Tutorium 12: Syntaktische Analyse

Paul Brinkmeier

09. Februar 2021

Tutorium Programmierparadigmen am KIT

Heutiges Programm

Programm

- Typinferenz
  - Herleitungsbäume
  - Let-Polymorphismus
- Übersicht Compilerbau
- Syntaktische Analyse

Typinferenz

Unifikation (Robinson, [Rob65])

Unifikationsalgorithmus: $\texttt{unify}(C) =$

$\texttt{if}\ C\ \texttt{==}\ \emptyset\ \texttt{then}\ []$
$\texttt{else let}\ \{\theta_l = \theta_r\} \cup C' = C\ \texttt{in}$
$\quad\texttt{if}\ \theta_l\ \texttt{==}\ \theta_r\ \texttt{then unify}(C')$
$\quad\texttt{else if}\ \theta_l\ \texttt{==}\ \texttt{Y and Y} \notin FV(\theta_r)\ \texttt{then unify}([\texttt{Y} \Leftarrow \theta_r]\, C') \circ [\texttt{Y} \Leftarrow \theta_r]$
$\quad\texttt{else if}\ \theta_r\ \texttt{==}\ \texttt{Y and Y} \notin FV(\theta_l)\ \texttt{then unify}([\texttt{Y} \Leftarrow \theta_l]\, C') \circ [\texttt{Y} \Leftarrow \theta_l]$
$\quad\texttt{else if}\ \theta_l\ \texttt{==}\ \texttt{f}(\theta_l^1, \ldots, \theta_l^n)\ \texttt{and}\ \theta_r\ \texttt{==}\ \texttt{f}(\theta_r^1, \ldots, \theta_r^n)$
$\qquad\quad\texttt{then unify}(C' \cup \{\theta_l^1 = \theta_r^1, \ldots, \theta_l^n = \theta_r^n\})$
$\quad\texttt{else}\ \textbf{fail}$

$\texttt{Y} \in FV(\theta)$ occur check, verhindert zyklische Substitutionen

Korrektheitstheorem

$\texttt{unify}(C)$ terminiert und gibt *mgu* für C zurück, falls C unifizierbar, ansonsten **`fail`**.

Beweis: Siehe [Pie02]

Wiederholung: Unifikationsalgorithmus

$$C = \{X = \mathtt{a}, Y = \mathtt{f}(X), \mathtt{f}(Z) = Y\}$$

- Problemstellung: Gleichungssystem C mit baumförmigen Termen
  - Bspw. Prolog-Terme, Typen
- Liefert: Unifikator σ_1, der alle Gleichungen erfüllt
- Bspw. $\sigma_1 = [X \Rightarrow \mathtt{a}, Y \Rightarrow \mathtt{f}(X), Z \Rightarrow X]$

Wiederholung: Unifikationsalgorithmus

$$C = \{X = \mathrm{a}, Y = \mathtt{f}(X), \mathtt{f}(Z) = Y\}$$

$$\begin{aligned}
\mathtt{unify}(C) &= \mathtt{unify}(\{\underline{X = \mathtt{a}}, Y = \mathtt{f}(X), \mathtt{f}(Z) = Y\}) \\
&= \mathtt{unify}(\{Y = \mathtt{f}(\mathrm{a}), \underline{\mathtt{f}(Z) = Y}\}) \circ [X \Rightarrow \mathtt{a}] \\
&= \mathtt{unify}(\{\underline{\mathtt{f}(Z) = \mathtt{f}(\mathrm{a})}\}) \circ [Y \Rightarrow \mathtt{f}(Z)] \circ [X \Rightarrow \mathtt{a}] \\
&= \mathtt{unify}(\{\underline{Z = \mathtt{a}}\}) \circ [Y \Rightarrow \mathtt{f}(Z)] \circ [X \Rightarrow \mathtt{a}] \\
&= [Z \Rightarrow \mathtt{a}] \circ [Y \Rightarrow \mathtt{f}(Z)] \circ [X \Rightarrow \mathtt{a}] \\
&= [Z \Rightarrow \mathtt{a}, Y \Rightarrow \mathtt{f}(\mathrm{a}), X \Rightarrow \mathtt{a}]
\end{aligned}$$

Klausuraufgabe WS16/18 A3 a) (6P.)

$$C_1 = \{\alpha_9 = \alpha_{10} \to \alpha_8, \alpha_9 = \alpha_4, \alpha_{10} = \texttt{bool}\}$$
$$C_2 = \{\alpha_{12} = \alpha_{13} \to \alpha_{11}, \alpha_{12} = \alpha_4, \alpha_{13} = \texttt{int}\}$$

i. Geben Sie allgemeinste Unifikatoren σ_1 für C_1 und σ_2 für C_2 an.
ii. Ist auch $C_1 \cup C_2$ unifizierbar?
iii. Ist der Ausdruck

$$\lambda a.\, \lambda f.\, f\ (a\ \texttt{true})\ (a\ 17)$$

typisierbar? Begründen Sie ihre Antwort <u>kurz</u>.

Klausuraufgabe WS16/18 A3 a) (6P.)

$$C_1 = \{\alpha_9 = \alpha_{10} \to \alpha_8, \alpha_9 = \alpha_4, \alpha_{10} = \texttt{bool}\}$$
$$C_2 = \{\alpha_{12} = \alpha_{13} \to \alpha_{11}, \alpha_{12} = \alpha_4, \alpha_{13} = \texttt{int}\}$$

Geben Sie allgemeinste Unifikatoren σ_1 für C_1 und σ_2 für C_2 an.

$$\begin{aligned}
\sigma_1 &= \texttt{unify}(\{\alpha_9 = \alpha_{10} \to \alpha_8, \alpha_9 = \alpha_4, \alpha_{10} = \texttt{bool}\}) \\
&= \ldots = [\alpha_9 \mapsto \texttt{bool} \to \alpha_8, \alpha_4 \mapsto \texttt{bool} \to \alpha_8, \alpha_{10} \mapsto \texttt{bool}] \\
\sigma_2 &= \texttt{unify}(\{\alpha_{12} = \alpha_{13} \to \alpha_{11}, \alpha_{12} = \alpha_4, \alpha_{13} = \texttt{int}\}) \\
&= \ldots = [\alpha_{12} \mapsto \texttt{int} \to \alpha_{11}, \alpha_4 \mapsto \texttt{int} \to \alpha_{11}, \alpha_{13} \mapsto \texttt{int}]
\end{aligned}$$

