Compilerbau Vorlesung – Prof. Dr. K. Indermark – Wintersemester 2002/03

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

14. Februar 2003

Inhaltsverzeichnis

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.
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 lassen 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 1.1.

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)

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

semantische Analyse: Kontextabhängigkeiten, statische Semantik, Typinformationen, 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: Anzahl der Läufe durch das Quellprogramm

Kapitel 1

Lexikalische Analyse

Ausgangspunkt: Quellprogramm P als Zeichenfolge, mit:

Σ_0 : Zeichensatz (`ASCII`, Unicode)
$a \in \Sigma_0$: Zeichen, lexikalisches Atom
$P \in \Sigma_0^*$: Quellprogramm

P besitzt aufgrund der Pragmatik von PS eine *lexikalische Struktur*:

- natürliche Sprache für Bezeichner/Schlüsselwörter
- mathematische Formelsprache für Zahlen, Formeln
- Leerzeichen, Zeilenwechsel, Einrückungen (Tabulatoren), ...
- Kommentare

Die Semantik von P und damit die Übersetzung 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*. 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* dargestellt.

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 Symbole.

Definition: *Scanner*: Der Scanner ist ein Programm für die lexikalische Analyse (siehe 1.2). Wichtigste Symbolklassen:

- *Bezeichner*
- *Zahlwörter*
- *Schlüsselwörter*
- *Einfache Symbole*: ein Sonderzeichen $(+,-,*,(,.,;,\ldots)$, bilden jeweils eine Symbolklasse
- *Zusammengesetzte Symbole*: Folge von zwei oder mehr Symbolzeichen $(:=,**,<=,\ldots)$, bilden ebenfalls eine Symbolklasse
- *Leerzeichen*: $\sqcup, \sqcup\ldots\sqcup, \ldots$
- *Spezielle Symbole*: Kommentare, Pragmas von Compileroptions

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

Definition: *Attribute*: Zeiger in Symboltabelle, Binärdarstellung einer Zahl

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

1.1 Scanner-Konstruktion

Definition: reguläre Ausdrücke: Für ein Alphabet Σ ist die Menge $RA(\Sigma)$ der regulären Σ-Ausdrücke definiert durch:

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

Ihre Semantik:

- $[\![.]\!] : RA(\Sigma) \rightarrow \mathfrak{p}(\Sigma^*)$
- $[\![\Lambda]\!] := \emptyset$
- $[\![a]\!] := \{a\}$
- $[\![\alpha \vee \beta]\!] := [\![\alpha]\!] \vee [\![\beta]\!]$
- $[\![\alpha \cdot \beta]\!] := [\![\alpha]\!] \cdot [\![\beta]\!]$
- $[\![\alpha^*]\!] := [\![\alpha]\!]^* = \bigcup_{n=0}^{\infty} [\![\alpha]\!]^n$
- $[\![\Lambda^*]\!] := [\![\Lambda]\!]^* = \emptyset^*$
- $L^* = \bigcup_{n=0}^{\infty} L^n, L^n = \{w_1 w_2 \ldots w_n | w_i \in L\} = \overbrace{L \cdot L \cdot L \cdot \ldots \cdot L}^{n}$

- $L^n \cdot L^m = L^{n+m}, L^0 \cdot L^m = L^m, L^0 := \{\varepsilon\}$
- Spracherweiterung von $RA(\Sigma)$ zur einfacheren Beschreibung von Symbolklassen

a. Vereinfachende Bezeichnungen:

   - Präzedensregeln 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“)
     - $[abc] := a|b|c$
     - $[a-z] := a|b|c|\ldots|z$

b. Reguläre Definitionen: Schrittweise Beschreibung von Symbolklassen durch zusätzliche freigewählte Bezeichner.

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

1.1.1 Das einfache Matching-Problem

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

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

mit $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to \mathfrak{p}(Q)$ (Epsilon-Transitionen sind zugelassen)

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

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

DFA-Methode

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

(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 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, reicht die Potenzmengen-Konstruktion für den „Lauf von w durch $\mathfrak{A}(\alpha)$“.
Der Algorithmus ist in Abbildung 1.4 gegeben.
Die Komplexität des Algorithmus ist wie folgt gegeben:

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

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

1.1.2 Das erweiterte Matching-Problem

Definition: Seien $\alpha_1, \ldots, \alpha_n \in RA(\Sigma)$ und $w \in \Sigma^*$. Sei ferner $\Delta := \{T_1, \ldots, T_n\}$ ein Alphabet von Token. Wenn $w = w_1 w_2 \ldots w_k$ und $w_j \in [\![\alpha_{i_j}]\!]$ 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.

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:

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

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

2. Prinzip des ersten Match 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 $\alpha_1 \ldots \alpha_k$. Dann heißt v eine *flm-Analyse* („first longest match“), falls für alle $j = 1, \ldots, k$ und $i = 1, \ldots, n$ gilt:

   $$w_j \in [\![\alpha_i]\!] \curvearrowright i_j \leq i$$

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

Berechnung der flm-Analyse

Voraussetzung: $\alpha_1, \ldots, \alpha_n \in RA(\Sigma)$, o.B.d.A. $\varepsilon \notin [\![\alpha_i]\!] \neq \emptyset$, für $i = 1, \ldots, n$
$w \in \Sigma^*$
$\Delta = \{T_1, \ldots, T_n\}$

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

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

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

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

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

   Dann gilt:

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

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

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 $\mathfrak{B}$: $\underbrace{(\{N\} \cup \Delta)} \times \underbrace{\Sigma^* Q \Sigma^*} \times \underbrace{\Delta^* \{\varepsilon, lexerr\}}$

   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 $w \in \Sigma^*$:

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

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

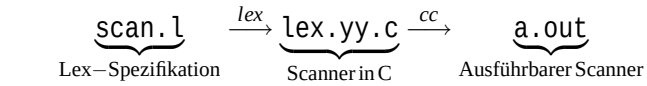
Beispiel: $\alpha_1 = abc \quad \alpha_2 = (abc)^* \quad w = (abc)^m d$ erfordert $O(m^2)$ Schritte.

1.2 Praktische Aspekte der Scanner-Konstruktion

- Spracherweiterung von $RA(\Sigma)$: Abkürzungen, reguläre Definition.
- 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 Lookahead (Backtrack erforderlich). Lookahead von 1 Zeichen in Pascal, allerdings kann diese Beschränkung die lexikalische Analyse verändern.
- Besondere Rolle von „blanks“ (Whitespace): Trennung von eigentlichen Lexemen, um mit 1-lookahead auszukommen.
- Attributberechnung:
  - erneutes Lesen des Eingabestrings
  - Dezimalzahl $\rightsquigarrow$ Binärzahl (Lex: `install_num`)
  - Bezeichner: Gleichheit prüfen, Symboltabelle (Lex: `install_id`)
  - Schlüsselwörter (alternativ) zunächst als Bezeichner behandeln, Symboltabelle mit Schlüsselwörtern initialisieren. Die Attributberechnung (`install_id`) liefert dann den Schlüsselwerttoken (z.B. (if,) statt (id, $\rightarrow$ SymTab)

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

Linux: `> man flex`, Unix-Tool



Programm $\xrightarrow{\texttt{a.out}}$ Symbolfolge

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

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 $\longrightarrow$ abfangen mit `.|\n` $\rightsquigarrow$ 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.

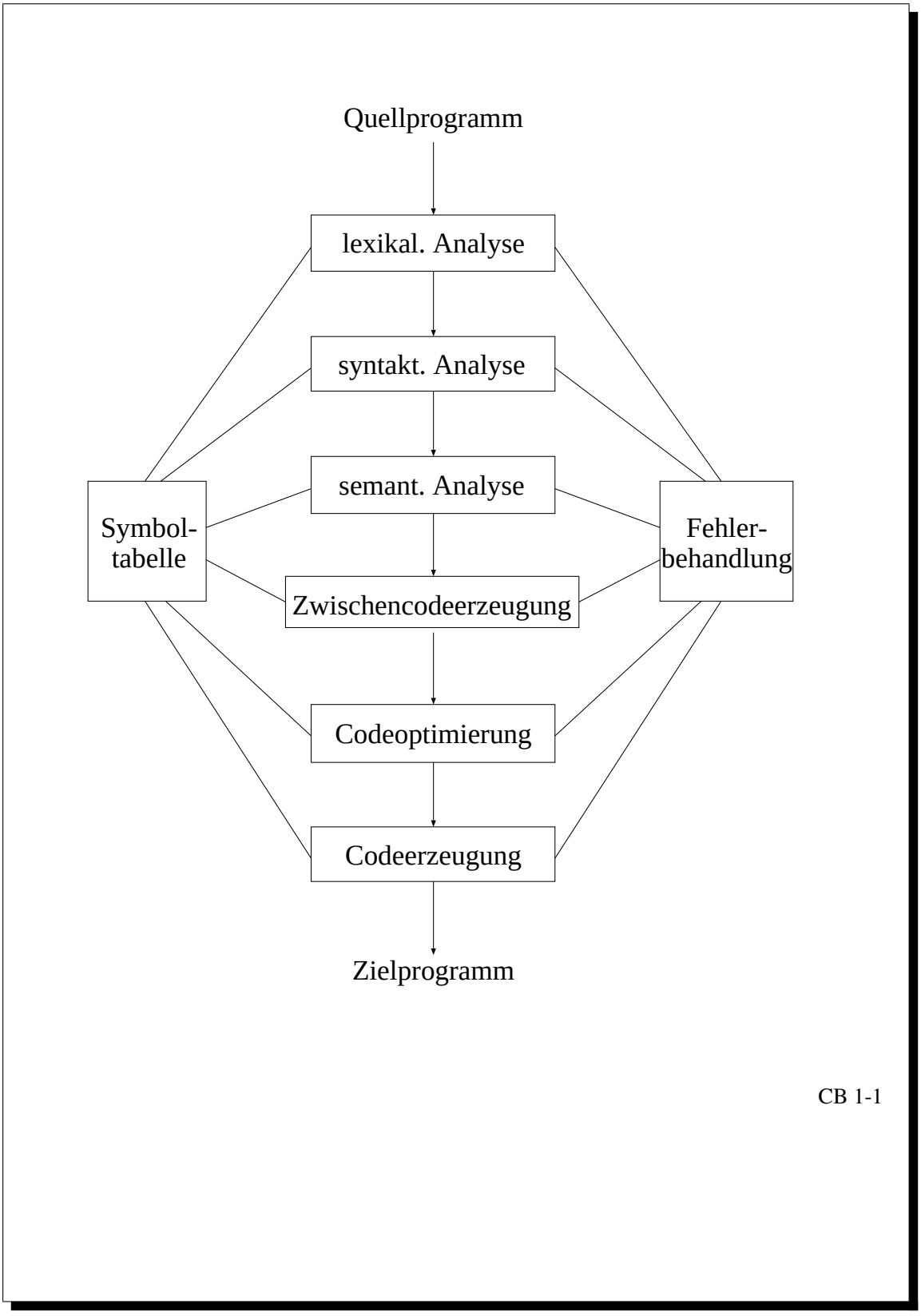


Abbildung 1.1: Struktur eines Compilers

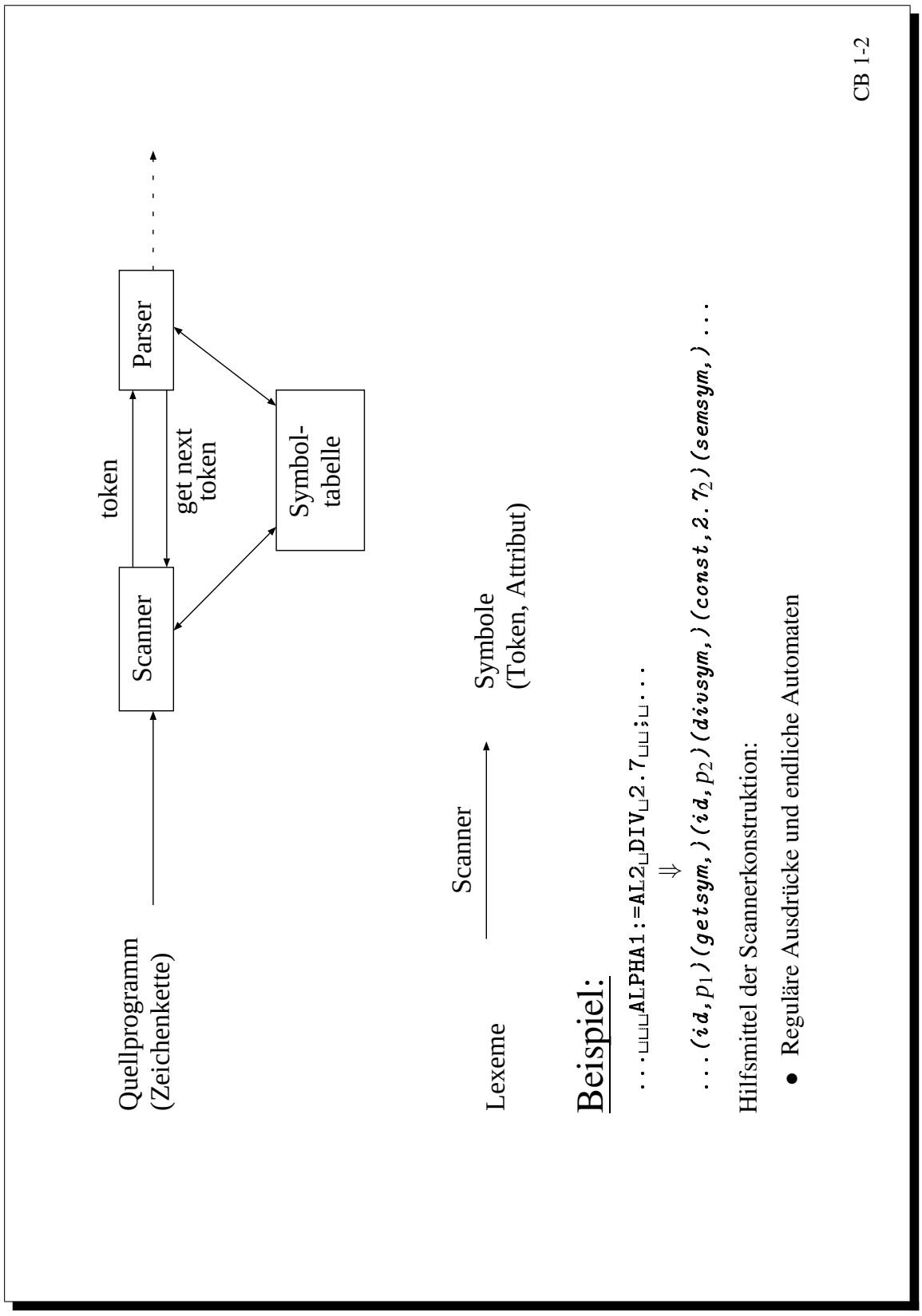


Abbildung 1.2: Scanner

Beispiel: Pascal / Teil 1: Symbolklassen

- Bezeichner: letter = [A-Za-z]
  digit = [0-9]
  id = letter (letter | digit)*
- Zahlwörter: digits = digit+
  (ohne Vorzeichen) 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 1.3: Symbolklassen in Pascal, Teil I

Abbildung 1.4: NFA-Methode

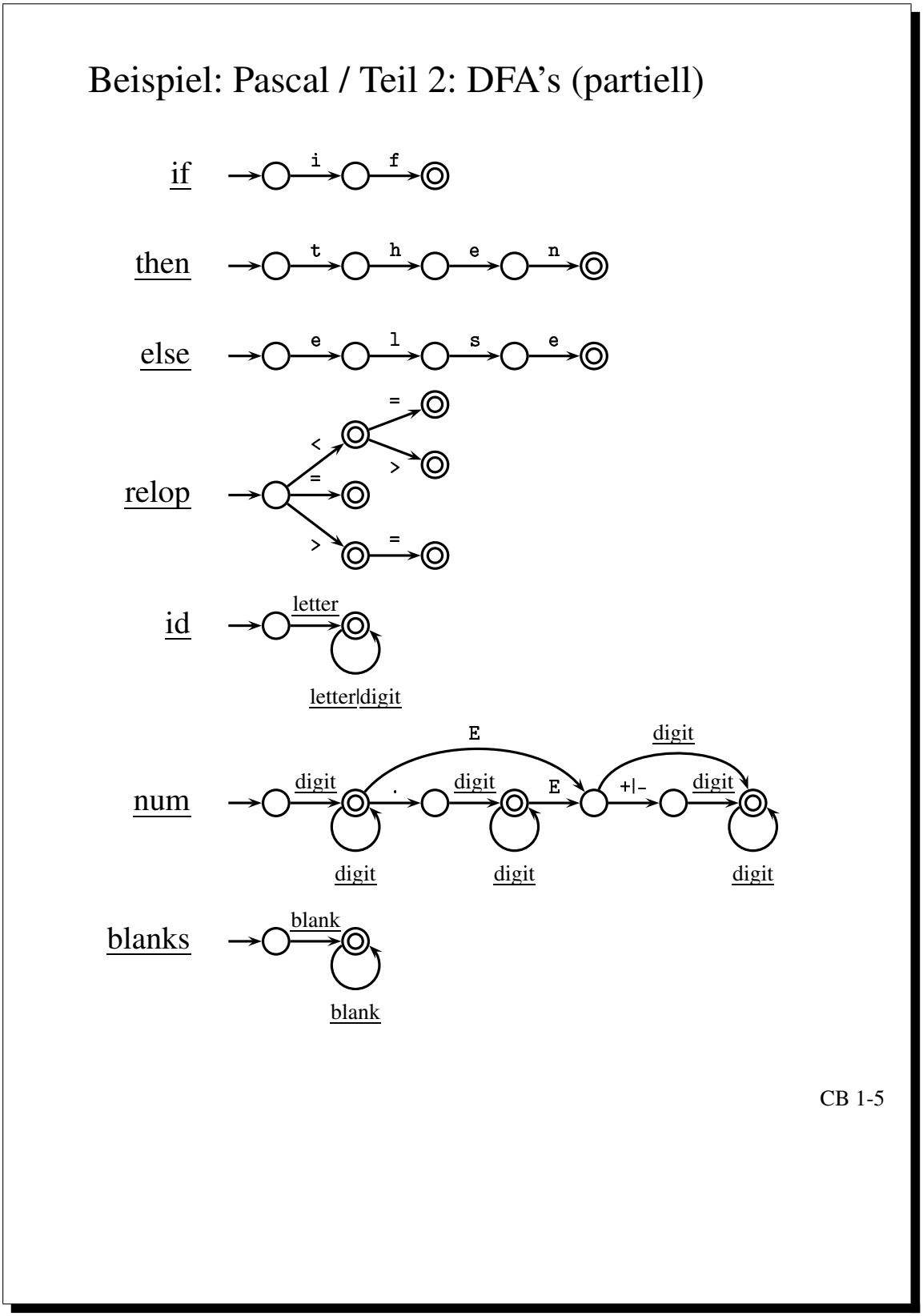


Abbildung 1.5: Pascal, Teil II

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`|`r_2	Alternative

Präzedenzregeln wie üblich

CB 1-6

Abbildung 1.6: 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 1.7: Scannergenerierung mit flex

Kapitel 2

Syntaktische Analyse

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

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

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

Parser: Programm für die syntaktische Analyse
Schnittstellen: siehe Abbildung 2.4
Beschreibung der syntaktischen Struktur durch eine *kontextfreie Grammatik*. 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

$$G =< N, \Sigma, P, S > \in \mathrm{CFG}(\Sigma)$$

Bezeichnungskonventionen:

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

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

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

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

$$\alpha_1 \underset{l}{\Rightarrow} \beta \qquad \text{bzw.} \qquad \alpha_2 \underset{r}{\Rightarrow} \beta$$

Sprechweise: Ableitungsschritt; α_1: Links-, α_2: Rechtsableitungsschritt

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

Offensichtlich gilt $L(G) = \{w \in \Sigma^* | S \underset{l}{\overset{*}{\Rightarrow}} w\}$ und $L(G) = \{w \in \Sigma^* | S \underset{r}{\overset{*}{\Rightarrow}} w\}$
Beispiel: $G : S \to aSb|\varepsilon$
$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aa\varepsilon bb = aabb$

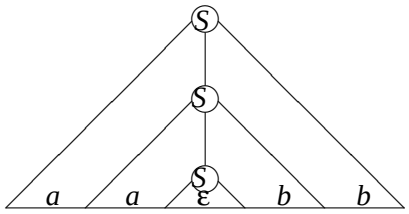


Abbildung 2.1: Der Ableitungsbaum, der die syntaktische Struktur von a^2b^2 bezüglich G repräsentiert

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

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

Definition: G ist *mehrdeutig*, wenn G nicht eindeutig ist.

