Beispiel

$$
\begin{array}{l}
\textbf{class}\ \mathsf{Eq}\ \textbf{where} \\
\qquad (=) : \quad \forall\, \alpha \in C.\ \alpha \to \alpha \to \textbf{bool}
\end{array}
$$

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

Bemerkungen

- I.a. ist es praktischer, mehrere Operationen zu einer Klasse zusammenzufassen – z.B. um eine Klasse Number zu definieren, mit den üblichen vier Grundrechenarten.

  In dieser Hinsicht verhält sich eine Klasse ganz ähnlich wie ein Interface ;-)

- Eine Klassendeklaration kann auch direkt diverse abgeleitete Operationen implementieren, wie z.B. eine Gleichheit, falls es nur ein $\leq$ gibt.

  Insofern könnte man damit generisch eine Klasse zu einer Unterklasse einer andern machen :-)

- Praktisch wird man zusätzlich zu den vom System bereit gestellten Typen auch Systemklassen bereitstellen, in die die eingebauten Typen eingeordnet sind.

Wie inferiert man Klassen?

Idee 1:

1. Ignoriere die Klassenbedingungen;
   Inferiere für jeden Ausdruck den polymorphen Typ;
2. Überprüfe für jedes Vorkommen von überladenen Operatoren, dass die entsprechenden Typen den Operator auch implementieren!
3. Wie übersetzt man getypte Programme?

Idee 2:

- Modifiziere polymorphe Typinferenz so, dass sie bei der Einführung eines Typschemas jeweils die notwendigen Bedingungen mit vermerkt;
- Verwalte dazu neben Γ eine Sortenumgebung S, die für jede Typvariable die Menge der für sie benötigten Klassen sammelt;
- neben dem Typ für jeden Teilausdruck eine Übersetzung liefert ...

  Aus $\Gamma\,,\; S \vdash e : \quad \forall \alpha \in \mathcal{C}.\sigma$ wird:

  $$\textbf{fn}\, \alpha \Rightarrow e'$$

- Insbesondere benötigen wir eine modifizierte Unifikation ...

Modifizierte Unifikation

Um den Algorithmus $\mathcal{W}$ zu modifizieren, benötigen wir eine Unifikationsfunktion, die die Klasseninformation mit verwaltet:

$$
\begin{array}{l}
\textbf{fun}\ \ \mathsf{class-unify}\ (\tau_1, \tau_2)\ S\ =\ \ \textbf{case}\ \ \mathsf{unify}\ (\tau_1, \tau_2)\ \emptyset \\
\qquad\qquad\qquad\qquad \textbf{of}\ \ \mathsf{Fail}\ \ \to\ \ \mathsf{Fail} \\
\qquad\qquad\qquad\qquad \mid\ \ \ \theta\ \ \ \ \to\ \ (\theta, \theta\, S)
\end{array}
$$

Dabei liefert $\theta\, S$ die Klassenannahmen, die sich aus den Klassenannahmen in S für die Typvariablen im Bild von θ ergeben, wenn man die Instanz-Deklarationen berücksichtigt ...

Beispiel

Instanz-Deklarationen:

$$\begin{array}{lcl} \mathsf{list} & : & \alpha \in \mathsf{Eq} \Rightarrow \mathsf{list}\,\alpha \in \mathsf{Eq} \\ \mathsf{set} & : & \alpha \in \mathsf{Comp} \Rightarrow \mathsf{set}\,\alpha \in \mathsf{Eq} \end{array}$$

Dann haben wir für:

$$S = \{\alpha \mapsto \mathsf{Eq}\} \qquad \theta = \{\alpha \mapsto \mathsf{list}\,(\mathsf{set}\,\beta)\}$$

die neue Menge:

$$\theta\,S = \{\beta \mapsto \mathsf{Comp}\}$$

Insbesondere ist die substituierte Variable aus S verschwunden.

Modifizierter Abschluss

Der Aufruf close $(t, e)\ \Gamma\ S$ macht alle Typ-Variablen in t beschränkt generisch gemäß S, die nicht auch in Γ vorkommen ...

