Compilerbau Vorlesung – Prof. Dr. K. Indermark – Sommersemester 2004

geTEX't von Matthias Hensler (matthias@wspse.de)

Dank an Frank Häßler (kni_ght@gmx.de) für die Korrekturen und Überarbeitung des Skriptes

13. August 2005

Inhaltsverzeichnis

0 Einleitung **7**
- 0.1 Definition (Compiler) 7
- 0.2 Aspekte einer Programmiersprache 7
- 0.3 Struktur eines Compilers 8
  - 0.3.1 Analyse 8
  - 0.3.2 Synthese 8
  - 0.3.3 Frontend / Backend 8

1 Lexikalische Analyse **11**
- 1.1 Scanner-Konstruktion 12
  - 1.1.1 Das einfache Matching-Problem 13
    - DFA-Methode 13
    - NFA-Methode 14
    - Kombination von NFA- und DFA-Methode 14
  - 1.1.2 Das erweiterte Matching-Problem 14
    - Konventionen für Eindeutigkeit 14
    - Berechnung der flm-Analyse 15
- 1.2 Praktische Aspekte der Scanner-Konstruktion 17
- 1.3 Automatische Scannergenerierung mit `flex` (`lex`) 18
  - 1.3.1 Aufbau einer Lex-Spezifikation 18
  - 1.3.2 Verwandte Probleme 19

2 Syntaktische Analyse **27**
- 2.1 Kontextfreie Grammatiken 27
  - 2.1.1 l-Analyse, r-Analyse 28
  - 2.1.2 Syntaxanalyse 29
- 2.2 Top-Down Analyse mit $LL(k)$-Grammatiken 29
  - 2.2.1 Der Fall $k = 1$: 32
  - 2.2.2 Berechnung der la-Mengen 33
  - 2.2.3 Der deterministische TD-Analyseautomat $DTA(\mathcal{G})$ für $\mathcal{G} \in LL(1)$ 35
- 2.3 Parserkonstruktion nach TD-Methode 35
  - 2.3.1 Transformationen nach $LL(1)$ 36
    - Beseitigung von Linksrekursionen 36
    - Komplexität der LL(1)-Analyse 37
    - Links-Faktorisieren 38
- 2.4 Top-Down Analyse mit rekursiven Prozeduren 38

2.5 Bottom-Up Analyse mit $LR(k)$-Grammatiken . . . 39
2.5.1 Nicht-Determinismus . . . 39
2.5.2 $LR(0)$-Grammatiken . . . 40
Berechnung der $LR(0)$-Mengen einer Grammatik . . . 40
Die $\underline{\text{goto}}$-Funktion . . . 41
Berechnung der LR(0)-Mengen und $\underline{\text{goto}}$-Funktion durch Potenzmengenkonstruktion nicht-deterministischer endlicher Automaten . . . 41
Konstruktion des deterministischen BU-Analyseautomaten für $\mathcal{G} \in LR(0)$ . . . 42
2.5.3 $SLR(1)$-Analyse . . . 43
Die $SLR(1)$-action Funktion . . . 43
2.5.4 $LR(1)$-Analyse . . . 44
Berechnung der $LR(1)$-Mengen . . . 44
Die $LR(1)$-action Funktion von $\mathcal{G}$ . . . 44
2.5.5 $LALR(1)$-Analyse . . . 45
Die $LALR(1)$-action Funktion von $\mathcal{G}$ . . . 45
2.6 Bottom-Up Analyse mehrdeutiger Grammatiken . . . 46
2.7 Ergebnis . . . 47

3 Semantische Analyse, Attributgrammatiken **77**
3.1 Attributgrammatiken . . . 77
3.1.1 Lösbarkeit von E_t . . . 80
3.1.2 Ein Zirkularitätstest für Attributgrammatiken . . . 81
Komplexität des Zirkularitätstest . . . 82
3.1.3 Stark-nichtzirkuläre Grammatiken . . . 82
3.1.4 Attributberechnung . . . 82
3.2 S-Attributgrammatiken . . . 83
3.3 L-Attributgrammatiken . . . 85
3.3.1 Syntaxanalyse mit L-Attributauswertung . . . 85
3.3.2 Anwendung von LAG . . . 86

4 Übersetzung in Zwischencode **99**
4.1 Übersetzung von Ausdrücken, Anweisungen, Blöcken und Prozeduren . . . 99
4.1.1 Semantik von BPS . . . 100
4.1.2 Zwischencode für BPS . . . 100
Berechnung des statischen Verweises . . . 101
4.1.3 Übersetzung von BPS-Programmen in AM-Code . . . 102
Aufbau der Symboltabelle . . . 102
Anfangstabelle . . . 103
4.1.4 Die Übersetzung . . . 103
Blockübersetzung . . . 104
Deklarationsübersetzung . . . 104
Anweisungsübersetzung . . . 104
4.1.5 Jumping Code für boolesche Ausdrücke . . . 106
4.1.6 Prozeduren mit Parametern . . . 107
Alternative zur Verweiskettentechnik: Display-Technik . . . 109
4.2 Übersetzung von Datenstrukturen . . . 109
4.2.1 Statische Datenstrukturen . . . 110

Erläuterung der Symboltabelle . . . 110
Aufbau einer Symboltabelle . . . 110
4.2.2 Dynamische Datenstrukturen . . . 111

A Exkurs: Yacc **143**
A.1 Automatische Parsergenerierung mit Yacc . . . 143
A.1.1 Aufbau einer Yacc-Spezifikation . . . 143
A.2 Behandlung von Konflikten in Yacc . . . 144
A.3 Präzedenz von Operatoren in Yacc . . . 145
A.3.1 Exkurs: Kodierung der Präzedenzregeln für arithmetische Ausdrücke über der Grammatik . . . 145

Kapitel 0

Einleitung

0.1 Definition (Compiler)

Ein Compiler ist ein Programm zur Übersetzung von *PS-Programmen* (Quellprogramme) in äquivalente *MS-Programme* (Zielprogramme). Wir fordern hierbei, daß der Compiler korrekt arbeitet. [20.04.2004]
PS bezeichne hierbei eine höhere Programmiersprache, bei denen wir folgende Typen unterscheiden:

imperative PS: Variablen, Wertzuweisungen, Kontrollstrukturen, Datenstrukturen

deklarative PS: funktionale und logische Sprachen (→ Vorlesung „Formale Sprachen“)

nebenläufige PS: kommunizierende Prozesse, verteilte Systeme

objektorientierte PS: Klassen, Vererbung

Wir betrachten hier *imperative* Programmiersprachen.
MS bezeichne die Maschinensprache auf dem Grundkonzept des *von-Neumann-Rechners*. Die Menge der elementaren Maschinenbefehle läßt sich dabei wie folgt aufteilen:

RISC: **R**educed **I**nstruction **S**et **C**omputer

CISC: **C**omplex **I**nstruction **S**et **C**omputer

0.2 Aspekte einer Programmiersprache

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 (compiler options)

Definition: *Äquivalenz von Programmen*: Zwei Programme sind äquivalent, wenn sie semantisch gleich sind.

0.3 Struktur eines Compilers

Ein Compiler ist in logisch unabhängige Phasen aufgeteilt, die aber verzahnt als *Passes* (Läufe) ablaufen. Wir unterscheiden die *Analysephase* und die *Synthesephase*. Eine Übersicht über das Zusammenspiel der einzelnen Phasen gibt Abbildung 1.

0.3.1 Analyse

Bei der Analyse wird die syntaktische Struktur bestimmt und Fehler erkannt. Sie wird unterteilt in folgende Schritte:

lexikalische Analyse: Erkennung von Symbolen, Trennzeichen und Kommentaren (implementierbar durch endliche Automaten durch Erkennung von regulären Ausdrücken)

syntaktische Analyse: Erkennung des hierarchischen Programmaufbaus und Darstellung des Aufbaus durch einen Ableitungsbaum (implementierbar durch CFGs (kontextfreie Grammatiken) und Kellerautomaten)

semantische Analyse: Kontextabhängigkeiten, statische Semantik, Typinformationen, attributierte Grammatiken ergeben dann eine Attributierung des Ableitungsbaums

0.3.2 Synthese

Die Synthese erzeugt MS-Code aus dem attributierten Ableitungsbaum, und läßt sich in die folgenden Schritte unterteilen:

Zwischencodeerzeugung: Übersetzung in Zwischencode für eine abstrakte Maschine (kann auch fehlen, erhöht bei Vorhandensein die Portabilität)

Optimierung: Verbesserung von Laufzeit und Speicherbedarf

Codegenerierung: effiziente Verwendung von Registern und MS-Befehlssatz zur Erzeugung von MS-Code

0.3.3 Frontend / Backend

Beim Compiler unterscheiden wir zwischen dem *Frontend*, welches MS-unabhängig arbeitet und dem MS-abhängigen *Backend*. Das Frontend ist zuständig für die Analyse und die Zwischencodeerzeugung. Ggf. wird eine MS-unabhängige Optimierung vorgenommen. Das Backend schließlich generiert dann den MS-Code und nimmt MS-abhängige Optimierungen vor.

Definition: One-Pass, *n*-Pass-Compiler, wobei n ist die Anzahl der Läufe durch das Quellprogramm ist

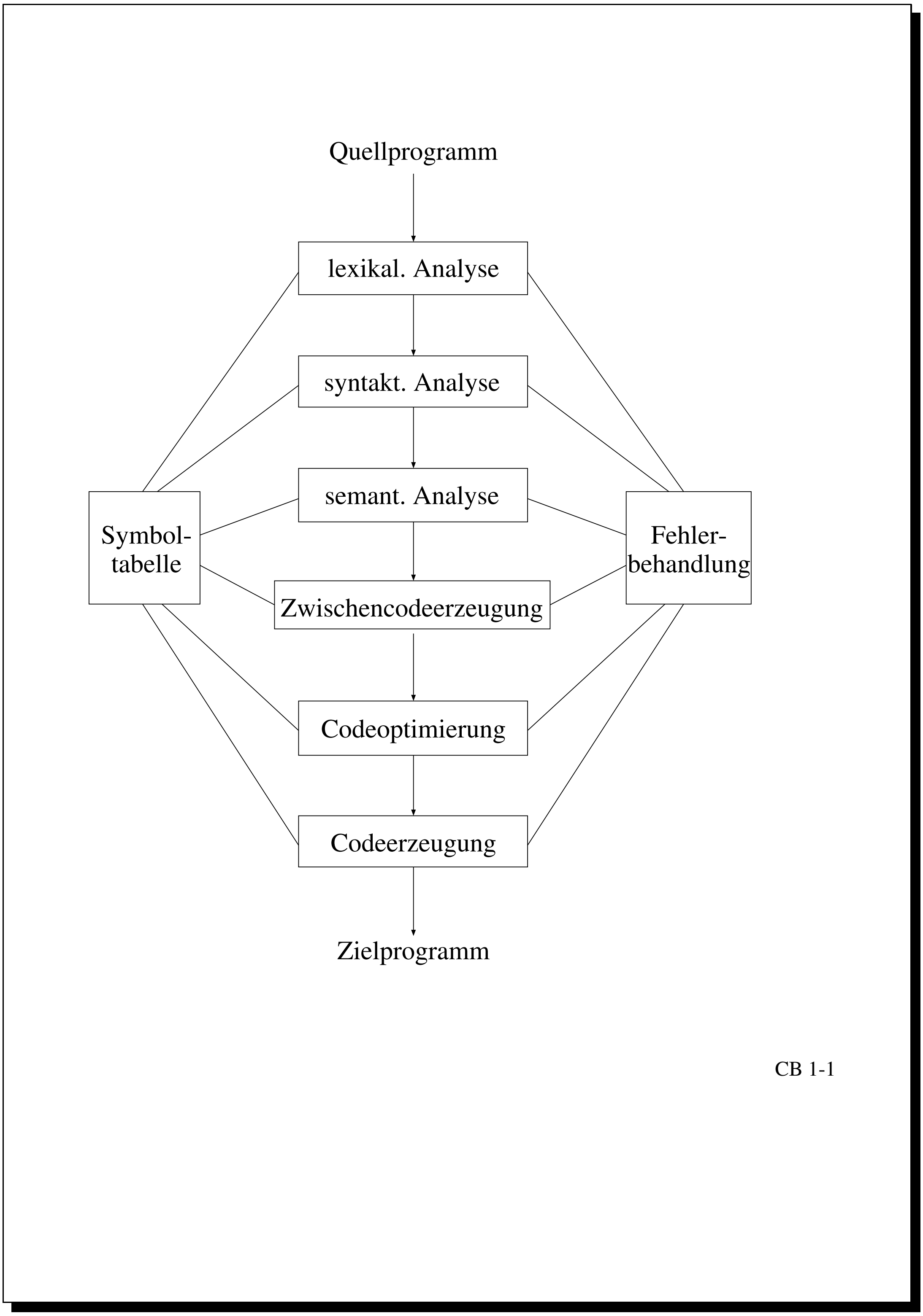


Abbildung 1: Struktur eines Compilers

Literatur

1. Wirth: *Compilerbau*, Teubner Studienbücher, 1977 (Pascal)
   4. Auflage, 1986 (Modula-2)
2. Aho/Sethi/Ullman: *Compilers — Principles, Techniques, and Tools*, Addison-Wesley, 1986
3. Waite/Goos: *Compiler Construction*, Springer, 1985
4. Wilhelm/Maurer: *Übersetzerbau: Theorie, Konstruktion, Generierung*, Springer-Lehrbuch, 2. überarb. u. erw. Aufl., 1997
5. Wirth: *Grundlagen und Techniken des Compilerbaus*, Addison–Wesley, 1996
6. Levine et al.: *lex&yacc*, O'Reilly, 2. Aufl., 1992
7. Schöning: *Theoretische Informatik – kurzgefaßt*, Spektrum, 2001

Abbildung 2: Literatur

Kapitel 1

Lexikalische Analyse

Im Folgenden wird davon ausgegangen, daß das zu übersetzende Programm als Characterstring vorliegt. Wichtig ist hier die Unterscheidung zwischen Zeichen und Symbolen.

Ausgangspunkt: Quellprogramm P als Zeichenfolge, mit:

$$\begin{array}{rl} \Sigma_0: & \text{Zeichensatz (ASCII, Unicode)} \\ a \in \Sigma_0: & \text{Zeichen, lexikalisches Atom} \\ P \in \Sigma_0^*: & \text{Quellprogramm} \end{array}$$

$P \in \Sigma_0^*$ besitzt aufgrund der Pragmatik von PS eine *lexikalische Struktur*:

- natürliche Sprache für Bezeichner und Schlüsselwörter
- mathematische Formelsprache für Zahlen, Formeln
- Leerzeichen, Zeilenwechsel, Einrückungen (Tabulatoren), ...
- Kommentare, als auch direkte Compileranweisungen (Pragmas)

Der syntaktische Aufbau ermöglicht eine gute Lesbarkeit und Wartbarkeit, die für den Compiler allerdings irrelevant ist.
Die Semantik von P und damit die Übersetzung (als Semantik erhaltender Prozess) ist syntaxorientiert: sie folgt dem hierarchischen Programmaufbau. Dabei ist der pragmatische Aspekt irrelevant.

1. Beobachtung: Syntaktische Atome (*Symbole*) werden dargestellt als Folgen lexikalischer Atome, sogenannter *Lexeme* (Zeichenreihen die Symbole darstellen). Die erste Aufgabe der lexikalischen Analyse ist damit die Zerlegung des Quellprogramms P in eine Folge von Lexemen.

2. Beobachtung: Für die syntaktische Analyse ist der Unterschied von Lexemen oft irrelevant (z.B. Bezeichner). Lexeme werden daher zu *Symbolklassen* zusammengefaßt. Diese Symbolklassen werden dann als *Token* (zum Beispiel `id`, `num`) dargestellt. Es ist beispielsweise unerheblich, ob an einer Stelle der Wert 10 oder der Wert 20 steht, lediglich die Symbolklasse (beispielsweise Zahlwort, Bezeichner, etc.) ist relevant. Bezeichner müssen nicht unterschieden werden.

1. Aufgabe: Zerlegung des Quellprogramms in eine Folge von Lexemen.

Die syntaktische Analyse bearbeitet eine Tokenfolge. Ein Symbol wird durch ein zusätzliches *Attribut* für die semantische Analyse und die Codegenerierung identifiziert. Symbol = (Token, Attribut).

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

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

Definition: *Scanner* (in der Literatur auch *Lexer*): Der Scanner ist ein Programm für die lexikalische Analyse (siehe Abbildung 1.1).
Wichtigste Symbolklassen (Lexemklassen):

- *Bezeichner* (Zeichenreihen mit Ausnahme von Schlüsselwörter)
- *Zahlwörter*
- *Schlüsselwörter* (Identifizierung über Symboltabelle)
- *Einfache Symbole*: ein Sonderzeichen $(+,-,*,(,.,;,\dots)$, bildet jeweils eine Symbolklasse
- *Zusammengesetzte Symbole*: Folge von zwei oder mehr Symbolzeichen $(:=,**,<=,\dots)$, bilden ebenfalls eine Symbolklasse
- *Leerzeichen*: ␣, ␣. . . ␣, . . .
- *Spezielle Symbole*: Kommentare, Pragmas von Compileroptionen

Definition: *Token*: $id, const, divsym, semsym, leer$

Definition: *Attribute*: Zeiger in Symboltabelle, Binärdarstellung einer Zahl, leer bei Symbolklassen mit einem Symbol

Feststellung: Symbolklassen sind reguläre Mengen → Beschreibung durch reguläre Ausdrücke; Erkennung durch endliche Automaten. Automatische Scannergenerierung, z.B. mit `lex` (UNIX).

1.1 Scanner-Konstruktion

[21.04.2004]

(Hinweis: in früheren Vorlesungen wurden reguläre Ausdrücke mit „*RA*“ bezeichnet)

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

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

Ihre Semantik:

- $[\![.]\!] : \mathrm{RegE}(\Sigma) \to \mathfrak{p}(\Sigma^*)$
- $[\![\Lambda]\!] := \emptyset$

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

Der folgende Teil bis zum einfachen Matching Problem ist nicht Bestandteil der Vorlesung im Sommersemester 2004.

- $[\![\Lambda^*]\!] := [\![\Lambda]\!]^* = \emptyset^*$
- $L^* = \bigcup_{n=0}^{\infty} L^n, L^n = \{w_1 w_2 \ldots w_n | w_i \in L\} = \overbrace{L \cdot L \cdot L \cdot \ldots \cdot L}^{n}$
- $L^n \cdot L^m = L^{n+m}, L^0 \cdot L^m = L^m, L^0 := \{\varepsilon\}$

1.1.1 Das einfache Matching-Problem

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

$$\mathfrak{A} =< Q, \Sigma, \delta, q_0, F >\in \text{NFA}(\Sigma)$$

mit $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to \mathfrak{p}(Q)$, $\Sigma_\varepsilon := \Sigma \cup \{\varepsilon\}$ (Epsilon-Transitionen sind zugelassen)
und $q_0 \in Q, F \subseteq Q$.

Definition: Für $T \subseteq Q$ ist die ε-*Hülle* $\varepsilon(T)$ definiert durch

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

Definition: Die *erweiterte Transitionsfunktion* $\overline{\delta} : P(Q) \times \Sigma^* \to P(Q)$ ist definiert durch:

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

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

DFA-Methode

Die DFA-Methode zur Lösung des Matching-Problems (bekannt aus der ATFS-Vorlesung) ist wie folgt definiert:

$$\alpha \in \text{RegE}(\Sigma) \xrightarrow{(1)} \mathfrak{A}(\alpha) \in \text{NFA}(\Sigma) \xrightarrow{(2)} \mathfrak{A}(\alpha)^P \in \text{DFA}(\Sigma)$$

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 Platz und Zeitbedarf

Trotz aufwendiger Konstruktion, entscheidet $\mathfrak{A}(\alpha)^P$ in $|w| + 1$ Schritten ob $w \in [\![\alpha]\!]$ oder nicht.

NFA-Methode

Die NFA-Methode zur Lösung des Matching-Problems verbessert den Platzbedarf, benötigt dafür jedoch eine längere Laufzeit. Der Automat wird in Bezug auf die zu prüfende Eingabe w konstruiert, durch Verzicht auf volle Potenzmengen-Konstruktion. Da die Eingabe $w \in \Sigma^*$ bekannt ist, reicht die Potenzmengen-Konstruktion für den „Lauf von w durch $\mathfrak{A}(\alpha)$“.
Der Algorithmus ist in Abbildung 1.3 gegeben.
Die Komplexität des Algorithmus ist wie folgt gegeben:

Platz: $O(|\alpha| + |w|)$

Zeit: $O(|\alpha| \cdot |w|)$

Kombination von NFA- und DFA-Methode

Es ist eine Optimierung möglich, indem bereits berechnete Transitionen $\overline{\delta}(T, a)$ im Cache zwischengespeichert werden.

1.1.2 Das erweiterte Matching-Problem

Definition: Seien $\alpha_1, \ldots, \alpha_n \in \text{RegE}(\Sigma)$ und $w \in \Sigma^*$.
Sei ferner $\Delta := \{T_1, \ldots, T_n\}$ ein Alphabet von Token zur Beschreibung der Zugehörigkeit zu einem Ausruck.
Wenn $w = w_1 w_2 \ldots w_k$ und $w_j \in [\![\alpha_{i_j}]\!]$ mit $1 \leq i_j \leq n$ für $j = 1 \ldots k$, dann heißt $(w_1, \ldots, w_k)$ eine *Zerlegung von w bezüglich* $\alpha_1, \ldots, \alpha_n$ und $v = T_{i_1} \ldots T_{i_k}$ eine *Analyse von w bezüglich* $\alpha_1, \ldots, \alpha_n$. v repräsentiert die lexikalische Struktur von w bezüglich $\alpha_1, \ldots, \alpha_n$.

Aufgabe: Analyse bestimmen bzw. Fehler melden.

Hinweis: Weder die Analyse noch die Zerlegung sind eindeutig bestimmt.

Konventionen für Eindeutigkeit

1. Prinzip des längsten Match (*maximal munch*) für die Eindeutigkeit der Zerlegung.

   Definition: Eine Zerlegung $(w_1, \ldots, w_k)$ von w bezüglich $\alpha_1, \ldots, \alpha_n$ heißt *lm-Zerlegung* („longest match“), wenn für alle $j = 1 \ldots k$ und $x, y \in \Sigma^*$ und $p, q \in \{1, \ldots, n\}$ gilt:

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

   Folgerung: Für $w, \alpha_1, \ldots, \alpha_n$ gibt es höchstens eine lm-Zerlegung. Motivation für die lm-Zerlegung ist unter anderem das Finden des längsten Bezeichners.

2. Prinzip des ersten Match (bzgl. $\alpha_1, \ldots, \alpha_n$) für die Eindeutigkeit der Analyse.

   Trotz der Eindeutigkeit der lm-Zerlegung sind mehrere Analysen möglich, weil $[\![\alpha_p]\!] \cap [\![\alpha_q]\!] \neq \emptyset$ möglich.

Konvention: der erste Match in der *Folge* $\alpha_1, \ldots, \alpha_n$ zählt.

Definition: Sei $(w_1, \ldots, w_k)$ 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 $\mu = 1, \ldots, n$ gilt:

$$w_j \in [\![\alpha_\mu]\!] \curvearrowright i_j \leq \mu$$

Folgerung: Für $w, \alpha_1, \ldots, \alpha_n$ gibt es höchstens eine flm-Analyse. Sie existiert genau dann, wenn die lm-Zerlegung existiert.

Berechnung der flm-Analyse

[23.04.2004]

Als Voraussetzung gelte im Folgenden: $\alpha_1, \ldots, \alpha_n \in \mathrm{RegE}(\Sigma)$, o.B.d.A. $\varepsilon \notin [\![\alpha_i]\!] \neq \emptyset$, für $i = 1, \ldots, n$. Sei ferner $w \in \Sigma^*$ und $\Delta = \{T_1, \ldots, T_n\}$ das Tokenalphabet.

1. Konstruiere für $i = 1, \ldots, n$:

$$\mathfrak{A}_i = < Q_i, \Sigma, \delta_i, q_0^{(i)}, F_i > \in \mathrm{DFA}(\Sigma)$$

so daß $[\![\alpha_i]\!] = L(\mathfrak{A}_i)$

2. Bilde aus diesen den Produktautomaten $\mathfrak{A} = < Q, \Sigma, \delta, q_0, F > \in \mathrm{DFA}(\Sigma)$, mit

$$\begin{array}{rcl} Q & := & Q_1 \times \ldots \times Q_n \\ q_0 & := & (q_0^{(1)}, \ldots, q_0^{(n)}) \\ \delta((q^{(1)}, \ldots, q^{(n)}), a) & := & (\delta_1(q^{(1)}, a), \ldots, \delta_n(q^{(n)}, a)) \\ (q^{(1)}, \ldots, q^{(n)}) \in F & :\curvearrowleft\!\!\curvearrowright & \exists i \in \{1, \ldots, n\}, q^{(i)} \in F_i \end{array}$$

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

Zerlege F wegen der Eigenschaft „first match" in $F = \dot\bigcup_{i=1}^n F^{(i)}$ durch die Forderung

$$(q^{(1)}, \ldots, q^{(n)}) \in F^{(i)} :\curvearrowleft\!\!\curvearrowright q^{(i)} \in F_i \text{ und } q^{(j)} \notin F_j \ \forall 1 \leq j < i$$

Dann gilt:

$$\overline{\delta}(q_0, w) \in F^{(i)} \curvearrowleft\!\!\curvearrowright w \in [\![\alpha_i]\!] \text{ und } w \notin \bigcup_{j=1}^{i-1} [\![\alpha_j]\!]$$

Definition: $q \in Q$ heißt *produktiv* $:\curvearrowleft\!\!\curvearrowright \exists v \in \Sigma^* : \overline{\delta}(q, v) \in F$

P ist die Menge der produktiven Zustände, also: $F \subseteq P$.

Bei der Minimierung des Automaten gibt es höchstens einen unproduktiven Zustand, nämlich den, der die leere Sprache akzeptiert.
Minimierung des Automaten: Einteilung in Äquivalenzklassen

3. Erweitere $\mathfrak{A} = < Q, \Sigma, \delta, q_0, F >$ zu dem *Backtrack-DFA* $\mathfrak{B}$ mit Ausgabe.

   Idee: Einweg-Leseband mit zwei Köpfen:

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

   Konfigurationsmenge von $\mathfrak{B}$: $\underbrace{(\{N\} \cup \Delta)}_{Mode} \times \underbrace{\Sigma^* Q \Sigma^*}_{Eingabe} \times \underbrace{\Delta^* \{\varepsilon, lexerr\}}_{Ausgabe}$

   Anfangskonfiguration für $w \in \Sigma^*$: $(N, q_0 w, \varepsilon)$

   Transitionen: $q' := \delta(q, a)$

   (a) normal mode (Match suchen):

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

   (b) backtrack mode (längsten Match suchen):

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

   (c) Eingabeende

   $$\begin{array}{lcll} (N, q, W) & \vdash & \text{Ausgabe} : W \cdot lexerr & \text{falls} \quad q \in P \backslash F \\ (T, q, W) & \vdash & \text{Ausgabe} : W \cdot T & \text{falls} \quad q \in F \\ (T, vaq, W) & \vdash & (N, q_0 va, WT) & \text{falls} \quad q \in P \backslash F \end{array}$$

   Dann gilt für $w \in \Sigma^*$:

   $$\begin{array}{lcll} (N, q_0 w, \varepsilon) & \overset{*}{\vdash} & W \in \Delta^* & \curvearrowleft\!\!\curvearrowright \; W \text{ ist flm} - \text{Analyse von } w \\ (N, q_0 w, \varepsilon) & \overset{*}{\vdash} & W \cdot lexerr & \curvearrowleft\!\!\curvearrowright \; \text{es gibt keine flm} - \text{Analyse von } w \end{array}$$

   Der Zeitaufwand beträgt im worst-case $O(|w|^2)$.

   Beispiel: $\alpha_1 = abc \quad \alpha_2 = (abc)^* d \quad w = (abc)^m$ erfordert $O(m^2)$ Schritte.

   Verbesserung durch Tabular-Methode (vgl. KMP für string pattern matching) in Linearzeit.

   Lit: TH. Raps „Maximal Munch Tokenization in Linear Time“ ACM-TOPLAS 20(1998)

1.2 Praktische Aspekte der Scanner-Konstruktion

[27.04.2004]

Unterscheide

- automatische Scannererzeugung (lex, u.a.) mit bel. $\alpha_1, \dots, \alpha_n$
- Scanner-Konstruktion für eine PS mit speziellem $\alpha_1, \dots, \alpha_n$

- Spracherweiterung von RegE(Σ) zur einfacheren Beschreibung von Symbolklassen (Abkürzungen, reguläre Definition)

  a. Vereinfachende Bezeichnungen:
     - Präzedenzregeln zur Vermeidung von Klammern: $* > \cdot > \vee$ ($*$ bindet stärker als $\cdot$)
       * $\cdot$ und $\vee$ sind linksassoziativ
       * $\cdot$ wird weggelassen, $|$ statt $\vee$
       * ***Beispiel:*** : $(a \vee ((b^*) \cdot c))$, einfacher: $a|b^*c$
     - Abkürzungen:
       * $\alpha^+ := \alpha\alpha^*$ („einmal oder mehrmals“)
       * $\alpha? := \alpha|\Lambda^*$ („einmal oder keinmal“)
       * $[abc] := a|b|c$
       * $[a-z] := a|b|c|\dots|z$
       * $. := a|\dots|z$ ($\Sigma = \{a, \dots, z\}$) Der Punkt steht für alle Zeichen

  b. Reguläre Definitionen: Schrittweise Beschreibung von Symbolklassen durch zusätzliche freigewählte Bezeichner.
  $$\begin{array}{ll} \underline{id}_1 = \alpha_1 & \text{mit } \underline{id}_1, \dots, \underline{id}_n \notin \Sigma \\ \vdots & \\ \underline{id}_n = \alpha_n & \text{und } \alpha_i \in \text{RegE}(\Sigma \cup \{\underline{id}_1, \dots, \underline{id}_{i-1}\} \end{array}$$
  Beachte: keine Rekursion, Entschachtelung möglich (Rekursion $\Rightarrow$ EBNF)

- Scannereffizienz wichtig: häufiger Aufruf des Scanners bei der syntaktischen Analyse durch „nexttoken“.

  - Programmierung des Scanners in Assembler
  - Erzeugung durch Scannergenerator (z.B. lex)

- Prinzip des längsten Match. Allgemein: beliebig langer look-ahead (Backtrack-Phase erforderlich). Look-ahead von 2 Zeichen in Pascal, allerdings kann diese Beschränkung die lexikalische Analyse verändern.

  Abweichungen vom lm-Prinzip möglich, z.B. in lex: α/β Match von α erfordert zusätzlich einen look-ahead-string, welcher β matcht.

- Besondere Rolle von „blanks“ (Whitespace): Trennung von eigentlichen Lexemen, um mit 1-lookahead auszukommen.

- Attributberechnung:

  - erneutes Lesen des Eingabestrings

- Dezimalzahl ⇝ Binärzahl (Lex: `install_num`)
- Bezeichner: Gleichheit prüfen, Verwendung einer Symboltabelle (Lex: `install_id`), Eintrag eines Bezeichers in LexTab, Zeiger als Attribut.
- Schlüsselwörter (alternativ) zunächst als Bezeichner behandeln, Symboltabelle mit Schlüsselwörtern initialisieren. Die Attributberechnung (`install_id`) liefert dann den Schlüsselwerttoken (z.B. (if,) statt (id, → SymTab)).

1.3 Automatische Scannergenerierung mit **flex** (**lex**)

Linux: > `man flex`, Unix-Tool („ **f**ast **lex**ical analyser“)

$$\underbrace{\texttt{scan.l}}_{\text{Lex-Spezifikation}} \xrightarrow{lex} \underbrace{\texttt{lex.yy.c}}_{\text{Scanner in C}} \xrightarrow{cc} \underbrace{\texttt{a.out}}_{\text{Ausführbarer Scanner}}$$

Programm $\xrightarrow{\texttt{a.out}}$ Symbolfolge Tokenfolge mit Atributberechnung

1.3.1 Aufbau einer Lex-Spezifikation

```
Definitionen
%%
Regeln
%%
C Hilfsprozeduren
```

Definitionen (optional)

1. Direkter C-Code `%{...C-Code ...%{`
2. Substitutionen (reguläre Definition)
3. Startzustände (zur Einleitung von besonderen Phasen)

Regeln

`muster {aktion}`

`muster` von der Form regexp[/regexp]
`aktion`: C-Code zur Berechnung von <Token, Attr>

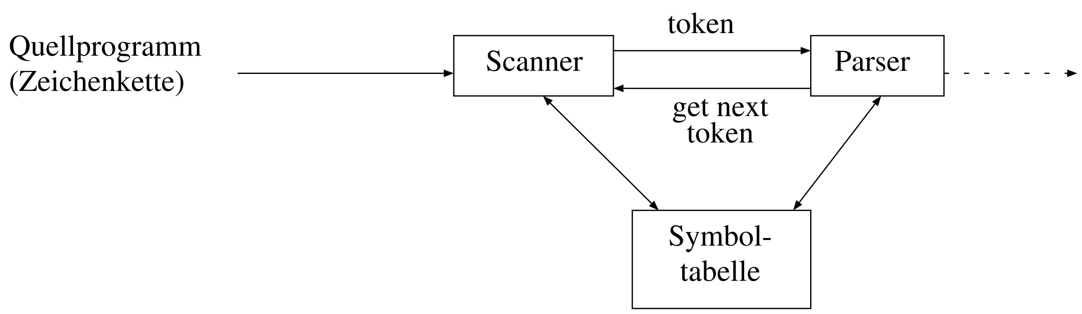
Finden der passenden Regel:

- longest match
- first match
- kein match ⟶ abfangen mit `.|\n` ⇝ Fehlerausgabe

Token werden als ganze Zahlen kodiert. *Attribute* werden über die globale Variable `yylval` weitergeleitet. Am Fileende wird die benutzerdefinierte Prozedur „`int yywrap (void)`“ aufgerufen. Beim Rückgabewert 1 liefert `lex` eine „0“ um zu charakterisieren, daß kein Token mehr folgt. Ist der Rückgabewert 0, so scannt `lex` weiter.

1.3.2 Verwandte Probleme

Teilwortsuche in Texten, Dokumenten, Dateien
Beispiel: (Unix) `grep`: **G**et **R**egular **E**xpression **P**rint
Syntax: `grep expr files`
Ausgabe: Zeilen der Dateien, in denen ein Match von `expr` vorkommt.



Lexeme $\xrightarrow{\text{Scanner}}$ Symbole (Token, Attribut)

Beispiel:

`...␣␣␣ALPHA1:=AL2␣DIV␣2.7␣␣;␣...`

$\Downarrow$

...(*id*,p_1)(*getsym*,)(*id*,p_2)(*divsym*,)(*const*,2.7_2)(*semsym*,)...