Klausuraufgabe WS16/18 A3 a) (6P.)

$$C_1 = \{\alpha_9 = \alpha_{10} \to \alpha_8, \alpha_9 = \alpha_4, \alpha_{10} = \texttt{bool}\}$$
$$C_2 = \{\alpha_{12} = \alpha_{13} \to \alpha_{11}, \alpha_{12} = \alpha_4, \alpha_{13} = \texttt{int}\}$$

Ist auch $C_1 \cup C_2$ unifizierbar?

$$\sigma_1 = ... = \left[\alpha_9 \Rightarrow \texttt{bool} \to \alpha_8, \underline{\alpha_4 \Rightarrow \texttt{bool} \to \alpha_8}, \alpha_{10} \Rightarrow \texttt{bool}\right]$$
$$\sigma_2 = ... = \left[\alpha_{12} \Rightarrow \texttt{int} \to \alpha_{11}, \underline{\alpha_4 \Rightarrow \texttt{int} \to \alpha_{11}}, \alpha_{13} \Rightarrow \texttt{int}\right]$$

A: Nein, da die allgemeinsten Unifikatoren σ_1 und σ_2 einen Konflikt für α_4 enthalten: $\texttt{unify}(\{\texttt{bool} = \texttt{int}\}) = \texttt{fail}$

Klausuraufgabe WS16/18 A3 a) (6P.)

$$C_1 = \{\alpha_9 = \alpha_{10} \to \alpha_8, \alpha_9 = \alpha_4, \alpha_{10} = \texttt{bool}\}$$
$$C_2 = \{\alpha_{12} = \alpha_{13} \to \alpha_{11}, \alpha_{12} = \alpha_4, \alpha_{13} = \texttt{int}\}$$

Ist der Ausdruck

$$\lambda a.\, \lambda f.\, f\ (a\ \texttt{true})\ (a\ 17)$$

typisierbar? Begründen Sie ihre Antwort kurz.

A: Nein, da a mit zwei verschiedenen Typen verwendet wird.

Typinferenz

Problemstellung bei Typinferenz: Zu einem gegebenen Term den passenden Typ finden.

- Struktur des Terms erkennen. Wo sind:
  - Lambdas?
  - Funktionsanwendungen?
  - Variablen/Konstanten?
- Entsprechenden Baum aufstellen.
- Typgleichungen finden.
- Gleichungssystem unifizieren.

Struktur von Lambda-Termen

Lambda-Terme bestehen aus

- Lambdas $\lambda p.\, b$
- Funktionsanwendungen $x\ y$
- Variablen x, Konstanten $\mathtt{true}, 17$

$$\lambda a.\, \lambda f.\, f\ (a\ \mathtt{true})$$

Struktur von Lambda-Termen

Lambda-Terme bestehen aus

- **Lambdas** $\lambda p.\, b$
- Funktionsanwendungen $x\ y$
- Variablen x, Konstanten $\texttt{true}, 17$

$$\underbrace{\lambda a.\ \underbrace{\lambda f.\, f\ (a\ \texttt{true})}_{\substack{\text{Lambda}\\ p=f,b=f\ (a\ \texttt{true})}}}_{\substack{\text{Lambda}\\ p=a,b=\lambda f.\, f\ (a\ \texttt{true})}}$$

Struktur von Lambda-Termen

Lambda-Terme bestehen aus

- Lambdas $\lambda p.\, b$
- **Funktionsanwendungen** $x\ y$
- Variablen x, Konstanten $\texttt{true}, 17$

$$\lambda a.\, \lambda f.\, \underbrace{f\ \underbrace{(a\ \texttt{true})}_{\substack{\text{Anwendung}\\ x=a, y=\texttt{true}}}}_{\substack{\text{Anwendung}\\ x=f, y=a\ \texttt{true}}}$$

Struktur von Lambda-Termen

Lambda-Terme bestehen aus

- Lambdas $\lambda p.\, b$
- Funktionsanwendungen $x\ y$
- **Variablen** x, Konstanten $\texttt{true}, 17$

$$\lambda a.\, \lambda f.\ \underbrace{f}_{\text{Variable}}\ (\underbrace{a}_{\text{Variable}}\ \texttt{true})$$

Struktur von Lambda-Termen

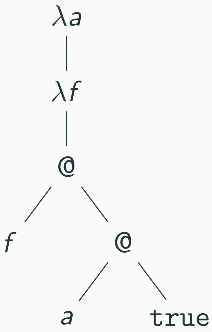
Lambda-Terme bestehen aus

- Lambdas $\lambda p.\, b$
- Funktionsanwendungen $x\ y$
- Variablen x, **Konstanten** $\texttt{true}, 17$

$$\lambda a.\, \lambda f.\, f\ (a\ \underbrace{\texttt{true}}_{\text{Konstante}})$$

Lambda-Terme als Bäume

Wir können Lambda-Terme also als Bäume mit Lambda- und Anwendungsknoten und Variablen- und Konstantenblättern betrachten, um ihre Struktur zu untersuchen:



$$\lambda a.\, \lambda f.\, f\ (a\ \mathtt{true})$$

Cheatsheet: Typisierter Lambda-Kalkül

$$\frac{\Gamma, p : \pi \vdash b : \rho}{\Gamma \vdash \lambda p.\, b : \pi \to \rho}\textsc{Abs} \qquad \frac{\Gamma \vdash f : \phi \to \alpha \qquad \Gamma \vdash x : \phi}{\Gamma \vdash f\ x : \alpha}\textsc{App}$$

$$\frac{\Gamma(t) = \tau}{\Gamma \vdash t : \tau}\textsc{Var} \qquad \frac{c \in \textsc{Const}}{\Gamma \vdash c : \tau_c}\textsc{Const}$$

- Typvariablen: τ, α, π, ρ
- Funktionstypen: $\tau_1 \to \tau_2$, rechtsassoziativ
- Typisierungsregeln sind eindeutig: Eine Regel pro Termform

Was bedeuten eigentlich $\vdash$, Γ und $:$?

$$\lambda a.\, \lambda f.\, f\ (a\ \texttt{true})$$

Um zu einem solchen Term ein Typisierungsproblem zu beschreiben, notieren wir:

$$\Gamma \vdash \lambda a.\, \lambda f.\, f\ (a\ \texttt{true}) : \tau$$

„Im Typkontext Γ hat der Term den Typen τ."

- Γ: Enthält Typen für freie Variablen.
- ... $\vdash$ ... : ... — Notation für Typisierungsproblem.

