Compilerbau SS2004

Prof. Indermark
GeTeXt von
Michael Neuendorf

July 27, 2004

Dieses Skript ist eine Abschrift meiner Vorlesungsmitschriften der Vorlesung Compilerbau im Sommersemester 2004 und entbehrt sicherlich nicht einiger Tippfehler. Es schafft hoffentlich eine leserliche Übersicht über die behandelten Themen. Leider fehlen zur Zeit noch die Grafiken, ich hoffe sie aber noch bald ergänzen zu können.
Da ich Fehler im Skript möglichst ausmerzen möchte, bitte ich darum, entdeckte Fehler per eMail an *michael.neuendorf@rwth-aachen.de* zu schicken.

Michael Neuendorf

1 Einleitung

Compiler = Programm zur Übersetzung von PS-Programm (Quellsprache) in äquivalente MS-Programme (Zielsprache)

PS: imperativ, deklarativ (funktional, logisch), nebenläufig, objekt-orientiert

MS: Maschinensprache, von-Neumann-Rechner

- RISC (reduced instruction set computer)
- CISC (complex instruction set computer)

Aspekte einer PS

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

Äquivalenz von Programmen: semantische Gleichheit

1.1 Struktur eines Compilers

logisch unabhängige Phasen, die aber verzahnt als Passes (Läufe) ablaufen:

- Analyse: Bestimmung der syntaktischen Struktur, Fehlerbestimmung

  lexikalische Analyse: Erkennung von Symbolen, Trennzeichen und Kommentaren (reguläre Ausdrücke, endliche Automaten)

  syntaktische Analyse: Erkennung des hierarchischen Programmaufbaus, Ableitungsbaum (CFG, Kellerautomaten)

  semantische Analyse: Kontextabhängigkeiten, statische Semantik, Typinformation, Attributierung des Ableitungsbams mit semantisch relevanten Informationen (attributierte Grammatiken)

- Synthese: Erzeugung von MS-Code aus attributiertem Ableitungsbaum
  - Übersetzung in Zwischencode für eine abstrakte Maschine

- Optimierung: Verbesserung von Laufzeit und Speicherbedarf
- Codegenerierung: effiziente Verwendung von Registern und MS-Befehlssatz zur Erzeugung von MS-Code

Frontend (MS-unabhängig) Analyse, Zwischencodeerzeugung und Optimierung

Backend (MS-abhängig) MS-Code-Erzeugung und Optimierung

One-Pass, n-Pass-Compiler Zahl der Läufe durch das Quellprogramm

2 Lexikalische Analyse

Ausgangspunkt Quellprogramm P als Zeichenfolge

- Σ_0 Zeichensatz (z.B. ASCII), $a \in \Sigma_0$ Zeichen, lexikalisches Atom (Lexem)
- $P \in \Sigma_0^*$ besitzt aufgrund der Pragmatik von PS eine lexikalische Struktur

Pragmatischer Aspekt

- natürliche Sprache für Bezeichner und Schlüsselwörter
- mathematische Formelsprache für Zahlen und Formeln
- Leerzeichen, Zeilenwechsel, Einrücken, ...
- Kommentare

$\rightsquigarrow$ gute Lesbarkeit und Wartbarkeit
Die Semantik von P und damit die Übersetzung von P ist syntaxorientiert: Sie folgt dem hierarchischen Aufbau. Dabei sind pragmatische Aspekte irrelevant.

1. Beobachtung Syntaktische Atome (Symbole) werden dargestellt als Folgen lexikalischer Atome, sogenannter Lexeme.

1. Aufgabe der lexikalischen Analyse Zerlegung des Quellprogrammes P in eine Folge von Lexemen.

2. Beobachtung Für die syntaktische Analyse ist der Unterschied zwischen Lexem oft irrelevant, z.B. müssen Bezeichner nicht unterschieden werden.

Symbolklassen Lexeme werden zu Symbolklassen zusammengefaßt. Darstellung einer Symbolklasse: Token (z.B. (id, num))
Die syntaktische Analyse bearbeitet eine Tokenfolge, Identifizierung eines Symbols durch zusätzliches Attribut für semantische Analyse und Codegenerierung.

$$Symbol = (Token, Attribut)$$

2. Aufgabe der lexikalischen Analyse Transformation einer Lexem-Folge in eine Symbolfolge.

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

Scanner (Lexer): Programm für die lexikalische Analyse

2.1 Wichtigste Symbolklassen (eigentlich Lexemklassen)

- Bezeichner
- Zahlwörter
- Schlüsselwörter
- einfache Symbole: ein Sonderzeichen, z.B. +, -, (, bildet jeweils eine Symbolklasse
- zusammengesetzte Symbole: Folgen von zwei oder mehr Zeichen
- Leerzeichen: $\sqcup, \sqcup \ldots \sqcup$
- Spezielle Symbole: Kommentare, Pragmas wie Compileroptionen

Token: id, const, divsym, semsym

Attribute: Zeiger in Symboltabelle, Binärdarstellung einer Zahl, leer bei Symbolklassen mit (nur) einem Symbol

Feststellung: Symbolklassen sind reguläre Mengen

$\rightsquigarrow$ Beschreibung durch reguläre Ausdrücke, Erkennung durch endliche Automaten

2.2 Scanner-Konstruktion

Definition (reguläre Ausdrücke) Für ein Alphabet Σ ist die Menge $\mathrm{RegE}(\Sigma)$ der regulären Σ-Ausdrücke definiert durch

- $\Lambda \in \mathrm{RegE}(\Sigma), \Sigma \subseteq \mathrm{RegE}(\Sigma)$
- $\alpha, \beta \in \mathrm{RegE}(\Sigma) \curvearrowright (\alpha \vee \beta), (\alpha \wedge \beta), (\alpha^*) \in \mathrm{RegE}(\Sigma)$

Ihre Semantik:

- $[\![.]\!] : \mathrm{RegE}(\Sigma) \to P(\Sigma^*)$
- $[\![\Lambda]\!] : \emptyset$
- $[\![a]\!] := \{a\}$
- $[\![(\alpha \vee \beta)]\!] := [\![\alpha]\!] \cup [\![\beta]\!]$
- $[\![(\alpha \cdot \beta)]\!] := [\![\alpha]\!] \cdot [\![\beta]\!]$
- $[\![(\alpha^*)]\!] := [\![\alpha]\!]^*$

2.3 Das einfache Matching-Problem

Entscheide für $\alpha \in \text{RegE}(\Sigma)$ und $w \in \Sigma^*$, ob $w \in [\![\alpha]\!]$ oder $w \notin [\![\alpha]\!]$.

2.3.1 Hilfsmittel: endliche Automaten

$\mathfrak{A} = \langle Q, \Sigma, \delta, q_0, F\rangle \in \text{NFA}(\Sigma)$ mit $\delta : Q \times \Sigma_\varepsilon \to P(Q)$, $\Sigma_\varepsilon := \Sigma \cup \{\varepsilon\}$ und $q_0 \in Q, F \subseteq Q$.
Für $T \subseteq Q$ ist die ε-Hülle $\varepsilon(T)$ definiert durch

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

2.3.2 Die erweiterte Transitionsfunktion

$\bar{\delta} : P(Q) \times \Sigma^* \to P(Q)$ ist definiert durch

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

$\mathfrak{A}$ erkennt die Sprache $L(\mathfrak{A}) := \{w \in \Sigma^* | \bar{\delta}(\{q_0\}, w) \cap F \neq 0\}$.

2.3.3 Die DFA-Methode zur Lösung des Matching-Problems

(bekannt aus ATFS)
$\alpha \in \text{RegE}(\Sigma) \overset{(1)}{\longmapsto} \mathfrak{A}(\alpha) \in \text{NFA}(\Sigma) \overset{(2)}{\longmapsto} \mathfrak{A}(\alpha)^P \in \text{DFA}(\Sigma)$ mit $[\![\alpha]\!] = L(\mathfrak{A}(\alpha)) = L(\mathfrak{A}(\alpha)^P)$.

- (1) Methode von Thompson (Satz von Kleene) (linearer Platz- und Zeitbedarf)
- (2) Potenzmengenkonstruktion (exponentieller Zeit- und Platzbedarf)

Aber $\mathfrak{A}(\alpha)^P$ löst das Matching-Problem ($w \in [\![\alpha]\!]$) in $(|w| + 1)$ Schritten.

2.3.4 Die NFA-Methode zur Lösung des Matching-Problems

Verkleinerung des Platzbedarf, jedoch längere Laufzeit durch den Verzicht auf die Potenzmengenkonstruktion.

Idee Berücksichtigung der Eingabe $w \in \Sigma^*$, Potenzmengenkonstruktion nur für den "Lauf von w durch $\mathfrak{A}(\alpha)$".

Komplexität (NFA-Methode)

- Platz: $O(|\alpha| + |w|)$
- Zeit: $O(|\alpha| \cdot |w|)$

Kombination von NFA- und DFA-Methode Zwischenspeichern von bereits berechneten Transitionen $\bar{\delta}(T, a)$ im Cache.

Das erweiterte Matching-Problem Seien $\alpha_1, \ldots, \alpha_n \in \mathrm{RegE}(\Sigma)$ und $w \in \Sigma^*$. Sei ferner $\Delta := \{T_1, \ldots, T_n\}$ ein Alphabet von Token. Wenn $w = w_1 w_2 \ldots w_k$ und $w_j \in [\![\alpha_{i_j}]\!]$ mit $1 \leq i_j \leq n$ für $j = 1, \ldots, j$, dann heißt $w_1, w_2, \ldots, w_k$ eine Zerlegung von w bezüglich $\alpha_1, \ldots, \alpha_n$ und $v = T_{i_1} \ldots t_{i_k}$ eine Analyse von w bezüglich $\alpha_1, \ldots, \alpha_n$.
Hinweis: Mehrdeutigkeit

Konventionen für Eindeutigkeit

1. Das Prinzip des längsten Match (max. munch) für die Eindeutigkeit der Zerlegung
   Eine Zerlegung $(w_1, \ldots, w_k)$ von w bezüglich $\alpha_1, \ldots, \alpha_n$ heißt lm-Zerlegung, wenn für alle $j = 1, \ldots, k$, $x, y \in \Sigma^*$ und $p, q \in \{1, \ldots, n\}$ gilt:

   $$w = w_1 w_2 \ldots w_j xy, \qquad w_j \in [\![\alpha_p]\!], \qquad w_j x \in [\![\alpha_q]\!] \curvearrowright x = \varepsilon$$

   Folgerung: Für $w, \alpha_1, \ldots, \alpha_n$ gibt es höchstens eine lm-Zerlegung.

2. Prinzip des ersten Matches (bezüglich $\alpha_1, \ldots, \alpha_n$ für die Eindeutigkeit der Analyse.
   Trotz der Eindeutigkeit der lm-Zerlegung kann es mehrere zugehörige Analysen geben, weil $[\![\alpha_p]\!] \cap [\![\alpha_q]\!] \neq \emptyset$ möglich ist.
   Konvention: Erster Match in einer Reihe $\alpha_1, \ldots, \alpha_n$ zählt.
   Sei $(w_1, \ldots, w_l)$ eine lm-Zerlegung und $v = T_{i_1} \ldots T_{i_k}$ eine (zugehörige) Analyse von w bezüglich $\alpha_1 \ldots \alpha_n$. Dann heißt v eine flm-Analyse (first longest match) falls für alle $j = 1, \ldots, k$ und $v = 1, \ldots, n$ gilt:

   $$w_j \in [\![\alpha_v]\!] \curvearrowright i, j \leq v$$

Das erweiterte Matching-Problem

1. Existiert für $w \in \Sigma^*$ bezüglich $\{\alpha_1 \ldots \alpha_n\} \subseteq \mathrm{RegE}(\Sigma)$ eine lm-Zerlegung ?

2. Berechnung einer flm-Analyse
   $\Delta = \{T_1, \ldots, T_n\}$ Tokenalphabet

   a) Kontruiere für $i = 1, \ldots, n$
   $\mathfrak{A}_i = \langle Q_i, \Sigma, \delta_i, q_0^{(i)}, F_i \rangle \in \mathrm{DFA}(\Sigma)$ mit $[\![\alpha_i]\!] = L(\mathfrak{A}_i)$.

   b) Bilde aus diesen den Produktautomaten
   $\mathfrak{A} = \langle Q, \Sigma, \delta, q_0, F \rangle \in \mathrm{DFA}(\Sigma)$ mit
   - $Q = Q_1 \times \ldots \times Q_n$,
   - $q_0 := (q_0^{(1)}, \ldots, q_0^{(n)})$,
   - $\delta(q^{(1)}, \ldots, q^{(n)}) := (\delta_1(q^{(1)}, a), \ldots, \delta_n(q^{(n)}, a))$ und

- $(q^{(1)}, \ldots, q^{(n)}) \in F :\leftrightarrow \exists i \in \{1, \ldots, n\} q^{(i)} \in F$

Dann gilt $L(\mathfrak{A}) = \bigcup_{i=1}^{n} [\![\alpha_i]\!]$.

Zerlege F wegen "first match" in $F = \bigcup_{i=1}^{n} f^{(i)}$. $(q^{(1)}, \ldots, q^{(n)}) \in F^{(i)} :\leftrightarrow$ $q^{(i)} \in F_i$ und $q^{(j)} \notin F_j$ f+r alle $1 \leq j \leq i$. Dann gilt: $\bar{\delta}(q_0, w) \in F^{(i)} \leftrightarrow w \in$ $[\![\alpha_i]\!]$ und $w \notin \bigcup_{j=1}^{i-1} [\![\alpha_j]\!]$.

