Compilerbau WS 2000

Dozent: Prof. Dr. K. Indermark

Eine Vorlesungsmitschrift

Thorsten Uthke
thorsten.uthke@post.rwth-aachen.de

27. Dezember 2000

Inhaltsverzeichnis

0 Einleitung

Compiler = Programm zur Übersetzung von PS-Programmen (**Quellprogramm**) in äquivalente MS-Programme (**Zielprogramm**)

PS = höhere Programmiersprache

- imperative PS: Wertzuweisungen, Kontrollstrukturen, Datenstrukturen, Programmstrukturen (Moduln, Klassen)
- deklarative PS: funktional, logisch
- nebenläufige PS: kommunizierende Prozesse, verteilte Systeme
- objektorientierte PS: Klassen, Vererbung

MS = Maschinensprache
Grundkonzept des Von-Neumann-Rechners
elementare Maschinenbefehle (vergl. Rechnerstrukturen)
RISC = reduced instruction set computer
CISC = complex instruction set computer
Parallelrechner, verteilte Systeme, Rechnernetze

Aspekte einer PS

1) **Syntax**: formaler hierarchischer Aufbau eines Programmes aus strukturellen Komponenten
2) **Semantik**: Bedeutung eines Programms, Zustandstransformation einer abstrakten Maschine
3) **Pragmatik**: benutzerfreundliche Formulierung, natürliche Sprache, Maschinenabhängigkeiten

Äquivalenz von Programmen $\widehat{=}$ semantische Gleichheit

Struktur eines Compilers
logisch unabhängige Phasen, die aber verzahnt als **Passes** (Läufe) durchgeführt werden: Analysephase & Synthesephase

$\boxed{\text{Analyse}}$: Bestimmung der syntaktischen Struktur, Fehlererkennung

- lexikalische Analyse: Erkennung von Symbolen, Trennzeichen, Kommentaren (endliche Automaten)
- syntaktische Analyse: Erkennung des hierarchischen Programmaufbaus, Ableitungsbaum (Kellerautomaten)
- semantische Analyse: Kontextabhängigkeiten, statische Semantik, Typinformation. Attributierung des Ableitungsbaums mit semantisch relevanter Information (attributierte Grammatiken)

$\boxed{\text{Synthese}}$: Erzeugung von MS-Code aus dem attributierten Ableitungsbaum

- Übersetzung in Zwischencode für eine abstrakte Maschine
(Kann fehlen, erhöht die Portabilität, vergl. Java Virtual Machine (JVM))
- Optimierung: Verbesserung von Laufzeit und Speicherbedarf
- Codegenerierung: effiziente Verwendung von Registern und MS-Befehlssatz zur Erzeugung von MS-Code

Frontend: (MS-unabhängig) Analyse & Zwischencode & MS-unabhängige Optimierung
Backend: (MS-abhängig) MS-Code-Generierung & Optimierung

One-Pass-Compiler, n-Pass-Compiler: n = Anzahl der Läufe durch das Quellprogramm

Ziele:

- Wirkungsweise und Konstruktion von Compilern
- Vertieftes Verständnis von PS und ihrem Rechnerbezug
- Compiler als gut verstandenes Beispiel eines SW-Systems mit Interaktion von SW-Bausteinen
- Erkennung und Transformation strukturierter Objekte

1 Lexikalische Analyse

Ausgangspunkt: Quellprogramm P als Zeichenfolge
Σ_0 **Zeichensatz**, $a \in \Sigma_0$ **Zeichen** bzw. **lexikalisches Atom**, $P \in \Sigma_0$.
P besitzt aufgrund der Pragmatik von PS eine **lexikalische Struktur**.
Pragmatischer Aspekt: Benutzerfreundlichkeit von PS

- natürliche Sprache für Bezeichner, Schlüsselworte, ...
- mathematische Formelsprache für Zahlen, Formeln. Indices linearisieren, z.B. $y^2 \rightsquigarrow y**2$
- Leerzeichen, Zeilenwechsel, Einrückung, ...
- Kommentare

$\rightsquigarrow$ gute Lesbarkeit und Wartbarkeit

Semantik von P und Übersetzung von P ist syntaxorientiert, sie folgt dem hierarchischen Programmaufbau, die pragmatischen Aspekte sind dabei unerheblich.

1. Beobachtung:
Syntaktische Atome (**Symbole**) werden als Folgen von lexikalischen Atomen, sogenannten **Lexemen**, dargestellt.
1. Aufgabe der lexikalischen Analyse: Zerlegung des Quellprogramms P in eine Folge von Lexemen.

2. Beobachtung:
Für die syntaktische Analyse ist der Unterschid von Lexemen oft irrelevant, z.B. müssen Bezeichner nicht unterschieden werden. Lexeme werden oft zu **Symbolklassen** zusammengefaßt.

Darstellung einer Symbolklasse: **Token**.
Syntaktische Analyse bearbeitet eine Tokenfolge. Identifizierung eines Symbols durch zusätzliches **Attribut** für die semantische Analyse und Codegenerierung.

Symbol = (Token, Attribut).

2. Aufgabe: Transformation einer Lexem-Folge in eine Symbol-Folge.

Lexikalische Analyse
Zerlegung eines Quellprogramms in eine Folge von Lexemen und deren Transformation in eine Folge von Symbolen.

Scanner: Programm für die lexikalische Analyse
Beachte: Leerzeichen werden gelöscht. Sie spielen eine Rolle bei der Lexem-Gliederung. Kommentare werden gelöscht.

Wichtigste Symbolklassen (eigentlich Lexem-Klassen)

- Bezeichner
- Zahlwörter
- Schlüsselwörter: Bezeichner mit vorgegebener Bedeutung
- Einfache Symbole: ein Sonderzeichen, z.B. `+`,`-`,`*`,`(`,...
- Zusammengesetzte Symbole: Folgen von 2 oder mehr Sonderzeichen, z.B. `:=`,`**`,`<=`,...
- Leerzeichen: $\sqcup, \sqcup \ldots \sqcup$
- Spezielle Symbole: Kommentare, Pragmas (Compiler-Optionen)

Token: id, const, divsym, getsym, ...
Attribute: Zeiger in die Symboltabelle, Binärdarstellung einer Zahl, leer bei Symbolklassen mit einem Symbol

Feststellung: Symbolklassen und reguläre Mengen $\rightsquigarrow$ Beschreibung durch reguläre Ausdrücke
Erkennung durch endliche Automaten, automatische Scanner-Generierung, z.B. Lex.

Scanner-Konstruktion

Definition 1.1 (Reguläre Ausdrücke)
Für ein Alphabet Σ ist die Menge $RA(\Sigma)$ der **regulären Ausdrücke über** Σ definiert durch

- $\Lambda \in RA(\Sigma),\ \Sigma \subseteq RA(\Sigma)$
- $\alpha, \beta \in RA(\Sigma) \curvearrowright (\alpha \vee \beta), (\alpha \cdot \beta), (\alpha^*) \in RA(\Sigma)$

Semantik: $[\![]\!] : RA(\Sigma) \to \mathcal{P}(\Sigma^*)$

- $[\![\Lambda]\!] := \emptyset, \quad [\![a]\!] := \{a\}$
- $[\![\alpha \vee \beta]\!] := [\![\alpha]\!] \cup [\![\beta]\!]$
- $[\![\alpha \cdot \beta]\!] := [\![\alpha]\!] \cdot [\![\beta]\!]$
- $[\![\alpha^*]\!] := [\![\alpha]\!]^* \qquad (L^* = \bigcup_{n=0}^{\infty} L^n)$

Spracherweiterung von $RA(\Sigma)$
zur einfacheren Beschreibung von Symbolklassen (regulären Sprachen)

a) Vereinfachende Bezeichnungen:

- **Präzedenzregeln** zur Vermeidung von Klammern
  $*$ bindet stärker als $\cdot$, $\cdot$ stärker als $\vee$
  $\cdot$ und $\vee$ sind linksassioziativ
  $\cdot$ wird weggelassen, $\vee$ als $|$ geschrieben
  Beispiel: $a|b^*c$ steht für $(a \vee ((b^*) \cdot c))$
- **Abkürzungen**
  $\alpha^+ := \alpha\alpha^*$
  $\alpha^? := \alpha|\Lambda^*$
  $[abc] := a|b|c$
  $[a-z] := a|b|\ldots|z$

b) **Reguläre Definitionen**
Schrittweise Beschreibung von Symbolklassen durch zusätzliche frei gewählte Bezeichner (Meta-Bezeichner)

$$\begin{array}{ll} \underline{id}_1 = \alpha_1 & \text{mit } \underline{id}_1, \ldots, \underline{id}_n \notin \Sigma \\ \quad\vdots & \text{und } \alpha_i \in RA(\Sigma \,\dot\cup \{\underline{id}_1, \ldots, \underline{id}_{n-1}\}) \\ \underline{id}_n = \alpha_n & \end{array}$$

Beachte: Keine Rekursion, Entschachtelung ist möglich, andernfalls EBNF (CFG).

<u>Das einfache Matching-Problem</u>
Entscheide für $\alpha \in RA(\Sigma)$ und $w \in \Sigma^*$, ob $w \in [\![\alpha]\!]$ oder nicht.
Hilfsmittel: Endliche Automaten
$\mathfrak{A} =< Q, \Sigma, \delta, q_0, F >\in NFA(\Sigma)$, wenn $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to \mathcal{P}(Q)$.
Für $T \subseteq Q$ ist die ε**-Hülle** $\varepsilon(T)$ definiert durch

- $T \subseteq \varepsilon(T)$
- $q \in \varepsilon(T) \curvearrowright \delta(q, \varepsilon) \subseteq \varepsilon(T)$

Die <u>erweiterte</u> Transitionsfunktion $\overline{\delta} : \mathcal{P}(Q) \times \Sigma^* \to \mathcal{P}(Q)$ ist definiert durch

- $\overline{\delta}(T, \varepsilon) := \varepsilon(T)$
- $\overline{\delta}(T, wa) := \varepsilon(\bigcup_{\delta \in \overline{\delta}(T,w)} \delta(q, a))$

$\mathfrak{A}$ erkennt die Sprache $L(\mathfrak{A}) := \{w \in \Sigma^* | \overline{\delta}(\{q_0\}, w) \cap F \neq \emptyset\}$
$\mathfrak{A} \in DFA(\Sigma)$, wenn gilt
  $|\delta(q, a)| = 1$ für alle $q \in Q,\ a \in \Sigma$
  und $|\delta(q, \varepsilon)| = 0$ für alle $q \in Q$
Also: $\delta : Q \times \Sigma \to Q$

<u>DFA-Methode</u>
$\alpha \in RA(\Sigma) \overset{(1)}{\mapsto} \mathfrak{A}(\alpha) \in NFA(\Sigma) \overset{(2)}{\mapsto} \mathfrak{A}(\alpha)^P \in DFA(\Sigma)$

1) **Methode von Thompson** (Beweis des Satzes von Kleene) (1968)
Korrektheit: folgende Eigenschaften
Jeder Automat hat genau einen Anfangszustand q_0 und genau einen Endzustand q_f mit $q_0 \neq q_f$, q_0 Quelle, q_f Senke.
Jeder Zustand hat höchstens zwei Folgezustände, einen a-Nachfolger oder höchstens 2 ε-Nachfolger.
Komplexität: linearer Platz- und Zeitbedarf
   - *Platz:* $\mathfrak{A}(\alpha)$ hat höchstens $2|\alpha|$ Zustände
   $|\alpha|$ = Anzahl der Grundzeichen von $\Sigma \cup \{\Lambda\}$ und der Op. $\vee, \cdot, *$
   - *Zeit:* Syntaxanalyse von α in $\mathcal{O}(|\alpha|)$ Schritten, z.B. Transformation von α in Postfix-Notation (ohne Klammern) & schrittweiser Aufbau von $\mathfrak{A}(\alpha)$ in $\mathcal{O}(|\alpha|)$ Schritten

2) $\mathfrak{A}(\alpha) \in NFA(\Sigma) \mapsto \mathfrak{A}(\alpha)^P \in DFA(\Sigma)$
<u>Konstruktion der ε-Hülle</u>
Sei $\mathfrak{A}(\alpha) \in NFA(\Sigma)$ mit Zustandsmenge Q und $T \subseteq Q$.
Aufgabe: Berechne $\varepsilon(T) \subseteq Q$
Idee: Ein Stack speichert Zustände, welche auf ε-Nachfolger zu testen sind.
Komplexität: Idee der Darstellung von $T \subseteq Q$ (als Boolsches Array)
$Q = \{1, \ldots, n\}$, $\widetilde{T} : Q \to \{0,1\}$, damit: $q \in T \iff \widetilde{T}(q) = 1$
Die Zugehörigkeit eines Zustandes q zu T ist also in $\mathcal{O}(1)$ zu prüfen.
   - *Platz:* $\mathcal{O}(n)$, weil $length(stack) \leq n$; denn ein Zustand q erscheint höchstens ein mal auf dem Stack (wegen der Überprüfung, ob q schon in $clos(T)$ enthalten ist).
   - *Zeit:* $\mathcal{O}(n)$, insbesondere weil die for-Schleife in konstanter Zeit durchlaufen wird (q hat höchstens 2 ε-Nachfolger).