Γ in Aktion

$$\Gamma \vdash a + 42 : \texttt{int}$$

$$\textsc{Const} = \{42\}, \tau_{42} = \texttt{int}$$

Damit die Aussage „$a + 42$ hat in Γ den Typen `int`“ stimmt, müssen wir für Γ wählen:

Γ in Aktion

$$\Gamma \vdash a + 42 : \texttt{int}$$

$$\textsc{Const} = \{42\}, \tau_{42} = \texttt{int}$$

Damit die Aussage „$a + 42$ hat in Γ den Typen `int`“ stimmt, müssen wir für Γ wählen:

- $\Gamma = a : \texttt{int}, + : \texttt{int} \to \texttt{int} \to \texttt{int}$

Γ in Aktion

$$\Gamma \vdash a + 42 : \mathtt{int}$$

$$\textsc{Const} = \{42\}, \tau_{42} = \mathtt{int}$$

Damit die Aussage „$a + 42$ hat in Γ den Typen $\mathtt{int}$“ stimmt, müssen wir für Γ wählen:

- $\Gamma = a : \mathtt{int}, + : \mathtt{int} \to \mathtt{int} \to \mathtt{int}$
- Allgemeiner: $\Gamma = a : \alpha, + : \alpha \to \mathtt{int} \to \mathtt{int}$

Typisierungsregel für Lambdas

- „Unter Einfügung des Typs π von p in den Kontext…“
- „… ist b als Funktion von p typisierbar.“

$$\frac{\boxed{\Gamma, p : \pi} \qquad \boxed{\vdash b : \rho}}{\boxed{\Gamma \vdash \lambda p.\, b : \pi \to \rho}}\ \textsc{Abs}$$

- Daraus folgt:
- „$\lambda p.\, b$ ist eine Funktion, die πs auf ρs abbildet“

Typisierungsregel für Funktionsanwendungen

- „f ist im Kontext Γ eine Funktion, die ϕs auf αs abbildet.“
- „x ist im Kontext Γ ein Term des Typs ϕ.“

$$\frac{\boxed{\Gamma \vdash f : \phi \to \alpha} \qquad \boxed{\Gamma \vdash x : \phi}}{\boxed{\Gamma \vdash f\ x : \alpha}}\ \textsc{App}$$

- Daraus folgt:
- „x eingesetzt in f ergibt einen Term des Typs α.“

Einfache Typisierungsregel für Variablen

- „Der Typkontext Γ enthält einen Typ τ für t."

$$\frac{\boxed{\Gamma(t) = \tau}}{\boxed{\Gamma \vdash t : \tau}}\ \textsc{Var}$$

- Daraus folgt:
- „Variable t hat im Kontext Γ den Typ τ."

Typisierung: Beispiel

$$x : \texttt{bool} \vdash \lambda f.\, f\ x : (\texttt{bool} \to \alpha) \to \alpha$$

„Unter der Annahme, dass x den Typ `bool` hat, hat $\lambda f.\, f\ x$ den Typ $(\texttt{bool} \to \alpha) \to \alpha$.“

Typisierung: Beispiel

$$\frac{x : \texttt{bool}, f : \texttt{bool} \to \alpha \vdash f\ x : \alpha}{x : \texttt{bool} \vdash \lambda f.\ f\ x : (\texttt{bool} \to \alpha) \to \alpha}\textsc{Abs}$$

Pattern-Matching: Der äußerste Term ist ein Lambda, also wenden wir die $\textsc{Abs}$-Regel an.

$$\Gamma = x : \texttt{bool}$$

$$p = f, b = f\ x$$

$$\pi = \texttt{bool} \to \alpha$$

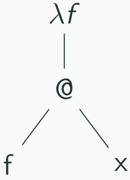
$$\rho = \alpha$$

$$\frac{\Gamma, p : \pi \vdash b : \rho}{\Gamma \vdash \lambda p.\ b : \pi \to \rho}\textsc{Abs}$$

Typisierung: Beispiel

$$\cfrac{\cfrac{\cfrac{\Gamma(f) = \texttt{bool} \to \alpha}{\Gamma \vdash f : \texttt{bool} \to \alpha}\textsc{Var} \qquad \cfrac{\Gamma(x) = \texttt{bool}}{\Gamma \vdash x : \texttt{bool}}\textsc{Var}}{x : \texttt{bool}, f : \texttt{bool} \to \alpha \vdash f\ x : \alpha}\textsc{App}}{x : \texttt{bool} \vdash \lambda f.\, f\ x : (\texttt{bool} \to \alpha) \to \alpha}\textsc{Abs}$$

$$\Gamma = x : \texttt{bool}, f : \texttt{bool} \to \alpha$$



$\lambda f.\, f\ x$

Von Typisierungsregeln zu Typinferenz

Beim inferieren wird das Pattern-matching der Typen durch die Unifikation übernommen. Deswegen schreiben wir anstelle von konkreten Typen immer α_i und merken uns die Gleichungen für später:

$$\frac{\Gamma, p : \pi \vdash b : \rho}{\Gamma \vdash \lambda p.\, b : \pi \to \rho}\textsc{Abs} \quad \rightsquigarrow \quad \begin{array}{c} \dfrac{\Gamma, p : \alpha_j \vdash b : \alpha_k}{\Gamma \vdash \lambda p.\, b : \alpha_i}\textsc{Abs} \\ \{\alpha_i = \alpha_j \to \alpha_k\} \end{array}$$

Von Typisierungsregeln zu Typinferenz

Beim inferieren wird das Pattern-matching der Typen durch die Unifikation übernommen. Deswegen schreiben wir anstelle von konkreten Typen immer α_i und merken uns die Gleichungen für später:

$$\frac{\Gamma \vdash f : \phi \to \alpha \qquad \Gamma \vdash x : \phi}{\Gamma \vdash f\ x : \alpha}\textsc{App} \quad \rightsquigarrow \quad \frac{\Gamma \vdash f : \alpha_j \qquad \Gamma \vdash x : \alpha_k}{\begin{array}{c}\Gamma \vdash f\ x : \alpha_i \\ \{\alpha_j = \alpha_k \to \alpha_i\}\end{array}}\textsc{App}$$

Von Typisierungsregeln zu Typinferenz

Beim inferieren wird das Pattern-matching der Typen durch die Unifikation übernommen. Deswegen schreiben wir anstelle von konkreten Typen immer α_i und merken uns die Gleichungen für später:

$$\frac{\Gamma(t) = \tau}{\Gamma \vdash t : \tau}\textsc{Var} \quad \rightsquigarrow \quad \begin{array}{c}\dfrac{\Gamma(t) = \alpha_j}{\Gamma \vdash t : \alpha_i}\textsc{Var} \\ \{\alpha_i = \alpha_j\}\end{array}$$