2.1.1 l-Analyse, r-Analyse

Darstellung von l/r-Ableitungen durch Nummernfolgen

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

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

Für $z = i_1 \ldots i_k \in [p]^+$ soll für passende $\alpha_1 \ldots \alpha_{k-1}$ gelten:

$$\alpha \underset{l}{\overset{z}{\Rightarrow}} \beta : \curvearrowright \alpha \underset{l}{\overset{i_1}{\Rightarrow}} \alpha_1 \underset{l}{\overset{i_2}{\Rightarrow}} \alpha_2 \ldots \underset{l}{\overset{i_k}{\Rightarrow}} \alpha_k = \beta$$

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

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

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

Beispiel: siehe Abbildung 2.5

2.1.2 Syntaxanalyse

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

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

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

Definition: Der *TD-Analyseautomat* von $G \in \mathrm{CFG}$, $(NTA(G))$ ist gekennzeichnet durch:

Eingabealphabet:	Σ	(Zustandsalphabet entfällt)
Kelleralphabet:	χ	Ausgabealphabet: $[p]$
Konfigurationsmenge:	$\Sigma^* \times \chi^* \times [p]^*$	(Kellerspitze links)

sowie Transitionen wie folgt:

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

Die Anfangskonfiguration für ein $w \in \Sigma^*$ wie folgt: (w, S, ε).

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

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

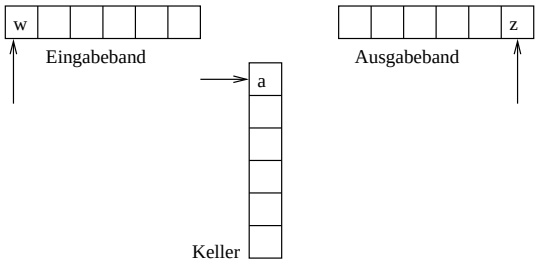


Abbildung 2.2: Analyseautomat

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

$\alpha \underset{l}{\overset{iz'}{\Rightarrow}} w$

Beweis: „$\curvearrowleft$“ ähnlich.

□

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

Definition: (first-Mengen). Sei $G \in \text{CFG}, \alpha \in \chi^*$ 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:

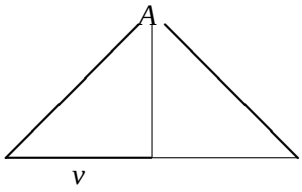


Abbildung 2.3: $|v| = k \curvearrowright v \in \underline{\text{first}}_k(A)$

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

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

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

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

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

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

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

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

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 simuliert werden.

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

Lemma: $G \in LL(k) \curvearrowright$ Für alle Linksableitungen der Form

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

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 $\underline{\text{first}}_k(y) \subseteq \underline{\text{first}}_k(\gamma\alpha)$.

□

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

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

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

Idee: Mögliche Rechtskontexte vereinigen.

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

Der Fall $k = 1$: Abkürzung $\underline{\text{fi}} := \underline{\text{first}}_1$ und $\underline{\text{fo}} := \underline{\text{follow}}_1$.

Satz: Für $G \in$ CFG (reduziert!) gilt:
$G \in LL(1) \curvearrowright$ 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: $\underline{\text{la}}(A \to \beta) := \underline{\text{fi}}(\beta\underline{\text{fo}}(A)) \subseteq \Sigma_\varepsilon := \Sigma \cup \{\varepsilon\}$ 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. „$\downarrow$' (Lemma $\to$ Satz): Angenommen, es gibt $A \to \beta|\gamma$ mit $\beta \neq \gamma$, aber: $c \in \underline{\text{fi}}(\beta\underline{\text{fo}}(A)) \cap \underline{\text{fi}}(\gamma\underline{\text{fo}}(A))$

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:

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

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

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

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

b. „↑' (Satz → Lemma):

$$\text{Sei } S \overset{*}{\underset{l}{\Rightarrow}} wA\alpha \quad \begin{matrix} \overset{l}{\nearrow} & w\beta\alpha \\ \searrow_{l} & w\gamma\alpha \end{matrix} \quad \text{mit } \beta \neq \gamma$$

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

□

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

2.2.1 Berechnung der la-Mengen

1. $\underline{\text{fi}}(X)$ für $X \in \chi$
   - $X = a \in \Sigma \curvearrowright \underline{\text{fi}}(X) = \{X\}$
   - $X \to a\alpha \curvearrowright a \in \underline{\text{fi}}(X)$
   - $X \to \varepsilon \curvearrowright \varepsilon \in \underline{\text{fi}}(X)$
   - $X \to A_1 \ldots A_k Y\alpha, k \geq 0, Y \in \chi, \varepsilon \in \underline{\text{fi}}(A_1) \cap \ldots \cap \underline{\text{fi}}(A_k), a \in \underline{\text{fi}}(Y) \curvearrowright a \in \underline{\text{fi}}(X)$
   - $X \to A_1 \ldots A_k, k \geq 1, \varepsilon \in \underline{\text{fi}}(A_1) \cap \ldots \cap \underline{\text{fi}}(A_k) \curvearrowright \varepsilon \in \underline{\text{fi}}(X)$
2. $\underline{\text{fi}}(X_1 \ldots X_n)$ für $X_i \in \chi, n \in \mathbb{N}$
   - $\varepsilon \in \underline{\text{fi}}(X_1) \cap \underline{\text{fi}}(X_2) \cap \ldots \cap \underline{\text{fi}}(X_{i-1}), a \in \underline{\text{fi}}(X_i) \curvearrowright a \in \underline{\text{fi}}(X_1 \ldots X_n)$
   - $\varepsilon \in \underline{\text{fi}}(X_1) \cap \underline{\text{fi}}(X_2) \cap \ldots \cap \underline{\text{fi}}(X_n) \curvearrowright \varepsilon \in \underline{\text{fi}}(X_1 \ldots X_n)$
   - $\underline{\text{fi}}(\varepsilon) = \{\varepsilon\}$
3. $\underline{\text{fo}}(A)$
   - $\varepsilon \in \underline{\text{fo}}(S)$
   - $A \to \alpha B\beta, a \in \underline{\text{fi}}(\beta) \curvearrowright a \in \underline{\text{fo}}(B)$
   - $A \to \alpha B, x \in \underline{\text{fo}}(A) \curvearrowright x \in \underline{\text{fi}}(B)$
   - $A \to \alpha B\beta, \varepsilon \in \underline{\text{fi}}(\beta), x \in \underline{\text{fo}}(A) \curvearrowright x \in \underline{\text{fo}}(B)$

Beispiel:

G'_{AE} :

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

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

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

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

Beachte: (neue Prüfungsfrage): $G \in$ CFG, G reduziert. Regel in $G: A \to \alpha B$.
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 $\underline{\text{fo}}(B) \subseteq \underline{\text{fo}}(A)$

2.2.2 Der deterministische TD-Analyseautomat $DTA(G)$ für $G \in LL(1)$

Idee: Die Zugehörigkeit des Eingabesymbols zu einer $\underline{\text{la}}$-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.

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

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

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

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

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

2.3 Parserkonstruktion nach TD-Methode

$G \in$ CFG reduziert gegeben.
Berechnung der la-Mengen (fi- und fo-Mengen)
Analysetabelle: Eindeutigkeit prüfen
tabellengesteuerter Parser

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

2.3.1 Transformationen nach $LL(1)$

2 Methoden um 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: $G \in CFG$ *linksrekursiv*: $\curvearrowright \exists A \in N, \alpha \in \chi^* : A \overset{+}{\Rightarrow} A\alpha$.

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

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

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

Beispiel:

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

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

Probe: $LL(1)$-Test

$$\underline{\text{fi}}(E) = \underline{\text{fi}}(T) = \underline{\text{fi}}(F) = \{(,a\} \curvearrowright \underline{\text{la}}(\Pi_i) = \{(,a\}, i = 1 \ldots 6$$

Das bestätigt, daß G_{AE} keine $LL(1)$-Grammatik ist.

Spezialfall: *direkte Linksrekursion* und ihre Beseitigung

$$A \to A\alpha|\beta(\beta \neq A, \alpha \neq \varepsilon)$$

wird ersetzt durch

$$A \to \beta A' \qquad A' \to \alpha A' | \varepsilon$$

Folgerung: $L(G)$ unverändert, jedoch neue syntaktische Struktur (kein Problem bei assoziativen Operatoren $+, *, \ldots$).
Beispiel: Beseitigung direkter Linksrekursionen in G_{AE} ergibt G'_{AE}.

Allgemeiner Fall: *indirekte Linksrekursion*

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

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

$$A \to aB_1 \ldots B_n \qquad S \to \varepsilon$$

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

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

Komplexität der LL(1)-Analyse

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

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

- $|w|$ Vergleichsschritte plus einem „accept“-Schritt.
- maximal $|N|$ aufeinanderfolgende Ableitungsschritte (Begründung: würde ein Nichtterminalsymbol doppelt auftauchen, ohne das ein Vergleichsschritt durchgeführt wurde, so würde eine links-Rekursion vorliegen).

  G nicht linksrekursiv. Dann ex. $c \in \mathbb{N}$ mit $A \overset{i}{\Rightarrow} \alpha$ und $|\alpha| \geq 1 \curvearrowright 1 \leq c|\alpha|$

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

2.4 Top-Down Analyse mit rekursiven Prozeduren

Idee: keine explizite Kellerbenutzung wie beim $DTA(G)$, sondern implizite Verwendung des Laufzeitkellers durch Einsatz rekursiver Prozeduren.

Spezialfall: $G \in LL(1)$ (Beispiel: G'_{AE}).

2.4.1 Analyseverfahren durch rekursiven Abstieg (ohne la-Mengen)

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

Annahme: Alternativen durch Eingabesymbol unterscheidbar.

Eingabe: *sym* Variable für das Eingabesymbol, *nextsym* zum Lesen des nächsten Eingabesymbols.
Ausgabe: $print(i)$ zur Ausgabe einer Regelnummer.

2.4.2 Zusätzliche Verwendung der la-Mengen

Vorteil: bessere Kontrolle der Regelanwendungen, frühere Fehlererkennung.

Weitere Möglichkeit: nicht-deterministische Programmierung (`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 dem Keller

Reduce-Schritte: Umkehrung von Ableitungsschritten.

Dies wird *Shift-Reduce-Verfahren* genannt.

Definition: Der (nicht-deterministische) Bottom-Up-Analyseautomat von $G \in \text{CFG}$ ($NBA(G)$):

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

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

Aufgabe: Gegeben $G \in \text{CFG}$. 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: $S \to aBBBBBBc, \quad B \to \varepsilon$).

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

Folgerung: (S', ε, z) ist eine Endkonfiguration. Also ergibt die Startseparitertheit ein deterministisches Analyseende.

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

$$S \begin{cases} \overset{*}{\underset{r}{\nearrow}} & \alpha A w \underset{r}{\Rightarrow} \alpha\beta w \\ & \qquad\qquad\quad \text{mit } \underline{\text{first}}_k(w) \\ \overset{*}{\underset{r}{\searrow}} & \alpha' A' w' \underset{r}{\Rightarrow} \alpha\beta v \quad = \underline{\text{first}}_k(v) \end{cases}$$

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

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 $G \in$ CFG mit $S' \to S$ und $S' \underset{r}{\overset{*}{\Rightarrow}} \alpha Aw \underset{r}{\Rightarrow} \alpha\beta_1\beta_2 w$. 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. $(\alpha\beta_i, w, z) \vdash (\alpha A, w, zi)$ für $\Pi_i = A \to \beta_1$ und $\gamma = \alpha\beta_1$.

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 G enthalten keine widersprüchlichen Auskünfte.

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

Satz: $G \in$ CFG mit $S' \to S$, 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 G auch $[B \to \cdot\beta]$ enthält

$\underline{\mathrm{LR}(0)}(\varepsilon).\alpha = \varepsilon$ und $\beta_1 = \varepsilon$
$\overline{[S' \to \cdot S]} \in \underline{\mathrm{LR}(0)}(\varepsilon)$
$S \to A\gamma \curvearrowright \overline{[S \to \cdot A\gamma]} \in \underline{\mathrm{LR}(0)}(\varepsilon)$
$A \to a \curvearrowright [S \to \cdot a] \in \underline{\mathrm{LR}(0)}(\varepsilon)$

Die goto-Funktion

$G: LR(0)$-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.

$\underline{\mathrm{goto}}: \underline{\mathrm{LR}(0)}(G)x\chi \to \underline{\mathrm{LR}(0)}(G)$ 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 $G \in \mathrm{CFG}$, G startsepariert mit $S' \to S$. Konstruktion eines $\mathfrak{A}(G) \in \mathrm{NFA}_\varepsilon$

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

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

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

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

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

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

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

Eindeutigkeit bei $G \in LR(0)$.

Die Funktionen $\underline{\text{act}}$ und $\underline{\text{goto}}$ bilden die $\underline{\text{LR}(0)}$-Analysetabelle von G (Beispiel: Abbildung 2.17). Diese Tabelle bestimmt den $\underline{\text{LR}(0)}$-Analyseautomaten.

Eingabealphabet:	Σ
Kelleralphabet:	$\Gamma := \underline{\text{LR}(0)}(G)$
Ausgabealphabet:	$\Delta := [p] \cup \{\underline{\text{error}}\} \cup \{0\}$

Transistionen:

- Shift-Schritt: $(\alpha I, aw, z) \vdash (\alpha I I', w, z)$, falls $\underline{\text{act}}(I) = \underline{\text{shift}}$ und $\underline{\text{goto}}(I, a) = I'$.
- Reduce-Schritt: (Bezeichnung: $wa_1 \ldots a_n - (n) := w$) $(\alpha I, w, z) \vdash (\tilde{\alpha} \tilde{I} I', w, zi)$, falls $\underline{\text{act}}(I) = \underline{\text{red}_i}, \Pi_i = A \to X_1 \ldots X_n, \alpha I - (n) = \tilde{\alpha}\tilde{I}$ und $\underline{\text{goto}}(\tilde{I}, A) = I'$.
- Accept-Schritt: $(I_0 I, \varepsilon, z) \vdash (\varepsilon, \varepsilon, z0)$, falls $\underline{\text{act}}(I) = \underline{\text{accept}}$.
- Fehlererkennung: $(\alpha I, w, z) \vdash (\varepsilon, \varepsilon, z \cdot \underline{\text{error}})$, sonst.

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

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

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

Beispiel: $w = aac$ (Grammatik G aus Abbildung 2.15):

$(0, aac, \varepsilon)$
$(04, ac, \varepsilon)$
$(044, c, \varepsilon)$
$(0446, \varepsilon, \varepsilon)$
$(0448, \varepsilon, 6)$
$(048, \varepsilon, 65)$
$(03, \varepsilon, 655)$
$(01, \varepsilon, 6552)$
$(\varepsilon, \varepsilon, 65520)$
$\overleftarrow{}$

2.5.3 $SLR(1)$-Analyse

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

Beispiel: Shift/Reduce-Konflikte in G_{AE} (siehe Abbildung 2.20: I_1, I_2, I_9).

Beseitigung der Konflikte durch nächstes Eingabesymbol. Beobachtung:

1. $[A \to \beta_1 \cdot a\beta_2] \in \underline{\mathrm{LR}(0)}(\alpha\beta_1) \curvearrowright S \underset{r}{\overset{*}{\Rightarrow}} \alpha Aw \underset{r}{\Rightarrow} \alpha\beta_1 a\beta_2 w$
   Also: Shift nur bei Eingabe von a.

2. $[A \to \beta\cdot] \in \underline{\mathrm{LR}(0)}(\alpha\beta) \curvearrowright S \underset{r}{\overset{*}{\Rightarrow}} \alpha Aaw \underset{r}{\Rightarrow} \alpha\beta aw \curvearrowright a \in \underline{\mathrm{fo}}(A)$
   Also: Reduktion mit $A \to \beta$ nur, falls $a \in \underline{\mathrm{fo}}(A)$.

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.

Die $SLR(1)$-action Funktion

$\underline{\mathrm{act}} : \underline{\mathrm{LR}(0)}(G) \times (\Sigma \cup \{\$\}) \to \{\underline{\text{shift}}, \underline{\text{red}_i}, \underline{\text{accept}}, \underline{\text{error}} | 1 \le i \le r\}$ sei definiert durch:

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

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

Die $\underline{\text{action}}$ und $\underline{\text{goto}}$ Funktionen bilden die $SLR(1)$-Analysetabelle von G (siehe Abbildung 2.21).
Jedoch sind 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

Nicht immer sind Konflikte über follow-Mengen lösbar (siehe Abbildung 2.22).
Aber nicht jedes Element von $\underline{\mathrm{fo}}(R)$ 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 $G \in$ CFG:

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

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

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

Berechnung der $LR(1)$-Mengen

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

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

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

$\underline{\mathrm{act}} : \underline{\mathrm{LR}(1)}(G) \times (\Sigma \cup \{\$\}) \to \{\underline{\mathrm{shift}}, \underline{\mathrm{red}_i}, \underline{\mathrm{accept}}, \underline{\mathrm{error}} | 1 \le i \le r\}$ ist definiert durch:

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

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

2.5.5 $LALR(1)$-Analyse

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	$\sim 10^2$	$\sim 10^3$

m. Beobachtung: ögliche Informationsredundanz bei $\underline{\mathrm{LR}(1)}(G_2)$.

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

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.

Definition:

- $I \in \underline{\mathrm{LR}(1)}(G)$ bestimmt eine $\underline{\mathrm{LALR}(1)}$-Menge.
$$\bigcup \{I' \in \underline{\mathrm{LR}(1)}(G) | I' \underset{0}{\sim} I\}$$
- $\underline{\mathrm{LALR}(1)}(G)$: Menge der $LALR(1)$-Mengen von G.

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

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

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

Beispiel: Die $LALR(1)$-Menge von G_2

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

$$I_i^{(1/0)} := I_i^{(1)} \text{ für } i = 0,1,2,3,6,9$$
$$I_4^{(1/0)} := I_4^{(1)} \cup I_{11}^{(1)}$$
$$I_5^{(1/0)} := I_5^{(1)} \cup I_{12}^{(1)}$$
$$I_7^{(1/0)} := I_7^{(1)} \cup I_{13}^{(1)}$$
$$I_8^{(1/0)} := I_8^{(1)} \cup I_{10}^{(1)}$$

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 $G \in$ CFG: G mehrdeutig $\curvearrowright G \notin LR = \bigcup_{k \in \mathbb{N}} LR(k)$.
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 2.29) $G_{AE}^m : E \to E + E | E * E | (E) | a$:

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	$a + a * a$	
$I_0 I_3$	$+ a * a$	
$I_0 I_1$	$+ a * a$	
$I_0 I_1 I_4$	$a * a$	
$I_0 I_1 I_4 I_3$	$* a$	
$I_0 I_1 I_4 I_7$	$* a$	Reduktionskonflikt, $a > +$
$\underline{\mathrm{act}}(I_7, *)$	$= \underline{\mathrm{shift}}$	
$\underline{\mathrm{act}}(I_7, *)$	$= \underline{\mathrm{red}}_1$	(falls $+ > *$)

Beispiel: Mehrdeutigkeit bei Verzweigungen („dangling else“)

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

siehe auch Abbildung 2.31 (*i*: if, *e*: else). Im $LR(0)$-Graph kommt es bei I_4 zum Konflikt: $e \in \underline{\text{fo}}(S) \curvearrowright$ nicht LR-lösbar.