<u>Potenzmengenkonstruktion</u> $\mathfrak{A}(\alpha) \in NFA(\Sigma) \mapsto \mathfrak{A}(\alpha)^P \in DFA(\Sigma)$
$\mathfrak{A}(\alpha) = < Q, \Sigma, \delta, q_0, F >$ mit $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to \mathcal{P}(Q)$
$\mathfrak{A}(\alpha)^P = < \widehat{Q}, \Sigma, \widehat{\delta}, \widehat{q_0}, \widehat{F} >$ mit

- $\widehat{Q} := \{T \subseteq Q | T = \overline{\delta}(\{q_0\}, w),\ w \in \Sigma^*\}$
- $\widehat{\delta} : \widehat{Q} \times \Sigma \to \widehat{Q}$ und $\widehat{\delta}(T, a) := \overline{\delta}(T, a)$
- $\widehat{q_0} := \varepsilon(\{q_0\})$
- $T \in \widehat{F} \iff T \cap F \neq \emptyset$

<u>Algorithmus</u>
Idee: $\widehat{Q}$ mit $\varepsilon(\{q_0\})$ initialisieren und sukkzessiv mit $\widehat{\delta}(T, a)$ erweitern; $T \in \widehat{Q}$ wird markiert, wenn seine Nachfolger in $\widehat{Q}$ liegen.

<u>Komplexität:</u>

- *Platz:* $\mathcal{O}(2^{|\alpha|})$ (jedoch nur erreichbare Zustände)
- *Zeit:* $\mathcal{O}(2^{|\alpha|})$, aber: Berechnung von $\widehat{\delta}(T, a)$ in Linearzeit, weil die ε-Hülle Linearzeit-Berechenbar ist und ein Zustand von $\mathfrak{A}(\alpha)$ höchstens einen a-Nachfolger hat.

<u>NFA-Methode</u>
Verbesserung des Platzbedarfs bei längerer Laufzeit durch Verzicht auf die volle Potenzmengenkonstruktion, da die Eingabe $w \in \Sigma^*$ bekannt ist. Potenzmengenkonstruktion nur für den Lauf von w durch $\mathfrak{A}(\alpha)$. Direkte Berechnung von $\overline{\delta}(\{q_0\}, w)$.

<u>Algorithmus:</u>

Eingabe: $\mathfrak{A}(\alpha) \in NFA(\Sigma)$ und $w = a_1 \dots a_n \in \Sigma^*$ in der Form $\underset{\uparrow}{\boxed{}}\boxed{a_1}\boxed{\dots}\boxed{a_n}\boxed{eof}$

<u>Komplexität:</u>

- *Platz:* $\mathcal{O}(|\alpha| + |w|)$
  1. Stack für T
  2. Stack für $\bigcup_{q \in T} \delta(q, a)$

  Stacklänge $\leq |Q|$, Berechnung der ε-Hülle mit zusätzlichem Bitvektor der Länge $|Q|$
- *Zeit:* $\mathcal{O}(|\alpha| \cdot |w|)$

<u>Kombination von NFA- und DFA- Methode</u>
Die Zwischenergebnisse von bereits berechneten Transitionen in einem Cache-Speicher ablegen.

<u>Das erweiterte Matching-Problem</u>
Seien $\alpha_1, \dots, \alpha_n \in RA(\Sigma)$ und $w \in \Sigma^*$. Sei ferner $\triangle = \{S_1, \dots, S_n\}$ ein Alphabet von Symbolen. Wenn $w = w_1 w_2 \dots w_k$ und $w_j \in [\![\alpha_{i_j}]\!]$ für $j = 1, \dots, k$, dann heißt $(w_1, \dots, w_k)$ eine **Zerlegung von** w **bezüglich** $\alpha_1, \dots, \alpha_n$ und $v = S_{i_1} \dots S_{i_k}$ eine **Analyse von** w **bezüglich** $\alpha_1, \dots, \alpha_n$.
v repräsentiert die lexikalische Struktur von w bezüglich $\alpha_1, \dots, \alpha_n$.

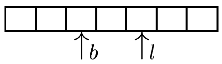
Aufgabe: Analyse bestimmen bzw. Fehler melden. Weder Zerlegung noch Analyse sind eindeutig bestimmt.

<u>Konvention für Eindeutigkeit</u>

1) **Prinzip des längsten Matches** (maximal munch)
   Eine Zerlegung $(w_1, \dots, w_k)$ von w bezüglich $\alpha_1, \dots, \alpha_n$ heißt
   lm-Zerlegung, wenn für alle $j = 1, \dots, k$, $x, y \in \Sigma^*$ und $p, q \in \{1, \dots, n\}$ gilt: $w = w_1 \dots w_j xy$ mit $w_j \in [\![\alpha_p]\!]$ und $w_j x \in [\![\alpha_q]\!] \quad \curvearrowright \quad x = \varepsilon$.
   Folgerung: Für $w, \alpha_1, \dots, \alpha_n$ gibt es höchstens eine lm-Zerlegung.

2) **Prinzip des ersten Matches** (für die Eindeutigkeit der Analyse)
Denn trotz Eindeutigkeit der lm-Zerlegung, sind im Allgemeinen mehrere zugehörige Analysen möglich, da $[\![\alpha_p]\!] \cap [\![\alpha_q]\!] \neq \emptyset$ möglich.
Konvention: Erster Match in der Reihe $\alpha_1, \ldots, \alpha_n$ zählt.
Sei $(w_1, \ldots, w_k)$ eine lm-Zerlegung und $v = S_1 \ldots S_k$ eine zugehörige Analyse von w bezüglich $\alpha_1, \ldots, \alpha_n$. Dann heißt v eine **flm-Analyse**, falls für alle $j = 1, \ldots, k$ und $i = 1, \ldots, n$ gilt: $w_j \in [\![\alpha_i]\!] \quad \curvearrowright \quad i_j \leq i$.
Folgerung: Für w und $\alpha_1, \ldots, \alpha_n$ gibt es höchstens eine flm-Analyse.

Berechnung der flm-Analyse
Vorrausetzung: $\alpha_1, \ldots, \alpha_n \in RA(\Sigma)$. O.B.d.A. $[\![\alpha_i]\!] \neq \emptyset$ und $\varepsilon \notin [\![\alpha_i]\!]$ für $i = 1, \ldots, n$, $w \in \Sigma^*$, $\triangle = \{S_1, \ldots, S_n\}$ Symbolalphabet.

1) Konstruiere für $i = 1, \ldots, n$ den Automat $\mathfrak{A}_i = \langle Q_i, \Sigma, \delta_i, q_0^{(i)}, F_i \rangle \in DFA(\Sigma)$, so daß $[\![\alpha_i]\!] = L(\mathfrak{A}_i)$.
2) Bilde aus diesen den Produktautomaten $\mathfrak{A} = \langle Q, \Sigma, \delta, q_0, F \rangle \in DFA(\Sigma)$ mit $Q := Q_1 \times \ldots \times Q_n$, $q_0 = (q_0^{(1)}, \ldots, q_0^{(n)})$,
$\delta((q^{(1)}, \ldots, q^{(n)}), a) = (\delta(q^{(1)}, a), \ldots, \delta(q^{(n)}, a))$ und
$(q^{(1)}, \ldots, q^{(n)}) \in F \iff \exists i \in \{1, \ldots, n\} : q^{(i)} \in F_i$.
Dann gilt: $L(\mathfrak{A}) = \bigcup\limits_{i=1}^{n} [\![\alpha_i]\!]$
Zerlege F wegen "first match" in $F = \bigcup\limits_{i=1}^{n} F^{(i)}$ durch die Forderung
$(q^{(1)}, \ldots, q^{(n)}) \in F^{(i)} \iff q^{(i)} \in F_i$ und $q^{(j)} \notin F_j$ für alle $1 \leq j < i$.
Dann gilt: $\overline{\delta}(q_0, w) \in F^{(i)} \iff w \in [\![\alpha_i]\!]$ und $w \notin \bigcup\limits_{j=1}^{i-1} [\![\alpha_j]\!]$
$q \in Q$ heißt **produktiv** $\iff \exists w \in \Sigma^* : \overline{\delta}(q, w) \in F$.
P : Menge der produktiven Zustände, also $F \subseteq P$.
Aufgabe: P bestimmen
3) Erweitere $\mathfrak{A}$ zu dem Backtrack-DFA $\mathcal{B}$ mit Ausgabe
Idee: Einweg-Leseband mit 2 Köpfen, ein Backtrack-Kopf b zur Markierung eines Matches und einen Lookahead-Kopf l zur Bestimmung des längsten Matches 

Konfigurationsmenge von $\mathcal{B}$: $(\{N\} \cup \triangle) \times \Sigma^* Q \Sigma^* \times \triangle^* \{\varepsilon, lexerror\}$
Anfangskonfiguration für $w \in \Sigma^*$: $(N, q_0 w, \varepsilon)$
Transitionen: (wobei $q' := \delta(q, a)$)

   a) normal mode: Match suchen

$$(N, qaw, W) \vdash \begin{cases} (N, q'w, W) & \text{, falls } q' \in P \setminus F \\ (S_i, q'w, W) & \text{, falls } q' \in F^{(i)} \\ \text{Ausgabe: } W lexxerror & \text{, falls } q' \notin P \end{cases}$$

b) backtrack mode: längsten Match suchen

$$(S, vqaw, W) \vdash \begin{cases} (S, vaq'w, W) & \text{, falls } q' \in P \setminus F \\ (S_i, q'w, W) & \text{, falls } q' \in F^{(i)} \\ (N, q_0 vaw, WS) & \text{, falls } q' \notin P \end{cases}$$

c) Eingabeende

$(N, q, W) \vdash$ Ausgabe: $Wlexxerror$,falls $q \in P \setminus F$
$(S, q, W) \vdash$ Ausgabe: WS ,falls $q \in F$
$(S, vaq, W) \vdash (N, q_0 va, WS)$,falls $q \in P \setminus F$

Dann gilt für $w \in \Sigma^*$:

$(N, q_0 w, \varepsilon) \overset{*}{\vdash}$ Ausgabe: $W \in \triangle^* \iff W$ ist flm-Analyse von w

$(N, q_0 w, \varepsilon) \overset{*}{\vdash}$ Ausgabe: $Wlexxerror \iff$ es gibt keine flm-Analyse von w

Zeitaufwand: $\mathcal{O}(|w|^2)$ im worst-case

<u>**Beispiel**</u> $\alpha_1 = abc$, $\alpha_2 = (abc) * d$ und $\Sigma = \{abcd\}$
$\implies w = (abc)^m$ erfordert $\mathcal{O}(m^2)$ Schritte

Verbesserung durch Tabular-Methode (vergleiche KMP für string pattern matching) in Linearzeit
Literatur: Th. Reps: *"Maximal munch"-Tokenization in Linear Time*, ACM-TOPLAS 20, 1998, p259-273

<u>**Beispiel**</u> Teil 1: siehe Folie 2-1 (Symbolklassen)
Teil 2: Anweisungen

```
stmt = if expr then stmt | if expr then stmt else stmt
expr = term relop term | term
term = id | num
```

<u>Annahme für die lexikalische Analyse:</u>
Die Lexeme eines `stmt` sind durch <u>`blanks`</u> getrennt.
$\implies$ besondere Rolle von <u>`blanks`</u> (White Space)

- Trennung von "eigentlichen" Lexemen, um mit "1-lookahead" auszukommen
- als Token für syntaktische Analyse überflüssig

Sieber: Programm zur Beseitigung überflüssiger Lexeme, wie Blanks, Kommentare, Pragmas, ...

<u>Attributberechnung</u>
Symbolklassen mit mehreren Lexemen erfordern zusätzlich eine Attributberechnung

- Dezimalzahl $\rightsquigarrow$ Binärzahl (`install_num`)

- Relationale Operatoren: LT,GT,EQ,...
- Bezeichner: Gleichheit von Bezeichenern relevant
  ⇒ Verwendung einer Lexemtabelle (LexTab)
  `install_id`: Eintrag eines Bezeichners in LexTab, Zeiger als Attribut

Alternative Behandlung von Schlüsselwörtern
Zunächst als Bezeichner behandeln, LexTab mit Schlüsselwörtern initialisieren, die Attributberechnung (`install_id`) liefert statt `(ID,[`*Zeiger in LexTab*`])` einfach `(IF,)`.