Algorithmus zur Typinferenz

- Stelle Typherleitungsbaum auf
  - In jedem Schritt werden neue Typvariablen α_i angelegt
  - Statt die Typen direkt im Baum einzutragen, werden Gleichungen in einem Constraint-System eingetragen
- Unifiziere Constraint-System zu einem Unifikator
  - Robinson-Algorithmus, im Grunde wie bei Prolog
  - I.d.R.: Allgemeinster Unifikator (mgu)

$$\frac{\Gamma(t) = \alpha_j}{\Gamma \vdash t : \alpha_i}\textsc{Var} \qquad \frac{\Gamma \vdash f : \alpha_j \qquad \Gamma \vdash x : \alpha_k}{\Gamma \vdash f\ x : \alpha_i}\textsc{App} \qquad \frac{\Gamma, p : \alpha_j \vdash b : \alpha_k}{\Gamma \vdash \lambda p.\, b : \alpha_i}\textsc{Abs}$$

Constraint:
$\{\alpha_i = \alpha_j\}$

Constraint:
$\{\alpha_j = \alpha_k \to \alpha_i\}$

Constraint:
$\{\alpha_i = \alpha_j \to \alpha_k\}$

Herleitungsbaum: Beispiel

$$\vdash \lambda x.\, \lambda y.\, x : \alpha_1$$

Beispielhafte Aufgabenstellung: Finde den Typen α_1.

Herleitungsbaum: Beispiel

$$\frac{\underline{x} : \alpha_2 \vdash \underline{\lambda y.\, x} : \alpha_3}{\vdash \lambda \underline{x}.\, \underline{\lambda y.\, x} : \alpha_1}\textsc{Abs}$$

Typgleichungen:

$$C = \{\underline{\alpha_1 = \alpha_2 \to \alpha_3}\}$$

Herleitungsbaum: Beispiel

$$\dfrac{\dfrac{\underline{x : \alpha_2}, \underline{y} : \alpha_4 \vdash \underline{x} : \alpha_5}{\underline{x : \alpha_2} \vdash \lambda \underline{y}. \underline{x} : \alpha_3}\textsc{Abs}}{\vdash \lambda x. \lambda y. x : \alpha_1}\textsc{Abs}$$

Typgleichungen:

$$\begin{aligned} C = \{&\alpha_1 = \alpha_2 \to \alpha_3 \\ &,\underline{\alpha_3 = \alpha_4 \to \alpha_5}\} \end{aligned}$$

Herleitungsbaum: Beispiel

$$\dfrac{\dfrac{\dfrac{\underline{x : \alpha_2, y : \alpha_4}(\underline{x}) = \alpha_2}{\underline{x : \alpha_2, y : \alpha_4} \vdash \underline{x} : \alpha_5}\,\textsc{Var}}{x : \alpha_2 \vdash \lambda y.\, x : \alpha_3}\,\textsc{Abs}}{\vdash \lambda x.\, \lambda y.\, x : \alpha_1}\,\textsc{Abs}$$

Typgleichungen:

$$\begin{aligned} C = \{&\alpha_1 = \alpha_2 \to \alpha_3 \\ ,&\alpha_3 = \alpha_4 \to \alpha_5 \\ ,&\underline{\alpha_5 = \alpha_2}\} \end{aligned}$$

Herleitungsbaum: Aufgabe

$$\frac{\dots}{\vdash \lambda f.\, f\ (\lambda x.\, x) : \alpha_1}\textsc{Abs}$$

Findet den Typen α_1. Teilpunkte gibt es für:

- Herleitungsbaum,
- Typgleichungsmenge C,
- Unifikation per Robinsonalgorithmus.

Herleitungsbaum: Aufgabe

$$
\dfrac{
  \dfrac{(f : \alpha_2)(f) = \alpha_2}{f : \alpha_2 \vdash f : \alpha_4}\,\textsc{Var}
  \qquad
  \dfrac{\dfrac{\Gamma(x) = \alpha_6}{\Gamma \vdash x : \alpha_7}\,\textsc{Var}}{f : \alpha_2 \vdash \lambda x.\, x : \alpha_5}\,\textsc{Abs}
}{
  \dfrac{f : \alpha_2 \vdash f\ (\lambda x.\, x) : \alpha_3}{\vdash \lambda f.\, f\ (\lambda x.\, x) : \alpha_1}\,\textsc{Abs}
}\,\textsc{App}
$$

$$\Gamma = f : \alpha_2, x : \alpha_6$$

$$
\begin{aligned}
C = \{&\alpha_1 = \alpha_2 \to \alpha_3, \alpha_4 = \alpha_5 \to \alpha_3, \\
&\alpha_2 = \alpha_4, \\
&\alpha_5 = \alpha_6 \to \alpha_7, \alpha_6 = \alpha_7\}
\end{aligned}
$$

Let-Polymorphismus

Let-Polymorphismus: Motivation

$$\lambda f.\, f\ f$$

- Diese Funktion verwendet f auf zwei Arten:
  - $\alpha \to \alpha$: Rechte Seite.
  - $(\alpha \to \alpha) \to (\alpha \to \alpha)$: Linke Seite, nimmt f als Argument und gibt es zurück.

Let-Polymorphismus: Motivation

$$\lambda f.\, f\ f$$

- Diese Funktion verwendet f auf zwei Arten:
  - $\alpha \rightarrow \alpha$: Rechte Seite.
  - $(\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)$: Linke Seite, nimmt f als Argument und gibt es zurück.
- Problem: $\alpha \rightarrow \alpha$ und $(\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)$ sind nicht unifizierbar!
  - „occurs check“: α darf sich nicht selbst einsetzen.
- Idee: Bei jeder Verwendung eines polymorphen Typen erzeugen wir <u>neue Typvariablen</u>, um diese Beschränkung zu umgehen.