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

Es existieren zwei Zerlegungen:

1. $\underline{\text{if}}\, b\, \underline{\text{then}}\, (\, \underline{\text{if}}\, b\, \underline{\text{then}}\, a\, \underline{\text{else}}\, a)$
2. $\underline{\text{if}}\, b\, \underline{\text{then}}\, (\, \underline{\text{if}}\, b\, \underline{\text{then}}\, a)\, \underline{\text{else}}\, a$

2.7 Ergebnis

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

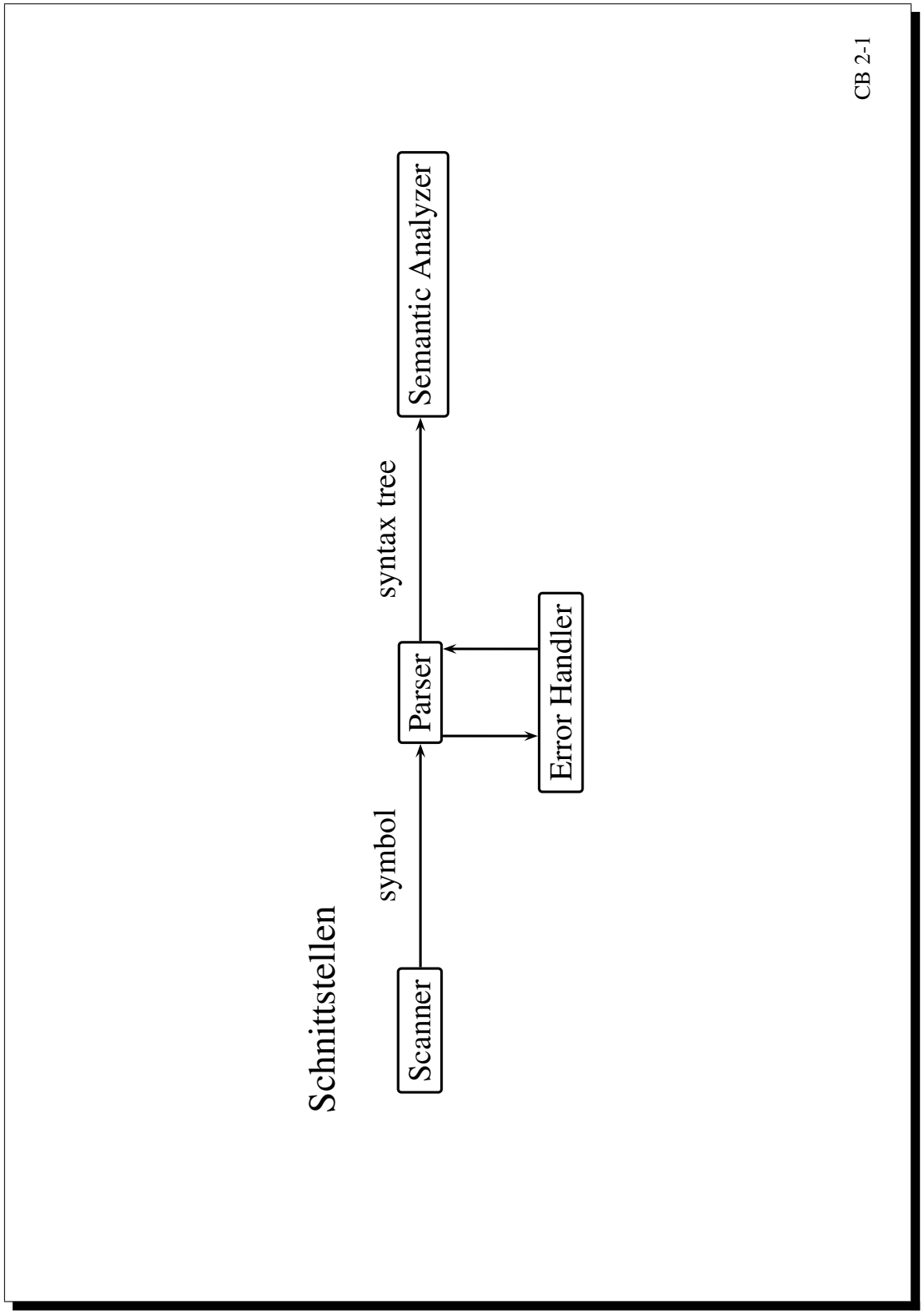


Abbildung 2.4: Schnittstellen

Links– und Rechtsableitungen

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

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

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

Linksableitung von $(a)*c$:

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

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

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

Rechtsableitung von $(a)*c$:

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

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

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

CB 2-2

Abbildung 2.5: Links- und Rechtsableitungen

Bsp: $NTA(G_{AE})$

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

Linksanalyse von $(a)*c$:

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

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	pop					
(	TE', 1		FT', 4		(E), 7		pop				
)		ε, 3		ε, 6				pop			
+		$+TE'$,2		ε, 6					pop		
*				$*FT'$, 5						pop	
ε		ε, 3		ε, 6							accept

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) , 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) , 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	⟩

CB 2-5

Abbildung 2.8: Arbeitsweise von G'_{AE}

RD Parser für G'_{AE}

```
proc T()                        proc E()
  if sym in {'(','a'}             if sym in {'(','a'}
  then print(4);                  then print(1);
       F(); T'();                      T(); E'();
  else print(ERROR);stop;         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'()                       proc T'()
  if sym='+' then                 if sym='*' then
     print(2);                       print(5);
     nextsym;                        nextsym;
     T(); E'();                      F(); T'();
  elseif sym in {')',ε}           elseif sym in {'+',ε}
  then  print(3)                  then print(6);
  else print(ERROR); stop;        else print(ERROR); stop;
```

CB 2-6

Abbildung 2.9: Recursive descent parser für G'_{AE}

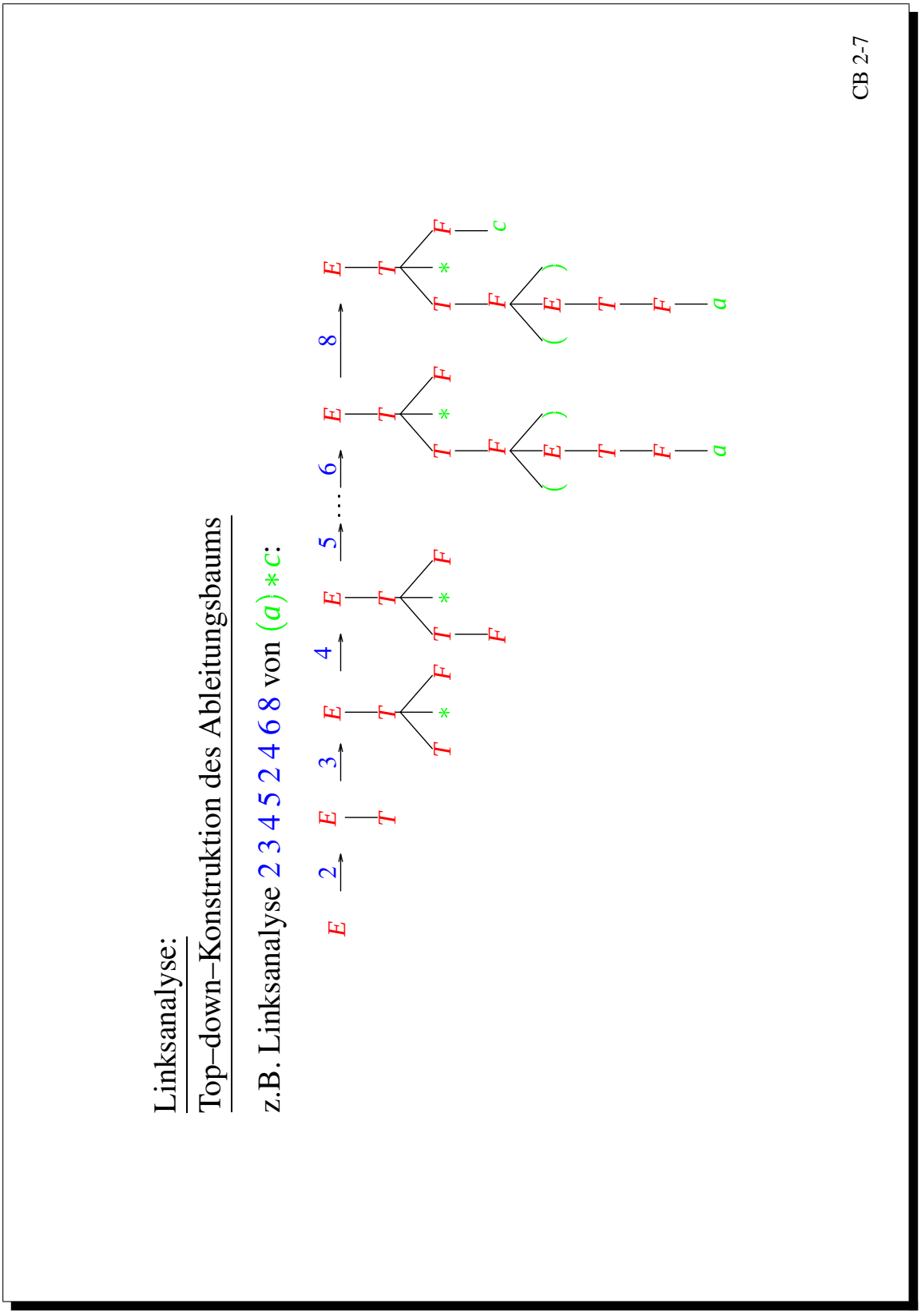


Abbildung 2.10: Linksanalyse / TD-Konstruktion

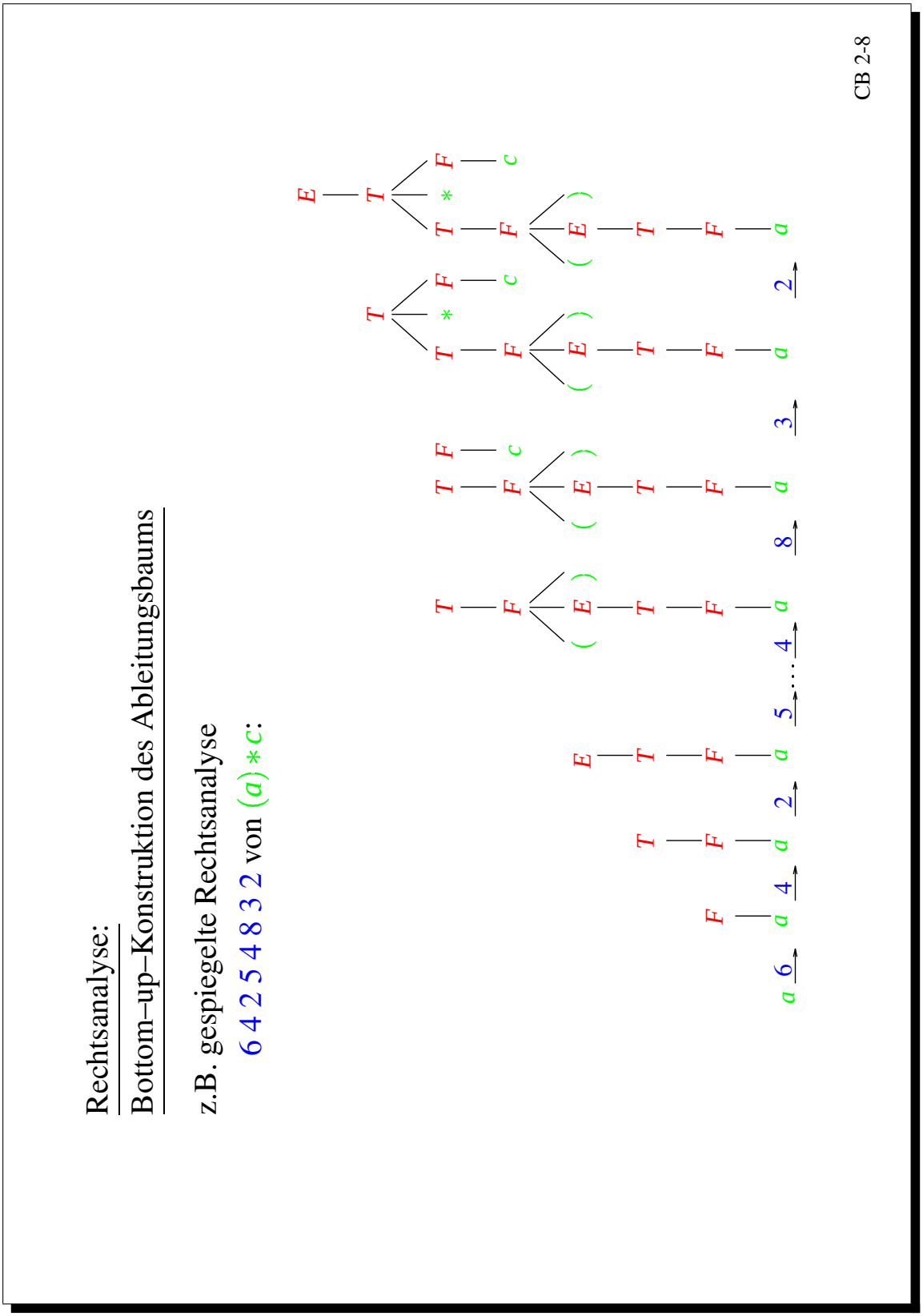


Abbildung 2.11: Rechtsanalyse / BU-Konstruktion

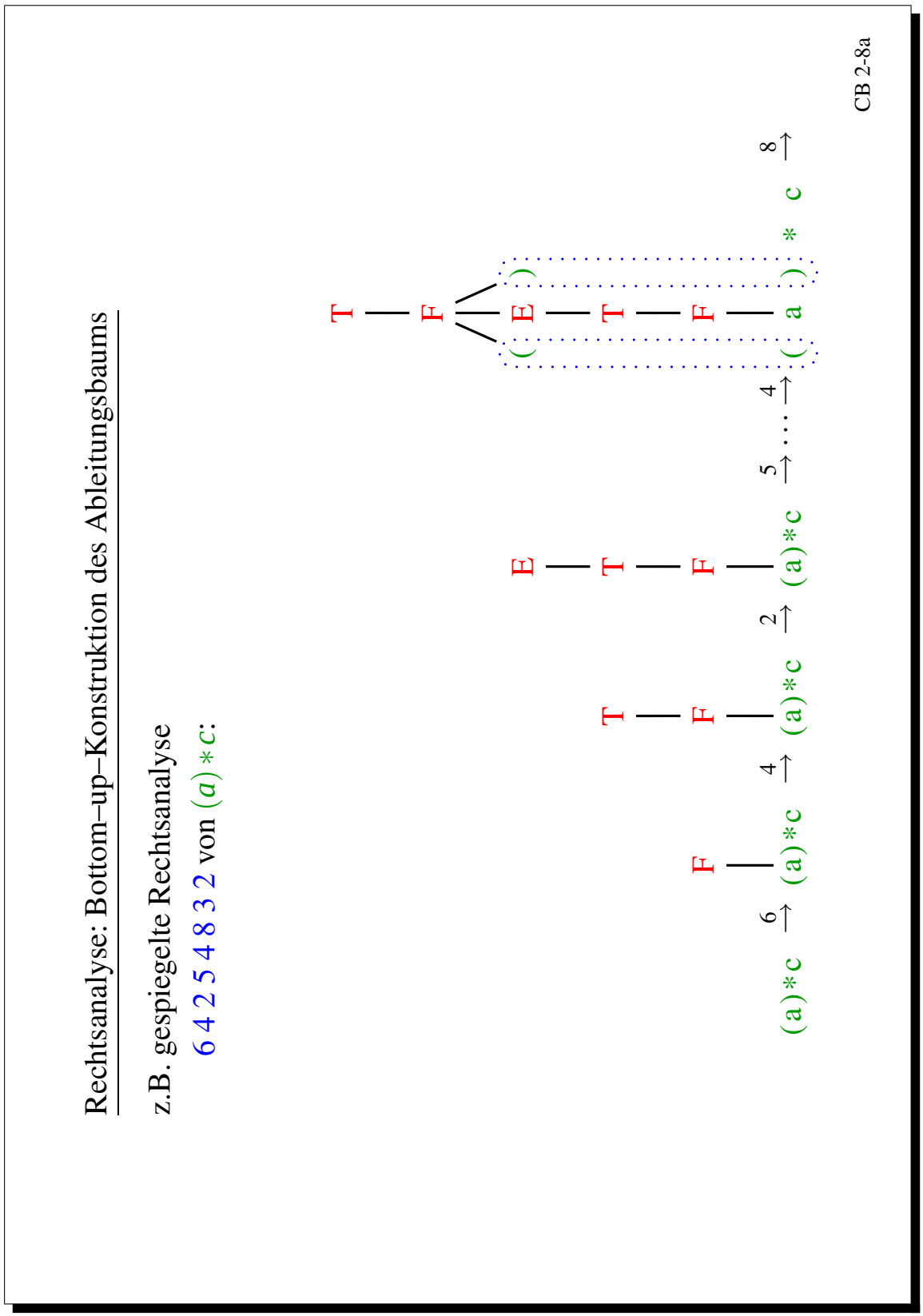


Abbildung 2.12: Rechtsanalyse I

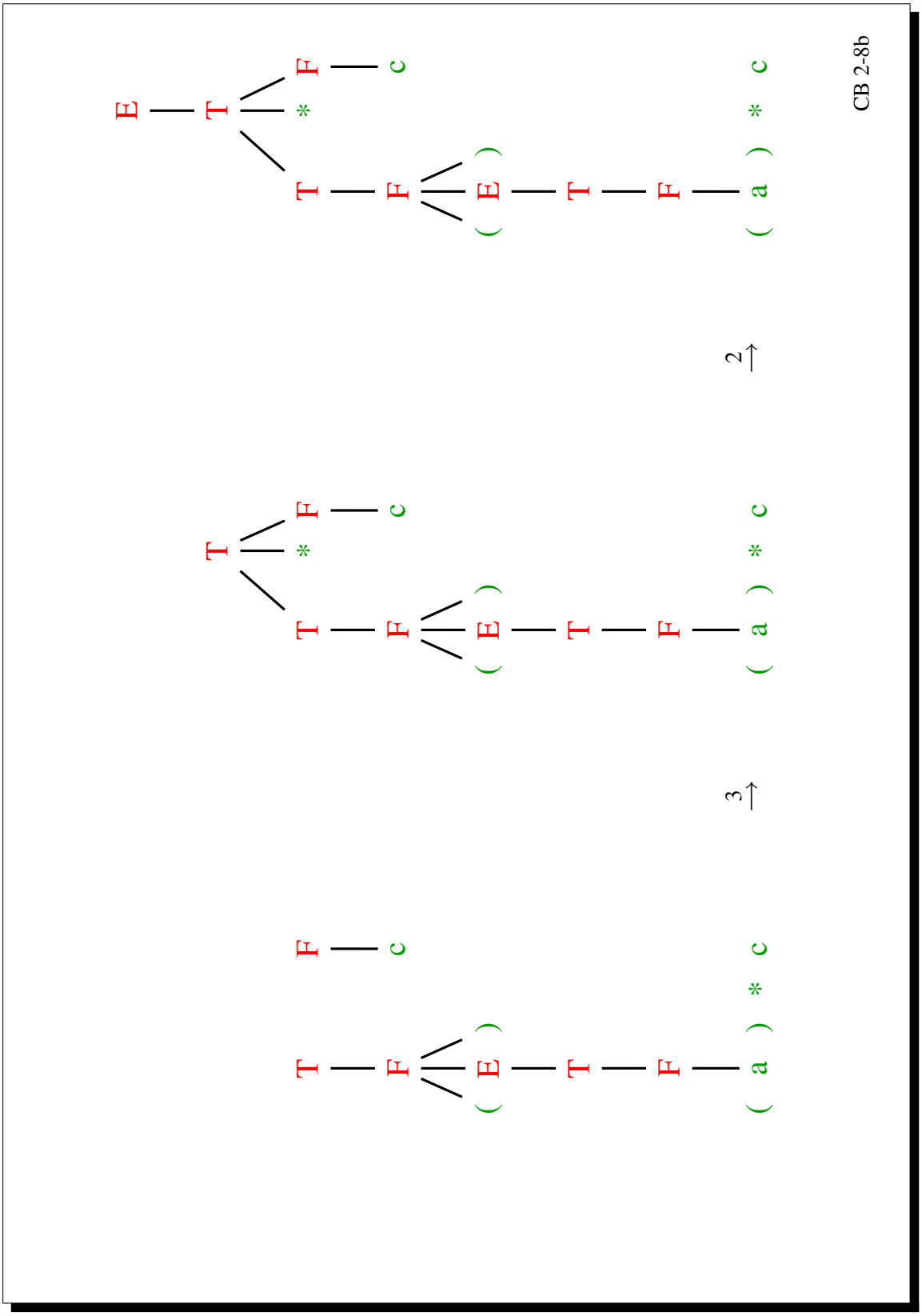
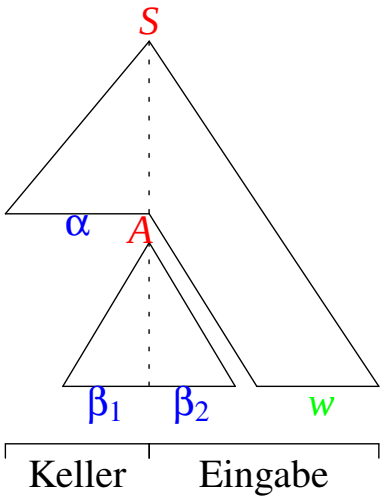