Hilfsmittel der Scannerkonstruktion:

- Reguläre Ausdrücke und endliche Automaten

CB 1-2

Abbildung 1.1: Scanner

Beispiel: Pascal / Teil 1: Symbolklassen

- Bezeichner:
  - letter = `[A-Za-z]`
  - digit = `[0-9]`
  - id = letter (letter | digit)*
- Zahlwörter: (ohne Vorzeichen)
  - digits = digit+
  - optfrac = (.digits)?
  - optexp = (`E[+-]`? digits)?
  - num = digits optfrac optexp
- Relat. Oper.:
  - relop = `<` | `<=` | `=` | `<>` | `>` | `>=`
- Schlüsselwörter:
  - if = `if`
  - then = `then`
  - else = `else`
- Zwischenraum:
  - blank = `[␣\t\n]`
  - blanks = blank+

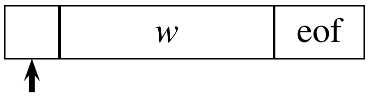
CB 1-3

Abbildung 1.2: Symbolklassen in Pascal, Teil I

NFA–Methode

Eingabe: $\mathfrak{A} \in \mathrm{NFA}(\Sigma)$

$w \in \Sigma^*$ als

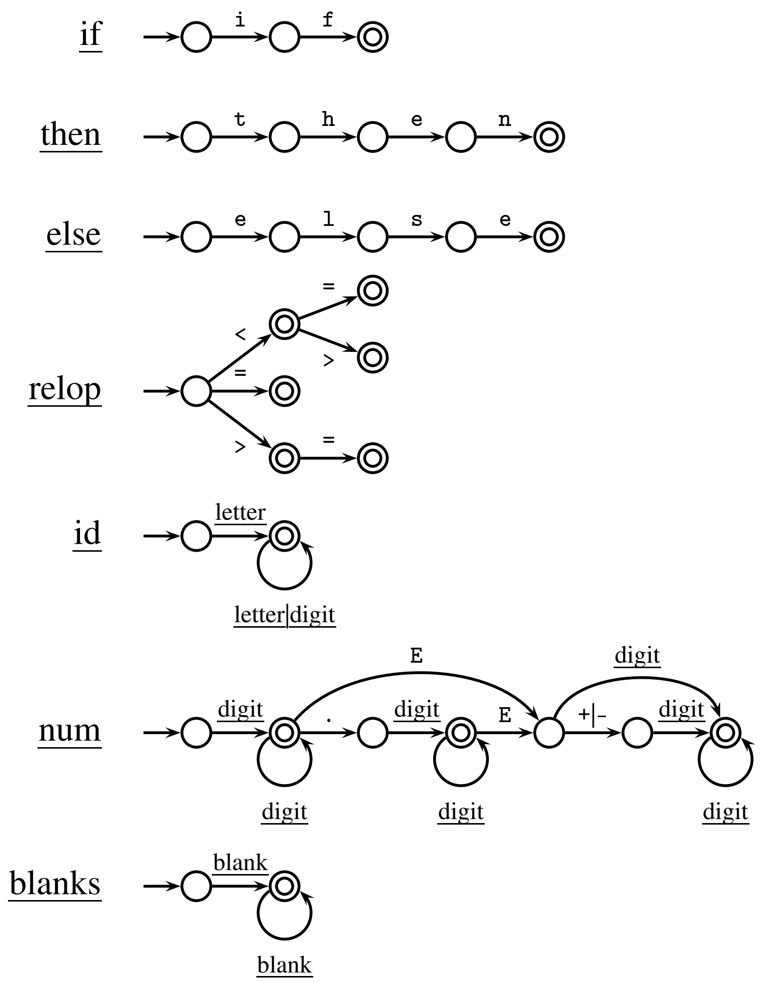


Algorithmus:

$\mathtt{T} := \varepsilon(\{q_0\});$
`a := nextchar();`
$\underline{\mathtt{while}}\ \mathtt{a} \neq \mathtt{eof}\ \underline{\mathtt{do}}$
  $\mathtt{T} := \varepsilon(\bigcup_{q \in T} \delta(q, a));$
  `a := nextchar();`
$\underline{\mathtt{endwhile}}$
$\underline{\mathtt{if}}\ \mathtt{T} \cap \mathtt{F} \neq \emptyset\ \underline{\mathtt{then}}\ \mathtt{ACCEPT}$
$\qquad\qquad \underline{\mathtt{else}}\ \mathtt{REJECT}$

Abbildung 1.3: NFA-Methode

Beispiel: Pascal / Teil 2: DFA's (partiell)



CB 1-5

Abbildung 1.4: Pascal, Teil II

Reguläre Ausdrücke in `lex`

Syntax	Semantik
normale Zeichen	das Zeichen
`\n`,`\t`, `\123`, etc.	wie in `C`
`.`	alle Zeichen außer `\n`
`[`*Zeichen*`]`	eins der *Zeichen*; Bereiche `0-9` mögl.
`[^`*Zeichen*`]`	keins der *Zeichen*
`\\`, `\.`, `\[`, etc.	`\`, `.`, `[`, etc.
`"`*Text*`"`	*Text* ohne Interpr. von `.`, `[`, `\`, etc.
`^`	als 1.Zeichen eines RA: Zeilenanfang; verbraucht kein Zeichen
`$`	als 1.Zeichen eines RA: Zeilenende
`{`*S*`}`	RA zu Substitution *S*
r`?`	ein– oder keinmal RA *r*
r`*`	keinmal, einmal oder mehrmals RA *r*
r`+`	einmal oder mehrmals RA *r*
r`{`n,m`}`	n bis m mal RA r ($,m$ optional)
`(`*r*`)`	*r*
$r_1 r_2$	Konkatenation
$r_1 \mid r_2$	Alternative

Präzedenzregeln wie üblich

CB 1-6

Abbildung 1.5: reguläre Ausdrücke in `lex`

```
%{
  #include <stdio.h>
  int yywrap();
  enum token {IF=1, ID, RELOP, LT, ...}
%}

LETTER  [A-Za-z]
DIGIT   [0-9]
L_OR_D  {LETTER}|{DIGIT}

%%
"if"              { return IF; }
"<"               { yylval = LT; return RELOP; }
{LETTER}{L_OR_D}* { install_id(); return ID; }
.|\n              { fprintf(stderr,"Ungültiges Zeichen'%c'\n",yytext[0]); }

%%
int yywrap() { return 1; }
void main(void) {
  int token;
  while (token=yylex()) printf("Token %d\n",token);
}
... install_id() {...} /* Zugriff auf Bezeichner: Stringvariable yytext. */
```

CB 1-7

Abbildung 1.6: Scannergenerierung mit `flex`

Kapitel 2

Syntaktische Analyse

[30.04.2004]

Aufgabe: Zerlegung der Symbolfolge, die der Scanner ausgibt, in syntaktische Einheiten, bzw. (wenn das nicht möglich ist) Behandlung syntaktischer Fehler.

Syntaktische Einheiten: Variablen, Ausdrücke, Anweisungen, ...

Beachte: Schachtelung syntaktischer Einheiten, Baumstruktur im Unterschied zur linearen Symbolfolge (linearen Strukturfolge).

Definition: *Parser*: Ein Parser ist ein Programm für die syntaktische Analyse.

Schnittstellen: siehe Abbildung 2.4
Beschreibung der syntaktischen Struktur durch eine *kontextfreie Grammatik* (BNF, EBNF, Syntaxdiagramme). Erkennung und Analyse durch *Kellerautomaten* mit Ausgabe.

Problem: deterministische Simulation.

Allgemeiner Fall: beliebige CFG (kontextfreie Grammatik): Tabularverfahren von Cocke, Yonnger, Kasami (CYK) mit $O(n^2)$ Platz- und $O(n^3)$ Zeitbedarf.

Im Fall einer Programmiersprache: spezielle CFG: Analyse durch deterministische Kellerautomaten mit „input look-ahead", bei linearem Platz- und Zeitbedarf.

a. *Top-Down-Analyse*: Konstruktion des Ableitungsbaums, von der Wurzel zu den Blättern hin, in Form einer *Linksanalyse*.

b. *Bottom-Up-Analyse*: umgekehrt, gespiegelte *Rechtsanalyse*

2.1 Kontextfreie Grammatiken

Für kontextfreie Grammatiken $\mathcal{G} =< N, \Sigma, P, S >\in \text{CFG}(\Sigma)$ gelten im weiteren Verlauf folgende Bezeichnungskonventionen:

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

Definition: Die *Ableitungsrelation* $\Rightarrow \subseteq (\chi^*)^2$ ist definiert durch

$$\alpha \Rightarrow \beta \quad :\Leftrightarrow \quad \begin{array}{l} \alpha = \alpha_1 A \alpha_2,\ A \to \gamma \in P \\ \beta = \alpha_1 \gamma \alpha_2 \end{array}$$

(Ein Nichtterminalsymbol in a wird ersetzt durch die rechte Seite)

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

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

Sprechweise: Ableitungsschritt; Links-, bzw. Rechtsableitungsschritt

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

Offensichtlich gilt $L(\mathcal{G}) = \{w \in \Sigma^* | S \overset{*}{\underset{l}{\Rightarrow}} w\}$ und $L(\mathcal{G}) = \{w \in \Sigma^* | S \overset{*}{\underset{r}{\Rightarrow}} w\}$

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

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

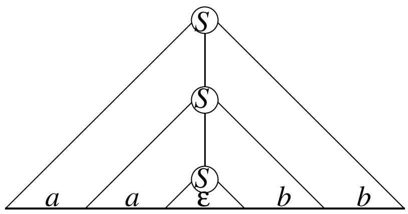


Abbildung 2.1: Der Ableitungsbaum, der die syntaktische Struktur von a^2b^2 bezüglich $\mathcal{G}$ repräsentiert

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

Folgerung: $\mathcal{G}$ ist eindeutig, wenn es für jedes $w \in L(\mathcal{G})$ genau eine Linksableitung (bzw. Rechtsableitung) gibt. In der Regel gibt es zu einem Ableitungsbaum mehrere Ableitungen, aber nur genau eine Links- bzw. Rechtsableitung.

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

2.1.1 *l*-Analyse, *r*-Analyse

Darstellung von l/r-Ableitungen durch Nummernfolgen

$$|P| = p, [p] := \{1, 2, \ldots, p\}, \Pi : [p] \xrightarrow{\text{bijektiv}} P$$

Sei $i \in [p]$:

$$wA\alpha \underset{l}{\overset{i}{\Rightarrow}} w\gamma\alpha \text{ bzw. } \alpha A w \underset{r}{\overset{i}{\Rightarrow}} \alpha\gamma w \text{ falls } \Pi_i := \Pi(i) = A \to \gamma$$

Für $z = i_1 \dots i_n \in [p]^+$ soll für passende $\alpha_1 \dots \alpha_{n-1}$ gelten:

$$\alpha \underset{l}{\overset{z}{\Rightarrow}} \beta : \curvearrowright \alpha \underset{l}{\overset{i_1}{\Rightarrow}} \alpha_1 \underset{l}{\overset{i_2}{\Rightarrow}} \alpha_2 \dots \underset{l}{\overset{i_n}{\Rightarrow}} \alpha_n = \beta$$

und $\alpha \underset{l}{\overset{\varepsilon}{\Rightarrow}} \alpha$ (ein „leerer“ l-Ableitungsschritt)

Definition: z heißt *l-Analyse* von $\alpha : \curvearrowright S \underset{l}{\overset{z}{\Rightarrow}} \alpha$.

Definition: z heißt *r-Analyse* von $\alpha : \curvearrowright S \underset{r}{\overset{z}{\Rightarrow}} \alpha$.

Bezeichnungskonvention: $i \in [p], z \in [p]^*$

Beispiel: siehe Abbildung 2.5

2.1.2 Syntaxanalyse

Gegeben ist eine Grammatik $\mathcal{G} \in \text{CFG}(\Sigma), w \in \Sigma^*$. Dafür: Berechnung einer l- bzw. r-Analyse, falls $w \in L(\mathcal{G})$, andernfalls Bestimmung der syntaktischen Fehler.

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

2.2 Top-Down Analyse mit $LL(k)$-Grammatiken

Definition: Sei $\mathcal{G} \in \text{CFG}(\Sigma)$. Der (nicht-deterministische) *TD-Analyseautomat* von $\mathcal{G} \in \text{CFG}$, (bezeichnet als $NTA(\mathcal{G})$) ist gekennzeichnet durch: [04.05.2004]

Eingabealphabet:	Σ	(Zustandsalphabet entfällt, „nur ein Zustand“)
Kelleralphabet:	χ	
Ausgabealphabet:	$[p] = \{1, \dots, p\}$	(falls p Regeln)
Konfigurationsmenge:	$\Sigma^* \times \chi^* \times [p]^*$	(Kellerspitze links)

sowie Transitionen wie folgt:

- Ableitungsschritte (nicht deterministisch): $(w, A\alpha, z) \vdash (w, \beta\alpha, zi)$ falls $\Pi_i = A \to \beta$
- Vergleichsschritte (deterministisch): $(aw, a\alpha, z) \vdash (w, \alpha, z)$ für $a \in \Sigma$

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

Beachte: Nichtdeterminismus wegen Mehrdeutigkeit ($A \to \beta|\gamma$)

Satz: Der $NTA(\mathcal{G})$ berechnet l-Analysen, dh. $(w, S, \varepsilon) \overset{*}{\vdash} (\varepsilon, \varepsilon, z) \curvearrowright z$ ist l-Analyse von w.

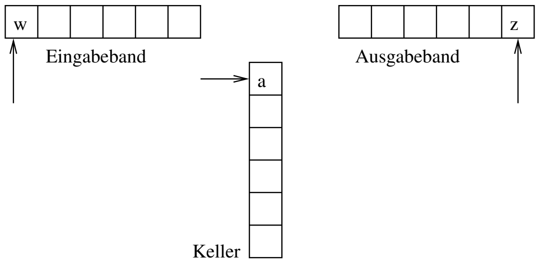


Abbildung 2.2: Analyseautomat

Beispiel: siehe Abbildung 2.2 und Ausführung in Abbildung 2.6.

Beweis: „$\curvearrowright$“ Automat arbeitet korrekt:

$$(*) \qquad (w,\alpha,y) \overset{*}{\vdash} (\varepsilon,\varepsilon,yz) \curvearrowright \alpha \underset{l}{\overset{z}{\Rightarrow}} w$$

Beweis von $(*)$ durch Induktion über $k := |z|$

Induktionsanfang $k = 0 : (w,\alpha,y) \overset{*}{\vdash} (\varepsilon,\varepsilon,y)$: nur Vergleichsschritte, also $w = \alpha$, es gilt $(*)$ mit $w \underset{l}{\overset{\varepsilon}{\Rightarrow}} w$.

Induktionsschritt $k \rightsquigarrow k+1 : z = iz', \alpha = uA\beta, w = uv, \Pi(i) = A \rightarrow \gamma$

$$(w,\alpha,y) = (uv,uA\beta,y) \overset{*}{\vdash} (v,A\beta,y) \vdash (v,\gamma\beta,yi) \overset{*}{\vdash} (\varepsilon,\varepsilon,yiz')$$

Nach Induktionsvoraussetzung: $\gamma\beta \underset{l}{\overset{z'}{\Rightarrow}} v$, und damit folgt $(*)$ wegen:

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

Beweis: „$\curvearrowleft$“ ähnlich.

$\square$

Ziel: Nichtdeterminismus des TD-Analyseautomaten von $\mathcal{G}$ durch *k-look-ahead* auf der Eingabe beseitigen, $k \in \mathbb{N}$.

Definition: (<u>first</u>-Mengen). Sei $\mathcal{G} \in \text{CFG}, \alpha \in \chi^*$ und $k \in \mathbb{N}$. Dann definieren wir $\underline{\text{first}}_k(\alpha) \subseteq \Sigma^*$ durch $\underline{\text{first}}_k(\alpha) := \{v \in \Sigma^* | \exists w \in \Sigma^* : \alpha \overset{*}{\Rightarrow} vw, |v| = k\} \cup \{v \in \Sigma^* | \alpha \overset{*}{\Rightarrow} v, |v| < k\}$

Folgerung:

i. $\underline{\text{first}}_k(\alpha) \neq \emptyset$, weil $\mathcal{G}$ reduziert.

ii. $\varepsilon \in \underline{\text{first}}_k(\alpha) \curvearrowleft\!\!\!\curvearrowright k = 0$ oder $\alpha \overset{*}{\Rightarrow} \varepsilon$

iii. $\alpha \overset{*}{\Rightarrow} \beta \curvearrowright \underline{\text{first}}_k(\beta) \subseteq \underline{\text{first}}_k(\alpha)$

iv. $v \in \underline{\text{first}}_k(\alpha) \curvearrowleft\!\!\!\curvearrowright \exists x \in \Sigma^* : \alpha \underset{l}{\overset{*}{\Rightarrow}} x, \{v\} = \underline{\text{first}}_k(x)$

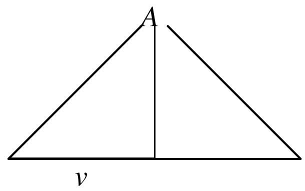


Abbildung 2.3: $|v| = k \curvearrowright v \in \underline{\text{first}}_k(A)$

$LL(k)$: Lesen der Eingabe von **l**inks nach rechts mit **k**-Lookahead. Berechnung einer **L**inksanalyse.

Definition: ($LL(k)$-*Grammatik*). Sei $\mathcal{G} \in$ CFG und $k \in \mathbb{N}$. $\mathcal{G} \in LL(k) :\curvearrowleft\!\curvearrowright$ Für alle Linksableitungen der Form

$$S \overset{*}{\underset{l}{\Rightarrow}} wA\alpha \begin{array}{l} \overset{l}{\nearrow} \quad w\beta\alpha \overset{*}{\underset{l}{\Rightarrow}} wx \\ \\ \underset{l}{\searrow} \quad w\gamma\alpha \overset{*}{\underset{l}{\Rightarrow}} wy \end{array}$$

gilt: $\underline{\text{first}}_k(x) = \underline{\text{first}}_k(y) \curvearrowright \beta = \gamma$.

Bemerkung:

- Linksableitungsschritt für $wA\alpha$ ist durch die nächsten k auf w folgenden Symbole bestimmt.
- Der TD-Analyseautomat für eine $LL(k)$-Grammatik kann deterministisch mit k-look-ahead auf Eingabe arbeiten.

[07.05.2004]

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

Lemma: $\mathcal{G} \in LL(k) \curvearrowleft\!\curvearrowright$ Für alle Linksableitungen der Form

$$S \overset{*}{\underset{l}{\Rightarrow}} wA\alpha \begin{array}{l} \overset{l}{\nearrow} \quad w\beta\alpha \\ \\ \underset{l}{\searrow} \quad w\gamma\alpha \end{array} \qquad \text{mit}\, \beta \neq \gamma$$

gilt $\underline{\text{first}}_k(\beta\alpha) \cap \underline{\text{first}}_k(\gamma\alpha) = \emptyset$.

Beweis: „$\downarrow$" (Definition $\rightarrow$ Lemma)
Angenommen, $\beta \neq \gamma$, aber $v \in \underline{\text{first}}_k(\beta\alpha) \cap \underline{\text{first}}_k(\gamma\alpha)$. Dann muß $\beta\alpha \overset{*}{\underset{l}{\Rightarrow}} x$ und $\gamma\alpha \overset{*}{\underset{l}{\Rightarrow}} y$ mit $\{v\} = \underline{\text{first}}_k(x) = \underline{\text{first}}_k(y)$.
Daraus folgt ein Widerspruch, weil $LL(k)$-Definition dann $\beta = \gamma$ erzwingt.

„$\uparrow$" (Lemma $\rightarrow$ Definition)
Angenommen, Lemma-Eigenschaft gilt, aber die Definitions-Eigenschaft ist mit $\beta \neq \gamma$ verletzt. Dann muß $\underline{\text{first}}_k(\beta\alpha) \cap \underline{\text{first}}_k(\gamma\alpha) = \emptyset$.
Dies widerspricht der Voraussetzung, weil $\underline{\text{first}}_k(x) \subseteq \underline{\text{first}}_k(\beta\alpha)$ und entsprechend auch $\underline{\text{first}}_k(y) \subseteq \underline{\text{first}}_k(\gamma\alpha)$.

$\square$

Folgerung: Bestimmung der A-Regel durch die look-ahead-Mengen $\underline{\text{first}}_k(\beta\alpha), \underline{\text{first}}_k(\gamma\alpha)$ für Regelpaar $A \to \beta|\gamma$.

Problem: Abhängigkeit der look-ahead-Menge vom Rechtskontext α.

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

Idee: Mögliche Rechtskontexte vereinigen.

Definition: $\underline{\text{follow}}$-Menge: Sei $\mathcal{G} \in \text{CFG}, A \in N$ und $k \in \mathbb{N}$. Dann definieren wir $\underline{\text{follow}}_k(A) \subseteq \Sigma^*$ durch $\underline{\text{follow}}_k(A) := \{v \in \Sigma^* | S \underset{l}{\overset{*}{\Rightarrow}} wA\alpha, v \in \underline{\text{first}}_k(\alpha)\}$.

Bemerkung: In $\underline{\text{follow}}_k(A)$ werden alle möglichen Rechtskontexte berücksichtigt.

2.2.1 Der Fall $k = 1$:

Im Allgemeinen ist $k = 1$ ausreichend. $k > 1$ findet in heutigen Parsergeneratoren durchaus Anwendung, ist aber deutlich aufwendiger.

Abkürzung $\underline{\text{fi}} := \underline{\text{first}}_1$ und $\underline{\text{fo}} := \underline{\text{follow}}_1$.

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

Definition: Look-Ahead-Menge: $\underline{\text{la}}(A \to \beta) := \underline{\text{fi}}(\beta\underline{\text{fo}}(A)) \subseteq \Sigma_\varepsilon := \Sigma \cup \{\varepsilon\}$ heißt *Look-ahead-Menge* von $A \to \beta$ (la-Menge).

Beachte:

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

Beweis: (des Satzes)

a. „$\curvearrowright$“ (Lemma $\to$ Satz): Angenommen, es gibt $A \to \beta|\gamma$ mit $\beta \neq \gamma$, aber:
   $c \in \underline{\text{fi}}(\beta\underline{\text{fo}}(A)) \cap \underline{\text{fi}}(\gamma\underline{\text{fo}}(A))$ für ein $c \in \Sigma_\varepsilon$.

Fall 1: $c = \varepsilon$

Es folgt: $\beta \overset{*}{\Rightarrow} \varepsilon, \gamma \overset{*}{\Rightarrow} \varepsilon$ und $\varepsilon \in \underline{\text{fo}}(A)$. Dies ergibt folgenden Widerspruch zum Lemma:

$$S \underset{l}{\overset{*}{\Rightarrow}} wA\alpha \begin{array}{l} \overset{l}{\nearrow}\ w\beta\alpha \\ \\ \underset{l}{\searrow}\ w\gamma\alpha \end{array} \qquad \text{mit}\, \beta \neq \gamma\, \mathit{und}\ \alpha \underset{l}{\overset{*}{\Rightarrow}} \varepsilon$$

aber: $\varepsilon \in \underline{\text{fi}}(\beta\alpha) \cap \underline{\text{fi}}(\gamma\alpha)$.

Fall 2: $c = a \in \Sigma$

Es folgt:

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

(2) $a \in \underline{\text{fi}}(\beta), \gamma \overset{*}{\Rightarrow} \varepsilon$ und $a \in \underline{\text{fo}}(A)$

(3) $a \in \underline{\text{fi}}(\gamma), \beta \overset{*}{\Rightarrow} \varepsilon$ und $a \in \underline{\text{fo}}(A)$

(4) $\beta \overset{*}{\Rightarrow}, \gamma \overset{*}{\Rightarrow} \varepsilon$ und $a \in \underline{\text{fo}}(A)$

In jedem Fall existiert eine Ableitung. $\mathcal{G}$ reduziert $\curvearrowright$ es ex. l-Ableitungen.

$$S \underset{l}{\overset{*}{\Rightarrow}} wA\alpha \begin{array}{l} \overset{l}{\nearrow}\ w\beta\alpha \\ \\ \underset{l}{\searrow}\ w\gamma\alpha \end{array} \qquad \text{aber} : a \in \underline{\text{fi}}(\beta\alpha) \cap \underline{\text{fi}}(\gamma\alpha)$$

also ein Widerspruch zu $LL(1)$, da $\beta \neq \gamma$

[11.05.2004]

b. „$\curvearrowleft$“ (Satz $\rightarrow$ Lemma):

$$\text{Sei}\, S \underset{l}{\overset{*}{\Rightarrow}} wA\alpha \begin{array}{l} \overset{l}{\nearrow}\ w\beta\alpha \\ \\ \underset{l}{\searrow}\ w\gamma\alpha \end{array} \qquad \text{mit}\, \beta \neq \gamma$$

Nach „Satz“ gilt: $\underline{\text{fi}}(\beta\underline{\text{fo}}(A)) \cap \underline{\text{fi}}(\gamma\underline{\text{fo}}(A)) = \emptyset$. Da $\underline{\text{fi}}(\beta\alpha) \subseteq \underline{\text{fi}}(\beta\underline{\text{fo}}(A))$, muß auch $\underline{\text{fi}}(\beta\alpha) \cap \underline{\text{fi}}(\gamma\alpha) = \emptyset$.

$\square$

Bemerkung: Für $k = 1$ kann der lokale Rechtskontext α durch $\underline{\text{fo}}(A)$ verallgemeinert werden. Die $\underline{\text{la}}$-Mengen $\underline{\text{la}}(A \rightarrow \beta)$ bestimmen einen deterministischen TD-Analyseautomaten.

2.2.2 Berechnung der la-Mengen

1. $\underline{\text{fi}}(X)$ für $X \in \chi$

   - $X = a \in \Sigma \curvearrowright \underline{\text{fi}}(X) = \{X\}$
   - $X \rightarrow a\alpha \curvearrowright a \in \underline{\text{fi}}(X)$
   - $X \rightarrow \varepsilon \curvearrowright \varepsilon \in \underline{\text{fi}}(X)$
   - $X \rightarrow A_1 \ldots A_k Y\alpha, k \geq 0, Y \in \chi, \varepsilon \in \underline{\text{fi}}(A_1) \cap \ldots \cap \underline{\text{fi}}(A_k), a \in \underline{\text{fi}}(Y) \curvearrowright a \in \underline{\text{fi}}(X)$

- $X \to A_1 \dots A_k, k \geq 1, \varepsilon \in \underline{\text{fi}}(A_1) \cap \dots \cap \underline{\text{fi}}(A_k) \curvearrowright \varepsilon \in \underline{\text{fi}}(X)$

2. $\underline{\text{fi}}(X_1 \dots X_n)$ für $X_i \in \chi, n \in \mathbb{N}$

   - $\varepsilon \in \underline{\text{fi}}(X_1) \cap \underline{\text{fi}}(X_2) \cap \dots \cap \underline{\text{fi}}(X_{i-1}), a \in \underline{\text{fi}}(X_i) \curvearrowright a \in \underline{\text{fi}}(X_1 \dots X_n)$
   - $\varepsilon \in \underline{\text{fi}}(X_1) \cap \underline{\text{fi}}(X_2) \cap \dots \cap \underline{\text{fi}}(X_n) \curvearrowright \varepsilon \in \underline{\text{fi}}(X_1 \dots X_n)$
   - $\underline{\text{fi}}(\varepsilon) = \{\varepsilon\}$

3. $\underline{\text{fo}}(A)$

   - $\varepsilon \in \underline{\text{fo}}(S)$, falls $A = S$
   - $B \to \alpha A \beta, a \in \underline{\text{fi}}(\beta) \curvearrowright a \in \underline{\text{fo}}(A)$
   - $B \to \alpha A, x \in \underline{\text{fo}}(B) \curvearrowright x \in \underline{\text{fo}}(A)$
   - $B \to \alpha A \beta, \varepsilon \in \underline{\text{fi}}(\beta), x \in \underline{\text{fo}}(B) \curvearrowright x \in \underline{\text{fo}}(A)$

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

Das Ganze wollen wir nun noch einmal durch ein **Beispiel** veranschaulichen. Gegeben sei die folgende Grammatik $\mathcal{G}'_{AE}$

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

Bei dieser Grammatik ergeben sich dann folgende Mengen:

- $\underline{\text{fi}}-$ und $\underline{\text{fo}}$-Mengen:

	E	E'	T	T'	F
$\underline{\text{fi}}$	(a	+ ε	(a	$*$ ε	(a
$\underline{\text{fo}}$	ε)	ε)	+ ε)	+ ε)	$*$ + ε)

- $\underline{\text{la}}$-Mengen: $\underline{\text{la}}(A \to \beta) := \underline{\text{fi}}(\beta\underline{\text{fo}}(A))$

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

Der Schnitt der Alternativen muß leer sein! Die Alternativen sind hier die Regeln $2+3$, $5+6$ und $7+8$.

Folgerung: $\mathcal{G}'_{AE}$ ist $LL(1)$.

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

Beachte: (neue Prüfungsfrage): $\mathcal{G} \in$ CFG, $\mathcal{G}$ reduziert. Regel in $\mathcal{G} : A \to \alpha B$.
Frage: Zusammenhang zwischen $\underline{\text{fo}}(A)$ und $\underline{\text{fo}}(B)$.
Antwort: $\underline{\text{fo}}(A) \subseteq \underline{\text{fo}}(B)$, aber im allgemeinen nicht $\underline{\text{fo}}(B) \subseteq \underline{\text{fo}}(A)$

2.2.3 Der deterministische 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 dem Eingabeband.
Modifikation des Ableitungsschritts:

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

Folgerung: Deterministische Arbeitsweise.

Beachte: Das Eingabesymbol wird bei Ableitungsschritten nicht gelöscht.

[14.05.2004]

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

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

$\underline{\text{act}}$ ist definiert durch

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

Die Analysetabelle für unsere Grammatik $\mathcal{G}'_{AE}$ ist in Abbildung 2.7 gegeben.

2.3 Parserkonstruktion nach TD-Methode

Sei $\mathcal{G} \in$ CFG reduziert gegeben. Dann verläuft die Parserkonstruktion in folgenden Schritten:

- Berechnung der la-Mengen (fi- und fo-Mengen)
- Analysetabelle erstellen, Eindeutigkeit prüfen
- tabellengesteuerter Parser

Problem: mehrdeutige Analysetabelle ($\mathcal{G} \notin LL(1)$).

2.3.1 Transformationen nach $LL(1)$

2 Methoden um $\mathcal{G}$ in äquivalente $LL(1)$-Grammatik zu transformieren (nicht immer möglich):

1. Beseitigung von Linksrekursionen
2. Links-Faktorisieren

Verwendung in Parser-erzeugenden Systemen.

Vorsicht: Transformationen erhalten zwar die Äquivalenz, im allgemeinen aber nicht die syntaktische Struktur (Ableitungsbaum).

Beseitigung von Linksrekursionen

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

Folgerung: $\mathcal{G}$ linksrekursiv $\curvearrowright \forall k \in \mathbb{N} \quad \mathcal{G} \notin LL(k)$.

Grund: Wenn ein TD-Parser $A \stackrel{+}{\Rightarrow} A\alpha$ simuliert, so bleibt der Eingabekopf stehen. Gleicher look-ahead. Damit: Schleife.

Es ist also nicht möglich Ableitungen der Form $S \underset{l}{\stackrel{*}{\Rightarrow}} wA\beta \underset{l}{\stackrel{+}{\Rightarrow}} wA\alpha\beta \underset{l}{\stackrel{*}{\Rightarrow}} wv$ zu bilden.

Beispiel:

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

$\mathcal{G}_{AE}$ ist linksrekursiv, also: $\mathcal{G}_{AE} \notin LL(k)$.

Probe: $LL(1)$-Test

$$\underline{\text{fi}}(E) = \underline{\text{fi}}(T) = \underline{\text{fi}}(F) = \{(,a\} \curvearrowright \underline{\text{la}}(\Pi_i) = \{(,a\},\ i = 1\ldots 4$$

Das bestätigt, daß $\mathcal{G}_{AE}$ keine $LL(1)$-Grammatik ist.

Spezialfall: *direkte Linksrekursion* und ihre Beseitigung

$$A \rightarrow A\alpha|\beta(\beta \neq A\ldots, \alpha \neq \varepsilon)$$

wird ersetzt durch

$$A \rightarrow \beta A' \qquad \textit{und} \qquad A' \rightarrow \alpha A'|\varepsilon \qquad \textit{mit neuem } A'$$

Folgerung: $L(\mathcal{G})$ unverändert, jedoch neue syntaktische Struktur (kein Problem bei assoziativen Operatoren $+, *, \ldots$, Semantik invariant).

Beispiel: Beseitigung direkter Linksrekursionen in $\mathcal{G}_{AE}$ ergibt $\mathcal{G}'_{AE}$.

Allgemeiner Fall: *indirekte Linksrekursion*

$$\begin{array}{lcl} A & \rightarrow & A_1\alpha_1|\ldots \\ A_1 & \rightarrow & A_2\alpha_2|\ldots \\ \vdots & & \\ A_n & \rightarrow & A\beta|\ldots \end{array}$$

Beseitigung durch Transformation in GNF (Greibach Normalform); Regeln der Form

$$A \rightarrow aB_1\ldots B_n \qquad S \rightarrow \varepsilon \qquad B_i \neq S$$

Beachte: Beseitigung von Linksrekursion ergibt nicht notwendig eine LL(1)-Grammatik. Der Grund hierfür ist, daß jede Grammatik $\mathcal{G} \in$ CFG in GNF äquivalent transformierbar ist, aber nicht jede kontextfreie Sprache ist durch eine LL(1)-Grammatik erzeugbar.
Es gilt:

$$\begin{array}{rcl} \mathscr{L}(LL(k)) & \subsetneq & \mathscr{L}(LL(k+1)), k \in \mathbb{N} \\ & \subsetneq & \mathscr{L}(DPDA) \\ & \subsetneq & \mathscr{L}(PDA) = CFL \end{array}$$

Komplexität der LL(1)-Analyse

$\mathcal{G}$ ist LL(1)-Grammatik $\curvearrowright$ $\mathcal{G}$ ist nicht links-rekursiv.

$DTA(G)$ mit Eingabe von $w \in \Sigma^*$:

- $|w|$ Vergleichsschritte plus einem „accept"-Schritt.
- $wA\alpha \overset{i}{\underset{l}{\rightarrow}} wB\beta$ [1]
  $\exists c \in \mathbb{N}$
  „$\mathcal{G}$ nicht linksrekursiv $\curvearrowright (A \overset{i}{\underset{l}{\Rightarrow}} B\gamma \curvearrowright i < c = |N|)$"

  $\underset{Vorsicht\ so\ falsch}{\curvearrowright} \quad (S \overset{n}{\underset{l}{\Rightarrow}} w \curvearrowright n \leq c(|w|+1)$

  <u>Besser</u>: Satz[2]: $\mathcal{G}$ nicht linksrekursiv, dann
  $\exists c \in \mathbb{N}\ S \overset{n}{\underset{l}{\Rightarrow}} w \curvearrowright n \leq c(|w|+1)$, wobei $c > |N|$ möglich

  Alte Version mit Widerlegungsbeispiel
  maximal $|N|$ aufeinanderfolgende Ableitungsschritte (Begründung: würde ein Nichtterminalsymbol doppelt auftauchen, ohne das ein Vergleichsschritt durchgeführt wurde, so würde eine links-Rekursion vorliegen).

  Widerspruch: Gegeben $\mathcal{G} \in$ CFG. $\mathcal{G}$ ist nicht linksrekursiv. Dann existiert $c \in \mathbb{N}$, so daß $A \overset{i}{\Rightarrow} B\alpha \curvearrowright i \leq c$. Der vorgestellte Lösungsvorschlag $c = |N|$ ist falsch (Gegenbeispiel: $S \rightarrow aBBBBBBc, \quad B \rightarrow \varepsilon$). [18.05.2004]

[1] s. Aho\Ullman : „The Theory of Parsing, ... VOL. I Parsing" (S. 356)
[2] O. Mayer, Syntaxanalyse (S. 48-55)

$\Rightarrow$ TD-Analyse in Linearzeit

$\curvearrowright$ $\underline{\max}|N| \cdot (|w|+1)$ Transitionen.

maximale Kellerlänge: $\underline{\max}\{|\alpha||A \to \alpha \text{ in } P\} \cdot |N| \cdot (|w|+1)$.

Also: linearer Platz und Zeitbedarf.

Links-Faktorisieren

Beispiel:

`statement` $\to$ `if` `condition` `then` `statement` `else` `statement` `fi`
`statement` $\to$ `if` `condition` `then` `statement` `fi`

Keine Regelentscheidung mit beschränkten look-ahead möglich.
Idee: Verschieben der Entscheidung bis Alternativen erkennbar.

Links-Faktorisieren: $A \to \alpha\beta|\alpha\gamma$ ersetzen durch $A \to \alpha A'$ und $A' \to \beta|\gamma$.

Beispiel:

`statement` $\to$ `if` `condition` `then` `statement` `S'`
`S'` $\to$ `else` `statement` `fi` $|$ `fi`

2.4 Top-Down Analyse mit rekursiven Prozeduren

Idee: keine explizite Kellerbenutzung wie beim $DTA(\mathcal{G})$, sondern implizite Verwendung des Laufzeitkellers durch Einsatz rekursiver Prozeduren.
Vorteil: leichte Programmierung
Spezialfall: $\mathcal{G} \in LL(1)$ (Beispiel: $\mathcal{G}'_{AE}$).