Typschemata und Instanziierung

- Idee: Bei jeder Verwendung eines polymorphen Typen erzeugen wir neue Typvariablen, um diese Beschränkung zu umgehen.
- Ein Typschema ist ein Typ, in dem manche Typvariablen allquantifiziert sind:

$$\phi = \forall\alpha_1. \dots \forall\alpha_n.\tau$$
$$\alpha_i \in FV(\tau)$$

- Typschemata kommen bei uns immer nur in Kontexten vor!
- Beispiele:
  - $\forall\alpha.\alpha \to \alpha$
  - $\forall\alpha.\alpha \to \beta \to \alpha$

Typschemata und Instanziierung

- Ein Typschema spannt eine Menge von Typen auf, mit denen es instanziiert werden kann:

$$\forall\alpha.\alpha \to \alpha \succeq \mathsf{int} \to \mathsf{int}$$

$$\forall\alpha.\alpha \to \alpha \succeq \tau \to \tau$$

$$\forall\alpha.\alpha \to \alpha \not\succeq \tau \to \sigma$$

$$\forall\alpha.\alpha \to \alpha \not\succeq \tau \to \tau \to \tau$$

$$\forall\alpha.\alpha \to \alpha \succeq (\tau \to \tau) \to (\tau \to \tau)$$

Let-Polymorphismus

Um Typschemata bei der Inferenz zu verwenden, müssen wir zunächst die Regel für Variablen anpassen:

$$\frac{\Gamma(x) = \phi \qquad \phi \succeq_{\text{frische } \alpha_i} \tau}{\Gamma \vdash x : \alpha_j}\textsc{Var}$$

Constraint: $\{\alpha_j = \tau\}$

- $\succeq_{\text{frische } \alpha_i}$ instanziiert ein Typschema mit α_i, die noch nicht im Baum vorkommen.
- Jetzt brauchen wir noch eine Möglichkeit, Typschemata zu erzeugen.

Let-Polymorphismus

Mit einen Let-Term wird ein Typschema eingeführt:

$$\frac{\Gamma \vdash t_1 : \alpha_i \qquad \Gamma' \vdash t_2 : \alpha_j}{\Gamma \vdash \texttt{let}\ x = t_1\ \texttt{in}\ t_2 : \alpha_k}\textsc{Let}$$

$$\begin{aligned} \sigma_{let} &= mgu(C_{let}) \\ \Gamma' &= \sigma_{let}(\Gamma), x : ta(\sigma_{let}(\alpha_i), \sigma_{let}(\Gamma)) \\ C'_{let} &= \{\alpha_n = \sigma_{let}(\alpha_n) \mid \sigma_{let}(\alpha_n)\ \text{ist definiert}\} \end{aligned}$$

Constraints: $C'_{let} \cup C_{body} \cup \{a_j = a_k\}$

Beispiel: Let-Polymorphismus

$$\dfrac{\dfrac{\dots}{\vdash \lambda x.\, x : \alpha_2}\,\textsc{Abs} \qquad \dfrac{\dfrac{\begin{array}{c}\Gamma'(f) = \forall \alpha_5.\alpha_5 \to \alpha_5 \\ \succeq \alpha_8 \to \alpha_8\end{array}}{\Gamma' \vdash f : \alpha_6}\,\textsc{Var} \qquad \dfrac{\begin{array}{c}\Gamma'(f) = \forall \alpha_5.\alpha_5 \to \alpha_5 \\ \succeq \alpha_9 \to \alpha_9\end{array}}{\Gamma' \vdash f : \alpha_7}\,\textsc{Var}}{\Gamma' \vdash f\ f : \alpha_3}\,\textsc{App}}{\vdash \texttt{let}\ f = \lambda x.\, x\ \texttt{in}\ f\ f : \alpha_1}\,\textsc{Let}$$

$$\begin{aligned}
C_{let} &= \{\alpha_2 = \alpha_4 \to \alpha_5, \alpha_4 \to \alpha_5\} \\
\sigma_{let} &= [\alpha_2 \Rightarrow \alpha_5 \to \alpha_5, \alpha_4 \Rightarrow \alpha_5] \\
\Gamma' &= x : \forall \alpha_5.\alpha_5 \to \alpha_5 \\
C'_{let} &= \{\alpha_2 = \alpha_5 \to \alpha_5, \alpha_4 = \alpha_5\} \\
C_{body} &= \{\alpha_6 = \alpha_7 \to \alpha_3, \alpha_6 = \alpha_8 \to \alpha_8, \alpha_7 = \alpha_9 \to \alpha_9\} \\
C &= C'_{let} \cup C_{body} \cup \{\alpha_3 = \alpha_1\}
\end{aligned}$$

Einführung in Compilerbau

Compiler in ProPa

- Ein bisschen...
  - Lexikalische Analyse (Tokenisierung)
  - Syntaktische Analyse (Parsen)
  - Semantische Analyse (Optimierung)
  - Codegenerierung

Compiler in ProPa

- Ein bisschen...
  - Lexikalische Analyse (Tokenisierung)
  - Syntaktische Analyse (Parsen)
  - Semantische Analyse (Optimierung)
  - Codegenerierung
- Klausur:
  - SLL(k)-Form beweisen
  - Rekursiven Abstiegsparser schreiben/vervollständigen
  - First/Follow-Mengen berechnen
  - Java-Bytecode

Syntaktische Analyse (18WS)

$$\begin{aligned} \mathit{SGML} &\rightarrow \texttt{< id >}\ \mathit{Children}\ \texttt{< / >} \\ \mathit{Children} &\rightarrow \epsilon \mid \mathit{SGML}\ \ \mathit{Children} \end{aligned}$$

$$\{\texttt{<id><id></><id></></>},\ \ \texttt{<id></>},\ \ \ldots\} \in G$$

Syntaktische Analyse (18WS)

$$SGML \rightarrow \texttt{<}\ \texttt{id}\ \texttt{>}\ Children\ \texttt{<}\ \texttt{/}\ \texttt{>}$$
$$Children \rightarrow \epsilon \mid SGML\ \ Children$$

$$\{\texttt{<id><id></><id></></>},\ \ \texttt{<id></>},\ \ ...\} \in G$$

- Begründen Sie formal, dass die obige Grammatik nicht in $\mathrm{SLL}(1)$-Form ist (3P.).

Syntaktische Analyse (18WS)

$$
\begin{aligned}
\mathit{SGML} &\rightarrow \ \texttt{< id >}\ \mathit{Children}\ \texttt{< / >} \\
\mathit{Children} &\rightarrow \ \epsilon \mid \mathit{SGML}\ \ \mathit{Children}
\end{aligned}
$$

$$\{\texttt{<id><id></><id></></>},\ \texttt{<id></>},\ \ldots\} \in G$$

- Begründen Sie formal, dass die obige Grammatik nicht in $\mathrm{SLL}(1)$-Form ist (3P.).
- Entwickeln Sie für [eine linksfaktorisierte Version der obigen Grammatik] einen rekursiven Abstiegsparser (16P.).