$q \in Q$ heißt *produktiv*, falls es ein $v \in \Sigma^*$ gibt, so daß $\bar{\delta}(q, v) \in F$. P ist die Menge der produktiven Zustände, also $F \subseteq P$.

c) Erweitere $\mathfrak{A} = \langle Q, \Sigma, \delta, q_0, F \rangle$ zu einem Backtrack-DFA $\mathfrak{B}$ mit Ausgabe.
Idee: Einweg-Leseband mit 2 Köpfen

- ein Backtrack-Kopf b zur Markierung eines Matches
- ein Lookahead-Kopf l zur Bestimmung des längsten Matches

Konfigurationsmenge von $\mathfrak{B}$: $\{N\} \cup \Delta) \times \Sigma^* Q \Sigma^* \times \Delta^* \{\varepsilon, \text{lexerr}\}$
Ausgangskonfiguration für $w \in \Sigma^*$: $(N, q_0 w, \varepsilon)$
Transitionen: $q' := \delta(q, a)$

i. normal mode: match suchen

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

ii. backtrack mode: längsten match suchen

$$(T, vqaw, W) \vdash \begin{cases} (T, vaq'w, W) & \text{falls } q' \in P \setminus F \\ (T_i, q'w, W) & \text{falls } q' \in F^{(i)} \\ (N, q_0 vaw, WT) & \text{falls } q' \notin P \end{cases}$$

iii. Eingabeende
- $(N, q, W) \vdash$ Ausgabe: Wlexerr, falls $q \in P \setminus F$
- $(T, q, W) \vdash$ Ausgabe: WT, falls $q \in F$
- $(T; vaq, W \vdash (N, q_0 va, WT)$, falls $q \in P \setminus F$

Dann gilt für $w \in \Sigma^*$:

- $(N, q_0 w, \varepsilon) \overset{*}{\vdash} W \in \Delta^* \leftrightarrow W$ ist flm-Analyse von w
- $(N, q_0 w, \varepsilon) \overset{*}{\vdash} W \cdot \text{lexerr} \leftrightarrow$ es gibt keine flm-Analyse von w

Zeitaufwand: $O(|w|^2$ im worst case, Verbesserung durch Tabular-Methode möglich zu Linearzeit. Literatur: Th. Reps "Maximal Munch Tokenization in Linear Time", ACM-TOPLAS 20(1998).

2.3.5 Praktische Aspekte der Scannerkonstruktion

- Unterschiede
  - automatische Scannererzeugung (lex u.a.) mit beliebigen $alpha_1 \dots \alpha_n$
  - Scanner-Konstruktion für eine Programmiersprachen mit speziellen $\alpha_1 \dots \alpha_n$ (Effizienz wichtig, häufiger Aufruf durch nexttoken, direkte Programmierung [Assembler])
  - Spracherweiterung von RegE(Σ)
    1. Vereinfachende Bezeichnungen
       - Präzedenzregeln zur Vermeidung von Klammern: $*$ bindet stärker als $\cdot$ bindet stärker als $\vee$, $\cdot$ und $\vee$ sind assoziativ (Linksklammerung), $\cdot$ weglassen, $|$ statt $\vee$
       - Beispiel: $a|b*c$ statt $(a \vee ((b^*) \cdot c))$
       - Abkürzungen:

$$\begin{array}{lcll} a^+ & := & aa^* & \\ a? & := & a|\Lambda^* & \\ \cdot & := & a|\dots|z & (\Sigma = \{a, \dots, z\}) \\ [abc] & := & a|b|c & \\ [0-9] & := & 0|1|\dots|9 & \end{array}$$

    2. Reguläre Definitionen
       Schrittweise Beschreibung von Symbolklassen (regulären Mengen) durch zusätzliche Bezeichner $id_1 = \alpha_1, \dots, id_n = \alpha_n$ mit $id_1, \dots, id_n \notin \Sigma$ und $\alpha_i \in \text{RegE}(\Sigma \cup \{id_1, \dots, id_{i-1}\})$
       Beachte: keine Rekursion, Entschachtelung möglich (Rekursion $\curvearrowright$ EBNF, ECFG)

- Prinzip des längsten Matches
  - Allgemein: beliebig langer "look-ahead" (backtrack-Phase) erforderlich
  - PS(Pascal): look-ahead um 2 Zeichen
  - aber: Beschränkung kann die lexikalische Analyse verändern
  - Abweichung von lm-Prinzip möglich:
    - lex: α/β) Match von α erfordert zusätzlich einen lookahead-String, der β matcht
  - Besondere Rolle von "blanks" (white space): Trennung von "eigentlichen" Lexem, um mit 1-look-ahead auszukommen
- Attributberechnung
  Nach Erkennen eines Lexems erneutes Lesen:
  - Dezimalzahl $\rightsquigarrow$ Binärzahl (lex: `install_num`)
  - Bezeichner: Gültigkeit prüfen, Symboltabelle (lex: `install_id`)

- Schlüsselwörter (alternativ): zunächst als Bezeichner behandeln, Symboltabelle mit Schlüsselwörtern initialisieren, Attributberechnung (`install_id`) liedert das Schlüsselworttoken

- Automatische Scannergenerierung mit Lex
  - Unixtool: `man flex` ("fast lexical analyzer") oder `info flex`
  - `scan.l` $\overset{lex}{\longmapsto}$ `lex.yy.c` $\overset{C-Compiler}{\longmapsto}$ `a.out`
  - Programm `a.out` Tokenfolge mit Attributbenennung
  - Aufbau eine Lex-Spezifikation

```
Definitionen (optional)

Regeln

C-Hilfsprozeduren (optional)
```

Definitionen 1. Direkter C-Code %{... C-Code ...%}
2. Substitutionen (reguläre Definitionen) `NAME regexp...`
3. Startzustände (zur Einleitung in besondere Phasen

Regeln muster {aktion}

muster von der Form `regexp [/regexp]`

aktion C-Code zur Berechnung von `<Token, Attribut>`

Finden der passenden Regel $*$ longest match
$*$ first match
$*$ kein match: abfangen mit
`.|\n`
$\rightsquigarrow$ Fehlerbehandlung

Attribute Weitergabe über globale Variable `yylval`

3 Syntaktische Analyse

Aufgabe: Zerlegung der Symbolfolge, die der Scanner ausgibt, in syntaktische Einheiten oder Behandlung syntaktischer Fehler

3.1 Syntaktische Einheiten

Variablen, Ausdrücke, Anweisungen ...
Beachte: Schachtelung syntaktischer Einheiten, Baumstruktur im Unterschied zur linearen Symbolfolge

Beschreibung der syntaktischen Struktur, kontextfreie Grammatiken (BNF, EBNF, Syntaxdiagramme)

Problem: deterministische Simulation

Allgemeiner Fall: beliebige CFG, Lösung durch das Tabularverfahren von Cocke, Younger, Kasami mit $O(n^2)$ Platz- und $O(n^3)$ Zeitbedarf

Programmiersprachen: spezielle CFG, Analyse durch deterministische Kellerautomaten mit "Input-Lookahead" bei linearem Platz- und Zeitbedarf

1. Top-Down-Analyse: Konstruktion des Ableitungsbaumes von der Wurzel zu den Blättern in Form einer Linksanalyse
2. Bottom-Up-Analyse: umgekehrt, gespiegelte Rechtsanalyse

Kontextfreie Grammatiken $\mathcal{G} = \langle N, \Sigma, P, S \rangle \in \mathrm{CFG}(\Sigma)$

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

Table 3.1: Bezeichnungskonventionen

Ableitungsrelation: $\Rightarrow \subseteq (\mathfrak{X}^*)^2$ definiert durch

$$\alpha \Rightarrow \beta :\leftrightarrow \begin{array}{ll} \alpha = \alpha_1 A \alpha_2 & A \mapsto \gamma \in P \\ \beta = \alpha_1 \gamma \alpha_2 & \end{array}$$

Gilt außerdem $\alpha_1 \in \Sigma^*$ bzw. $\alpha_2 \in \Sigma^*$, so

$$\alpha \underset{l}{\Rightarrow} \text{ bzw. } \alpha \underset{r}{\Rightarrow} \beta$$

Sprechweise: Ableitungsschritt, Linksableitungsschritt, Rechtsableitungsschritt

Erzeugte Sprachen

$$\begin{array}{rcl} L(\mathcal{G}) & := & \{w \in \Sigma^* | S \overset{*}{\Rightarrow} w\} \\ & = & \{w \in \Sigma^* | S \underset{l}{\overset{*}{\Rightarrow}} w\} \\ & = & \{w \in \Sigma^* | S \underset{r}{\overset{*}{\Rightarrow}} w\} \end{array}$$

Beispiel: $\mathcal{G} : S \mapsto aSb|\varepsilon$

$$S \Rightarrow aSb \Rightarrow aaSbb \overset{*}{\Rightarrow} a^n S b^n \Rightarrow a^n \cdot \varepsilon \cdot b^n \Rightarrow a^n b^n$$

Ableitungsbläume repräsentieren die syntaktische Struktur des abgeleiteten Wortes a^2b^2.

Definition Eine Grammatik $\mathcal{G}$ heißt eindeutig, wenn es für jedes $w \in L(\mathcal{G})$ genau einen Ableitungsbaum gibt.
Folgerung: $\mathcal{G}$ ist mehrdeutig, wenn es für jedes $w \in L(\mathcal{G})$ genau eine Linksableitung (bzw. Rechtsableitung) gibt.

Definition $\mathcal{G}$ heißt mehrdeutig, wenn $\mathcal{G}$ nicht eindeutig ist.

l-Analyse, r-Analyse Darstellung von l/r-Ableitungen durch Nummernfolgen:

$|P| = p \quad [p] := \{1, \ldots, p\}$

$\pi : [p] \to P$ bijektiv

Sei $i \in [p]$:

$$wA\alpha \underset{l}{\overset{i}{\Rightarrow}} w\gamma\alpha \text{ bzw. } \alpha A w \underset{r}{\overset{i}{\Rightarrow}} \alpha\gamma W \text{ falls } \pi_i := \pi(i) = A \mapsto \gamma$$

Für $z = i_1 \ldots i_n \in [p]^+$ soll $\alpha \underset{l}{\overset{z}{\Rightarrow}} :\leftrightarrow \alpha \underset{l}{\overset{i_1}{\Rightarrow}} \alpha_1 \underset{l}{\overset{i_1}{\Rightarrow}} \alpha_2 \ldots \underset{l}{\overset{i_n}{\Rightarrow}} \alpha_n = \beta$ und $\alpha \underset{l}{\overset{\varepsilon}{\Rightarrow}} \alpha$ (ein "leerer" Ableitungsschritt) gelten.

Definition Sei $z \in [p]^*$. z heißt l-Analyse von $\alpha : S \underset{l}{\overset{z}{\Rightarrow}} \alpha$. Analog: r-Analyse.

Syntaxanalyse Gegeben: $\mathcal{G} \in \mathrm{CFG}(\Sigma)$ und $w \in \Sigma^*$. Berechnung einer Links- bzw. Rechtsanalyse, falls $w \in L(\mathcal{G})$. Andernfalls: Bestimmung syntaktischer Fehler.

Generalvoraussetzung $G \in \mathrm{CFG}$ ist reduziert, d.h. für jedes $A \in N$ gibt es $\alpha, \beta \in \mathfrak{X}^*$ und $w \in \Sigma^*$, sodaß

- $S \overset{*}{\Rightarrow} \alpha A \beta$ (A erreichbar) und
- $A \overset{*}{\Rightarrow} w$ (A produktiv).

3.1.1 Top-Down-Analyse, $LL(k)$-Grammatiken

Definition Sei $\mathcal{G} \in \mathrm{CFG}(\Sigma)$. Der (nicht-deterministische Top-Down-Analyseautomat von $\mathcal{G}$ (Bezeichnung: $\mathrm{NTA}(\mathcal{G})$

- Eingabealphabet: Σ
- Kelleralphabet: $\mathfrak{X} = N \cup \Sigma$
- Ausgabealphabet: $[p] = \{1, \ldots, p\}$
- ("nur 1 Zustand")
- Bild (einfügen)
- Konfigurationsmenge: $\Sigma^* \times \mathfrak{X}^* \times [p]^*$
- Transitionen:
  - Ableitungsschritt: $(w, A\alpha, z) \vdash (w, \beta\alpha, z_i)$, falls $\pi_i = A \mapsto \beta$
  - Vergleichsschritt: $(aw, a\alpha, z) \vdash (w, \alpha, z)$, für $a \in \Sigma$
- Anfangskonfiguration für $w \in \Sigma : (w, S, \varepsilon)$
- Endkonfigurationen: $(\varepsilon, \varepsilon, z)$
- Beachte: Nicht-Determinismus wegen $A \mapsto \beta | \gamma$

Satz Der $\mathrm{NTA}(\mathcal{G})$ berechnet l-Analysen, d.h. $(w, S, \varepsilon) \overset{*}{\longmapsto} (\varepsilon, \varepsilon, z) \leftrightarrow$ z ist l-Analyse von w.
Beweis:

- $''\curvearrowright''$ Automat arbeitet korrekt
  $(*)(w, \alpha, y) \overset{*}{\longmapsto} (\varepsilon, \varepsilon, yz) \curvearrowright \alpha \overset{z}{\underset{l}{\Rightarrow}} w$
  Beweis von $(*)$ durch Induktion über $k = |z|$.
  1. $k = 0$: Nur Vergleichsschritte, $w = \alpha$, also $(*)$ mit $w \overset{\varepsilon}{\underset{l}{\Rightarrow}} w$.

2. $k \to k+1$: $z = iz', \alpha = wA\beta, w = uv, \pi_i = A \to \gamma$
   $(w, \alpha, y) = (uv, uA\beta, y) \overset{*}{\longmapsto} (v, A\beta, y) \longmapsto (v, \gamma\beta, yi) \overset{*}{\longmapsto} (\varepsilon, \varepsilon, yiz')$
   Nach Induktionsvoraussetzung: $\gamma\beta \underset{l}{\overset{z'}{\Rightarrow}} v$. Damit folgt $(*)$:

$$\alpha = uA\beta \underset{l}{\overset{i}{\Rightarrow}} u\gamma\beta \underset{l}{\overset{z'}{\Rightarrow}} uv = w$$

   q.e.d.

Vorüberlegung $3 : A \to a\beta \qquad 4 : A \to b\beta$
$(aw, A\alpha, z) \longmapsto (aw, a\beta\alpha, z3) \longmapsto (w, \beta\alpha, z3)$
$A \to BC \qquad B \to \varepsilon \qquad C \to \varepsilon$

Ziel: Nicht-Determinismus des NTA($\mathcal{G}$) durch k-look-ahead auf der Eingabe beseitigen, $k \in \mathbb{N}$.