1. Analyseverfahren durch rekursiven Abstieg (ohne la-Mengen)

   Methode („Recursive descent parser“): $A \in N \mapsto A()$ parameterlose Prozedur, zur Simulation eines Ableitungsschritts (Beispiel: Abbildung 2.10).

   Annahme: 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.

   Der Folgende Abschnitt war nicht Bestandteil der VL SS 2004

2. Zusätzliche Verwendung der la-Mengen
   Vorteil: bessere Kontrolle der Regelanwendungen, frühere Fehlererkennung.

   Weitere Möglichkeit: nicht-deterministische Programmierung (z.B. `PROLOG`: CFG als „Definite Clause Grammar“).

2.5 Bottom-Up Analyse mit $LR(k)$-Grammatiken

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 zu Reduktionsschritten.

Dies wird *Shift-Reduce-Verfahren* genannt.

Definition: Der (nicht-deterministische) Bottom-Up-Analyseautomat von $\mathcal{G} \in$ CFG (im folgenden als $NBA(\mathcal{G})$ bezeichnet):

Eingabealphabet:	Σ
Kelleralphabet:	χ
Ausgabealphabet:	$[p]$
Konfigurationsmenge:	$\underbrace{\chi^*}_{\text{Keller}} \times \underbrace{\Sigma^*}_{\text{Eingabe}} \times \underbrace{[p]^*}_{\text{Ausgabe}}$

Transitionen:

Shift-Schritt: $(\alpha, aw, z) \vdash (\alpha a, w, z)$ für $a \in \Sigma$

Reduce-Schritt: $(\beta\alpha, w, z) \vdash (\beta A, w, zi)$ für $\Pi_i = A \to \alpha$

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

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

[21.05.2004]

2.5.1 Nicht-Determinismus

1. Shift- oder Reduce-Schritt
2. Reduce-Schritt : linker Henkelrand (Henkel: rechte Regelseite auf Keller)
3. Reduce-Schritt: linke Regelseite
4. Analyseende

Ziel: Nicht-Determinismus durch k-look-ahead auf Eingabe beseitigen $\rightsquigarrow$ $LR(k)$-Grammatiken (R: Rechts-Analyse).

Generalvoraussetzung: $\mathcal{G}$ ist *startsepariert*, d.h. S nur in $S \to A$, mit $A \neq S$. Jedes $\mathcal{G} \in$ CFG läßt sich durch Hinzufügen von $S' \to S$ in eine äquivalente startseparierte Grammatik transformieren. Im folgenden: $\mathcal{G}$ mit Sonderregel $S' \to S$ mit der Nummer 0.

Folgerung: (S', ε, z) ist eine Endkonfiguration, die zwar Folgekonfigurationen haben kann (ε-Regeln), aber ohne Übergang in weitere Endkonfigurationen (Kellerlänge ≥ 2). Also ergibt die Startsepariertheit ein deterministisches Analyseende.

Beseitigung des restlichen Nicht-Determinismus durch:

Definition: ($LR(k)$-Grammatik): Sei $\mathcal{G} \in$ CFG, startsepariert mit $S' \to S$, $k \in \mathbb{N}$. Dann ist $\mathcal{G} \in LR(K)$: $\curvearrowright$ Für alle Rechtsableitungen der Form

$$S \begin{array}{l} \overset{*}{\underset{r}{\nearrow}} \; \alpha A w \underset{r}{\Rightarrow} \alpha\beta w \quad \text{mit}\, \underline{\mathrm{first}}_k(w) \\ \overset{*}{\underset{r}{\searrow}} \; \alpha' A' v' \underset{r}{\Rightarrow} \alpha\beta v \quad = \underline{\mathrm{first}}_k(v) \end{array}$$

gilt: $\alpha' = \alpha$, $A' = A$, $v' = v$.

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

2.5.2 $LR(0)$-Grammatiken

$k = 0 \curvearrowright$ Entscheidung ohne look-ahead, allein durch den Kellerinhalt $\alpha\beta$.

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 $S' \overset{*}{\underset{r}{\Rightarrow}} \alpha A w \underset{r}{\Rightarrow} \alpha\beta_1\beta_2 w$. Dann heißt $[A \to \beta_1 \cdot \beta_2]$ eine $LR(0)$-*Auskunft* für $\alpha\beta_1$.
Für $\gamma \in \chi^*$ bezeichnet $\underline{\mathrm{LR}(0)}(\gamma)$ die Menge aller $LR(0)$-Auskünfte für γ, die sogenannte $LR(0)$-*Menge* von γ ($LR(0)$-Information).

Folgerung:

i. $\underline{\mathrm{LR}(0)}(\gamma)$ ist endlich.

ii. $\underline{\mathrm{LR}(0)}(G) := \{\underline{\mathrm{LR}(0)}(\gamma) | \gamma \in \chi^*\}$ ist endlich.

iii. $[A \to \beta_1 \cdot] \in \underline{\mathrm{LR}(0)}(\gamma)$ signalisiert Reduktionsmöglichkeit. $(\alpha\beta_1, w, z) \vdash (\alpha A, w, zi)$ für $\Pi_i = A \to \beta_1$ und $\gamma = \alpha\beta_1$. Hier muß eine Eindeutigkeit vorliegen, ansonsten handelt es sich um einen reduce-reduce Konflikt.

iv. $[A \to \beta_1 \cdot \beta_2] \in \underline{\mathrm{LR}(0)}(\gamma)$ mit $\beta_2 \neq \varepsilon$ bedeutet Shift-Möglichkeit wegen unvollständigen Henkel.

v. $G \in LR(0) \curvearrowright$ Die $\underline{\mathrm{LR}(0)}$-Mengen von $\mathcal{G}$ 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. $\underline{\mathrm{LR}(0)}(\varepsilon)$ ist die kleinste Menge, welche
   (a) $[S' \to \cdot S]$ enthält
   (b) mit $[A \to \cdot B\delta]$ und $B \to \beta$ auch $[B \to \cdot\beta]$ enthält
2. $\underline{\mathrm{LR}(0)}(\alpha X)$ mit $X \in \chi$ ist die kleinste Menge, welche
   (a) $[A \to \beta_1 X \cdot \beta_2]$ enthält, falls $[A \to \beta_1 \cdot X\beta_2] \in \underline{\mathrm{LR}(0)}(\alpha)$
   (b) und mit $[A \to \gamma \cdot B\delta]$ und $B \to \beta$ in $\mathcal{G}$ auch $[B \to \cdot\beta]$ enthält

[25.05.2004]

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

Folgerung: $[A \to \beta_1 \cdot B\beta_2] \in \underline{\mathrm{LR}(0)}(\alpha\beta_1), B \to \gamma \curvearrowright [B \to \cdot\gamma] \in \underline{\mathrm{LR}(0)}(\alpha\beta_1)$.

Die goto-Funktion

$\mathcal{G} : LR(0)$-Grammatik $\curvearrowright \underline{\mathrm{LR}(0)}(\gamma)$ liefert Shift/Reduce-Entscheidung für den BU-Analyseautomaten mit Kellerinschrift γ.

- neues Kelleralphabet: $\underline{\mathrm{LR}(0)}(G)$ statt χ

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

$\underline{\mathrm{goto}} : \underline{\mathrm{LR}(0)}(G) \times \chi \to \underline{\mathrm{LR}(0)}(G)$ ist definiert durch

$$\underline{\mathrm{goto}}(I, X) := I' \curvearrowright \exists \gamma \in \chi^*; I = \underline{\mathrm{LR}(0)}(\gamma) \,\mathrm{und}\, I' = \underline{\mathrm{LR}(0)}(\gamma X)$$

Berechnung der LR(0)-Mengen und goto-Funktion durch Potenzmengenkonstruktion nichtdeterministischer endlicher Automaten

Sei $\mathcal{G} \in$ CFG, $\mathcal{G}$ startsepariert mit $S' \to S$. Konstruktion eines $\mathfrak{A}(G) \in \mathrm{NFA}_\varepsilon$

Zustandsmenge	$Q := \{[A \to \beta_1 \cdot \beta_2] \mid A \to \beta_1\beta_2 \text{ in } \mathcal{G}\}$
Eingabealphabet	$\chi := N \cup \Sigma$
Anfangszustand	$q_0 := [S' \to \cdot S]$
(Endzustandsmenge	$F := Q$ ohne Bedeutung)
Transitionsfunktion	$\delta : Q \times \chi_\varepsilon \to \mathfrak{p}(Q)$
	$\delta([A \to \beta_1 \cdot X\beta_2], X) \ni [A \to \beta_1 X \cdot \beta_2]$
	$\delta([A \to \beta_1 \cdot B\beta_2], \varepsilon) \ni [B \to \cdot\gamma]$, falls $B \to \gamma$ in $\mathcal{G}$

Potenzmengenkonstruktion nach Thompson: Konstruktion von $\widehat{\mathfrak{A}(G)} :< \hat{Q}, \chi, \hat{\delta}, \hat{q_0}, \emptyset > \in$ DFA. Erweitere Transitionsfunktion:

$\overline{\delta}$: $\mathfrak{p}(Q) \times \chi^* \to \mathfrak{p}(Q)$
$\overline{\delta}(T, \varepsilon) := \varepsilon(T)$ („ε-Hülle von T“)
$\overline{\delta}(T, wa) := \varepsilon(\bigcup_{q \in \overline{\delta}(T,w)} \delta(q, a))$
$\hat{Q} := \{\overline{\delta}(\{[S' \to \cdot S]\}, \alpha) \mid \alpha \in \chi^*\}$
$\hat{q_0} := \varepsilon(\{[S' \to \cdot S]\})$
$\hat{\delta}(T, X) := \overline{\delta}(T, X)$
Dann gilt: $\hat{Q} = \underline{\mathrm{LR}(0)}(G)$ $\qquad \hat{\delta} = \underline{\mathrm{goto}}$

Konstruktion des deterministischen BU-Analyseautomaten für $\mathcal{G} \in LR(0)$

Hilfsmittel: $\underline{\mathrm{LR}(0)}(G)$ und $\underline{\mathrm{goto}}$-Funktion.
Die $\underline{\mathrm{action}}$-Funktion von $\mathcal{G}$ gibt die Shift/Reduce-Entscheidung an:

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

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

[28.05.2004]

Eindeutigkeit bei $\mathcal{G} \in LR(0)$.
Die Funktionen $\underline{\mathrm{act}}$ und $\underline{\mathrm{goto}}$ bilden die $\underline{\mathrm{LR}(0)}$-Analysetabelle von $\mathcal{G}$ (Beispiel: Abbildung 2.18).
Diese Tabelle bestimmt den $\underline{\mathrm{LR}(0)}$-Analyseautomaten.

Eingabealphabet:	Σ
Kelleralphabet:	$\Gamma := \underline{\mathrm{LR}(0)}(G)$
Ausgabealphabet:	$\Delta := [p] \cup \{\underline{\mathrm{error}}\} \cup \{0\}$
Konfigurationsmenge:	$\Gamma^* \times \Sigma^* \times \Delta^*$
Bezeichnungskonvention:	$wa_1 \ldots a_n - (n) := w$

Transistionen:

- Shift-Schritt: $(\alpha I, aw, z) \vdash (\alpha I I', w, z)$, falls $\underline{\mathrm{act}}(I) = \underline{\mathrm{shift}}$ und $\underline{\mathrm{goto}}(I, a) = I'$.
- Reduce-Schritt: $(\alpha I, w, z) \vdash (\tilde{\alpha}\tilde{I}I', w, zi)$, falls $\underline{\mathrm{act}}(I) = \underline{\mathrm{red}}_i$, $\Pi_i = A \to X_1 \ldots X_n, \alpha I - (n) = \tilde{\alpha}\tilde{I}$ und $\underline{\mathrm{goto}}(\tilde{I}, A) = I'$.
- Accept-Schritt: $(I_0 I, \varepsilon, z) \vdash (\varepsilon, \varepsilon, z0)$, falls $\underline{\mathrm{act}}(I) = \underline{\mathrm{accept}}$.
- Fehlererkennung: $(\alpha I, w, z) \vdash (\varepsilon, \varepsilon, z \cdot \underline{\mathrm{error}})$, sonst.

Anfangskonfiguration für $w \in \Sigma^*$: (I_0, w, ε) mit $I_0 = \underline{\mathrm{LR}(0)}(\varepsilon)$.

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

- $(I_0, w, \varepsilon) \overset{*}{\vdash} (\varepsilon, \varepsilon, z) \curvearrowright \overleftarrow{z}$ r-Analyse von w
- $(I_0, w, \varepsilon) \overset{*}{\vdash} (\varepsilon, \varepsilon, z \cdot \underline{\mathrm{error}}) \curvearrowright w \notin L(G)$

Beispiel: $w = aac$ (Grammatik $\mathcal{G}$ aus Abbildung 2.17):

(I_0, aac, ε)
$(I_0 I_4, ac, \varepsilon)$
$(I_0 I_4 I_4, c, \varepsilon)$
$(I_0 I_4 I_4 I_6, \varepsilon, \varepsilon)$
$(I_0 I_4 I_4 I_8, \varepsilon, 6)$
$(I_0 I_4 I_8, \varepsilon, 65)$
$(I_0 I_3, \varepsilon, 655)$
$(I_0 I_1, \varepsilon, 6552)$
$(\varepsilon, \varepsilon, \underleftarrow{65520})$

Es ergibt sich somit: $\overleftarrow{z} = 02556$, also die folgende r-Analyse:

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

2.5.3 $SLR(1)$-Analyse

Die $SLR(1)$-Analyse ist ein vereinfachte (Simple) $LR(1)$-Analyse.

In der Praxis treten widersprüchliche $LR(0)$-Mengen auf: $\mathcal{G} \notin LR(0)$.

Beispiel: Shift/Reduce-Konflikte in $\mathcal{G}_{AE}$ (siehe Abbildung 2.21: I_1, I_2, I_9).

Beseitigung der Konflikte durch nächstes Eingabesymbol. Beobachtung:

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

Für obiges Beispiel:

I_1: $\underline{\text{shift}}$ bei Eingabe von $+$
  $\underline{\text{accept}}$ bei Eingabe von \$
I_2: $\underline{\text{shift}}$ bei Eingabe von $*$
  $\underline{\text{red}_2}$ bei Eingabe von $+,), \$$
I_9: $\underline{\text{shift}}$ bei Eingabe von $*$
  $\underline{\text{red}_1}$ bei Eingabe von $+,), \$$

So sind die Konflikte beseitigt. Weiterer Vorteil: frühere Fehlererkennung (genauere Kontrolle der Aktionen).

Die $SLR(1)$-action Funktion

$\underline{\text{act}} : \underline{\text{LR}(0)}(G) \times (\Sigma \cup \{\$\}) \to \{\underline{\text{shift}}, \underline{\text{red}_i}, \underline{\text{accept}}, \underline{\text{error}} | i \in [p]\}$ sei definiert durch:

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

Definition: $\mathcal{G}$ ist eine $SLR(1)$-Grammatik $\curvearrowright \underline{\text{act}}(I,a)$ stets eindeutig.

Die action und goto Funktionen bilden die $SLR(1)$-Analysetabelle von $\mathcal{G}$ (siehe Abbildung 2.22).
Bemerkung: Durch Tabellenkompression ist hier eine Platzersparnis möglich (füge goto-Anweisungen für Eingabesymbole in die Actiontabelle mit ein $\Rightarrow$ z.B. $\underline{\text{shift}} \cdot I_1$).
Jedoch sind immer noch Konflikte möglich. Eine bessere Konfliktbeseitigung ergibt sich durch Verwendung des Look-ahead Symbols in der Auskunft:

$$[A \to \beta_1 \cdot \beta_2, a] \to LR(1), LALR(1)$$

2.5.4 $LR(1)$-Analyse

[08.06.2004] Nicht immer sind Konflikte über follow-Mengen lösbar (siehe Abbildung 2.23).
Aber nicht jedes Element von $\underline{\text{fo}}(R)$ in beliebiger Rechtsableitung hinter R möglich $\curvearrowright$ Verfeinerung der $LR(0)$-Auskünfte durch Mitführen der möglichen look-ahead Symbole.

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

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

$$\underline{\text{LR}(1)}(G) := \{\underline{\text{LR}(1)}(\gamma) | \gamma \in \chi^*\}$$

Beobachtung: $\underline{\text{LR}(1)}(\gamma)$ „enthält in der 1. Kompontente“ gerade $\underline{\text{LR}(0)}(\gamma)$.

Berechnung der $LR(1)$-Mengen

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

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

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

$\underline{\text{act}} : \underline{\text{LR}(1)}(G) \times (\Sigma \cup \{\$\}) \to \{\underline{\text{shift}}, \underline{\text{red}}_i, \underline{\text{accept}}, \underline{\text{error}} | i \in [p]\}$ ist definiert durch:

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

Dann gilt: $\mathcal{G} \in LR(1) \curvearrowright \underline{\text{act}}(I,x)$ eindeutig (konfliktfrei).

2.5.5 $LALR(1)$-Analyse

$LALR(1)$ steht hier für $LR(1)$-Analyse mit look-ahead.

Beseitigung von Entscheidungskonflikten nach $LR(1)$-Methode zu aufwendig (die Tabellen werden zu groß).

Im Beispiel	$\lvert\underline{\mathrm{LR}(0)}(G_2)\rvert = 11$	$\lvert\underline{\mathrm{LR}(1)}(G_2)\rvert = 15$
ALGOL 60	$\sim 10^2$	$\sim 10^3$

Beobachtung: mögliche Informationsredundanz bei $\underline{\mathrm{LR}(1)}(G_2)$. Jede $\underline{\mathrm{LR}(1)}$-Menge „enthält" eine $\underline{\mathrm{LR}(0)}$-Menge.

Definition: $I_1, I_2 \in \underline{\mathrm{LR}(1)}(G)$ heißen $\underline{\mathrm{LR}(0)}$-äquivalent, $I_1 \underset{0}{\sim} I_2$, falls die $LR(0)$-Anteile von I_1 und I_2 gleich sind.

Beispiel: $I_4^{(1)} \underset{0}{\sim} I_{11}^{(1)}, I_5^{(1)} \underset{0}{\sim} I_{12}^{(1)}, I_7^{(1)} \underset{0}{\sim} I_{13}^{(1)}, I_8^{(1)} \underset{0}{\sim} I_{10}^{(1)}$.

[11.06.2004]

Folgerung: $\lvert\underline{\mathrm{LR}(1)}(G)/\underset{0}{\sim}\rvert = \lvert\underline{\mathrm{LR}(0)}(G)\rvert$

Oft können $LR(0)$-äquivalente $LR(1)$-Informationen *vereinigt* werden, ohne daß die Konfliktlösbarkeit verloren geht.
Insbesondere: Es können nur Reduce/Reduce-Konflike durch Vereinigung entstehen! Evtl. wird eine Fehlererkennung verzögert.

Definition:

- $I \in \underline{\mathrm{LR}(1)}(G)$ bestimmt eine $\underline{\mathrm{LALR}(1)}$-Menge.

$$\bigcup\{I' \in \underline{\mathrm{LR}(1)}(G) \mid I' \underset{0}{\sim} I\}$$

- $\underline{\mathrm{LALR}(1)}(G)$: Menge der $LALR(1)$-Mengen von $\mathcal{G}$.

Es gilt: $\lvert\underline{\mathrm{LALR}(1)}(G)\rvert = \lvert\underline{\mathrm{LR}(0)}(G)\rvert$; aber: im Unterschied zu den $\underline{\mathrm{LR}(0)}$-Informationen enthalten die $\underline{\mathrm{LALR}(1)}$-Mengen look-ahead-Symbole zur Lösung von Konflikten.

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

Die $LALR(1)$-action Funktion von $\mathcal{G}$ ist analog zu der von $LR(1)$ definiert.
Definition: $\mathcal{G} \in LALR(1) :\curvearrowleft LALR(1)$-action Funktion ist eindeutig.

Beispiel: Die $LALR(1)$-Menge von $\mathcal{G}_2$

$$\underline{\mathrm{LALR}(1)}(G_2) := \{I_i^{(1/0)} \mid 1 \leq i \leq 9\}$$

$$I_i^{(1/0)} := I_i^{(1)} \text{ für } i = 0,1,2,3,6,9$$
$$I_4^{(1/0)} := I_4^{(1)} \cup I_{11}^{(1)}$$
$$I_5^{(1/0)} := I_5^{(1)} \cup I_{12}^{(1)}$$

$$I_7^{(1/0)} := I_7^{(1)} \cup I_{13}^{(1)}$$

$$I_8^{(1/0)} := I_8^{(1)} \cup I_{10}^{(1)}$$

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

2.6 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 aber ein natürliches Beschreibungsmittel bei Programmiersprachen, um aufwendige Klammerung zu vermeiden.
Auflösung der Mehrdeutigkeit durch Regeln für Präzendenz und Assoziativität von Op-Symbolen (allgemeiner: von syntaktischen Konstrukten).

Beispiel: (Abbildung 2.30) $\mathcal{G}_{AE}^m : E \to E+E|E*E|(E)|a$:

Präzendenz:	$*$ vor $+$	in $a+a*a$
Assoziativität:	links	in $a+a+a$

Die Konflikte in I_1 sind $SLR(1)$-lösbar, jedoch sind aufgrund der Mehrdeutigkeit die Konflikte in I_7 und I_8 nicht auflösbar.

Beispiel:

I_0	$a+a*a$	
I_0I_3	$+a*a$	
I_0I_1	$+a*a$	
$I_0I_1I_4$	$a*a$	
$I_0I_1I_4I_3$	$*a$	
$I_0I_1I_4I_7$	$*a$	Reduktionskonflikt, $*>+$
$\underline{\mathrm{act}}(I_7,*)$	$=\underline{\mathrm{shift}}$	
$\underline{\mathrm{act}}(I_7,+)$	$=\underline{\mathrm{red}}_1$	(falls $+>*$)

Beispiel: Mehrdeutigkeit bei Verzweigungen („dangling else")

$$S \to iSeS|iS|a \qquad (1,2,3)$$

siehe auch Abbildung 2.32 (i: if, e: else). Im $LR(0)$-Graph kommt es bei I_4 zum Konflikt: $e \in \underline{\mathrm{fo}}(S) \curvearrowright$ nicht LR-lösbar.

$$\underline{\mathrm{if}}\, b\, \underline{\mathrm{then}}\, \underline{\mathrm{if}}\, b\, \underline{\mathrm{then}}\, a\, \underline{\mathrm{else}}\, a$$

Es existieren zwei Zerlegungen:

1. $\underline{\mathrm{if}}\, b\, \underline{\mathrm{then}}\, (\,\underline{\mathrm{if}}\, b\, \underline{\mathrm{then}}\, a\, \underline{\mathrm{else}}\, a)$ (dies ist die Konvention)

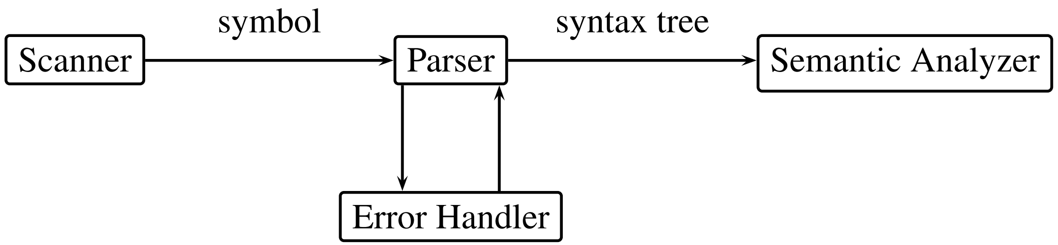
2. $\underline{\mathrm{if}}\, b\, \underline{\mathrm{then}}\, (\,\underline{\mathrm{if}}\, b\, \underline{\mathrm{then}}\, a)\, \underline{\mathrm{else}}\, a$ (so nicht in der Praxis)

2.7 Ergebnis

Die Syntaxanalyse setzt eine lineare Struktur in eine Baumstruktur um. Dies ist wichtig für syntaxgerichtete Softwarewerkzeuge (Compiler, Editoren, Textformatierer, Befehlsinterpreter).

Weitere Informationen in K. John Gaugh: Syntax Analysis and Software Tools (Addison-Wesley, 1988).

Schnittstellen



CB 2-1

Abbildung 2.4: Schnittstellen

Links– und Rechtsableitungen

Bsp: G_{AE} : $E \rightarrow E+T \mid T \quad (1,2)$

$T \rightarrow T*F \mid F \quad (3,4)$

$F \rightarrow (E) \mid a \mid b \mid c \quad (5-8)$

Linksableitung von $(a)*c$:

$$E \overset{2}{\underset{l}{\Rightarrow}} T \overset{3}{\underset{l}{\Rightarrow}} T*F \overset{4}{\underset{l}{\Rightarrow}} F*F \overset{5}{\underset{l}{\Rightarrow}} (E)*F \overset{2}{\underset{l}{\Rightarrow}}$$

$$(T)*F \overset{4}{\underset{l}{\Rightarrow}} (F)*F \overset{6}{\underset{l}{\Rightarrow}} (a)*F \overset{8}{\underset{l}{\Rightarrow}} (a)*c$$

$\rightsquigarrow$ Linksanalyse von $(a)*c$: 23452468

Rechtsableitung von $(a)*c$:

$$E \overset{2}{\underset{r}{\Rightarrow}} T \overset{3}{\underset{r}{\Rightarrow}} T*F \overset{8}{\underset{r}{\Rightarrow}} T*c \overset{4}{\underset{r}{\Rightarrow}} F*c \overset{5}{\underset{r}{\Rightarrow}}$$

$$(E)*c \overset{2}{\underset{l}{\Rightarrow}} (T)*c \overset{4}{\underset{l}{\Rightarrow}} (F)*c \overset{6}{\underset{l}{\Rightarrow}} (a)*c$$

$\rightsquigarrow$ Rechtsanalyse von $(a)*c$: 23845246

CB 2-2

Abbildung 2.5: Links- und Rechtsableitungen

COMPILERBAU 2004

Beispiel:

$NTA(G_{AE})$

$$
\begin{array}{llll}
G_{AE}: & E \to E+T \mid T & & (1,2) \\
 & T \to T*F \mid F & & (3,4) \\
 & F \to (E) \mid a \mid b \mid c & & (5-8)
\end{array}
$$

Linksanalyse von $(a)*c$:

$$
\begin{array}{llll}
(& (a)*c, & E & , \varepsilon &) \\
(& (a)*c, & T & , 2 &) \\
(& (a)*c, & T*F & , 23 &) \\
(& (a)*c, & F*F & , 234 &) \\
(& (a)*c, & (E)*F & , 2345 &) \\
(& a)*c, & E)*F & , 2345 &) \\
(& a)*c, & T)*F & , 23452 &) \\
(& a)*c, & F)*F & , 234524 &) \\
(& a)*c, & a)*F & , 2345246 &) \\
(&)*c, &)*F & , 2345246 &) \\
(& *c, & *F & , 2345246 &) \\
(& c, & F & , 2345246 &) \\
(& c, & c & , 23452468 &) \\
(& \varepsilon, & \varepsilon & , 23452468 &)
\end{array}
$$

CB 2-3

Abbildung 2.6: Beispiel NTA

Analysetabelle für G'_{AE}

	E	E'	T	T'	F	`a`	`(`	`)`	`+`	`*`	ε
`a`	TE', 1		FT', 4		`a`, 8	<u>pop</u>					
`(`	TE', 1		FT', 4		`(`E`)`, 7		<u>pop</u>				
`)`		ε, 3		ε, 6				<u>pop</u>			
`+`		+TE',2		ε, 6					<u>pop</u>		
`*`				$*FT'$, 5						<u>pop</u>	
ε		ε, 3		ε, 6							<u>accept</u>

CB 2-4

Abbildung 2.7: Analysetabelle für G'_{AE}

Zustandsfolge des DTA für G'_{AE}

⟨ `(a*a)`	E	ε ⟩
⟨ `(a*a)`	TE'	1 ⟩
⟨ `(a*a)`	$FT'E'$	14 ⟩
⟨ `(a*a)`	$(E)T'E'$	147 ⟩
⟨ `a*a)`	$E)T'E'$	147 ⟩
⟨ `a*a)`	$TE')T'E'$	1471 ⟩
⟨ `a*a)`	$FT'E')T'E'$	14714 ⟩
⟨ `a*a)`	$\mathtt{a}T'E')T'E'$	147148 ⟩
⟨ `*a)`	$T'E')T'E'$	147148 ⟩
⟨ `*a)`	$*FT'T'E')T'E'$	1471485 ⟩
⟨ `a)`	$FT'T'E')T'E'$	1471485 ⟩
⟨ `a)`	$\mathtt{a}T'T'E')T'E'$	14714858 ⟩
⟨ `)`	$T'T'E')T'E'$	14714858 ⟩
⟨ `)`	$T'E')T'E'$	147148586 ⟩
⟨ `)`	$E')T'E'$	1471485866 ⟩
⟨ `)`	$)T'E'$	14714858663 ⟩
⟨	$T'E'$	14714858663 ⟩
⟨	E'	147148586636 ⟩
⟨	ε	1471485866363 ⟩

Abbildung 2.8: Arbeitsweise von $\mathcal{G}'_{AE}$

G'_{AE}

	Produktionen	la–Mengen
(1)	$E \longrightarrow TE'$	(a
(2)	$E' \longrightarrow +TE'$	$+$
(3)	$\longrightarrow \varepsilon$	ε)
(4)	$T \longrightarrow FT'$	(a
(5)	$T' \longrightarrow *FT'$	$*$
(6)	$\longrightarrow \varepsilon$	$+$ ε)
(7)	$F \longrightarrow (E)$	(
(8)	$\longrightarrow a$	a

Abbildung 2.9: Produktionen

RD Parser für G'_{AE}

```
proc T()
  if sym in {'(','a'}
  then print(4);
       F(); T'();
  else print(ERROR);stop;
```

```
proc E()
  if sym in {'(','a'}
  then print(1);
       T(); E'();
  else print(ERROR);stop;
```

```
proc F()
  if sym='(' then
     print(7);
     nextsym; E();
     if sym=')' then nextsym;
                else print(ERROR); stop;
  elseif sym='a' then
     print(8);
     nextsym;
  else
     print(ERROR); stop
```

```
proc E'()
  if sym='+' then
     print(2);
     nextsym;
     T(); E'();
  elseif sym in {')',ε}
  then  print(3)
  else print(ERROR); stop;
```

```
proc T'()
  if sym='*' then
     print(5);
     nextsym;
     F(); T'();
  elseif sym in {'+',ε}
  then print(6);
  else print(ERROR); stop;
```

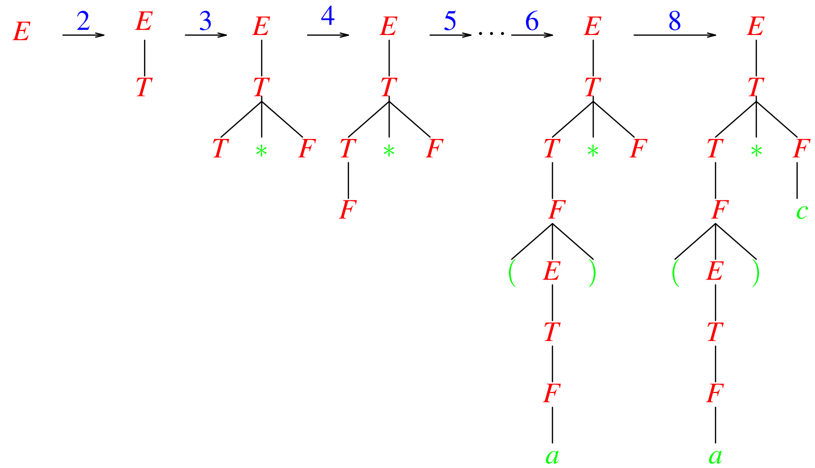
CB 2-6

Abbildung 2.10: Recursive descent parser für $\mathcal{G}'_{AE}$

Linksanalyse:

Top–down–Konstruktion des Ableitungsbaums

z.B. Linksanalyse 2 3 4 5 2 4 6 8 von $(a) * c$:



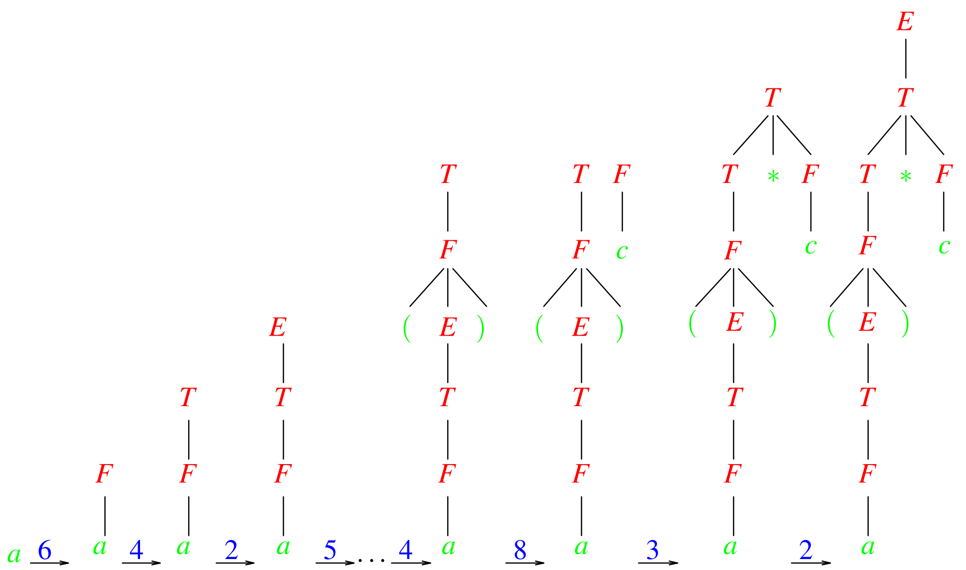
CB 2-7

Abbildung 2.11: Linksanalyse / TD-Konstruktion

Rechtsanalyse:
Bottom–up–Konstruktion des Ableitungsbaums

z.B. gespiegelte Rechtsanalyse

$6\ 4\ 2\ 5\ 4\ 8\ 3\ 2$ von $(a) * c$:



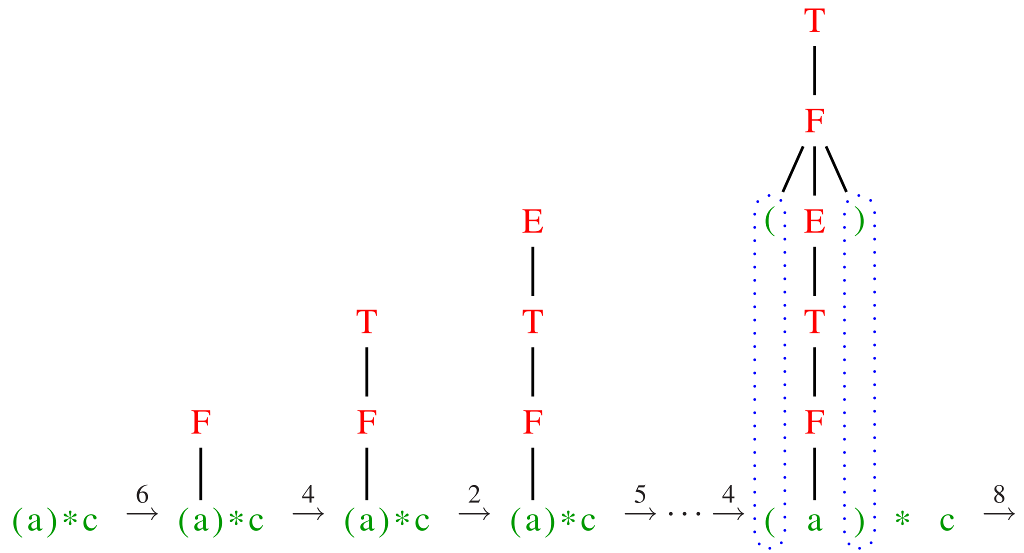
CB 2-8

Abbildung 2.12: Rechtsanalyse / BU-Konstruktion

Rechtsanalyse: Bottom–up–Konstruktion des Ableitungsbaums

z.B. gespiegelte Rechtsanalyse

6 4 2 5 4 8 3 2 von $(a) * c$:



CB 2-8a

Abbildung 2.13: Rechtsanalyse I

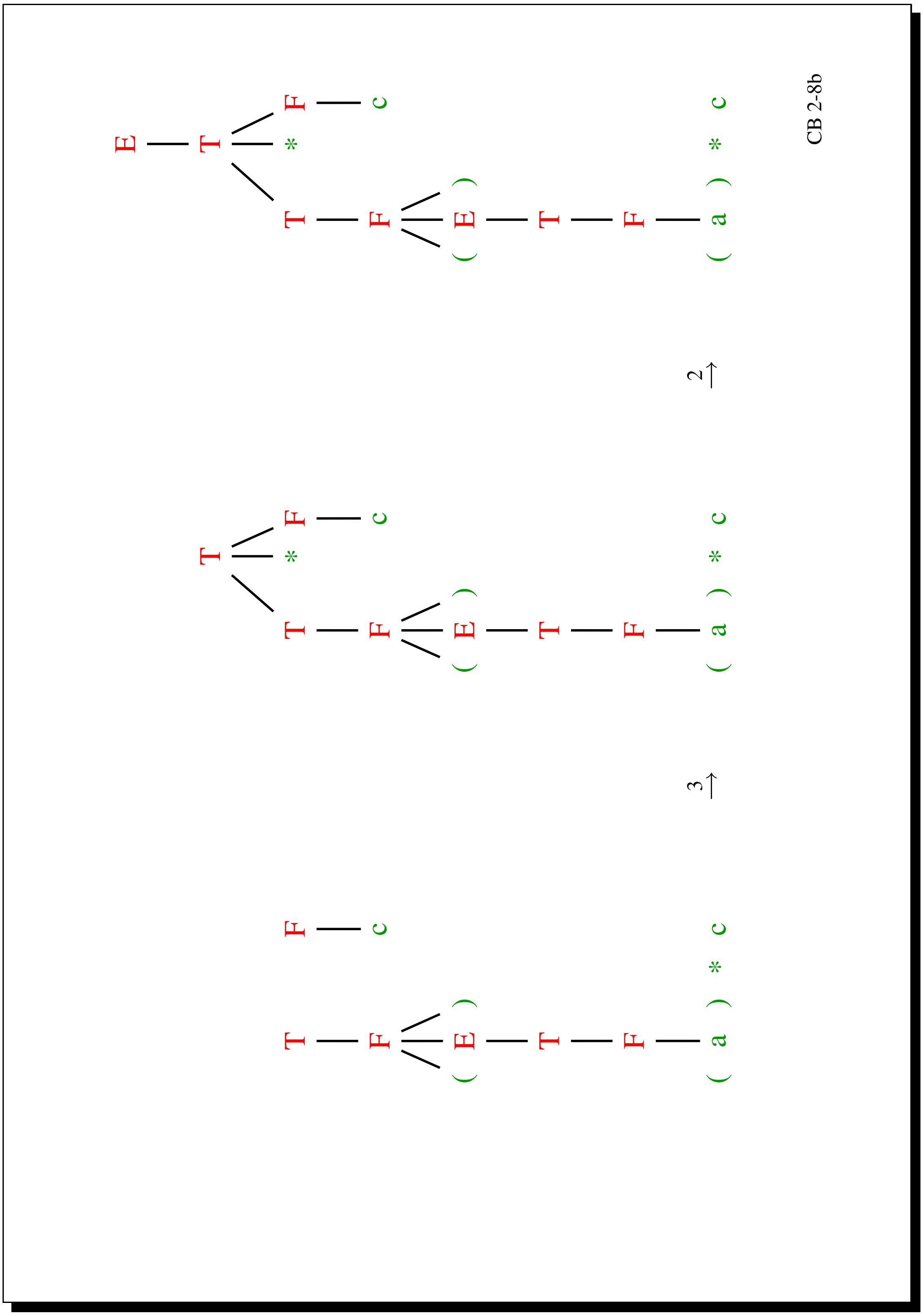


Abbildung 2.14: Rechtsanalyse II

Bottom–up–Analyseautomat von G_{AE}

$$
\begin{array}{llcll}
G_{AE}: & E & \to & E\texttt{+}T \mid T & (1,2) \\
 & T & \to & T\texttt{*}F \mid F & (3,4) \\
 & F & \to & \texttt{(}E\texttt{)} \mid \texttt{a} \mid \texttt{b} \mid \texttt{c} & (5-8)
\end{array}
$$

Analyse von `(a)*c`:

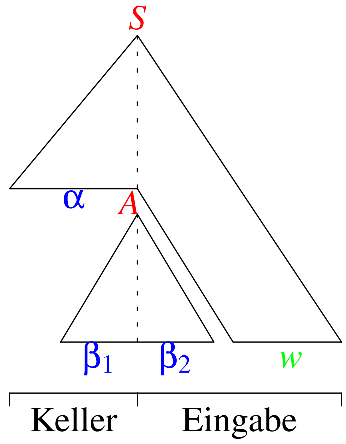
$$
\begin{array}{ll}
(\ \varepsilon,\ \texttt{(a)*c},\ \varepsilon\ ) & (\ T,\ \texttt{*c},\ 64254\ ) \\
(\ \texttt{(},\ \texttt{a)*c},\ \varepsilon\ ) & (\ T\texttt{*},\ \texttt{c},\ 64254\ ) \\
(\ \texttt{(a},\ \texttt{)*c},\ \varepsilon\ ) & (\ T\texttt{*c},\ \varepsilon,\ 64254\ ) \\
(\ \texttt{(}F,\ \texttt{)*c},\ 6\ ) & (\ T\texttt{*}F,\ \varepsilon,\ 642548\ ) \\
(\ \texttt{(}T,\ \texttt{)*c},\ 64\ ) & (\ T,\ \varepsilon,\ 6425483\ ) \\
(\ \texttt{(}E,\ \texttt{)*c},\ 642\ ) & (\ E,\ \varepsilon,\ 64254832\ ) \\
(\ \texttt{(}E\texttt{)},\ \texttt{*c},\ 642\ ) & \\
(\ F,\ \texttt{*c},\ 6425\ ) &
\end{array}
$$

CB 2-9

Abbildung 2.15: Bottom-Up-Analyseautomat von G_{AE}

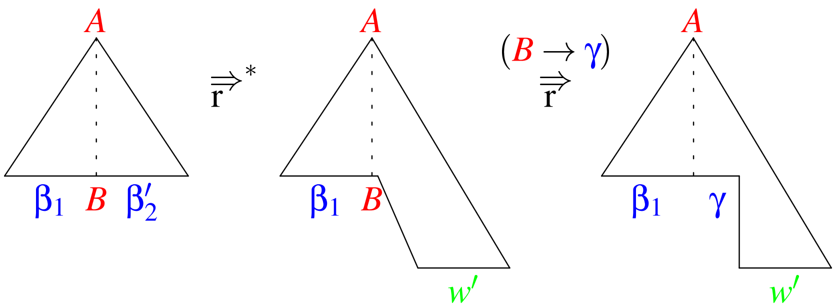
LR(0)–Analyse und Ableitungsbäume

Allgemeine Situation:



$$\Rightarrow [A \to \beta_1.\beta_2] \in \mathbf{LR(0)}(\alpha\beta_1)$$

ε–Abschluß:



$$\Rightarrow [B \to .\gamma] \in \mathbf{LR(0)}(\alpha\beta_1)$$

CB 2-10

Abbildung 2.16: *LR*(0)-Analyse und Ableitungsbäume

LR(0)–Informationen von G

$$G: \quad S' \to S \quad (0) \qquad S \to B \mid C \quad (1/2)$$
$$B \to aB \mid b \quad (3/4) \qquad C \to aC \mid c \quad (5/6)$$

$$I_0 := \mathbf{LR(0)}(\varepsilon): \quad [S' \to \cdot S] \quad [S \to \cdot B] \quad [S \to \cdot C] \quad [B \to \cdot aB] \quad [B \to \cdot b] \quad [C \to \cdot aC] \quad [C \to \cdot c]$$

$$I_1 := \mathbf{LR(0)}(S): \quad [S' \to S\cdot]$$

$$I_2 := \mathbf{LR(0)}(B): \quad [S \to B\cdot]$$

$$I_3 := \mathbf{LR(0)}(C): \quad [S \to C\cdot]$$

$$I_4 := \mathbf{LR(0)}(a): \quad [B \to a \cdot B] \quad [C \to a \cdot C] \quad [B \to \cdot aB] \quad [B \to \cdot b] \quad [C \to \cdot aC] \quad [C \to \cdot c]$$

$$I_5 := \mathbf{LR(0)}(b): \quad [B \to b\cdot]$$

$$I_6 := \mathbf{LR(0)}(c): \quad [C \to c\cdot]$$

$$I_7 := \mathbf{LR(0)}(aB): \quad [B \to aB\cdot]$$

$$I_8 := \mathbf{LR(0)}(aC): \quad [C \to aC\cdot]$$

$$I_9 := \mathbf{LR(0)}(Sa): \quad \emptyset$$

CB 2-11

Abbildung 2.17: $LR(0)$-Informationen von $\mathcal{G}$

goto– und **action**–Funktion zu G

$$G: \quad S' \to S \quad (0) \qquad S \to B \mid C \quad (1/2)$$
$$B \to aB \mid b \quad (3/4) \qquad C \to aC \mid c \quad (5/6)$$

goto	I_0	I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8	I_9
S	I_1	-	-	-	-	-	-	-	-	-
B	I_2	-	-	-	I_7	-	-	-	-	-
C	I_3	-	-	-	I_8	-	-	-	-	-
a	I_4	-	-	-	I_4	-	-	-	-	-
b	I_5	-	-	-	I_5	-	-	-	-	-
c	I_6	-	-	-	I_6	-	-	-	-	-

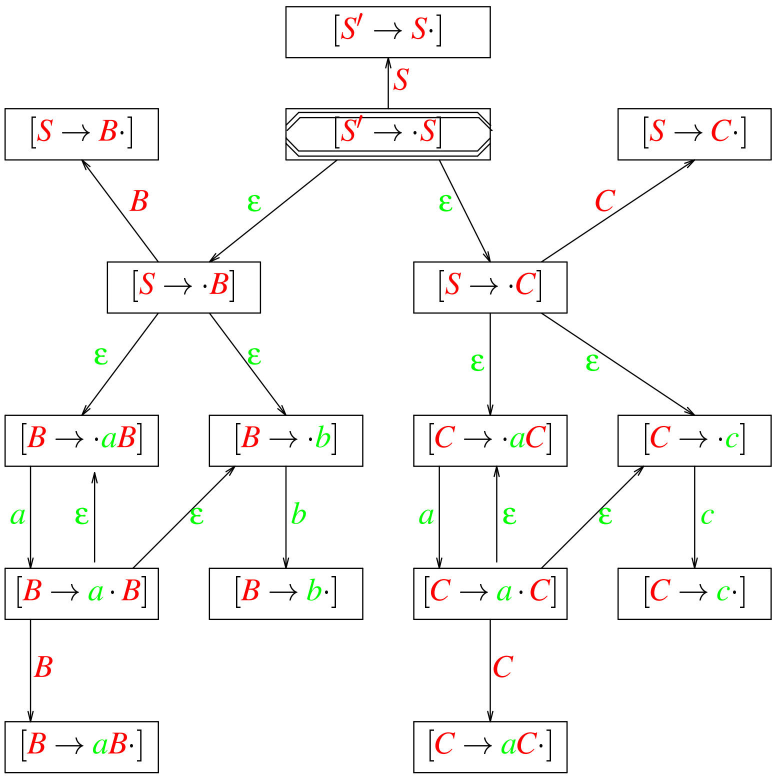
LR(0)(G)	**act**	**goto**					
		S	B	C	a	b	c
I_0	shift	I_1	I_2	I_3	I_4	I_5	I_6
I_1	accept						
I_2	red 1						
I_3	red 2						
I_4	shift		I_7	I_8	I_4	I_5	I_6
I_5	red 4						
I_6	red 6						
I_7	red 3						
I_8	red 5						
I_9	error						

CB 2-12

Abbildung 2.18: goto- und action-Funktion zu $\mathcal{G}$

Berechnung von $\mathbf{LR}(\mathbf{0})(G)$ und **goto** durch Potenzmengenkonstruktion

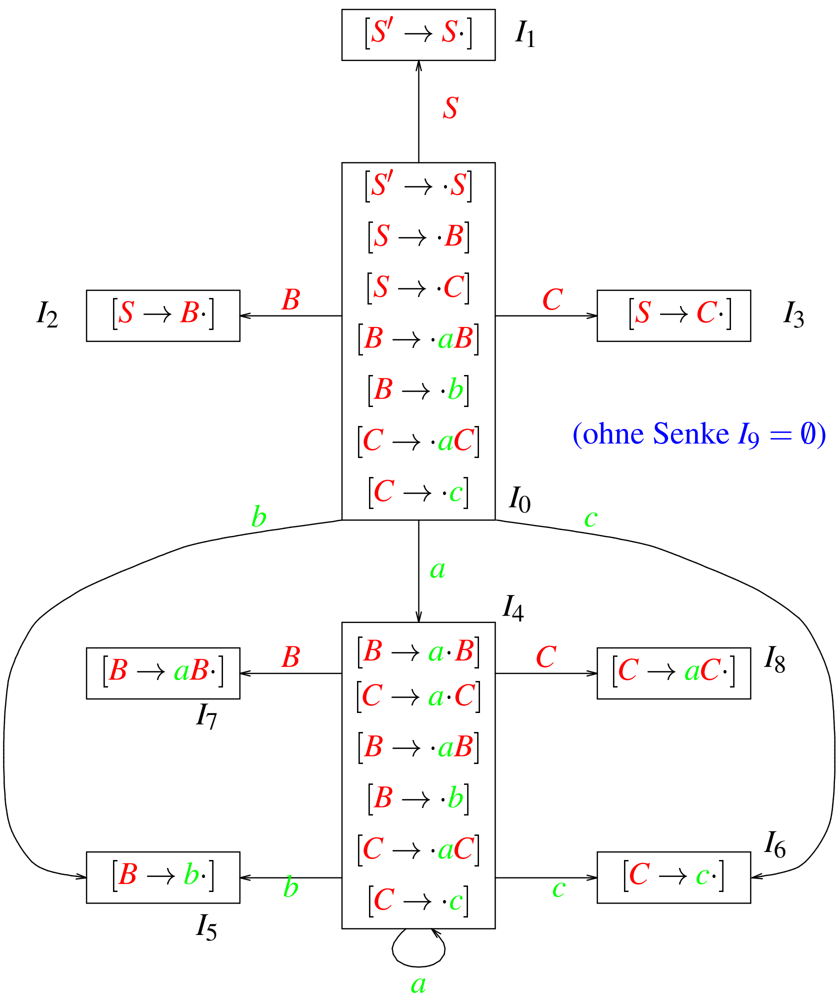
$$G: \quad \begin{array}{lclllcll} S' & \to & S & (0) & S & \to & B \mid C & (1/2) \\ B & \to & aB \mid b & (3/4) & C & \to & aC \mid c & (5/6) \end{array}$$



CB 2-13

Abbildung 2.19: Potenzmengenkonstruktion I

Potenzmengenkonstruktion (Fortsetzung)



CB 2-14

Abbildung 2.20: Potenzmengenkonstruktion II

SLR(1)–Analyse

$$
\begin{array}{llcll}
G_A E: & E' & \rightarrow & E & (0) \\
 & E & \rightarrow & E{+}T \mid T & (1,2) \\
 & T & \rightarrow & T{*}F \mid F & (3,4) \\
 & F & \rightarrow & (E) \mid a & (5,6)
\end{array}
$$

LR(0)–Informationen zu G_1:

$$
\begin{array}{ll}
I_0: & E' \rightarrow \cdot E \\
 & E \rightarrow \cdot E{+}T \\
 & E \rightarrow \cdot T \\
 & T \rightarrow \cdot T{*}F \\
 & T \rightarrow \cdot F \\
 & F \rightarrow \cdot (E) \\
 & F \rightarrow \cdot a
\end{array}
$$

$$
\boxed{\begin{array}{ll}
I_1: & E' \rightarrow E \cdot \\
 & E \rightarrow E \cdot {+}T
\end{array}} \;\lightning
$$

$$
\boxed{\begin{array}{ll}
I_2: & E \rightarrow T \cdot \\
 & T \rightarrow T \cdot {*}F
\end{array}} \;\lightning
$$

$$
\begin{array}{ll}
I_3: & T \rightarrow F \cdot
\end{array}
$$

$$
\begin{array}{ll}
I_4: & F \rightarrow (\cdot E) \\
 & E \rightarrow \cdot E{+}T \\
 & E \rightarrow \cdot T \\
 & T \rightarrow \cdot T{*}F \\
 & T \rightarrow \cdot F \\
 & F \rightarrow \cdot (E) \\
 & F \rightarrow \cdot a
\end{array}
$$

$$
\begin{array}{ll}
I_5: & F \rightarrow a \cdot
\end{array}
$$

$$
\begin{array}{ll}
I_6: & E \rightarrow E{+} \cdot T \\
 & T \rightarrow \cdot T{*}F \\
 & T \rightarrow \cdot F \\
 & F \rightarrow \cdot (E) \\
 & F \rightarrow \cdot a
\end{array}
$$

$$
\begin{array}{ll}
I_7: & T \rightarrow T{*} \cdot F \\
 & F \rightarrow \cdot (E) \\
 & F \rightarrow \cdot a
\end{array}
$$

$$
\begin{array}{ll}
I_8: & F \rightarrow (E \cdot) \\
 & E \rightarrow E \cdot {+}T
\end{array}
$$

$$
\boxed{\begin{array}{ll}
I_9: & E \rightarrow E{+}T \cdot \\
 & T \rightarrow T \cdot {*}F
\end{array}} \;\lightning
$$

$$
\begin{array}{ll}
I_{10}: & T \rightarrow T{*}F \cdot
\end{array}
$$

$$
\begin{array}{ll}
I_{11}: & F \rightarrow (E) \cdot
\end{array}
$$

CB 2-15

Abbildung 2.21: $SLR(1)$-Analyse von $\mathcal{G}_{AE}$

$$
\begin{array}{llll}
G_1: & E' \to E & & (0)\\
 & E \to E{+}T \mid T & & (1,2)\\
 & T \to T{*}F \mid F & & (3,4)\\
 & F \to (E) \mid a & & (5,6)
\end{array}
$$

SLR(1)–Analysetabelle zu G_1:

	action						**goto**							
	$+$	$*$	$($	$)$	a	$\$$	$+$	$*$	$($	$)$	a	E	T	F
I_0			shift		shift				I_4		I_5	I_1	I_2	I_3
I_1	shift					acc	I_6							
I_2	red 2	shift		red 2		red 2		I_7						
I_3	red 4	red 4		red 4		red 4								
I_4			shift		shift				I_4		I_5	I_8	I_2	I_3
I_5	red 6	red 6		red 6		red 6								
I_6			shift		shift				I_4		I_5		I_9	I_3
I_7			shift		shift				I_4		I_5			I_{10}
I_8	shift			shift			I_6			I_{11}				
I_9	red 1	shift		red 1		red 1		I_7						
I_{10}	red 3	red 3		red 3		red 3								
I_{11}	red 5	red 5		red 5		red 5								

A	**fo**(A)
E'	$\$$
E	$\$, +,)$
T	$\$, +,), *$
F	$\$, +,), *$

CB 2-16

Abbildung 2.22: $SLR(1)$-Analysetabelle

Motivation **LR(1)**–Analyse

$$
\begin{array}{llcll}
G_2: & S' & \rightarrow & S & (0) \\
 & S & \rightarrow & L{=}R \mid R & (1,2) \\
 & L & \rightarrow & *R \mid a & (3,4) \\
 & R & \rightarrow & L & (5)
\end{array}
$$

LR(0)–Informationen zu G_2:

$$
\begin{array}{lll}
I_0: & [S' \rightarrow \cdot S] \\
 & [S \rightarrow \cdot L{=}R] \\
 & [S \rightarrow \cdot R] \\
 & [L \rightarrow \cdot *R] \\
 & [L \rightarrow \cdot a] \\
 & [R \rightarrow \cdot L] \\
I_4: & [L \rightarrow * \cdot R] \\
 & [R \rightarrow \cdot L] \\
 & [L \rightarrow \cdot *R] \\
 & [L \rightarrow \cdot a] \\
I_7: & [L \rightarrow *R\cdot] \\
I_8: & [R \rightarrow L\cdot]
\end{array}
\qquad
\begin{array}{lll}
I_1: & [S' \rightarrow S\cdot] \\
\boxed{I_2:} & \boxed{[S \rightarrow L \cdot {=}R] \quad [R \rightarrow L\cdot]} & ↯ \\
I_3: & [S \rightarrow R\cdot] \\
I_5: & [L \rightarrow a\cdot] \\
I_6: & [S \rightarrow L{=}\cdot R] \\
 & [R \rightarrow \cdot L] \\
 & [L \rightarrow \cdot *R] \\
 & [L \rightarrow \cdot a] \\
I_9: & [S \rightarrow L{=}R\cdot]
\end{array}
$$

Der Konflikt in I_2 ist mit der SLR–Methode <u>nicht</u> lösbar, da

$$= \in \mathbf{fo}(R)$$

CB 2-17

Abbildung 2.23: Motiviation $LR(1)$-Analyse

LR(1)–Informationen zu G_2:

$I_0^{(1)}:$	$[S' \to \cdot S \quad ,\$]$	$[S \to \cdot L{=}R, \$]$	$[S \to \cdot R \quad ,\$]$
	$[L \to \cdot{*}R \quad ,=]$	$[L \to \cdot a \quad ,=]$	$[R \to \cdot L \quad ,\$]$
	$[L \to \cdot{*}R \quad ,\$]$	$[L \to \cdot a \quad ,\$]$	
$I_1^{(1)}:$	$[S' \to S\cdot \quad ,\$]$		
$I_2^{(1)}:$	$[S \to L\cdot{=}R, \$]$	$[R \to L\cdot \quad ,\$]$	$\begin{cases} \text{shift} & \text{, bei } = \\ \text{reduce} & \text{, bei } \$ \end{cases}$
$I_3^{(1)}:$	$[S \to R\cdot \quad ,\$]$		
$I_4^{(1)}:$	$[L \to {*}\cdot R \quad ,=]$	$[R \to \cdot L \quad ,=]$	$[L \to \cdot{*}R, =]$
	$[L \to \cdot a \quad ,=]$	$[L \to {*}\cdot R \quad ,\$]$	$[R \to \cdot L \quad ,\$]$
	$[L \to \cdot{*}R \quad ,\$]$	$[L \to \cdot a \quad ,\$]$	
$I_5^{(1)}:$	$[L \to a\cdot \quad ,=]$	$[L \to a\cdot \quad ,\$]$	
$I_6^{(1)}:$	$[S \to L{=}\cdot R, \$]$	$[R \to \cdot L \quad ,\$]$	$[L \to \cdot{*}R, \$]$
	$[L \to \cdot a \quad ,\$]$		
$I_7^{(1)}:$	$[L \to {*}R\cdot \quad ,=]$	$[L \to {*}R\cdot \quad ,\$]$	
$I_8^{(1)}:$	$[R \to L\cdot \quad ,=]$	$[R \to L\cdot \quad ,\$]$	
$I_9^{(1)}:$	$[S \to L{=}R\cdot \ ,\$]$		
$I_{10}^{(1)}:$	$[R \to L\cdot \quad ,\$]$		
$I_{11}^{(1)}:$	$[L \to {*}\cdot R \quad ,\$]$	$[R \to \cdot L \quad ,\$]$	$[L \to \cdot{*}R, \$]$
	$[L \to \cdot a \quad ,\$]$		
$I_{12}^{(1)}:$	$[L \to a\cdot \quad ,\$]$		
$I_{13}^{(1)}:$	$[L \to {*}R\cdot \quad ,\$]$		

CB 2-18

Abbildung 2.24: $LR(1)$-Informationen

$$
\begin{array}{lllll}
G_2: & S' & \rightarrow & S & (0) \\
 & S & \rightarrow & L{=}R \mid R & (1,2) \\
 & L & \rightarrow & *R \mid a & (3,4) \\
 & R & \rightarrow & L & (5)
\end{array}
$$

LR(1)–Analysetabelle zu G_2:

	action				goto					
	$*$	$=$	a	$\$$	$*$	$=$	a	S	L	R
$I_0^{(1)}$	shift		shift		$I_4^{(1)}$		$I_5^{(1)}$	$I_1^{(1)}$	$I_2^{(1)}$	$I_3^{(1)}$
$I_1^{(1)}$				acc						
$I_2^{(1)}$		shift		red 5		$I_6^{(1)}$				
$I_3^{(1)}$				red 2						
$I_4^{(1)}$	shift		shift		$I_4^{(1)}$		$I_5^{(1)}$		$I_8^{(1)}$	$I_7^{(1)}$
$I_5^{(1)}$		red 4		red 4						
$I_6^{(1)}$	shift		shift		$I_{11}^{(1)}$		$I_{12}^{(1)}$		$I_{10}^{(1)}$	$I_9^{(1)}$
$I_7^{(1)}$		red 3		red 3						
$I_8^{(1)}$		red 5		red 5						
$I_9^{(1)}$				red 1						
$I_{10}^{(1)}$				red 5						
$I_{11}^{(1)}$	shift		shift		$I_{11}^{(1)}$		$I_{12}^{(1)}$		$I_{10}^{(1)}$	$I_{13}^{(1)}$
$I_{12}^{(1)}$				red 4						
$I_{13}^{(1)}$				red 3						

CB 2-19

Abbildung 2.25: $LR(1)$-Analysetabelle

Berechnung des **LR(1)**–Analyseautomaten für $*a = a$:

$$
\begin{array}{llll}
 & \langle\, I_0^{(1)} & ,\ *a{=}a\$,\ \varepsilon & \rangle \\
\vdash & \langle I_0^{(1)} I_4^{(1)} & ,\ a{=}a\$,\ \varepsilon & \rangle \\
\vdash & \langle I_0^{(1)} I_4^{(1)} \underline{I_5^{(1)}} & ,\ {=}a\$,\ \varepsilon & \rangle \\
\vdash & \langle I_0^{(1)} I_4^{(1)} \underline{I_8^{(1)}} & ,\ {=}a\$,\ 4 & \rangle \\
\vdash & \langle I_0^{(1)} \underline{I_4^{(1)} I_7^{(1)}} & ,\ {=}a\$,\ 45 & \rangle \\
\vdash & \langle I_0^{(1)} I_2^{(1)} & ,\ {=}a\$,\ 453 & \rangle \\
\vdash & \langle I_0^{(1)} I_2^{(1)} I_6^{(1)} & ,\ a\$,\ 453 & \rangle \\
\vdash & \langle I_0^{(1)} I_2^{(1)} I_6^{(1)} \underline{I_{12}^{(1)}} & ,\ \$,\ 453 & \rangle \\
\vdash & \langle I_0^{(1)} I_2^{(1)} I_6^{(1)} \underline{I_{10}^{(1)}} & ,\ \$,\ 4534 & \rangle \\
\vdash & \langle I_0^{(1)} \underline{I_2^{(1)} I_6^{(1)} I_9^{(1)}} & ,\ \$,\ 45345 & \rangle \\
\vdash & \langle \underline{I_0^{(1)} I_1^{(1)}} & ,\ \$,\ 453451 & \rangle \\
\vdash & \langle\, \varepsilon & ,\ \$,\ 4534510 & \rangle
\end{array}
$$

CB 2-20

Abbildung 2.26: $LR(1)$-Analyseautomat

$$
\begin{array}{llll}
G_2: & S' & \rightarrow & S & (0) \\
 & S & \rightarrow & L{=}R \mid R & (1,2) \\
 & L & \rightarrow & *R \mid a & (3,4) \\
 & R & \rightarrow & L & (5)
\end{array}
$$

LALR(1)–Analysetabelle zu G_2:

	action				**goto**					
	$*$	$=$	a	$\$$	$*$	$=$	a	S	L	R
$I_0^{(1/0)}$	shift		shift		$I_4^{(1/0)}$		$I_5^{(1/0)}$	$I_1^{(1/0)}$	$I_2^{(1/0)}$	$I_3^{(1/0)}$
$I_1^{(1/0)}$				acc						
$I_2^{(1/0)}$		shift		red 5		$I_6^{(1/0)}$				
$I_3^{(1/0)}$				red 2						
$I_4^{(1/0)}$	shift		shift		$I_4^{(1/0)}$		$I_5^{(1/0)}$		$I_8^{(1/0)}$	$I_7^{(1/0)}$
$I_5^{(1/0)}$		red 4		red 4						
$I_6^{(1/0)}$	shift		shift		$I_4^{(1/0)}$		$I_5^{(1/0)}$		$I_8^{(1/0)}$	$I_9^{(1/0)}$
$I_7^{(1/0)}$		red 3		red 3						
$I_8^{(1/0)}$		red 5		red 5						
$I_9^{(1/0)}$				red 1						

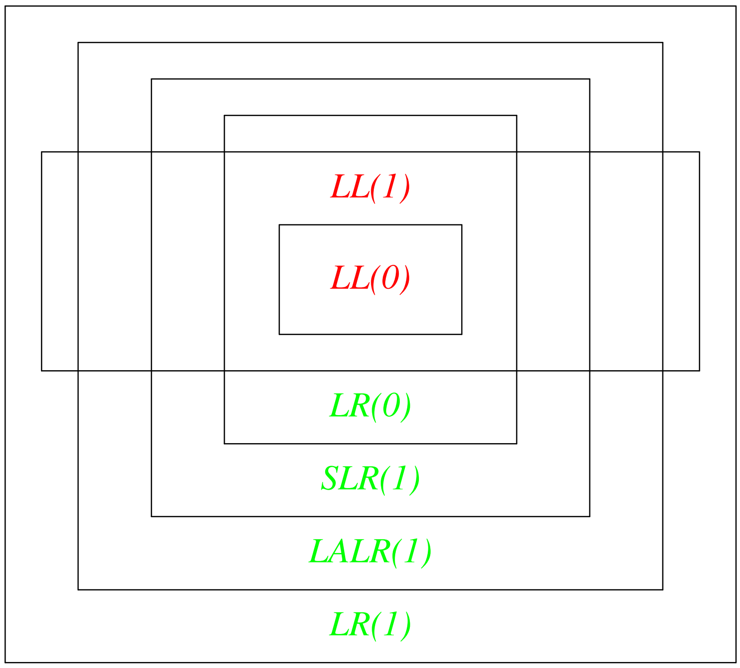
Dabei ist:

$$
\begin{array}{lcll}
I_i^{(1/0)} & = & I_i^{(1)} & i \in \{0,1,2,3,6,9\} \\
I_4^{(1/0)} & = & I_4^{(1)} \cup I_{11}^{(1)} & \\
I_5^{(1/0)} & = & I_5^{(1)} \cup I_{12}^{(1)} & \\
I_7^{(1/0)} & = & I_7^{(1)} \cup I_{13}^{(1)} & \\
I_8^{(1/0)} & = & I_8^{(1)} \cup I_{10}^{(1)} &
\end{array}
$$

CB 2-21

Abbildung 2.27: $LALR(1)$-Analysetabelle

Übersicht der Grammatikklassen

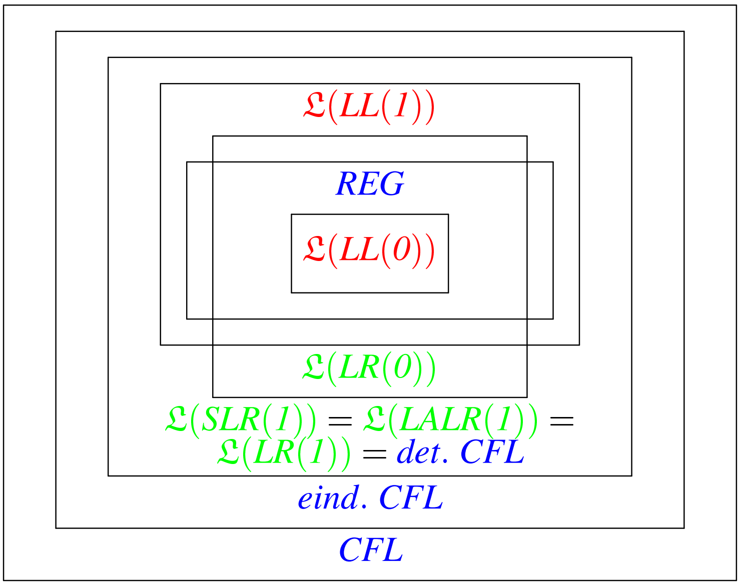


- $LL(0) \subsetneq LL(1) \subsetneq LR(1)$
- $LR(0) \subsetneq SLR(1) \subsetneq LALR(1) \subsetneq LR(1)$

CB 2-22

Abbildung 2.28: Übersicht der Grammatikklassen

Übersicht der Sprachklassen



Also (vgl. O. Mayer: *Syntaxanalyse*, BI 1978):

- $\mathfrak{L}(LL(0)) \subsetneqq REG \subsetneqq \mathfrak{L}(LL(1)) \subsetneqq \mathfrak{L}(SLR(1))$
- $\mathfrak{L}(LR(0)) \subsetneqq \mathfrak{L}(SLR(1)) = \mathfrak{L}(LALR(1))$
  $= \mathfrak{L}(LR(1)) = det.\ CFL \subsetneqq eind.\ CFL \subsetneqq CFL$
- $\mathfrak{L}(LR(0)) \not\subseteq \left\{ \begin{matrix} LL(1) \\ REG \end{matrix} \right\}, \left\{ \begin{matrix} LL(1) \\ REG \end{matrix} \right\} \not\subseteq \mathfrak{L}(LR(0)),$

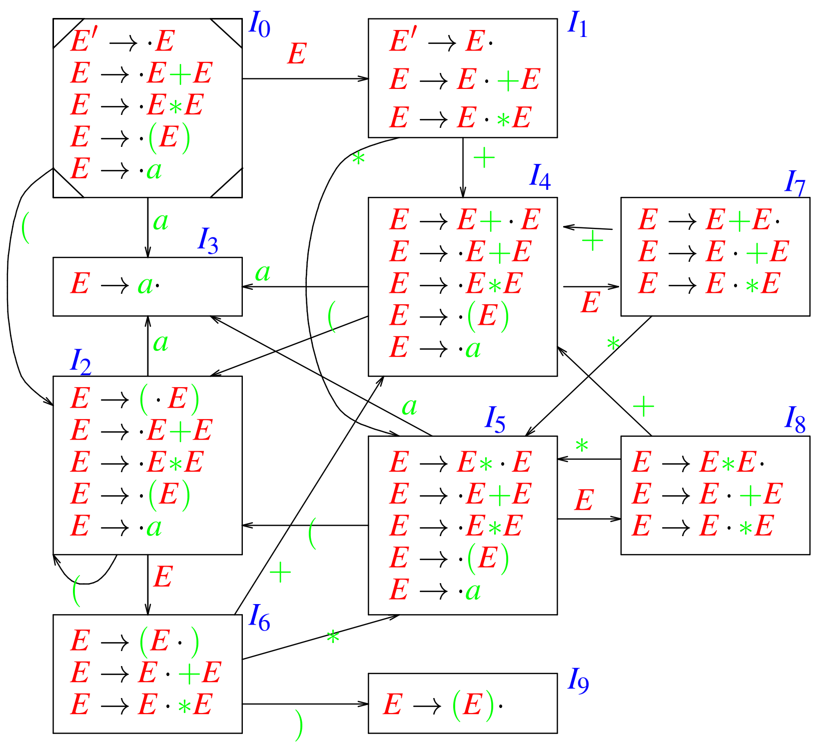
  $\mathfrak{L}(LR(0)) \cap \left\{ \begin{matrix} LL(1) \\ REG \end{matrix} \right\} \neq \emptyset$

CB 2-23

Abbildung 2.29: Übersicht der Sprachklassen

$$G_3 : E \to E{+}E \mid E{*}E \mid (E) \mid a \qquad (1,2,3,4)$$

LR(0)–Informationen zu G_3:



CB 2-24

Abbildung 2.30: $LR(0)$-Informationen von $\mathcal{G}_3$

$$G_3 : E \to E{+}E \mid E{*}E \mid (E) \mid a \qquad (1,2,3,4)$$

Analysetabelle zu G_3:

	action/goto						goto
	a	$+$	$*$	(	)	\$	E
I_0	shift 3			shift 2			I_1
I_1		shift 4	shift 5			acc	
I_2	shift 3			shift 2			I_6
I_3		red 4	red 4		red 4	red 4	
I_4	shift 3			shift 2			I_7
I_5	shift 3			shift 2			I_8
I_6		shift 4	shift 5		shift 9		
I_7		red 1	shift 5		red 1	red 1	
I_8		red 2	red 2		red 2	red 2	
I_9		red 3	red 3		red 3	red 3	

SLR

$* \cdot > +$

linksass.

wobei **action**(I, b) = **shift i** bedeutet:

$$\textbf{action}(I,b) = \textbf{shift}, \quad \textbf{goto}(I,b) = I_i$$

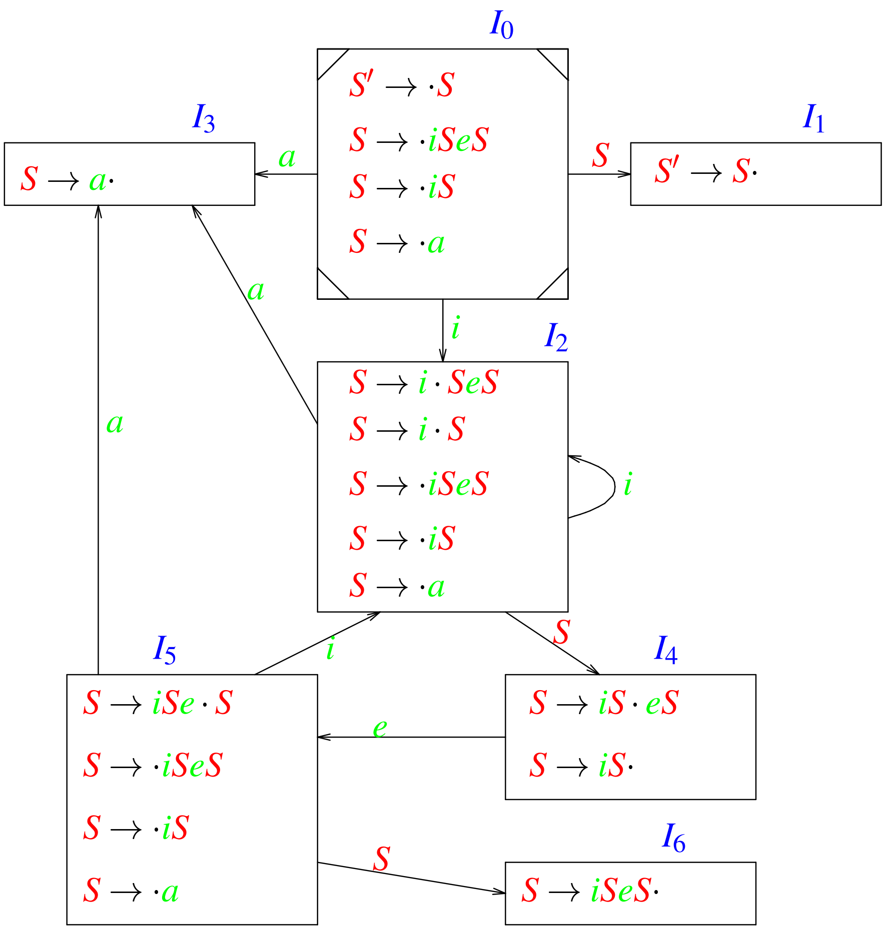
(Tabellenkompression)

CB 2-25

Abbildung 2.31: Analysetable von $\mathcal{G}_3$

$$G_4 : S \to iSeS \mid iS \mid a \qquad (1,2,3)$$

LR(0)–Informationen zu G_4:



CB 2-26

Abbildung 2.32: $LR(0)$-Informationen von $\mathcal{G}_4$

Kapitel 3

Semantische Analyse, Attributgrammatiken

[15.06.2004]

Das Ergebnis der syntaktischen Analyse ist der Ableitungsbaum.

hier werden jetzt kontextabhängige Eigenschaften betrachtet:

- Deklariertheit von Bezeichnern (vgl. ATFS: $\{ww|w \in \Sigma^*\} \notin CFL$)
- Typinformationen sind nicht durch CFG beschreibbar

Festlegung dieser Eigenschaften durch:

- Gültigkeitsregeln: Gültigkeitsbereich einer Deklaration
- Sichtbarkeitsregeln: Sichtbarkeit im Gültigkeitsbereich (Überdeckung globaler durch lokale Deklaration)
- Typvorschriften: Typkonsistenz

Definition: Statische Semantik: kontextabhängige, laufzeitunabhängige Eigenschaften eines Programms.