Java-Bytecode (16SS)

Übersetzen Sie folgenden Java-Programmausschnitt in Java-Bytecode (10P.):

```
if (((a < b) || !((a < c) || (c < b))) && !(c < 0)) {
    c = b + a;
}
```

Java-Bytecode (16SS)

Übersetzen Sie folgenden Java-Programmausschnitt in Java-Bytecode (10P.):

```
if (((a < b) || !((a < c) || (c < b))) && !(c < 0)) {
    c = b + a;
}
```

Hinweise:

- Codeerzeugung für bedingte Sprünge: Folien 447ff.
- Um eine Bedingung der Form `!cond` zu übersetzen, reicht es, `cond` zu übersetzen und die Sprungziele anzupassen.

Compiler

Compiler: Motivation

- Maschine(-nmodell) versteht i.d.R. eingeschränkten Instruktionssatz
  - Es gibt/gab zwar auch mal CISC-Maschinen, heute ist sind aber RISC(-ähnliche) Prozessoren am weitesten verbreitet
  - Gründe: RISC-Prozessoren sind wesentlich einfacher (= billiger) zu bauen
- Programme in Maschinensprache sind i.d.R. für Menschen nicht einfach zu Schreiben.

Compiler: Motivation

- Maschine(-nmodell) versteht i.d.R. eingeschränkten Instruktionssatz
  - Es gibt/gab zwar auch mal CISC-Maschinen, heute ist sind aber RISC(-ähnliche) Prozessoren am weitesten verbreitet
  - Gründe: RISC-Prozessoren sind wesentlich einfacher (= billiger) zu bauen
- Programme in Maschinensprache sind i.d.R. für Menschen nicht einfach zu Schreiben.
- Also: Erfinde einfacher zu Schreibende ($\approx$ mächtigere) Sprache, die dann in die Sprache der Maschine übersetzt wird.
- Diesen Übersetzungsschritt sollte optimalerweise ein Programm erledigen, da wir sonst auch einfach direkt Maschinensprache-Programme schreiben können.

Compiler

- Übersetzer für formale Sprachen nennt man Compiler
- Beispiele:
  - C, Haskell, Rust, Go $\rightarrow$ X86
  - Java, Clojure, Kotlin $\rightarrow$ Java-Bytecode
  - TypeScript $\rightarrow$ JavaScript
  - Python $\rightarrow$ Python-AST

Compiler

- Übersetzer für formale Sprachen nennt man Compiler
- Beispiele:
  - C, Haskell, Rust, Go $\rightarrow$ X86
  - Java, Clojure, Kotlin $\rightarrow$ Java-Bytecode
  - TypeScript $\rightarrow$ JavaScript
  - Python $\rightarrow$ Python-AST
- Interpreter kann man auch als Compiler kategorisieren, sie zählen aber i.A. nicht dazu

Compiler

- Übersetzer für formale Sprachen nennt man Compiler
- Beispiele:
  - C, Haskell, Rust, Go $\rightarrow$ X86
  - Java, Clojure, Kotlin $\rightarrow$ Java-Bytecode
  - TypeScript $\rightarrow$ JavaScript
  - Python $\rightarrow$ Python-AST
- Interpreter kann man auch als Compiler kategorisieren, sie zählen aber i.A. nicht dazu
- Single-pass vs. Multi-pass
  - Single-pass: Eingabe wird einmal gelesen, Ausgabe währenddessen erzeugt (ältere Compiler)
  - Multi-pass: Eingabe wird in Zwischenschritten in verschiedene Repräsentationen umgewandelt
    - Quellsprache, Tokens, AST, Zwischensprache, Zielsprache

Lexikalische Analyse

```
int x1 = 123;
print("123");
```

```
int, id[x1], assign,
intlit[123], semi,
id[print], lp,
stringlit["123"], ...
```

- Lexikalische Analyse (Tokenisierung) verarbeitet eine Zeichensequenz in eine Liste von Tokens.
- Tokens sind Zeichengruppen, denen eine Semantik innewohnt:
  - `int` — Typ einer Ganzzahl
  - = — Zuweisungsoperator
  - `x1` — Variablen- oder Methodenname
  - `123` — Literal einer Ganzzahl
  - `"123"` — String-Literal
  - etc.
- Lösbar mit regulären Ausdrücken, Automaten

Syntaktische Analyse

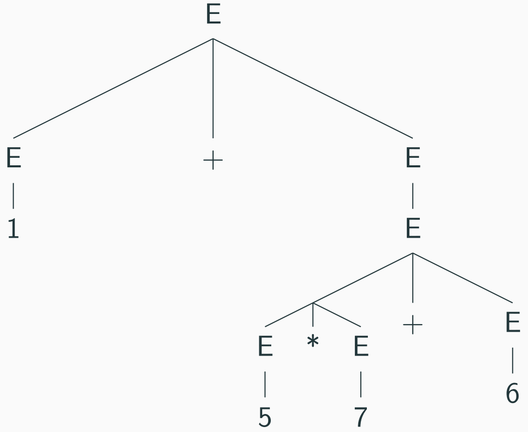
- Syntaktische Analyse stellt die unterliegende Struktur der bisher linear gelesenen Eingabe fest:
  - Blockstruktur von Programmen
  - Baumstruktur von HTML-Dateien
  - Header, Inhalt-Struktur von Mails
  - Verschachtelte mathematische Ausdrücke
- Syntaktische Analyse ist das größte Compiler-Thema in PP.

Syntaktische Analyse

- Syntaktische Analyse stellt die unterliegende Struktur der bisher linear gelesenen Eingabe fest:
  - Blockstruktur von Programmen
  - Baumstruktur von HTML-Dateien
  - Header, Inhalt-Struktur von Mails
  - Verschachtelte mathematische Ausdrücke
- Syntaktische Analyse ist das größte Compiler-Thema in PP.
- Übliche Vorgehensweise (in PP):
  - Grammatik G erfinden
  - ggf. G in einfache Form G' bringen
  - rekursiven Abstiegsparser für G' implementieren
- Alternativ: Parser-Kombinatoren, Yacc, etc.