Abbildung 2.13: Rechtsanalyse II

Abbildung 2.14: Bottom-Up-Analyseautomat von G_{AE}

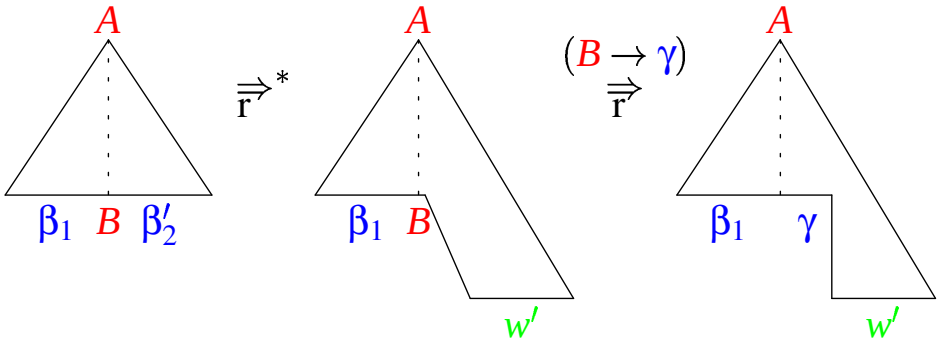
LR(0)–Analyse und Ableitungsbäume

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 2.15: $LR(0)$-Analyse und Ableitungsbäume

LR(0)–Informationen von G

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

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

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

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

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

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

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

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

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

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

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

CB 2-11

Abbildung 2.16: $LR(0)$-Informationen von G

goto– und **action**–Funktion zu G

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

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	-	-	-	-	-

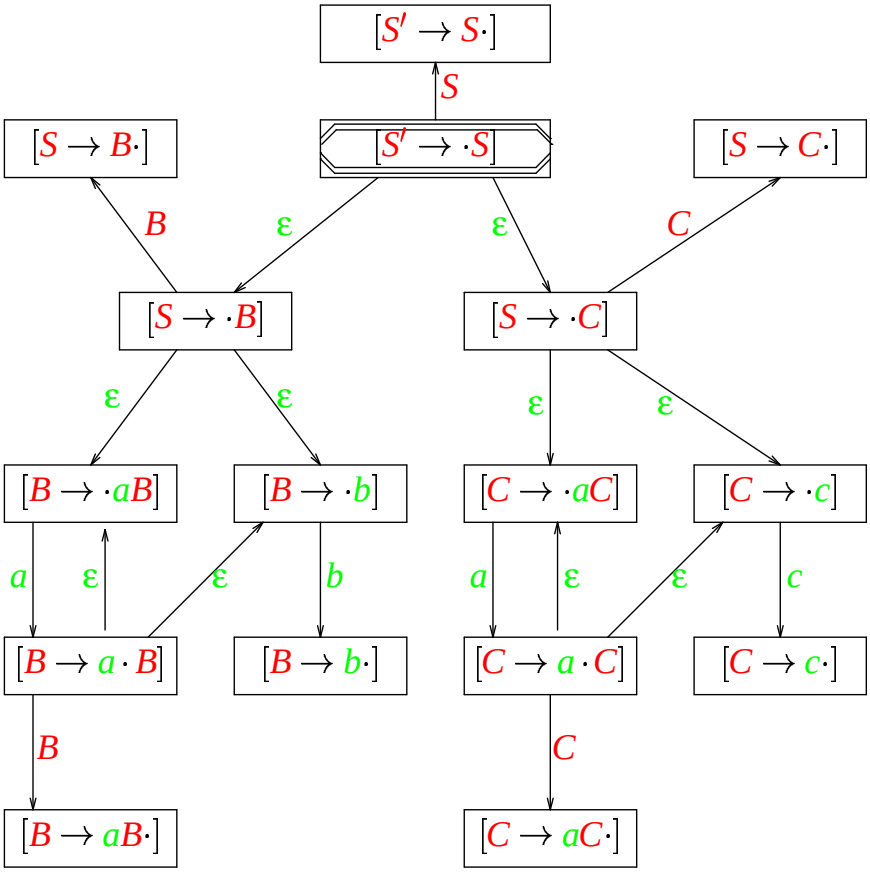
LR(0)(G)	**act**	**goto**					
		S	B	C	a	b	c
I_0	shift	I_1	I_2	I_3	I_4	I_5	I_6
I_1	accept						
I_2	red 1						
I_3	red 2						
I_4	shift		I_7	I_8	I_4	I_5	I_6
I_5	red 4						
I_6	red 6						
I_7	red 3						
I_8	red 5						
I_9	error						

CB 2-12

Abbildung 2.17: goto- und action-Funktion zu G

Berechnung von **LR(0)**(G) und **goto**
durch Potenzmengenkonstruktion

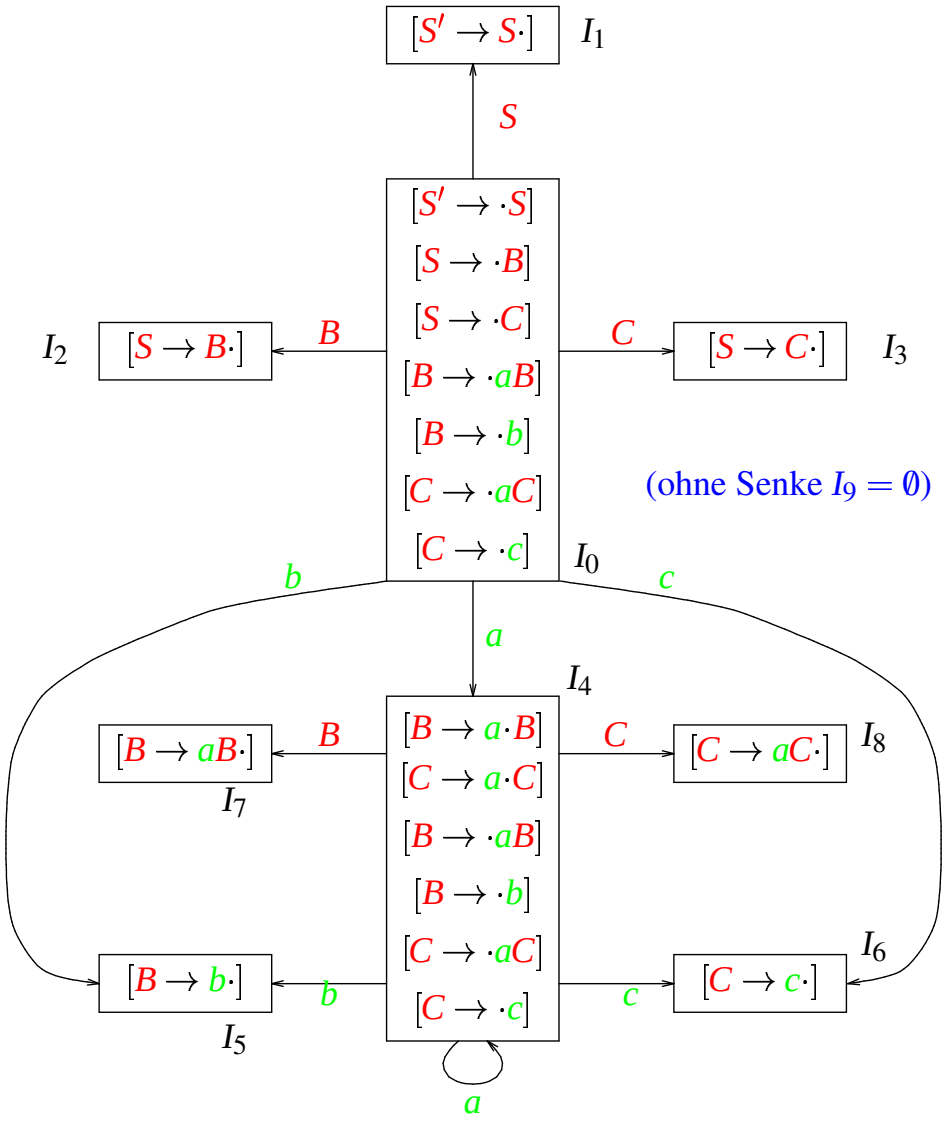
G: $S' \to S$ (0) $S \to B \mid C$ (1/2)
$B \to aB \mid b$ (3/4) $C \to aC \mid c$ (5/6)



CB 2-13

Abbildung 2.18: Potenzmengenkonstruktion I

Potenzmengenkonstruktion (Fortsetzung)



CB 2-14

Abbildung 2.19: Potenzmengenkonstruktion II

SLR(1)–Analyse

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

LR(0)–Informationen zu G_1:

I_0: $E' \to \cdot E$, $E \to \cdot E+T$, $E \to \cdot T$, $T \to \cdot T*F$, $T \to \cdot F$, $F \to \cdot(E)$, $F \to \cdot a$

I_1: $E' \to E\cdot$, $E \to E \cdot +T$ ↯

I_2: $E \to T\cdot$, $T \to T \cdot *F$ ↯

I_3: $T \to F\cdot$

I_4: $F \to (\cdot E)$, $E \to \cdot E+T$, $E \to \cdot T$, $T \to \cdot T*F$, $T \to \cdot F$, $F \to \cdot(E)$, $F \to \cdot a$

I_5: $F \to a\cdot$

I_6: $E \to E+\cdot T$, $T \to \cdot T*F$, $T \to \cdot F$, $F \to \cdot(E)$, $F \to \cdot a$

I_7: $T \to T*\cdot F$, $F \to \cdot(E)$, $F \to \cdot a$

I_8: $F \to (E\cdot)$, $E \to E \cdot +T$

I_9: $E \to E+T\cdot$, $T \to T \cdot *F$ ↯

I_{10}: $T \to T*F\cdot$

I_{11}: $F \to (E)\cdot$

CB 2-15

Abbildung 2.20: $SLR(1)$-Analyse von G_{AE}

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

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 2.21: $SLR(1)$-Analysetabelle

Motivation **LR(1)**–Analyse

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

LR(0)–Informationen zu G_2:

I_0: $[S' \to \cdot S]$, $[S \to \cdot L{=}R]$, $[S \to \cdot R]$, $[L \to \cdot {*}R]$, $[L \to \cdot a]$, $[R \to \cdot L]$

I_1: $[S' \to S\cdot]$

I_2: $[S \to L \cdot {=}R]$, $[R \to L\cdot]$ ↯

I_3: $[S \to R\cdot]$

I_4: $[L \to {*} \cdot R]$, $[R \to \cdot L]$, $[L \to \cdot {*}R]$, $[L \to \cdot a]$

I_5: $[L \to a\cdot]$

I_6: $[S \to L{=} \cdot R]$, $[R \to \cdot L]$, $[L \to \cdot {*}R]$, $[L \to \cdot a]$

I_7: $[L \to {*}R\cdot]$

I_8: $[R \to L\cdot]$

I_9: $[S \to L{=}R\cdot]$

Der Konflikt in I_2 ist mit der SLR–Methode nicht lösbar, da

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

CB 2-17

Abbildung 2.22: Motiviation $LR(1)$-Analyse

LR(1)–Informationen zu G_2:

$I_0^{(1)}$: $[S' \to \cdot S \quad ,\$]$ $[S \to \cdot L{=}R,\$]$ $[S \to \cdot R \quad ,\$]$
$[L \to \cdot{*}R \quad ,=]$ $[L \to \cdot a \quad ,=]$ $[R \to \cdot L \quad ,\$]$
$[L \to \cdot{*}R \quad ,\$]$ $[L \to \cdot a \quad ,\$]$

$I_1^{(1)}$: $[S' \to S\cdot \quad ,\$]$

$I_2^{(1)}$: $[S \to L \cdot {=}R,\$]$ $[R \to L\cdot \quad ,\$]$ $\left\{\begin{array}{l}\text{shift} \quad ,\text{bei } = \\ \text{reduce} \ ,\text{bei } \$\end{array}\right.$

$I_3^{(1)}$: $[S \to R\cdot \quad ,\$]$

$I_4^{(1)}$: $[L \to {*}\cdot R \quad ,=]$ $[R \to \cdot L \quad ,=]$ $[L \to \cdot{*}R,=]$
$[L \to \cdot a \quad ,=]$ $[L \to {*}\cdot R \quad ,\$]$ $[R \to \cdot L \quad ,\$]$
$[L \to \cdot{*}R \quad ,\$]$ $[L \to \cdot a \quad ,\$]$

$I_5^{(1)}$: $[L \to a\cdot \quad ,=]$ $[L \to a\cdot \quad ,\$]$

$I_6^{(1)}$: $[S \to L{=}\cdot R,\$]$ $[R \to \cdot L \quad ,\$]$ $[L \to \cdot{*}R,\$]$
$[L \to \cdot a \quad ,\$]$

$I_7^{(1)}$: $[L \to {*}R\cdot \quad ,=]$ $[L \to {*}R\cdot \quad ,\$]$

$I_8^{(1)}$: $[R \to L\cdot \quad ,=]$ $[R \to L\cdot \quad ,\$]$

$I_9^{(1)}$: $[S \to L{=}R\cdot \ ,\$]$

$I_{10}^{(1)}$: $[R \to L\cdot \quad ,\$]$

$I_{11}^{(1)}$: $[L \to {*}\cdot R \quad ,\$]$ $[R \to \cdot L \quad ,\$]$ $[L \to \cdot{*}R,\$]$
$[L \to \cdot a \quad ,\$]$

$I_{12}^{(1)}$: $[L \to a\cdot \quad ,\$]$

$I_{13}^{(1)}$: $[L \to {*}R\cdot \quad ,\$]$

CB 2-18

Abbildung 2.23: $LR(1)$-Informationen

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

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 2.24: $LR(1)$-Analysetabelle

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

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

CB 2-20

Abbildung 2.25: $LR(1)$-Analyseautomat

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

LALR(1)–Analysetabelle zu G_2:

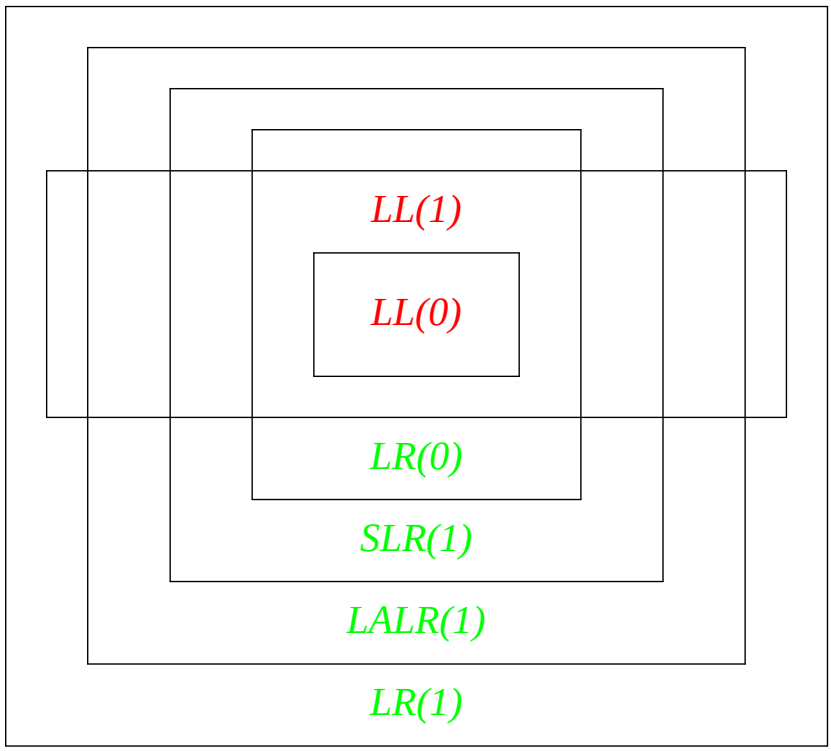
	action				**goto**					
	$*$	$=$	a	\$	$*$	$=$	a	S	L	R
$I_0^{(1/0)}$	shift		shift		$I_4^{(1/0)}$		$I_5^{(1/0)}$	$I_1^{(1/0)}$	$I_2^{(1/0)}$	$I_3^{(1/0)}$
$I_1^{(1/0)}$				acc						
$I_2^{(1/0)}$		shift		red 5		$I_6^{(1/0)}$				
$I_3^{(1/0)}$				red 2						
$I_4^{(1/0)}$	shift		shift		$I_4^{(1/0)}$		$I_5^{(1/0)}$		$I_8^{(1/0)}$	$I_7^{(1/0)}$
$I_5^{(1/0)}$		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 2.26: $LALR(1)$-Analysetabelle

Übersicht der Grammatikklassen

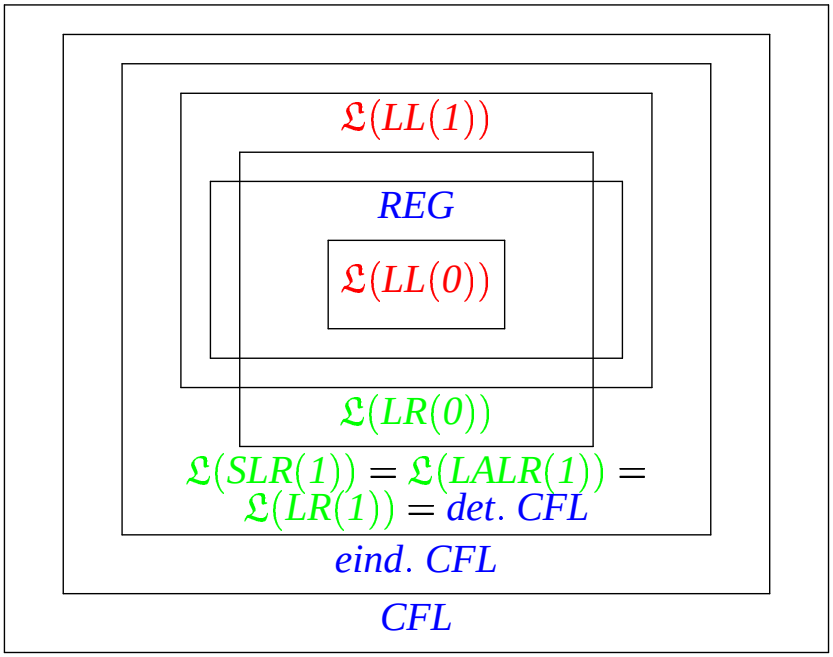


- $LL(0) \subsetneqq LL(1) \subsetneqq LR(1)$
- $LR(0) \subsetneqq SLR(1) \subsetneqq LALR(1) \subsetneqq LR(1)$

