Compilerbau

Vorlesungsmitschrift
von Achim Lücking

zur Vorlesung von
Prof. Dr. Klaus Indermark

Wintersemster 2000/2001
(Neuüberarbeitung 2002, korrigierte Version 2003)

Vorwort

Endlich ist das Skript zur Vorlesung *Compilerbau* fertig. Das Skript basiert auf der Vorlesung von Prof. Dr. Indermark aus dem Wintersemester 2000/2001 an der RWTH Aachen. Dabei weise ich ausdrücklich daraufhin, daß es sich hierbei um **kein offizielles Skript** handelt und **kein Anspruch auf Vollständigkeit und Richtigkeit** erhoben wird. Dieses Skript stützt sich ausschließlich auf meine Vorlesungsaufzeichnungen und -mitschriften. Ferner liegen die Rechte der Vorlesung immer noch bei Prof. Indermark.

Sollten sich noch Fehler eingeschlichen haben, sei es inhaltlich oder einfach nur satztechnisch, so möchte ich Euch bitten, mir dieses per e-Mail zu melden, damit ich das Skript entsprechend verbessern und optimieren kann. e-Mails bitte an folgende Adresse: `cb-skript@achim-luecking.de`. Die jeweils neueste Version des Skriptes gibt es im Postscriptformat auf meiner Homepage unter `http://www.achim-luecking.de` zum Download. Da das Skript aufgrund von Fehlermeldungen immer mal wieder aktualisiert wird, empfiehlt es sich, ab und an mal vorbeischauen.

Hinweisen möchte ich ferner auf die Notation im Skript: Kästchen der Form $\boxed{\mathbf{1.2}}$ bezeichnen einen Verweis auf eine in der Vorlesung an dieser Stelle gezeigte Folie mit der angegebenen Nummer. Die Folien beinhalten noch zusätzliche Abbildungen, Grafiken und Übersichten, die im Skript so nicht berücksichtigt sind. Die **Folien sind nicht Bestandteil des Skriptes** und müssen, sofern noch vorhanden, getrennt vom Server des Lehrstuhls heruntergeladen werden oder von jemandem, der sie noch hat, kopiert werden.

Für das Korrekturlesen und gemeinsame Durcharbeiten danke ich Arndt Baars.

Für die vielen, ntzlichen Hinweise und Korrekturen danke ich ferner Ernest Hammerschmidt, Bastian Braun und Diego Biurrun.

Viel Spaß mit dem Skript !

Aachen, den 4. Mai 2003

Achim Lücking

Inhaltsverzeichnis

Kapitel 0

Einleitung

0-1

0.1 Einführung

Ein **Compiler** ist ein Programm zur Übersetzung von PS-Programmen (Quellprogrammen) in äquivalente MS-Programme (Zielprogramme). Dabei bezeichnet **PS** eine höhere Programmiersprache. Diese kann man in 4 wesentliche Bereiche unterteilen:

- **imperative Programmiersprachen:** Wertzuweisungen, Kontrollstrukturen, Datenstrukturen, Programmstrukturen (Module, Klassen)
- **deklarative Programmiersprachen:** funktional, logisch → andere Vorlesung
- **nebenläufige Programmiersprachen:** kommunizierende Prozeße, verteilte Systeme
- **objektorientierte Programmiersprachen:** Klassen, Vererbung

MS bezeichnet die Maschinensprache.

Die Grundkonzepte des von-Neumann-Rechners mit elementaren Maschinenbefehlen[1] werden in folgende beiden Kategorien eingeteilt:

- **RISC** = reduced instruction set computer
- **CISC** = complex instruction set computer

Ferner kann man noch Grundkonzepte Parallelrechner, verteilte Systeme und Rechnernetze annehmen.

[1] vgl. Vorlesung "Rechnerstrukturen" aus dem Grundstudium

0.2 Aspekte einer Programmiersprache

Bei den Aspekten einer Programmiersprache unterscheiden wir drei Punkte:

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

Man kann sagen, daß aus der Äquivalenz von Programmen quasi die semantische Gleichheit der Programme folgt.

