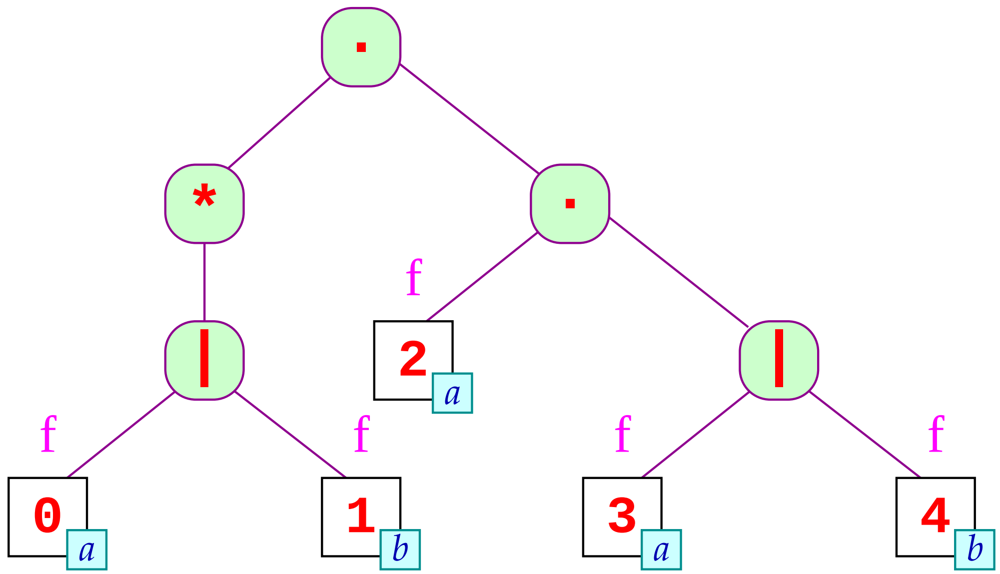
## 3.4 Attributierte Grammatiken

- 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 ...

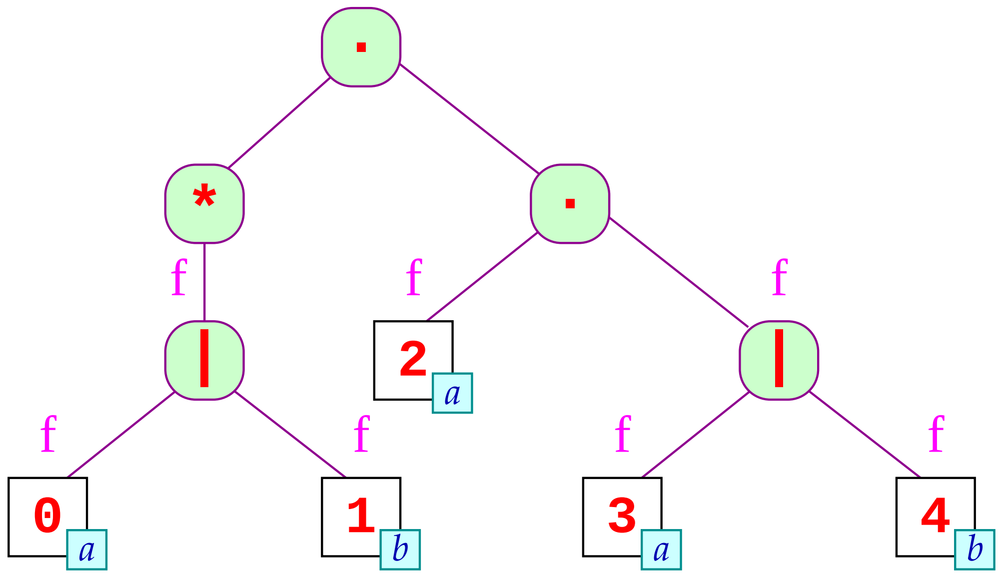
**Beispiel:** Berechnung des Prädikats empty$[r]$



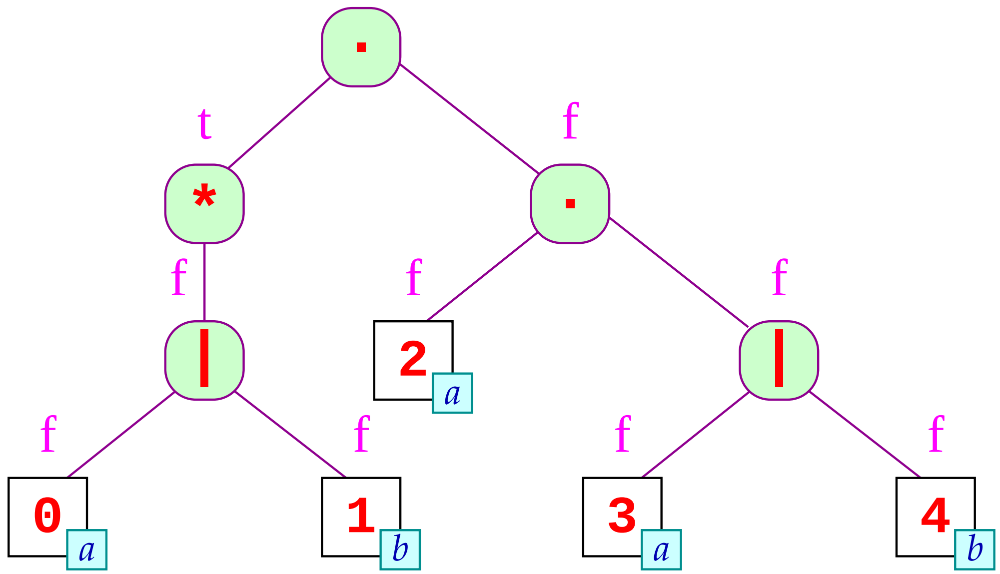
**Beispiel:** Berechnung des Prädikats empty$[r]$