Formale Beschreibung durch *Attributgrammatiken*.

Idee: (lt. Knuth) CFG + semantische Regeln $\rightsquigarrow$ Zusatzinformationen für den Ableitungsbaum.

Semantische Analyse = Attributberechnung.

Als Ergebnis erhält man den attributierten Ableitungsbaum. Dieser bildet die Grundlage für die anschließende Synthesephase (Codegenerierung).

3.1 Attributgrammatiken

Idee: Attribute für $A \in N$ und zusätzliche semantische Regeln für ihre Berechnung.

synthetische Attribute: Bottom-Up-Berechnung
inherite Attribute: Top-Down-Berechnung

$\curvearrowright$ beliebiger Informationstransfer im Ableitungsbaum.

Attributwerte: Symboltabellen, Typen, Code, Fehler. Daher breite Anwendbarkeit von Attributgrammatiken, syntaxgerichtete Programmierung. In Compilergeneratoren erfolgt eine automatische Attributauswertung.

Im Compilerbau: statische Semantik, Programmanalyse für Compileroptimierung, Codegenerierung und Fehlerbehandlung. Zurückgehend auf Knuth 1968: Semantics of CFLs.

Beispiel: (Binärzahlen) G_B:

1. Attributisierung von G_B mit synthetischen Attributen:

$$\begin{array}{lcll}
B & \to & 0 & v.0 = 0 \\
B & \to & 1 & v.0 = 1 \\
L & \to & B & v.0 = v.1 \\
 & & & l.0 = 1 \\
L & \to & LB & v.0 = 2 * v.1 + v.2 \\
 & & & l.0 = l.1 + 1 \\
N & \to & L & v.0 = v.1 \\
N & \to & L.L & v.0 = v.1 + v.2/2^{l.2}
\end{array}$$

G_B erzeugt Binärzahlen mit und ohne Punkt. N ist das Startsymbol. Als synthetische Attribute sind $B,N : v$ („value“) und $L : v,l$ („length“). Semantische Regeln sind Attributgleichungen mit Attributvariablen. Mit dem Index i bezeichnen wir das i-te Nichtterminal-Symbol.

Ziel: Bestimmung des Zahlwertes

- Attributwerte von v: rationale Zahlen (A^v)
- Attributwerte von l: natürliche Zahlen (A^l)

2. Attributisierung von G_B mit synthetischen und inheriten Attributen:

Zusätzliches *inherites* Attribut für Bits und Listen: p („position“)

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

Berechnung des Wurzelattributs:

1. Längen ($\uparrow$: Bottom-Up)

2. Positionen ($\downarrow$: Top-Down)

3. Wert ($\uparrow$: Bottom-Up)

[18.06.2004]

Definition: (Attributgrammatik): Sei $G =< N, \Sigma, P, S >\in$ CFG. Sei $\underline{\text{Att}}$ eine Menge von *Attributen*, $A = (A^\alpha | \alpha \in \underline{\text{Att}})$ eine Familie von *Attributwertmengen* und $\underline{\text{att}} : \chi \to \mathfrak{p}(\underline{\text{Att}})$ eine *Attributzuordnung*.

Sei $\underline{\text{Att}} = \underline{\text{Syn}} \uplus \underline{\text{Inh}}$ eine Zerlegung in Teilmengen *synthetischer* und *inheriter* Attribute, so daß $\underline{\text{att}}$ zerfällt in

$$
\begin{array}{lllll}
 & \underline{\text{syn}}: & \chi \to \mathfrak{p}(\underline{\text{Syn}}) & \text{mit } \underline{\text{syn}}(X) & = \underline{\text{att}}(X) \cap \underline{\text{Syn}} \\
\text{und} & \underline{\text{inh}}: & \chi \to \mathfrak{p}(\underline{\text{Inh}}) & \text{mit } \underline{\text{inh}}(X) & = \underline{\text{att}}(X) \cap \underline{\text{Inh}}
\end{array}
$$

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

$$
\begin{array}{llll}
 & \underline{\text{IVar}}_\Pi := & \{\alpha.i| & (i = 0 \text{ und } \alpha \in \underline{\text{syn}}(X_0)) \text{ oder} \\
 & & & (1 \leq i \leq r \text{ und } \alpha \in \underline{\text{inh}}(X_i))\} \\
\text{und} & \underline{\text{OVar}}_\Pi := & \multicolumn{2}{l}{\underline{\text{Var}}_\Pi \backslash \underline{\text{IVar}}_\Pi}
\end{array}
$$

der *Innenvariablen* und *Außenvariablen*

Eine *Attributgleichung* von Π (semantische Regel) hat die Form

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

mit $\alpha.i \in \underline{\text{IVar}}_\Pi, \alpha_1.i_1, \dots, \alpha_n.i_n \in \underline{\text{OVar}}_\Pi, f : A^{\alpha_1} \times \dots \times A^{\alpha_n} \to A^\alpha$ 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 $\mathfrak{A} = (G, E)$ eine *Attributgrammatik*: $\mathfrak{A} \in AG$.

Definition: (Semantik von $G \in AG$ = Das Attributgleichungssystem eines Ableitungsbaums):
Sei $\mathfrak{A} = (G, E) \in AG$. $\mathfrak{A}$ induziert für jeden Ableitungsbaum t von G ein *Attibutgleichungssystem* E_t.

Sei $\underline{\mathrm{Kn}}(t)$ die Menge der Knoten von t. Sie bestimmt die Menge

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

der aktuellen *Attributvariablen* von t.

Wird an einem inneren Knoten (kein Blatt) $k_0 \in \underline{\mathrm{Kn}}(t)$ die Regel $\Pi = X_0 \to X_1 \dots X_r$ angewandt und sind $k_1, \dots, k_r \in \underline{\mathrm{Kn}}(t)$ die entsprechenden Nachfolgerknoten, so erhält man das *Attributgleichungssystem* E_{k_0} von k_0 aus E_Π durch Indexsubstitution $(i \mapsto k_i | 0 \leq i \leq r)$ 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 Wurzel und die synthetischen der Blätter, gibt es genau eine Gleichung der Form $\alpha.k = \dots$

Annahme:

- keine inheriten Attribute des Startsymbols
- synthetische Attribute der Terminalsymbole vom Scanner

3.1.1 Lösbarkeit von E_t

E_t kann keine, genau eine, oder mehrere Lösungen haben.

Beispiel: G enthalte die Regeln $A \to aBc$ (mit B an der 2-ten Position) und $B \to w$. Ferner sei $\alpha \in \underline{\mathrm{syn}}(B)$ und $\beta \in \underline{\mathrm{inh}}(B)$. Als Attributgleichungen erhalten wir $\beta.2 = f(\alpha.2), \alpha.0 = g(\beta.0)$.

Der Ableitungsbaum t ergibt in E_t die *zirkuläre Abhängigkeit* mit

$$\beta.k = f(\alpha.k) \,\text{und}\, \alpha.k = g(\beta.k) \Rightarrow \beta.k = h(\beta.k)$$

Für $A^\alpha = A^\beta = \mathbb{N}, g = \underline{\mathrm{id}}_\mathbb{N}$ und

1. $f(x) = x + 1 \rightsquigarrow$ keine Lösung
2. $f(x) = 2x \rightsquigarrow$ genau eine Lösung
3. $f(x) = x \rightsquigarrow$ mehrere Lösungen (unendlich viele)

Folgerung: Zirkularitäten können auftreten, sie sind unerwünscht wegen Mehrdeutigkeiten.
Sie entstehen erst im $E_t = \bigcup\{E_k | k \text{ innerer Knoten von } t\}$.

[22.06.2004]

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

3.1.2 Ein Zirkularitätstest für Attributgrammatiken

Sei $\mathfrak{A} =< G,E >\in AG$. Eine Regel $\Pi = X_0 \to X_1 \ldots X_r$ bestimmt ihren *Abhängigkeitsgraphen* DG_Π mit der Knotenmenge $\underline{\text{Kno}}(DG_\Pi) := \underline{\text{OKno}}(DG_\Pi) \cup \underline{\text{UKno}}(DG_\Pi)$

$$\begin{array}{lll} \underline{\text{OKno}}(DG_\Pi) & := \{\alpha.0 | \alpha \in \underline{\text{att}}(X_0)\} & \text{(Oberknoten)} \\ \underline{\text{UKno}}(DG_\Pi) & := \{\alpha.i | \alpha \in \underline{\text{att}}(X_i), 1 \leq i \leq r\} & \text{(Unterknoten)} \end{array}$$

und der *Kantenmenge* $\underline{\text{Kan}}(DG_\Pi)$:

$$(x_1,x_2) \in \underline{\text{Kan}}(DG_\Pi) : \curvearrowright x_2 = f(\ldots x_1 \ldots) \in E_\Pi$$

Beachte: $Var_\Pi = \underline{\text{UKno}}(DG_\Pi) \uplus \underline{\text{OKno}}(DG_\Pi) = \underline{\text{OVar}}_\Pi \uplus \underline{\text{IVar}}_\Pi$

insbesondere: $\underline{\text{Kan}}(DG_\Pi) \subseteq \underline{\text{OVar}}_\Pi \times \underline{\text{IVar}}_\Pi$, also: keine Zirkularität in DG_Π.

Problem: Beim Verkleben der DG_Π zu DG_t für einen Ableitungsbaum t können Schleifen auftreten.

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

Bemerkung: DG_t entsteht aus $(DG_\Pi | \Pi \in P)$, wie E_t aus $(E_\Pi | \Pi \in P)$ und t aus $(\Pi | \Pi \in P)$.

Beobachtung: Eine Schleife in DG_t bestimmt einen Teilgraphen bei der eine Regel Π mit verbindbaren Unterknoten von DG_Π existiert. Eine Verbindung gehört zu einem Nichtterminal-Symbol und verläuft von einer inheriten zu einer synthetischen Variable.

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

$$(\alpha.k, \alpha'.k) \in \underline{\text{trans}}(\underline{\text{Kan}}(DG_t))$$

Beachte: Paare abhängiger Attribute von verschiedenen Ableitungsbäumen mit gleicher Wurzel müssen bei der Ermittlung von Schleifen getrennt behandelt werden, weil sonst unzulässige Abhängigkeiten entstehen können.

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

$$\begin{array}{ll} D(A,t) & := \{(\alpha,\alpha') | \alpha \xrightarrow{A} \alpha' \text{ in } t\} \\ \mathscr{D}(A) & := \{D(A,t) | t \text{ Ableitungsbaum mit Wurzel } A\} \end{array}$$

Bei der induktiven Bestimmung benutzen wir folgende Bezeichnung:
Definition: Für $\Pi = A_0 \to w_0 A_1 w_1 A_2 \ldots A_r w_r$ und $D_i \subseteq \underline{\text{inh}}(A_i) \times \underline{\text{syn}}(A_i)$ definieren wir

$$D[\Pi; D_1 \ldots D_r] \subseteq \underline{\text{inh}}(A_0) \times \underline{\text{syn}}(A_0)$$

durch

$$\{(\alpha,\alpha') | (\alpha.0, \alpha'.0) \in \underline{\text{trans}}(\underline{\text{Kan}}(DG_\Pi) \cup \bigcup_{i=1}^{r} \{(\beta.i, \beta'.i) | (\beta,\beta') \in D_i\})\}$$

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

1. $\Pi = A \to w \curvearrowright D[\Pi;] \in \mathscr{D}(A)$
2. $\Pi = A \to w_0 A_1 w_1 \dots A_r w_r, D_i \in \mathscr{D}(A_i), (1 \leq i \leq r) \curvearrowright D[\Pi; D_1, \dots, D_r] \in \mathscr{D}(A)$

Beweis: durch Induktion über die Struktur der Ableitungsbäume.

Bemerkung: Da für $D \in \mathscr{D}(A)$ gilt: $D \subseteq \underline{\text{inh}}(A) \times \underline{\text{syn}}(A)$ bricht der Berechnungsprozeß nach endlich vielen Schritten ab.

[25.06.2004]

Folgerung: (Zirkularitätstest): $\mathfrak{A} \in AG$ ist zirkulär $\curvearrowleftright$ es gibt

$$\Pi = A_0 \to w_0 A_1 w_1 \dots A_r w_r, \alpha.k \in \underline{\text{UKno}}(DG_\Pi) \text{ und } D_i \in \mathscr{D}(A_i)(1 \leq i \leq r)$$

so daß

$$(\alpha.k, \alpha.k) \in \underline{\text{trans}}(\underline{\text{Kan}}(DG_\Pi) \cup \bigcup_{i=1}^{r} \{(\beta.i, \beta'.i) | (\beta, \beta') \in D_i\})$$

Komplexität des Zirkularitätstest

$n = |\mathfrak{A}|, T(n)$ Zeit zur Entscheidung der Zirkularität.

$$2^{c \cdot n / \log(n)} \leq T(n) \leq 2^{d \cdot n^2}$$

3.1.3 Stark-nichtzirkuläre Grammatiken

Keine Trennung der Abhängigkeitsmengen verschiedener Ableitungsbäume.

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

Hinreichendes Kriterium für die Nichtzirkularität; Test in Polynomzeit.
Beachte: Es gibt $\mathfrak{A} \in AG$, so daß A nicht-zirkulär ist, ohne stark-nichtzirkulär zu sein.

3.1.4 Attributberechnung

1. E_t als Termersetzungsssystem, Top-Down-Berechnung. Keine Zirkularität $\curvearrowright$ Termination mit eindeutiger Lösung.
2. Bottom-Up-Auflösung von E_t: Variablen durch Werte ersetzen und Terme ausrechnen.
3. Uniforme Berechnung, unabhängig vom Ableitungsbaum
   (a) Berechnungspläne für $(E_\Pi | \Pi \in P)$ aufstellen.
   (b) Rekursive Prozeduren/Funktionen für $(E_\Pi | \Pi \in P)$ aufstellen.
   Idee: Jedem synthetischen Attribut einer Variablen wird eine Prozedur/Funktion zugeordnet mit ihren inheriten Attributen als Parameter.
4. Spezialfall: SAG, LAG mit Attributberechnung während der Syntaxanalyse.

3.2 S-Attributgrammatiken

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

Folgerung: Bottom-Up-Berechnung der Attributwerte.

Spezialfall: nur Bestimmung der Wurzelattribute, der Rest interessiert nicht und existiert nur temporär auf dem Keller während der Ableitung.

Idee: Durchführung während der Bottom-Up-Syntaxanalyse, zwischenspeichern von Attributwerten im Analysekeller.

Beispiel: $A \to BaD$

I_1	$v1$
I_2	$v2$
I_3	$v3$

$\longrightarrow$

I_4	$v4$

Beim Reduktionsschritt werden simultan die Werte der synthetischen Attribute von A aus den Werten unter $v1, v2, v3$ berechet („Records“).
Beim Acceptschritt dann Ausgabe der Wurzelattribute.

[29.06.2004]

Beispiel: Stackcode für arithmetische Ausdrücke

$$\begin{array}{llll} E & \to & (E+E)| & c.0 = c.2; c.4; ADD \\ & | & (E*E)| & c.0 = c.2; c.4; MULT \\ & | & id| & c.0 = LOAD\, a.1 \text{ (lexikalische Adresse)} \\ & | & num & c.0 = LIT\, n.1 \text{ (lexikalischer Wert)} \end{array}$$

Werte von $a.1, n.1$: lexikalische Attribute, die mit dem Token vom Scanner übergeben: $(id, X)\,(num, 5)$

$$((3+X)*(Y+5))$$
$$\downarrow \text{Scanner}$$
$$((< num, 3 > + < id, X >) * (< id, Y > + < num, 5 >))$$

Codegenerierung durch Attributauswertung während der Syntaxanalyse.

Shift: Aufruf des Scanners durch „nextsym“

(i) Token $\mapsto LR(0)$-Menge (mit goto)

(ii) lexikalisches Attribut

Beide Informationen auf dem Analysekeller.

Reduce: (i) Reduktion auf dem Analysekeller

(ii) simultane Attributberechnung

Accept: Wurzelattribut; in diesem Fall: der Stackcode des Ausdrucks.

Das Wurzelattribut ergibt sich dann als: $LIT\,3; LOADX; ADD; LOADY; LIT\,5; ADD; MULT$

Beispiel: Berechnung des *Abstrakten Syntaxbaums* (AST). Konkrete Syntax und abstrakte Syntax:

$$\Pi = A_0 \to w_0 A_1 w_1 \dots A_r w_r$$

repräsentiert Operationssymbol F_Π vom Typ $A_1 \times \dots \times A_r \to A_0$ (A_i Sorte, Typ)

Das „Verkleben“ der Regeln zu einem Ableitungsbaum entspricht der funktionalen Applikation der zugehörigen Operationssymbole.

Folgerung: Der Ableitungsbaum vereinfacht sich zu dem AST. Nur der AST ist für die Übersetzung erforderlich.

Die *konkrete Syntax* ist die benutzerfreundliche Darstellung in „Mixfix“-Notation unter Verwendung natürlicher Sprache für die Terminalsymbole der kontextfreien Grammatik.
Die *abstrakte Syntax* ist eine darstellungsunabhängige algebraische Struktur (→ algebraischer Datentyp, Konstruktoren).
Beispiel: (siehe Abbildung 3.6) $\underline{\text{if}}\ \underline{\text{id}} < \underline{\text{num}}\ \underline{\text{then}}\ \underline{\text{id}} = \underline{\text{num}} + \underline{\text{id}}\ \underline{\text{fi}}$

Die Regeln selbst können auch durch Operationssymbole dargestellt werden: Regelbaum.

Aufgabe: Berechnung des AST während der Bottom-Up-Analyse. Dazu Methode der S-Attributgrammatik.

Darstellung des Aufbaus von ASTs durch Konstruktion von Graphen:

1. Graphen

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

2. Konstruktorfunktionen für Graphen

$$f^{(n)} \in \Omega \mapsto \underline{\text{mk} - \text{f}} : \underline{\text{Graph}}^n \to \underline{\text{Graph}}$$

$$\underline{\text{mk} - \text{f}}((a_1, k_1), \dots, (a_n, k_n)) := (a, k)$$

mit neuer Adresse a und

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

Die Disjunktheit der k_i erfolgt automatisch durch die Konstruktion.

S-Attributierung von G_s: Wurzeladresse als S-Attribut im Analysekeller, Heap statt Ausgabebandes benutzen. Anstatt mit $\underline{\text{print}}$-Funktion die einzelnen Regelnummern auszugeben, wird jetzt die entsprechende $\underline{\text{mk} - \text{f}_i}$-Funktion benutzt.
Die Attributwerte sind nun Graphen, das Analyseergebnis ist schließlich die Graphdarstellung des AST auf dem Heap.
$\rightsquigarrow$ dies bildet die Basis für weitere Attributierung.

Beachte: Unterschied zwischen Wurzelattributberechnung und voller Attributierung (Erweiterung).

Beispiel: Typberechnung für arithmetische Ausdrücke

$\underline{\text{Typ}} := \{int, real\}$ mit $\underline{\text{max}} : \underline{\text{Typ}}^2 \to \underline{\text{Typ}}$, bzgl. $int < real$

Der Aufruf von „nextsym()“ (Lexer) bestimmt das Typattribut von $\underline{\text{num}}$, Typattribut von $\underline{\text{id}}$ über lookup-table $\rightsquigarrow$ Syntaxbaum mit Typinformationen (weiteres Knotenfeld) annotiert.

3.3 L-Attributgrammatiken

[02.07.2004]

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

Folgerung: $G \in LAG$ nicht zirkulär. Attributberechnung in „depth-first, left-to-right order“. Baumreise mit 2 Knotenbesuchen:

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

3.3.1 Syntaxanalyse mit L-Attributauswertung

Sei $\mathfrak{A} =< G, E >\in LAG$ mit $G \in LL(1)$

Ziel: Erweiterung der Top-Down Analyse zur Berechnung eines synthetischen Wurzelattributs.

Methode: Expansion von $A \in N$ auf Analysekeller so gestalten, daß spätere Reduktion möglich: Kombination von Top-Down (inherite Attribute) und Bottom-Up (synthetische Attribute).

Kelleralphabet: $\Gamma := \bigcup_{\Pi \in P} (\underline{\text{LR}(0)}_\Pi(G) \times \underline{\text{Val}}_\Pi) \cup \{(\to \cdot S, \emptyset), (\to S\cdot, \underline{\text{val}})\}$ mit $\underline{\text{val}} : \underline{\text{syn}}(S) \to A$,
$\underline{\text{LR}(0)}_\Pi(G) := \{[A \to \alpha.\beta] | \Pi = A \to \alpha\beta\}$ und $\underline{\text{Val}}_\Pi = \{\underline{\text{val}}_\Pi | \underline{\text{val}}_\Pi : \underline{\text{Var}}_\Pi - \to A$ („Attributwertmege“) $\}$

Konfigurationen: (w, γ) mit w als Eingabe und $\gamma \in \Gamma$.

1. „expand“-Schritt mit Berechnung inheriter Attribute.

$\vdots$	
$[A \to X_1 \ldots X_{i-1} \cdot B\gamma]$	$\underline{\text{val}}$

$\Rightarrow$

$\vdots$	
$[A \to X_1 \ldots X_{i-1} \cdot B\gamma]$	$\underline{\text{val}}$
$[B \to \cdot\beta]$	$\underline{\text{val}}'$

Dabei ist $B \to \beta$ wegen $G \in LL(1)$ durch Eingabe-„look-ahead“ bestimmt.

$\underline{\text{val}}'$ belegt die inheriten Attributvariablen von B nach Attributgleichungen für $A \to X_1 \dots X_{i-1} B\gamma$.

Sei $\alpha \in \underline{\text{inh}}(B)$ und $(\alpha.i = t) \in E_{A \to X_1 \dots \gamma}$, so $\underline{\text{val}}'(\alpha.0) = \underline{\text{val}}(t) \in A$ (Wert von t bezüglich $\underline{\text{val}}$) „Einsetzen der Attributwerte und ausrechnen“.

2. „match“-Schritt:

$$\begin{array}{|c|c|}\hline \vdots & \\ \hline [A \to \gamma \cdot a\gamma'] & \underline{\text{val}} \\ \hline \end{array} \Rightarrow \begin{array}{|c|c|}\hline \vdots & \\ \hline [A \to \gamma a \cdot \gamma'] & \underline{\text{val}} \\ \hline \end{array}$$

falls a nächstes Eingabesymbol. Eingabekopf rückt vor

3. „reduce“-Schritt mit Berechnung der synthetischen Attribute.

$$\begin{array}{|c|c|}\hline \vdots & \\ \hline [A \to X_1 \dots X_{i-1} \cdot B\gamma] & \underline{\text{val}} \\ \hline [B \to \beta \cdot] & \underline{\text{val}}' \\ \hline \end{array} \Rightarrow \begin{array}{|c|c|}\hline \vdots & \\ \hline [A \to X_1 \dots X_{i-1} B \cdot \gamma] & \underline{\text{val}}'' \\ \hline \end{array}$$

$\underline{\text{val}}''$ erweitert $\underline{\text{val}}$ um synthetische Attribute von B.

$\underline{\text{val}}''(\alpha.i) = \underline{\text{val}}(t)$ (bezüglich $\underline{\text{val}}'$) falls $\alpha.0 = t \in E_{B \to \beta}$, bzw. $\underline{\text{val}}''(\alpha.i) = \underline{\text{val}}(\alpha.i)$ sonst (unverändert).

3.3.2 Anwendung von LAG

[06.07.2004]

Überprüfung der Deklariertheit von Bezeichnern. G sei gegeben durch:

Programm	$P \to DL; SL$
Deklarations-Liste	$DL \to V \mid V; DL$
Statement-Liste	$SL \to S \mid S; SL$
Variable	$V \to a \mid b \mid \dots$
Statement	$S \to V := V$
Beispiel	$c; b; a; a := c; b := a; d := a$

<u>Attribute</u>

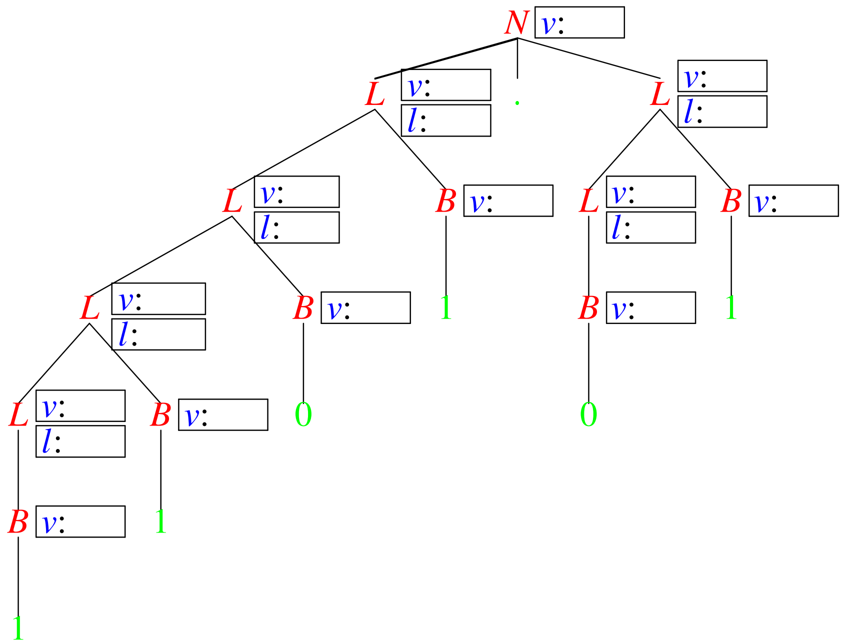
syn. A.	v	deklarierte oder benutze Variable für V mit Wert aus $\{a, b, c, \dots\}$
syn. A.	dv	Menge von deklarierten Variablen für DL mit Werten aus $\subseteq \{a, b, c, \dots\}$
inh. A.	env	Umgebung für S, SL mit Werten aus $\subseteq \{a, b, c, \dots\}$
syn. A.	$decl$	„deklariert“ für S, SL, P mit Werten aus $\{true, false\}$

Attributgleichungen		
$P \rightarrow DL;SL$	$decl.0 = decl.3$	
	$env.3 = dv.1$	(Beachte: $1 < 3! \Rightarrow LAG$)
$DL \rightarrow V$	$dv.0 = \{v.1\}$	
$DL \rightarrow V;DL$	$dv.0 = \{v.1\} \cup dv.3$	
$V \rightarrow a$	$v.0 = a$	
$SL \rightarrow S$	$env.1 = env.0$	
	$decl.0 = decl.1$	
$SL \rightarrow S;SL$	$env.1 = env.0$	
	$env.3 = env.0$	
	$decl.0 = decl.1 \ \underline{AND}\ decl.3$	
$S \rightarrow V := V$	$decl.0 = v.1 \in env.0 \ \underline{AND}\ v.3 \in env.0$	

3. Semantische Analyse

Bsp.: (Binärzahlen von Knuth)

$$
\begin{array}{lllll}
G_B: & (1) & B \to 0 & (2) & B \to 1 \\
 & (3) & L \to B & (4) & L \to LB \\
 & (5) & N \to L & (6) & N \to L.L
\end{array}
$$

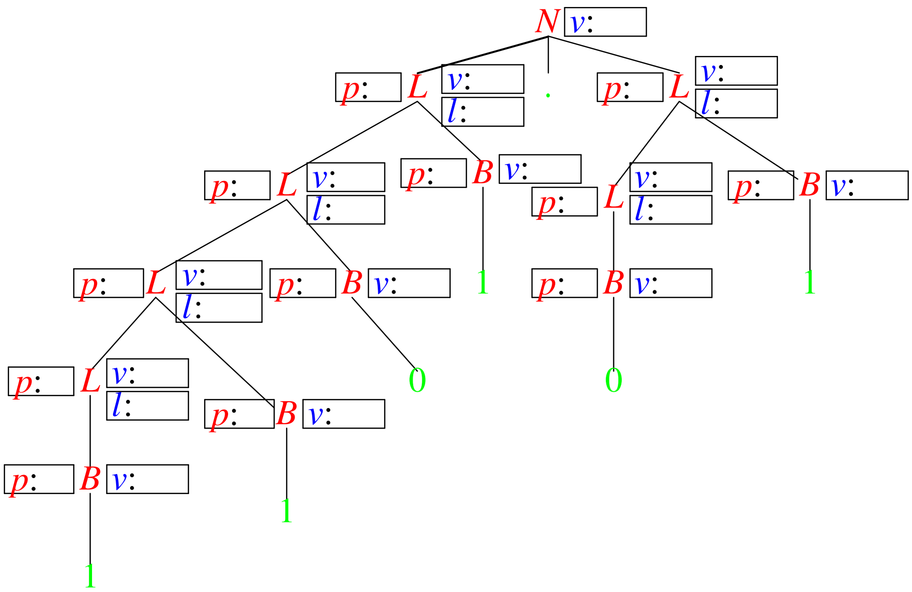


CB 3-1

Abbildung 3.1: Binärzahlen von Knuth

Attributierung von G_B mit synthetischen und inheriten Attributen:

$$
\begin{array}{llll}
G_B: & (1) \quad B \rightarrow 0 & & (2) \quad B \rightarrow 1 \\
 & (3) \quad L \rightarrow B & & (4) \quad L \rightarrow LB \\
 & (5) \quad N \rightarrow L & & (6) \quad N \rightarrow L.L
\end{array}
$$



CB 3-2

Abbildung 3.2: Attributierung von G_B mit synthetischen und inheriten Attributen

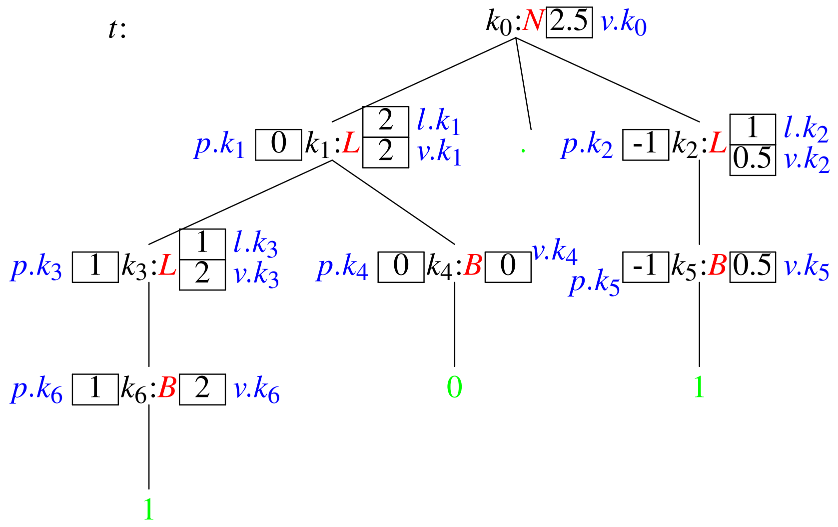
Attributgleichungen:

π			E_π		
B	$\rightarrow$	0	$v.0$	$=$	$\mathbf{0}$
B	$\rightarrow$	1	$v.0$	$=$	$\mathbf{2}{*}{*}(p.0)$
L	$\rightarrow$	B	$v.0$	$=$	$v.1$
			$l.0$	$=$	$\mathbf{1}$
			$p.1$	$=$	$p.0$
L	$\rightarrow$	LB	$v.0$	$=$	$+(v.1,v.2)$
			$l.0$	$=$	$+(l.1,\mathbf{1})$
			$p.1$	$=$	$+(p.0,\mathbf{1})$
			$p.2$	$=$	$p.0$
N	$\rightarrow$	L	$v.0$	$=$	$v.1$
			$p.1$	$=$	$\mathbf{0}$
N	$\rightarrow$	$L.L$	$v.0$	$=$	$+(v.1,v.3)$
			$p.1$	$=$	$\mathbf{0}$
			$p.3$	$=$	$-(l.3)$