CB 2-22

Abbildung 2.27: Übersicht der Grammatikklassen

Übersicht der Sprachklassen



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

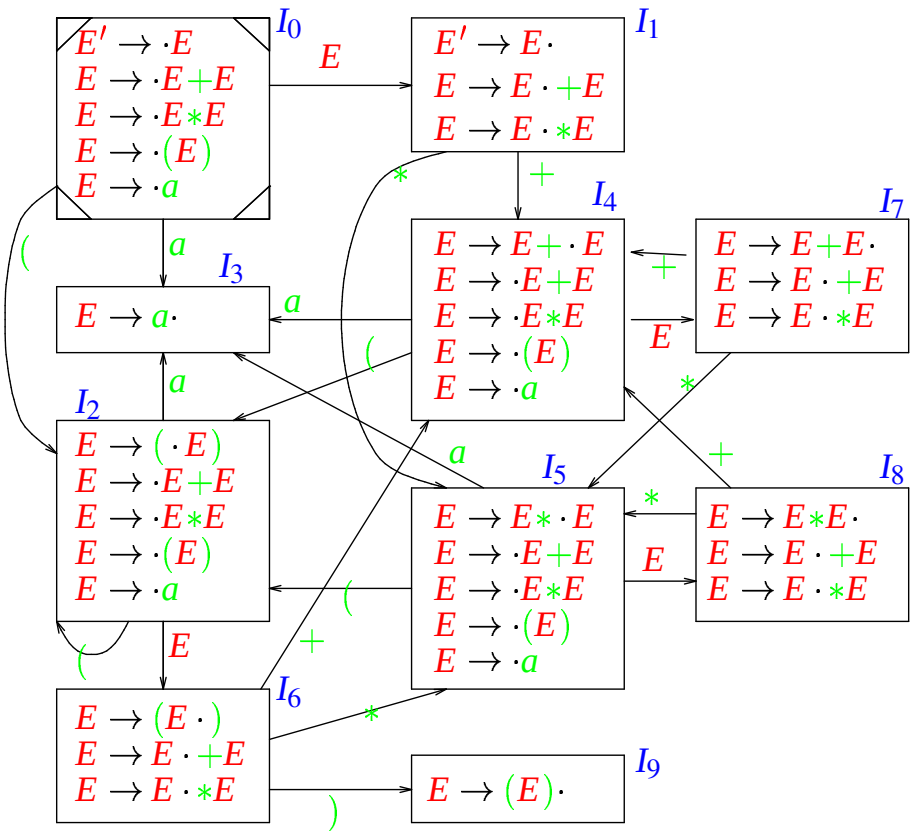
- $\mathfrak{L}(LL(0)) \subsetneqq REG \subsetneqq \mathfrak{L}(LL(1)) \subsetneqq \mathfrak{L}(SLR(1))$
- $\mathfrak{L}(LR(0)) \subsetneqq \mathfrak{L}(SLR(1)) = \mathfrak{L}(LALR(1))$
  $= \mathfrak{L}(LR(1)) = det.\ CFL \subsetneqq eind.\ CFL \subsetneqq CFL$
- $\mathfrak{L}(LR(0)) \not\subseteq \left\{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

CB 2-23

Abbildung 2.28: 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 2.29: $LR(0)$-Informationen von 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		SLR
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		$*\cdot> +$
I_8		red 2	red 2		red 2	red 2		linksass.
I_9		red 3	red 3		red 3	red 3		

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

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

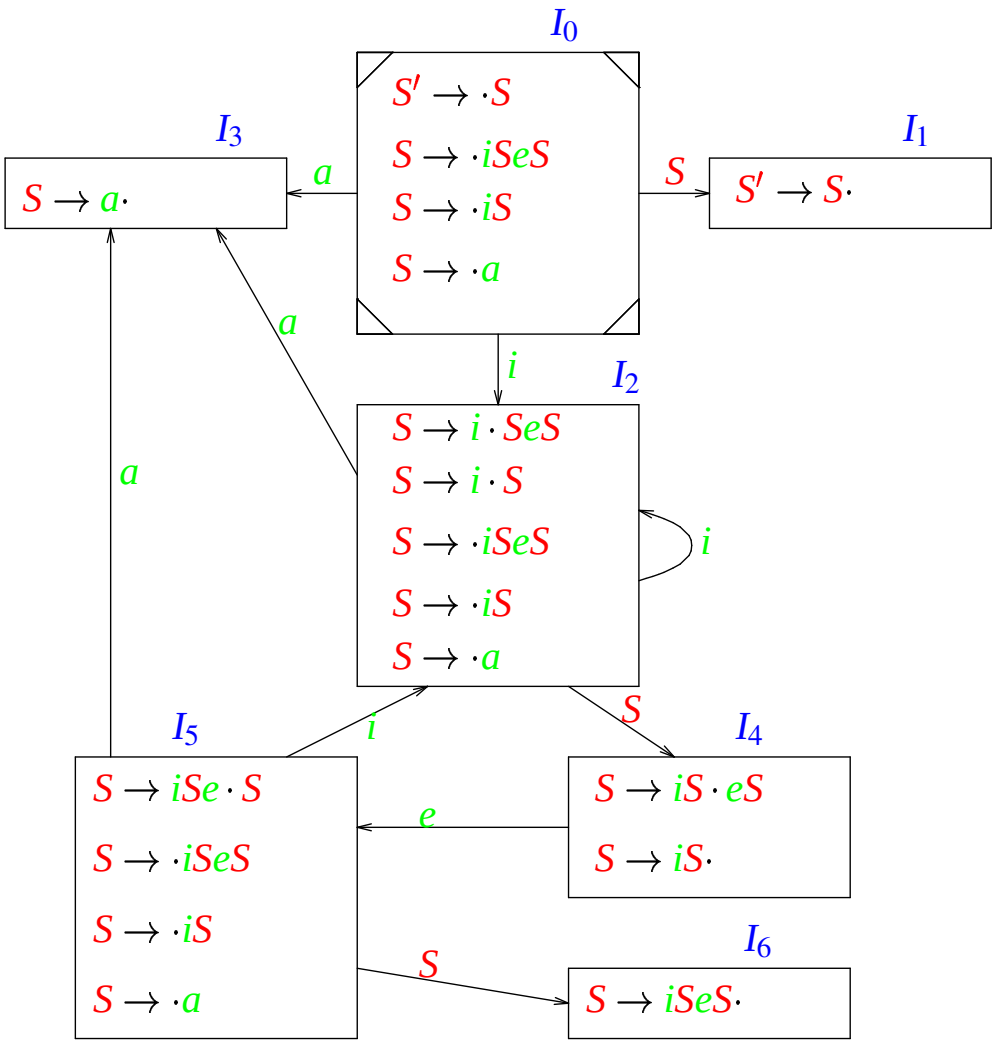
(Tabellenkompression)

CB 2-25

Abbildung 2.30: Analysetable von 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 2.31: $LR(0)$-Informationen von G_4

Kapitel 3

Semantische Analyse, Attributgrammatiken

Ergebnis der syntaktischen Analyse: Ableitungsbaum
kontextabhängige Eigenschaften:

- Deklariertheit von Bezeichnern ($\{ww|w \in \Sigma^*\} \notin CFL$)
- Typinformationen ind 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: 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.

Beispiel: (Binärzahlen) G_B:

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

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 Nichtterminalsymbol.

Ziel: Bestimmung des Zahlwertes

- Attributwerte von v: rationale Zahlen
- Attributwerte von l: natürliche Zahlen

2. Attributisierung von G_B mit synthetischen und inheriten Attributen:
Zusätzliches *inherites* Attribut für Bits und Listen: p („position“)

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

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

$\underline{\text{syn}}: \quad \chi \to \mathfrak{p}(\underline{\text{Syn}}) \quad$ mit $\underline{\text{syn}}(X) \quad = \underline{\text{att}}(X) \cap \underline{\text{Syn}}$

und $\underline{\text{inh}}: \quad \chi \to \mathfrak{p}(\underline{\text{Inh}}) \quad$ mit $\underline{\text{inh}}(X) \quad = \underline{\text{att}}(X) \cap \underline{\text{Inh}}$

Eine Regel $\Pi = X_0 \to X_1 \ldots X_r \in P$ 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

$\underline{\text{IVar}}_\Pi := \quad \{\alpha.i | \quad (i = 0 \text{ und } \alpha \in \underline{\text{syn}}(X_0)) \text{ oder}$

$\qquad\qquad\qquad\qquad (1 \leq i \leq r \text{ und } \alpha \in \underline{\text{inh}}(X_i))\}$

und $\underline{\text{OVar}}_\Pi := \quad \underline{\text{Var}}_\Pi \backslash \underline{\text{IVar}}_\Pi$ der *Innenvariablen* und *Außenvariablen*

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

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

mit $\alpha.i \in \underline{\text{IVar}}_\Pi, \alpha_1.i_1, \ldots, \alpha_n.i_n \in \underline{\text{OVar}}_\Pi$, $f : A^{\alpha_1} \times \ldots \times A^{\alpha_n} \to A^\alpha$ und $n \in \mathbb{N}$.

Sei E_Π eine Menge von Attributgleichungen, in der jede Innenvariable genau einmal vorkommt.

Dann heißt $\mathfrak{A} =< G, (E_\Pi | \Pi \in P) >$ eine *Attributgrammatik*: $\mathfrak{A} \in AG$.

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

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

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

der aktuellen *Attributvariablen* von t.

Wird an einem inneren Knoten (kein Blatt) $k_0 \in \underline{\text{Kn}}(t)$ die Regel $\Pi = X_0 \to X_1 \ldots X_r$ angewandt und sind $k_1, \ldots, k_r \in \underline{\text{Kn}}(t)$ 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 $\alpha.k = \ldots$

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 $A \to uBv$ (mit B an der i-ten Position) und $B \to w$. Ferner sei $\alpha \in \underline{\text{syn}}(B)$ und $\beta \in \underline{\text{inh}}(B)$. Als Attributgleichungen erhalten wir $\beta.i = f(\alpha.i), \alpha.0 = g(\beta.0)$.

Der Ableitungsbaum t ergibt in E_t die *zirkuläre Abhängigkeit*

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

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

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

Folgerung: Zirkularitäten können auftreten, sie sind unerwünscht wegen Mehrdeutigkeiten.

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

3.1.2 Ein Zirkularitätstest für Attributgrammatiken

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

$\underline{\text{OKno}}(DG_\Pi) := \{\alpha.0 | \alpha \in \underline{\text{att}}(X_0)\}$ (Oberknoten)
$\underline{\text{UKno}}(DG_\Pi) := \{\alpha.i | \alpha \in \underline{\text{att}}(X_i), 1 \le i \le r\}$ (Unterknoten)
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: $Var_\Pi = \underline{\text{UKno}}(DG_\Pi) \cup \underline{\text{OKno}}(DG_\Pi) = \underline{\text{OVar}}_\Pi \cup \underline{\text{IVar}}_\Pi$

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.

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 es einen 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 getreent behandelt werden, weil sonst unzulässige Abhängigkeiten

entstehen können.

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

$D(A,t) \quad := \{(\alpha,\alpha') | \alpha \xrightarrow{A} \alpha' \text{ in } t\}$
$\mathscr{D}(A,t) \quad := \{D(A,t) | t \text{ Abbleitungsbaum mit Wurzel } A\}$
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. $\Pi = A \to w \curvearrowright D[\Pi_i]$
2. $\Pi = A \to w_0 A_1 w_1 \ldots A_r w_r, D_i \in \mathscr{D}(A_i) \curvearrowright D[\Pi, D_1, \ldots, D_r] \in \mathscr{D}(A)$

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.

Folgerung: (Zirkularitätstest): $\mathfrak{A} \in AG$ ist zirkulär $\curvearrowright$ es gibt $\Pi = A_0 \to w_0 A_1 \ldots A_r w_r, a.k = \underline{\text{UKno}}$ und $D_i \in \mathscr{D}(A_i) (1 \le i \le r)$, 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.

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

3.1.3 Stark-nichtzirkuläre Grammatiken

Keine Trennung der Abhängigkeitsmengen verschiedener Ableitungsbäume.

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

Hinzureichendes 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 Funktionen für $(E_\Pi | \Pi \in P)$ aufstellen.

   Idee: Jedem synthetischen Attribut einer Variablen wird eine Funktion zugeordnet mit ihrem inheriten Attribut als Parameter.
4. Spezialfall: SAG, LAG

3.2 S-Attributgrammatiken

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

Folgerung: Bottom-Up-Berechnung der Attributwerte.

Idee: Durchführung während der Bottom-Up-Syntaxanalyse, zwischenspeichern von Attributwerten im Analysekeller.
Beispiel: $A \to BaD$

I_1	$v3$
I_7	$v2$
I_3	$v1$

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

Beispiel: Stackcode für arithmetische Ausdrücke

$$\begin{array}{lcll} E & \to & (E+E)| & c.0 = c.2; c.4; ADD \\ & | & (E*E)| & c.0 = c.2; c.4; MULT \\ & | & id| & c.0 = LOADc.1 \\ & | & num & c.0 = LITc.1 \end{array}$$

Werte von $c.1$: 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:

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

repräsentiert Operationssymbol F_Π vom Typ $A_1 \times \ldots \times A_r \to A_0$ (A_r 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.

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
$f^{(n)} \in \Omega =$	$\{\underline{\text{assign}}^{(1)}, \underline{\text{seq}}^{(2)}, \underline{\text{cond}}^{(2)}, \underline{\text{less}}^{(2)}, \underline{\text{plus}}^{(2)}, \underline{\text{id}}^{(0)}, \underline{\text{num}}(0)$
$k \in \underline{\text{Kno}} :=$	$\{(a, f, a_1, \ldots, a_n) \mid a, a_i \in \underline{\text{Adr}}, f \in \Omega^{(n)}\}$
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

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

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

mit neuer Adresse a und

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

Beispiel: Typberechnung für arithmetische Ausdrücke

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

3.3 L-Attributgrammatiken

Definition: $\mathfrak{A} =< G, (E_\Pi | \Pi \in P) > \in AG$ 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 $\mathfrak{A} =< G, (E_\Pi | \Pi \in P) > \in LAG$ 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: $\bigcup_{\Pi \in P} (\underline{\text{LR}(0)}_\Pi(G) \times \underline{\text{Val}}_\Pi)$ mit $\underline{\text{LR}(0)}_\Pi(G) := \{[A \to \alpha.\beta] | \Pi = A \to \alpha\beta\}$ und $\underline{\text{Val}}_\Pi = \{\underline{\text{val}}_\Pi | \underline{\text{val}}_\Pi : \underline{\text{Var}}_\Pi - \to$ „Attributwerte" $\}$

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

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

   $\Rightarrow$

$\vdots$	
$[A \to X_1 \ldots X_{i_1} \cdot B\gamma]$	$\underline{\text{val}}$
$[B \to .\beta]$	$\underline{\text{val}}'$

   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 $A \to X_1 \ldots X_{i-1} B\gamma$.

   Sei $\alpha \in \underline{\text{inh}}(B)$ und $\alpha.i = t \in E_{A \to X_1 \ldots \gamma}$, so $\underline{\text{val}}'(\alpha.0) = \hat{t}$ (Wert von t bezüglich $\underline{\text{val}}$).

2. „match"-Schritt:

$\vdots$	
$[A \to \gamma \cdot a\gamma']$	$\underline{\text{val}}$

   $\Rightarrow$

$\vdots$	
$[A \to \gamma a \cdot \gamma']$	$\underline{\text{val}}$

   falls nächstes Eingabesymbol. Eingabekopf rückt vor

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

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

   $\Rightarrow$

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

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

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

3.3.2 Anwendung von LAG

Überprüfung der Deklariertheit von Bezeichnern. G sei gegeben durch:

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

Attribute

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

Attributgleichungen

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

3. Semantische Analyse

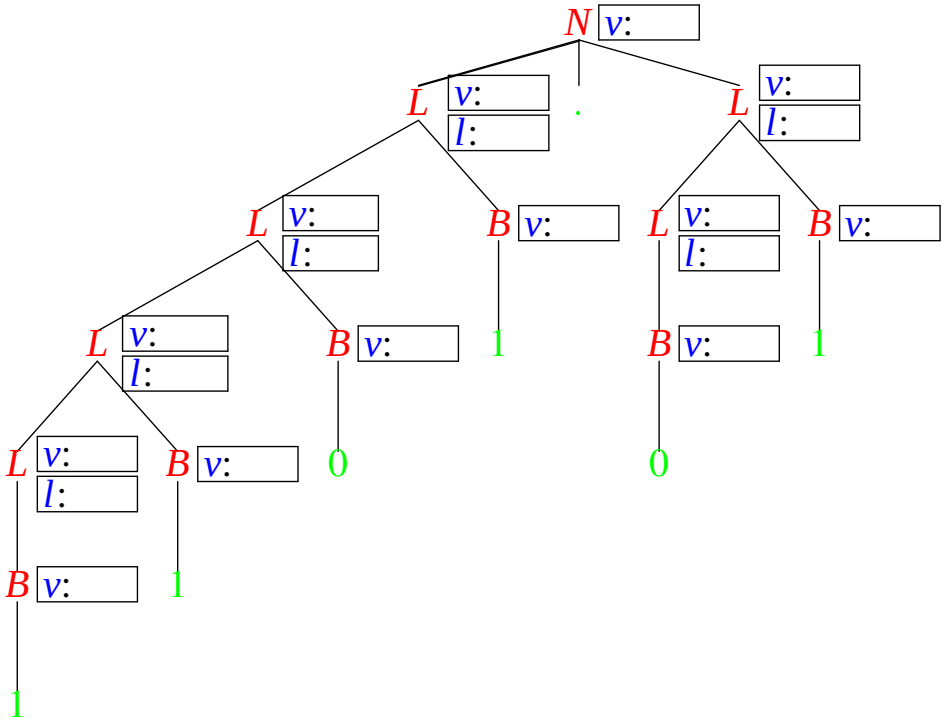
Bsp.: (Binärzahlen von Knuth)

G_B :

(1) $B \to 0$ (2) $B \to 1$

(3) $L \to B$ (4) $L \to LB$

(5) $N \to L$ (6) $N \to L.L$



CB 3-1

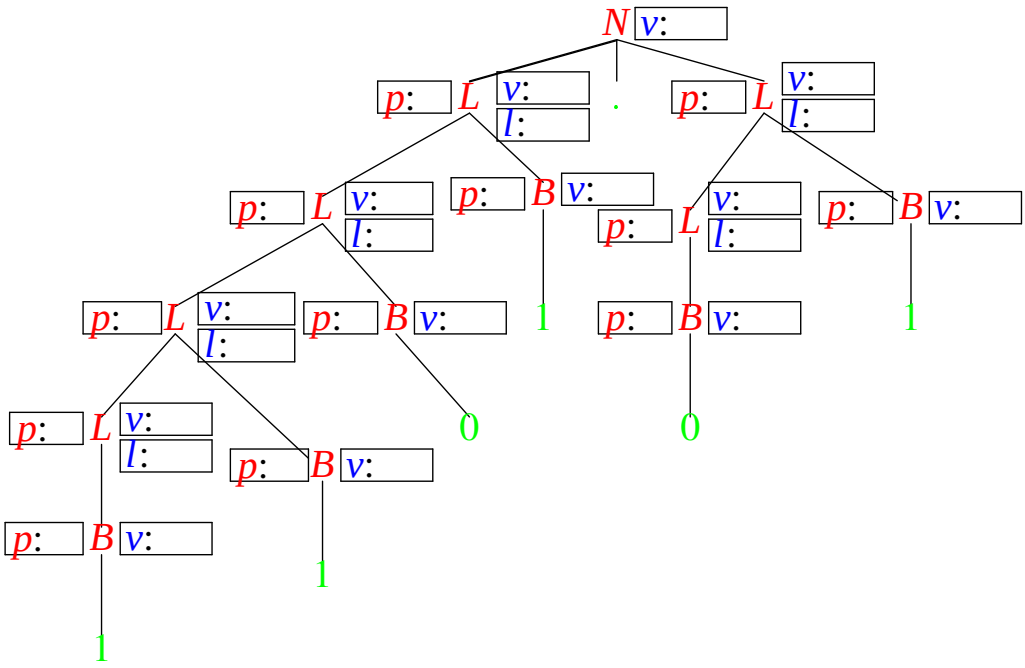
Abbildung 3.1: Binärzahlen von Knuth

Attributierung von G_B mit synthetischen und inheriten Attributen:

G_B : (1) $B \to 0$ (2) $B \to 1$

(3) $L \to B$ (4) $L \to LB$

(5) $N \to L$ (6) $N \to L.L$



CB 3-2

Abbildung 3.2: Attributierung von G_B mit synthetischen und inheriten Attributen

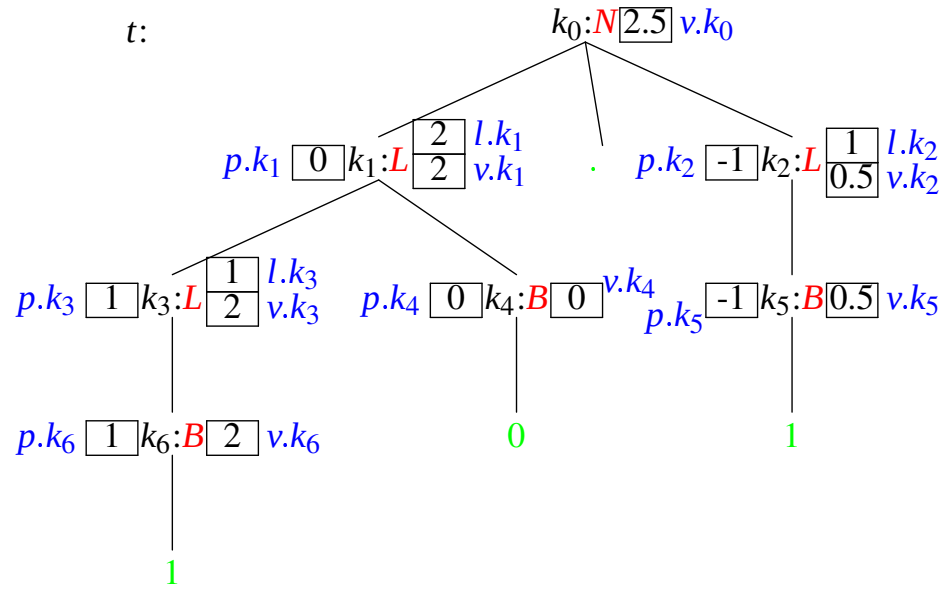
Attributgleichungen:

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

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

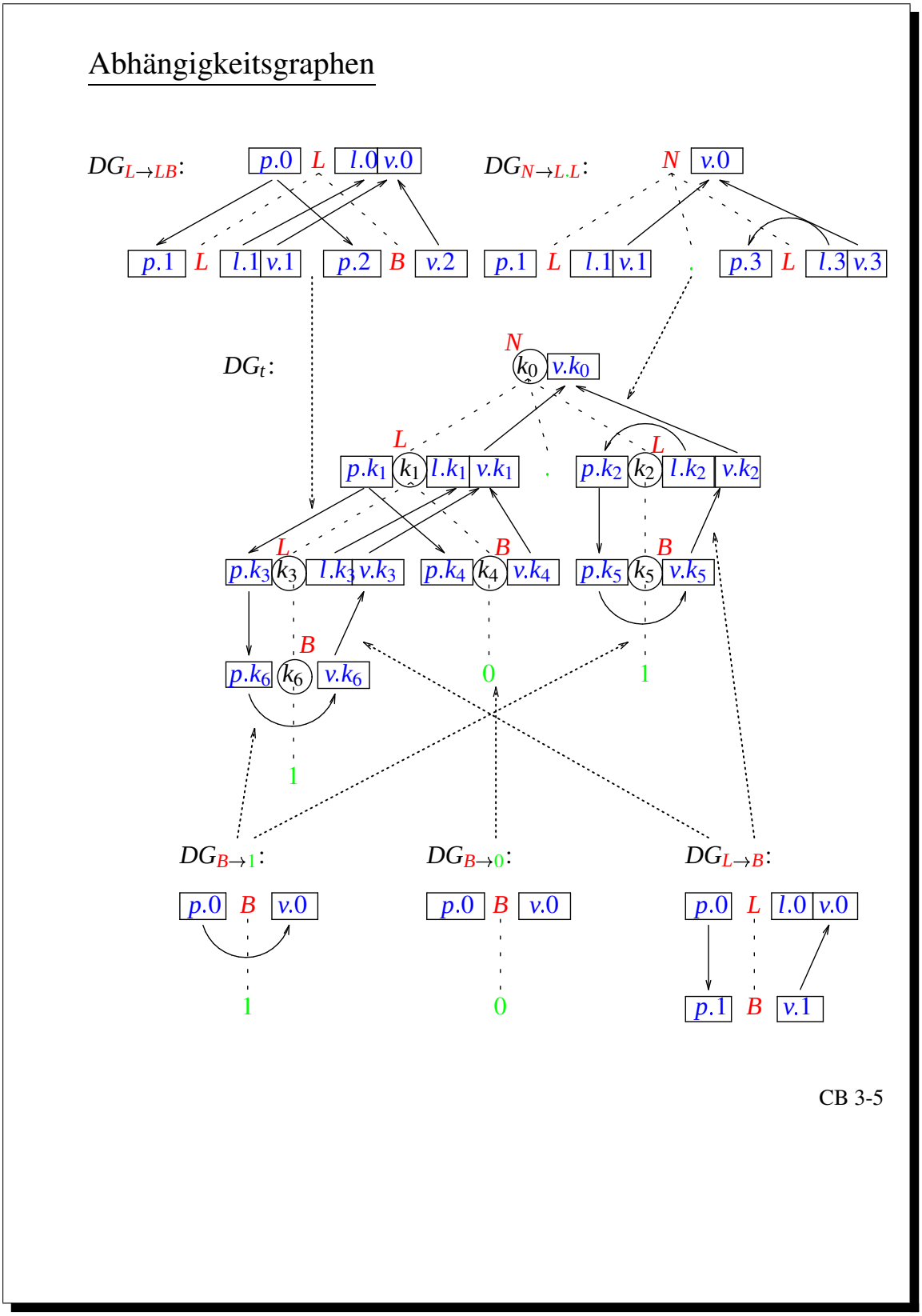


Abbildung 3.5: Abhängigkeitsgraphen

Konkrete und abstrakte Syntax (Beispiel)

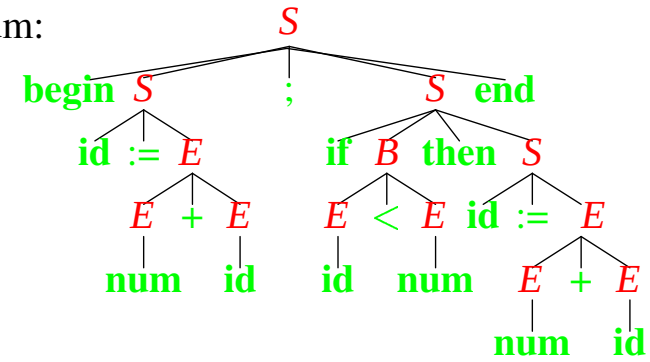
G :

$S \rightarrow$ **id** $:= E \mid$ **begin** S ; S **end** $\mid$ **if** B **then** S

$B \rightarrow E < E$

$E \rightarrow E + E \mid$ **id** $\mid$ **num**

Ableitungsbaum:



Operationsalphabet:

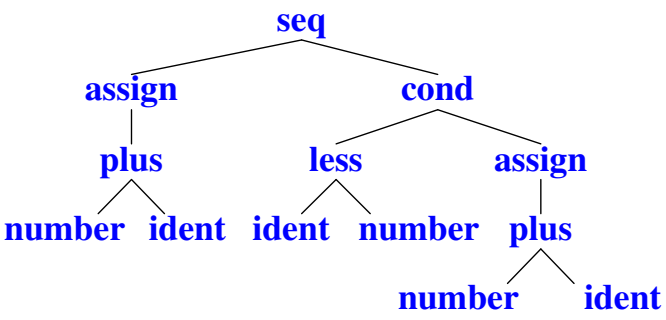
assign : $E \rightarrow S$ **less** : $E \times E \rightarrow B$

seq : $S \times S \rightarrow S$ **plus** : $E \times E \rightarrow E$

cond : $B \times S \rightarrow S$ **ident** : $\rightarrow E$

number : $\rightarrow E$

Abstrakter Syntaxbaum:



CB 3-6

Abbildung 3.6: konkrete und abstrakte Syntax

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 → Portabilität, Transparenz, Codeoptimierung.

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
- 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

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 Deklaration 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 $ca, dif, loc \in N$), RET
- Transportbefehle: LOAD(dif, off), STORE(dif, off), mit $dif, off \in \mathbb{N}$ und LIT z mit $z \in \mathbb{Z}$

Befehlssemantik
$Z = (m,d,p) \in ZR$ (Z: Zustand der abstrakten Maschine)
$[\![B]\!] : ZR \dashrightarrow ZR$ für jeden AM-Befehl B

$$[\![ADD]\!](m, d : Z_1 : Z_2, p) := (m+1, d : (Z_2 + Z_1), p)$$

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

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

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

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

p zerfällt in Aktivierungsblöcke („Frames“) der Form

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

mit *sv*: statischer Verweis, *dv*: dynamischer Verweis, *ra*: Rücksprungadresse nach Beendigung des Prozeduraufrufs und l_i: lokale Variable eines Prozessaufrufs.

sv zeigt auf den Aktivierungsblock der Deklarationsumgebung, *dv* zeigt auf den letzen Aktivierungsblock.

Berechnung des statischen Verweises

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*).

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

Beispiel: (Folie 4.7)

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

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

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

  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
$P \in \underline{\text{AM-Code}} := P = a_1 : B_1; \ldots; a_p : B_{pi}$ 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

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

4.1.3 Übersetzung von BPS-Programmen in AM-Code

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

Hilfsmittel: Symboltabelle

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

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

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

Aufbau der Symboltabelle

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

$\underline{\text{up}}(\Delta, st, a, l)$ 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

$P = \underline{\text{in/out}} I_1, \ldots, I_n, B\cdot$ 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 $st_{I/O}(I_j) = (\underline{\text{var}}, 0, j)$.
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

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

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

Blockübersetzung

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

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

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

Beachte:

- $st(I_j) = (\underline{\mathrm{proc}}, a_j, l+1)$, weil $\underline{bt}$ die Funktionen $\underline{dt}$ und $\underline{\mathrm{up}}$ 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

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

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

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

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

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

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

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

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

Striker 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: $\mathscr{M}[\![P]\!](z_1,\ldots,z_n) = (z'_1,\ldots,z'_i)\curvearrowright$

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

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.

Idee: Vererbung von Sprungzielen für boolesche Ergebnisse

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

(Adr: 1. freie Anfangsadresse, 2. true-Adresse, 3. false-Adresse).

mit geeigneter Modifikation der Übersetzung von Verzweigung und Iteration.

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

4.1.6 Prozeduren mit Parametern

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)

Semantik:

- 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) folgende neue Befehle:

CALL ca	$SP \leftarrow SP-1, <SP> \leftarrow IC+1, IC \leftarrow ca$
RET k	$IC \leftarrow <SP>, SP \leftarrow SP+k+1$
PUSH z	$SP \leftarrow SP-1, <SP> \leftarrow z$
PUSH FP **PUSH** $<FP>$ **PUSH** $<R+2>$	analog
POP FP	$FP \leftarrow <SP>, SP \leftarrow SP+1$
POP $<n>$	$Sn \leftarrow <SP>, SP \leftarrow SP+1$
SUB SP, n	$SP \leftarrow SP-n$
LOAD FP, SP	$FP \leftarrow SP$
LOAD $R, <n>$	
LOAD $R, <R+2>$	

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 $\underline{\text{trans}} : \underline{\text{BSP} - \text{Prog}} \rightarrow \underline{\text{AM} - \text{Code}}$ 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, 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.

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

Typkompatibilität: Polymorphie (Ad-hoc) von Operationssymbolen.

Beispiel: int $\times$ int $\rightarrow$ int, real $\times$ real $\rightarrow$ real, int ¡ real, *ldots*

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.
- (type, array, $\underbrace{Z_1, Z_2}_{\text{Grenzen}}$, $\underbrace{\quad I \quad}_{\text{Komponententyp}}$, $\underbrace{\quad n \quad}_{\text{Speicherbedarf}}$)
- (type, record, $\underbrace{I_1}_{\text{Selektor}}$, $\underbrace{I_1}_{\text{Komp.Typ}}$, $\underbrace{\sigma_1, \ldots, \sigma_n}_{\text{Offset}}$, $\underbrace{\quad s \quad}_{\text{Speicherbedarf}}$

- $st(I.I_j) = (I'_j, \sigma_j)$ 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

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

4.2.2 Dynamische Datenstrukturen

Variante Verbunde: Speicherbedarf für größte Variante vorgesehen. Denselben Bereich für verschiedene Varianten verwenden.

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. Erzeugung von Objekten nicht durch Deklaration mit statischer Adresse, sondern durch Anweisung, zum Beispiel new.

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:
if $NP- < SP > \leq EP$ then error(„stack overflow“)
else $NP := NP- < SP >$;
$<< SP-1 >> := NP$;
$SP := SP-2$;

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.

code$(new(p)) :=$ **LOAD** $adr(p)$;
LOAD $size(p)$;
NEW;