Definition (first-Mengen): Sei $\mathcal{G} \in \text{CFG}, \alpha \in \mathfrak{X}^*$ und $k \in \mathbb{N}$. Wir definieren $first_k(\alpha) \subseteq \Sigma^*$ durch

$$\begin{aligned} first_k(\alpha) &:= \{v \in \Sigma^* | \exists w \in \Sigma^* : \alpha \overset{*}{\Rightarrow} vw, |v| = k\} \\ &\cup \quad \{v \in \Sigma^* | \alpha \overset{*}{\Rightarrow} v, |v| < k\} \end{aligned}$$

Folgerungen (Übung)

1. $first_k(\alpha) \neq \emptyset$, weil $\mathcal{G}$ reduziert ist.
2. $\varepsilon \in first_k(\alpha) \leftrightarrow k = 0$ oder $\alpha \overset{*}{\Rightarrow} \varepsilon$
3. $\alpha \overset{*}{\Rightarrow} \beta \curvearrowright first_k(\beta) \leq first_k(\alpha)$
4. $v \in first_k(\alpha) \leftrightarrow \exists x \in \Sigma^* : \alpha \underset{l}{\overset{*}{\Rightarrow}} x, \{v\} = first_k(x)$

$LL(k)$: Lesen der Eingabe von links nach rechts mit k-look-ahead, Berechnung einer l-Analyse

$LL(k)$-**Grammatik** Sei $\mathcal{G} \in \text{CFG}$ und $k \in \mathbb{N}$.
$\mathcal{G} \in LL(k) :\leftrightarrow$ für alle Linksableitungen der Form

$$S \underset{l}{\overset{*}{\Rightarrow}} wA\alpha \begin{cases} \underset{l}{\Rightarrow} w\beta\alpha \underset{l}{\overset{*}{\Rightarrow}} wx \\ \underset{l}{\Rightarrow} w\gamma\alpha \underset{l}{\overset{*}{\Rightarrow}} wy \end{cases}$$

gilt: $first_k(x) = first_k(y) \curvearrowright \beta = \gamma$.
Bemerkung:

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

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

Lemma $\mathcal{G} \in LL(k) \leftrightarrow$ für alle Linksableitungen der Form $S \overset{*}{\underset{l}{\Rightarrow}} wA\alpha \begin{cases} \underset{l}{\Rightarrow} w\beta\alpha \\ \underset{l}{\Rightarrow} w\gamma\alpha \end{cases}$ gilt:

$$\beta \neq \gamma \curvearrowright first_k(\beta\alpha) \cap first_k(\gamma\alpha) = \emptyset$$

Beweis:

1. "Definition" $\curvearrowright$ "Lemma"
   $\beta \neq \gamma$, aber $v \in first_k(\beta\alpha)$ und $v \in first_k(\gamma\alpha)$
   $\curvearrowright$ es existiert $\beta\alpha \overset{*}{\underset{l}{\Rightarrow}} x$ und $\gamma\alpha \overset{*}{\underset{l}{\Rightarrow}} y$ mit $\{v\} = first_k(x) = first_k(y)$
   $\curvearrowright$ (nach "Definition") $\beta = \gamma$, Widerspruch!
2. "Lemma" $\curvearrowright$ "Definition"
   "Lemma" gilt, $first_k(X) = first_k(y)$, aber angenommen $\beta \neq \gamma$:

   $$first_k(x) \subseteq first_k(\beta\alpha)$$

   $$first_k(y) \subseteq first_k(\gamma\alpha)$$

   Widerspruch zum Lemma!

q.e.d.

Folgerung: Bestimmung der A-Regel durch die look-ahead-Mengen

$$first_k(\beta\alpha), first_k(\gamma\alpha), \ldots \text{ für } (A \to \beta|\gamma| \ldots)$$

Problem Abhängigkeit der look-ahead-Mengen vom Rechtskontext α

Ziel Bestimmung der look-ahead-Menge aus der Regel allein

Idee Mögliche Rechtskontexte vereinigen

Definition (follow-Menge) Sei $\mathcal{G} \in \text{CFG}, A \in N$ und $k \in \mathbb{N}$. Wir definieren $follow_k(A) \subseteq \Sigma^*$ durch

$$follow_k(A) := \{v \in \Sigma^* | S \overset{*}{\underset{l}{\Rightarrow}} wA\alpha, v \in first_k(\alpha)\}$$

Bemerkung: In $follow_k(A)$ werden alle möglichen Rechtskontexte berücksichtigt.

Der Fall $k = 1$ Im allgemeinen genügt $k = 1$. $k > 1$ aufwendiger (Hinweis: ANTLR)

Abkürzungen $F_i := first_1$ und $\text{fo} := follow_1$

Satz Für $\mathcal{G} \in \text{CFG}$ ($\mathcal{G}$ reduziert) gilt: $\mathcal{G} \in LL(1) \leftrightarrow$ Für alle Regelpaare $A \to \beta|\gamma$ folgt: $\text{fi}(\beta\text{fo}(A)) \cap \text{fi}(\gamma\text{fo}(A)) = \emptyset$.

Definition $la(A \to \beta) := \text{fi}(\beta\text{fo}(A))$ heißt look-ahead-Menge von $A \to \beta$ (la-Menge).

Beachte:

- $\text{fi}(\alpha) \subseteq \Sigma_\varepsilon$
- $\text{fo}(A) \subseteq \Sigma_\varepsilon$
- $\beta\text{fo}(A) \subseteq \mathfrak{X}^*$
- für $\Gamma \subseteq \mathfrak{X}^*$ ist $\text{fi}(\Gamma) := \bigcup_{\alpha \in \Gamma} \text{fi}(\alpha)$
- $la(A \to \beta) = \text{fi}(\beta\text{fo}(A) \subseteq \Sigma_\varepsilon$
- $\varepsilon \in \text{fi}(\beta\text{fo}(A)) \leftrightarrow \beta \overset{*}{\Rightarrow} \varepsilon$ und $\varepsilon \in \text{fo}(A)$
- $a \in \text{fi}(\beta\text{fo}(A)) \leftrightarrow a \in \text{fi}(\beta)$ oder $(\beta \overset{*}{\Rightarrow} \varepsilon$ und $a \in \text{fo}(A))$

Beweis:
"$\curvearrowright$": Angenommen, es gibt $A \to \beta|\gamma$ mit $\beta \neq \gamma$ und $c \in \text{fi}(\beta\text{fo}(A)) \cap \text{fi}(\gamma\text{fo}(A))$ für ein $c \in \Sigma_\varepsilon$, dann treten folgende Fälle auf:

1. $c = \varepsilon \curvearrowright \beta \overset{*}{\Rightarrow} \varepsilon, \gamma \overset{*}{\Rightarrow} \varepsilon, \varepsilon \in \text{fo}(A)$

   $$S \overset{*}{\underset{l}{\Rightarrow}} wA\alpha \begin{cases} \underset{l}{\Rightarrow} w\beta\alpha \\ \underset{l}{\Rightarrow} w\gamma\alpha \end{cases}$$

   mit $\alpha \overset{*}{\underset{l}{\Rightarrow}} \varepsilon \curvearrowright \varepsilon \in \text{fi}(\beta\alpha) \cap \text{fi}(\gamma\alpha)$ Widerspruch!

2. $c \in \Sigma$. Es folgt:

   a) $c \in \text{fi}(\beta) \cap \text{fi}(\gamma)$

   b) $c \in \text{fi}(\beta), \gamma \overset{*}{\Rightarrow} \varepsilon$ und $c \in \text{fo}(A)$

   c) $\beta \overset{*}{\Rightarrow} \varepsilon, c \in \text{fo}(A), c \in \text{fi}(\gamma)$

   d) $\beta \overset{*}{\Rightarrow} \varepsilon, \gamma \overset{*}{\Rightarrow} \varepsilon$ und $c \in \text{fo}(A)$

$\mathcal{G}$ ist reduziert $\curvearrowright$ es existieren Linksableitungen

$$S \overset{*}{\underset{l}{\Rightarrow}} wA\alpha \begin{cases} \overset{}{\underset{l}{\Rightarrow}} w\beta\alpha \\ \overset{}{\underset{l}{\Rightarrow}} w\gamma\alpha \end{cases}$$

mit $a \in \text{fi}(\beta\alpha) \cap \text{fi}(\gamma\alpha)$.

"$\curvearrowleft$": Sei $S \overset{*}{\underset{l}{\Rightarrow}} wA\alpha \begin{cases} \overset{}{\underset{l}{\Rightarrow}} w\beta\alpha \\ \overset{}{\underset{l}{\Rightarrow}} w\gamma\alpha \end{cases}$ mit $\beta \neq \gamma$. Nach Voraussetzung gilt $\text{fi}(\beta\text{fo}(A)) \cap \text{fi}(\gamma\text{fo}(A)) = \emptyset$. Da $\text{fi}(\beta\alpha) \subseteq \text{fi}(\beta\text{fo}(A))$, muß auch $\text{fi}(\beta\alpha) \cap \text{fi}(\gamma\alpha) = \emptyset$ sein.

q.e.d

Berechnung der la-Mengen

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

Lemma Die Mengen fi(α) für $\alpha \in \mathfrak{X}^*$ und fo(A) für $A \in N$ sind die kleinsten unter den Regeln (1) - (3) abgeschlossenen Teilmengen von Σ_ε.

Beispiel

$\mathcal{G}'_{AE}$: $E \to TE'$ (1)
$E' \to +TE'|\varepsilon$ (2,3)
$T \to FT'$ (4)
$T' \to *FT'|\varepsilon$ (5,6)
$F \to (E)|a$ (7,8)

$la(A \to \alpha) := \text{fi}(\alpha\text{fo}(A))$

Nichtterminalsymbole	E	E'	T	T'	F
fi	$(, a$	$+, \varepsilon$	$(, a$	$*, \varepsilon$	$(, a$
fo	$\varepsilon,)$	$\varepsilon,)$	$+, \varepsilon$	$+, \varepsilon,)$	$*, +, \varepsilon,)$

la-Mengen:

1	2	3	4	5	6	7	8
$(, a$	$+$	$\varepsilon,)$	$(, a$	$*$	$+, \varepsilon,)$	$($	a

Folgerung Die la-Mengen der Alternative (2,3), (5,6) und (7,8) sind disjunkt. Also ist $\mathcal{G}'_{AE}$ eine $LL(1)$-Grammatik.

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

Deterministischer TD-Analyseautomat DTA($\mathcal{G}$) **für** $\mathcal{G} \in LL(1)$

Idee: Die Zugehörigkeit des Eingabesymbols zu einer la-Menge steuert die Regelauswahl; 1-look-ahead auf der Eingabe
Modifikation des Ableitungsschrittes:

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

Folge: Deterministische Arbeitsweise

Beachte: Das Eingabesymbol wird bei Ableitungsschritten nicht gelöscht

Beispiel

$\vdots$
$A \to BC(1)$
$B \to bB|\varepsilon$
$C \to cC|\varepsilon$
$\vdots$

$$
\begin{array}{ll}
la(A \to BC) & = \text{fi}(BC\text{fo}(A)) \\
\text{fi}(B) & = \{b, \varepsilon\} \\
\text{fi}(C) & = \{c, \varepsilon\} \\
la(1) & = \{b, c\} \cup \text{fo}(A)
\end{array}
$$

action-Funktion Darstellung des $DTA(\mathcal{G})$ durch die Analysetabelle von $\mathcal{G}$, auch action-Funktion genannt.

$$\text{act} : \Sigma_\varepsilon \times (N \cup \Sigma_\varepsilon) \to \{\alpha | A \to \alpha \text{ in } \mathcal{G}\} \times [p] \cup \{pop, error, accept\}$$

$$
\begin{array}{ll}
\text{act}(x, A) & := (\alpha, i) \text{ falls } \pi_i : A \to \alpha \wedge x \in la(\pi_i) \\
\text{act}(a, a) & := pop \\
\text{act}(\varepsilon, \varepsilon) & := accept \\
\text{act}(x, X) & := error \text{ sonst}
\end{array}
$$

Parser-Konstruktion nach TD-Methode

- $\mathcal{G} \in$ CFG
- Berechnung der la-Mengen (fi- und fo-Mengen)
- Analysetabelle, Eindeutigkeit prüfen
- tabellengesteuerter Parser $\rightsquigarrow$ ANTLR
- Problem: Mehrdeutige Analysetabelle ($\mathcal{G} \notin LL(1)$)

Transformation nach $LL(1)$ Es gibt 2 Methoden, $\mathcal{G}$ in eine äquivalente $LL(1)$-Grammatik zu transformieren; dies ist jedoch nicht immer möglich.

1. Beseitigung von Linksrekursion
2. Linksfaktorisieren

Dazu werden parsererzeugende Systeme verwendet. Es ist jedoch Vorsicht geboten, da die Transformationen zwar die Äquivalenz erhalten, i.a. aber nicht die syntaktische Struktur.

1. Beseitigung von Linksrekursion

   Definition: $\mathcal{G} \in$ CFG heißt linksrekursiv: $\leftrightarrow \exists \forall A \in N, \alpha \in \mathfrak{X}^* : A \overset{+}{\Rightarrow} A\alpha$

   Folgerung: $\mathcal{G}$ linksrekursiv $\curvearrowright \mathcal{G} \notin LL(k)$ für alle $k \in \mathbb{N}$

   Grund: Wenn ein TD-Parser $A \overset{+}{\Rightarrow} A$ simuliert, so bleibt der Eingabekopf stehen, gleicher look-ahead. Schleife. Keine Ableitung der Form $S \underset{l}{\overset{*}{\Rightarrow}} wA\beta \underset{l}{\overset{+}{\Rightarrow}} wA\alpha\beta \underset{l}{\overset{*}{\Rightarrow}} wv$ simulierbar.

Beispiel:

$$\begin{array}{lll} \mathcal{G}_{AE}: & E \to E+T|T & (1,2) \\ & T \to T*F|F & (3,4) \\ & F \to (E)|a & (5,6) \end{array}$$

$$\text{fi}(F) = \{(, a\} = \text{fi}(T) = \text{fi}(E)$$

$$la(1) = \{(, a)\} = la(2) = la(3) = la(4)$$

Darauf folgt: $\mathcal{G}_{AE} \notin LL(1)$.