CB 3-3

Abbildung 3.3: Attributgleichungen

Ein Attributgleichungssystem



E_t

k_0: $v.k_0 = +(v.k_1, v.k_2)$
$p.k_1 = \mathbf{0}$
$p.k_2 = -(l.k_2)$

k_1: $v.k_1 = +(v.k_3, v.k_4)$
$l.k_1 = +(l.k_2, \mathbf{1})$
$p.k_3 = +(p.k_1, \mathbf{1})$
$p.k_4 = p.k_1$

k_2: $v.k_2 = v.k_5$
$l.k_2 = \mathbf{1}$
$p.k_5 = p.k_2$

k_3: $v.k_3 = v.k_6$
$l.k_3 = \mathbf{1}$
$p.k_6 = p.k_3$

k_4: $v.k_4 = \mathbf{0}$

k_5: $v.k_5 = \mathbf{2} ** (p.k_5)$

k_6: $v.k_6 = \mathbf{2} ** (p.k_6)$

CB 3-4

Abbildung 3.4: Das Attributgleichungssystem

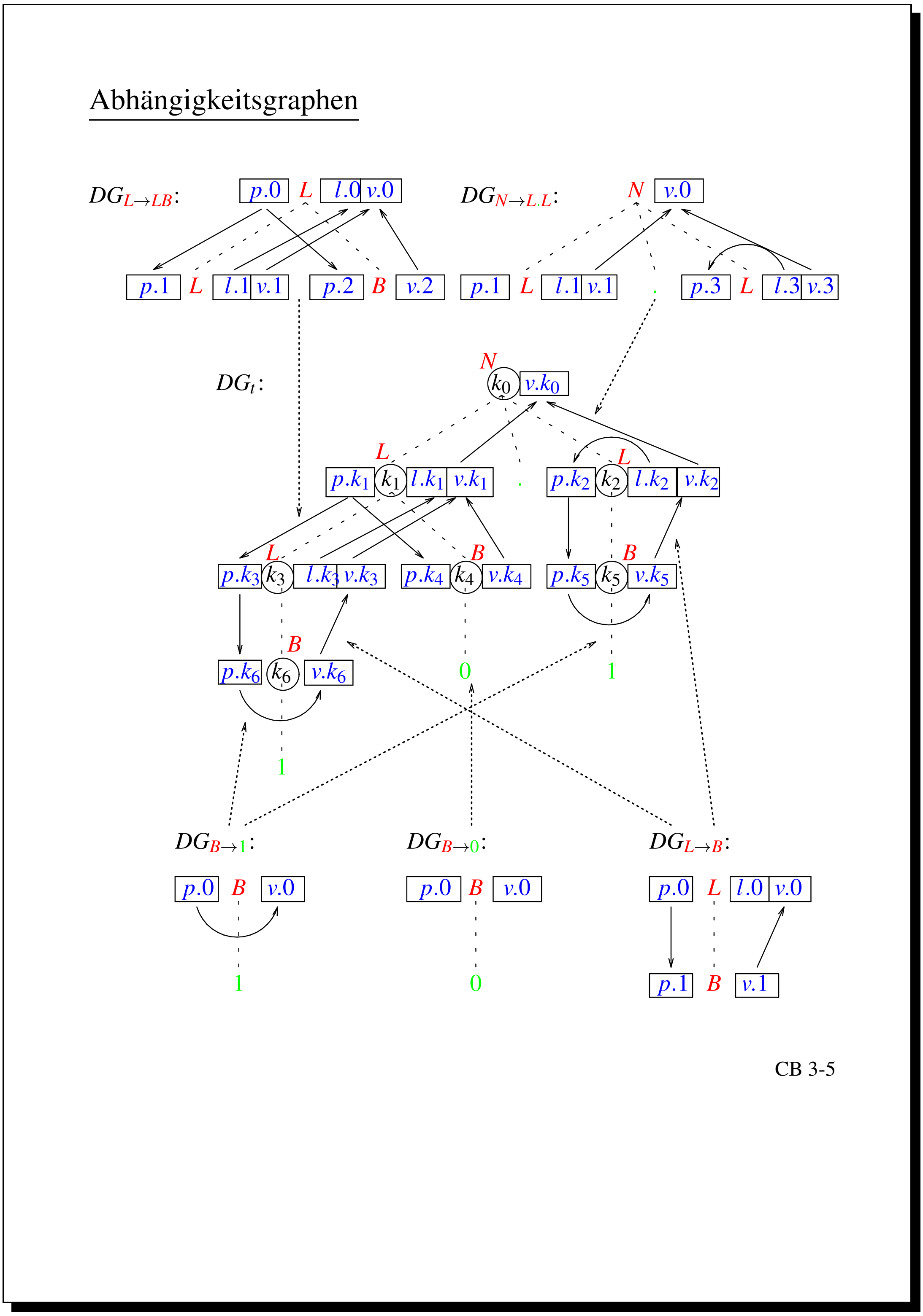
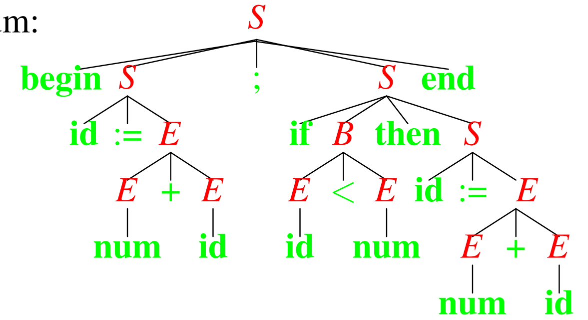


Abbildung 3.5: Abhängigkeitsgraphen

Konkrete und abstrakte Syntax (Beispiel)

$$
\begin{array}{llll}
G: & S & \rightarrow & \mathbf{id} \mathbf{:=} E \mid \mathbf{begin}\ S ; S\ \mathbf{end} \mid \mathbf{if}\ B\ \mathbf{then}\ S \\
 & B & \rightarrow & E < E \\
 & E & \rightarrow & E + E \mid \mathbf{id} \mid \mathbf{num}
\end{array}
$$

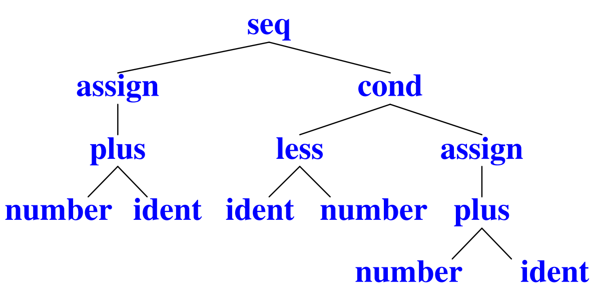
Ableitungsbaum:



Operationsalphabet:

$$
\begin{array}{ll}
\mathbf{assign}: E \rightarrow S & \mathbf{less}: E \times E \rightarrow B \\
\mathbf{seq}: S \times S \rightarrow S & \mathbf{plus}: E \times E \rightarrow E \\
\mathbf{cond}: B \times S \rightarrow S & \mathbf{ident}: \ \rightarrow E \\
\mathbf{number}: \ \rightarrow E &
\end{array}
$$

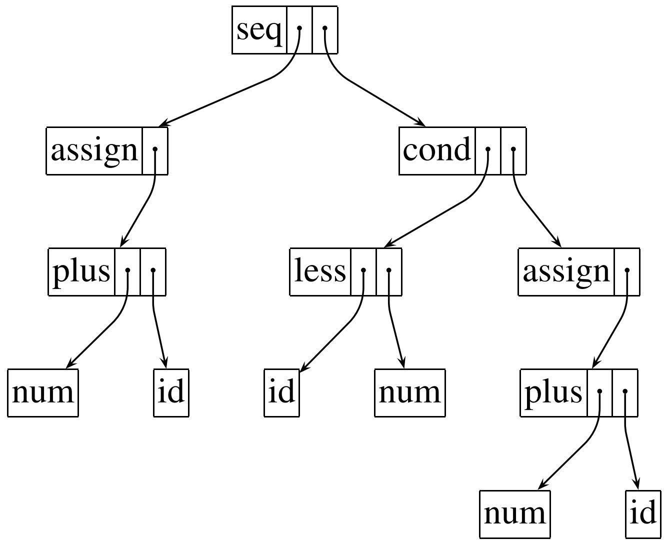
Abstrakter Syntaxbaum:



CB 3-6

Abbildung 3.6: konkrete und abstrakte Syntax

Beispiel: Graphrepräsentation



CB 3-7

Abbildung 3.7: Beispiel: Graphrepräsentation

COMPILERBAU 2004

AST–Erzeugung mit S–Attributierung

$S \rightarrow \texttt{id} \texttt{:=} E$	$a.0 = \texttt{mk-assign}(a.3)$
$S \rightarrow \texttt{begin}\ S\texttt{;}\ S\ \texttt{end}$	$a.0 = \texttt{mk-seq}(a.2, a.4)$
$S \rightarrow \texttt{if}\ B\ \texttt{then}\ S$	$a.0 = \texttt{mk-cond}(a.2, a.4)$
$B \rightarrow E < E$	$a.0 = \texttt{mk-less}(a.1, a.3)$
$E \rightarrow E + E$	$a.0 = \texttt{mk-plus}(a.1, a.3)$
$E \rightarrow \texttt{id}$	$a.0 = \texttt{mk-id}()$
$E \rightarrow \texttt{num}$	$a.0 = \texttt{mk-num}()$

S–Anteil der Typberechnung arith. Ausdrücke

$E \rightarrow E + T$	$t.0 = \max(t.1, t.3)$
$E \rightarrow T$	$t.0 = t.1$
$T \rightarrow T * F$	$t.0 = \max(t.1, t.3)$
$T \rightarrow F$	$t.0 = t.1$
$F \rightarrow \texttt{num}$	$t.0 = t.1$
$F \rightarrow \texttt{id}$	$t.0 = \mathrm{lookup}(env.0, id.1)$

CB 3-8

Abbildung 3.8: AST-Erzeugung und S-Anteil der Typberechnung

COMPILERBAU 2004

Syntaxanalyse mit Attributauswertung

Ein Beispiel:

Grammatik:

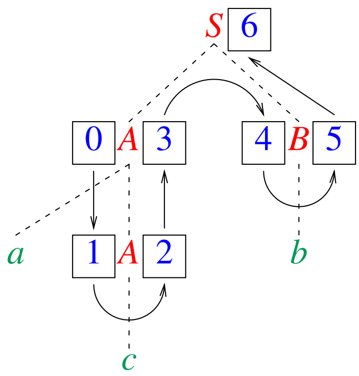
$S \longrightarrow AB$:
- $i.1 = zero()$
- $i.2 = suc(s.1)$
- $s.0 = suc(s.2)$

$A \longrightarrow aA$:
- $i.2 = suc(i.0)$
- $s.0 = suc(s.2)$

$A \longrightarrow c$:
- $s.0 = suc(i.0)$

$B \longrightarrow b$:
- $s.0 = suc(i.0)$

Ableitungsbaum:



CB 3-9

Abbildung 3.9: Syntaxanalyse mit Attributauswertung

COMPILERBAU 2004

Syntaxanalyse mit Attributauswertung

Berechnungsprotokoll (ohne Zustand und Eingabe):

$[\rightarrow \cdot S]$	

↓ "expand S"

$[\rightarrow \cdot S]$	
$[S \rightarrow \cdot AB]$	

↓ "expand A"

$[\rightarrow \cdot S]$	
$[S \rightarrow \cdot AB]$	
$[A \rightarrow \cdot aA]$	$i.0 = 0$

↓ "match a"

$[\rightarrow \cdot S]$	
$[S \rightarrow \cdot AB]$	
$[A \rightarrow a \cdot A]$	$i.0 = 0$

↓ "expand A"

$[\rightarrow \cdot S]$	
$[S \rightarrow \cdot AB]$	
$[A \rightarrow a \cdot A]$	$i.0 = 0$
$[A \rightarrow \cdot c]$	$i.0 = 1$

↓ "match c"

$[\rightarrow \cdot S]$	
$[S \rightarrow \cdot AB]$	
$[A \rightarrow a \cdot A]$	$i.0 = 0$
$[A \rightarrow c \cdot]$	$i.0 = 1$

↓ "reduce $A \rightarrow c$"

$[\rightarrow \cdot S]$	
$[S \rightarrow \cdot AB]$	
$[A \rightarrow aA \cdot]$	$i.0 = 0, s.2 = 2$

↓ "reduce $A \rightarrow aA$"

$[\rightarrow \cdot S]$	
$[S \rightarrow A \cdot B]$	$s.1 = 3$

↓ "expand B"

$[\rightarrow \cdot S]$	
$[S \rightarrow A \cdot B]$	$s.1 = 3$
$[B \rightarrow \cdot b]$	$i.0 = 4$

↓ "match b"

$[\rightarrow \cdot S]$	
$[S \rightarrow A \cdot B]$	$s.1 = 3$
$[B \rightarrow b \cdot]$	$i.0 = 4$

↓ "reduce $B \rightarrow b$"

$[\rightarrow \cdot S]$	
$[S \rightarrow AB \cdot]$	$s.1 = 3, s.2 = 5$

↓ "reduce $S \rightarrow AB$"

$[\rightarrow S \cdot]$	$s.0 = 6$

CB 3-10

Abbildung 3.10: Syntaxanalyse Fortsetzung

Kapitel 4

Übersetzung in Zwischencode

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

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

Vorteil dieser Zerlegung: Z ist maschinenunabhängig $\rightarrow$ Portabilität, Transparenz, Codeoptimierung.

Beispiel: Java Virtual Machine (JVM), P-Code von Pascal.

4.1 Übersetzung von Ausdrücken, Anweisungen, Blöcken und Prozeduren

Beispiel: Programmiersprache *BPS* („Pascal ohne Datenstrukturen und Prozedurparameter“)

- arithmetische und boolesche Ausdrücke mit strikter bzw. nicht-strikter Semantik
- Kontrollstrukturen: Sequenz, Verzweigung, Iteration
- Blöcke und Prozeduren: lokale und globale Variablen. Erfordert dynamische Speicherverwaltung mit Laufzeitkeller. Unterschiedliche Verwendung in Programmiersprachen:
  - FORTRAN: Unterprogramme, nicht geschachtelt, keine Rekursion
    $\rightsquigarrow$ statische Speicherverwaltung, Speicherbedarf zur Übersetzungszeit bekannt
  - C: rekursive Prozeduren, aber nicht geschachtelt
    $\rightsquigarrow$ dynamische Speicherverwaltung, Speicherbedarf erst zur Laufzeit bekannt, keine statischen Verweise auf dem Laufzeitkeller
  - ALGOL-Familie: ALGOL60, Pascal, Modula, geschachtelte rekursive Prozedurdeklarationen
    $\rightsquigarrow$ dynamische Speicherverwaltung, Laufzeitkeller mit statischen Verweisen
- keine Datenstrukten, keine Prozedurparameter
- Beschränkung auf Datentyp int

[09.07.2004]

4.1.1 Semantik von BPS

- Bezeichner einer Deklaration Δ müssen paarweise verschieden sein.
- Ein im Anweisungsteil Γ eines Blockes $\Delta\Gamma$ auftretender Bezeichner muß deklariert sein, und zwar in Δ oder in der Deklarationsliste eines $\Delta\Gamma$ umschließenden Blocks.
- Mehrfachdeklaration eines Bezeichners auf verschiedenen Niveaus möglich, die „innerste" Deklaration ist für ein Auftreten gültig.
- **Static Scope**: Beim Aufruf einer Prozedur ist ihre Deklarations-, und nicht ihre Aufruf-Umgebung gültig.

4.1.2 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) und Prozedurkeller $PK = \mathbb{Z}^*$ (Spitze links).

Zustand $s = (m, d, p) \in ZR$
mit Befehlsmarke $m \in \mathbb{N}$, DK-Zustand $d = d.r : \ldots : d.1$ und PK-Zustand $p = p.1 \ldots p.t$.

AM-Befehle:

- arithmetische Befehle: ADD, …
- logische Befehle: NOT, AND, OR, LT, …
- Sprungbefehle: JMP n, JFALSE n $\quad (n \in \mathbb{N})$
- Prozedurbefehle: CALL(ca, dif, loc) (mit $ca, dif, loc \in \mathbb{N}$), RET
- Transportbefehle: LOAD(dif, off), STORE(dif, off), mit $dif, off \in \mathbb{N}$ und LIT z mit $z \in \mathbb{Z}$

Befehlssemantik

$[\![B]\!] : ZR - \to ZR$ für jeden AM-Befehl B

$[\![ADD]\!](m, d : z_1 : z_2, p) := (m+1, d : (z_2 + z_1), p)$

$[\![LT]\!](m, d : z_1 : z_2, p) := (m+1, d : b, p)$ mit $b = \begin{cases} 1 & , z_1 < z_2 \\ 0 & , z_1 \geq z_2 \end{cases}$

$$\left.\begin{array}{lcl} [\![AND]\!](m, d : b_1 : b_2, p) & := & (m+1, d : b1 \wedge b_2, p) \\ [\![OR]\!](m, d : b_1 : b_2, p) & := & (m+1, d : b1 \vee b_2, p) \\ [\![NOT]\!](m, d : b, p) & := & (m+1, d : \neg b, p) \end{array}\right\} \text{mit } b, b_1, b_2 \in \{0, 1\}$$

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

$[\![JMPn]\!](m, d, p) := (n, d, p)$

Für die Semantik der Prozedur- und Transportbefehle besondere Struktur des Prozedurkellers p:

p zerfällt in Aktivierungsblöcke („Frames") der Form

$$sv : dv : ra : l_1 : \ldots : l_k$$

mit sv: statischer Verweis (zeigt auf den Aktivierungsblock der Deklarationsumgebung), dv: dynamischer Verweis (zeigt auf den letzten Aktivierungsblock), ra: Rücksprungadresse (Codeadresse) nach Beendigung des Prozeduraufrufs und l_i: lokale Variable eines Prozessaufrufs.

Berechnung des statischen Verweises

[13.07.2004]

sv liefert die Differenz zwischen Aufruf und Deklarationsniveau und damit die Länge der Verweiskette. Dazu

$$\text{Hilfsfunktion} \quad \underline{\text{base}} : PK \times \mathbb{N} - \to \mathbb{N}$$

bestimmt für einen Prozedurkeller bezüglich einer Niveaudifferenz den Beginn der Deklarationsumgebung (als absolute Adresse des aktuellen PK).

$\underline{base}(p, 0) := 1$
$\underline{base}(p, dif + 1) := \underline{base}(p, dif) + p.\underline{base}(p, dif)$

Beispiel: (Abbildung 4.7)

$$\begin{array}{rcccccl}
\underline{\text{base}}(p,2) & = & \underline{\text{base}}(p,1) & + & p.\underline{\text{base}}(p,1) & & \\
\underline{\text{base}}(p,1) & = & \underline{\text{base}}(p,0) & + & p.\underline{\text{base}}(p,0) & & \\
 & = & 1 & + & p.1 & = 6 & \\
\underline{\text{base}}(p,2) & = & 6 & + & p.6 & = 11 &
\end{array}$$

Es folgt: $\underbrace{s}_{\text{rel. Adr.}} = \underline{\text{base}}(p,2) + \underbrace{2}_{\text{lok. Var.}} + \underbrace{2}_{ra+dv} = 15$

- $CALL(ca, dif, loc)$
  mit $ca \in \mathbb{N}$ als Codeadresse, $dif \in \mathbb{N}$ als Niveaudifferenz und $loc \in \mathbb{N}$ für die Anzahl lokaler Variablen. Erzeugt neuen Aktivierungsblock und springt zum Code des Prozedurrumpfs.

$$[\![CALL(ca, dif, loc)]\!](m, d, p) :=$$
$$(ca, d, \underbrace{(base(p, dif) + loc + 2)}_{sv} : \underbrace{loc + 2}_{dv} : \underbrace{m + 1}_{ra} : \underbrace{0 : \ldots : 0}_{\text{lok. Var.}} : p)$$

- RET löscht den letzten Aktivierungsblock und kehrt zur Aufrufstelle zurück

$$[\![RET]\!](m, d, p.1 : \ldots : p.t) :=$$
$$\underline{\text{if}}\, t \geq 2 + p.2\, \underline{\text{then}}\, (p.3, d, p. \quad \underbrace{(2 + p.2)}_{\text{Start nächster Aktivierungsblock}} \quad : \ldots : p.t)$$

  PK wird stets mindestens einen Aktivierungsblock enthalten.

- $LOAD(dif, off)$ und $STORE(dif, off)$ laden und speichern zwischen.

  DK und PK: relative Adressierung mit

- Niveaudifferenz *dif* zwischen Auftreten und Deklaration
- Offset *off* als relative Adresse in Aktivierungsblock

Die Kette der statischen Verweise bestimmt die sichtbare Umgebung auf *PK*.

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

Definition: AM-Code: Befehlsfolgen mit aufsteigenden Befehlsmarken
$P \in \underline{\text{AM-Code}} \Leftrightarrow P = a_1 : B_1; \ldots; a_p : B_p$ mit $a_i \in \underline{\text{Adr}} := \mathbb{N}, a_i = a_1 + i - 1$ und B_i ein AM-Befehl ($1 \leq i \leq p$).

Semantik von P: Iteration der Befehle gemäß *BZ*

$$\begin{aligned}&\mathscr{I} : \underline{\text{AM}-\text{Code}} \times ZR - \to ZR\\ &\mathscr{I}[\![P]\!](m,d,p) := \underline{\text{if}}\ a_1 \leq m \leq a_p\ \underline{\text{then}}\\ &\qquad \mathscr{I}[\![P]\!](\underbrace{[\![B_m]\!](m,d,p)}_{(m',d',p')})\\ &\quad \underline{\text{else}}\,(m,d,p)\end{aligned}$$

[16.07.2004]

4.1.3 Übersetzung von BPS-Programmen in AM-Code

$\underline{\text{trans}} : \underline{\text{BPS-Prog}} \to \underline{\text{AM-Code}}$ (Zwischencode Z)

Hilfsmittel: Symboltabelle

$$\underline{\text{Tab}} := \{st | st : \underline{\text{Ide}} - \to (\{\underline{\text{const}}\} \times \mathbb{Z})$$
$$\cup (\{\underline{\text{var}}\} \times \underline{\text{Lev}} \times \underline{\text{Off}}) \cup (\{\underline{\text{proc}}\} \times \underline{\text{Adr}} \times \underline{\text{Lev}} \times \underline{\text{Size}})$$

Variablendeklaration: Deklarationsniveau $dl \in \underline{\text{Lev}}$, Offset $off \in \underline{\text{Off}} := \mathbb{N}$

Prozessdeklaration: Startadresse $ca \in \underline{\text{Adr}}$ das Prozedurcodes, Deklarationsniveau $dl \in \underline{\text{Lev}}$, Anzahl der lokalen Variablen $loc \in \underline{\text{Size}} := \mathbb{N}$

Aufbau der Symboltabelle

$$\underline{\text{up}} : \underline{\text{Decl}} \times \underline{\text{Tab}} \times \underline{\text{Adr}} \times \underline{\text{Lev}} - \to \underline{\text{Tab}}$$

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

$$\underline{\text{up}}(\Delta_c\Delta_v\Delta_p, st, a, l) := \underline{\text{if}}\ \text{diffid}(\Delta_c\Delta_v\Delta_p)\ \underline{\text{then}}$$
$$\underline{\text{up}}(\Delta_p, \underline{\text{up}}(\Delta_v, \underline{\text{up}}(\Delta_c, st, a, l), a, l), a, l)$$

- $\underline{\text{up}}(\varepsilon, st, a, l) := st$
- $\underline{\text{up}}(\underbrace{\underline{\text{const}}\, I_1 = Z_1, \ldots, I_n = Z_n}_{\Delta_c}; st, a, l) := st[I_1/(\underline{\text{const}}, Z_1), \ldots, I_n/(\underline{\text{const}}, Z_n)]$
- $\underline{\text{up}}(\underbrace{\underline{\text{var}}\, I_1, \ldots, I_n}_{\Delta_v}; st, a, l) := st[I_1/(\underline{\text{var}}, l, 1), \ldots I_n/(\underline{\text{var}}, l, n)]$
- $\underline{\text{up}}(\underbrace{\underline{\text{proc}}\, I_1, B_1; \ldots; I_n, B_n}_{\Delta_p}; st, a, l) :=$

  $st[I_1/(\underline{\text{proc}}, a_1, l, \underline{\text{size}}(B_1)), \ldots, I_n/(\underline{\text{proc}}, a_n, l, \underline{\text{size}}(B_n))]$

Dabei sind $a_1, \ldots, a_n$ symbolische „freie“ Adressen.

$\underline{\text{size}} : \underline{\text{Block}} \to \mathbb{N}$
ermittelt den Speicherbedarf als Anzahl der lokalen Variablen.

$$\underline{\text{size}}(\Delta_c \underline{\text{var}}\, I_1, \ldots, I_n; \Delta_p \Gamma) := n$$

Anfangstabelle

$P = \underline{\text{in/out}}\, I_1, \ldots, I_n, B.$ hat die Semantik $\mathscr{M}[\![P]\!] : \mathbb{Z}^n - \to \mathbb{Z}^n$. Für $(z_1, \ldots, z_n) \in \mathbb{Z}^n$ wählen wir den Anfangszustand

$$s = (1, \varepsilon, 0 : 0 : 0 : z_1 : z_2 : \ldots : z_n) \in ZR$$

mit 1 als Startadresse des AM-Codes von P, ε als leeren Datenkeller DK und $0 : 0 : 0 : z_1 : \ldots : z_n$ als I/O-Block auf dem Prozedurkeller PK. Die entsprechend initialisierte Anfangstabelle hat daher n Einträge $st_{I/O}(I_j) = (\underline{\text{var}}, 0, j)$ für $1 \leq j \leq n$.
Weitere Hilfsfunktionen zur Definition der Übersetzung $\underline{\text{trans}}$ sind die folgenden:

- $\underline{bt} : \underline{\text{Block}} \times \underline{\text{Tab}} \times \underline{\text{Adr}} \times \underline{\text{Lev}} - \to \underline{\text{AM} - \text{Code}}$
- $\underline{dt} : \underline{\text{Decl}} \times \underline{\text{Tab}} \times \underline{\text{Adr}} \times \underline{\text{Lev}} - \to \underline{\text{AM} - \text{Code}}$
- $\underline{ct} : \underline{\text{Cmd}} \times \underline{\text{Tab}} \times \underline{\text{Adr}} \times \underline{\text{Lev}} - \to \underline{\text{AM} - \text{Code}}$
- $\underline{et} : \underline{\text{AExp}} \times \underline{\text{Tab}} \times \underline{\text{Adr}} \times \underline{\text{Lev}} - \to \underline{\text{AM} - \text{Code}}$
- $\underline{sbt} : \underline{\text{BExp}} \times \underline{\text{Tab}} \times \underline{\text{Adr}} \times \underline{\text{Lev}} - \to \underline{\text{AM} - \text{Code}}$

Der Parameter $l \in \underline{\text{Lev}} := \mathbb{N}$ beschreibt die Blockschachtelungstiefe.

4.1.4 Die Übersetzung

Start der Übersetzung mit

$$\begin{aligned} \underline{\text{trans}}(\underline{\text{in/out}} I_1, \ldots, I_n; B.) \quad := \quad & 1 : \textbf{CALL}(a_\Gamma, 0, \underline{\text{size}}(B)); \\ & 2 : \textbf{JMP}\ 0; \\ & \underline{bt}(B, st_{I/O}, a_\Gamma, 1) \end{aligned}$$

a_Γ: Startadresse des Anweisungsteils Γ in $B = \Delta\Gamma$

Blockübersetzung

$$\begin{array}{rcl}
\underline{bt}(\Delta\Gamma, st, a, l) & := & \underline{dt}(\Delta, \underline{\text{up}}(\Delta, st, \overbrace{a_1, l), a_1}^{!}, l) \\
& & \underline{ct}(\Gamma, \underline{\text{up}}(\Delta, st, a_1, l), a, l) \\
& & a' : \mathbf{RET};
\end{array}$$

Dabei erzeugt $\underline{dt}$ Code für die Prozedurrümpfe von Δ, in der zugehörigen Symboltabelle sind nicht nur die Konstanten und Variablen von Δ, sondern auch die Prozedurbezeichner (Rekursion!) eingetragen. $\underline{ct}$ erhält die übergebene Startadresse und erzeugt Codes für einen Prozessrumpf, der mit einem Rücksprung endet.

Deklarationsübersetzung

$$\begin{array}{rcl}
\underline{dt}(\Delta_c\Delta_v\Delta_p, st, a, l) & := & \underline{dt}(\Delta_p, st, a, l) \\
\underline{dt}(\varepsilon, st, a, l) & := & \varepsilon \\
\underline{dt}(\underline{\text{proc}}\, I_1, B_1; \ldots; I_n, B_n;, st, a, l) & := & \underline{bt}(B_1, st, a_1, l+1) \\
& & \vdots \\
& & \underline{bt}(B_n, st, a_n, l+1)
\end{array}$$

[20.07.2004]

Beachte:

- $st(I_j) = (\underline{\text{proc}}, a_j, l+1, \mathit{off})$, weil $\underline{bt}$ die Funktionen $\underline{dt}$ und $\underline{\text{up}}$ mit den gleichen Anfangsparameter aufruft und beide Funktionen aus diesen in gleicher Weise die Adressen für die Prozedurrümpfe erzeugt.
- $\underline{bt}$ wird mit dem Level $l+1$ aufgerufen.

Anweisungsübersetzung

$$\begin{array}{rcl}
\underline{ct}(I := E, st, a, l) & := \underline{\text{if}} & st(I) = (\underline{\text{var}}, dl, \mathit{off})\ \underline{\text{then}} \\
& & \underline{et}(E, st, a, l) \\
& & a' : \mathbf{STORE}(l - dl, \mathit{off});
\end{array}$$

$$\begin{array}{rcl}
\underline{ct}(I(), st, a, l) & := \underline{\text{if}} & st(I) = (\underline{\text{proc}}, ca, dl, loc)\ \underline{\text{then}} \\
& & a : \mathbf{CALL}(ca, l - dl, loc);
\end{array}$$

$$\begin{array}{rcl}
\underline{ct}(\Gamma_1; \Gamma_2, st, a, l) & := & \underline{ct}(\Gamma_1, st, a, l) \\
& & \underline{ct}(\Gamma_2, st, a', l)
\end{array}$$

$$\begin{aligned}
\underline{ct}(\underline{\text{if}}\ BE\ \underline{\text{then}}\ \Gamma_1\ \underline{\text{else}}\ \Gamma_2, st, a, l) \quad := \quad & \underline{sbt}(BE, st, a, l) \\
& a' : \textbf{JFALSE}\ a''; \\
& \underline{ct}(\Gamma_1, st, a'+1, l) \\
& a''-1 : \textbf{JMP}\ a'''; \quad // \text{ hinter } \underline{ct}(\Gamma_2, \ldots) \\
& \underline{ct}(\Gamma_2, st, a'', l) \\
& a''' : \ldots
\end{aligned}$$

$$\begin{aligned}
\underline{ct}(\underline{\text{while}}\ BE\ \underline{\text{do}}\ \Gamma, st, a, l) \quad := \quad & \underline{sbt}(BE, st, a, l) \\
& a' : \textbf{JFALSE}\ a''+1 \\
& \underline{ct}(\Gamma, st, a'+1, l) \\
& a'' : \textbf{JMP}\ a;
\end{aligned}$$

$$\begin{aligned}
\underline{et}(z, st, a, l) \quad &:= \quad a : \mathit{LIT}\ z; \\
\underline{et}(I, st, a, l) \quad &:= \quad \underline{\text{if}}\ st(I) = (\underline{\text{const}}, z)\ \underline{\text{then}} \\
&\qquad\qquad a : \textbf{LIT}\ z; \\
&\qquad\ \ \underline{\text{if}}\ st(I) = (\underline{\text{var}}, dl, \mathit{off}) \\
&\qquad\qquad a : \textbf{LOAD}(l-dl, \mathit{off}); \\
\underline{et}(E_1+E_2, st, a, l) \quad &:= \quad \underline{et}(E_1, st, a, l) \\
&\qquad\ \ \underline{et}(E_2, st, a', l) \\
&\qquad\ \ a'' : \textbf{ADD};
\end{aligned}$$

Bemerkung: <u>et</u> erzeugt Stackcode, dessen Berechnung den Wert des Ausdruckes auf dem Datenkeller liefert.

$$\begin{aligned}
\underline{sbt}(E_1 < E_2, st, a, l) \quad &:= \quad \underline{et}(E_1, st, a, l) \\
&\qquad\ \ \underline{et}(E_2, st, a', l) \\
&\qquad\ \ a'' : \textbf{LT}; \\
\underline{sbt}(\underline{\text{not}}\ BE, st, a, l) \quad &:= \quad \underline{sbt}(BE, st, a, l) \\
&\qquad\ \ a' : \textbf{NOT}; \\
\underline{sbt}(BE_1\ \underline{\text{and}}\ BE_2, st, l) \quad &:= \quad \underline{sbt}(BE_1, st, a, l) \\
&\qquad\ \ \underline{sbt}(BE_2, st, a', l) \\
&\qquad\ \ a'' : \textbf{AND};
\end{aligned}$$

Strikter Stackcode für boolesche Ausdrücke: beide Argumente werden stets berechnet.

Satz: (Korrektheit der Übersetzung): Für jedes Programm $P \in \underline{Prog}^{(n)}$ und $(z_1,\ldots,z_n)$, $(z'_1,\ldots,z'_n) \in \mathbb{Z}^n$ gilt: $\mathscr{M}[\![P]\!](z_1,\ldots,z_n) = (z'_1,\ldots,z'_n) \curvearrowright$

$$\begin{aligned}\mathscr{I}[\![\underline{\text{trans}}(P)]\!] \quad & (1,\varepsilon,0:0:0:z_1:\ldots:z_n) = \\ & (0,\varepsilon,0:0:0:z'_1:\ldots:z'_n)\end{aligned}$$

Beweis: M Mohnen: A Compiler Correctness Proof for the Static Link Technique by means of Evolving Algebras (Fund. Inf. 29(1997)3,257-303).

4.1.5 Jumping Code für boolesche Ausdrücke

Übersetzung boolescher Ausdrücke mit nicht-strikter Semantik, durch Sprungbefehle anstelle logischer Befehle.

Idee: Vererbung von Sprungzielen für boolesche Ergebnisse

$$\underline{nbt} : \underline{\text{BExp}} \times \underline{\text{Tab}} \times \underline{\text{Adr}}^3 \times \underline{\text{Lev}} - \rightarrow \underline{\text{AM}-\text{Code}}$$

($\underline{\text{Adr}}$: 1. freie Anfangsadresse, 2. $\underline{\text{true}}$-Adresse, 3. $\underline{\text{false}}$-Adresse).