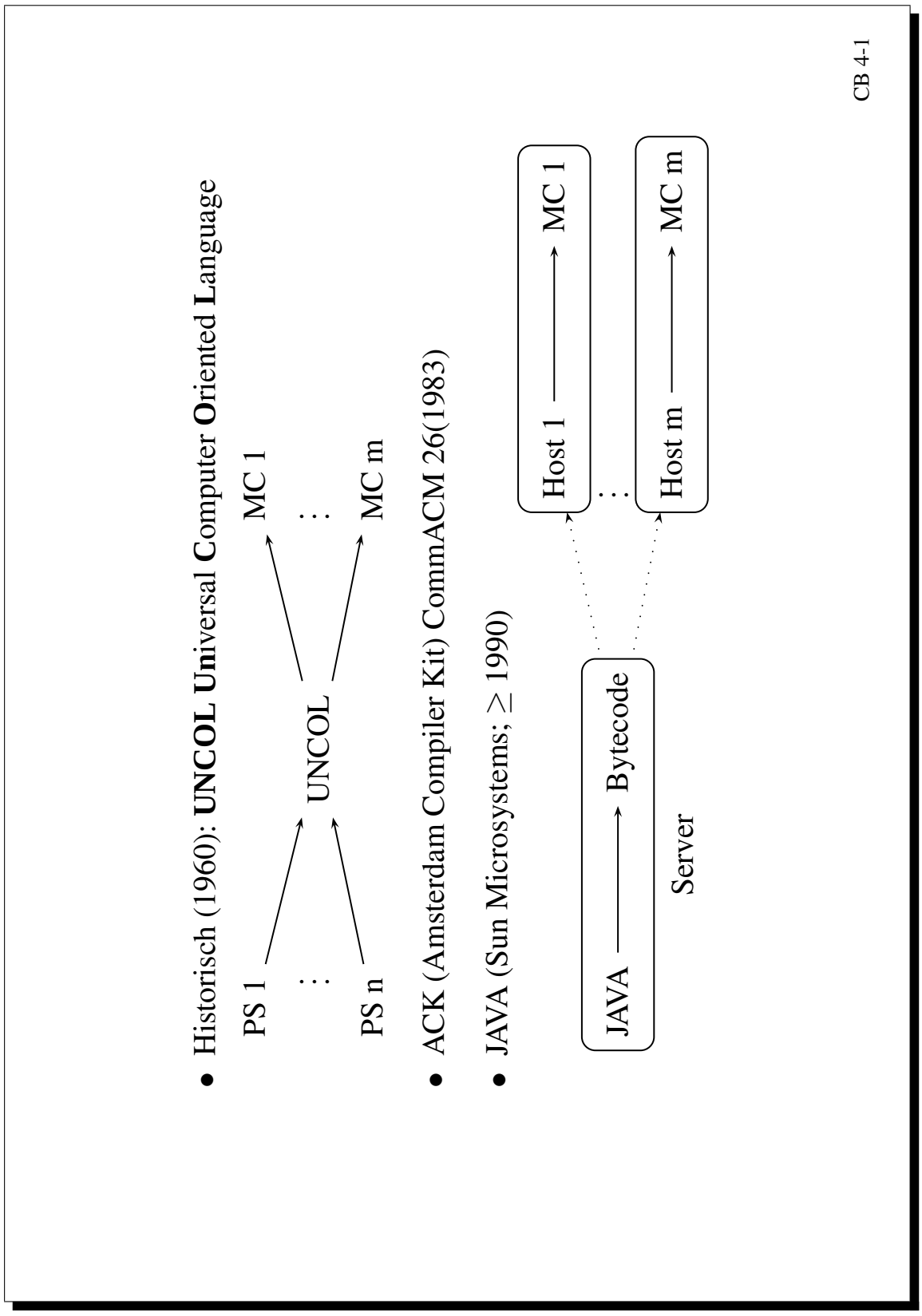


Abbildung 4.1: Historischer Überblick

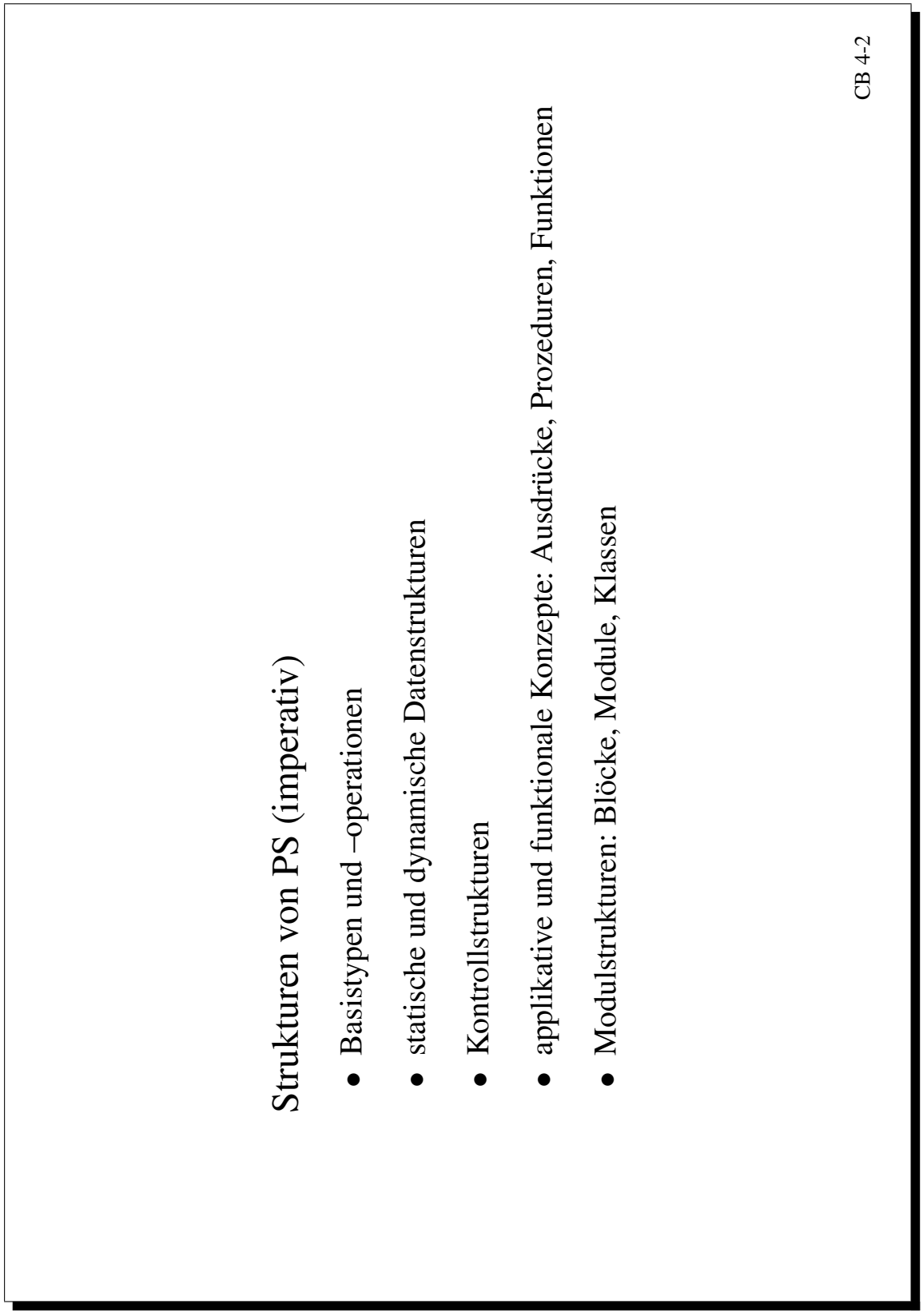


Abbildung 4.2: Strukturen von Programmiersprachen

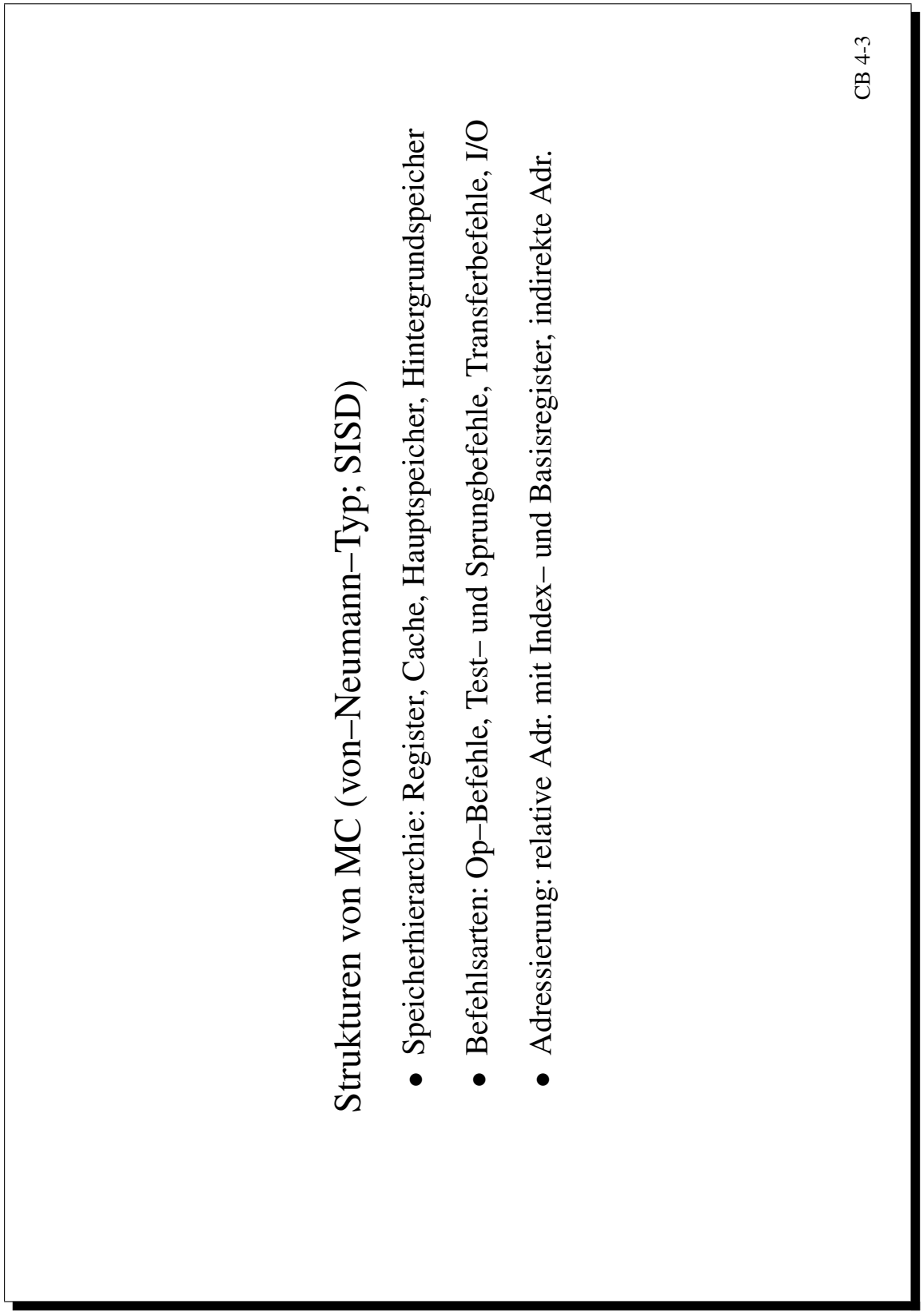


Abbildung 4.3: Strukturen von Maschinencode

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

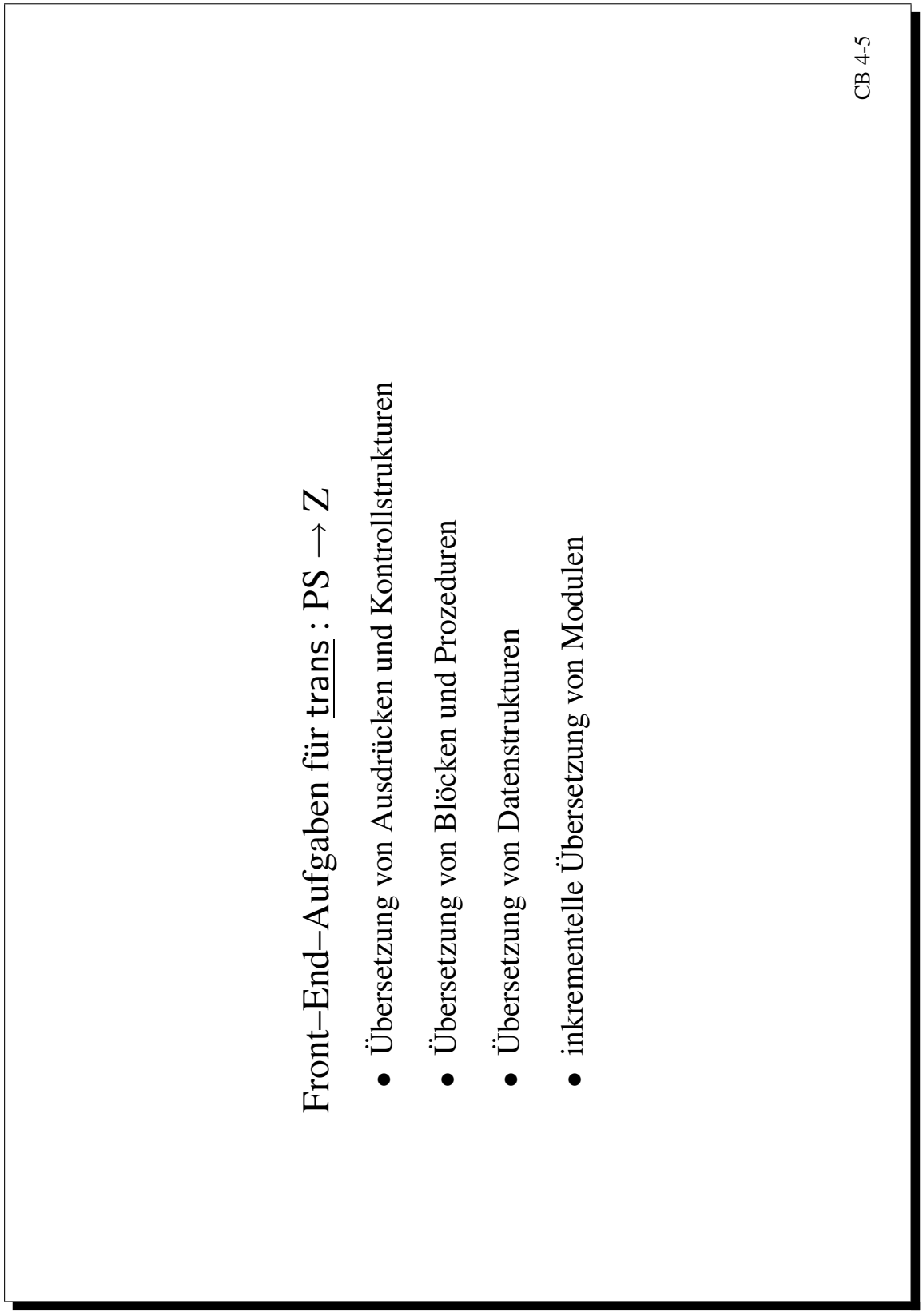


Abbildung 4.5: Aufgaben des Front-ends

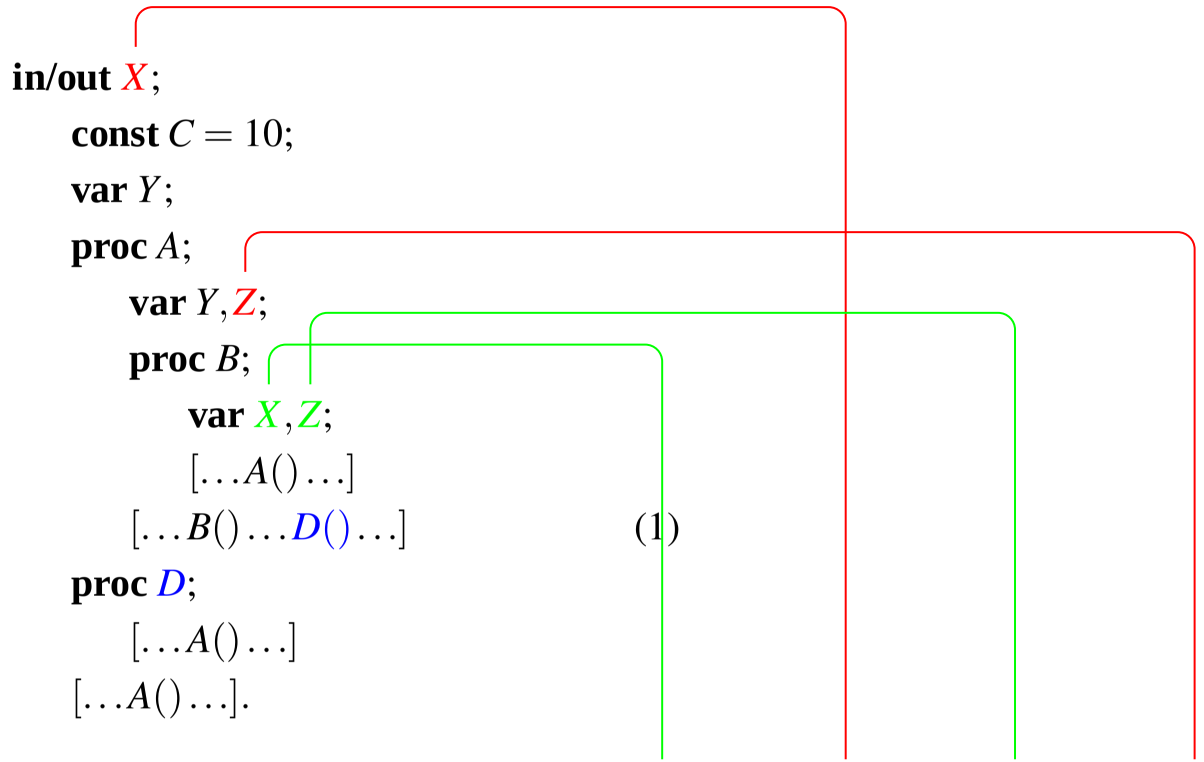
Syntax von BSP

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

CB 4-6

Abbildung 4.6: Syntax von BSP

Beispiel zum “static scope”–Prinzip



(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“

Semantische Bereiche

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

CB 4-8

Abbildung 4.8: Semantische Bereiche

Ausdruckssemantik:

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

$$\mathcal{E}_a[\![Z]\!]\rho\sigma := Z$$

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

$$\mathcal{E}_a[\![E_1 + E_2]\!]\rho\sigma := \mathcal{E}_a[\![E_1]\!]\rho\sigma + \mathcal{E}_a[\![E_2]\!]\rho\sigma$$

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

$$\mathcal{E}_b[\![E_1 < E_2]\!]\rho\sigma := \mathcal{E}_b[\![E_1]\!]\rho\sigma < \mathcal{E}_b[\![E_2]\!]\rho\sigma$$

$$\mathcal{E}_b[\![\mathbf{not}\ BE]\!]\rho\sigma := \neg\mathcal{E}_b[\![BE]\!]\rho\sigma$$

$$\mathcal{E}_b[\![BE_1 \overset{and}{or} BE_2]\!]\rho\sigma := \mathcal{E}_b[\![BE_1]\!]\rho\sigma \overset{\wedge}{\vee} \mathcal{E}_b[\![BE_2]\!]\rho\sigma$$

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

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

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

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

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

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

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

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

CB 4-10

Abbildung 4.10: Anweisungssemantik

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

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

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

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

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

mit $l =$ höchster Index eines belegten Speicherplatzes in σ

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

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

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

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

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

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

CB 4-11

Abbildung 4.11: Deklarationssemantik

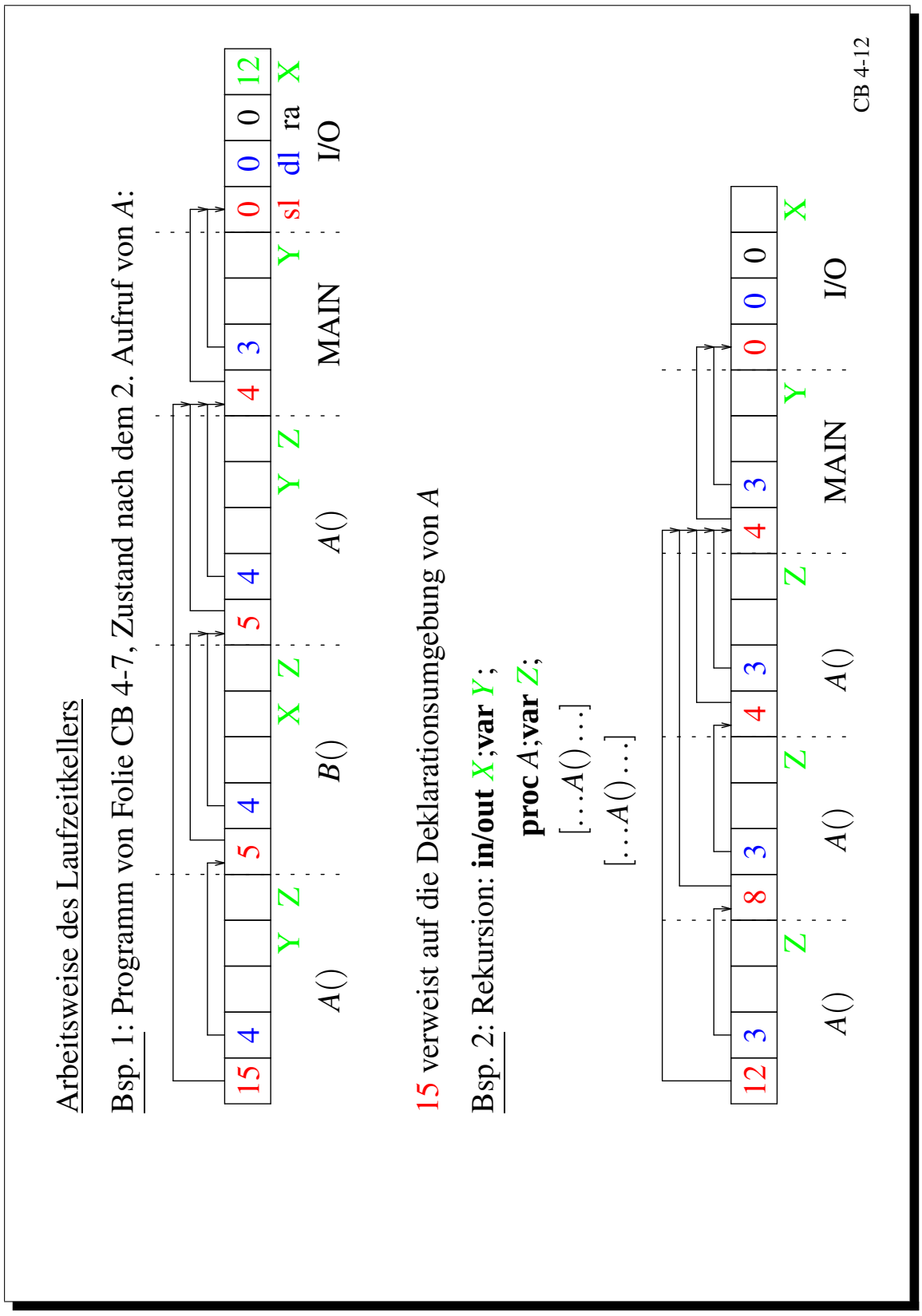


Abbildung 4.12: Arbeitsweise des Laufzeitkellers

$$
\begin{array}{rcl}
\mathbf{up}(\Delta_C\Delta_V\Delta_P, st, a, l) & := & \mathbf{up}(\Delta_P, \mathbf{up}(\Delta_V, \mathbf{up}(\Delta_C, st, a, l), a, l), a, l) \\
 & & \text{falls } \mathbf{diff\text{–}id}(\Delta_C\Delta_V\Delta_P) \\
\mathbf{up}(\varepsilon, st, a, l) & := & st \\
\mathbf{up}(\mathbf{const}\ I_1 = Z_1, \ldots, I_n = Z_n; , st, a, l) & := & st[I_1/(\mathbf{const}, Z_1), \ldots, I_n/(\mathbf{const}, Z_n)] \\
\mathbf{up}(\mathbf{var}\ I_1, \ldots, I_n; , st, a, l) & := & st[I_1/(\mathbf{var}, l, 1), \ldots, I_n/(\mathbf{var}, l, n)] \\
\mathbf{up}(\mathbf{proc}\ I_1; B_1; \ldots; \mathbf{proc}\ I_n; B_n; , st, a, l) & := & st[I_1/(\mathbf{proc}, a.1, l, \mathbf{size}(B_1)), \ldots, \\
 & & \quad I_n/(\mathbf{proc}, a.n, l, \mathbf{size}(B_n))] \\
 & & \\
\mathbf{dt}(\Delta_c, \Delta_v, \Delta_p, st, a, l) & := & \mathbf{dt}(\Delta_p, st, a, l) \\
\mathbf{dt}(\varepsilon, st, a, l) & := & \varepsilon \\
\mathbf{dt}(\mathbf{proc}\ I_1; B_1; \ldots; \mathbf{proc}\ I_n; B_n;\ st, a, l) & := & \mathbf{bt}(B_1, st, a.1, l+1)\ \ldots \\
 & & \mathbf{bt}(B_n, st, a.n, l+1) \\
 & & \\
\mathbf{bt}(\Delta\Gamma, st, a, l) & := & \mathbf{dt}(\Delta, \mathbf{up}(\Delta, st, a.1, l), a.1, l) \\
 & & \mathbf{ct}(\Gamma, \mathbf{up}(\Delta, st, a.1, l), a.1, l)
\end{array}
$$

CB 4-13

Abbildung 4.13: Semantik

Beispiel zur Übersetzung von BSP in ZP–Code

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

B_F / Γ_F : $B_F \parallel \Gamma_F$

$\tilde{\Delta}$: $B_F \parallel \Gamma_F$

Δ : $\tilde{\Delta}$

Γ : begin … end.