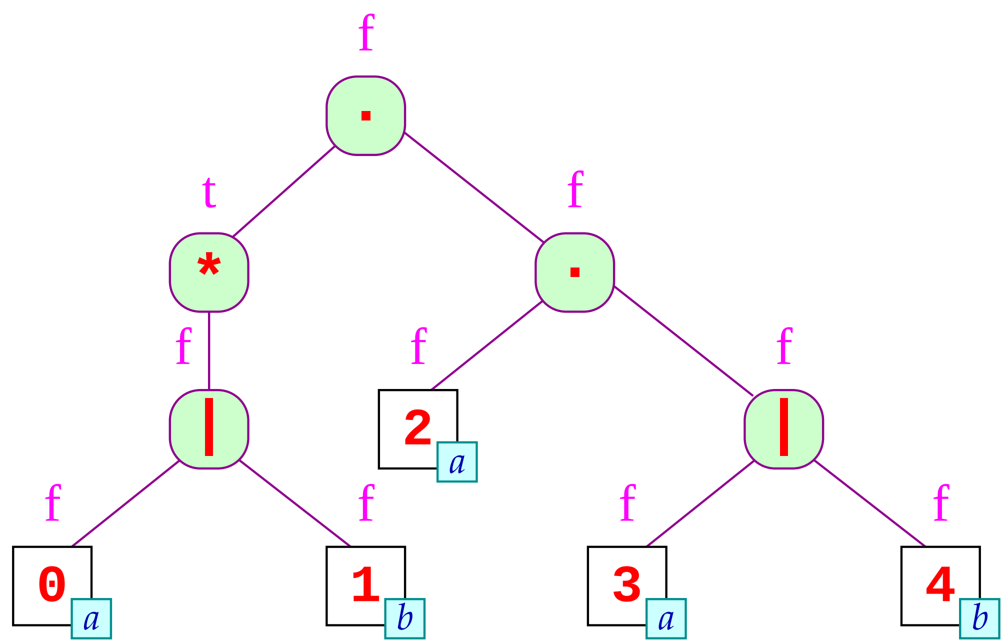
**Beispiel:** Berechnung des Prädikats $\text{empty}[r]$



**Beispiel:** Berechnung des Prädikats empty[$r$]



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



# Idee zur Implementierung:

- Für jeden Knoten führen wir ein Attribut `empty` ein.
- Die Attribute werden in einer DFS post-order Traversierung berechnet:

  $\longrightarrow$ An einem Blatt lässt sich der Wert des Attributs unmittelbar ermitteln ;-)

  $\longrightarrow$ 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 Einfachkeit geben wir ihnen einen fortlaufenden Index:

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

  $\mathsf{empty}[i]$ : 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 Einfachheit geben wir ihnen einen fortlaufenden Index:

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

# ... im Beispiel:

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

# Diskussion:

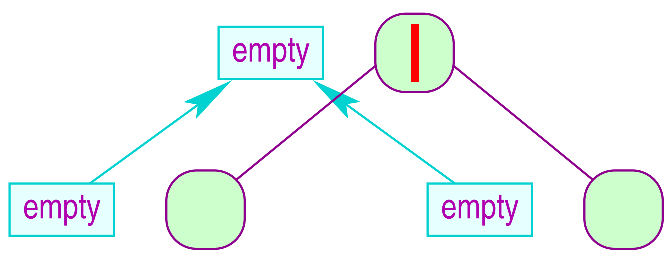
- Die lokalen Berechnungen der Attributwerte müssen zu einem globalen Algorithmus zusammen gesetzt werden :-)
- Dazu benötigen wir:

  (1) eine Besuchsreihenfolge der Knoten des Baums;

  (2) lokale Berechnungsreihenfolgen ...

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

# ... im Beispiel:



# Achtung:

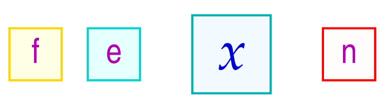
- Zur Ermittlung einer Auswertungsstrategie reicht es nicht, sich die lokalen Attribut-Abhängigkeiten anzusehen.
- Es kommt auch darauf an, wie sie sich global zu einem Abhängigkeitsgraphen zusammen setzen !!!
- Im Beispiel sind die Abhängigkeiten stets von den Attributen der Söhne zu den Attributen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

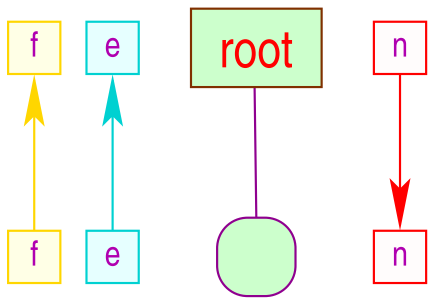
  $\Longrightarrow$ Postorder-DFS-Traversierung
- Die Variablen-Abhängigkeiten können aber auch komplizierter sein ...

**Beispiel:** Simultane Berechnung von empty, first, next :

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

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



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