[23.07.2004] mit geeigneter Modifikation der Übersetzung von Verzweigung und Iteration.

$$
\begin{array}{rcl}
\underline{nbt}(E_1 < E_2, st, a, a_t, a_f, l) & := & \underline{et}(E_1, st, a, l) \\
 & & \underline{et}(E_2, st, a', l) \\
 & & a'' : \mathbf{LT}; \\
 & & a''+1 : \mathbf{JFALSE}\ a_f; \\
 & & a''+2 : \mathbf{JMP}\ a_t; \\
\underline{nbt}(\underline{\text{not}}\ BE, st, a, a_t, a_f, l) & := & \underline{nbt}(BE, st, a, a_f, a_t, l) \\
\underline{nbt}(BE_1\ \underline{\text{and}}\ BE_2, st, a, a_t, a_f, l) & := & \underline{nbt}(BE_1, st, a, a', a_f, l) \\
 & & \underline{nbt}(BE_2, st, a', a_t, a_f, l) \\
\underline{nbt}(BE_1\ \underline{\text{or}}\ BE_2, st, a, a_t, a_f, l) & := & \underline{nbt}(BE_1, st, a, a_t, a', l) \\
 & & \underline{nbt}(BE_2, st, a', a_t, a_f, l) \\
\underline{nct}(\underline{\text{if}}\ BE\ \underline{\text{then}}\ \Gamma_1\ \underline{\text{else}}\ \Gamma_2, st, a, l) & := & \underline{nbt}(BE, st, a, a_t, a_f, l) \\
 & & \underline{nct}(\Gamma_1, st, a_t, l) \\
 & & a' : \mathbf{JMP}\ a''; \qquad // \text{ Sprung ans Codeende} \\
 & & \underline{nct}(\Gamma_2, st, a_f, l) \\
 & & a'' : \ldots \\
\underline{nct}(\underline{\text{while}}\ BE\ \underline{\text{do}}\ \Gamma, st, a, l) & := & \underline{nbt}(BE, st, a, a_t, a_f, l) \\
 & & \underline{nct}(\Gamma, st, a_t, l) \\
 & & a_f - 1 : \mathbf{JMP}\ a; \\
 & & a_f : \ldots
\end{array}
$$

4.1.6 Prozeduren mit Parametern

(Hinweis: der folgende Teil wurde im Sommersemester 2004, „mehr oder weniger“, als nicht Prüfungsrelevant gekennzeichnet. Ein großer Teil wurde lediglich anhand der Folien (bis einschließlich Abbildung 4.24) vorgestellt. Abgesehen vom Block „Sukzessiven Aufbau eines Frames“, sowie einen Teil der statischen Datenstrukturen, basiert der Rest der Aufzeichnungen auf der Vorlesung im Wintersemester 2002/03.)

Erweiterung von BPS um Wert- und Variablenparameter.

Syntax (Idee): I, J, V

$$\underline{\text{Decl}} : \Delta_p := \varepsilon | \underline{\text{proc}}\, I(I_1, \ldots, I_p; \underline{\text{var}}\, J_1, \ldots, J_q), B;$$

(Hierbei sind I_n und J_n die formalen Wert- und Variablenparameter)

$$\underline{\text{Cmd}} : \Gamma := \ldots | I(E_1, \ldots, E_p, V_1, \ldots, V_q)$$

(mit E_n und V_n die aktuellen Parameterausdrücke)

<u>Semantik</u>:

- Prozessdeklaration: formale Parameter werden behandelt als in der Umgebung des Prozessrumpfes deklarierte Variablen (entsprechende Vererbung im Rumpf).
- Prozessaufruf: Variablenparameter durch Zeiger auf entsprechenden Speicherplatz aktualisieren, Wertparameter als lokale Variable (neuer Speicherplatz).

Zwischencode ZPP (Zwischencode mit Prozedurparametern): Aktivierungsblöcke: zusätzlicher Speicher für aktuelle Parameter.

Zustandsraum: linearer Speicher mit vier Registern:

- IC: instruction counter
- SP: stack pointer
- FP: frame pointer
- R: index register

Sukzessiver Aufbau eines Frames:

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

Die Punkte 1,2 und 3 erfolgen durch Code der aufrufenden Prozedur, die Punkte 4 und 5 durch Code der aufgerufenen Prozedur.

Befehlssatz ZPP:

- arithmetische, logische und Sprungsbefehle wie bisher: DK auf Stackspitze, SP jeweils setzen.
- anstelle von **CALL**(ca, dif, loc), **RET**, **LOAD**(dif, off) und **STORE**(dif, off) die folgenden neue Befehle:

$$
\begin{array}{ll}
\textbf{CALL}\ ca & SP \leftarrow SP-1, <SP> \leftarrow IC+1, IC \leftarrow ca \\
\textbf{RET}\ k & IC \leftarrow <SP>, SP \leftarrow SP+k+1 \\
\textbf{PUSH}\ z & SP \leftarrow SP-1, <SP> \leftarrow z \\
\left.\begin{array}{l}\textbf{PUSH}\ FP \\ \textbf{PUSH}\ <FP> \\ \textbf{PUSH}\ <R+2>\end{array}\right\} & \text{analog} \\
\textbf{POP}\ FP & FP \leftarrow <SP>, SP \leftarrow SP+1 \\
\textbf{POP}\ <n> & Sn \leftarrow <SP>, SP \leftarrow SP+1 \\
\textbf{SUB}\ SP, n & SP \leftarrow SP-n \\
\textbf{LOAD}\ FP, SP & FP \leftarrow SP \\
\textbf{LOAD}\ R, <n> & \\
\textbf{LOAD}\ R, <R+2> &
\end{array}
$$

(S_n bezeichne die Stackzelle mit der Adresse n).

Ein-/Ausgabe: $P = \underline{\text{in/out}} I_1, \ldots, I_n; B$. Die I/O-Parameter werden als Wertparameter (und nicht als lokale Variablen) gespeichert.

Zwischencodeerzeugung: Modifikation von $\underline{\text{trans}} : \underline{\text{BSP} - \text{Prog}} \to \underline{\text{AM} - \text{Code}}$ unter Berücksichtigung von

a. Prozedur mit Parametern

b. neue abstrakte Maschine (neuer Befehlssatz)

Erweiterung der Symboltabelle um Einträge für Variablenparameter, aber Wertparameter wie lokale Variablen behandeln.

$$\ldots \cup (\{\underline{\text{vpar}}\} \times \underline{\text{Lev}} \times \underline{\text{Off}})$$

positive und negative Offsets: $\underline{\text{Off}} := \mathbb{Z}$

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

Aufbau der Symboltabelle mit $\underline{\text{up}}$ wie oben, aber negativen Offsets bei lokalen Variablen.

Alternative zur Verweiskettentechnik: Display-Technik

Im Unterschied zur Verweiskettentechnik schnellerer Variablenzugriff, dadurch aber höherer Speicheraufwand.

a. lokale Displays: die aufgerufene Prozedur trägt alle statischen Verweise (*sv*) in den Frame ein.

b. globale Displays: die *sv* werden global als SLA (static link array) gespeichert.

[27.07.2004]

4.2 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 $\rightsquigarrow$ Adressberechnung.

statische Datenstrukturen: Speicherbedarf zur Übersetzungszeit bekannt.

dynamische Datenstrukturen: Speicherbedarf ist laufzeitabhängig.

Heap/Halde als zusätzliche Maschinenkomponente.
Zeigertypen: garbage collection/Speicherbereinigung

4.2.1 Statische Datenstrukturen

PSSD: Programmiersprache mit statischen Datenstrukturen (Felder/Arrays und Verbunde/Records)

Typsemantik: Ein Typ bezeichnet eine Menge.

Beispiel: $\mathscr{T}[\![int]\!] = \underline{\text{restr}}(\mathbb{Z})$, $\mathscr{T}[\![real]\!] = \underline{\text{restr}}\ (\mathbb{R})$, ...

Typkompatibilität: Polymorphie (Ad-hoc) von Operationssymbolen.

Beispiel: int $\times$ int $\rightarrow$ int, real $\times$ real $\rightarrow$ real, int $<$ real, ...

Zuweisungskompatibilität: starkes/schwaches Typkonzept.

Beispiel: : $type\ I_1 = T, I_2 = T$
$var\ V_1 : I_1; V_2 : I_2$
$V_1 := V_2?$

Beim starken Typkonzept: Bezeichner berücksichtigen! Vorteil: Kontrollmöglichkeit während der Übersetzung durch Typechecking, sorgt für Sicherheit von Softwaresystemen.

Beim schwachen Typkonzept: große Kompatibilität, beliebt für das Programmieren im Kleinen.

Erläuterung der Symboltabelle

- Basiswerte benötigen 1 Speicherplatz.
- $(\underline{\text{type}}, \underline{\text{array}}, \underbrace{Z_1, Z_2}_{\text{Grenzen}}, \underbrace{I}_{\text{Komponententyp}}, \underbrace{n}_{\text{Speicherbedarf}})$
- $(\underline{\text{type}}, \underline{\text{record}}, \underbrace{I_1}_{\text{Selektor}}, \underbrace{I_1'}_{\text{Komp.Typ}}, \underbrace{\sigma_1, \ldots, \sigma_n}_{\text{Offset}}, \underbrace{s}_{\text{Speicherbedarf}})$
- $st(I.I_j) = (I_j', \sigma_j)$ falls $st(I) = (\underline{\text{type}}, \underline{\text{record}}, \ldots, I_j, I_j', \sigma_j \ldots)$
- $(\underline{\text{var}}, \underbrace{I}_{\text{Typ}}, \underbrace{k}_{\text{Offset}})$

Aufbau einer Symboltabelle

$$\underline{\text{up}} : \underline{\text{Decl}} \times \underline{\text{Tab}} - \rightarrow \underline{\text{Tab}}$$

Explizite Typschachtelung mit neuen Bezeichnern auflösen. Der Einfachheit halber nehmen wir an, daß

$$\Delta = \Delta_C \Delta_T \Delta_V \in \underline{\text{Decl}}$$

entschachtelt ist, d.h.

i. Die Bezeichner sind paarweise verschieden.

ii. $\Delta_T = \underline{\text{type}}\, I_1 = T_1; \ldots; I_n = T_n \curvearrowright$

- $T_i \in \{\underline{\text{bool}}, \underline{\text{real}}, \underline{\text{int}}\}$
- $T_i = \underline{\text{array}}[Z_1 \ldots Z_n]$ of I_j mit $1 \leq j \leq i-1$
- $T_i = \underline{\text{record}} S_1 : I_{j_1}; \ldots \underline{\text{end}}$ mit $1 \leq j_1, \ldots j_n \leq i-1$

iii. $\Delta_V = \underline{\text{var}} V_1 : I_{j_1}; \ldots V_s : I_{j_s}; \curvearrowright 1 \leq j_1, \ldots j_s \leq n$

Für entschachtelte Deklarationen ist

- $\underline{\text{up}}(\Delta_C \Delta_T \Delta_V, st) := \underline{\text{up}}(\Delta_V, \underline{\text{up}}(\Delta_T, \underline{\text{up}}(\Delta_C, st)))$
- $\underline{\text{up}}(\varepsilon, st) = st$
- $\underline{\text{up}}(\underline{\text{const}}\, I_1 = C_1, \ldots, st) := st[I_1/(\underline{\text{const}}, C_1), \ldots]$
- $\underline{\text{up}}(\underline{\text{type}}\, I = \text{bool}; \text{drest}, st) := \underline{\text{up}}(\underline{\text{type}}\ \text{drest}, st[I/(\underline{\text{type}}, \underline{\text{bool}}, 1)]$
- $\underline{\text{up}}(\underline{\text{type}}\, I = J; \text{drest}, st) := \underline{\text{up}}(\underline{\text{type}}\ \text{drest}, st[I/st(J)])$
- $\underline{\text{up}}(\underline{\text{type}}\, I = \underline{\text{array}}[Z_1 \ldots Z_2]\, \text{of}\, J; \underline{\mathit{drest}}, st) :=$
  $\underline{\text{if}}\ st(J) = (\underline{\text{type}}, \ldots, n)\ \underline{\text{and}}\ k = Z_2 - Z_1 + 1 \in \mathbb{N}\ \underline{\text{then}}$
  $\underline{\text{up}}(\underline{\text{type}}\ \text{drest}, st[I/(\underline{\text{type}}, \underline{\text{array}}, Z_1, Z_2, J, k \cdot n)])$
- $\underline{\text{up}}(\underline{\text{type}}\, I = \underline{\text{record}}\, S_1 : J_1, \ldots, S_r : J_r, \text{drest}, st) :=$
  $\underline{\text{if}}\ st(J_i) = (\underline{\text{type}}, \ldots, n_i), 1 \leq i \leq r\ \underline{\text{then}}$
  $\underline{\text{up}}(\underline{\text{type}}\ \text{drest}, st[I/(\underline{\text{type}}, \underline{\text{record}}, S_1, J_1, 0, S_2, J_2, n_1, \ldots, S_r, J_r, \sum_{i=1}^{r-1} n_i, \sum_{i=1}^{r} n_i)])$
- $\underline{\text{up}}(\underline{\text{type}}, st) = st$
- $\underline{\text{up}}(\underline{\text{var}}\, I_1 : J_1, \ldots, I_n : J_n, st) :=$
  $\underline{\text{if}}\ st(J_i) = (\underline{\text{type}}, \ldots, n_i), 1 \leq i \leq r\ \underline{\text{then}}$
  $st[I_1/(\underline{\text{var}}, J_1, 1), I_2/(\underline{\text{var}}, J_2, 1 + n_1), \ldots, I_n/(\underline{\text{var}}, J_n, 1 + \sum_{i=1}^{n-1} n_i)]$

4.2.2 Dynamische Datenstrukturen

Variante Verbunde: Speicherbedarf für größte Variante vorgesehen. Es wird derselbe Bereich für verschiedene Varianten verwendet.

Dynamische Felder: Felder als formale Prozedurparameter. Variable Feldgröße (z.B. ISO-Pascal). Feldgrenzen nicht bei Deklaration, sondern erst bei Aufruf der Prozedur durch aktuelle Parameter festlegen.

Speicherreservierung: Bedarf zur Übersetzungszeit unbekannt, aber bei Prozedureintritt mit aktuellen Parametern bestimmbar.

Implementierung ohne Heap. *Idee:* indirekte Adressierung mit Hilfe eines Felddescriptors.

Zeiger (Pointer): dynamisch veränderbare Strukturen. Die Erzeugung von Objekten erfolgt nicht durch Deklaration mit statischer Adresse, sondern durch Anweisung, zum Beispiel <u>new</u>.

Zeigervariable: p: POINTER TO t;
Anweisung: new(p) zur Erzeugung von neuen Speicherplatz.

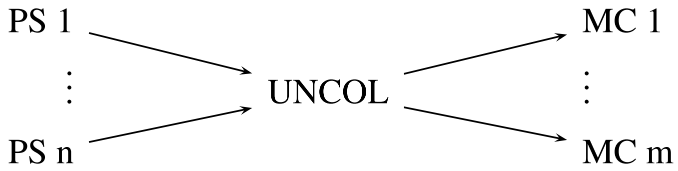
Die Speicherfreigabe erfolgt nicht automatisch, es werden spezielle Algorithmen für die Speicherbereinigung benötigt. Neuer Speicherbereich wird auf der Halde (heap) angelegt.

Semantik des new-Befehls:
$\underline{\text{if}}\ NP - <SP> \leq EP\ \underline{\text{then}}$ error(„stack overflow")
$\underline{\text{else}}\ NP := NP - <SP>;$
$\quad << SP - 1 >> := NP;$
$\quad SP := SP - 2;$

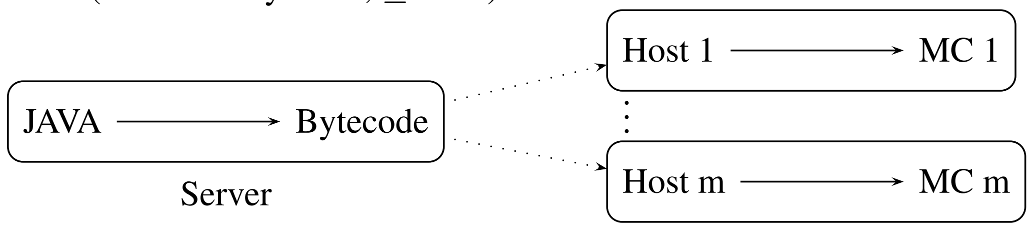
EP (extrem stack pointer): Oberschranke des Kellers zur Durchführung eines Prozeduraufrufs, bestimmt durch die Größe der Ausdrücke im Prozedurrumpf, und ggf. die Größe der dynamischen Felder. $\Rightarrow EP$ ist bei Prozedureintritt berechenbar.

$\underline{\text{code}}(new(p)) :=$ **LOAD** $adr(p)$;
LOAD $size(p)$;
NEW;

- Historisch (1960): **UNCOL** **Un**iversal **C**omputer **O**riented **L**anguage



- ACK (Amsterdam Compiler Kit) CommACM 26(1983)
- JAVA (Sun Microsystems; $\geq$ 1990)



CB 4-1

Abbildung 4.1: Historischer Überblick

Strukturen von PS (imperativ)

- Basistypen und –operationen
- statische und dynamische Datenstrukturen
- Kontrollstrukturen
- applikative und funktionale Konzepte: Ausdrücke, Prozeduren, Funktionen
- Modulstrukturen: Blöcke, Module, Klassen

CB 4-2

Abbildung 4.2: Strukturen von Programmiersprachen

Strukturen von MC (von–Neumann–Typ; SISD)

- Speicherhierarchie: Register, Cache, Hauptspeicher, Hintergrundspeicher
- Befehlsarten: Op–Befehle, Test– und Sprungbefehle, Transferbefehle, I/O
- Adressierung: relative Adr. mit Index– und Basisregister, indirekte Adr.

CB 4-3

Abbildung 4.3: Strukturen von Maschinencode

Strukturen von Z

- Typen und Operationen aus PS übernehmen
- Datenkeller mit Basisoperationen
- Sprungbefehle für Kontrollstrukturen
- Prozedurkeller für Blöcke
- Heap für dynamische Datenstrukturen

CB 4-4

Abbildung 4.4: Strukturen der Zwischenmaschine

Front–End–Aufgaben für $\underline{\mathsf{trans}}$: PS $\rightarrow$ Z

- Übersetzung von Ausdrücken und Kontrollstrukturen
- Übersetzung von Blöcken und Prozeduren
- Übersetzung von Datenstrukturen
- inkrementelle Übersetzung von Modulen

CB 4-5

Abbildung 4.5: Aufgaben des Front-ends

Syntax von BSP

Int :	Z		(* Z ist eine ganze Zahl *)
Ide :	I		(* I ist ein Bezeichner *)
Decl :	Δ	::=	$\Delta_C \Delta_V \Delta_P$
	Δ_C	::=	$\varepsilon \mid$ **const** $I_1 = Z_1, \ldots, I_n = Z_n;$
	Δ_V	::=	$\varepsilon \mid$ **var** $I_1, \ldots, I_n;$
	Δ_P	::=	$\varepsilon \mid$ **proc** $I_1;B_1;\ldots;$ **proc** $I_n;B_n;$
AExp :	E	::=	$Z \mid I \mid (E_1+E_2) \mid \ldots$
BExp :	BE	::=	$(E_1<E_2) \mid$ **not** $BE \mid (BE_1$ **and** $BE_2) \mid \ldots$
Cmd :	Γ	::=	$I := E \mid I() \mid$
			begin $\Gamma_1;\ldots;\Gamma_n$ **end** $\mid$
			if BE **then** Γ **else** $\Gamma \mid$ **while** BE **do** Γ
Block :	B	::=	$\Delta\Gamma$
Prog :	P	::=	**in/out** $I_1, \ldots, I_n;B.$ (* $n \geq 1$ *)

CB 4-6

Abbildung 4.6: Syntax von BSP

Beispiel zum “static scope”–Prinzip

in/out X;
    const $C = 10$;
    var Y;
    proc A;
        var Y, Z;
        proc B;
            var X, Z;
            $[\ldots A() \ldots]$
        $[\ldots B() \ldots D() \ldots]$ (1)
    proc D;
        $[\ldots A() \ldots]$
    $[\ldots A() \ldots]$.

(1) Im Rumpf von $A()$ bezeichnet X nicht X, sondern X. Z nicht Z, sondern Z.
Außerdem ist D hier gültig.

CB 4-7

Abbildung 4.7: „static-scope“

<u>Semantische Bereiche</u>

$$
\begin{array}{rcll}
\mathbb{Z} & := & \{0, 1, -1, \ldots\} & \text{ganze Zahlen} \\
\mathbb{B} & := & \{\mathbf{true}, \mathbf{false}\} & \text{Boolesche Werte} \\
\mathbf{Loc} & := & \{\alpha_1, \alpha_2, \ldots\} & \text{Speicherplätze (Locations)} \\
S & := & \{\sigma \mid \sigma : \mathbf{Loc} \dashrightarrow \mathbb{Z}\} & \text{Zustände (States)} \\
C & := & \{\theta \mid \theta : S \dashrightarrow S\} & \text{Zustandstransformationen (Continuations)} \\
U & := & \{\rho \mid \rho : \mathbf{Ide} \dashrightarrow \mathbb{Z} \cup \mathbf{Loc} \cup C\} & \text{Umgebungen (Environments)}
\end{array}
$$

CB 4-8

Abbildung 4.8: Semantische Bereiche

Ausdruckssemantik:

a) $\mathcal{E}_a : \mathbf{AExp} \times U \times S \dashrightarrow \mathbb{Z}$

$$
\begin{array}{lcl}
\mathcal{E}_a[\![Z]\!]\rho\sigma & := & Z \\
\mathcal{E}_a[\![I]\!]\rho\sigma & := & \begin{cases} Z & \text{, falls } \rho(I) = Z \in \mathbb{Z} \\ \sigma(\alpha) & \text{, falls } \rho(I) = \alpha \in \mathbf{Loc} \end{cases} \\
\mathcal{E}_a[\![E_1 + E_2]\!]\rho\sigma & := & \mathcal{E}_a[\![E_1]\!]\rho\sigma + \mathcal{E}_a[\![E_2]\!]\rho\sigma
\end{array}
$$

b) $\mathcal{E}_b : \mathbf{BExp} \times U \times S \dashrightarrow \mathbb{B}$

$$
\begin{array}{lcl}
\mathcal{E}_b[\![E_1 < E_2]\!]\rho\sigma & := & \mathcal{E}_b[\![E_1]\!]\rho\sigma < \mathcal{E}_b[\![E_2]\!]\rho\sigma \\
\mathcal{E}_b[\![\mathbf{not}\ BE]\!]\rho\sigma & := & \neg\mathcal{E}_b[\![BE]\!]\rho\sigma \\
\mathcal{E}_b[\![BE_1 \overset{and}{\underset{or}{}} BE_2]\!]\rho\sigma & := & \mathcal{E}_b[\![BE_1]\!]\rho\sigma \overset{\wedge}{\underset{\vee}{}} \mathcal{E}_b[\![BE_2]\!]\rho\sigma
\end{array}
$$

wobei $\wedge, \vee : \mathbb{B}^2_\perp \longrightarrow \mathbb{B}_\perp$ ($\perp$ = undefiniert)

i) strikt $\perp \overset{\wedge}{\underset{\vee}{}} b \;=\; b \overset{\wedge}{\underset{\vee}{}} \perp \;=\; \perp$

ii) nicht-strikt $\mathbf{false} \wedge \perp = \mathbf{false}$

$\mathbf{true} \vee \perp = \mathbf{true}$

sonst wie i)

CB 4-9

Abbildung 4.9: Ausdruckssemantik

Anweisungssemantik: $\mathcal{C} : \mathbf{Cmd} \times U \times S \dashrightarrow S$

$$\mathcal{C}[\![I := E]\!]\rho\sigma := \sigma[\alpha/\mathcal{E}[\![E]\!]\rho\sigma], \text{ falls } \rho(I) = \alpha \in \mathbf{Loc}$$

$$\mathcal{C}[\![\Gamma_1;\Gamma_2]\!]\rho\sigma := \mathcal{C}[\![\Gamma_2]\!]\rho(\mathcal{C}[\![\Gamma_1]\!]\rho\sigma)$$

$$\mathcal{C}[\![\mathbf{if}\ BE\ \mathbf{then}\ \Gamma]\!]\rho\sigma := \begin{cases} \mathcal{C}[\![\Gamma]\!]\rho\sigma & \text{, falls } \mathcal{E}_b[\![BE]\!]\rho\sigma = \mathbf{true} \\ \sigma & \text{, falls } \mathcal{E}_b[\![BE]\!]\rho\sigma = \mathbf{false} \end{cases}$$

$$\mathcal{C}[\![\underbrace{\mathbf{while}\ BE\ \mathbf{do}\ \Gamma}_{\Gamma'}]\!]\rho\sigma := \begin{cases} \mathcal{C}[\![\Gamma']\!]\rho(\mathcal{C}[\![\Gamma]\!]\rho\sigma) & \text{, falls } \mathcal{E}_b[\![BE]\!]\rho\sigma = \mathbf{true} \\ \sigma & \text{, falls } \mathcal{E}_b[\![BE]\!]\rho\sigma = \mathbf{false} \end{cases}$$

$$\mathcal{C}[\![I()]\!]\rho\sigma := \theta(\sigma), \text{ falls } \rho(I) = \theta \in C$$

CB 4-10

Abbildung 4.10: Anweisungssemantik

Deklarationssemantik: $\mathcal{D} : \textbf{Decl} \times U \times S \dashrightarrow U \times S$

$$\mathcal{D}[\![\Delta_C \Delta_V \Delta_P]\!]\rho\sigma \quad := \quad \mathcal{D}[\![\Delta_P]\!](\mathcal{D}[\![\Delta_V]\!](\mathcal{D}[\![\Delta_C]\!]\rho\sigma))$$
$$\text{, falls } \textbf{diff–id}(\Delta_C \Delta_V \Delta_P)$$

$$\mathcal{D}[\![\varepsilon]\!]\rho\sigma \quad := \quad \rho\sigma$$

$$\mathcal{D}[\![\textbf{const } I_1 = Z_1, \ldots, I_n = Z_n;]\!]\rho\sigma := \rho[I_1/Z_1, \ldots, I_n/Z_n]\sigma$$

$$\mathcal{D}[\![\textbf{var } I_1, \ldots, I_n;]\!]\rho\sigma :=$$
$$\rho[I_1/\alpha_{l+1}, \ldots, I_n/\alpha_{l+n}]\sigma[\alpha_{l+1}/0, \ldots, \alpha_{l+n}/0]$$

mit l = höchster Index eines belegten Speicherplatzes in σ

$$\mathcal{D}[\![\textbf{proc } I_1;B_1;\ldots;\textbf{proc } I_n;B_n;]\!]\rho\sigma :=$$
$$\rho[I_1/\theta_1, \ldots, I_n/\theta_n]\sigma$$
$$\text{mit } \theta_i := \mathcal{B}[\![B_i]\!]\rho[I_1/\theta_1, \ldots, I_n/\theta_n] \text{ für } 1 \le i \le n$$

Blocksemantik: $\mathcal{B} : \textbf{Block} \times U \times S \dashrightarrow S$

$$\mathcal{B}[\![\Delta\Gamma]\!]\rho\sigma := \mathcal{C}[\![\Gamma]\!](\mathcal{D}[\![\Delta]\!]\rho\sigma)$$

Programmsemantik: $\mathcal{M} : \textbf{Prog}^{(n)} \times \mathbb{Z}^n \dashrightarrow \mathbb{Z}^n$

$$\mathcal{M}[\![\textbf{in/out } I_1, \ldots, I_n;B.]\!](z_1, \ldots, z_n) := (\sigma(\alpha_1), \ldots, \sigma(\alpha_n))$$

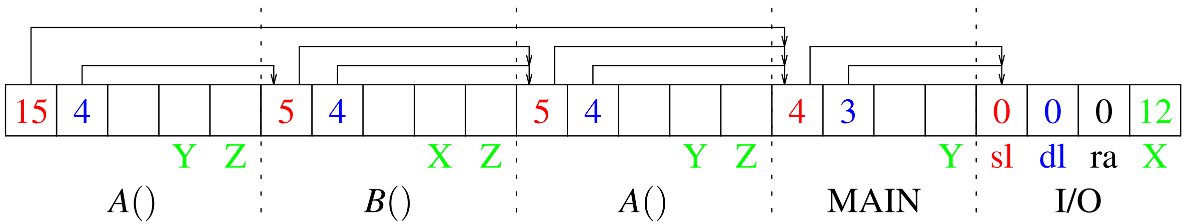
mit $\sigma := \mathcal{B}[\![B]\!]\rho_\emptyset[I_1/\alpha_1, \ldots, I_n/\alpha_n]\sigma_\emptyset[\alpha_1/z_1, \ldots, \alpha_n/z_n]$

CB 4-11

Abbildung 4.11: Deklarationssemantik

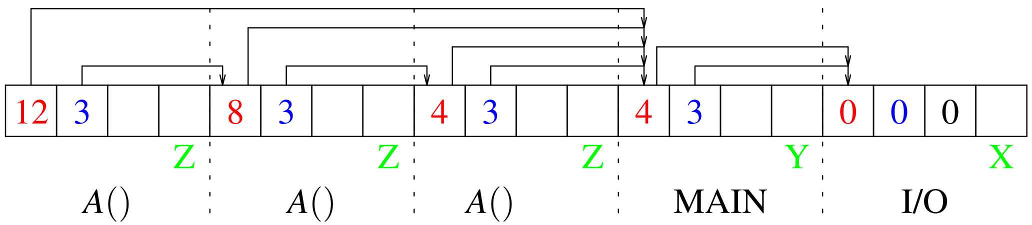
Arbeitsweise des Laufzeitkellers

Bsp. 1: Programm von Folie CB 4-7, Zustand nach dem 2. Aufruf von A:



15 verweist auf die Deklarationsumgebung von A

Bsp. 2: Rekursion: **in/out** X;**var** Y;
proc A;**var** Z;
$[\ldots A()\ldots]$
$[\ldots A()\ldots]$



CB 4-12

Abbildung 4.12: Arbeitsweise des Laufzeitkellers

$$
\begin{aligned}
\mathbf{up}(\Delta_C\Delta_V\Delta_P, st, a, l) &:= \mathbf{up}(\Delta_P, \mathbf{up}(\Delta_V, \mathbf{up}(\Delta_C, st, a, l), a, l), a, l)\\
&\ \text{falls } \mathbf{diff\text{–}id}(\Delta_C\Delta_V\Delta_P)\\
\mathbf{up}(\varepsilon, st, a, l) &:= st\\
\mathbf{up}(\mathbf{const}\ I_1 = Z_1, \ldots, I_n = Z_n;, st, a, l) &:= st[I_1/(\mathbf{const}, Z_1), \ldots, I_n/(\mathbf{const}, Z_n)]\\
\mathbf{up}(\mathbf{var}\ I_1, \ldots, I_n;, st, a, l) &:= st[I_1/(\mathbf{var}, l, 1), \ldots, I_n/(\mathbf{var}, l, n)]\\
\mathbf{up}(\mathbf{proc}\ I_1; B_1; \ldots; \mathbf{proc}\ I_n; B_n;, st, a, l) &:= st[I_1/(\mathbf{proc}, a.1, l, \mathbf{size}(B_1)), \ldots,\\
&\ \ I_n/(\mathbf{proc}, a.n, l, \mathbf{size}(B_n))]
\end{aligned}
$$

$$
\begin{aligned}
\mathbf{dt}(\Delta_c, \Delta_v, \Delta_p, st, a, l) &:= \mathbf{dt}(\Delta_p, st, a, l)\\
\mathbf{dt}(\varepsilon, st, a, l) &:= \varepsilon\\
\mathbf{dt}(\mathbf{proc}\ I_1; B_1; \ldots; \mathbf{proc}\ I_n; B_n;\ st, a, l) &:= \mathbf{bt}(B_1, st, a.1, l+1)\ \ldots\\
&\ \mathbf{bt}(B_n, st, a.n, l+1)
\end{aligned}
$$

$$
\begin{aligned}
\mathbf{bt}(\Delta\Gamma, st, a, l) &:= \mathbf{dt}(\Delta, \mathbf{up}(\Delta, st, a.1, l), a.1, l)\\
&\ \mathbf{ct}(\Gamma, \mathbf{up}(\Delta, st, a.1, l), a.1, l)
\end{aligned}
$$

CB 4-13

Abbildung 4.13: Semantik

Beispiel zur Übersetzung von BSP in ZP–Code

$\mathbf{in/out}\ X;$
$\quad\mathbf{var}\ E;$
$\quad\mathbf{proc}\ F;$
$\qquad\mathbf{if}\ 1 < X\ \mathbf{then\ begin}$
$\qquad\quad E := E * X;\qquad B_F\qquad \tilde{\Delta}\qquad \Delta$
$\qquad\quad X := X - 1;\qquad \|$
$\qquad\quad F()\qquad \Gamma_F$
$\qquad\mathbf{end};$
$\qquad\mathbf{begin}$
$\qquad\quad E := 1;$
$\qquad\quad F();\qquad \Gamma$
$\qquad\quad X := E$
$\qquad\mathbf{end}.$

$$\begin{aligned}\mathbf{trans}(\mathbf{in/out}\ X;\Delta\Gamma) \quad = \quad & 1 : \mathrm{CALL}(a_\Gamma, 0, 1);\\ & 2 : \mathrm{JMP}(0);\\ & \mathbf{bt}(\Delta\Gamma, st_{I/O}, a_\Gamma, 1)\\ & \text{mit } st_{I/O}(X) = (\mathbf{var}, 0, 1)\end{aligned}$$

CB 4-14

Abbildung 4.14: Beispiel: Übersetzung BSP in ZP

$$
\begin{aligned}
\mathbf{bt}(\Delta\Gamma, st_{I/O}, a_\Gamma, 1) \quad &= \quad \mathbf{dt}(\Delta, \mathbf{up}(\Delta, st_{I/O}, a_1, 1), a_1, 1) \\
& \qquad \mathbf{ct}(\Gamma, \mathbf{up}(\Delta, st_{I/O}, a_1, 1), a_\Gamma, 1) \\
& \qquad a_2 : \mathrm{RET};
\end{aligned}
$$

$$
\mathbf{up}(\Delta, st_{I/O}, a_1, 1) = \underbrace{st_{I/O}[E/(\mathbf{var}, 1, 1), F/(\mathbf{proc}, a_{11}, 1, 0)]}_{\overline{st}}
$$

$$
\begin{aligned}
\mathbf{dt}(\Delta, \overline{st}, a_1, 1) \quad &= \quad \mathbf{bt}(B_F, \overline{st}, a_{11}, 2) \\
&= \quad \mathbf{ct}(\Gamma_F, \overline{st}, a_{11}2, 2) \\
& \qquad a_3 : \mathrm{RET}; \\
\mathbf{ct}(\Gamma, \overline{st}, a_\Gamma, 1) \quad &= \quad a_\Gamma : \mathrm{LIT}(1); \\
& \qquad \mathrm{STO}(0, 1); \\
& \qquad \mathrm{CALL}(a_{11}, 0, 0); \\
& \qquad \mathrm{LOAD}(0, 1); \\
& \qquad \mathrm{STO}(1, 1); \\
\mathbf{ct}(\Gamma_F, \overline{st}, a_{11}, 2) \quad &= \quad \mathbf{sbt}(1 < x, \overline{st}, a_{11}, 2) \\
& \qquad a_4 : \mathrm{JFALSE}(a_5); \\
& \qquad \mathbf{ct}(\mathbf{begin} \ldots \mathbf{end}, \overline{st}, a_4 + 1, 2) \\
& \qquad a_5 : \\
\mathbf{sbt}(1 < x, \overline{st}, a_{11}, 2) \quad &= \quad a_{11} : \mathrm{LIT}(1); \\
& \qquad \mathrm{LOAD}(2, 1); \\
& \qquad \mathrm{LT}; \\
\mathbf{ct}(\mathbf{begin} \ldots \mathbf{end}, \overline{st}, a_4 + 1, 2) \quad &= \quad \mathbf{ct}(E := E * X, \overline{st}, a_4 + 1, 2) \\
& \qquad \mathbf{ct}(X := X - 1, \overline{st}, a_6, 2) \\
& \qquad \mathbf{ct}(F(), \overline{st}, a_7, 2) \\
&= \quad a_4 + 1 : \mathrm{LOAD}(1, 1); \\
& \qquad \mathrm{LOAD}(2, 1); \\
& \qquad \mathrm{MULT}; \\
& \qquad \mathrm{STO}(1, 1); \\
& \qquad \mathrm{LOAD}(2, 1); \\
& \qquad \mathrm{LIT}(1); \\
& \qquad \mathrm{SUB}; \\
& \qquad \mathrm{STO}(2, 1); \\
& \qquad \mathrm{CALL}(a_{11}, 1, 0)
\end{aligned}
$$