Spezialfall: direkte Linksrekursion und ihre Beseitigung
$A \to A\alpha|\beta$ mit $\beta \neq \alpha \ldots$ und $\alpha \neq \varepsilon$ wird ersetzt durch $A \to \beta A'$ mit neuem A' und $A' \to \alpha A'|\varepsilon$.

Folgerung: $L(\mathcal{G})$ ist unverändert, bekommt jedoch eine neue syntaktische Struktur. Dies ist kein Problem bei assoziativen Operatoren wie $*$ bzw. $+$.
Die Semantik ist invariant.

$$\begin{array}{ll} \mathcal{G}_{AE}: & E \to TE' \\ & E' \to +TE'|\varepsilon \\ & T \to FT' \\ & T' \to *FT'|\varepsilon \\ & F \to (E)|a \end{array}$$

Allgemeiner Fall: indirekte Linksrekursion

$$\begin{array}{l} A \to A_1\alpha_1| \ldots \\ A_1 \to A_2\alpha_2| \ldots \\ \vdots \\ A_n \to A\beta| \ldots \end{array}$$

Beseitigung durch Transformation in GNF (Greibach-Normalform)

$$\begin{array}{ll} \text{GNF}: & A \to aB_1 \ldots B_n \quad (B_i \neq S) \\ & S \to \varepsilon \end{array}$$

Beachte: Die Beseitigung von Linksrekursion ergibt nicht notwendigerweise $\mathcal{G}' \in LL(1)$.

Grund: Jedes $\mathcal{G} \in$ CFG ist äquivalent transformierbar in GNF, aber nicht jede $L \in CFL$ ist durch $\mathcal{G} \in LL(1)$ erzeugbar.

Es gilt: $\mathcal{L}(LL(k)) \subsetneq \mathcal{L}(LL(k+1)) \subsetneq \mathcal{L}(\text{DPDA}) \subsetneq \text{CFL}$

Komplexität der $LL(1)$-Analyse $\mathcal{G}LL(1)$-Grammatik $\curvearrowright \mathcal{G}$ nicht linksrekursiv DTA($\mathcal{G}$) mit Eingabe $w \in \Sigma^*$:

- $|w|$ Vergleichsschritte und 1 accept-Schritt
- Frage: Wieviele Schritte i kann der DTA($\mathcal{G}$) durchführen, ohne den Eingabekopf zu verschieben ? $wA\alpha \overset{i}{\underset{l}{\Rightarrow}} wB\beta$
  Aho/Ullman - The Theory Of Parsing,..., Vol. I, Parsing, S.356: $\exists c \in \mathbb{N}\mathcal{G}$ nicht linksrekursiv $\curvearrowright A \overset{i}{\underset{l}{\Rightarrow}} B\gamma \curvearrowright i < c = |N| \underset{falsch}{\curvearrowright} (S \overset{n}{\underset{l}{\Rightarrow}} w \curvearrowright n \leq$

$c(|w|+1))$
Besser: O. Mayer, Syntaxanalyse, S.48-55:
Satz: $\mathcal{G}$ nicht linksrekursiv: $\exists c \in \mathbb{N}\ S \overset{n}{\underset{l}{\Rightarrow}} w \curvearrowright n \leq c(|w|+1)$

2. Links-Faktorisierung:

   Beispiel:

statement	→ if cond then stat else stat fi	Keine Regelentscheidung
stat	→ if cond then stat fi	

   mit beschränktem look-ahead.

   Idee: Verschieben der Entscheidung bis Alternative erkennbar. Links-Faktorisieren:

A	$\rightarrow \alpha\beta \mid \alpha\gamma$	ersetzen durch
A	$\rightarrow \alpha A'$	und
A'	$\beta \mid \gamma$	

   Beispiel:

stat	→ if cond then stat S'
S'	→ else stat fi — fi

Top-Down-Analyse mit rekursiven Prozeduren

Idee: Keine explizite Kellerbenutzung wie im TDA($\mathcal{G}$), sondern Verwendung des Laufzeitkellers durch Einsatz rekursiver Prozeduren

Vorteil: Leichte Programmierung

Spezialfall: $\mathcal{G} \in LL(1)$ - Beispiel $\mathcal{G}'_{AE}$
"Recursive Descent Parser" (Analyse durch rekursiven Abstieg)

Methode: $A \in N \rightarrow A()$ Parameterlose Prozedur r (zur Simulation des Ableitungsschrittes)
Alternativen durch Eingabesymbol unterscheidbar, da $\mathcal{G} \in LL(1)$.

Eingabe:

sym	Variable für das Eingabesymbol
$nextsym$	Zum Lesen des nächsten Eingabesymbols

Ausgabe: $print(i)$ zur Ausgabe einer Regelnummer

Folie: Beispiel zu $\mathcal{G}'_{AE}$

3.2 Bottom-Up-Analyse, $LR(k)$-Grammatiken

Idee: Bottom-Up-Berechnung des Ableitungsbaumes in Form einer gespiegelten Rechtsanalyse durch einen Kellerautomaten:

Shift-Schritte: Verschieben von Eingabesymbolen auf den Keller

Reduce-Schritte: Umkehrung von Ableitungs- zu Reduktionsschritten → Shift-Reduce-Verfahren

Definition: Der nichtdeterministische Bottom-Up-Analyseautomat von $\mathcal{G}$, kurz: NBA($\mathcal{G}$):

Eingabealphabet: Σ

Kelleralphabet: $\mathfrak{X}$

Ausgabealphabet: $[p]$

Konfigurationsmenge: $\mathfrak{X}^* \times \Sigma^* \times [p]^*$

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 $\pi_i = A \to \alpha$

Ausgangskonfiguration für $w \in \Sigma^*$**:** $(\varepsilon, w, \varepsilon)$

Endkonfigurationen: (S, ε, z)

Satz: Der NBA($\mathcal{G}$) berechnet gespiegelte Rechtsanalysen, d.h. für $w \in \Sigma^*$ und $z \in [p]^*$ gilt: z ist eine r-Analyse von $w \leftrightarrow (\varepsilon, w, \varepsilon) \overset{*}{\longmapsto} (s, w, \overleftarrow{z})$

Nicht-Determinismus des NBA($\mathcal{G}$)

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

Ziel: Nicht-Determinismus durch k-look-ahead auf der Eingabe beseitigen. $LR(k)$-Grammatiken

Generelle Voraussetzung $\mathcal{G}$ startsepariert, d.h. S nur in $S \to A$ mit $A \neq S$. Jedes $\mathcal{G} \in$ CFG startsepariert machen durch Hinzufügen von $S' \to S$ (neues S'). Im folgenden nur startseparierte Grammatiken mit $S' \to S(0)$.

Folgerung: (S', ε, z) ist eine Endkonfiguration, die zwar Folgekonfiguration haben kann (ε-Regeln) ohne Übergang in weitere Endkonfigurationen (Kellerlänge ≥ 2).
Beseitigung des restlichen Nicht-Determinismus durch

Definition ($LR(k)$-Grammatiken: Sei $\mathcal{G} \in$ CFG, startsepariert durch $S' \to S, k \in \mathbb{N}$.
$\mathcal{G} \in LR(k) :\leftrightarrow$ Für alle Rechtsableitungen der Form

$$S' \Rightarrow S \begin{cases} \overset{*}{\underset{r}{\Rightarrow}} \alpha A w \underset{r}{\Rightarrow} \alpha\beta w \\ \overset{*}{\underset{r}{\Rightarrow}} \alpha' A' v' \underset{r}{\Rightarrow} \alpha\beta v \end{cases}$$

mit $first_k(w) = first_k(v)$ gilt: $\alpha = \alpha', A' = A$ und $v' = v$.
Folgerung: Der NBA($\mathcal{G}$) kann mit k-look-ahead auf der Eingabe die Transition entscheiden.

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

Definition ($LR(0)$-**Auskünfte,** $LR(0)$-**Mengen:** Sei $\mathcal{G} \in$ CFG mit $S' \to S$ und $S' \underset{r}{\overset{*}{\Rightarrow}} \alpha A w \underset{r}{\Rightarrow} \alpha\beta_1\beta_2 w$. Dann heißt $[A \to \beta_1 \cdot \beta_2]$ eine $LR(0)$-Auskunft für $\alpha\beta_1$.
Für $\gamma \in \mathfrak{X}^*$ 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. $LR(0)(\mathcal{G}) =: \{LR(0)(\gamma) | \gamma \in \mathfrak{X}^*\}$ 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. $[A \to \beta_1\beta_2] \in LR(0)(\gamma)$ mit $\beta_2 \neq \varepsilon$ bedeutet Shift-Möglichkeit wegen unvollständigem Henkel.
5. $\mathcal{G} \in LR(0) \leftrightarrow$ Die $LR(0)$-Mengen enthalten keine widersprüchlichen Auskünfte.

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

Satz $\mathcal{G} \in$ CFG mit $S' \to S, \mathcal{G}$ reduziert. Dann gilt

1. $LR(0)(\varepsilon)$ ist die kleinste Menge, welche
   a) $[S' \to \cdot S]$ enthält
   b) mit $[Ato \cdot B\delta$ und $B \to \beta$ in $\mathcal{G}$ auch $[B \to \cdot\beta]$ enthält.
2. $LR(0)(\alpha X)$ mit $X \in \mathfrak{X}$ 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 $\mathcal{G}$ auf $[B \to \cdot\beta]$ enthält.

Wiederholung: $\mathcal{G} \in LR(0) \leftrightarrow$ Für alle Ableitungen $S' \underset{r}{\overset{*}{\Rightarrow}} \alpha A w \underset{r}{\Rightarrow} \alpha\beta w$ und $S' \underset{r}{\overset{*}{\Rightarrow}} \alpha' A' v' \underset{r}{\Rightarrow} \alpha\beta v$ gilt:

$$\alpha' = \alpha, \quad A' = A, \quad v' = v$$

Beispiel

$A \to \beta_1 B \beta_2, \quad B \to \gamma$

$S' \underset{r}{\overset{*}{\Rightarrow}} \alpha A w \underset{r}{\overset{(1)}{\Rightarrow}} \alpha\beta_1 B\beta_2 w \underset{r}{\overset{*}{\Rightarrow}} \alpha\beta_1 w' w \underset{r}{\overset{(2)}{\Rightarrow}} \alpha\beta_1\gamma w' w$

1. a) $[A \to \beta_1 B\beta_2 \cdot] \in LR(0)(\alpha\beta_1 B\beta_2)$
   b) $[A \to \cdot\beta_1 B\beta_2] \in LR(0)(\alpha)$
   c) $[A \to \beta_1 \cdot B\beta_2] \in LR(0)(\alpha\beta_1)$
   d) $[A \to \beta_1 B \cdot \beta_2] \in LR(0)(\alpha\beta_1 B)$

   Folgerung: Auskünfte längerer Keller folgen durch Punktverschiebung aus Auskünften kürzerer Keller.

2. c) $[B \to \cdot\gamma] \in LR(0)(\alpha\beta_1)$
   Folgerung: $[A \to \beta_1 \cdot B\beta_2] \in LR(0)(\alpha\beta_1), \quad B \to \gamma \quad \curvearrowright [B \to \cdot\gamma] in LR(0)(\alpha\beta_1)$

Die goto-Funktion $\mathcal{G} \in LR(0) \curvearrowright LR(0)(\gamma)$ liefert eine Shift/Reduce-Entscheidung für den Bottom-Up-Analyseautomaten mit Kellerinschrift γ.

- Neues Kelleralphabet: $LR(0)(\mathcal{G})$ statt $\mathfrak{X}$
- Beachte: $LR(0)(\gamma X)$ ist bestimmt durch $LR(0)(\gamma)$ und X, aber unabhängig von γ.
- goto: $LR(0)(\gamma) \times \mathfrak{X} \to LR(0)(\mathcal{G})$ ist definiert durch $goto(I, X) := I' \leftrightarrow \exists \gamma \in \mathfrak{X}^*$ mit $I = LR(0)(\gamma)$ und $I' = LR(0)(\gamma X)$.

Berechnung der $LR(0)$-Mengen und goto-Funktion durch Potenzmengenkonstruktion eines NFA Sei $\mathcal{G} \in$ CFG startsepariert mit $S' \to S$. Konstruktion eines $\mathfrak{A}(\mathcal{G}) \in \mathrm{NFA}_\varepsilon$.

- Zustandsmenge $Q = \{[A \to \beta_1 \cdot \beta_2] | A \to \beta_1\beta_2 \in \mathcal{G}\}$
- Eingabealphabet $\mathfrak{X} = N \cup \Sigma$
- Anfangszustand $q_0 := [S' \to \cdot S]$
- Endzustandsmenge $F := Q$ (ohne Bedeutung)
- Transitionsfunktion: $\delta : Q \times \mathfrak{X}_{veps} \to \mathcal{P}(Q)$

  $\delta([A \to \beta_1 \cdot X\beta_2], x) \ni [A \to \beta_1 X \cdot \beta_2]$

  $\delta([A \to \beta_1 \cdot B\beta_2], \varepsilon) \ni [B \to \cdot\gamma], \text{ falls } B \to \gamma \in \mathcal{G}$

Potenzmengenkonstruktion (Thompson) Konstruktion von $\mathfrak{A}(\hat{\mathcal{G}}) = \langle \hat{Q}, \mathfrak{X}, \hat{\delta}, \hat{q_0}\emptyset \rangle \in$ DFA
Erweiterte Transitionsfunktion:

- $\bar{\delta} : \mathcal{P}(Q) \times \mathfrak{X}^* \to \mathcal{P}(Q)$
- $\bar{\delta}(T, \varepsilon) := \varepsilon(T)$
- $\bar{\delta}(T, wa) := \varepsilon(\bigcup_{q \in \bar{\delta}(T,w)} \delta(q, a))$
- $\hat{Q} := \{\bar{\delta}(\{[S' \to \cdot S]\}, \alpha) | \alpha \in \mathfrak{X}^*\}$
- $\hat{q_0} := \varepsilon(\{[S' \to \cdot S]\})\}$
- $\hat{\delta}(T, X) := \bar{\delta}(T, X)$

Dann gilt: $\hat{Q} = LR(0)(\mathcal{G})$ und $\hat{\delta} = goto$.