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

Modifizierte Instantiierung

Die Instantiierung mit frischen Typ-Variablen leistet die Funktion:

$$
\begin{array}{l}
\textbf{fun}\ \ \mathsf{inst}\ (\sigma, x)\ \ =\ \ \textbf{let} \\
\qquad\qquad \textbf{val}\ \ \forall\, \alpha_1 \in S_1, \ldots, \alpha_k \in S_k.\, t = \sigma \\
\qquad\qquad \textbf{val}\ \ \beta_1 = \mathsf{new}() \quad \ldots \quad \textbf{val}\ \ \beta_k = \mathsf{new}() \\
\qquad\qquad \textbf{val}\ \ t = t[\beta_1/\alpha_1, \ldots, \beta_k/\alpha_k] \\
\qquad\qquad \textbf{in}\ \ (t, x\, \beta_1 \ldots \beta_k, \{\beta_1 \mapsto S_1, \ldots, \beta_k \mapsto S_k\}) \\
\qquad\qquad \textbf{end}
\end{array}
$$

Bemerkung

- Bei der Transformation sollten nur diejenigen Typparameter zu Funktionsparametern werden, die durch Typklassen beschränkt sind :-)
- Die Transformation fügt nicht-generische Typvariablen in die Ausgabeausdrücke ein :-)
- Während der Unifikation werden diese Variablen gebunden. Entsprechend werden sie nicht nur in den Typen, sondern auch in den Ausdrücken substituiert.
- Ein Aufruf op τ für einen Operator op der Klasse C kann dann zur Laufzeit aufgelöst werden, indem die Implementierung des Operators in der Instanzdeklaration von τ nachgeschlagen wird.

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

$$
\begin{array}{llll}
 & \ldots & & \\
| & x & \rightarrow & \textbf{let}\ (t, e, S') = \mathsf{inst}\ (\Gamma(x), x) \\
 & & & \textbf{in}\ (t, e, S \cup S', \theta) \\
 & & & \textbf{end} \\
| & \multicolumn{3}{l}{(\textbf{let}\ x_1 = e_1; \ldots; x_m = e_m\ \textbf{in}\ e_0)} \\
 & & \rightarrow & \textbf{let}\ \ \textbf{val}\ \ (t_1, e_1, S, \theta) = \mathcal{W}\ e_1\ (\Gamma, S, \theta) \\
 & & & \qquad \textbf{val}\ \ (\sigma_1, e_1, S) = \mathsf{close}\ (\theta\ t_1, \theta\ e_1)\ (\theta\ \Gamma)\ S \\
 & & & \qquad \textbf{val}\ \ \Gamma = \Gamma \oplus \{x_1 \mapsto \sigma_1\} \\
 & & & \qquad\qquad\qquad \ldots \\
 & & & \qquad \textbf{val}\ \ (t_m, e_m, S, \theta) = \mathcal{W}\ e_m\ (\Gamma, S, \theta) \\
 & & & \qquad \textbf{val}\ \ (\sigma_m, e_m, S) = \mathsf{close}\ (\theta\ t_m, \theta\ e_m)\ (\theta\ \Gamma)\ S \\
 & & & \qquad \textbf{val}\ \ \Gamma = \Gamma \oplus \{x_m \mapsto \sigma_m\} \\
 & & & \qquad \textbf{val}\ \ (t_0, e_0, S, \theta) = \mathcal{W}\ e_0\ (\Gamma, S, \theta) \\
 & & & \qquad \textbf{val}\ \ e = \textbf{let}\ x_1 = e_1; \ldots; x_m = e_m\ \textbf{in}\ e_0 \\
 & & & \textbf{in}\ \ (t_0, e, S, \theta) \\
 & & & \textbf{end}
\end{array}
$$

Bemerkungen

- Die Typinferenz/Transformation startet mit $S_0 = \emptyset$ und

$$\Gamma_0 = \{\mathsf{op} \mapsto \sigma_{\mathsf{op}} \mid \mathsf{op} \text{ Operator}\}$$