CB 4-15

Abbildung 4.15: Beispiel Fortsetzung

Ergebnis der Übersetzung:

$$\mathbf{trans}(\mathbf{in}/\mathbf{out}\ X;\Delta\Gamma) =$$

$$\begin{array}{rll}
1: & \mathrm{CALL}(a_\Gamma,0,1); & \text{Also:} \\
2: & \mathrm{JMP}(0); & \\
a_{11}: & \mathrm{LIT}(1); & a_{11} = 3 \\
 & \mathrm{LOAD}(2,1); & \\
 & \mathrm{LT}; & \\
a_4: & \mathrm{JFALSE}(a_5); & a_4 = 6 \\
 & \mathrm{LOAD}(1,1); & \\
 & \mathrm{LOAD}(2,1); & \\
 & \mathrm{MULT}; & \\
 & \mathrm{STO}(1,1); & \\
 & \mathrm{LOAD}(2,1); & \\
 & \mathrm{LIT}(1); & \\
 & \mathrm{SUB}; & \\
 & \mathrm{STO}(2,1); & \\
 & \mathrm{CALL}(a_{11},1,0); & \\
a_3: a_5: & \mathrm{RET}; & a_3 = 16 = a_5 \\
a_\Gamma: & \mathrm{LIT}(1); & a_\Gamma = 17 \\
 & \mathrm{STO}(0,1); & \\
 & \mathrm{CALL}(a_{11},0,0); & \\
 & \mathrm{LOAD}(0,1); & \\
 & \mathrm{STO}(1,1); & \\
a_2: & \mathrm{RET}; & a_2 = 22
\end{array}$$

CB 4-16

Abbildung 4.16: Ergebnis der Berechnung des Beispiels

Berechnungsprotokoll für $X = 3$:

$m \in BZ$	$d \in DK$	$p \in PK$
1	ε	0 : 0 : 0 : 3
17	ε	4 : 3 : 2 : 0 : 0 : 0 : 0 : 3
18	1	4 : 3 : 2 : 0 : 0 : 0 : 0 : 3
19	ε	4 : 3 : 2 : 1 : 0 : 0 : 0 : 3
3	ε	3 : 2 : 20 : 4 : 3 : 2 : 1 : 0 : 0 : 0 : 3
4	1	3 : 2 : 20 : 4 : 3 : 2 : 1 : 0 : 0 : 0 : 3
5	1 : 3	3 : 2 : 20 : 4 : 3 : 2 : 1 : 0 : 0 : 0 : 3
6	1	3 : 2 : 20 : 4 : 3 : 2 : 1 : 0 : 0 : 0 : 3
7	ε	3 : 2 : 20 : 4 : 3 : 2 : 1 : 0 : 0 : 0 : 3
8	1	3 : 2 : 20 : 4 : 3 : 2 : 1 : 0 : 0 : 0 : 3
9	1 : 3	3 : 2 : 20 : 4 : 3 : 2 : 1 : 0 : 0 : 0 : 3
10	3	3 : 2 : 20 : 4 : 3 : 2 : 1 : 0 : 0 : 0 : 3
11	ε	3 : 2 : 20 : 4 : 3 : 2 : 3 : 0 : 0 : 0 : 3
12	3	3 : 2 : 20 : 4 : 3 : 2 : 3 : 0 : 0 : 0 : 3
13	3 : 1	3 : 2 : 20 : 4 : 3 : 2 : 3 : 0 : 0 : 0 : 3
14	2	3 : 2 : 20 : 4 : 3 : 2 : 3 : 0 : 0 : 0 : 3
15	ε	3 : 2 : 20 : 4 : 3 : 2 : 3 : 0 : 0 : 0 : 2
3	ε	6 : 2 : 16 : 3 : 2 : 20 : 4 : 3 : 2 : 3 : 0 : 0 : 0 : 2
4	1	6 : 2 : 16 : 3 : 2 : 20 : 4 : 3 : 2 : 3 : 0 : 0 : 0 : 2
5	1 : 2	6 : 2 : 16 : 3 : 2 : 20 : 4 : 3 : 2 : 3 : 0 : 0 : 0 : 2
6	1	6 : 2 : 16 : 3 : 2 : 20 : 4 : 3 : 2 : 3 : 0 : 0 : 0 : 2
7	ε	6 : 2 : 16 : 3 : 2 : 20 : 4 : 3 : 2 : 3 : 0 : 0 : 0 : 2
8	3	6 : 2 : 16 : 3 : 2 : 20 : 4 : 3 : 2 : 3 : 0 : 0 : 0 : 2
9	3 : 2	6 : 2 : 16 : 3 : 2 : 20 : 4 : 3 : 2 : 3 : 0 : 0 : 0 : 2
10	6	6 : 2 : 16 : 3 : 2 : 20 : 4 : 3 : 2 : 3 : 0 : 0 : 0 : 2
11	ε	6 : 2 : 16 : 3 : 2 : 20 : 4 : 3 : 2 : 6 : 0 : 0 : 0 : 2
12	2	6 : 2 : 16 : 3 : 2 : 20 : 4 : 3 : 2 : 6 : 0 : 0 : 0 : 2
13	2 : 1	6 : 2 : 16 : 3 : 2 : 20 : 4 : 3 : 2 : 6 : 0 : 0 : 0 : 2
14	1	6 : 2 : 16 : 3 : 2 : 20 : 4 : 3 : 2 : 6 : 0 : 0 : 0 : 2
15	ε	6 : 2 : 16 : 3 : 2 : 20 : 4 : 3 : 2 : 6 : 0 : 0 : 0 : 1
3	ε	9 : 2 : 16 : 6 : 2 : 16 : 3 : 2 : 20 : 4 : 3 : 2 : 6 : 0 : 0 : 0 : 1
4	1	9 : 2 : 16 : 6 : 2 : 16 : 3 : 2 : 20 : 4 : 3 : 2 : 6 : 0 : 0 : 0 : 1
5	1 : 1	9 : 2 : 16 : 6 : 2 : 16 : 3 : 2 : 20 : 4 : 3 : 2 : 6 : 0 : 0 : 0 : 1
6	0	9 : 2 : 16 : 6 : 2 : 16 : 3 : 2 : 20 : 4 : 3 : 2 : 6 : 0 : 0 : 0 : 1
16	ε	9 : 2 : 16 : 6 : 2 : 16 : 3 : 2 : 20 : 4 : 3 : 2 : 6 : 0 : 0 : 0 : 1
16	ε	6 : 2 : 16 : 3 : 2 : 20 : 4 : 3 : 2 : 6 : 0 : 0 : 0 : 1
16	ε	3 : 2 : 20 : 4 : 3 : 2 : 6 : 0 : 0 : 0 : 1
20	ε	4 : 3 : 2 : 6 : 0 : 0 : 0 : 1
21	6	4 : 3 : 2 : 6 : 0 : 0 : 0 : 1
22	ε	4 : 3 : 2 : 6 : 0 : 0 : 0 : 6
0	ε	0 : 0 : 0 : 6

CB 4-17

Abbildung 4.17: Berechnungsprotokoll des Beispiels

Syntax von BPS mit parametrisierten Prozeduren

Int : Z (* Z ist eine ganze Zahl *)

Ide : I (* I ist ein Bezeichner *)

Decl : Δ ::= $\Delta_C \Delta_V \Delta_P$

Δ_C ::= $\varepsilon \mid$ **const** $I_1 = Z_1, \ldots, I_n = Z_n;$

Δ_V ::= $\varepsilon \mid$ **var** $I_1, \ldots, I_n;$

Δ_P ::= $\varepsilon \mid$ **proc** $I(\underbrace{I_1, \ldots, I_p;}\underbrace{\mathbf{var}\ J_1, \ldots, J_q;})B; \ldots$

(* formale Wert- und Variablenparameter *)

AExp : E ::= $Z \mid I \mid (E_1 + E_2) \mid \ldots$

BExp : BE ::= $(E_1 < E_2) \mid$ **not** $BE \mid (BE_1$**and**$BE_2) \mid \ldots$

V : V ::= I (* Variablenparameter *)

Cmd : Γ ::= $I := E \mid I(\underbrace{E_1, \ldots, E_p; V_1, \ldots, V_q}) \mid$

(* aktuelle Parameter(ausdrücke) *)

begin $\Gamma_1; \ldots; \Gamma_n$ **end** $\mid$

if E **then** $\Gamma \mid$ **while** E **do** Γ

Block : B ::= $\Delta\Gamma$

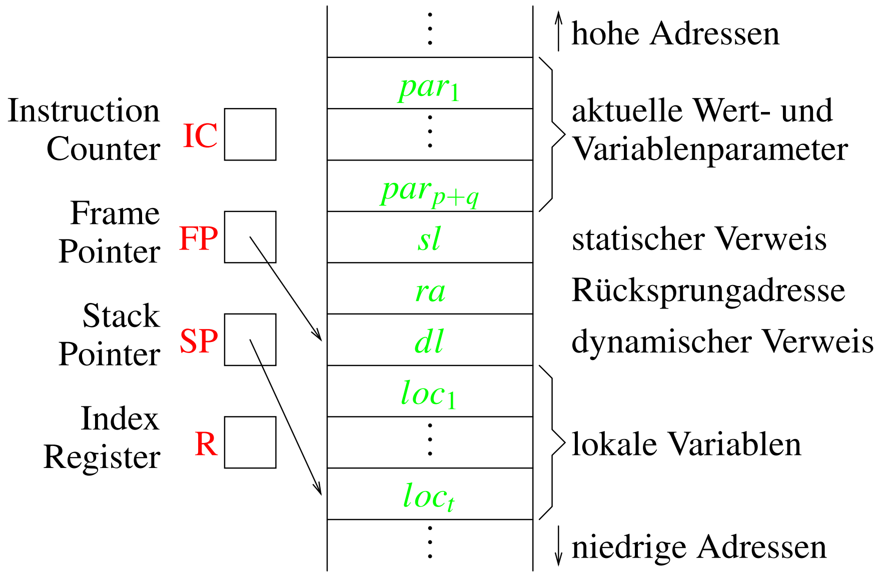
Prog : P ::= **in/out** $I_1, \ldots, I_n; B.$ (* $n \geq 1$ *)

CB 4-18

Abbildung 4.18: Syntax von BPS mit parametrisierten Prozeduren

Stackmaschine für Zwischencode

Zustandsraum:



- nur noch ein Keller für Daten und Aktivierungsblöcke
- Kellerspeicherzellen mit fester Adresse
- SP zeigt auf Kellerspitze (unten)
- FP zeigt auf den *dl*–Eintrag des obersten Aktivierungsblocks

CB 4-19

Abbildung 4.19: Stackmaschine für ZPP-Code

Übersetzung von BPS mit Parametern in Zwischencode

$\mathbf{trans}(\mathbf{in/out}\ I_1,\ldots,I_n;B.) :=$

$$\left.\begin{array}{l} 1: \mathit{PUSH\ FP}; \\ 2: \mathit{CALL}\ a_\Gamma; \end{array}\right\}\text{Anlegen von } \mathit{sl} \text{ und } \mathit{ra}$$

$3: \mathit{JMP}\ 0;$

$\mathbf{bt}(B, st_{I/O}, a_\Gamma, 1, 0)$

$\mathbf{bt}(\Delta\Gamma, st, a, l, r) :=$

$\mathbf{dt}(\Delta, \mathbf{up}(\Delta, st, a.1, l), a.1, l)$

$$\left.\begin{array}{l} a\colon \mathit{PUSH\ FP}; \\ -: \mathit{LOAD\ FP,SP}; \\ -: \mathit{SUB\ SP,}\mathbf{size}(\Delta\Gamma); \end{array}\right\}\text{Entry–Code}$$

$\mathbf{ct}(\Gamma, \mathbf{up}(\Delta, st, a.1, l), a+3, l)$

$$\left.\begin{array}{l} -: \mathit{LOAD\ SP,FP}; \\ -: \mathit{POP\ FP}; \\ -: \mathit{RET}\ r+1; \end{array}\right\}\text{Exit–Code}$$

$\mathbf{dt}(\mathbf{proc}\ I(I_1,\ldots,I_p;\mathbf{var}\ J_1,\ldots,J_q);B;\ldots, st, a, l) :=$

$\mathbf{bt}(B, st', a.1, l+1, p+q)\ \mathbf{bt}(\ldots)$

falls $\mathbf{diff\text{–}id}(I_1,\ldots,I_p,J_1,\ldots,J_q)$ **and** $\mathbf{diff\text{–}id}(\ldots)$

wobei $st' := st[I_1/(\mathbf{var}, l+1, p+q+2),\ldots,$
$I_p/(\mathbf{var}, l+1, q+3),$
$J_1/(\mathbf{vpar}, l+1, q+2),\ldots,$
$J_q/(\mathbf{vpar}, l+1, 3)]$

(Beachte: Parameterlevel = Blocklevel)

CB 4-20

Abbildung 4.20: Übersetzung von BPS mit Parametern in Zwischencode I

Übersetzung von BPS mit Parametern in Zwischencode (Forts.)

$\mathbf{ct}(I := E, st, a, l) :=$

$\quad \mathbf{et}(E, st, a, l)$

$\quad \textbf{if } st(I) = (\textbf{var}, dl, o) \textbf{ then}$

$\quad\quad \textbf{if } l - dl = 0 \textbf{ then } - : \mathit{POP}\ \langle FP{+}o \rangle;$

$$\quad\quad \textbf{if } l - dl = k+1 \textbf{ then } \begin{array}{l} - : \mathit{LOAD}\ R, \langle FP+2 \rangle; \\ \left.\begin{array}{l} - : \mathit{LOAD}\ R, \langle R+2 \rangle; \\ \vdots \\ - : \mathit{LOAD}\ R, \langle R+2 \rangle; \end{array}\right\} k\text{–mal} \\ - : \mathit{POP}\ \langle R{+}o \rangle; \end{array}$$

$\quad \textbf{if } st(I) = (\textbf{vpar}, dl, o) \textbf{ then}$

$$\quad\quad \textbf{if } l - dl = 0 \textbf{ then } \begin{array}{l} - : \mathit{LOAD}\ R, \langle FP{+}o \rangle; \\ - : \mathit{POP}\ \langle R \rangle; \end{array}$$

$$\quad\quad \textbf{if } l - dl = k+1 \textbf{ then } \begin{array}{l} - : \mathit{LOAD}\ R, \langle FP+2 \rangle; \\ \left.\begin{array}{l} - : \mathit{LOAD}\ R, \langle R+2 \rangle; \\ \vdots \\ - : \mathit{LOAD}\ R, \langle R+2 \rangle; \end{array}\right\} k\text{–mal} \\ - : \mathit{LOAD}\ R, \langle R{+}o \rangle; \\ - : \mathit{POP}\ \langle R \rangle; \end{array}$$

CB 4-21

Abbildung 4.21: Übersetzung von BPS mit Parametern in Zwischencode II

Übersetzung von BPS mit Parametern in Zwischencode (Forts.)

$\mathbf{ct}(I(E_1,\ldots,E_p;V_1,\ldots,V_q),st,a,l) :=$

if $st(I) = (\mathbf{proc},ca,dl,t)$ **and**

$\mathbf{typ}(E_i) = \mathbf{int}$ **and**

$st(V_j) = (\mathbf{var},l_j,o_j)$ **then**

$$\left.\begin{array}{l}\mathbf{et}(E_1,st,a,l)\\ \vdots\\ \mathbf{et}(E_p,st,a,l)\end{array}\right\}\text{ aktuelle Wertparameter}$$

if $l - l_1 = 0$ **then** $- : PUSH\ FP{+}o_1;$

if $l - l_1 = k+1$ **then** $- : LOAD\ R,\langle FP+2\rangle;$

$$\left.\begin{array}{l}- : LOAD\ R,\langle R+2\rangle;\\ \vdots\\ - : LOAD\ R,\langle R+2\rangle;\end{array}\right\}k\text{–mal}$$

$- : PUSH\ R{+}o_1;$

$\vdots$ (* analog für $2,\ldots,q$ *)

if $l - dl = 0$ **then** $- : PUSH\ FP;$

if $l - dl = k+1$ **then** $- : LOAD\ R,\langle FP+2\rangle;$

$$\left.\begin{array}{l}- : LOAD\ R,\langle R+2\rangle;\\ \vdots\\ - : LOAD\ R,\langle R+2\rangle;\end{array}\right\}k\text{–mal}$$

$- : PUSH\ R;$

$- : CALL\ ca;$

(Ansonsten bleibt **ct** unverändert.)

CB 4-22

Abbildung 4.22: Übersetzung von BPS mit Parametern in Zwischencode III

Übersetzung von BPS mit Parametern in Zwischencode (Forts.)

$$
\begin{array}{l}
\mathbf{et}(I, st, a, l) := \\
\quad \textbf{if } st(I) = (\textbf{const}, Z) \textbf{ then } a\colon PUSH\ Z; \\
\quad \textbf{if } st(I) = (\textbf{var}, dl, o) \textbf{ then} \\
\qquad \textbf{if } l - dl = 0 \textbf{ then } a\colon PUSH\ \langle FP{+}o\rangle; \\
\qquad \textbf{if } l - dl = k+1 \textbf{ then } a\colon LOAD\ R, \langle FP+2\rangle; \\
\qquad\qquad\qquad\qquad\qquad \left.\begin{array}{l} -: LOAD\ R, \langle R+2\rangle; \\ \vdots \\ -: LOAD\ R, \langle R+2\rangle; \end{array}\right\} k\text{–mal} \\
\qquad\qquad\qquad\qquad\qquad -: PUSH\ \langle R{+}o\rangle; \\
\quad \textbf{if } st(I) = (\textbf{vpar}, dl, o) \textbf{ then} \\
\qquad \textbf{if } l - dl = 0 \textbf{ then } a\colon LOAD\ R, \langle FP{+}o\rangle; \\
\qquad\qquad\qquad\qquad -: PUSH\ \langle R\rangle; \\
\qquad \textbf{if } l - dl = k+1 \textbf{ then } a\colon LOAD\ R, \langle FP+2\rangle; \\
\qquad\qquad\qquad\qquad\qquad \left.\begin{array}{l} -: LOAD\ R, \langle R+2\rangle; \\ \vdots \\ -: LOAD\ R, \langle R+2\rangle; \end{array}\right\} k\text{–mal} \\
\qquad\qquad\qquad\qquad\qquad LOAD\ R, \langle R{+}o\rangle; \\
\qquad\qquad\qquad\qquad\qquad PUSH\ \langle R\rangle;
\end{array}
$$

(Ansonsten bleibt **et** unverändert.)

CB 4-23

Abbildung 4.23: Übersetzung von BPS mit Parametern in Zwischencode IV

Syntax von PSSD

Int :	Z		
Bool :	B	::=	**true** | **false**
Real :	R		
Const :	C	::=	Z | B | R
Ide :	I		
Typ :	T	::=	I | **Bool** | **Int** | **Real** |
			array $[Z_1..Z_2]$ **of** T |
			record $I_1 : T_1; \ldots; I_n : T_n$ **end**
Var :	V	::=	I | $V[E]$ | $V.I$
Exp :	E	::=	C | V | $(E_1 + E_2)$ | $\ldots$
Cmd :	Γ	::=	$V := E$ | **begin** $\ldots$ **end** | **if** $\ldots$ |
			while $\ldots$
Decl :	Δ	::=	$\Delta_C \Delta_T \Delta_V$
	Δ_C	::=	ε | **const** $I_1 = C_1, \ldots, I_n = C_n;$
	Δ_T	::=	ε | **type** $I_1 = T_1, \ldots, I_n = T_n;$
	Δ_V	::=	ε | **var** $I_1 \colon T_1, \ldots, I_n \colon T_1;$
Prog :	P	::=	$\Delta\Gamma$

CB 4-24

Abbildung 4.24: Syntax von PSSD

Semantik von PSSD (statisch)

- In Δ vereinbarte Bezeichner müssen paarweise verschieden sein.
- Recordselektoren müssen paarweise verschieden sein.
- **array** $[Z_1..Z_2]$ **of** $T \quad \rightharpoondown \quad Z_1 \leq Z_2$
- Keine Rekursion in Δ_T, d.h. induktiver Typaufbau:
  type $I_1 = T_1, \ldots, I_n = T_n;$ und Typbezeichner I in T_j
  $\rightharpoondown I \in \{I_1, \ldots, I_{j-1}\}$
- Analoge Bedingungen für Typbezeichner in Δ_V
- Bezeichner in Γ müssen in Δ deklariert sein.
- Variablen in Ausdrücken und Wertzuweisungen sind von einem Basistyp.

CB 4-25

Abbildung 4.25: statische Semantik von PSSD

Code für eine abstrakte Maschine mit Datenkeller DK und Hauptspeicher HS

Alte Befehle: arithmetische, logische, Sprungbefehle

Neue Transportbefehle:

$$\mathcal{C}[\![\text{LODI}]\!](m, d:n, h) := \textbf{if } h(n) = z \textbf{ then } (m+1, d:z, h)$$

Bei Indexvariablen wird auf dem DK zunächst die Adresse berechnet und damit dann indirekt ein Wert geladen.

$$\mathcal{C}[\![\text{STOI}]\!](m, d:z:n, h) := (m+1, d, h[n/z])$$

Laufzeitkontrolle bei Arraygrenzen:

$$\begin{aligned}\mathcal{C}[\![\text{JAC}(Z_1, Z_2)]\!](m, d:z, h) := \textbf{if } & Z_1 \le z \le Z_2 \textbf{ then } (m+1, d:z, h)\\ & \textbf{else } (\text{STOP}, d:\underbrace{\text{RTE}}_{\text{„run–time error“}}, h)\end{aligned}$$

CB 4-26

Abbildung 4.26: Code für AM mit DK und HS

Symboltabelle: Informationen über Speicherbedarf und Offsets

$st \in \mathbf{Tab}$:

$$
\begin{aligned}
\mathbf{Ide} \dashrightarrow\ & \{\mathbf{const}\} \times (\mathbb{B} \cup \mathbb{Z} \cup \mathbb{R}) \\
\cup\ & \{\mathbf{type}\} \times \{\mathbf{bool}, \mathbf{real}, \mathbf{int}\} \times \{1\} \\
\cup\ & \{\mathbf{type}\} \times \{\mathbf{array}\} \times \mathbb{Z}^2 \times \mathbf{Ide} \times \mathbb{N} \\
\cup\ & \{\mathbf{type}\} \times \{\mathbf{record}\} \times (\mathbf{Ide} \times \mathbf{Ide} \times \mathbb{N})^* \times \mathbb{N} \\
\cup\ & \{\mathbf{var}\} \times \mathbf{Ide} \times \mathbb{N}
\end{aligned}
$$

Für $\Delta = \mathbf{type}\ \mathrm{Bool} = \mathbf{bool}; \mathrm{Int} = \mathbf{int};$
$\quad\quad \mathrm{Array} = \mathbf{array}[1..20]\,\mathbf{of}\ \mathrm{Bool};$
$\quad\quad \mathrm{Record} = \mathbf{record}\ \mathrm{S} : \mathrm{Array}; \mathrm{T} : \mathrm{Int}\ \mathbf{end};$
$\quad \mathbf{var}\ \mathrm{X} : \mathrm{Int}; \mathrm{Y} : \mathrm{Array}; \mathrm{Z} : \mathrm{Record};$
gilt:

$$
\begin{aligned}
&\mathbf{up}(\Delta, st) = \\
&\quad st[\mathrm{Bool}/(\mathbf{type}, \mathbf{bool}, 1), \\
&\quad\quad \mathrm{Int}/(\mathbf{type}, \mathbf{int}, 1), \\
&\quad\quad \mathrm{Array}/(\mathbf{type}, \mathbf{array}, 1, 20, \mathrm{Bool}, 20), \\
&\quad\quad \mathrm{Record}/(\mathbf{type}, \mathbf{record}, \mathrm{S}, \mathrm{Array}, 0, \mathrm{T}, \mathrm{Int}, 20, 21), \\
&\quad\quad \mathrm{X}/(\mathbf{var}, \mathrm{Int}, 1), \\
&\quad\quad \mathrm{Y}/(\mathbf{var}, \mathrm{Array}, 2), \\
&\quad\quad \mathrm{Z}/(\mathbf{var}, \mathrm{Record}, 22)]
\end{aligned}
$$

CB 4-27

Abbildung 4.27: Symboltabelle

Übersetzung von PSSD–Programmen in Maschinencode

Hilfsfunktion $\boxed{\mathbf{vtyp} : \mathbf{Var} \times \mathbf{Tab} \dashrightarrow \mathbf{Ide}}$ bestimmt zu einer Variablen bzgl. einer Symboltabelle den zugehörigen Typbezeichner:

$$\begin{aligned}
\mathbf{vtyp}(I, st) &:= \mathbf{if}\ st(I) = (\mathbf{var}, J, k)\ \mathbf{then}\ J \\
\mathbf{vtyp}(V[E], st) &:= \mathbf{if}\ \mathbf{vtyp}(V, st) = I\ \mathbf{and} \\
&\ st(I) = (\mathbf{type}, \mathbf{array}, Z_1, Z_2, J, n) \\
&\ \mathbf{then}\ J \\
\mathbf{vtyp}(V.I, st) &:= \mathbf{if}\ \mathbf{vtyp}(V, st) = J\ \mathbf{and}\ st(J) = \\
&\ (\mathbf{type}, \mathbf{record}, S_1, J_1, 0, \ldots, S_n, J_n, p, q) \\
&\ \mathbf{and}\ I = S_i \\
&\ \mathbf{then}\ J_i
\end{aligned}$$

Übersetzungsfunktion $\boxed{\mathbf{vt} : \mathbf{Var} \times \mathbf{Tab} \dashrightarrow \mathrm{Code}}$ berechnet auf DK die Anfangsadresse einer Variablen im HS:

$$\begin{aligned}
\mathbf{vt}(I, st) &:= \mathbf{if}\ st(I) = (\mathbf{var}, J, k)\ \mathbf{then}\ \mathrm{LIT}\ k; \\
\mathbf{vt}(V[E], st) &:= \mathbf{if}\ \mathbf{vtyp}(V, st) = I \\
&\ \mathbf{and}\ st(I) = (\mathbf{type}, \mathbf{array}, Z_1, Z_2, J, n) \\
&\ \mathbf{and}\ st(J) = (\mathbf{type}, \ldots, k)\ \mathbf{then} \\
&\ \mathbf{vt}(V, st) \\
&\ \mathbf{et}(E, st) \\
&\ \mathrm{JAC}(Z_1, Z_2); \\
&\ \mathrm{LIT}\ Z_1; \\
&\ \mathrm{SUB}; \\
&\ \mathrm{LIT}\ k; \\
&\ \mathrm{MULT}; \\
&\ \mathrm{ADD}; \\
\mathbf{vt}(V.I, st) &:= \mathbf{if}\ \mathbf{vtyp}(V, st) = J\ \mathbf{and}\ st(J) = \\
&\ (\mathbf{type}, \mathbf{record}, S_1, J_1, p_1, \ldots, S_n, J_n, p_n, q) \\
&\ \mathbf{and}\ I = S_i\ \mathbf{then} \\
&\ \mathbf{vt}(V, st) \\
&\ \mathrm{LIT}\ p_i; \\
&\ \mathrm{ADD};
\end{aligned}$$

CB 4-28

Abbildung 4.28: Übersetzung von PSSD I

Übersetzung PSSD (Forts.)

Übersetzungsfunktion $\boxed{\mathbf{et} : \mathbf{Exp} \times \mathbf{Tab} \dashrightarrow \text{Code}}$ erzeugt den üblichen DK–Berechnungscode, wobei jetzt Variablenwerte mit LODI geladen werden:

$$
\begin{aligned}
\mathbf{et}(I, st) :=\ & \mathbf{if}\ st(I) = (\mathbf{const}, C)\\
& \mathbf{then}\ \text{LIT}\ C;\\
& \mathbf{else\ if}\ st(I) = (\mathbf{var}, J, k)\\
& \mathbf{and}\ st(J) = (\mathbf{type}, bas, 1)\ \mathbf{then}\\
& \text{LIT}\ k;\\
& \text{LODI};\\
\mathbf{et}(C, st) :=\ & \text{LIT}\ C;\\
\mathbf{et}(V, st) :=\ & \mathbf{if}\ \mathbf{vtyp}(V, st) = J\\
& \mathbf{and}\ st(J) = (\mathbf{type}, bas, 1)\ \mathbf{then}\\
& \mathbf{vt}(V, st)\\
& \text{LODI};\\
\mathbf{et}(E_1 + E_2, st) :=\ & \ldots\ \text{(wie bei PSA)}
\end{aligned}
$$

Übersetzungsfunktion $\boxed{\mathbf{ct} : \mathbf{Cmd} \times \mathbf{Tab} \dashrightarrow \text{Code}}$:

$$
\begin{aligned}
\mathbf{ct}(V := E, st) :=\ & \mathbf{vt}(V, st)\\
& \mathbf{et}(E, st)\\
& \text{STOI};\\
\vdots\ \ & \text{(sonst wie PSA)}
\end{aligned}
$$

CB 4-29

Abbildung 4.29: Übersetzung von PSSD II

Kapitel A

Exkurs: Yacc

A.1 Automatische Parsergenerierung mit Yacc

Parsergenerierung einschließlich synthetischer Attributauswertung (*LALR*(1)). Yacc steht für **Y**et **A**-nother **C**ompiler **C**ompiler (1995).

Mit `lex` wird aus einer lex-Spezifikation eine `lex.yy.c`-Datei erzeugt (der Scanner in C). Mit `yacc` wird eine yacc-Spezifikation in eine `y.tab.c` und `y.tab.h` (Datei mit Tokennummern) umgewandelt. Die `y.tab.h` dient als Eingabe für `lex`. Mit Hilfe des C-Compilers wird dann ein ausführbarer Parser erzeugt.

A.1.1 Aufbau einer Yacc-Spezifikation

Eine yacc-Spezifikation hat eine ähnliche Struktur wie die lex-Spezifikation. Sie besteht aus drei Teilen, die durch `%y` getrennt sind.

1. Definitionen

- von Token: z.B. `%token NAME NUMBER`
  Beachte: nicht deklarierte Token möglich, Benutzung '...', z.B. `'+', '='`
- C-Code für Definitionen (Datenstrukturen, includes, ...). Bezeichnung: `%{ C-Code %}`

2. Regeln CFG mit Attributierung (Anweisungen), z.B.

```
stmt: NAME '=' expr
    | expr  { printf("%d", $1) }
    ;

expr : expr '+' NUMBER { $$ = $1 + $3 }
     | NUMBER          { $$ = $1 }
     ;
```

Die lex-Attribute sind bestimmt durch den Anfang des Scanners mit `nextsym`.

3. C-Funktionen zur direkten Übernahme in die Datei `y.tab.c`. Verwendung bei Attributfunktionen.
Passende lex-Spezifikation zu obigen Beispiel:

```
%{
  #include "y.tab.h"
  extern int yylval;
%}

%%

[0-9]+     { yylval = atoi(yytext); return NUMBER; }
[a-zA-Z]+  { return NAME; }
[ |\t]     /* Leerzeichen */
\n         { return 0; }
.          { return yytext[0]; }

%%
```

weitere Möglichkeiten

- mehrere synthetische Attribute mit C-Strukturen
- inherite Attribute mit C-Funktionen; Zugriff auf vorher berechnete, tiefer im Stack liegende Werte durch $\$0, \$1, \ldots$
- Konfliktbehandlung
- Präzendenz zur Behandlung von Mehrdeutigkeiten, z.B. Punkt-vor-Strichrechnung, dangling else

Beispielgrammatik

$$
\begin{array}{ll}
B \to 0 & v.0 = 0 \\
B \to 1 & v.0 = 1 \\
L \to B & v.0 = v.1 \\
 & l.0 = 1 \\
L \to LB & v.0 = 2 * v.1 + v.2 \\
 & l.0 = l.1 + 1 \\
N \to L & v.0 = v.1 \\
N \to L.L & v.0 = v.1 + v.2/2^{l.2}
\end{array}
$$

A.2 Behandlung von Konflikten in Yacc

Shift/Reduce: Shift

Reduce/Reduce: Reihenfolge: erste Alternative bevorzugt

Beispiel: (O'Reilly: lex & yacc, S.258)

```
S: girls
 | boys
 ;
```

```
girls: Alice
     | Chris
     ;

boys: Bob
    | Chris
    ;

> yacc -d -v rr.y
yacc: 1 rule never reduced
yacc: 1 reduce/reduce conflict
```

Die Option „`-v`“ im obigen Beispiel erzeugt die Datei `y.output` mit Zusatzinformationen. Aufgrund des reduce/reduce-Konfliktes wird „`chris`“ immer zu „`girls`“ reduziert. Eine mögliche Lösung für dieses Problem ist eine Änderung der Grammatik:

```
S: girls
 | boys
 | either
 ;

girls: Alice;

boys: Bob;

either: Chris;
```

Beachte: Es kommt zu einer Änderung des Parsebaums $\curvearrowright$ es sind also evtl. weitere Änderungen in anderen Phasen der semantischen Analyse erforderlich.

A.3 Präzedenz von Operatoren in Yacc

Assoziativität und Präzedenz werden gleichzeitig festgelegt. Syntax:

`%[left|right]` $op_{1,1} \ldots op_{1,n}$

$\ldots$

`%[left|right]` $op_{m,1} \ldots op_{m,n}$

- Operatoren in **einer** Zeile haben angegebene Assoziativität und **gleiche** Präzedenz.
- In **späteren** Zeilen deklarierte Operatoren haben höhere Präzendenz.

A.3.1 Exkurs: Kodierung der Präzedenzregeln für arithmetische Ausdrücke über der Grammatik

Idee: $+|-$ und $*|/$ in getrennten Regeln

```
expr: expr '+' mulexp
    | expr '-' mulexp
    | mulexp
```

```
    ;

mulexp: mulexp '*' primary
      | mulexp '/' primary
      | primary
      ;

primary: '(' expr ')'
       | '+' primary
       | '-' primary
       | NUMBER
       ;
```

Index

$LL(k)$-Grammatik, 31
$LR(0)$-Auskunft, 40
$LR(0)$-Menge, 40
ε-Hülle, 13
k-look-ahead, 30
Äquivalenz von Programmen, 7

Abhängigkeitsgraphen, 81
Ableitungsrelation, 28
abstrakte Syntax, 84
Abstrakten Syntaxbaums, 84
action-Funktion, 35
Analysephase, 8
Analysetabelle, 35
Attibutgleichungssystem, 79
Attribut, 12
Attribute, 12, 18
Attributen, 79
Attributgleichung, 79
Attributgleichungssystem, 80
Attributgrammatik, 79
Attributgrammatiken, 77
Attributvariablen, 80
Attributwertmengen, 79
Attributzuordnung, 79
Außenvariablen, 79

Backend, 8
Backtrack-DFA, 16
Bezeichner, 12
Bottom-Up-Analyse, 27

direkte Linksrekursion, 36

eindeutig, 28
Einfache Symbole, 12
erweiterte Transitionsfunktion, 13

flm-Analyse, 15
Folge, 15
formalen Attributvariablen, 79
Frontend, 8

imperative, 7
indirekte Linksrekursion, 37
inheriter, 79
inherites, 78
Innenvariablen, 79

Kantenmenge, 81
Kellerautomaten, 27
konkrete Syntax, 84
kontextfreie Grammatik, 27

l-Analyse, 29
L-Attributgrammatik, 85
Leerzeichen, 12
Lexeme, 11
Lexer, 12
Lexikalische Analyse, 12
lexikalische Struktur, 11
Linksanalyse, 27
linksrekursiv, 36
lm-Zerlegung, 14
Look-ahead-Menge, 32

maximal munch, 14
mehrdeutig, 28
MS-Programme, 7

Parser, 27
Passes, 8
Pragmatik, 7
produktiv, 15
PS-Programmen, 7

r-Analyse, 29
Rechtsanalyse, 27
reduziert, 29
reguläre Ausdrücke, 12

Scanner, 12
Schlüsselwörter, 12
Schnittstellen, 27
Semantik, 7
Shift-Reduce-Verfahren, 39
Spezielle Symbole, 12
startsepariert, 39
Symbole, 11
Symbolklassen, 11
Syntaktische Einheiten, 27
Syntax, 7
Synthesephase, 8
synthetischer, 79

TD-Analyseautomat, 29
Token, 11, 12, 18
Top-Down-Analyse, 27

unterhalb *A* abhängig, 81

vereinigt, 45
von-Neumann-Rechners, 7

Zahlwörter, 12
zirkulär, 80
zirkuläre Abhängigkeit, 80
Zusammengesetzte Symbole, 12