Konstruktion des deterministischen BU-Analyseautomaten für $\mathcal{G} \in LR(0)$ Hilfsmittel: $LR(0)(\mathcal{G})$ und goto-Funktion
Die action-Funktion von $\mathcal{G}$ gibt die Shift/Reduce-Entscheidung an:

$$act : LR(0)(\mathcal{G}) \to \{shift, red \quad i, error, accept | i \in [p]\}$$

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

DBA für $\mathcal{G} \in LR(0)$

$$act : LR(0)(\mathcal{G}) \to \{shift, red \quad i, error, accept | i \in [p]\}$$

Die act und goto-Funktionen bilden $LR(0)$-Analysetabelle von $\mathcal{G}$. Die Tabelle bestimmt $LR(0)(\mathcal{G})$-Analyseautomaten.

- Eingabealphabet: Σ
- Kelleralphabet: $\Gamma := LR(0)(\mathcal{G})$
- Ausgabealphabet: $\Delta := [p] \cup \{error\} \cup \{0\}$
- Konfigurationsmenge: $\Gamma^* \times \Sigma^* \times \Delta^*$
- Bezeichnung: $wa_1 \dots a_n - (n) = w$
- Transitionen:
  - Shift-Schritt: $(\alpha I, aw, z) \vdash (\alpha I I', w, z)$ falls $act(I) = shift$ und $goto(I) = I'$

- Reduce-Schritt: $(\alpha I, w, z) \vdash (\tilde{\alpha}\tilde{I}\tilde{I}', w, zi)$ falls $act(I) = red \quad i, \quad \pi_i = A \to X_1, \ldots, X_n, X_i \in \mathfrak{X}$ und $\alpha I - (n) = \tilde{\alpha}\tilde{I}$ und $goto(\tilde{I}, A) = I'$
- Accept-Schritt: $(I_0 I, \varepsilon, z) \vdash (\varepsilon, \varepsilon, z0)$ falls $act(I) = accept$
- Fehlererkennung: $(\alpha I, w, u) \vdash (\varepsilon, \varepsilon, z \quad error)$ in allen anderen Fällen

- Anfangskonfiguration für $w \in \Sigma^*$: (I_0, w, ε), $I_0 = LR(0)(\varepsilon)$
- Folgerung: Wenn $LR(0)$-Mengen konfliktfrei sind, also *act* eindeutig, so arbeitet der $LR(0)$-Analyseautomat deterministisch, und es gilt: Für $w \in \Sigma^*$ und $z \in ([p] \cup \{0\})^*$
  - $(I_0, w, \varepsilon) \overset{A}{\vdash} (\varepsilon, \varepsilon, z) \leftrightarrow \overleftarrow{z}$ r-Analyse von w
  - $(I_0, w, \varepsilon) \overset{*}{\vdash} (\varepsilon, \varepsilon, z \cdot error) \leftrightarrow w \notin L(\mathcal{G})$
- Beispiel: $w = aac$

$$
\begin{array}{lll}
(I_0, aac, \varepsilon) & \vdash (I_0 I_4, ac, \varepsilon) & \vdash (I_0 I_4 I_4, c, \varepsilon) \\
\vdash (I_0 I_4 I_4 I_6, \varepsilon, \varepsilon) & \vdash (I_0 I_4 I_4 I_8, \varepsilon, 6) & \vdash (I_0 I_4 I_8, \varepsilon, 65) \\
\vdash (I_0 I_3, \varepsilon, 655) & \vdash (I_0 I_1, \varepsilon, 6552) & \vdash (\varepsilon, \varepsilon, 65520)
\end{array}
$$

$\overleftarrow{z} = 02556$

$S' \underset{r}{\overset{0}{\Rightarrow}} S \underset{r}{\overset{2}{\Rightarrow}} C \underset{r}{\overset{5}{\Rightarrow}} aC \underset{r}{\overset{5}{\Rightarrow}} aaC \underset{r}{\overset{6}{\Rightarrow}} aac = w$

- Praxis: Widersprüchliche Informationen / $LR(0)$-Mengen $\rightsquigarrow$ $\mathcal{G} \notin LR(0)$ $\rightsquigarrow$ $SLR(1)$-Analyse
- Beachte: Shift/Reduce-Konflikte in I_1, I_2, I_9 (CB 2-16)
  "Trick": Beseitigung des Konfliktes durch Eingabesymbol

Beobachtung:

1. $[A \to \beta_1 a \beta_2] \in LR(0)(\alpha\beta_1) \curvearrowright S \underset{r}{\overset{*}{\Rightarrow}} \alpha A w \underset{r}{\Rightarrow} \alpha\beta_1 a \beta_2 w$
   Also: Shift nur bei Eingabe von a
2. $[A \to \beta\cdot] \in LR(0)(\alpha\beta) \curvearrowright S \underset{r}{\overset{*}{\Rightarrow}} \alpha A a w \Rightarrow \alpha\beta a w \curvearrowright a \in \text{fo}(A)$
   Reduktion mit $A \to \beta$ nut falls $a \in \text{fo}(A)$.

Für obiges Beispiel (CB 2-16):

- I_1: shift bei Eingabe von $+$
- I_1: accept bei Eingabe von \$, $\text{fo}(E') = \{\varepsilon\}$
- I_2: shift bei Eingabe von *

- I_2: reduce 2 bei Eingabe von +,), \$, fo$(E) = \{+,), \varepsilon\}$
- I_9: shift bei Eingabe von *
- I_9: reduce 1 bei Eingabe von +,), \$, fo$(E) = \{+,), \varepsilon\}$

Konflikte beseitigt. Weiterer Vorteil: Frühere Fehlererkennung (genauere Kontrolle der Aktionen).

SLR(1)-action-Funktion (simple)

$$act : LR(0)(\mathcal{G}) \times (\Sigma \dot{\cup} \{\$\}) \to \{shift, red \quad i, accept, error | i \in [p]\}$$

sei definiert durch:

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

Definition: $\mathcal{G} \in SLR(1) :\leftrightarrow act(I, a)$ ist stets eindeutig. action und goto bilden $SLR(1)$-Analysetabelle von $\mathcal{G}$. (CB 2-17)
Bemerkgung:

- Tabellenkompression
- Bessere Konfliktbeseitigung durch Verwendung des lookahead-Symbols in der $[A \to \beta_1 \cdot \beta_2 a] \rightsquigarrow LR(1), LALR(1)$

(CB 2-18) Konflikt in I_2 nicht nach der SLR-Methode lösbar: $= \in$ fo(R), aber nicht jedes Element von fo(R) in beliebiger r-Ableitung hinter R möglich. $\curvearrowright$ Verfeinerung der $LR(0)$-Auskünfte durch Mitführen der möglichen la-Symbole.

Definition: ($LR(1)$-Auskünfte und $LR(1)$-Mengen für $\mathcal{G} \in$ CFG)

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

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

Beobachtung: $LR(1)(\gamma)$ "enthält in der ersten Komponente" gerade $LR(0)(\gamma)$.

Berechnung der $LR(1)$-Mengen Modifikation der Berechnung von $LR(0)(\mathcal{G})$ unter Berücksichtigung des r-Kontextes.

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

Die $LR(1)$-action-Funktion von $\mathcal{G}$

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

sei definiert durch

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

Dann gilt: $\mathcal{G} in LR(1) \leftrightarrow act$ eindeutig (keine Konflikte)

$LALR(1)$-**Analyse** Beseitigung von Entscheidungskonflikten nach $LR(1)$ zu aufwendig (s. CB 2-19)
Beobachtung: Mögliche Informationsredundanz bei $LR(1)(\mathcal{G})$. Jede $LR(1)$-Menge "enthält" eine $LR(0)$-Menge.

Definition: $I_1, I_2 \in LR(1)(\mathcal{G})$ heißen $LR(0)$-äquivalent, $\underset{0}{\sim}: \leftrightarrow$ Die $LR(0)$-Anteile von I_1 und I_2 sind gleich.
Beispiel: $I'_4 \underset{0}{\sim} I'_{11}, \qquad I'_5 \underset{0}{\sim} I'_{12}, \qquad I'_7 \underset{0}{\sim} I'_{13}, \qquad I'_8 \underset{0}{\sim} I'_{10} \qquad (CB2-19)$

Folgerung: $|LR(1)(\mathcal{G})/\underset{0}{\sim}| = |LR(0)(\mathcal{G})|$
Oft können $LR(0)$-äquivalente $LR(1)$-Informationen vereinigt werden, ohne daß die Konfliktlösbarkeit verloren geht. Insbesondere können nur RR-Konflikte durch Vereinigung entstehen! Eventuell entsteht dadurch eine spätere Fehlererkennung.

Definition: $I \in LR(1)(\mathcal{G})$ bestimmt eine $LALR(1)$-Menge $\bigcup\{I' \in LR(1)(\mathcal{G}) | I' \underset{0}{\sim} I\}$.
$LALR(1)(\mathcal{G})$: Menge der $LALR(1)$-Mengen von $\mathcal{G}$.
Es gilt: $|LALR(1)(\mathcal{G})| = |LR(0)(\mathcal{G})|$

Die $LALR(1)$-action-Funktion von $\mathcal{G}$ ist analog zu $LR(1)$ definiert.

Definition: $\mathcal{G} \in LALR(1) :\leftrightarrow LALR(1)$-action-Funktion eindeutig.
Die $LR(1)$-goto-Funktion überträgt sich auf $LALR(1)(\mathcal{G})$, 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 $LR(1)(\alpha X)$ und X vollständig bestimmt.

3.2.1 Bottom-Up-Analyse mehrdeutiger Grammatiken

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

1. Beispiel:

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

   Präzedenz: $*$ vor $+$
   Assoziativität: links
   Konflikte (CB 2-25): I_1 $SLR(1)$-lösbar; I_7, I_8 nicht auflösbar wegen Mehrdeutigkeit
   Beispielrechnung: $a + a * a$

I_0	$a + a * a$
$I_0 I_3$	$+a * a$
$I_0 I_1$	$+a * a$
$I_0 I_1 I_4$	$a * a$
$I_0 I_1 I_4 I_3$	$*a$
$I_0 I_1 I_4 I_7$	$*a$

   $\{+, *\} \subseteq \text{fo}(E) \curvearrowright$ keine Konfliktlösung; Präzedenz:

   a) $* \gtrdot + \curvearrowright act(I_7, *) = shift$
   b) $+ \gtrdot * \curvearrowright act(I_7, *) = red \quad 1$

2. Beispiel (CB 2-27): Mehrdeutigkeit bei Verzweigungen ("dangling else")

   $$S \to iSeS | iS | a$$

   Konflikt bei $I_4 : e \in \text{fo}(S) \leftrightarrow$ nicht LR-lösbar: 2 Zerlegungen

   a) if b then (if b then a else a) [korrekt]
   b) if b then (if b then a) else a [nicht!]

K. John Gough: Syntax Analysis and Software Tools, Addison-Wesley, 1988

4 Semantische Analyse, Attributgrammatiken

- Ergebnis der syntaktischen Analyse: Ableitungsbaum
- Kontextabhängige Grammatiken (nicht durch CFG beschreibbar):
  - Deklariertheit von Bezeichnern (vgl. ATFS: $\{ww|w \in \Sigma^* \not\in \text{CFL}\}$)
  - Typinformation
- Festlegung dieser Eigenschaften:
  - Gültigkeitsregeln: Gültigkeitsbereich einer Deklaration
  - Sichtbarkeitsregeln: Sichtbar im Gültigkeitsbereich (Überdeckung globaler durch lokale Deklaration)
  - Typvorschriften: Typkonsistenz

Statische Grammatik: kontextabhängige, laufzeitunabhängige Eigenschaften eines Programms

- Formale Beschreibung durch Attributgrammatiken
- Idee (Knuth): CFG + semantische Regeln $\rightsquigarrow$ Zusatzinformationen für 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: BU-Berechnung
- inherite Attribute: TD-Berechnung $\curvearrowright$ beliebiger Informationstransfer im Ableitungsbaum
- Attributwerte: Symboltabellen, Typen, Code, Fehler, ...; breite Anwendbarkeit von Attributgrammatiken (AG), syntaxgerichtete Programmierung, automatische Attributauswertung in Compilergeneratoren

CB:

- statische Semantik
- Programmanalyse für Optimierung
- Codegenerierung
- Fehlerbehandlung

Historisch: Knuth [68]: Semantics of CFLs

Beispiel (Binärzahlen) $\mathcal{G}_B$:

Bits	$B \to 0$	$v.0 = 0$
	$B \to 1$	$v.0 = 1$
Lists	$L \to B$	$v.0 = v.1$
		$l.0 = 1$
	$L \to LB$	$v.0 = w * v.1 + v.2$
		$l.0 = l.1 + 1$
Numbers	$N \to L$	$v.0 = v.1$
	$N \to L.L$	$v.0 = v.1 + \frac{v.2}{2^{l.2}}$

$\mathcal{G}_B$ erzeugt Binärzahlen mit und ohne Punkt,

- Synthetische Attribute:
  $B, N : v(value)$
  $L : v.l(length)$
- Semantische Regeln: Attributgleichungen mit Attributvariablen
- Index i: i-tes (Nichtterminal-) Symbol
- Ziel: Bestimmung des Zahlenwerts
- Attribute von
  - v : rationale Zahlen ($A^v = \mathbb{Q}$)
  - l : natürliche Zahlen ($A^l = \mathbb{N}$)

Attributierung von $\mathcal{G}_B$ mit synthetischen und inheriten Attributen, zusätzliches inherites Attribut für Bits und Listen: p (position)

$B \to 0$	$v.0$			
$B \to 1$	$v.0 = 2^{p.0}$			
$L \to B$	$v.0 = v.1$	$l.0 = 1$	$p.1 = p.0$	
$L \to LB$	$v.0 = v.1 + v.2$	$l.0 = l.1 + 1$	$p.1 = p.0 + 1$	$p.2 = p.0$
$N \to L$	$v.0 = v.1$		$p.1 = 0$	
$N \to L.L$	$v.0 = v.1 + v.2$	$p.1 = 0$	$p.2 = -l.2$	

Berechnung des Wurzelattributs:

1. Längen ↑ BU
2. Positionen ↓ TD
3. Werte ↑ BU

Definition (Attributgrammatik): Sei $\mathcal{G} = \langle N, \Sigma, P, S\rangle \in$ CFG. Sei Att eine Menge von Attributen, $A = (A^\alpha | \alpha \in \text{Att})$ eine Familie von Attribut-Wertmengen und $att : \mathfrak{X} \to P(Att)$ eine Attributzuordnung.
Sei $\text{Att} = \text{Syn} \dot\cup \text{Inh}$ eine Zerlegung in Teilmengen synthetischer und inheriter Attribute, sodaß att zerfällt in

$$\text{syn} : \mathfrak{X} \to P(\text{Syn}) \text{ mit } \text{syn}(X) := \text{att}(X) \cap \text{Syn}$$

und

$$\text{inh} : \mathfrak{X} \to P(\text{Inh}) \text{ mit } \text{inh}(X) := \text{att}(X) \cap \text{Inh}.$$

Eine Regel $\pi = X_0 \to X_1 \dots X_r \in P$ bestimmt die Menge $\text{Var}_\pi := \{\alpha.i | \alpha \in \text{att}(X_i), 0 \leq i \leq r\}$ der formalen Attributvariablen von π mit den Teilmengen

$$\text{IVar}_\pi := \{\alpha.i | (i = 0 \wedge \alpha \in \text{syn}(X_0)) \vee (1 \leq i \leq r \wedge \alpha \in \text{inh}(X_i))\}$$

und

$$\text{OVar}_\pi := \text{Var}_\pi \setminus \text{IVar}_\pi$$

der Innen- bzw. Außenvariablen.