Beispiel: Mathematische Ausdrücke

$1 + 5 * 7 + 6$



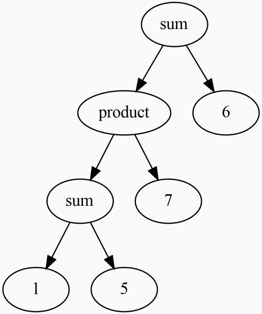
- Zu beachten: Punkt-vor-Strich (Präzedenz), Klammerung, etc.
- Nicht mehr mit regulären Ausdrücken lösbar
- „Offensichtliche“ Grammatik oft nicht einfach zu Parsen

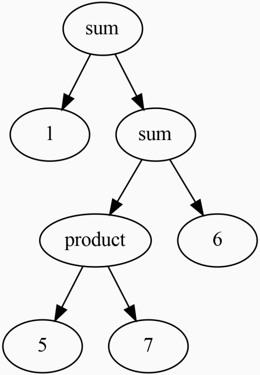
Beispiel: Mathematische Ausdrücke

$$E \to \ \texttt{num} \mid \texttt{(}\ E\ \texttt{)} \mid E\ \texttt{+}\ E \mid E\ \texttt{*}\ E$$

Beispiel: Mathematische Ausdrücke

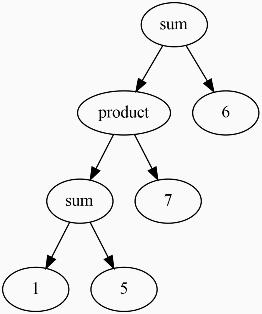
$$E \rightarrow \texttt{num} \mid \texttt{(}\ E\ \texttt{)} \mid E\ \texttt{+}\ E \mid E\ \texttt{*}\ E$$

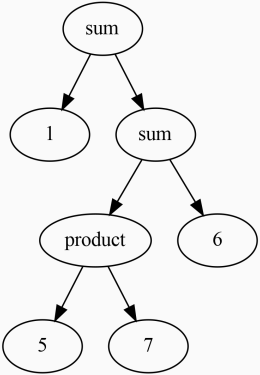




Beispiel: Mathematische Ausdrücke

$$E \rightarrow \texttt{num} \mid \texttt{(}\ E\ \texttt{)} \mid E\ \texttt{+}\ E \mid E\ \texttt{*}\ E$$





- Ableitungsbaum nicht eindeutig $\rightsquigarrow$ schlecht
- Ableitungsbaum garantiert nicht Punkt-vor-Strich $\rightsquigarrow$ schlecht

Präzedenz, Linksfaktorisierung

Wie zeichnen sich „gute“ Grammatiken aus?

Präzedenz, Linksfaktorisierung

Wie zeichnen sich „gute“ Grammatiken aus?

Operatorpräzedenz schon in Grammatik definiert:

$$E \to E \texttt{ + } T \mid E \texttt{ - } T \mid T$$

$$T \to T \texttt{ * } F \mid T \texttt{ / } F \mid F$$

$$F \to \texttt{num} \mid \texttt{(} E \texttt{)}$$

Präzedenz, Linksfaktorisierung

Wie zeichnen sich „gute“ Grammatiken aus?

Operatorpräzedenz schon in Grammatik definiert:

$$
\begin{aligned}
E &\rightarrow E \texttt{ + } T \mid E \texttt{ - } T \mid T \\
T &\rightarrow T \texttt{ * } F \mid T \texttt{ / } F \mid F \\
F &\rightarrow \texttt{num} \mid \texttt{(} E \texttt{)}
\end{aligned}
$$

Vermeidung von Linksrekursion (Linksfaktorisierung):

$$
\begin{aligned}
E &\rightarrow T \ \mathit{EList} \\
\mathit{EList} &\rightarrow \epsilon \mid \texttt{+} \ T \ \mathit{EList} \mid \texttt{-} \ T \ \mathit{EList} \\
T &\rightarrow F \ \mathit{TList} \\
\mathit{TList} &\rightarrow \epsilon \mid \texttt{*} \ F \ \mathit{TList} \mid \texttt{/} \ F \ \mathit{TList} \\
F &\rightarrow \texttt{num} \mid \texttt{(} E \texttt{)}
\end{aligned}
$$

First-/Followmenge, Indizmenge

$$\mathit{EList} \rightarrow \ \epsilon \mid \texttt{+} \ \ T \ \ \mathit{EList} \mid \texttt{-} \ \ T \ \ \mathit{EList}$$

Wie können wir bspw. bei *EList* entscheiden, welche Produktion anzuwenden ist?

First-/Followmenge, Indizmenge

$$\textit{EList} \rightarrow \ \epsilon \mid \texttt{+} \ \ T \ \ \textit{EList} \mid \texttt{-} \ \ T \ \ \textit{EList}$$

Wie können wir bspw. bei *EList* entscheiden, welche Produktion anzuwenden ist?

- $\rightsquigarrow$ definiere Indizmenge $IM_k(A \rightarrow \alpha) = \mathrm{First}_k(\alpha \mathrm{Follow}_k(A))$
- Wenn nächste k Token in $IM_k(\textit{EList} \rightarrow \phi) \rightsquigarrow$ weiter mit ϕ

First-/Followmenge, Indizmenge

$$\mathit{EList} \rightarrow \ \epsilon \mid \texttt{+} \ \ T \ \ \mathit{EList} \mid \texttt{-} \ \ T \ \ \mathit{EList}$$

Wie können wir bspw. bei *EList* entscheiden, welche Produktion anzuwenden ist?

- $\rightsquigarrow$ definiere Indizmenge $IM_k(A \rightarrow \alpha) = \mathrm{First}_k(\alpha \mathrm{Follow}_k(A))$
- Wenn nächste k Token in $IM_k(\mathit{EList} \rightarrow \phi) \rightsquigarrow$ weiter mit ϕ
- $IM_1(\mathit{EList} \rightarrow \ \epsilon) = \mathrm{First}_1(\epsilon \mathrm{Follow}_1(\mathit{EList})) = \{\texttt{)}, \texttt{\#}\}$
- $IM_1(\mathit{EList} \rightarrow \ \texttt{+} \ \ T \ \ \mathit{EList}) = \mathrm{First}_1(\texttt{+} \ T \ \mathit{EList} \ \mathrm{Follow}_1(\mathit{EList})) = \{\texttt{+}\}$
- $IM_1(\mathit{EList} \rightarrow \ \texttt{-} \ \ T \ \ \mathit{EList}) = \mathrm{First}_1(\texttt{-} \ T \ \mathit{EList} \ \mathrm{Follow}_1(\mathit{EList})) = \{\texttt{-}\}$

First-/Followmenge, Indizmenge

$$\textit{EList} \rightarrow \ \epsilon \mid \texttt{+} \ \ T \ \ \textit{EList} \mid \texttt{-} \ \ T \ \ \textit{EList}$$

Wie können wir bspw. bei *EList* entscheiden, welche Produktion anzuwenden ist?