- Bei jeder Instanz-Deklaration

$$\beta_1 \in \mathcal{C}_1, \ldots, \beta_k \in \mathcal{C}_k \Rightarrow \mathsf{c}(\beta_1, \ldots, \beta_k) \in C$$

  muss überprüft werden, ob für die Definition des Operators
  $\mathsf{op} : \quad \forall\, \alpha \in C.\, \tau$ gilt:

$$\mathcal{W}\, e\, (\Gamma_0, \emptyset, \emptyset) \quad = \quad (\tau', S, _) \qquad \text{mit} \qquad \tau' = \tau[\mathsf{c}(\beta_1, \ldots, \beta_k)/\alpha]$$

  wobei:

$$S \subseteq \{\beta_1 \mapsto \mathcal{C}_1, \ldots, \beta_k \mapsto \mathcal{C}_k\}$$

Bemerkungen (Forts.)

...

- Am Ende wird die Substitution θ auf alle (freien Vorkommen von) Typvariablen im transformierten Ausdruck angewendet.
- Durch Pattern Matching auf den Typausdrücken wird die richtige Implementierung der Operatoren ausgewählt ...

$$
\begin{array}{lll}
\mathsf{op} = \textbf{fn}\ \beta \Rightarrow & \textbf{case}\ \beta\ \textbf{of} \\
& \ldots \\
& \mathsf{c}(\beta_1, \ldots, \beta_k) \to \mathsf{op}_{\mathsf{c}}\ \beta_1\ \ldots\ \beta_k \\
& \ldots
\end{array}
$$

… im Beispiel:

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

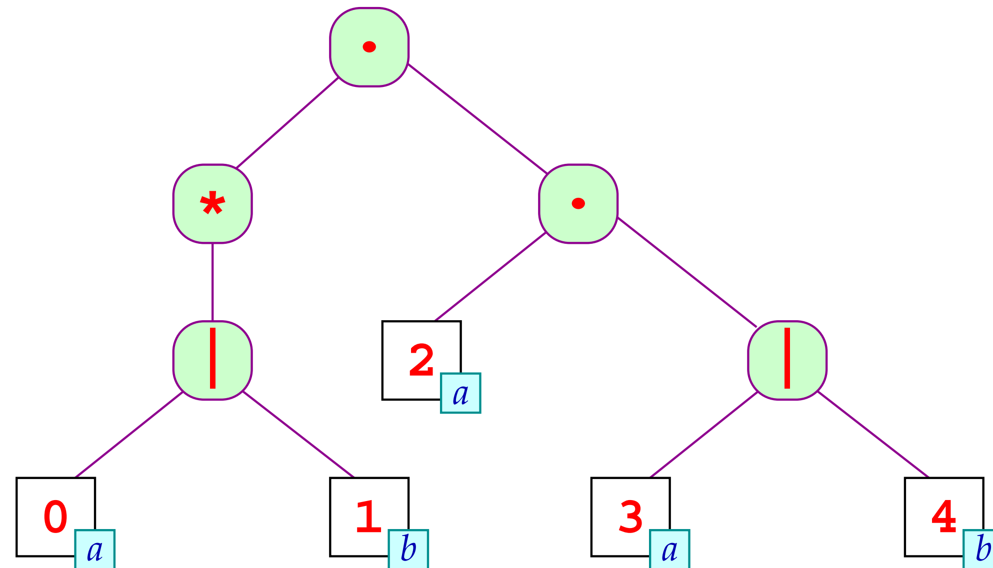
Schlussbemerkung

- 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 :-(

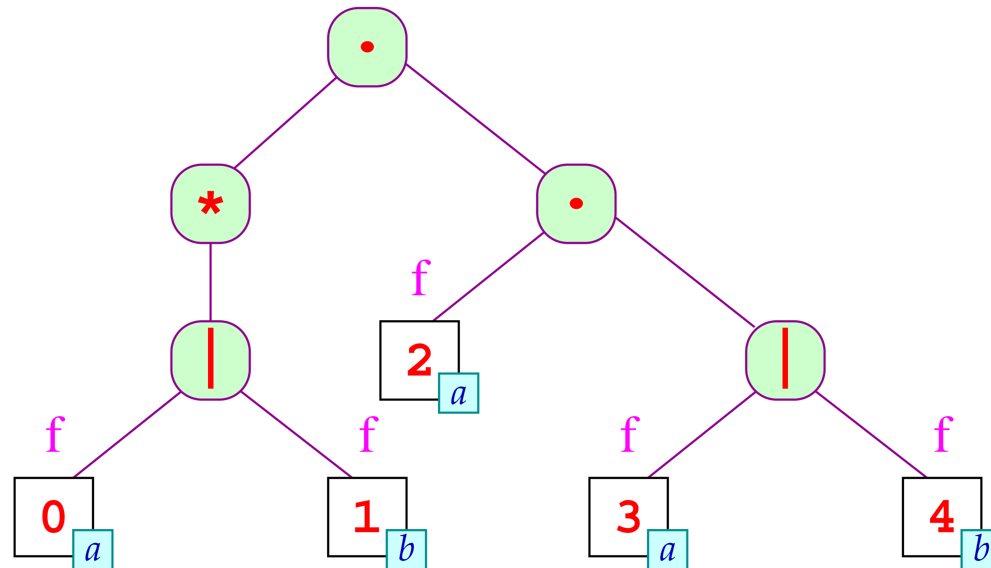
3.4 Attributierte Grammatiken

- Viele 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 ...

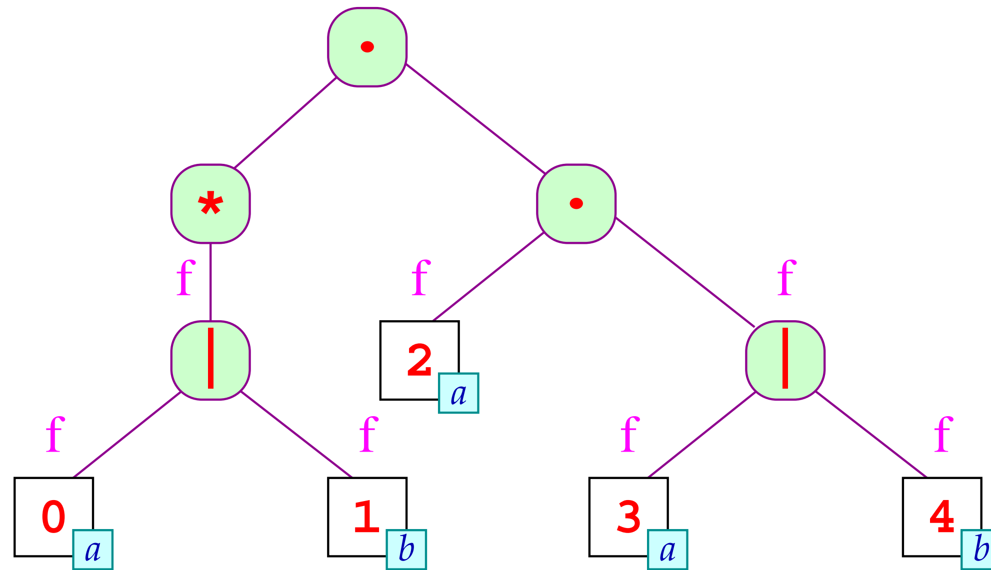
Beispiel: Berechnung des Prädikats $\mathsf{empty}[r]$