Eine Attributgleichung von π (semantische Regel) hat die Form

$$\alpha.i = f(\alpha_1 \cdot i_1, \dots, \alpha_n \cdot i_n)$$

mit $\alpha.i \in \text{IVar}_\pi$, $\alpha_1 \cdot i_1, \dots \alpha_n \cdot i_n \in \text{OVar}_\pi$, $f : A^{\alpha_1} \times \dots \times A^{\alpha_n} \to A^\alpha$ und $n \in \mathbb{N}$.
Sei E_π eine Menge von Attributgleichungen, in der jede Innenvariable genau einmal vorkommt und $E := (E_\pi | \pi \in P)$. Dann heißt $\mathfrak{A} = (\mathcal{G}, E)$ eine Attributgrammtik $\mathfrak{A} \in$ AG.

Definition (Das Attributgleichungssystem eines Ableitungsbaumes) Sei $\mathfrak{A} = (\mathcal{G}, E) \in$ AG. $\mathfrak{A}$ induziert für jeden Ableitungsbaum t von $\mathcal{G}$ ein Attributgleichungssystem E_t. Sei $\text{Kn}(t)$ die Menge der Knoten von t. Sie bestimmt die Menge

$$\text{Var}_t := \{\alpha.k | k \in \text{Kn}(t) \text{ markiert durch } X \in \mathfrak{X}, \alpha \in \text{att}(X)\}$$

der aktuellen Attributvariablen von t.
Wird auf einen inneren Knoten (kein Blattknoten) $k_0 \in \text{Kn}(t)$ die Regel $\pi = X_0 \to X_1 \dots X_r$ angewandt und sind $k_1, \dots, k_r \in \text{Kn}(t)$ die entsprechenden Nachfolgerknoten, so erhält man das Attributgleichungssystem E_{k_0} von k_0 aus E_π durch die Indexsubstitution $(i \to k_i | 0 \leq i \leq r)$ bei den Attributvariablen. Dann ist $E_t := \bigcup\{E_k | k \text{ innerer Knoten von } t\}$.

- Beachte: Zu jeder aktuellen Attributvariablen $\alpha.k$, ausgenommen die inheriten der Wurzeln und die synthetischen der Blätter, gibt es genau eine Gleichung $\alpha.k = \ldots$.
- Annahme:
  - keine inheriten Attribute des Startsymbols
  - synthetische Attribute der Terminalsymbole vom Scanner

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

Beispiel: $\mathcal{G}$ enthalte die Regeln $A \to aBc$ und $B \to w$. Ferner $\alpha \in \text{syn}(B)$ und $\beta \in \text{inh}(B)$.

- Attributgleichungen: ...

  $$\beta.2 = f(\alpha, 2)$$

  $$\alpha.0 = g(\beta.0)$$

- E_t: zirkuläre Abhängigkeit

  $$\beta.k = f(\alpha.k)$$

  $$\alpha.k = g(\beta.k)$$

- Für $A^\alpha = A^\beta = \mathbb{N}, g = id_\mathbb{N}$ und
  1. $f(x) = x + 1$ ergibt keine Lösung
  2. $f(x) = 2x$ ergibt genau eine Lösung
  3. $f(x) = x$ ergibt unendlich viele Lösungen

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

Definition: $\mathfrak{A} \in \text{AG}$ heißt zirkulär: $\leftrightarrow$ Es gibt einen Ableitungsbaum t mit zirkulärem Attributsgleichungssystem E_t, d.h. eine aktuelle Attributvariable hängt von sich selbst ab.

Ein Zirkularitätstest für Attributgrammatiken Sei $\mathfrak{A} = \langle \mathcal{G}, E \rangle \in \text{AG}$. Eine Regel $\pi = X_0 \to X_1 \ldots X_r$ bestimmt ihren Abhängigkeitsgraphen DG_π mit der Knotenmenge $\text{Kno}(\text{DG}_\pi) := \text{OKno}(\text{DG}_\pi) \cup \text{UKno}(\text{DG}_\pi)$

$\text{OKno}(\text{DG}_\pi) := \{\alpha.0 | \alpha \in \text{att}(X_0)\}$ Oberknoten

$\text{UKno}(\text{DG}_\pi) := \{\alpha.i | \alpha \in \text{att}(X_i), 1 \le i \le r\}$ Unterknoten

und der Kantenmenge $\text{Kan}(\text{DG}_\pi)$:

$(x_1, x_2) \in \text{Kan}(\text{DG}_\pi) :\leftrightarrow x_2 = f(\ldots x_1 \ldots) \in E_\pi$

Beispiel:

Beachte: $\mathrm{Var}_\pi = \mathrm{UKno}(\mathrm{DG}_\pi) \cup \mathrm{OKno}(\mathrm{DG}_\pi) = \mathrm{OVar}_\pi \cup \mathrm{IVar}_\pi$

Problem: Beim Verkleben der DG_π zu DG_t für einen Ableitungsbaum A können Schleifen auftreten.

Folgerung: $\mathfrak{A}$ ist zirkulär $\leftrightarrow$ Es gibt einen Ableitungsbaum t von $\mathfrak{A}$, so daß DG_t eine Schleife enthält.

Bemerkgung: DG_t entsteht aus $(\mathrm{DG}_\pi | \pi \in P)$ wie E_t aus $(E_\pi | \pi \in P)$ und t aus $(\pi | \pi \in P)$ entstehen.

Beobachtung: Eine Schleife in DG_t bestimmt einen Teilgraphen der Form, also eine Regel π mit verbindbaren Unterknoten von DG_π. Eine Verbindung gehört zu einem Nichtterminalsymbol und verläuft von einer inheriten zu einer synthetischen Attributvariablen.

Definition (Attributabhängigkeit): Sei $A \in N, \alpha \in \mathrm{inh}(A), \alpha' \in \mathrm{syn}(A)$. Dann heißt α' von α unterhalb A abhängig, Bezeichnung: $\alpha \underset{\frown}{A} \alpha'$, wenn ein Ableitungsbaum t mit Wurzel A und Wurzelknoten k existiert, sodaß in DG_t ein Pfad von $\alpha.k$ nach $\alpha'.k$ führt:

$$(\alpha.k, \alpha'.k) \in \mathrm{trans}(\mathrm{Kan}(\mathrm{DG}_t))$$

Definition (Attributabhängigkeitsmengen): Sei t ein Ableitungsbaum mit Wurzel A.

$$D(A,t) := \{(\alpha, \alpha') | \, \alpha \underset{\frown}{A} \alpha' \text{ in } t\}$$

$$\mathcal{D}(A) := \{D(A,t) | t \text{ Ableitungsbaum mit Wurzel } A\}$$

Bei der induktiven Bestimmung benutzen wir folgende Bezeichnung:

Definition: Für $\pi = A_0 \to w_0 A_1 w_1 \ldots A_r w_r$ und $D_i \subseteq \mathrm{inh}(A_i) \times \mathrm{syn}(A_i)$ definieren wir

$$D[\pi_i; D_1, \ldots D_r] \subseteq \mathrm{inh}(A_0) \times \mathrm{syn}(A_0)$$

durch

$$\{(\alpha, \alpha') | (\alpha.0, \alpha'.0) \in \mathrm{trans}(\mathrm{Kan}(\mathrm{DG}_{pi}) \cup \bigcup_{i=1}^{r} \{(\beta.i, \beta'.i) | (\beta, \beta') \in D_i\})\}.$$

Lemma: Die Attributabhängigkeitssysteme $\mathcal{D}(A)$ mit $A \in N$ sind induktiv bestimmt durch:

1. $\pi = A \to w \curvearrowright D[\pi_i] \in \mathcal{D}(A)$
2. $\pi = A \to w_0 A_1 w_1 \ldots A_r w_r, \quad D_i \in \mathcal{D}(A_i), \quad 1 \leq i \leq r \leftrightarrow D(\pi; D_1, \ldots, D_r] \in \mathcal{D}(A)$

Beweis: Induktion über die Struktur der Ableitungsbäume.

Beachte: Da für $D \in \mathcal{D}(A)$ gilt: $D \subseteq \text{inh}(A) \times \text{syn}(A)$, bricht der Berechnungsprozeß nach endlichen vielen Schritten ab.

Folgerung (Zirkularitätstest): $\mathfrak{A} \in \text{AG}$ ist zirkulär $\leftrightarrow$ es gibt $\pi = A_0 \to w_0 A_1 w_1 \ldots A_r w_r$, $\alpha.k \in \text{UKno}(\text{DG}_\pi)$ und $D_i \in \mathcal{D}(A_i)(1 \leq i \leq r)$, so daß $(\alpha.k, \alpha.k) \in \text{trans}(\text{Kan}(\text{DG}_\pi) \cup \bigcup_{i=1}^{r} \{\beta.i, \beta'.i | (\beta, \beta') \in D_i\}$

Komplexität $n = |\Omega|, T(n)$ Zeit zur Entscheidung der Zirkularität

$$2^{\frac{cn}{\log n}} \leq T(n) \leq 2^{dn^2}$$

Stark nicht-zirkuläre Grammatiken Keine Trennung der Abhängigkeitsmengen verschiedener Ableitungsbäume

$$D(A) := \{(\alpha, \alpha') | \, \alpha \underset{\curvearrowright}{A} \alpha'\}$$

Hinreichendes Kriterium für die Nichtzirkulariät. Test in Polynomzeit.
Beachte: Es gibt $\mathfrak{A} \in \text{AG}$, sodaß Ω nicht zirkulär ist, ohne stark nicht-zirkulär zu sein.

Attributberechnung:

1. E_t als Termersetzungssystem, TD-Berechnung
   keine Zirkularität $\curvearrowright$ Termination mit eindeutiger Lösung
2. BU-Auflösung von E_t: Variablen durch Werte ersetzen und Terme ausrechnen
3. Uniforme Berechnung, unabhängig von Ableitungsbaum
   Rekursive Prozeduren / Funktionen für $(E_\pi | \pi \in P)$
   Idee: Jedem synthetischen Attribut einer Variablen wird eine Prozedur / Funktion zugeordnet mit inheriten Attributen als Parameter
4. Spezialfälle: SAG, LAG mit Attributberechnung während der Syntaxanalyse

S-Attributgrammatiken

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

- Folgerung: BU-Berechnung der Attributwerte
- Spezialfall: Bestimmung der Wurzelattribute
- Idee: Durchführung der Attributberechnung während der BU-Syntaxanalyse, Zwischenspeichern von Attributwerten im Analysekeller (YACC).
  Beim Reduktionsschritt werden simultan die Werte der synthetischen Attribute von A aus Werten unter v_1, v_2, v_3 berechnet (records); Acc-Schritt: Ausgabe der Wurzelattribute

Beispiel 1: Stackcode für arithmetische Ausdrücke

$E \to (E + E)|(E * E)|id|num \qquad (1, 2, 3, 4)$

1. $c.0 = c.2; c.4; ADD$
2. $c.0 = c.2; c.3; MULT$
3. $c.0 = LOADa.1$, a: lexical address
4. $c.0 = LITn.1$, n: lexical value

Lexikalische Attribute, die mit dem Token vom Lexer übergeben werden:

$(id, X) \qquad (num, 42)$

$((3 + X) * (Y + 5)) \rightsquigarrow ((\langle num, 3\rangle + \langle id, X\rangle) * (\langle id, X\rangle + \langle num, 5\rangle))$

Codegenerierung durch Attributauswertung während der Syntaxanalyse:

1. shift: Aufrufe des Lexers durch "nextsym"
   a) Token $\mapsto LR(0)$-Menge (mit goto)
   b) lexikalisches Attribut

   Beide Informationen werden auf den Analysekeller abgelegt.
2. reduce:
   a) Reduktionen auf Analysekeller
   b) simultane Attributberechnung
3. accept: Wurzelattribut

$$LIT3; LOADX; ADD; LOADY; LIT5; ADD; MULT$$

Beispiel 2: Berechnung des abstrakten Syntaxbaumes

$\pi = A_0 \to w_0 A_1 w_1 \ldots A_r w_r$

repräsentiert Op-Symbol F_π vom Typ $A_1 \times \ldots \times A_0$ (A_i: Sorte, Typ).