0-2

0.3 Struktur eines Compilers

Bei der Struktur des Compilers werden zwei logisch unabhängige **Phasen** unterschieden, die aber verzahnt als **Passes** (Läufe) ablaufen: die Analysephase und die Synthesephase

Die **Analyse** dient der Bestimmung der syntaktischen Struktur und der Fehlererkennung. Folgende verschiedenen Analysen werden durchgeführt:

- lexikalische Analyse: Erkennung von Symbolen, Trennzeichen, Kommentaren (Hilfsmittel: endliche Automaten)
- syntaktische Analyse: Erkennung des hierachischen Programmaufbaus, Ableitungsbaum (Hilfsmittel: Kellerautomaten)
- semantische Analyse: Kontextabhängigkeiten, statische Semantik, Typinformationen, Attributierung des Ableitungsbaums mit semantisch relevanten Informationen (Hilfsmittel: attributierte Grammatiken)

Die **Synthese** hingegen dient der Erzeugung von MS-Code aus einem attributiertem Ableitungsbaum.

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

0-3

Bei der Synthese wird ferner in Frontend und Backend unterteilt.

- **Frontend** (MS-unabhängig): Analyse + Zwischencode + ggf. MS-unabhängige Optimierung
- **Backend** (MS-abhängig): MS-Code-Generierung + Optimierung

Insgesamt bezeichnet man Compiler als One-Pass-Compiler bzw. n-Pass-Compiler (n ist die Zahl der Durchläufe des Quellprogramms).

Die **Ziele** der Vorlesung sind:

- Wirkungsweise und Konstruktion von Compilern
- vertieftes Verständnis von Programmiersprachen und ihrem Rechnerbezug
- Compiler als gut verstandenes Beispiel eines Softwaresystems mit Interaktion von Softwarebausteinen[2]
- Erkennung und Transformation strukturierter Objekte

[2] vgl. Vorlesung "Softwaretechnik" von Prof. Nagl im Hauptstudium

Kapitel 1

Lexikalische Analyse

Ausgangspunkt: Quellprogramm P als Zeichenfolge
Es ist Σ_0 ein Zeichensatz (z.B. ASCII) und $a \in \Sigma_0$ ein Zeichen (oder **lexikalisches Atom**)[1]. Insgesamt kann man sagen:

$$P \in \Sigma_0^*$$

P besitzt aufgrund der Pragmatik von PS eine **lexikalische Struktur**. Der pragmatische Aspekt dabei ist die Benutzerfreundlichkeit von PS.

- natürliche Sprachen für Bezeichner, Schlüsselwörter, usw.
- mathematische Formelsprache für Zahlen, Formalen (z.B. Indizes linearisieren: $y^2 \rightsquigarrow$ `y**2`)
- Leerzeichen, Zeilenwechsel, Einrücken
- Kommentare

Sinn der Sache ist die gute Lesbarkeit und Wartbarkeit.
Die Semantik von P und die Übersetzung von P ist syntaxorientiert: sie folgt dem hierachischen Programmaufbau; **der pragmatische Aspekt dabei ist irrelevant!**

1. Beobachtung
syntaktische Atome (**Symbole**) werden dargestellt als Folgen von lexikalischen Atomen, sogen. **Lexemen**.

1. Aufgabe der Analyse
Zerlegung des Quellprogramms P in eine Folge von **Lexemen**.

2. Beobachtung
Für syntaktische Analyse ist der Unterschied von Lexemen oft irrelevant, z.B. müssen Bezeichner nicht unterschieden werden. Lexeme werden zu **Symbolklassen** zusammengefaßt. Die Darstellung einer Symbolklasse ist ein **Token**.

[1] Als Erläuterung: es existiert ein Unterschied zwischen Symbol und Zeichen; ein Symbol besteht aus Zeichen.

Die syntaktische Analyse bearbeitet eine Tokenfolge. Die Identifiziuerung eines Symbols erfolgt durch ein zusätzliches **Attribut** für Semantische Analyse und Codegenerierung.

$$\text{Symbol} = (\text{Token}\ ,\ \text{Attribut})$$

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

Lexikalische Analyse:
Zerlegung eines Quellprogramms in eine Folge von Lexema und deren Transformation in eine Folge von Symbolen
$\boxed{\textbf{1-1}}$

Ein **Scanner** ist ein Programm zur lexikalischen Analyse. Beachte dabei, daß Leerzeichen und Kommentare gelöscht werden, da sie keine Rolle bei der Lexem-Gliederung spielen.