Automatische Scannergenerierung mit Lex (Unix-Tool)

`scan.l` $\overset{Lex}{\rightsquigarrow}$ `lex.yy.c` $\overset{C-Compiler}{\rightsquigarrow}$ `a.out`
(LexSpec) (Scanner als C-Programm) (ausführbarer Scanner)

Programm $\overset{a.out}{\rightsquigarrow}$ Tokenfolge

Aufbau einer Lex-Spezifikation

```
Definitionen
%%
Regeln
%%                  } optional
C-Hilfsprozeduren   }
```

Definitionen

- Direkter C-Code
- Substitutionen (reguläre Def.): `Name regexp`
- Startzustände (hier nicht vorgestellt)

Regeln: `muster {aktion}`
`muster` hat folgenden Aufbau: $\texttt{regexp}_1$ $\texttt{[/regexp}_2\texttt{]}$
Eine Zeichenkette paßt zu `muster`, falls sie $\texttt{regexp}_1$ matcht und ein lookahead $\texttt{regexp}_2$ matcht.
`aktion`: C-Code zur Berechnung von Token und Attribut

Finden der passenden Regel

1) longest match
2) first match
3) keine passende Regel: Zeichen wird ausgegeben, kann mit
   `regexp: .|\n` vermieden werden (`regexp` matcht alles)

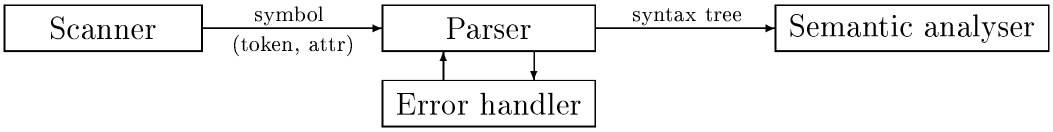
Token werden als Int-Werte kodiert
Attribute werden über globale Variablen weitergeleitet: vordef. Variable `yylval`
Fileends: Anfang einer benutzerdefinierten Prozedur
  `int yywrap(void)`
Rückgabe 1 ⇝ Lex liefert 0 zurück (kein Token/Ende der Tokensequenz)

2 Syntaktische Analyse

Aufgabe: Zerlegung der Symbolfolge, die der Scanner ausgibt, in syntaktische Einheiten, bzw. Behandlung syntaktischer Fehler
Syntaktische Einheiten: Variablen, Ausdrücke, Anweisungen,...
Beachte: Schachtelung syntaktischer Einheiten, Baumstruktur im Unterschied zur linearen Symbolfolge

Parser: Programm für die syntaktische Analyse

Schnittstelle



Beschreibung der syntaktischen Struktur: kontextfreie Grammatiken
Erkennung und Analyse: Kellerautomaten mit Ausgabe
Problem: deterministische Simulation

Allgemeiner Fall: beliebige CFG
Tabularverfahren von CYK (Cocke, Younger & Kasami) mit $\mathcal{O}(n^3)$-Zeit- und $\mathcal{O}(n^2)$-Platzkomplexität

Programmiersprachen: spezielle CFG
Analyse durch deterministischen Automaten mit "input look ahead" bei linearem Platz- und Zeitbedarf

a) **Top-Down-Analyse:** Konstruktion des Ableitungsbaums von der Wurzel zu den Blättern in Form einer Linksanalyse
b) **Bottom-Up-Analyse:** umgekehrt, entspricht gespiegelter Rechtsanalyse

Kontextfreie Grammatiken
$G = \langle N, \Sigma, P, S\rangle \in CFG(\Sigma)$
Bezeichnungskonventionen:

$A, B, C, \ldots$	$\in N$	Nichtterminalsymbole
$a, b, c, \ldots$	$\in \Sigma$	Terminalsymbole
$u, v, w, \ldots$	$\in \Sigma^*$	Terminalwörter
$\alpha, \beta, \gamma, \ldots$	$\in \mathcal{X}$	Satzformen (mit $\mathcal{X} = N \cup \Sigma$)
$A \to \alpha$	$\in P$	Produktion/Regel

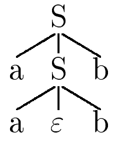
Ableitungsrelation: $\Rightarrow \subseteq (\mathcal{X}^*) \times (\mathcal{X}^*)$ ist definiert durch

$$\alpha \Rightarrow \beta \iff \alpha = \alpha_1 A \alpha_2,\ A \to \gamma \in P,\ \beta = \alpha_1 \gamma \alpha_2$$

Gilt außerdem $\alpha_1 \in \Sigma^*$ bzw. $\alpha_2 \in \Sigma^*$, so $\alpha \Rightarrow_l \beta$ bzw. $\alpha \Rightarrow_r \beta$.
Sprechweise: Ableitungsschritt, Links..., Rechts...

Erzeugte Sprache: $L(G) = \{w \in \Sigma^* | S \overset{*}{\Rightarrow} w\} = \{w \in \Sigma^* | S \overset{*}{\Rightarrow}_l w\} = \{w \in \Sigma^* | S \overset{*}{\Rightarrow}_r w\}$

Beispiel: $G: \ S \to aSb|\varepsilon$
$S \Rightarrow aSb \Rightarrow a^2Sb^2 \Rightarrow \dots$



Ableitungsbaum: repräsentiert die syntaktische Struktur des abgeleiteten Wortes

Definition 2.1
G heißt **eindeutig**, wenn es für jedes $w \in L(G)$ genau einen Ableitungsbaum gibt.

Folgerung: G ist eindeutig, genau dann wenn es für jedes $w \in L(G)$ genau eine Links- bzw. Rechtsanalyse gibt.

Definition 2.2
G heißt **mehrdeutig**, wenn G nicht eindeutig ist.

l-Analyse, r-Analyse
Darstellung von l- bzw. r-Ableitungen durch Nummernfolgen
$|P| = p, \ [p] := \{1, \dots, p\}, \ \pi : [p] \to P$ bijektiv, $\pi_i = \pi(i)$
$wA\alpha \overset{i}{\Rightarrow}_l w\gamma\alpha$ bzw. $\alpha Aw \overset{i}{\Rightarrow}_r \alpha\gamma w$, falls $\pi_i = A \to \gamma$
Für $z = i_1 i_2 \dots i_n \in [p]^+$ soll $\alpha \overset{z}{\Rightarrow}_l \beta \iff \alpha \overset{i_1}{\Rightarrow}_l \alpha_1 \overset{i_2}{\Rightarrow}_l \dots \overset{i_n}{\Rightarrow}_l \alpha_n = \beta$ für passende $\alpha_1, \dots, \alpha_{n-1}$.
$\alpha \overset{\varepsilon}{\Rightarrow}_l \alpha$ (ein leerer l-Ableitungsschritt)

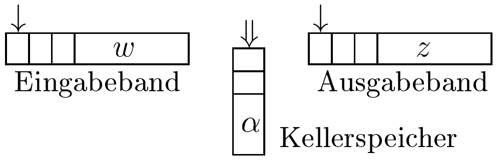
Definition 2.3
z heißt **l-Analyse** von $\alpha \iff S \overset{z}{\Rightarrow}_l \alpha$.
z heißt **r-Analyse** von $\alpha \iff S \overset{z}{\Rightarrow}_r \alpha$.

Syntaxanlyse
Gegeben: $G \in CFG, \ w \in \Sigma^*$. Berechnung einer l/r-Analyse, falls $w \in L(G)$; andernfalls Bestimmung syntaktischer Fehler.

Generalvoraussetzung
$G \in CFG$ ist reduziert, d.h. für jedes $A \in N$ gibt es $\alpha, \beta \in \mathcal{X}^*$ und $w \in \Sigma^*$, so daß $S \overset{*}{\Rightarrow} \alpha A \beta$ (A erreichbar) und $A \overset{*}{\Rightarrow} w$ (A produktiv).

2.1 Top-Down-Analyse, LL(k)-Grammatiken

Definition 2.4 (Top-Down-Analyseautomat von $G \in CFG$, $NTA(G)$)
Eingabealphabet: Σ
Kelleralphabet: $\mathcal{X} = N \cup \Sigma$
Ausgabealphabet: $[p]$
Zustandsalphabet: entfällt, der Automat arbeitet mit einem Kontrollzustand
Konfigurationsmenge: $\Sigma^* \times \mathcal{X}^* \times [p]^*$ (Kellerspitze links)



Transitionen:

- Ableitungsschritt: $(w, A\alpha, z) \vdash (w, \beta\alpha, zi)$ falls $\pi_i = A \to \beta$
- Vergleichsschritt: $(aw, a\alpha, z) \vdash (w, \alpha, z)$ für $a \in \Sigma$

Anfangskonfiguration für $w \in \Sigma^*$: (w, S, ε)
Endkonfiguration: $(\varepsilon, \varepsilon, z)$ falls $w \in L(G)$ (bzw. Fehler)

Beachte: Nichtdeterminismus wegen $A \to \beta|\gamma$

Satz 2.1 Der NTA(G) berechnet l-Analysen, d.h. $(w, S, \varepsilon \overset{*}{\vdash} (\varepsilon, \varepsilon, z) \iff z$ ist l-Analyse von w.

Beweis: "$\Rightarrow$", d.h. der Automat arbeitet korrekt
$(*)$ $(w, \alpha, y) \overset{*}{\vdash} (\varepsilon, \varepsilon, yz) \quad \curvearrowright \quad \alpha \overset{z}{\Rightarrow}_l w$
Beweis von $(*)$ durch Induktion über $k = |z|$

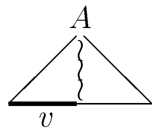
$k = 0$: $(w, \alpha, y) \overset{*}{\vdash} (\varepsilon, \varepsilon, y)$ nur Vergleichsschritte,
$w = \alpha$, also $(*)$ mit $w \overset{\varepsilon}{\Rightarrow}_l w$

$k \rightsquigarrow k+1$: $z = iz',\ \alpha = uA\beta,\ w = uv,\ \pi_i = A \to \gamma$
$(w, \alpha, y) = (uv, uA\beta, y) \overset{*}{\vdash} (v, A\beta, y) \vdash (v, \gamma\beta, yi) \overset{*}{\vdash} (\varepsilon, \varepsilon, yiz')$
Nach IV: $\gamma\beta \overset{z'}{\Rightarrow}_l v$ (in $|z'| = k$ Schritten) und damit folgt $(*)$ wegen
$\alpha = uA\beta \overset{i}{\Rightarrow}_l u\gamma\beta \overset{z'}{\Rightarrow}_l uv = w$, also $\alpha \overset{z}{\Rightarrow}_l w$.
"$\Leftarrow$" in der 3. Übung

Ziel: Nichtdeterminismus des NTA(G) durch k-lok-ahead auf der Eingabe beseitigen ($k \in \mathbb{N}$).

Definition 2.5 (first-Mengen)
Sei $G \in CFG$, $\alpha \in \mathcal{X}^*$ und $k \in \mathbb{N}$. Wir definieren $first_k(\alpha) \subseteq \Sigma^*$ durch $first_k(\alpha) := \{v \in \Sigma^* | \exists w \in \Sigma^* : \alpha \overset{*}{\Rightarrow} vw,\ |v| = k\} \cup \{v \in \Sigma^* | \alpha \overset{*}{\Rightarrow} v,\ |v| < k\}$

Folgerungen



(i) $first_k(\alpha) \neq \emptyset$ weil G reduziert
(ii) $\varepsilon \in first_k(\alpha) \iff k = 0$ oder $\alpha \overset{*}{\Rightarrow} \varepsilon$
(iii) $\alpha \overset{*}{\Rightarrow} \beta \quad \curvearrowright \quad first_k(\beta) \subseteq first_k(\alpha)$
(iv) $v \in first_k(\alpha) \iff \exists x \in \Sigma^* : \alpha \overset{*}{\Rightarrow}_l x, \{v\} = first_k(x)$

Wunsch: A-Regel durch v festlegen $\rightsquigarrow$ LL(k)-Grammatiken
(Lesen der Eingabe von links nach rechts mit k-look-ahead, Berechnung einer Linksanalyse)

Definition 2.6 (LL(k)-Grammatik)
Sei $G \in CFG$ und $k \in \mathbb{N}$. $G \in LL(k) \iff$ Für alle l-Ableitungen der Form

$$S \overset{*}{\Rightarrow}_l wA\alpha \begin{array}{l} \overset{l}{\nearrow} w\beta\alpha \overset{*}{\Rightarrow}_l wx \\ \\ \underset{l}{\searrow} w\gamma\alpha \overset{*}{\Rightarrow}_l wy \end{array} \quad \text{mit } first_k(x) = first_k(y) \text{ gilt: } \beta = \gamma.$$

Bemerkung:

- Linksableitungsschritt für $wA\alpha$ ist durch die nächsten k auf w folgenden Symbole bestimmt.
- Der NTA(G) kann für $G \in LL(k)$ mit k-look-ahead auf der Eingabe deterministisch simuliert werden.