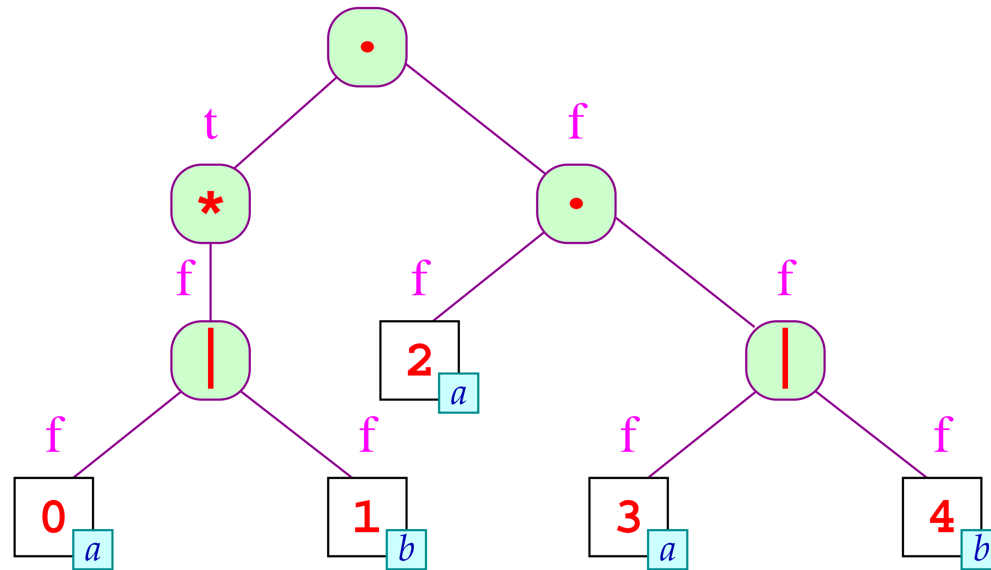
Beispiel: Berechnung des Prädikats $\mathsf{empty}[r]$



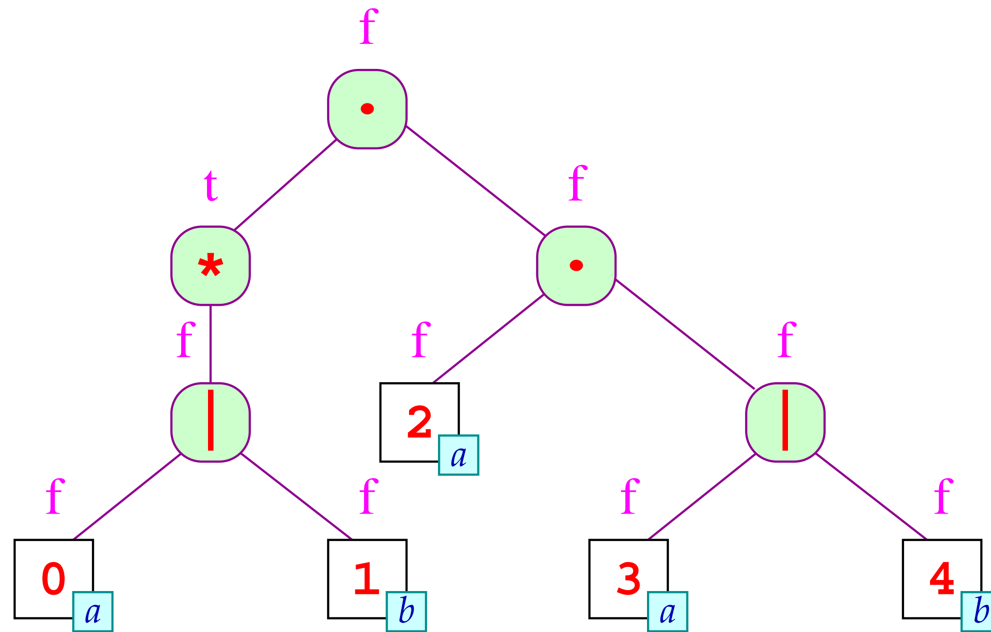
Beispiel: Berechnung des Prädikats $\mathsf{empty}[r]$



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



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



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:

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

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:

  empty$[0]$: das Attribut des aktuellen Knotens

  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 Einfachheit geben wir ihnen einen fortlaufenden Index:

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

... im Beispiel:

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

Diskussion:

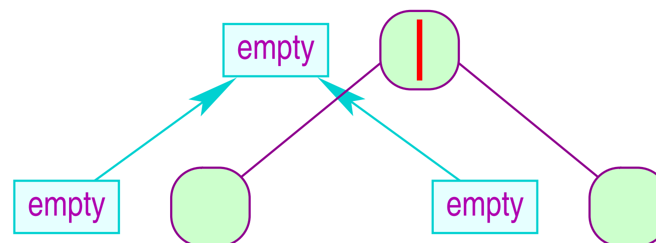
- Die lokalen Berechnungen der Attributwerte müssen zu einem globalen Algorithmus zusammen gesetzt werden :-)
- Dazu benötigen wir:

  (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 ...