Die wichtigsten **Symbolklassen** (eigentlich Lexemklassen):

- **Bezeicher**
- **Zahlwörter**
- **Schlüsselwörter**: Bezeichner mit vorgegebener Bedeutung.
- **einfache Symbole**: ein Sonderzeichen, z.B. $+, -, \cdot, (, \ldots$, bilden **eine** Symbolklasse.
- **zusammengesetzte Symbole**: Folgen von 2 oder mehr Sonderzeichen, z.B. $:=$ $, **, <=, \ldots$.
- **Leerzeichen**: $\sqcup, \sqcup \ldots \sqcup$
- **spezielle Symbole**: Kommentare, Pragmas (wie z.B. Compiler Options)

Token: id, const, divsym, getsym, ...

Attribute: Zeiger in die Symboltabelle, Binärdarstellung einer Zahl, kann aber auch bei Symbolklassen mit einem Symbol leer sein.

Feststellung: Symbolklassen sind reguläre Mengen $\rightsquigarrow$ Beschreibung durch reguläre Ausdrücke und damit Erkennung durch endliche Automaten ist durchführbar; automatische Scanner-Generierung, z.B. mit `Lex`, ist möglich.

1.1 Scannerkonstruktion

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

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

Die Semantik $[\![.]\!] : RA(\Sigma) \longrightarrow \wp(\Sigma^*)$ der regulären Ausdrücke war wie folgt definiert:

- $[\![\Lambda]\!] := \emptyset$
- $[\![a]\!] := \{a\}$
- $[\![(\alpha \vee \beta)]\!] := [\![\alpha]\!] \cup [\![\beta]\!]$
- $[\![(\alpha \cdot \beta)]\!] := [\![\alpha]\!] \cdot [\![\beta]\!]$
- $[\![(\alpha^*)]\!] := [\![\alpha]\!]^*$

1.1.1 Spracherweiterung von $RA(\Sigma)$

Zur einfacheren Beschreibung von Symbolklassen (regulärer Sprachen) werden folgende Erweiterungen vereinbart:

1. Vereinfachende Bezeichnungen
   Präzedenzregel, um Klammern zu sparen:

   - $*$ bindet stärker als $\cdot$, $\cdot$ bindet stärker als $\vee$
   - $\cdot$ und $\vee$ sind linksassoziativ
   - $\cdot$ wird unterdrückt, $\vee$ wird auch $|$ geschrieben

   Beispiel: $a \mid b^*c$ statt $(a \vee ((b^*) \cdot c))$

2. Reguläre Definitionen
   Schrittweise Beschreibung von Symbolklassen durch zusätzliche, frei gewählte Bezeichner (Meta-Bezeichner)

$$\begin{array}{rcl} \underline{id}_1 & = & \alpha_1 \\ & \vdots & \\ \underline{id}_n & = & \alpha_n \end{array}$$

   mit $\underline{id}_1, \ldots, \underline{id}_n \notin \Sigma$ und $\alpha_i \in RA(\Sigma \,\dot{\cup}\, \{\underline{id}_1, \ldots, \underline{id}_{(i-1)}\})$.

   Beachte: Keine Rekursion ! Eine Entschachtelung ist möglich, andernfalls liegt eine EBNF vor.

$\boxed{\textbf{1-2}}$

1.1.2 Das einfache Matching-Problem

Entscheide für $\alpha \in RA(\Sigma)$ und $w \in \Sigma^*$, ob $w \in [\![\alpha]\!]$ oder $w \notin [\![\alpha]\!]$. Als Hilfsmittel dienen endliche Automaten $\mathfrak{A} = \langle Q, \Sigma, \delta, q_0, F \rangle \in NFA(\Sigma)$, wenn gilt:

$$\delta : Q \times (\Sigma \cup \{\varepsilon\}) \longrightarrow \wp(Q)$$

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

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

Die **erweiterte** Transitionsfunktion $\overline{\delta} : \wp(Q) \times \Sigma^* \longrightarrow \wp(Q)$ ist dann definiert durch

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

$\mathfrak{A}$ erkennt die Sprache $L(\mathfrak{A}) := \{w \in \Sigma^* \mid \overline{\delta}(\{q_0\}, w) \cap F \neq \emptyset\}$. Es gilt ferner: $\mathfrak{A} \in DFA(\Sigma)$, wenn gilt: $|\delta(q, a)| = 1 \quad \forall q \in Q, a \in \Sigma$ und außerdem $|\delta(q, \varepsilon)| = 0 \quad \forall q \in Q$. Also: $\delta : Q \times \Sigma \longrightarrow Q$.