Problem: Bestimmung der A-Regel aus k-look-ahead.

Lemma 2.1 $G \in LL(k) \iff$ Für alle l-Ableitungen der Form

$$S \overset{*}{\Rightarrow}_l wA\alpha \begin{array}{l} \overset{l}{\nearrow} w\beta\alpha \\ \\ \underset{l}{\searrow} w\gamma\alpha \end{array} \quad \text{mit } \beta \neq \gamma \text{ gilt: } first_k(\beta\alpha) \cap first_k(\gamma\alpha) = \emptyset.$$

Beweis: $\downarrow$ ("Def. $\curvearrowright$ Lemma")
Angenommen, die Definition und die Voraussetzung des Lemma ($\beta \neq \gamma$) gilt, aber $v \in first_k(\beta\alpha) \cap first(\gamma\alpha)$. Dann muß $\beta\alpha \overset{*}{\Rightarrow}_l x$ und $\gamma\alpha \overset{*}{\Rightarrow}_l y$ mit $first_k(x) = first_k(y)$. Also nach Def. $\beta = \gamma \Rightarrow$ Widerspruch!

$\uparrow$ ("Lemma-Eigenschaft $\curvearrowright$ Definition-Eigenschaft")
Angenommen Lemma-Eigenschaft und Voraussetzung der Definition gilt <u>und</u> $\beta \neq \gamma$. Daraus folgt: $first_k(\beta\alpha) \cap first_k(\gamma\alpha) = \emptyset$. Widerspruch, da $first_k(x) \subseteq first_k(\beta\alpha)$ und $first_k(y) \subseteq first_k(\gamma\alpha)$.

Problem: Abhängig der look-ahead-Menge vom Rechtskontext α.
Ziel: Betsimmung der look-ahead-Menge aus Regel allein.
Idee: Mögliche Rechtskontexte vereinigen.

Definition 2.7 (follow-Mengen)
Sei $G \in CFG$, $A \in N$ und $k \in \mathbb{N}$. Wir definieren $follow_k(A) \subseteq \Sigma^*$ durch $follow_k(A) := \{v \in \Sigma^* | S \overset{*}{\Rightarrow}_l wA\alpha,\ v \in first_k(\alpha)\}$.

$\boxed{\text{Der Fall } k = 1}$
I.A. genügt $k = 1$. $k > 1$ ist aufwendiger (wird aber zum Teil in heutigen Parser-Generatoren verwendet).

Satz 2.2 Für $G \in CFG$ (G reduziert) gilt: $G \in LL(1) \iff$ Für alle Regelpaare $A \to \beta|\gamma$ mit $\beta \neq \gamma$ folgt: $first_1(\beta follow_1(A)) \cap first(\gamma follow_1(A)) = \emptyset$.

Abkürzung: $fi := first_1$ und $fo := follow_1$

Definition 2.8
$la(A \to \beta) := fi(\beta fo(A))$ heißt **Look-ahead-Menge** von $A \to \beta$.

Beachte:

- $fi(\alpha) \subseteq \Sigma_\varepsilon := \Sigma \cup \{\varepsilon\}$
- $fo(A) \subseteq \Sigma_\varepsilon$
- $\beta fo(A) \subseteq \mathcal{X}^*$
- Für $\Gamma \subset \mathcal{X}^*$ ist $fi(\Gamma) := \bigcup\limits_{\alpha \in \Gamma} fi(\alpha)$
- $\boxed{la(A \to \beta)} = fi(\beta fo(A)) \boxed{\subseteq \Sigma_\varepsilon}$

Eigenschaft:

- $\varepsilon \in fi(\beta fo(A)) \quad \Longleftrightarrow \quad \beta \overset{*}{\Rightarrow} \varepsilon$ und $\varepsilon \in fo(A)$
- $a \in fi(\beta fo(A)) \quad \Longleftrightarrow \quad a \in fi(\beta)$ oder $[\beta \overset{*}{\Rightarrow} \varepsilon$ und $a \in fo(A)]$

Satz 2.3 $G \in LL(1) \quad \Longleftrightarrow \quad$ Für alle Regelpaare $A \to \beta | \gamma$ $(\beta \neq \gamma)$ folgt: $la(A \to \beta) \cap la(A \to \gamma) = \emptyset$.

Beweis: "$\curvearrowright$" $G \in LL(1)$
Angenommen, es gibt $A \to \beta|\gamma$ mit $\beta \neq \gamma$ und $c \in fi(\beta fo(A)) \cap fi(\gamma fo(A))$ für $c \in \Sigma_\varepsilon$. <u>Fall 1:</u> $c = \varepsilon$, alsp $\varepsilon \in fo(A) \quad \curvearrowright \quad S \overset{*}{\Rightarrow}_l wA\alpha$ und $\alpha \overset{*}{\Rightarrow} \varepsilon$. Außerdem $\beta \overset{*}{\Rightarrow} \varepsilon$ und $\gamma \overset{*}{\Rightarrow} \varepsilon$. Dies ist bestimmt ein Widerspruch zu $LL(1)$, weil

$$S \overset{*}{\Rightarrow}_l wA\alpha \begin{array}{l} \nearrow_l w\beta\alpha \\ \searrow_l w\gamma\alpha \end{array} \qquad \text{mit } \varepsilon \in fi(\beta\alpha) \cap fi(\gamma\alpha).$$

<u>Fall 2:</u> $c = a \in \Sigma$
Es folgt:

1) $a \in fi(\beta) \cap fi(\gamma)$
2) $a \in fi(\beta),\ \gamma \overset{*}{\Rightarrow} \varepsilon$ und $a \in fo(A)$
3) $a \in fi(\gamma),\ \beta \overset{*}{\Rightarrow} \varepsilon$ und $a \in fo(A)$
4) $\beta \overset{*}{\Rightarrow} \varepsilon,\ \gamma \overset{*}{\Rightarrow} \varepsilon$ und $a \in fo(A)$

Aus (1) ergibt sich (da G reduziert) eine Ableitung $S \overset{*}{\Rightarrow}_l wA\alpha \begin{array}{l} \nearrow_l w\beta\alpha \\ \searrow_l w\gamma\alpha \end{array}$ mit $a \in fi(\beta\alpha) \cap fi(\gamma\alpha)$, also Widerspruch zu $G \in LL(1)$, da $\beta \neq \gamma$.

In den Fällen (2)-(4) folgt aus $a \in fo(A)$ ebenso eine Ableitung

$$S \overset{*}{\Rightarrow}_l wA\alpha \begin{array}{l} \nearrow_l w\beta\alpha \\ \searrow_l w\gamma\alpha \end{array} \qquad \text{mit } a \in fi(\alpha).$$

Für (2) folgt: $a \in fi(\beta)$, also $a \in fi(\beta\alpha)$, $\gamma \overset{*}{\Rightarrow} \varepsilon$, also $a \in fi(\gamma\alpha)$.
Analog für (3) und (4).

"$\curvearrowleft$": $S \overset{*}{\Rightarrow}_l wA\alpha \begin{cases} \Rightarrow_l w\beta\alpha \\ \Rightarrow_l w\gamma\alpha \end{cases}$ mit $\beta = \gamma$.

Nach Voraussetzung gilt: $fi(\beta fo(A)) \cap fi(\gamma fo(A)) = \emptyset$

$fi(\beta\alpha) \subseteq fi(\beta fo(A))$ und $fi(\gamma\alpha) \subseteq fi(\gamma fo(A)) \quad \curvearrowright \quad fi(\beta\alpha) \cap fi(\gamma\alpha) = \emptyset$

Berechnung der la-Mengen

1) $fi(X)$ für $X \in \mathcal{X}$
   - $X = a \in \Sigma \quad \curvearrowright \quad fi(X) = \{X\}$
   - $X \to a\alpha \quad \curvearrowright \quad a \in fi(X)$
   - $X \to \varepsilon \quad \curvearrowright \quad \varepsilon \in fi(X)$
   - $X \to A_1 \dots A_k Y\alpha,\ k \geq 0,\ Y \in \mathcal{X},\ \varepsilon \in fi(A_1) \cap \dots \cap fi(A_k),\ a \in fi(Y) \quad \curvearrowright \quad a \in fi(X)$
   - $X \to A_1 \dots A_k,\ k \geq 1,\ \varepsilon \in fi(A_1) \cap \dots \cap fi(A_k) \quad \curvearrowright \quad \varepsilon \in fi(X)$
2) $fi(X_1 \dots X_n)$ für $X_i \in \mathcal{X},\ n \in \mathbb{N}$
   - $\varepsilon \in fi(X_1) \cap \dots \cap fi(X_n) \quad \curvearrowright \quad \varepsilon \in fi(X_1 \dots X_n)$
   - $\varepsilon \in fi(X_1) \cap \dots \cap fi(X_{i-1}),\ a \in fi(X_i) \quad \curvearrowright \quad a \in fi(X_1 \dots X_n)$
   - $fi(\varepsilon) = \{\varepsilon\}$
3) $fo(B)$ für $B \in \mathcal{X}$
   - $\varepsilon \in fo(S)$
   - $A \to \alpha B\beta,\ a \in fi(\beta),\ a \in \Sigma \quad \curvearrowright \quad a \in fo(B)$
   - $A \to \alpha B\beta,\ \varepsilon \in fi(\beta),\ X \in fo(A),\ X \in \Sigma \cup \{\varepsilon\} \quad \curvearrowright \quad X \in fo(B)$
   - $A \to \alpha B,\ X \in fo(A),\ X \in \Sigma \cup \{\varepsilon\} \quad \curvearrowright \quad X \in fo(B)$

Beispiel G'_{AE} :

$E \to TE'$	(1)	$E' \to +TE' \mid \varepsilon$	$(2,3)$	$T \to FT'$	(4)
$T \to *FT' \mid \varepsilon$	$(5,6)$	$F \to (E) \mid a$	$(7,8)$		

First- und Follow-Mengen:

	E	E'	T	T'	F
fi	(a	+ ε	(a	$*$ ε	(a
fo	ε)	ε+)	+ ε (	$*$ ε)	+ ε)

Look-ahead-Mengen:

1	(a	
2	+	durchschnittsfrei
3	ε)	
4	(a	
5	$*$	durchschnittsfrei
6	+ ε)	
7	(	durchschnittsfrei
8	a	

$\Longrightarrow \boxed{G'_{AE} \in LL(1)}$

$LL(1)$-Test la-Mengen berechnen und Alternativen auf Disjunktheit prüfen.

Deterministischer TD-Analyseautomat für $G \in LL(1)$, $DTA(G)$
Idee: Die Zugehörigkeit des Eingabesymbols zu einer la-Menge steuert die Regelauswahl. 1-llok-ahead auf der Eingabe.

Modifikation des Ableitungsschtritts des NTA(G):

- $(aw, A\alpha, z) \vdash (aw, \beta\alpha, zi)$ falls $\pi_i = A \to \beta$ und $\boxed{a \in la(\pi_i)}$
- $(\varepsilon, A\alpha, z) \vdash (\varepsilon, \beta\alpha, zi)$ falls $\pi_i = A \to \beta$ und $\boxed{\varepsilon \in la(\pi_i)}$

Folgerung: Deterministische Arbeitsweise des Analyseautomaten.
Beachte: Das Eingabesymbol wird bei Ableitungsschritten nicht gelöscht!

Darstellung des $DTA(G)$ durch die **Analysetabelle** von G oder die **action-Funktion** von G.

$$act: \underbrace{\Sigma_\varepsilon}_{Eingabe} \times \underbrace{(N \cup \Sigma_\varepsilon)}_{Kellerspitze} \to \underbrace{\big[\{\alpha | A \to \alpha \text{ in } G\} \times [p]\big]}_{Behandlung\ NT} \cup \underbrace{\{pop, error, accept\}}_{Behandlung\ Terminalsymbol}$$

ist definiert durch

- $act(x, \alpha) := (\alpha, i)$, falls $\pi_i = A \to \alpha,\ x \in la(\pi_i)$
- $act(a, a) := pop$
- $act(\varepsilon, \varepsilon) := accept$
- $act(x, x) := error$, sonst

Beachte: Look-ahead beseitigt Nichtdeterminismus und erlaubt frühe Fehlererkennung.