- Folge: Ableitungsbaum vereinfacht sich zu abstraktem Syntaxbaum (AST); nur dieser ist für die Übersetzung notwendig
- Konkrete Syntax: Benutzerfreundliche "Mixfix"-Notation unter Verwendung natürlicher Sprache für Terminalsymbol von CFG
- Abstrake Syntax: Darstellungsabhängige algebraische Struktur ($\to$ ADT, Konstruktoren) (CB 3-6):
  `begin id:=num+id; if id<num then id:=num+id; end`
- Regeln selbst als Op-Symbole möglich: Regelbaum
- Aufgabe: S-Attributgrammatik
- Darstellung des Aufbaus von Syntaxbäumen durch Konstruktion von Graphen

  1. Definiere

$a \in Adr$	unendliche Menge von Adressen für Speicherplätze (Heap)
Ω	Op-Alphabet mit Stelligkeit, $f^{(n)} \in \Omega = \{assign^{(1)}, seq^{(2)}, cond^{(2)}, less^{(2)}, plus^{(2)}, id^{(0)}, num^{(0)}\}$
Kno	Menge der Ω-Knoten
$k \in$ Kno	$\{(a, f, a_1, \ldots, a_n) \mid a, a_i \in Adr, A \in \Omega^{(n)}$
Graph:	Menge von Knoten mit einer Wurzel
$G \in$ Graph	$\{(a, k) \mid a \in Adr, k \subseteq$ Kno$\}$

     (CB 3-7)
  2. 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

     $$k := \{(a, f, a_1, \ldots, a_n)\} \cup \bigcup_{i=1}^{n} k_i$$

     (k_i disjunkt, erfolgt automatisch)

- S-Attributierung von $\mathcal{G}_s$:
  Wurzeladressen als S-Attribute im Analysekeller; Heap statt Ausgabeband
  `/* printf("%d", i) -> mk-f_i */`
  Attributwerte als Graphen; Analyseergebnis: Graphdarstellung des AST auf dem Heap $\rightsquigarrow$ Basis für weitere Attributierung
- Beachte: Unterschied zwischen Wurzelattributberechnung (Beispiel 1) und voller Attributierung (Erweiterung von Beispiel 2)

Beispiel 3: Typenberechnung für arithmetische Ausdrücke $Typ : \{int, real\}$ mit $\max Typ^2 \mapsto Typ$ bzgl. $int < real$. $nextsym()$ (Lexer) bestimmt Typattribut von num, Typattribut von id über lookup-table $\rightsquigarrow$ Syntaxbaum mit Typinformationen (weiteres Knotenfeld) annotiert.

L-Attributgrammatiken

Definition: $\mathfrak{A} \in \langle \mathcal{G}, E \rangle \in$ AG ist eine L-Attributgrammatik ($\mathfrak{A} \in$ LAG) $\leftrightarrow$ Für jede Attributgleichung $\alpha.i = f(\ldots \beta.j \ldots)$ gilt: $\alpha \in \text{Inh}, \beta \in \text{Syn} \curvearrowright j < i$

Beispiel (DG_π): Bei L-Attributgrammatiken fallen die gestrichenen Kanten weg.

Folgerung: $\mathcal{G} \in$ LAG nicht zirkulär. Die Attributberechnung erfolgt in "depth-first, left-to-right order". Die Baumreise geschieht dabei mit 2 Kantenbesuchen:

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

Auswertungsreihenfolge:

Syntaxanalyse mit L-Attributauswertung Sei $\mathfrak{A} = \langle \mathcal{G}, E \rangle \in$ LAG mit $\mathcal{G} \in LL(1)$.

- Ziel: Erweiterung der TD-Analyse zur Berechnung der synthetischen Wurzelattribute
- Methode: Expansion von $A \in N$ auf dem Analysekeller so gestalten, daß spätere Reduktion möglich: TD + BU, TD: inherite Attribute, BU: synthetische Attribute
- Kelleralphabet: $LR(0)_\pi(\mathcal{G}) := \{[A \to \alpha.\beta] | \pi = A \to \alpha.\beta\}$

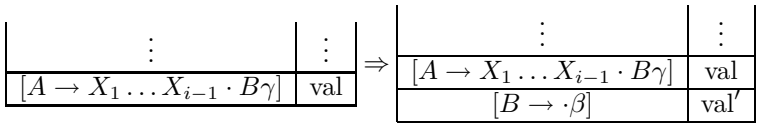
  $$\text{Val}_\pi = \{\text{val}_\pi | \text{val}_\pi : \text{Var}_\pi \to A\}$$

  $$\Gamma := \bigcup_{\pi \in P} (LR(0)_\pi(\mathcal{G}) \times \text{Val}_\pi \cup \{\to \cdot S, \emptyset), (\to S \cdot, \text{val})\})$$

  $$\text{val} : \text{syn}(S) \to A$$

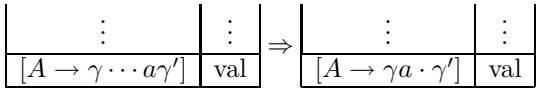
- Konfigurationen: (w, γ) (w: Eingabeband, $\gamma \in \Gamma$)
- Arbeitsweise (3 Fälle):

1. "expand"-Schritt mit Berechnung der inheriten Attribute



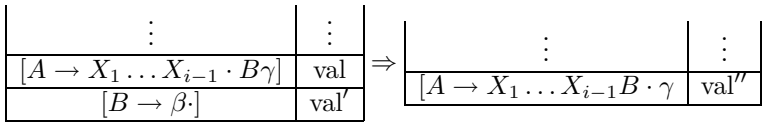
(Kellerspitze unten) Dabei ist $B \to \beta$ wegen $\mathcal{G} \in LL(1)$ durch Eingabe-Lookahead bestimmt. val′ belegt die inheriten Attributvariablen von B nach Attributgleichungen für $A \to X_1 \dots X_{i-1}B\gamma$: Sei $\alpha \in \text{inh}(B)$ und $\alpha.i = t \in E_{A \to X_1 \dots \gamma}$, so $\text{val}'(\alpha.0) := \text{val}(t) \in A$. (Die Funktion val übernimmt das Einsetzen und ausrechnen der Attibutwerte.)

2. "match"-Schritt (ohne Attributberechnung)



Falls a nächstes Eingabesymbol ist; Eingabekopf rückt vor.

3. "reduce"-Schritt mit Berechnung der synthetischen Attribute



val″ erweitert val um synthetische Attribute von B:

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

Anwendung von LAG Überprüfung der Deklariertheit von Bezeichnern: $\mathcal{G}$ sei gegeben durch

$P \to DL; SL$	Programm	
$DL \to V \mid N; DL$	Deklarationsliste	
$SL \to S \mid S; SL$	Statementliste	Beispiel: $a; c; a; b; a := c; c := d$
$V \to a \mid b \mid \dots$	Variablen	
$S \to V := V$	Statements	

Attribute Die Attribute werden wie folgt deklariert:

syn v	deklarierte oder benutze Variable für $V \in \{a, b, \dots\}$
syn dv	Menge von deklarierten Variablen für $DL \subseteq \{a, b, \dots\}$
inh v	Umgebung für $S, SL \subseteq \{a, b, \dots\}$
syn $decl$	deklariert für $S, SL, P \in \{true, false\}$

Attributgleichungen

$P \to DL; SL$	$decl.0 = decl.3$
	$env.3 = dv.1(LAG!)$
$DL \to V$	$dv.0 = \{v.1\}$
$DL \to V; DL$	$dv.0 = \{v.1\} \cup dv.3$
$V \to a \mid \ldots$	$v.0 = a$
$SL \to S$	$env.1 = env.0$
	$decl.0 = decl.1$
$SL \to S; SL$	$env.1 = env.0$
	$env.3 = env.0$
	$decl.0 = decl.1 AND decl.3$
$S \to V := V$	$decl.0 = v.1 \in env.1 AND v.0 \in env.0$

Attributierter Ableitungsbaum

5 Übersetzung in Zwischencode

Aufteilung der Codeerzeugung: $PS \xrightarrow{trans} Z \xrightarrow{code} MC$

- front-end: *trans* erzeugt maschinenunabhängigen Zwischencode für eine abstrake Stackmaschine
- back-end: *code* erzeugt Maschinencode

Vorteil dieser Zerlegung: Z ist maschinenunabhängig → Portabilität, Transparenz, Codeoptimierung
Beispiel:

- Java Virtual Machine (JVM)
- P-Code von Pascal

5.1 Übersetzung von Ausdrücken, Anweisungen, Blöcken und Prozeduren

Beispiel-Programmiersprache: BPS ("Pascal ohne Datenstrukturen und ohne Prozedurparameter")

- arithmetische und boolesche Ausdrücke mit strikter bzw. nicht-strikter Semantik
- Kontrollstrukturen: Sequenz, Verzweigung, Iteration
- Blöcke und Prozeduren: Lokale und globale Variablen, dynamische Speicherverwaltung mit Laufzeitkeller;
  unterschiedliche Verwendung in Programmiersprachen:
  - FORTRAN: Unterprogramme nicht geschachtelt, keine Rekursion ⇝ statische Speicherverwaltung, Speicherbedarf zur Übersetzungszeit bekannt
  - C: rekursive Prozeduren ohne Schachtelung ⇝ dynamische Speicherverwaltung, keine statischen Verweise
  - ALGOL-Familie: ALGOL 60, Pascal, Modula: Geschachtelte rekursive Prozedurdeklaration ⇝ dynamische Speicherverwaltung mit statischen Verweisen
- keine Datenstrukturen, keine Prozedurparameter
- Beschränkung auf Datentyp int
- Syntax: CB 4-6

Semantik

- Bezeichner einer Deklaration Δ müssen paarweise verschieden sein.
- Ein im Anweisungsteil Γ eines Blocks $\Delta\Gamma$ auftretender Bezeichner muß deklariert sein, und zwar in Δ oder in der Deklarationsliste eines $\Delta\Gamma$ umfassenden Blocks.
- Mehrfach-Deklaration eines Bezeichners auf verschiedenen Niveaus möglich: Die "innerste" Deklaration ist für ein Auftreten gültig.
- Static scope: Beim Aufruf einer Prozedur ist ihre Deklarationsumgebung gültig und nicht ihre Aufrufumgebung.

Zwischencode für BPS AM abstrakte Maschine mit Datenkeller und Prozedurkeller

- Zustandsraum $ZR := BZ \times DK \times PK$ mit
  - Befehlszähler $BZ := \mathbb{N}$
  - Datenkeller $DK := \mathbb{Z}^*$ (Spitze rechts)
  - Prozedurkeller $PK := \mathbb{Z}^*$ (Spitze links)
- Zustand $s = (m, d, p) \in ZR$ mit
  - Befehlsmarke m
  - DK-Zustand $d = d.r : \ldots : d.1$
  - PK-Zustand $p = p.1 : \ldots : p.t$

AM-Befehle

- arithmetische: ADD, ...
- logische: LT, NOT, AND, OR, ...
- Sprungbefehle: JMP n, JFALSE n ($n \in \mathbb{N}$)
- Prozedurbefehle: CALL(ca, dif, loc) ($ca, dif, loc \in \mathbb{N}$), RET
- Transportbefehle: LOAD(dif, off), STORE(dif, off) ($dif, off) \in \mathbb{N}$)

Befehlssemantik

- $[\![B]\!] : ZR \to ZR$ für jedem AM-Befehl
- $[\![ADD]\!](m, d : z_1 : z_2, p) := (m + 1, d : (z_1 + z_2), p)$
- $[\![LT]\!](m, d : z_1 : z_2, p) := (m + 1, d : b, p)$ mit $b = \begin{cases} 1 & \text{falls } z_1 < z_2 \\ 0 & \text{falls } z_1 \geq z_2 \end{cases}$
- $[\![AND]\!](m, d : b_1 : b_2, p) := (m + 1, d : b_1 \wedge b_2, p)$, entsprechend für OR und NOT

- $[\![JFALSEn]\!](m, d : b, p) := \begin{cases} (n, d, p) & \text{falls } b = 0 \\ (m+1, d, p) & \text{falls } b = 1 \end{cases}$
- $[\![JMPn]\!](m, d, p) := (n, d, p)$

Für die Semantik der Prozedur- und Transportbefehle: Besondere Struktur des Prozedurkellers p: p zerfällt in Aktivierungsblöcke (Frames) der Form $sv : dv : ra : l_1 : \ldots : l_k$

- sv: statischer Verweis, zeigt auf Aktivierungsblock der Deklarationsumgebung
- dv: dynamischer Verweis, zeigt auf den letzten Aktivierungsblock
- ra: Rücksprungadresse, ist die Codeadresse nach Beendigung des Prozeduraufrufs
- l_i: lokale Variablen des Prozeduraufrufs

Berechnung des statischen Verweises sv liefert die Differenz zwischen Aufruf- und Deklationsniveau und damit die Länge der Verweiskette.

Abstrakte Maschine

$s = (m, d, p)$

p zerfällt in Aktivierungsblöcke $sv : dv : ra : l_1 : \ldots : l_k$

Berechnung des statischen Verweises sv liefert die Differenz zwischen Aufruf- und Deklarationsniveau und damit die Länge der Verweiskette.

Hilfsfunktion: $base : PK \times \mathbb{N} \to \mathbb{N}$
bestimmt für einen Prozedurkeller bzgl. einer Niveadifferenz dif den Beginn der Deklarationsumgebung (als absolute Adresse des aktuellen Prozedurkellers).

$$base(p, 0) := 1$$

$$base(p, dif + 1) := base(p, dif) + p.base(p, dif)$$

Beispiel: 2. Aufruf von A (s. Folie CB 4-7)

$dif = 2$
$base(p, 0) = 1$
$base(p, 1) = 1 + p.1 = 6$
$base(p, 2) = 6 + p.6 = 11$

Es folgt: $sv = base(p, 2) + 2 + 2$

- $CALL(ca, dif, loc)$ mit
  - $ca \in \mathbb{N}$ Codeadresse

- $dif \in \mathbb{N}$ Niveaudifferenz
- $loc \in \mathbb{N}$ Zahl der lokalen Variablen

erzeugt neuen Aktivierungsblock und springt zum Code des Prozedurrumpfes

$$[\![CALL(ca, diff, loc)]\!](m, d, p) := \\ (ca, d, (base(p, dif) + 2 + loc) : (loc + 2) : (m + 1) : 0 : \ldots : 0 : p)$$

- *RET* löscht den letzten Aktivierungsblock und kehrt zur Aufruftabelle zurück

$$[\![RET]\!](m, d, p.1 : p.2 : \ldots : p.t) := \\ \text{if } t \geq 2 + p.2 \text{ then } (p.3, d, p.(2 + p.2) : \ldots : p.t)$$

Der Prozedurkeller wird stets mindestens einen Aktivierungsblock enthalten.