1.1.3 DFA-Methode

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

- zu (1): **Methoden von Thomson (Beweis des Kleene-Satzes)**[2]
  Für die **Korrektheit** müssen folgende Eigenschaften erfüllt sein:
  - jeder Automat hat genau einen Anfangszustand q_0, genau einen Endzustand q_f mit $q_f \neq q_0$, q_0 ist Quelle und q_f ist Senke.
  - jeder Zustand hat höchstens zwei Folgezustände: entweder einen a-Nachfolger oder höchstens zwei ε-Nachfolger.

  Die **Komplexität** zeigt linearen Platz- und Zeitbedarf.
  - **Platz:** $\mathfrak{A}(\alpha)$ hat höchstens $2|\alpha|$ Zustände, wobei $|\alpha|$ die Anzahl der Grundzeichen von $\Sigma \cup \{\Lambda\}$ und der Operatoren $\vee, \cdot, *$ ist.
  - **Zeit:** Syntaxanalyse von α in $O(|\alpha|)$ durchzuführen, z.B. Transformation von α in Postfix-Notation (ohne Klammern), und schrittweise Aufbau von $\mathfrak{A}(\alpha)$ in $O(|\alpha|)$ Schritten.

[2]Nachzulesen in: Communications ACM, 1968

- zu (2): **Konstruktion der ε-Hülle**
  Sei $\mathfrak{A}(\alpha) \in NFA(\Sigma)$ mit Zustandsmenge Q und $T \subseteq Q$. Dann ist die Aufgabe, $\varepsilon(T) \subseteq Q$ zu berechnen. Die Idee ist, die Zustände, welche auf ε-Nachfolger zu testen sind, in einem Stack zu speichern.

  1-4

  Die **Komplexität** sieht man recht einfach bei folgender Darstellung von $T \subseteq Q$: $Q = \{1, \ldots, n\}$ und $\widetilde{T} : Q \longrightarrow \{0, 1\}$. Damit gilt: $q \in T \quad \curvearrowright \quad \widetilde{T}(q) = 1$ (in konstanter Zeit prüfbar).

  - **Platz:** $O(n)$, weil $l(Stack) \leq n$; denn ein Zustand q erscheint höchstens einmal auf dem Stack.
  - **Zeit:** $O(n)$, insbesondere, weil die for-Schleife in konstanter Zeit durchlaufen wird (q hat höchstens 2 ε-Nachfolger).

  Potenzmengenkonstruktion
  Sei $\mathfrak{A} \in NFA(\Sigma)$. Dann ist der Potenzmengenautomat $\mathfrak{A}_P = \left\langle \widehat{Q}, \Sigma, \widehat{\delta}, \widehat{q_0}, \widehat{F} \right\rangle$ definiert durch

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

  1-5

  Die Idee des Algorithmus ist, $\widehat{Q}$ mit $\varepsilon(\{q_0\})$ zu initialisieren und sukzessiv mit $\widehat{\delta}(T, a)$ zu erweitern. $T \in \widehat{Q}$ wird markiert, wenn seine Nachfolger in $\widehat{Q}$ sind.

  Die **Komplexität der Potenzmengenkonstruktion**:

  - **Platz:** $O(2^{|Q|})$, jedoch werden nur erreichbare Zustände berücksichtigt.
  - **Zeit:** $O(2^{|Q|})$, aber die Berechnung von $\delta(T, a)$ läuft in Linearzeit, weil dies für die ε-Hülle gilt und außerdem ein Zustand von $\mathfrak{A}(\alpha)$ höchstens einen a-Nachfolger hat.

1.1.4 NFA-Methode

Verbesserung des Platzbedarfs bei längerer Laufzeit durch Verzicht auf die volle Potenzmengenkonstruktion. Da die Eingabe $w \in \Sigma^*$ bekannt ist, erfolgt die Potenzmengenkonstruktion nur noch für den *"Lauf von w durch* $\mathfrak{A}(\alpha)$". Es erfolgt die direkte Berechnung von $\overline{\delta}(\{q_0\}, w)$.