Parserkonstruktion nach TD-Methode
$G \in CFG$. Berechnung der la-Mengen (fi- und fo-Mengen). Analysetabelle, Eindeutigkeit prüfen $\Longrightarrow$ tabellengesteuerter Parser
Problem: mehrdeutige Analysetabelle, d.h. $G \notin LL(1)$

Transformation nach LL(1)
2 Methoden, G in eine äquivalente LL(1)-Grammatik zu transformieren (nicht immer möglich):

1) Beseitigung von Linksrekursion
2) Links-Faktorisieren

Die Methoden finden Verwendung in Parser-erzeugenden Systemen
Vorsicht! Transformationen erhalten zwar die Äquivalenz, im Allgemeinen aber nicht die syntaktische Struktur (Veränderung des Ableitungsbaums).

(1) Beseitigung von Linksrekursion

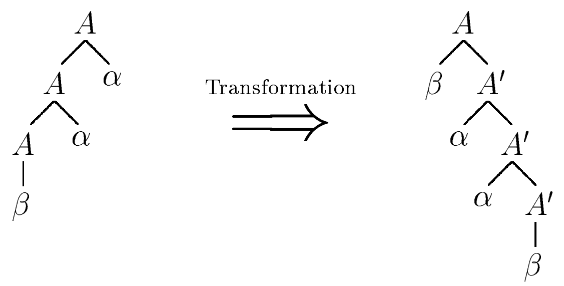
Definition 2.9
$G \in CFG$ **linksrekursiv** $\iff \exists A \in N,\ \alpha \in \mathcal{X}^* : A \stackrel{+}{\Rightarrow} A\alpha$.

Folgerung: G linksrekursiv $\curvearrowright \forall k \in \mathbb{N} : G$ ist keine $LL(k)$-Grammatik.
Grund: Wenn ein DTA $A \stackrel{+}{\Rightarrow} A\alpha$ simuliert, so bleibt der Eingabekopf stehen. Gleicher Look-ahead, also Endlosschleife! Keine Ableitung der Form $S \stackrel{*}{\Rightarrow}_l wA\beta \stackrel{+}{\Rightarrow}_l wa\alpha\beta \stackrel{*}{\Rightarrow}_l wv$

<u>**Beispiel**</u> G_{AE} ist linksrekursiv:
$G_{AE}:\ E \to E + T|T\ (1,2) \qquad T \to T * F|F\ (3,4) \qquad F \to (E)|a\ (5,6)$

LL(1)-Test: $fi(E) = fi(T) = fi(F) = \{(, a\}$
$\curvearrowright \quad la(\pi_i) = \{(, a\}$ für $i = 1, \ldots, 4 \quad \curvearrowright \quad G_{AE} \notin LL(1)$.

<u>Spezialfall</u> Direkte Linksrekursion und ihre Beseitigung
$A \to A\alpha|\beta$ mit $\beta \neq A \ldots$ und $\alpha \neq \varepsilon$ (ergibt$\beta\alpha^*$)
wird ersetzt durch $A \to \beta A'$ und $A' \to \alpha A'|\varepsilon$
Folgerung: $L(G)$ unverändert, jedoch neue syntaktische Struktur.



Kein Problem bei **assoziativen** Operatoren wie $+, *$, da Semantik invariant.

<u>**Beispiel**</u> Die Beseitigung der direkten Linksrekursion in G_{AE} ergibt die $LL(1)$-Grammatik G'_{AE}.

<u>Allgemeiner Fall</u> indirekte Linksrekursion
$A \to A_1\alpha_1| \ldots$
$A_1 \to A_2\alpha_2| \ldots$
$\vdots$
$A_n \to A\alpha_{n+1}| \ldots$

Idee: Beseitigung durch Transformation in Greibach Normalform (GNF).
GNF: Alle Regeln von der Form $A \to a\beta_1 \ldots \beta_n \quad (\beta_1 \neq S)$ oder $S \to \varepsilon$.
Also $fi(\beta\alpha) \cap fi(\gamma\alpha) = \emptyset$.
Beachte: Beseitigung von Linksrekursion ergibt nicht notwendigerweise eine $LL(1)$-Grammatik.
Grund: Jede $G \in CFG$ ist äquivalent in GNF transformierbar, aber $\mathcal{L}(LL(k)) \subsetneqq \mathcal{L}(LL(k+1)) \subsetneqq \mathcal{L}(DPDA)) \subsetneqq \mathcal{L}(PDA) = CFL$.

Komplexität der $LL(1)$-Analyse
$G \in LL(1) \quad \curvearrowright \quad G$ nicht linksrekursiv. Der $DTA(G)$ mit Eingabe von $w \in \Sigma^*$:

- $|w|$ Vergleichsschritte + 1 Erkennungsschritt für das Ende des Wortes
- maximal $|N|$ aufeinanderfolgende Ableitungsschritte

$\curvearrowright$ maximal $|N| \cdot (|w| + 1)$ Transitionen.
Maximale Kellerlänge: $\max\{|\alpha| \mid A \to \alpha \text{ in } P\} \cdot |N| \cdot (|w| + 1)$
Also: linearer Zeit- und Platzbedarf

(2) Links-Faktorisieren

Beispiel *statement* $\to$ `if` *condition* `then` *statement* `else` *statement*|
`if` *condition* `then` *statement*

Idee: Verschieben der Entscheidung bis Alternative erkennbar:
$A \to \alpha\beta|\alpha\gamma$ ersetzen durch $A \to \alpha A'$ und $A' \to \beta|\gamma$.

TD-Analyse mit rekursiven Prozeduren
Idee: Keine explizite Kellerbenutzung, sondern implizite Verwendung des Laufzeitkellers durch den Einsatz von rekursiven Prozeduren.
Vorteil: leichte Programmierung (Wirth)
Spezialfall: $G \in LL(1)$, wie z.B. G'_{AE}

(1) Analyseverfahren durch rekursiven Abstieg (ohne la-Mengen), "Recursive descent parser"
Methode: $A \in N \mapsto A()$, parameterlose Prozedur zur Simulationen eines Ableitungsschrittes
Annahme: Alternativen durch Eingabesymbol entscheidbar
Eingabe: `sym`: Variable für das Eingabesymbol
`nextsym`: zum Lesen des nächsten Eingabesymbols
Ausgabe: `print(i)`: Ausgabe einer Regelnummer
Beispiel: siehe Folie

(2) Zusätzliche Verwendung der la-Mengen
Vorteil: bessere Kontrolle der Regelverwendung, frühere Fehlererkennung

2.2 Bottom-Up-Analyse, LR(k)-Grammatik

Idee: Bottom-Up-Berechnung des Ableitungsbaums in Form einer gespiegelten Rechtsanalyse durch einen Kellerautomaten.
Shift-Schritte: Verschieben von Eingabesymbolen auf den Keller
Reduce-Schritte: Umkehrung von Ableitungsschritten (rechte Seite durch linke ersetzen)
$\rightsquigarrow$ **Shift-Reduce-Verfahren**

Definition 2.10
Der (nichtdeterministische) Bottom-Up-Analyseautomat von $G \in CFG$, Bezeichnung $NBA(G)$:
Eingabelalphabet: Σ
Kelleralphabet: $\mathcal{X}$
Ausgabealphabet: $[p]$
Konfigurationsmenge: $\mathcal{X}^* \times \Sigma^* \times [p]^*$ (Kellerspitze rechts)
Transitionen:

- Shift-Schritt: $(\alpha, aw, z) \vdash (\alpha a, w, z)$ für $a \in \Sigma$
- Reduce-Schritt: $(\beta\alpha, w, z) \vdash (\beta A, w, zi)$ falls $\pi_i = A \to \alpha$

Anfangskonfiguration für ein $w \in \Sigma^*$: $(\varepsilon, w, \varepsilon)$
Endkonfiguration: (S, ε, z)

Satz 2.4 Der $NBA(G)$ berechnet gespiegelte r-Analysen, d.h. für $w \in \Sigma^*$ und $z \in [p]^*$ gilt: z ist eine r-Analyse von $w \iff (\varepsilon, w, \varepsilon) \overset{*}{\vdash} (S, \varepsilon, \overleftarrow{z})$
(Dabei ist $\overleftarrow{z}$ die Spiegelung von z.)

<u>Nichtdeterminismus:</u>

1) Shift- oder Reduce-Schritt?
2) Reduce-Schritt: linker Henkelrand
3) Reduce-Schritt: linke Regelseite
4) Analyseende

Ziel: Nichtdeterminismus durch k-look-ahead auf der Eingabe beseitigen.

<u>LL(k)-Grammatiken</u>
Generalvorraussetzung: G **startsepariert**, d.h. S nur in $S \to A$ mit $A \neq \varepsilon$.
Jede $G \in CFG$ läßt sich durch Hinzufügen von $S' \to S$ in eine startseparierte Grammatik transformieren. (Regelnummer: 0)
Folgerung: (S', ε, z) ist eine Endkonfiguration, die nicht erneut in eine weitere Endkonfiguration übergehen kann (Analyseende).

Beseitigung des restlichen Nichtdeterminismus durch $LR(k)$-Grammatiken.

Definition 2.11 (LR(k)-Grammatiken)
Sei $G \in CFG$, startsepariert durch $S' \to S$, $k \in \mathbb{N}$. $G \in LR(k) \iff$ Für alle Rechtsableitungen der Form $S \begin{cases} \overset{*}{\Rightarrow}_r \alpha A w \overset{*}{\Rightarrow}_r \alpha\beta w \\ \overset{*}{\Rightarrow}_r \alpha' A' w' \overset{*}{\Rightarrow}_r \alpha\beta v \end{cases}$ mit $first_k(w) = first_k(v)$ gilt: $\alpha' = \alpha,\ A' = A,\ w' = v$.

Folgerung: Der BU-Analyseautomat kann mit k-look-ahead auf der Eingabe die Transition entscheiden.

$LR(0)$-Grammatiken
$k = 0 : \curvearrowright$ Entscheidung ohne look-ahead, allein durch Kellerinhalt $\alpha\beta$. Abstraktion endlicher Information aus $\alpha\beta$, welche für die Entscheidung ausreicht.

Definition 2.12 ($LR(0)$-Auskünfte, $LR(0)$-Mengen)
Sei $G \in CFG$ mit $S' \to S$ und $S' \overset{*}{\Rightarrow}_r \alpha A w \Rightarrow_r \alpha\beta_1\beta_2 w$. Dann heißt $[A \to \beta_1 \cdot \beta_2]$ eine $LR(0)$**-Auskunft für** $\alpha\beta_1$. Für $\gamma \in \mathcal{X}^*$ bezeichnet $LR(0)(\gamma)$ die Menge aller $LR(0)$-Auskünfte für γ, die sogenannte $LR(0)$**-Menge von** γ.

Folgerung:

1) $LR(0)(\gamma)$ endlich
2) $LR(0)(G) := \{LR(0)(\gamma) | \gamma \in \mathcal{X}^*\}$ endlich
3) $[A \to \beta_1 \cdot] \in LR(0)(\gamma)$ signalisiert Reduktionsmöglichkeit $(\alpha\beta_1, w, z) \vdash (\alpha A, w, zi)$ für $\pi_i = A \to \beta_1$ und $\gamma = \alpha\beta_1$.
4) $[A \to \beta_1 \cdot \beta_2] \in LR(0)(\gamma)$ mit $\beta_2 \neq \varepsilon$ bedeutet Shift-Möglichkeit wegen unvollständigem Henkel (rechte Regelseite).
5) $G \in LR(0) \iff$ Die $LR(0)$-Mengen enthalten keine wiedersprüchlichen Auskünfte (immer eindeutig).

Berechnung der $LR(0)$-Mengen einer Grammatik

Satz 2.5 $G \in CFG$ mit $S' \to S$. G reduziert. Dann gilt:

1) $LR(0)(\varepsilon)$ ist die kleinste Menge, welche
   a) $[S' \to \cdot S]$ enthält und
   b) mit $[A \to \cdot B\delta]$ und $B \to \beta$ in G auch $[B \to \cdot\beta]$ enthält
2) $LR(0)(aX)$ mit $X \in \mathcal{X}$ ist die kleinste Menge, welche
   a) $[A \to \beta_1 X \cdot \beta_2]$ enthält, falls $[A \to \beta_1 \cdot X\beta_2] \in LR(0)(\alpha)$
   b) und mit $[A \to \gamma \cdot B\delta]$ und $B \to \beta$ in G auch $[B \to \cdot\beta]$ enthält

Beispiel Siehe Folie 2.11 $\Longrightarrow$ Keine widersprüchlichen Auskünfte $\Longrightarrow$ G ist $LR(0)$-Grammatik.

Die goto-Funktion
G ist $LR(0)$-Grammatik $\curvearrowright$ $LR(0)(\gamma)$ liefert Shift-Reduce-Entscheidung für den BU-Analyseautomat mit Keller γ. Neues Kelleralphabet: $LR(0)(G)$ statt $\mathcal{X}$.

Beachte: $LR(0)(\gamma X)$ ist bestimmt durch $LR(\gamma)$ und X, aber unabhängig von γ.