- $LOAD(dif, off)$ und $STORE(dif, off)$ laden und speichern Variablen zwischen Datenkeller und Prozedurkeller. Relative Adressierung mit
  - Niveaudifferenz dif zwischen Auftreten und Deklaration
  - Offset off als relative Adresse im Aktivierungsblock

  Die Kette der statischen Verweise bestimmt die sichtbare Umgebung auf dem Prozedurkeller.

$$[\![LOAD(dif, off)]\!](m, d, p) := (m + 1, d : p.[base(p, dif) + 2 + off], p) \\ [\![STORE(dif, off)]\!](m, d : z, p) := (m + 1, d, p.[base)p, dif) + 2 + off/z]) \\ [\![LITz]\!](m, d, p) := (m + 1, d : z, p)$$

AM-Code Befehlsfolgen mit aufsteigenden Befehlsmarken
$P \in$ AM-Code: $\leftrightarrow P = a_1 : B_1; \ldots; a_p : B_p$ mit $a_i \in Adr := \mathbb{N}, a_i = a_1 + i - 1$ und B_i ein AM-Befehl $(1 \leq i \leq p)$.
Semantik von P: Iteration der Befehle gemäß Befehlszähler

$$\mathfrak{I} : AM - Code \times ZR \to ZR$$

$$\mathfrak{I}[\![P]\!](m, d, p) := \text{ if } a_1 \leq m \leq a_p \text{ and } m = a \text{ then } \mathfrak{I}[\![P]\!]([\![B_i]\!](m, d, p)) \text{ else } (m, d, p)$$

Übersetzung von BPS-Programmen in AM-Code

- trans: BPS-Prog → AM-Code (als Zwischencode Z)
- Hilfsmittel: Symboltabelle

$$Tab := \{st | st : IDE \to \\ (\{const\} \times \mathbb{Z}) \vee (\{var\} \times Lev \times Off) \vee (\{proc\} \times Adr \times Lev \times Size)\}$$

  - Variablen-Deklaration:

  - Deklarationsniveau: $dl \in Lev$
  - Offset: $off \in Off := \mathbb{N}$
- Prozedurdeklaration:
  - Startadresse: $ca \in Adr$ des Prozedurcodes
  - Deklarationsniveau: $dl \in Lev$
  - Anzahl der lokalen Variablen: $loc \in Size := \mathbb{N}$

Aufbau der Symboltabelle

$up : Decl \times Tab \times Adr \times Lev \dashrightarrow Tab$

$up(\Delta, st, a, l)$ beschreibt den Update einer Symboltabelle st bzgl. einer Deklaration Δ bei freier Adresse a und aktuellem Niveau l (Blockschachtelungstiefe).

$$
\begin{array}{ll}
up(\Delta_c\Delta_v\Delta_p, st, a, l) & := \text{if } diff - id(\Delta_c\Delta_v\Delta_p) \\
 & \quad \text{then } up(\Delta_p, up(\Delta_v, up(\Delta_c, st, a, l), a, l), a, l) \\
up(\varepsilon, st, a, l) & := st \\
up(constI_1 = Z_1, \ldots, I_n = Z_n; st, a, l) & := st[I_1/(const, Z_1), \ldots, I_n/(const, Z_n)] \\
up(varI_1, \ldots, I_n; st, a, l) & := st[I_1/(var, l, 1), \ldots, I_n/(var, l, n)] \\
up(procI_1; B_1; \ldots; procI_n; B_n; st, a, l) & := st[I_1/(proc, a_1, l, size(B_1)), \ldots, I_n/(proc, a_n, l, size(B_n))] \\
size : Block \to \mathbb{N} & \\
size(\Delta_c varI_1, \ldots, I_n; \Delta_p\Gamma) & := n
\end{array}
$$

Anfangstabelle $P = in/outI_1, \ldots, I_n; B.$ hat die Semantik $\mathcal{M}[\![P]\!] : \mathbb{Z}^n \to \mathbb{Z}^n$. Für $(z_1, \ldots, z_n) \in \mathbb{Z}^n$ wählen wir den Anfangszustand $(1, \varepsilon, 0 : 0 : 0 : z_1 : \ldots : z_n)$. Die entsprechende Anfangstabelle hat daher n Einträge:

$st_{I/O}(I_j) = (var, 0, j)$ für $1 \leq j \leq n$

$$
\begin{array}{rl}
bt: & Block \times Tab \times Adr \times Lex \dashrightarrow AM - Code \\
dt: & Decl \times Tab \times Adr \times Lex \dashrightarrow AM - Code \\
ct: & Cmd \times Tab \times Adr \times Lex \dashrightarrow AM - Code \\
et: & AExp \times Tab \times Adr \times Lex \dashrightarrow AM - Code \\
sbt: & BExp \times Tab \times Adr \times Lex \dashrightarrow AM - Code
\end{array}
$$

$trans : Prog \to AM - Code$

$$
\begin{array}{ll}
trans(in/outI_1, \ldots, I_n; B.) := & 1 : CALL(a_\Gamma, 0, size(B)); \\
 & 2 : JMP0; \\
 & bt(B, st_{I/O}, a_\Gamma, 1)
\end{array}
$$

a_Γ: Startadresse des Anweisungscodes von Γ in $B = \Delta\Gamma$.

$$
\begin{array}{ll}
bt(\Delta\Gamma, st, a, l) := & dt(\Delta, up(\Delta, st, a1, l), a1, l) \\
 & ct(\Gamma, up(\Delta, st, a1, l), a, l) \\
 & a' : RET;
\end{array}
$$

Dabei erzeugt dt Cpde für die Prozedurrümpfe von Δ; in der zugehörigen Symboltabelle sind nicht nur Konstanten und Variablen von Δ, sondern auch die Prozedur-Bezeichner eingetragen (Rekursion!). ct enthält die übergebene Startadresse und erzeugt Code, der mit einem Rücksprung endet.

$$
\begin{array}{ll}
dt(\Delta_c\Delta_v\Delta_p, st, a, l) & := dt(\Delta_p, st, a, l) \\
dt(\varepsilon, st, a, l) & := \varepsilon \\
dt(procI_1; B_1; \ldots; procI_n; B_n; st, a, l) & := bt(B_1, st, a1, l+1) \\
 & \vdots \\
 & bt(B_n, st, an, l+1)
\end{array}
$$

Beachte:

1. $st(I_j) = (proc, a_j, l+1, off)$, weil bt die Funktionen bt und up mit dem gleichen Anfangsparameter aufruft und beide Funktionen aus diesen in gleicher Weise die Adressen für die Prozedurrümpfe erzeugen.

2. bt wird mit dem Level $l+1$ aufgerufen.

$ct(I := E, st, a, l) :=$ if $st(I) = (var, dl, off)$ then
$ct(E, st, a, l)$
$a' : STO(l - dl, off)$

$ct(I(), st, a, l) :=$ if $st(I) = (proc, ca, dl, loc)$ then
$a : CALL(ca, l - dl, loc)$

$ct(\Gamma_1; \Gamma_2, st, a, l) :=$
$ct(\Gamma_1, st, a, l)$
$ct(\Gamma_2, st, a, l)$

$ct($if BE then Γ_1 else $\Gamma_2, st, a, l) :=$
$sbt(BE, st, a, l)$
$a' : JFALSE a''$
$ct(\Gamma_1, st, a' + 1, l)$
$a'' - 1 : JMP a'''$
$ct(\Gamma_2, st, a'', l)$
$a''' : \ldots$

$ct($while BE do $\Gamma, st, a, l) :=$
$sbt(B, st, a, l)$
$a' : JFALSE a'' + 1$
$ct(\Gamma, st, a' + 1, l)$
$a'' : JMP a$

$et(Z, st, a, l) := a : LIT Z$

$et(I, st, a, l) :=$
  if $st(I) = (const, Z)$ then $a : LITZ$
  if $st(I) = (var, dl, off)$ then $a : LOAD(l - dl, off)$

$et(E_1 + E_2, st, a, l) :=$
  $et(E_1, st, a, l)$
  $et(E_2, st, a', l)$
  $a'' : ADD$

$sbt(E_1 < E_2, st, a, l) :=$
  $et(E_1, st, a, l)$
  $et(E_2, st, a', l)$
  $a'' : LT$

$sbt(notBE, st, a', l) :=$
  $sbt(BE, st, a, l)$
  $a' : not$

$sbt(BE_1 and BE_2, st, a, l) :=$
  $sbt(BE_1, st, a, l)$
  $sbt(BE_2, st, a', l)$
  $a'' : AND$

Satz (Korrektheit der Übersetzung) Für jedes $P \in Prog^{(n)}$ und $(z_1, \ldots, z_n), (z'_1, \ldots, z'_n) \in \mathbb{Z}$ gilt: $\mathcal{M}[\![P]\!](z_1, \ldots, z_n) = (z'_1, \ldots, z'_n) \leftrightarrow \exists \mathfrak{I}[\![\text{trans}(P)]\!](1, \varepsilon, 0 : 0 : 0 : z_1 : \ldots : z_n) = (0, \varepsilon, 0 : 0 : 0 : z'_1 : \ldots : z'_n)$

Beweis: M. Mohnen: A Compiler Correctness Proof for the Static Link Technique, Fund. Inf. 29 (1997)

Beispiel (Fakultät)

```
in/out X;
  var E;
  proc F;
    if (1<x) then
      begin
        E:=E*X;
        X:=X-1;
        F()
      end
  begin
    E:=1;
    F();
    X:=E;
  end.
```

$\text{trans}(in/outX; \Delta\Gamma) =$
$1 : CALL(a_\Gamma, 0, 1)$
$2 : JMP0;$
$bt(\Delta\Gamma, st_{I/O}, a_\Gamma, 1)$

mit $st_{I/O} = (var, 0, 1)$.

Jumping-Code für boolesche Ausdrücke Übersetzung boolescher Ausdrücke mit nicht-strikter Semantik durch Sprungbefehle anstelle logischer Befehle.

- Idee: Vererbung von Sprungzeilen für boolesche Ergebnisse
- $nbt : BExp \times Tab \times Adr^3 \dashrightarrow AM - Code$
  1. freie Anfangsadresse
  2. true-Adresse
  3. false-Adresse

mit geeigneter Modifikation der Übersetzung von Verzweigung und Iteration:

$nct(\text{if } BE \text{ then } \Gamma_1 \text{ else } \Gamma_2, st, a, l) :=$
$nbt(BE, st, a, a_t, a_f, l)$
$nct(\Gamma_1, st, a_t, l)$
$a' : JMPA'';$
$nct(\Gamma_2, st, a_f, l)$

$nct(\text{while } BE \text{ do } \Gamma, st, a, l) :=$
$nbt(BE, st, a, a_t, a_f, l)$
$nct(\Gamma, st, at, l)$
$a_t - 1 : JMPa$

$nbt(E_1 < E_2, st, a, a_t, a_f, l) :=$
$et(E_1, st, a, l)$
$et(E_2, st, a', l)$
$a'' : LT$
$a'' + 1 : JFALSEa_f$
$a'' + 2 : JMPa_t$

$nbt(notBE, st, a, a_t, a_f, l) :=$
$nbt(BE, st, a, a_f, a_t, l)$

$nbt(BE_1 and BE_2, st, a, a_t, a_f, l) :=$
$nbt(BE_1, st, a, a', a_f, l)$
$nbt(BE_2, st, a', a_t, a_f, l)$

$nbt(BE_1 or BE_2, st, a, a_t, a_f, l) :=$
$nbt(BE_1, st, a, a_t, a', l)$
$nbt(BE_2, st, a', a_t, a_f, l)$

Prozeduren mit Parametern Erweiterung von BPS um Wert- und Variablenparameter für Prozeduren.

Sukzessiver Aufbau eines Frames

1. Berechnung der aktuellen Parameter
2. Berechnen des statischen Verweises (Indexregister R)
3. Sprung zur aufgerufenen Prozedur mit Eintrag der Rücksprungadresse
4. Alten FP (Framepointer) als dl speichern
5. Speicherplatz für lokale Variablen bereitstellen

Befehlssatz (ZPP) Arithmetische, logische und Sprungbefehle wie bisher.

CALL ca	$SP \leftarrow SP-1; \langle SP\rangle \leftarrow IC+1; IC \leftarrow ca$
RET k	$IC \leftarrow \langle SP\rangle; SP \leftarrow SP+k+1$
PUSH z	$SP \leftarrow SP-1; \langle SP\rangle \leftarrow z$
PUSH FP	analog
PUSH $\langle FP\rangle$	analog
PUSH $\langle R+2\rangle$	analog
POP FP	$FP \leftarrow \langle SP\rangle; SP \leftarrow SP+1$
POP $\langle n\rangle$	$S_n \leftarrow \langle SP\rangle; SP \leftarrow SP+1$
SUB SP, n	$SP \leftarrow SP-n$
LOAD FP, SP	$FP \leftarrow SP$
LOAD $R, \langle n\rangle$	analog
LOAD $R, \langle R+2\rangle$	analog

$S_n :=$ Stackzelle mit Adresse n

Ein- / Ausgabe $P = in/outI_1, \ldots, I_n; B.$

Anfangstabelle $st_{I/O}(I_j) := (var, 0, 3+n-j)$

Zur weiteren Erläuterung siehe Folien (CB 4-19) bis (CB 4-23).

5.2 Übersetzung von Datenstrukturen

- Datenstrukturen $\curvearrowright$ Variablen mit Komponenten, strukturierter Zustandsraum
- Abstrakte Maschine: Weiterhin lineare Speicherstruktur, Speicherzellen für atomare Daten
- Übersetzungsaufgabe: Abbildung des strukturierten Zustandsraumes auf linearen Speicherbereich
  - statische Datenstrukturen: Speicherbedarf zur Übersetzungszeit bekannt

  - dynamische Datenstrukturen: Speicherbedarf laufzeitabhängig
- Heap / Halde als zusätzliche Maschinenkomponente
- Zeigertypen, garbage collection / Speicherbereinigung