$$\textbf{trans}(\textbf{in/out } X;\Delta\Gamma) = 1 : \mathrm{CALL}(a_\Gamma, 0, 1);\ 2 : \mathrm{JMP}(0);\ \textbf{bt}(\Delta\Gamma, st_{I/O}, a_\Gamma, 1)$$

mit $st_{I/O}(X) = (\textbf{var}, 0, 1)$

CB 4-14

Abbildung 4.14: Beispiel: Übersetzung BSP in ZP

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

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

$$
\begin{array}{lcl}
\mathbf{dt}(\Delta, \overline{st}, a_1, 1) & = & \mathbf{bt}(B_F, \overline{st}, a_{11}, 2) \\
 & = & \mathbf{ct}(\Gamma_F, \overline{st}, a_{11}2, 2) \\
 & & a_3 : \mathrm{RET};
\end{array}
$$

$$
\begin{array}{lcl}
\mathbf{ct}(\Gamma, \overline{st}, a_\Gamma, 1) & = & a_\Gamma : \mathrm{LIT}(1); \\
 & & \mathrm{STO}(0,1); \\
 & & \mathrm{CALL}(a_{11}, 0, 0); \\
 & & \mathrm{LOAD}(0,1); \\
 & & \mathrm{STO}(1,1);
\end{array}
$$

$$
\begin{array}{lcl}
\mathbf{ct}(\Gamma_F, \overline{st}, a_{11}, 2) & = & \mathbf{sbt}(1 < x, \overline{st}, a_{11}, 2) \\
 & & a_4 : \mathrm{JFALSE}(a_5); \\
 & & \mathbf{ct}(\mathbf{begin} \ldots \mathbf{end}, \overline{st}, a_4 + 1, 2) \\
 & & a_5 :
\end{array}
$$

$$
\begin{array}{lcl}
\mathbf{sbt}(1 < x, \overline{st}, a_{11}, 2) & = & a_{11} : \mathrm{LIT}(1); \\
 & & \mathrm{LOAD}(2,1); \\
 & & \mathrm{LT};
\end{array}
$$

$$
\begin{array}{lcl}
\mathbf{ct}(\mathbf{begin} \ldots \mathbf{end}, \overline{st}, a_4 + 1, 2) & = & \mathbf{ct}(E := E * X, \overline{st}, a_4 + 1, 2) \\
 & & \mathbf{ct}(X := X - 1, \overline{st}, a_6, 2) \\
 & & \mathbf{ct}(F(), \overline{st}, a_7, 2) \\
 & = & a_4 + 1 : \mathrm{LOAD}(1,1); \\
 & & \mathrm{LOAD}(2,1); \\
 & & \mathrm{MULT}; \\
 & & \mathrm{STO}(1,1); \\
 & & \mathrm{LOAD}(2,1); \\
 & & \mathrm{LIT}(1); \\
 & & \mathrm{SUB}; \\
 & & \mathrm{STO}(2,1); \\
 & & \mathrm{CALL}(a_{11}, 1, 0)
\end{array}
$$

CB 4-15

Abbildung 4.15: Beispiel Fortsetzung

Ergebnis der Übersetzung:

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

1 :	$\text{CALL}(a_\Gamma, 0, 1);$	Also:
2 :	$\text{JMP}(0);$	
a_{11} :	$\text{LIT}(1);$	$a_{11} = 3$
	$\text{LOAD}(2,1);$	
	$\text{LT};$	
a_4 :	$\text{JFALSE}(a_5);$	$a_4 = 6$
	$\text{LOAD}(1,1);$	
	$\text{LOAD}(2,1);$	
	$\text{MULT};$	
	$\text{STO}(1,1);$	
	$\text{LOAD}(2,1);$	
	$\text{LIT}(1);$	
	$\text{SUB};$	
	$\text{STO}(2,1);$	
	$\text{CALL}(a_{11}, 1, 0);$	
$a_3 : a_5$:	$\text{RET};$	$a_3 = 16 = a_5$
a_Γ :	$\text{LIT}(1);$	$a_\Gamma = 17$
	$\text{STO}(0,1);$	
	$\text{CALL}(a_{11}, 0, 0);$	
	$\text{LOAD}(0,1);$	
	$\text{STO}(1,1);$	
a_2 :	$\text{RET};$	$a_2 = 22$

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 :
$\Delta ::= \Delta_C \Delta_V \Delta_P$
$\Delta_C ::= \varepsilon \mid \textbf{const}\ I_1 = Z_1, \ldots, I_n = Z_n;$
$\Delta_V ::= \varepsilon \mid \textbf{var}\ I_1, \ldots, I_n;$
$\Delta_P ::= \varepsilon \mid \textbf{proc}\ I(\underbrace{I_1, \ldots, I_p;}\textbf{var}\ \underbrace{J_1, \ldots, J_q;}) B; \ldots$
(* formale Wert- und Variablenparameter *)

AExp : $E ::= Z \mid I \mid (E_1 + E_2) \mid \ldots$

BExp : $BE ::= (E_1 < E_2) \mid \textbf{not}\ BE \mid (BE_1\ \textbf{and}\ BE_2) \mid \ldots$

V : $V ::= I$ (* Variablenparameter *)

Cmd : $\Gamma ::= I := E \mid I(\underbrace{E_1, \ldots, E_p; V_1, \ldots, V_q}) \mid$
(* aktuelle Parameter(ausdrücke) *)
$\textbf{begin}\ \Gamma_1; \ldots; \Gamma_n\ \textbf{end} \mid$
$\textbf{if}\ E\ \textbf{then}\ \Gamma \mid \textbf{while}\ E\ \textbf{do}\ \Gamma$

Block : $B ::= \Delta\Gamma$

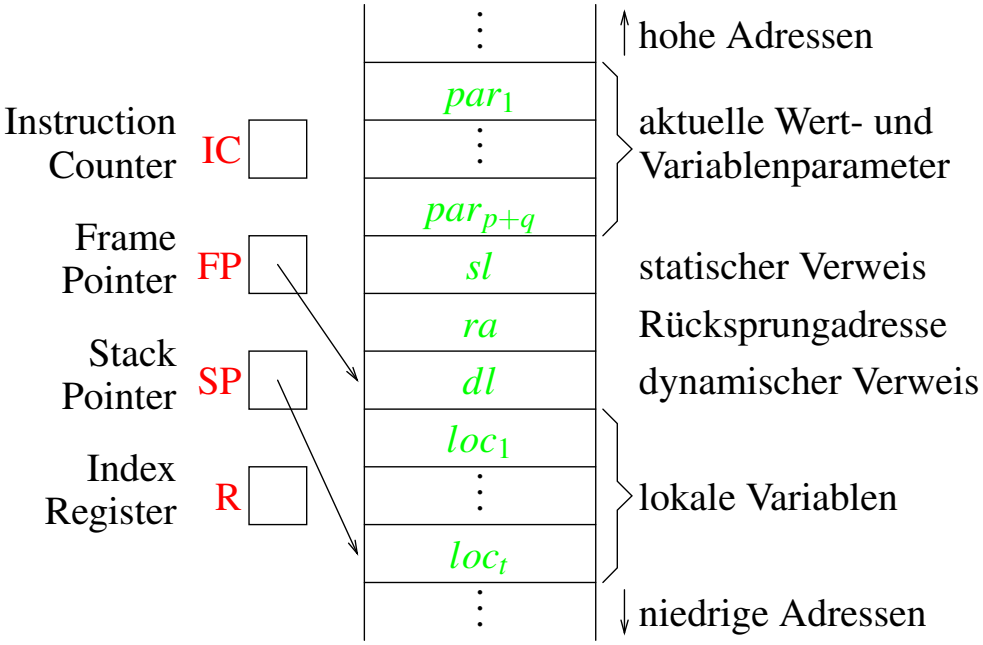
Prog : $P ::= \textbf{in/out}\ I_1, \ldots, I_n; B.$ (* $n \geq 1$ *)

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

$$\textbf{trans}(\textbf{in/out}\ I_1,\ldots,I_n;B.) :=$$
$$\left.\begin{array}{l} 1: PUSH\ FP; \\ 2: CALL\ a_\Gamma; \end{array}\right\} \text{Anlegen von } sl \text{ und } ra$$
$$3: JMP\ 0;$$
$$\textbf{bt}(B, st_{I/O}, a_\Gamma, 1, 0)$$

$$\textbf{bt}(\Delta\Gamma, st, a, l, r) :=$$
$$\textbf{dt}(\Delta, \textbf{up}(\Delta, st, a.1, l), a.1, l)$$
$$\left.\begin{array}{l} a: PUSH\ FP; \\ -: LOAD\ FP, SP; \\ -: SUB\ SP, \textbf{size}(\Delta\Gamma); \end{array}\right\} \text{Entry–Code}$$
$$\textbf{ct}(\Gamma, \textbf{up}(\Delta, st, a.1, l), a+3, l)$$
$$\left.\begin{array}{l} -: LOAD\ SP, FP; \\ -: POP\ FP; \\ -: RET\ r+1; \end{array}\right\} \text{Exit–Code}$$

$$\textbf{dt}(\textbf{proc}\ I(I_1,\ldots,I_p;\textbf{var}\ J_1,\ldots,J_q);B;\ldots, st, a, l) :=$$
$$\textbf{bt}(B, st', a.1, l+1, p+q)\ \textbf{bt}(\ldots)$$

falls **diff–id**$(I_1,\ldots,I_p,J_1,\ldots,J_q)$ **and** **diff–id**$(\ldots)$

wobei $st' := st[I_1/(\textbf{var}, l+1, p+q+2), \ldots,$
$I_p/(\textbf{var}, l+1, q+3),$
$J_1/(\textbf{vpar}, l+1, q+2), \ldots,$
$J_q/(\textbf{vpar}, l+1, 3)]$

(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.)

$$
\begin{array}{l}
\textbf{ct}(I := E, st, a, l) := \\
\quad \textbf{et}(E, st, a, l) \\
\quad \textbf{if } st(I) = (\textbf{var}, dl, o) \textbf{ then} \\
\qquad \textbf{if } l - dl = 0 \textbf{ then } - : POP\ \langle FP{+}o \rangle; \\
\qquad \textbf{if } l - dl = k + 1 \textbf{ then } - : LOAD\ R, \langle FP + 2 \rangle; \\
\qquad\qquad\qquad\qquad \left.\begin{array}{l} - : LOAD\ R, \langle R + 2 \rangle; \\ \vdots \\ - : LOAD\ R, \langle R + 2 \rangle; \end{array}\right\} k\text{–mal} \\
\qquad\qquad\qquad\qquad - : POP\ \langle R{+}o \rangle; \\
\quad \textbf{if } st(I) = (\textbf{vpar}, dl, o) \textbf{ then} \\
\qquad \textbf{if } l - dl = 0 \textbf{ then } - : LOAD\ R, \langle FP{+}o \rangle; \\
\qquad\qquad\qquad\qquad - : POP\ \langle R \rangle; \\
\qquad \textbf{if } l - dl = k + 1 \textbf{ then } - : LOAD\ R, \langle FP + 2 \rangle; \\
\qquad\qquad\qquad\qquad \left.\begin{array}{l} - : LOAD\ R, \langle R + 2 \rangle; \\ \vdots \\ - : LOAD\ R, \langle R + 2 \rangle; \end{array}\right\} k\text{–mal} \\
\qquad\qquad\qquad\qquad - : LOAD\ R, \langle R{+}o \rangle; \\
\qquad\qquad\qquad\qquad - : POP\ \langle R \rangle;
\end{array}
$$

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**
    $\left.\begin{array}{l}\mathbf{et}(E_1,st,a,l)\\ \vdots\\ \mathbf{et}(E_p,st,a,l)\end{array}\right\}$ 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–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–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.)

$\mathbf{et}(I, st, a, l) :=$
  if $st(I) = (\mathbf{const}, Z)$ **then** a: $PUSH\ Z$;
  if $st(I) = (\mathbf{var}, dl, o)$ **then**
    if $l - dl = 0$ **then** a: $PUSH\ \langle FP+o \rangle$;
    if $l - dl = k+1$ **then** a: $LOAD\ R, \langle FP+2 \rangle$;
$$\left.\begin{array}{l} -: LOAD\ R, \langle R+2 \rangle; \\ \vdots \\ -: LOAD\ R, \langle R+2 \rangle; \end{array}\right\} k\text{–mal}$$
$-: PUSH\ \langle R+o \rangle$;
  if $st(I) = (\mathbf{vpar}, dl, o)$ **then**
    if $l - dl = 0$ **then** a: $LOAD\ R, \langle FP+o \rangle$;
$-: PUSH\ \langle R \rangle$;
    if $l - dl = k+1$ **then** a: $LOAD\ R, \langle FP+2 \rangle$;
$$\left.\begin{array}{l} -: LOAD\ R, \langle R+2 \rangle; \\ \vdots \\ -: LOAD\ R, \langle R+2 \rangle; \end{array}\right\} k\text{–mal}$$
$LOAD\ R, \langle R+o \rangle$;
$PUSH\ \langle R \rangle$;

(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	::=	$Z \mid B \mid R$
Ide :	I		
Typ :	T	::=	$I \mid$ **Bool** | **Int** | **Real** |
			array $[Z_1..Z_2]$ **of** $T \mid$
			record $I_1 : T_1; \ldots; I_n : T_n$ **end**
Var :	V	::=	$I \mid V[E] \mid V.I$
Exp :	E	::=	$C \mid V \mid (E_1 + E_2) \mid \ldots$
Cmd :	Γ	::=	$V := E \mid$ **begin** ... **end** | **if** ... |
			while ...
Decl :	Δ	::=	$\Delta_C \Delta_T \Delta_V$
	Δ_C	::=	$\varepsilon \mid$ **const** $I_1 = C_1, \ldots, I_n = C_n;$
	Δ_T	::=	$\varepsilon \mid$ **type** $I_1 = T_1, \ldots, I_n = T_n;$
	Δ_V	::=	$\varepsilon \mid$ **var** $I_1 : T_1, \ldots, I_n : T_1;$
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** $T \quad \rightsquigarrow \quad Z_1 \leq Z_2$
- 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
  $\rightsquigarrow I \in \{I_1, \ldots, I_{j-1}\}$
- 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:

$$\mathcal{C}[\![\text{JAC}(Z_1,Z_2)]\!](m,d:z,h) := \textbf{if } Z_1 \leq z \leq Z_2 \textbf{ then } (m+1,d:z,h)$$
$$\textbf{else } (\text{STOP},d:\underbrace{\text{RTE}}_{\text{„run–time error“}},h)$$

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 $\Delta =$ **type** Bool = **bool**; Int = **int**;
Array = **array**[1..20]**of** Bool;
Record = **record** S : Array; T : Int **end**;
var X : Int; Y : Array; Z : Record;

gilt:

$$\begin{aligned}&\mathbf{up}(\Delta, st) = \\ &\quad st[\mathrm{Bool}/(\mathbf{type}, \mathbf{bool}, 1), \\ &\quad\ \mathrm{Int}/(\mathbf{type}, \mathbf{int}, 1), \\ &\quad\ \mathrm{Array}/(\mathbf{type}, \mathbf{array}, 1, 20, \mathrm{Bool}, 20), \\ &\quad\ \mathrm{Record}/(\mathbf{type}, \mathbf{record}, \mathrm{S}, \mathrm{Array}, 0, \mathrm{T}, \mathrm{Int}, 20, 21), \\ &\quad\ \mathrm{X}/(\mathbf{var}, \mathrm{Int}, 1), \\ &\quad\ \mathrm{Y}/(\mathbf{var}, \mathrm{Array}, 2), \\ &\quad\ \mathrm{Z}/(\mathbf{var}, \mathrm{Record}, 22)]\end{aligned}$$

CB 4-27

Abbildung 4.27: Symboltabelle

Übersetzung von PSSD–Programmen in Maschinencode

Hilfsfunktion $\boxed{\textbf{vtyp} : \textbf{Var} \times \textbf{Tab} \dashrightarrow \textbf{Ide}}$ bestimmt zu einer Variablen bzgl. einer Symboltabelle den zugehörigen Typbezeichner:

$$
\begin{aligned}
\textbf{vtyp}(I, st) &:= \textbf{if } st(I) = (\textbf{var}, J, k) \textbf{ then } J \\
\textbf{vtyp}(V[E], st) &:= \textbf{if } \textbf{vtyp}(V, st) = I \textbf{ and} \\
&\quad st(I) = (\textbf{type}, \textbf{array}, Z_1, Z_2, J, n) \\
&\quad \textbf{then } J \\
\textbf{vtyp}(V.I, st) &:= \textbf{if } \textbf{vtyp}(V, st) = J \textbf{ and } st(J) = \\
&\quad (\textbf{type}, \textbf{record}, S_1, J_1, 0, \ldots, S_n, J_n, p, q) \\
&\quad \textbf{and } I = S_i \\
&\quad \textbf{then } J_i
\end{aligned}
$$

Übersetzungsfunktion $\boxed{\textbf{vt} : \textbf{Var} \times \textbf{Tab} \dashrightarrow \text{Code}}$ berechnet auf DK die Anfangsadresse einer Variablen im HS:

$$
\begin{aligned}
\textbf{vt}(I, st) &:= \textbf{if } st(I) = (\textbf{var}, J, k) \textbf{ then LIT } k; \\
\textbf{vt}(V[E], st) &:= \textbf{if } \textbf{vtyp}(V, st) = I \\
&\quad \textbf{and } st(I) = (\textbf{type}, \textbf{array}, Z_1, Z_2, J, n) \\
&\quad \textbf{and } st(J) = (\textbf{type}, \ldots, k) \textbf{ then} \\
&\quad \textbf{vt}(V, st) \\
&\quad \textbf{et}(E, st) \\
&\quad \text{JAC}(Z_1, Z_2); \\
&\quad \text{LIT } Z_1; \\
&\quad \text{SUB}; \\
&\quad \text{LIT } k; \\
&\quad \text{MULT}; \\
&\quad \text{ADD}; \\
\textbf{vt}(V.I, st) &:= \textbf{if } \textbf{vtyp}(V, st) = J \textbf{ and } st(J) = \\
&\quad (\textbf{type}, \textbf{record}, S_1, J_1, p_1, \ldots, S_n, J_n, p_n, q) \\
&\quad \textbf{and } I = S_i \textbf{ then} \\
&\quad \textbf{vt}(V, st) \\
&\quad \text{LIT } p_i; \\
&\quad \text{ADD};
\end{aligned}
$$

CB 4-28

Abbildung 4.28: Übersetzung von PSSD I

Übersetzung PSSD (Forts.)

Übersetzungsfunktion $\boxed{\textbf{et} : \textbf{Exp} \times \textbf{Tab} \dashrightarrow \text{Code}}$ erzeugt den üblichen DK–Berechnungscode, wobei jetzt Variablenwerte mit LODI geladen werden:

$$
\begin{aligned}
\textbf{et}(I, st) := \; & \textbf{if } st(I) = (\textbf{const}, C) \\
& \textbf{then } \text{LIT } C; \\
& \textbf{else if } st(I) = (\textbf{var}, J, k) \\
& \textbf{and } st(J) = (\textbf{type}, bas, 1) \textbf{ then} \\
& \text{LIT } k; \\
& \text{LODI;} \\
\textbf{et}(C, st) := \; & \text{LIT } C; \\
\textbf{et}(V, st) := \; & \textbf{if vtyp}(V, st) = J \\
& \textbf{and } st(J) = (\textbf{type}, bas, 1) \textbf{ then} \\
& \textbf{vt}(V, st) \\
& \text{LODI;} \\
\textbf{et}(E_1 + E_2, st) := \; & \ldots \text{ (wie bei PSA)}
\end{aligned}
$$

Übersetzungsfunktion $\boxed{\textbf{ct} : \textbf{Cmd} \times \textbf{Tab} \dashrightarrow \text{Code}}$:

$$
\begin{aligned}
\textbf{ct}(V := E, st) := \; & \textbf{vt}(V, st) \\
& \textbf{et}(E, st) \\
& \text{STOI;} \\
& \vdots \quad \text{(sonst wie PSA)}
\end{aligned}
$$

CB 4-29

Abbildung 4.29: Übersetzung von PSSD II

Kapitel 5

Exkurs: Yacc

5.1 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.

5.1.1 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

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

5.2 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.

5.3 Präzedenz von Operatoren in Yacc

Assoziativität und Präzedenz werden gleichzeitig festgelegt. Syntax:
`%[left|right]` $op_{1,1} \ldots op_{1,n}$
...
`%[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.

5.3.1 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