$goto : LR(0)(G) \times \mathcal{X} \to LR(0)(G)$ ist definiert durch $goto(I, X) := I' \iff \exists\gamma \in \mathcal{X}^* : I = LR(0)(\gamma),\ I' = LR(0)(\gamma X)$.

Berechnung der $LR(0)$-Mengen und goto-Funktion durch Potenzmengenkonstruktion des nichtdeterministischen endlichen Automaten.

Sei $G \in CFG$, G startsepariert mit $S' \to S$. Konstruktion eines $\mathfrak{A}(G) \in NFA_\varepsilon$:
Zustandsmenge: $Q := \{[A \to \beta_1 \cdot \beta_2] | A \to \beta_1\beta_2 \text{ in } G\}$
Eingabealphabet: $\mathcal{X} \cup \Sigma$
Anfangszustand: $q_0 := [S' \to S]$
(Endzustandsmenge irrelevant, $F = Q$)
Transitionsfunktion $\delta : Q \times \mathcal{X}_\varepsilon \to \mathcal{P}(Q)$
$\delta([A \to \beta_1 \cdot X\beta_2], X) \ni [A \to \beta_1 X \cdot \beta_2]$
$\delta([A \to \beta_1 \cdot B\beta_2], \varepsilon) \ni [B \to \cdot\gamma]$ falls $B \to \gamma$

<u>Potenzmengenkonstuktion (Thompson)</u>

Konstruktion von $\widehat{\mathfrak{A}(G)} = \langle \widehat{Q}, \mathcal{X}, \widehat{\delta}, \widehat{q_0}, \emptyset \rangle \in DFA$.
Erweiterte Transitionsfunktion $\overline{\delta} : \mathcal{P}(Q) \times \mathcal{X}^* \to \mathcal{P}(Q)$
$\overline{\delta}(T, \varepsilon) := \varepsilon(T)$
$\overline{\delta}(T, wa) := \varepsilon(\bigcup_{q \in \overline{\delta}(T,w)} \delta(q, a))$
$\widehat{Q} := \{\overline{\delta}(\{[S' \to \cdot S]\}, \alpha) | \alpha \in \mathcal{X}^*\}$
$\widehat{\delta}(T, X) := \overline{\delta}(T, X)$
$\widehat{q_0} := \varepsilon(\{[S' \to \cdot S]\})$
Dann gilt: $\widehat{Q} = LR(0)(G)$, $\widehat{\delta} = goto$

<u>Konstruktion des deterministischen BU-Analyseautomat für $G \in LR(0)$</u>

Hilfsmittel: $LR(0)(G)$ und goto-Funktion
Die action-Funktion von G gibt die Shift-Reduce-Entscheidung an:
$act : LR(0)(G) \to \{shift, red\ i, accept, error | i \in [p]\}$

$$act(I) := \begin{cases} red\ i & \text{, falls } \pi_i = A \to \alpha \text{ und } [A \to \alpha \cdot] \in I \\ shift & \text{, falls } [A \to \alpha \cdot X\beta] \in I \\ accept & \text{, falls } [S' \to S \cdot] \in I \\ error & \text{, falls } I = \emptyset \end{cases}$$

Eindeutig bei $LR(0)$.

Die Funktionen act und $goto$ bilden die $LR(0)$-Analysetabelle von G. Diese Tabelle bestimmt den <u>$LR(0)$-Analyseautomaten, $DBA(G)$</u>:
Eingabealphabet: Σ
Kelleralphabet: $\Gamma := LR(0)(G)$
Ausgabealphabet: $\Delta := [p] \cup \{0\} \cup \{error\}$
Konfigurationsmenge: $\Gamma^* \times \Sigma^* \times \Delta*$
Transitionen:

- Shift-Schritt: $(\alpha I, aw, z) \vdash (\alpha I I', w, z)$ falls $act(I) = shift$ und $goto(I, a) = I'$
- Reduce-Schritt: Abkürzung: $wa_1 \dots a_n | n := w$ (Entfernen der letzen n Zeichen)
  $(\alpha I, w, z) \vdash (\widetilde{\alpha}\widetilde{I} I', w, zi)$ falls $act(I) = red\ i$, $\pi_i = A \to X_1 \dots X_n, \alpha I | n = \widetilde{\alpha}\widetilde{I}$ und $goto(I, A) = I'$
- Accept-Schritt: $(I_0 I, \varepsilon, z) \vdash (\varepsilon, \varepsilon, z0)$ falls $act(I) = accept$

- Fehlererkennung: $(\alpha I, w, z) \vdash (\varepsilon, \varepsilon, z\ error)$, sonst

Anfangskonfiguration für $w \in \Sigma^* : (I_0, w, \varepsilon)$ mit $I_0 = LR(0)(\varepsilon)$

Folgerung: Wenn $LR(0)(G)$ konfliktfrei, also *act* eindeutig, so arbeitet der $LR(0)$-Analyseautomat deterministisch und es gilt für $w \in \Sigma^*$ und $z \in [p]^*$:

- $(I_0, w, \varepsilon) \vdash^* (\varepsilon, \varepsilon, z0) \iff \overleftarrow{z}$ ist Rechtsanlyse von w
- $(I_0, w, \varepsilon) \vdash^* (\varepsilon, \varepsilon, z\ error) \iff w \notin L(G)$

Beispiel $w = aac$ bestimmt die folgende Berechnung:
$(0, aac, \varepsilon) \vdash (04, ac, \varepsilon) \vdash (044, c, \varepsilon) \vdash (0446, \varepsilon, \varepsilon) \vdash (0448, \varepsilon, 6) \vdash (048, \varepsilon, 65)$
$\vdash (03, \varepsilon, 655) \vdash (01, \varepsilon, 6552) \vdash (\varepsilon, \varepsilon, 65520)$
Überprüfung: $S \xrightarrow{0} S' \xrightarrow{2} C \xrightarrow{5} aC \xrightarrow{5} aaC \xrightarrow{6} aac$

SLR(1)-Analyse
S - simple, look-ahead mit einem Symbol

Beispiel siehe Folie CB 2-15

Konflikttypen: RR (reduce/reduce)-Konflikt und SR (shift/reduce)-Konflikt
Im Beispiel nur SR-Konflikte.
Ziel: Beseitigung des Konflikts durch das Eingabesymbol (look-ahead)
Beobachtung

1) $[A \to \beta_1 \cdot a\beta_2] \in LR(0)(\alpha\beta_1) \quad \curvearrowright \quad S \overset{*}{\Rightarrow}_r \alpha A w \Rightarrow_r \underbrace{\alpha\beta_1}_{Keller} \quad \underbrace{a}_{Eingabesymbol} \quad \beta_2 w$

   Also: Shift nur bei Eingabe von a.
2) $[A \to \beta\cdot] \in LR(0)(\alpha\beta) \quad \curvearrowright S \overset{*}{\Rightarrow}_r \alpha A a w \Rightarrow_r \underbrace{\alpha\beta}_{Keller} \quad \underbrace{a}_{Eingabesymbol} \quad w \quad \curvearrowright$

   $a \in fo(A)$ Also: Reduktion mit $A \to \beta$ nur falls $a \in fo(A)$.

Für obiges Beispiel:
I_1: Shift bei Eingabe von $+$, Accept bei Eingabe von \$ (\$ entspricht dem rechten Eingaberand $\widehat{=}$ leeres Wort ε)
I_2: Shift bei Eingabe von $*$, Reduce bei Eingabe von aus $fo(E) = \{\underbrace{\varepsilon}_{\widehat{=}\$}, +,)\}$

I_9: Shift bei Eingabe von $*$, Reduce bei Eingabe von $\$, +,)$
$\Longrightarrow$ Konflikte beseitigt.
Weiterer Vorteil: Frühere Fehlererkennung durch genauere Kontrolle der Aktionen.

SLR(1)-action-Funktion

$act : LR(0)(G) \times (\Sigma \cup \{\$\}) \to \{shift, red\ i, accept, error | 1 \leq i \leq r\}$

$$act(I,a) := \begin{cases} shift & \text{, falls } [A \to \alpha \cdot a\beta] \in I \\ red\ i & \text{, falls } \pi_i = A \to \alpha,\ [A \to \alpha \cdot] \in I \text{ und } a \in fo(A) \\ accept & \text{, falls } [S \to S'] \in I \text{ und } a = \$ \\ error & \text{, sonst} \end{cases}$$

Definition 2.13
$G \in SLR(1) \iff act(I,a)$ eindeutig.

LR(1)-Analyse

Nicht immer sind Konflikte über follow-Mengen lösbar. Beispiel: Folie CB 2-17
Verfeinerung der $LR(0)$-Auskünfte durch Mitführen der möglichen look-ahead-Symbole.

Definition 2.14 ($LR(1)$-Auskünfte und $LR(1)$-Mengen für $G \in CFG$)

1) Wenn $S \overset{*}{\Rightarrow}_r \alpha A a w \Rightarrow_r \alpha\beta_1\beta_2 a w$, so $[A \to \beta_1 \cdot \beta_2, a] \in LR(1)(\alpha\beta_1)$
2) Wenn $S \overset{*}{\Rightarrow}_r \alpha A \Rightarrow_r \alpha\beta_1\beta_2$, so $[A \to \beta_1 \cdot \beta_2, \$] \in LR(1)(\alpha\beta_1)$

$LR(1)(G) := \{LR(1)(\gamma) | \gamma \in \mathcal{X}^*\}$

Berechnung der $LR(1)$-Mengen

Modifikation der Berechnung von $LR(0)(G)$ unter Berücksichtigung des Rechtskontexts

- $LR(1)(\varepsilon)$:
  - $[S' \to \cdot S, \$]$
  - Wenn $[A \to \cdot B\delta, x]$, $B \to \beta$ in G, $x \in \Sigma \cup \{\$\}$ und $y \in fi(\delta x)$, so $[B \to \cdot\beta, y] \in LR(1)(\varepsilon)$
- $LR(1)(\alpha X)$:
  - Wenn $[A \to \beta_1 \cdot X\beta_2, x] \in LR(1)(\alpha)$, so $[A \to \beta_1 X \cdot \beta_2, x] \in LR(1)(\alpha X)$
  - Wenn $[A \to \gamma \cdot B\delta, x] \in LR(1)(\alpha X)$, $B \to \beta$ in G und $y \in fi(\delta x)$, so $[B \to \cdot\beta, y] \in LR(1)(\alpha X)$

Die $LR(1)$-action-Funktion von G

$act : LR(1)(G) \times (\Sigma \cup \{\$\} \to \{shift, accept, error\} \cup \{red\ i | 1 \leq i \leq r\}$ sei definiert durch

$$act(I,x) := \begin{cases} red\ i & \text{, falls } \pi_i = A \to \alpha \text{ und } [A \to \alpha \cdot, x] \in I \\ accept & \text{, falls } x = \$ \text{ und } [S' \to S \cdot] \in I \\ shift & \text{, falls } x \neq \$ \text{ und } [A \to \alpha_1 \cdot x\alpha_2, y] \in I \\ error & \text{, sonst} \end{cases}$$

Dann gilt: $G \in LR(1) \iff act$ eindeutig (konfliktfrei).

Beispiel Folie CB 2-19

$LALR(1)$-Analyse
Beseitigung von Konflikten nach $LR(1)$ zu aufwendig.
Im Beispiel: $|LR(0)(G)| = 11$, $|LR(1)(G)| = 15$.
Algol60: $|LR(0)(G)| = 100$, $|LR(1)(G)| = 1000$.
Beobachtung: Informationsredundanz bei $LR(1)(G_2)$

Definition 2.15
$I_1, I_2 \in LR(1)(G)$ heißen $LR(0)$-**äquivalent**, $I_1 \underset{0}{\sim} I_2 \iff$ die "$LR(0)$-Anteile" von I_1 und I_2 sind gleich.

Beispiel $I'_4 \underset{0}{\sim} I'_{11},\ I'_5 \underset{0}{\sim} I'_{12},\ I'_7 \underset{0}{\sim} I'_{13},\ I'_8 \underset{0}{\sim} I'_{10}$

Folgerung: $|LR(1)(G)/\underset{0}{\sim}| = |LR(0)(G)|$
Oft können $LR(0)$-äquivalente $LR(1)$-Informationen vereinigt werden, ohne daß die Konfliktlösbarkeit verloren geht.

Definition 2.16
$I \in LR(1)(G)$ bestimmt eine $LALR(1)$-Menge $\bigcup\{I' \in LR(1)(G) | I' \underset{0}{\sim} I\}$.

Es gilt: $|LALR(1)(G)| = |LR(0)(G)|$.
Aber im Unterschied zu $LR(0)$-Mengen enthalten $LALR(1)$-Mengen look-ahead-Symbole zur Lösung von Konflikten.
Die $LALR(1)$-action-Funktion von G ist analog zu $LR(1)$ definiert.