- $\leadsto$ definiere Indizmenge $IM_k(A \rightarrow \alpha) = \mathrm{First}_k(\alpha \mathrm{Follow}_k(A))$
- Wenn nächste k Token in $IM_k(\textit{EList} \rightarrow \phi) \leadsto$ weiter mit ϕ
- $IM_1(\textit{EList} \rightarrow \ \epsilon) = \mathrm{First}_1(\epsilon \mathrm{Follow}_1(\textit{EList})) = \{\texttt{)}, \texttt{\#}\}$
- $IM_1(\textit{EList} \rightarrow \ \texttt{+} \ \ T \ \ \textit{EList}) = \mathrm{First}_1(\texttt{+} \ T \ \textit{EList} \ \mathrm{Follow}_1(\textit{EList})) = \{\texttt{+}\}$
- $IM_1(\textit{EList} \rightarrow \ \texttt{-} \ \ T \ \ \textit{EList}) = \mathrm{First}_1(\texttt{-} \ T \ \textit{EList} \ \mathrm{Follow}_1(\textit{EList})) = \{\texttt{-}\}$
- $\mathrm{First}_k(A)$: Menge an möglichen ersten k Token in A
- $\mathrm{Follow}_k(A)$: Menge an möglichen ersten k Token nach A

SLL-Kriterium

Grammatik ist in $\mathrm{SLL}(k)$-Form
$:\Leftrightarrow \forall A \to \alpha, A \to \beta \in P : IM_k(A \to \alpha) \cap IM_k(A \to \beta) = \emptyset$

- $\mathrm{SLL}(k)$: Bei jedem Nichtterminal muss die zu wählende Produktion an den nächsten k Token wählbar sein.
- Nichtterminale mit nur einer Produktion sind hier irrelevant
- Schwierig daran: Follow-Mengen berechnen

SLL-Kriterium

Grammatik ist in $\mathrm{SLL}(k)$-Form
$:\Leftrightarrow \forall A \to \alpha, A \to \beta \in P : IM_k(A \to \alpha) \cap IM_k(A \to \beta) = \emptyset$

- $\mathrm{SLL}(k)$: Bei jedem Nichtterminal muss die zu wählende Produktion an den nächsten k Token wählbar sein.
- Nichtterminale mit nur einer Produktion sind hier irrelevant
- Schwierig daran: Follow-Mengen berechnen

$$
\begin{aligned}
E &\to E \texttt{ + } T \mid E \texttt{ - } T \mid T \\
T &\to T \texttt{ * } F \mid T \texttt{ / } F \mid F \\
F &\to \texttt{num} \mid \texttt{(} E \texttt{)}
\end{aligned}
$$

- Begründet formal, dass obige Grammatik nicht $\mathrm{SLL}(1)$.
- Berechnet $\mathrm{Follow}_1(N)$ für $N \in \{E, T, F\}$.

Rekursive Abstiegsparser

$$
\begin{aligned}
E &\rightarrow T \;\; EList \\
EList &\rightarrow \epsilon \mid \texttt{+} \;\; T \;\; EList \mid \texttt{-} \;\; T \;\; EList \\
T &\rightarrow F \;\; TList \\
TList &\rightarrow \epsilon \mid \texttt{*} \;\; F \;\; TList \mid \texttt{/} \;\; F \;\; TList \\
F &\rightarrow \texttt{num} \mid \texttt{(} \;\; E \;\; \texttt{)}
\end{aligned}
$$

- Yay, unsere Grammatik hat jetzt $\mathrm{SLL}(1)$-Form!
- Aber was bringt das?

Rekursive Abstiegsparser

$$
\begin{aligned}
E &\rightarrow\ T\ \ \mathit{EList} \\
\mathit{EList} &\rightarrow\ \epsilon \mid \texttt{+}\ \ T\ \ \mathit{EList} \mid \texttt{-}\ \ T\ \ \mathit{EList} \\
T &\rightarrow\ F\ \ \mathit{TList} \\
\mathit{TList} &\rightarrow\ \epsilon \mid \texttt{*}\ \ F\ \ \mathit{TList} \mid \texttt{/}\ \ F\ \ \mathit{TList} \\
F &\rightarrow\ \texttt{num} \mid \texttt{(}\ \ E\ \ \texttt{)}
\end{aligned}
$$

- Yay, unsere Grammatik hat jetzt $\mathrm{SLL}(1)$-Form!
- Aber was bringt das?
- *G* ist jetzt einfach ausprogrammierbar:
    - 1 Methode per Nichtterminal: `parseE()`, `parseEList()`, ...
    - `Token[k]`-Instanzattribut für *k* langen Lookahead
    - `expect(TokenType)`-Methode, um Token zu verarbeiten

Rekursive Abstiegsparser

$$
\begin{aligned}
E &\to \; T \;\; EList \\
EList &\to \; \epsilon \mid \texttt{+} \;\; T \;\; EList \mid \texttt{-} \;\; T \;\; EList \\
T &\to \; F \;\; TList \\
TList &\to \; \epsilon \mid \texttt{*} \;\; F \;\; TList \mid \texttt{/} \;\; F \;\; TList \\
F &\to \; \texttt{num} \mid \texttt{(} \;\; E \;\; \texttt{)}
\end{aligned}
$$

- Yay, unsere Grammatik hat jetzt $\mathrm{SLL}(1)$-Form!
- Aber was bringt das?
- G ist jetzt einfach ausprogrammierbar:
  - 1 Methode per Nichtterminal: `parseE()`, `parseEList()`, ...
  - `Token[k]`-Instanzattribut für k langen Lookahead
  - `expect(TokenType)`-Methode, um Token zu verarbeiten
- Vervollständigt `demos/java/exprparser/ExprParser.java`!

Semantische Analyse

- PP beschäftigt sich (bis auf Typinferenz) nur kurz mit semantischer Analyse
- Hier geht es um Optimierungen, Typchecks, etc.
- $\rightsquigarrow$ weiterführende (Master-)Vorlesungen am IPD

Ende

Ende

- Di, 16.02.: Letztes Tut
- Mi, 17.02.: Letzte Vorlesung
- Themen: Mehr Syntaxanalyse (Beispiele), Codeerzeugung