Definition 2.17
$G \in LALR(1) \iff LALR(1)$-action-Funktion ist eindeutig.

Beispiel Die $LALR(1)$-Mengen von G_2, $LALR(1)(G_2) := \{I_i^{(1/0)} | 1 \leq i \leq 9\}$
$I_i^{(1/0)} = I'_i$ für $i \in \{0, 1, 2, 3, 6, 9\}$
$I_4^{(1/0)} = I'_4 \cup I'_{11}:\ [L \to * \cdot R, = /\$]\ [R \to \cdot L, = /\$]\ [L \to \cdot * R, = /\$]\ [L \to \cdot a, = /\$]$
$I_5^{(1/0)} = I'_5 \cup I'_{12}:\ [L \to a\cdot, = /\$]$ (Vergröberung von I'_{12})
$I_7^{(1/0)} = I'_7 \cup I'_{13}:\ [L \to *R\cdot, = /\$]$
$I_8^{(1/0)} = I'_8 \cup I'_{10}:\ [R \to L\cdot, = /\$]$
Die $LR(1)$-action-Funktion zeigt, daß beim Übergang zu $LALR(1)$ keine Konflikte auftreten $\implies G_2 \in LALR(1)$.

Die $LR(1)$-goto-Funktion überträgt sich auf $LALR(1)(G)$, weil für $LR(1)$-Mengen I_1 und I_2 gilt: $I_1 \underset{0}{\sim} I_2 \curvearrowright goto(I_1, x) \underset{0}{\sim} goto(I_2, x)$.
Grund: Der "$LR(0)$-Kern" von $LR(1)(\alpha X)$ ist durch den "$LR(0)$-Kern" von $LR(1)(\alpha)$ und X vollständig bestimmt.

2.3 Bottom-Up-Analyse mehrdeutiger Grammatiken

Es gilt für $G \in CFG:$ G mehrdeutig $\curvearrowright$ $G \notin LR = \bigcup_{k \in \mathbb{N}} LR(k)$.
Mehrdeutigkeit: natürliches Beschreibungsmittel bei Programmiersprachen zur Vermeidung aufwendiger Klammerung.
Auflösung durch Regeln für Präzendenz und Assoziativität von Op-Symbolen (allgemeiner: von Synt. Konstruktionen).

Beispiel $G^m_{AE}:$ $E \to E + E|E * E|(E)|id$

Mehrdeutige Grammatiken
G^m_{AE}, Präzedenz: $*$ vor $+$, Assoziativität: links.
Konflikte: I_1 $SLR(1)$-lösbar, I_7, I_8 nicht LR-lösbar

Beispiel-Rechnung: I_0 $a + a * a$, $I_0 I_3$ $+ a * a$, $I_0 I_1$ $+ a * a$, $I_0 I_1 I_4$ $a * a$, $I_0 I_1 I_4 I3$ $* a$, $I_0 I_1 I_4 I_7$ $* a$ $\implies$ shift oder reduce?

Beispiel Mehrdeutigkeit bei Verzweigungen ("dangling else")
$S \to iSeS|iS|a|b$

$$\texttt{if}\ b\ \texttt{then}\ \underbrace{\underbrace{\texttt{if}\ b\ \texttt{then}\ a}_{\text{Klammerung 1}}\ \texttt{else}\ a}_{\text{Klammerung 2}}$$

Klammerung 1 macht man nicht, Klammerung 2 ist Standard
$act(I_4, e) = shift$

3 Semantische Analyse, Attributgrammatiken

Ergebnis der syntaktischen Analyse: Ableitungsbaum
Kontextabhängige Eigenschaften:

- Deklariertheit von Bezeichnern
- Typinformationen

nicht durch CFG beschreibbar. Festlegung dieser Eigenschaften durch

- Gültigkeitsregeln: Gültigkeitsbereich einer Deklaration
- Sichtbarkeitsregeln: Sichtbarkeit im Gültigkeitsbereich
- Typvorschriften: Typkonsistenz

Statische Semantik: Kontextabhängige, laufzeitunabhängige Eigenschaften eines Programms. Formale Beschreibung durch **Attributgrammatiken**.
Idee: $CFG+$ semantische Regeln $\rightsquigarrow$ Zusatzinformationen für den Ableitungsbaum
Semantische Analyse = Attributberechnung
Ihr Ergebnis: attributierter Ableitungsbaum
Grundlage für die anschließende Synthesephase.

Attributgrammatiken
Idee: Attribute für $A \in N$, semantische Regeln für ihre Berechnung
synthetische Attribute: Bottom-Up-Berechnung
inherite[1] **Attribute**: Top-Down-Berechnung
$\curvearrowright$ belibiger Informationstransfer im Ableitungsbaum
Attributwerte: Symboltabellen, Typen, Code, Fehler, ...
Breite Anwendbarkeit von Attributgrammatiken, syntaxgerichtete Programmierung
Automatische Attributauswertung in Compilergeneratoren
Hist.: Knuth[68], Semantics of CFLs

Beispiel Folie CB 3-1

$B \to 0$	$v.0 = 0$	
$B \to 1$	$v.0 = 1$	
$L \to B$	$v.0 = v.1$	$l.0 = 1$
$L \to LB$	$v.0 = 2 * v.1 + v.2$	$l.0 = l.1 + 1$
$N \to L$	$v.0 = v.1$	
$N \to L.L$	$v.0 = v.1 + v.3/2^{l.3}$	

G_B erzeugt Binärzahlen mit und ohne Punkt, N Startsymbol
Synthetische Attribute: B, N: v (value)
L: v, l (value, length)

[1] vererbende

Semantische Regeln: Attributgleichungen mit Attributvariablen
Index i: i-tes Symbol
Ziel: Bestimmung des Zahlenwertes
Attributwerte von v: rationale Zahlen ($A^v = \mathbb{Q}$)
Attributwerte von l: natürliche Zahlen ($A^l = \mathbb{N}$)

2. Attributierungsart von G_B mit synthetischen und inheriten Attributen
Zusätzlich inherites Attribut für Bits und Listen: p (position)

Beispiel Folie CB 3-3

Berechnung des Wurzelattributs:

1) Längen $\uparrow$
2) Position $\downarrow$
3) Werte $\uparrow$

Definition 3.1 (Attributgrammatik)
Sei $G = \langle N, \Sigma, P, S\rangle \in CFG$. Sei Att eine Menge von Attributen, sei $A = (A^\alpha | \alpha \in Att)$ eine Familie von **Attributwertungen** und sei $att : \mathcal{X} \to \mathcal{P}(Att)$ eine **Attributzuordnung**.
Sei $Att = Syn \uplus Inh$ eine Zerlegung in Teilmengen **synthetischer** und **inheriter Attribute**, so daß att in $syn : \mathcal{X} \to \mathcal{P}(Syn)$ mit $syn(X) = att(X) \cap Syn$ und $inh : \mathcal{X} \to \mathcal{P}(Inh)$ mit $inh(X) = att(X) \cap Inh$ zerfällt.
Eine Regel $\pi = X_0 \to X_1X_2\ldots X_r \in P$ bestimmt die Menge $Var_\pi := \{\alpha.i | \alpha \in att(X_i), 0 \leq i \leq r\}$ der formalen Attributvariablen von π mit den Teilmengen $IVar_\pi := \{\alpha.i | (i = 0 \text{ und } \alpha \in syn(X_i)) \text{ oder } (1 \leq i \leq r \text{ und } \alpha \in inh(X_i))\}$ und $OVar_\pi := Var_\pi \setminus IVar_\pi$ der **Innen-** und **Außenvariablen**.
Eine **Attributgleichung** von π (semantische Regel) hat die Form
$\alpha.i = f(\alpha_1.i_1, \ldots, \alpha_n.i_n)$ mit $\alpha.i \in IVar_\pi,\ \alpha_1.i_1, \ldots, \alpha_n.i_n \in OVar_\pi$,
$f : A^{\alpha_1} \times A^{\alpha_2} \times \ldots \times A^{\alpha_n} \to A^\alpha$ und $n \in \mathbb{N}$.
Sei E_π eine (endliche) Menge von Attributgleichungen, in der jede Innenvariable genau einmal vorkommt. Dann heißt $\mathfrak{A} = \langle G, \{E_\pi | \pi \in P\}\rangle$ eine **Attributgrammatik**, Bezeichnung: $\mathfrak{A} \in AG$.

Definition 3.2 (Attributgleichungssystem eines Ableitungsbaums)
Sei $\mathfrak{A} = \langle G, \{E_\pi | \pi \in P\}\rangle \in AG$. $\mathfrak{A}$ induziert für jeden Ableitungsbaum t von G ein **Attributgleichungssystem** E_t:

> Sei $Kn(t)$ die Menge der Knoten von t. Sie bestimmt die Menge $Var_t := \{\alpha.k | k \in KN(t) \text{ markiert durch } X \in \mathcal{X},\ \alpha \in att(X)\}$ der **aktuellen Attributvariablen von** t. Wird an einem inneren Knoten (kein Blatt knoten) $k_0 \in Kn(t)$ die Regel $\pi = X_0 \to X_1X_2\ldots X_r$ angewandt und sind $k_1, \ldots, k_n \in Kn(t)$ die entsprechenden Nachfolgeknoten, so erhält man das **Attributgleichungssystem** E_{k_0} **von** k_0 aus E_π durch die Indextransformation $(i \to k_i | 0 \leq i \leq r)$ bei den Attributvariablen (formale $\to$ aktuelle Parameter).

Dann ist $E_t := \bigcup\{E_k | k \text{ innerer Knoten von } t\}$.

Beachte: Zu jeder aktuellen Attributvariablen $\alpha.k$, ausgenommen die inheriten der Wurzel und die synthetischen der Blätter, gibt es genau eine Gleichung $\alpha.k = \ldots$

Annahme

- keine inheriten Attribute des Startsymbols (ausgenommen inkrementelle Übersetzung)
- synthetische Attribute der Terminalsymbole vom Scanner geliefert

Lösbarkeit von E_t:
E_t kann keine, genau eine oder mehrere Lösungen haben.

Beispiel G enthalte die Regeln $A \to u \underbrace{B}_{\text{Symbol } i} v$ und $B \to w$. Ferner: $\alpha \in syn(B)$ und $\beta \in inh(B)$.
Attributgleichungen: $\ldots \quad \beta.i = f(\alpha.i) \quad \alpha.0 = g(\beta.0) \quad \ldots$
Also existiert in E_t eine **zirkuläre Abhängigkeit**: $\beta.k = f(\alpha.k) \quad \alpha.k = g(\beta.k)$
Für $A^\alpha = A^\beta = \mathbb{N},\ g = id_{\mathbb{N}} \quad \curvearrowright \quad \beta.k = f(\beta.k)$

1) $f(x) = x + 1 \quad \curvearrowright$ keine Lösung
2) $f(x) = 2x \quad \curvearrowright$ eine Lösung ($\beta.k = 0$)
3) $f(x) = x \quad \curvearrowright$ unendlich viele Lösungen

Folgerung: Zirkularitäten sind unerwünscht. Sie entstehen erst in $E_t := \bigcup\{E_k | k$ innerer Knoten von $t\}$, nicht in E_π.

Definition 3.3
$\mathfrak{A} \in AG$ heißt **zirkulär** $\iff$ Es gibt einen Ableitungsbaum t mit zirkulärem Attributgleichungssystem E_t, d.h. eine aktuelle Attributvariable hängt von sich selbst ab.

Satz 3.1 Es ist entscheidbar, ob $\mathfrak{A} \in AG$ zirkulär ist.
Zeitkomplexität: $\mathcal{O}(2^n)$ für $n = |\mathfrak{A}|$.

Hilfsmittel für Zirkularitätstests und Attributberechnung

Definition 3.4
Sei $\mathfrak{A} \in AG$ und t ein Abbleitungsbaum von $\mathfrak{A}$. E_t bestimmt den Abhängigkeitsgraphen DG (Dependency graph):
Knotenmenge $K_n(DG_t) := Var_t$
Kantenmenge $(x_1, x_2) \in Kan(DG_t) \iff x_2 = f(\ldots, x_1, \ldots) \in E_t$

Also: $x_1 \longrightarrow x_2$ in DG_t, wenn x_2 direkt von x_1 abhängt.

Attributberechnung

1) E_t als Termersetzungssystem, TD-Berechnung: Keine Zirkularität $\Longrightarrow$ Termination mit eindeutiger Lösung
2) BU-Berechnung: Variablen durch Werte ersetzen und Terme ausrechnen
3) Uniforme Berechnung: Unabhängigkeit vom Ableitungsbaum
   a) Berechnungspläne für $(E_\pi | \pi \in P)$ aufstellen [Kennedy/Warren]
   b) Rekursive Prozeduren/Funktionen für $(E_\pi | \pi \in P)$ [Jourdan/Gallie]
      Idee: Jeder synthetischen Attributvariablen wird eine Prozedur/Funktion zugeordnet mit den inheriten Attributvariablen als Parameter. Verwendung vor allem in der automatischen Compilererzeugung
4) Spezialfälle: SAG, LAG mit Attributberechnungen während der Syntaxanalyse

S-Attributgrammatiken

Definition 3.5
$\mathfrak{A} \in AG$ heißt **S-Attributgrammatik**, $\mathfrak{A} \in SAG$, wenn $Inh = \emptyset$, also $Att = Syn$.

Folgerung: BU-Berechnung der Attributwerte
Spezialfall: Bestimmung des Wurzelattributs
Idee: Durchführung während der BU-Syntaxanalyse

$A \to BaD$	
$I_1\ v3$	$v.0 = f(\ldots$
$I_7 v2$	$w.0 = f(\ldots$
$I_8 v1$	
I_2	Beim Reduce-Schritt werden simultan die Werte der synthetischen Attribute von A aus den Werten unter $v1\ v2\ v3$ berechnet (Records) Accept-Schritt: Werte der Wurzelattribute ausgeben

Beispiel Stackcode für arithmetische Ausdrücke
$E \to (E + E) \mid (E * E) \mid id \mid num$
Attribut c: Stackcode $\quad a$: Adresse $\quad z$: Zahl

$E \to (E + E)$	$c.0 = c.2;\ c.4;\ ADD$	(auch $+$)
$E \to (E * E)$	$c.0 = c.2;\ c.4;\ MULT$	(auch $*$)
id	$c.0 = LOAD\ a.1$	
num	$c.0 = LIT\ z.1$	

Werte von $a.1, z.1$: lexikalische Attribute, die mit dem Token vom Scanner übergeben werden: (id, x) $(num, 7)$

$((3 + x) * (y + 5)) \quad \rightsquigarrow \quad 3;\ x;\ +;\ y;\ 5;\ +;\ *$
Scanner: $((< num, 3 > + < id, x >) * (< id, y > + < num, 5 >))$

<u>Codegenerierung durch Attributsauswertung während der Syntayanalyse</u>

a) Shift: Aufruf des Scanners durch "`nextsym`"

   1) Token $\mapsto$ $LR(0)$.Menge mit goto
   2) lexikalisches Attribut ebenfalls auf Analysekeller

b) Reduce:

   1) Reduktion auf dem Analysekeller
   2) Simultane Attributberechnung

c) Accept: Wurzelattribute ausgeben

<u>Beispiel</u>

$$\begin{array}{l|l} num & 3 \\ (& \\ (& \end{array} \overset{Reduce}{\Longrightarrow} \begin{array}{l|l} id & x \\ + & \\ E & LIT\ 3 \\ (& \\ (& \end{array} \overset{Reduce}{\Longrightarrow} \begin{array}{l|l}) & \\ E & LOAD\ x \\ + & \\ E & LIT\ 3 \\ (& \\ (& \end{array}$$

$$\overset{Reduce}{\Longrightarrow} \begin{array}{l|l}) & \\ E & LIT\ 3;\ LOAD\ x;\ ADD \\ (& \end{array}$$

Ergebnis: $LIT\ 3;\ LOAD\ x;\ ADD;\ LOAD\ y;\ LIT\ 5;\ ADD;\ MUL$

<u>Beispiel</u> Berechnung des abstrakten Syntaxbaums
<u>Konkrete und abstrakte Syntax</u>
$\pi = A_0 \to w_0 A_1 w_1 \dots A_r w_r$ repräsentiert ein Op.-Symbol
F_π vom Typ $A_1 x \dots x A_r \to A_0 \quad (A_i :=$ Sorte|Typ)

Bemerkung: "Verkleben" der Regeln zu einem Ableitungsbaum entspricht der funkt. Applikation der zugehörigen Op.-Symbole.
Folgerung: Ableitungsbaum vereinfacht sich zum abstrakten Syntaxbaum (AST). Nur dieser ist für die Übersetzung erforderlich.

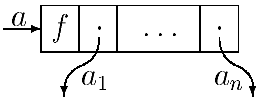
Konkrete Syntax: benutzerfreundlich: "Mix-Fix-Notation" (synthetischer Zucker) Verwendung natürlicher Sprache für Terminalsymbole von CFG.
Abstrakte Syntax: Darstellungsunabhängige algebraische Struktur

<u>Beispiel</u> Folie CB 3-6 `id=num+id; if id<num then id=num+id end`

<u>Aufgabe</u> Berechnung des AST während der BU-Analyse
<u>Methode</u> S-Attributgrammatik

Darstellung des Aufbaus von abstrakten Syntaxbäumen durch Konstruktion von Graphen

a) Graphen
$a \in Adr$ unendliche Menge von Adressen für Speicheradressen (Heap)
Ω Operationssymbolalphabet mit Stelligkeit
$f^{(n)} \in \Omega = \{assign^{(1)}, seq^{(2)}, cond^{(2)}, less^{(2)}, plus^{(2)}, ident^{(0)}, number^{(0)}\}$
Kno Menge der Ω-Knoten
$k \in Kno := \{(a, f, a_1, \ldots, a_n) | a, a_i \in Adr, f \in \Omega\}$



$Graph$: Menge von Knoten mit einer Wurzel
$G \in Graph := \{(a, K) | a \in Adr, K \subseteq Kno\}$

b) Konstruktorfunktionen für Graphen
$f^{(n)} \in \Omega \mapsto mk - f : Graph^n \to Graph$
$mk - f((a_1, K_1), \ldots, (a_n, K_n)) := (a, K)$ mit neuer Adresse a und
$K := \{a, f, a_1, \ldots, a_n)\} \cup \bigcup_{i=1}^{n} K_i$

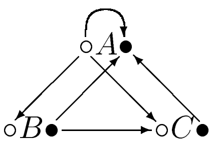
S-Attributierung von G_S
Wurzeladressen als S-Attribute im Analysekeller, Heap anstelle eines Ausgabebandes

Beispiel Typberechnung für arithmetische Ausdrücke
$Typ := \{int, real\}$ mit $\max : Typ^2 \to Typ$ bzgl. $int < real$

L-Attributgrammatiken

Definition 3.6
$\mathfrak{A} = \langle G, (E_\pi | \pi \in P) \rangle \in AG$ ist eine **L-Attributgrammatik**, $\mathfrak{A} \in LAG \iff$
Für jede Attributgleichung $\alpha.i = f(\ldots, \beta.j, \ldots)$ gilt: $\alpha \in Inh, \beta \in Syn \curvearrowright j < i$

Beispiel (DG_π)
Erlaubter Abhängigkeitsgraph:

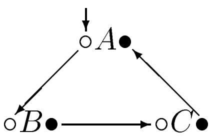


$\circ$ inherites Attribut
$\bullet$ synthetisches Attribut

Folgerung: $G \in LAG$ nicht zirkulär.
Attributberechnung in "depth-first, left to right-order"
Baumreise mit zwei Knotenbesuchen

1) top-down: inherite Attribute
2) bottom-up: synthetische Attribute

Auswertungsreihenfolge:



Syntaxanalyse mit L-Attributierung
Sei $\mathfrak{A} = \langle G, (E_\pi | \pi \in P) \rangle \in LAG$ mit $G \in LL(1)$.
Ziel Erweiterung der Top-Down-Analyse zur Berechnung der synthetischen Wurzelattribute.

<u>Methode</u> Expansion von $A \in N$ auf Analysetabelle so gestalten, daß spätere Reduktion möglich, TD $\rightarrow$ inherite Attribute, BU $\rightarrow$ synthetische Attribute

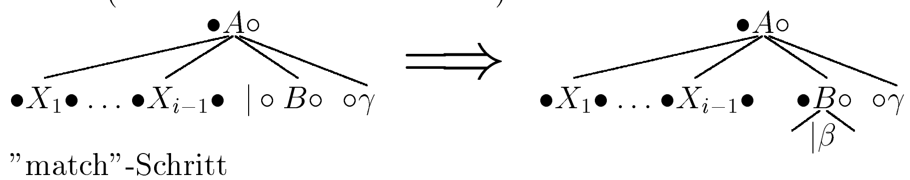
Kelleralphabet: $\bigcup_{\pi \in P} (LR(0)_\pi(G) \times Val_\pi)$ mit $LR(0)_\pi(G) := \{[A \rightarrow \alpha \cdot \beta] | \pi = A \rightarrow \alpha\beta\}$ und $Val_\pi := \{val_\pi | val_\pi : Var_\pi \dashrightarrow \text{"Wertmenge (Attribute)"}\}$

<u>Arbeitsweise</u> 3 Fälle

1) "expand"-Schritt mit Berechnung inheriter Attribute

$$\begin{array}{|l|l|} \vdots & \\ \hline [A \rightarrow X_1 ... X_{i-1} \cdot B\gamma] & val \\ \hline \end{array} \Longrightarrow \begin{array}{|l|l|} \vdots & \\ \hline [A \rightarrow X_1 ... X_{i-1} \cdot B\gamma] & val \\ \hline [B \rightarrow \cdot \beta] & val' \\ \hline \end{array}$$

Dabei ist $B \rightarrow \beta$ wegen $G \in LL(1)$ durch Eingabe-look-ahead bestimmt. val' belegt die inheriten Attributvariablenvon B nach den Attributgleichungen für $A \rightarrow X_1 \ldots X_{i-1} B\gamma$. Sei $\alpha \in inh(B)$ und $\alpha.i = t \in E_{X_1 \ldots X_{i-1} B\gamma}$, so ist $val'(\alpha.0) = \widehat{val}(t)$, wobei $\widehat{val}$ die homomorphe Fortsetzung von val auf Terme ("einsetzen und ausrechnen") ist.



2) "match"-Schritt

$$\begin{array}{|l|l|} \vdots & \\ \hline [A \rightarrow \gamma \cdot a\gamma'] & val \\ \hline \end{array} \Longrightarrow \begin{array}{|l|l|} \vdots & \\ \hline [A \rightarrow \gamma a \cdot \gamma'] & val \\ \hline \end{array}$$

falls a nächstes Eingabesymbol ist

3) "reduce"-Schritt mit Berechnung synthetischer Attribute

$$\begin{array}{|l|l|} \vdots & \\ \hline [A \rightarrow X_1 ... X_{i-1} \cdot B\gamma] & val \\ \hline [B \rightarrow \beta \cdot] & val' \\ \hline \end{array} \Longrightarrow \begin{array}{|l|l|} \vdots & \\ \hline [A \rightarrow X_1 ... X_{i-1} B \cdot \gamma] & val'' \\ \hline \end{array}$$

val'' erweitert val um die synthetischen Attribute von B

$$val''(\alpha.i) = \begin{cases} \widehat{val}(t) & \text{, falls } \alpha.0 = t \in E_{B \rightarrow \beta} \\ val(\alpha.i) & \text{, sonst} \end{cases}$$

<u>Anwendung von LAG</u>

Überprüfung der Deklariertheit von Bezeichnern.

<u>**Beispiel**</u> Sei G gegeben durch

$P \rightarrow \underline{DL}; \underline{SL}$	Programm
$\underline{DL} \rightarrow V \| V; \underline{DL}$	Deklarationsliste
$\underline{SL} \rightarrow S \| S; \underline{SL}$	Statementliste
$V \rightarrow a \| b \| \ldots$	Variablen
$S \rightarrow V \underline{:=} V$	Statement

```
a;b;a;c;c:=a;d:=b;...
```

Attribute

syn.	v	deklarierte/benutzte Variable	V	$\in \{a, b, \ldots\}$
syn.	dv	deklarierte Variablen	DL	$\subseteq \{a, b, \ldots\}$
inh.	env	Umgebung	S, SL	$\subseteq \{a, b, \ldots\}$
syn.	$decl$	korrekt deklariert?	S, SL, P	$\in \{true, false\}$

Attributgleichungen

$P \to \underline{DL}; \underline{SL}$	$decl.0 = decl.3$	$\underline{SL} \to S; \underline{SL}$	$decl.0 = decl.1$ AND $decl.3$
	$env.3 = dv.1$		$env.1 = env.0$
$\underline{DL} \to V$	$dv.0 = \{v.1\}$		$env.3 = env.0$
$\underline{DL} \to V; \underline{DL}$	$dv.0 = \{v.1\} \cup dv.3$	$V \to a\|b\| \ldots$	$v.0 = a\| \ldots$
$\underline{SL} \to S$	$env.1 = env.0$	$S \to V \underline{:=} V$	$decl.0 = v.1 \in env.0$
	$decl.0 = decl.1$		AND $v.3 \in env.0$

Index