Der **Algorithmus** arbeitet mit einer Eingabe $\mathfrak{A}(\alpha) \in NFA(\Sigma)$ und einem Wort $w \in \Sigma^*$.

$$w = a_1 \ldots a_n$$

	a_1	$\cdots$	a_n	eof

$\uparrow$

$\boxed{\textbf{1-5}}$

Die **Komplexität** kann wie folgt angegeben werden:

- **Platz:** $O(|\alpha| + |w|)$; der 1. Stack für T, der 2. Stack für $\bigcup_{q \in T} \delta(q, a)$ mit einer jeweiligen Stacklänge $\leq |Q|$. Die Berechnung der ε-Hülle erfolgt mit einem zusätzlichen Bitvektor der Länge $|Q|$.
- **Zeit:** $O(|\alpha| \cdot |w|)$

Kombination von NFA- und DFA-Methode

Zwischenergebnisse von bereits berechneten Transitionen werden in einem Cache-Speicher gespeichert.

1.1.5 Das erweiterte Matching-Problem

Definition 1.2 *Seien $\alpha_1, \ldots, \alpha_n \in RA(\Sigma)$ und $w \in \Sigma^*$. Sei ferner $\Delta := \{S_1, \ldots, S_n\}$ ein Alphabet von Symbolen. 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 $v = S_{i_1} \ldots S_{i_k}$ eine **Analyse von** w **bezüglich** $\alpha_1, \ldots, \alpha_n$. v repräsentiert die lexikalische Struktur von w bezüglich $\alpha_1, \ldots, \alpha_n$.*

Die **Aufgabe** besteht in der Bestimmung einer Analyse bzw. darin, Fehler zu melden. Weder die Zerlegung, noch die Analyse sind im allgemeinen eindeutig bestimmt.

Konventionen für Eindeutigkeit

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

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

   Daraus läßt sich folgern, daß es für $w, \alpha_1, \ldots, \alpha_n$ höchstens eine lm-Zerlegung gibt.

2. **Prinzip des ersten Matches**
   Das Prinzip dient der Eindeutigkeit der Analyse. Trotz der Eindeutigkeit der lm-Zerlegung sind im allgemeinen mehrere zugehörige Analysen möglich, weil $[\![\alpha_p]\!] \cap [\![\alpha_q]\!] \neq \emptyset$ für $p \neq q$ möglich ist.

Die **Konvention** ist nun, daß der erste Match $\alpha_1, \ldots, \alpha_n$ zählt. Sei nun $(w_1, \ldots, w_k)$ eine lm-Zerlegung und $v = S_{i_1} \ldots S_{i_k}$ eine zugehörige Analyse von w bezüglich $\alpha_1, \ldots, \alpha_n$. Dann heißt v eine **flm-Analyse** (first longest match), falls für alle $j = 1, \ldots, k$ und $i = 1, \ldots, n$ gilt:

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

Daraus läßt sich folgern, daß es für w und für $\alpha_1, \ldots, \alpha_n$ höchstens eine flm-Analyse gibt.

Berechnung der flm-Analyse

Als Vorausetzung gelte im folgenden: $\alpha_1, \ldots, \alpha_n \in RA(\Sigma)$, o.B.d.A $[\![\alpha_i]\!] \neq \emptyset$ und $\varepsilon \notin [\![\alpha_i]\!]$ für $i = 1, \ldots, n$. Sei ferner $w \in \Sigma^*$ und $\Delta = \{S_1, \ldots, S_n\}$.

1. Konstruiere für $i = 1, \ldots, n$ einen Automaten $\mathfrak{A}_i = \left\langle Q_i, \Sigma, \delta_i, q_0^{(i)}, F_i \right\rangle \in DFA(\Sigma)$, so daß $[\![\alpha_i]\!] = L(\mathfrak{A}_i)$.

2. Bilde aus diesem den Produktautomaten $\mathfrak{A} = \langle Q, \Sigma, \delta, q_0, F \rangle \in DFA(\Sigma)$ mit

   - $Q = Q_1 \times \ldots \times Q_n$
   - $q_0 := (q_0^{(1)}, \ldots, q_0^{(n)})$
   - $(q^{(1)}, \ldots, q^{(n)}) \in F \quad :\curvearrowright \quad \exists i \in \{1, \ldots, n\} : \quad q^{(i)} \in F_i$
   - $\delta\left((q^{(1)}, \ldots, q^{(n)}), a\right) := \left(\delta_1(q^{(1)}, a), \ldots, \delta_n(q^{(n)}, a)\right)$

   Dann gilt:

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

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

   $$(q^{(1)}, \ldots, q^{(n)}) \in F^{(i)} \quad :\curvearrowright \quad q^{(i)} \in F$$

   und

   $$q^{(j)} \notin F_j \text{ für alle } 1 \leq j < i$$

   Dann gilt:

   $$\overline{\delta}(q_0, w) \in F^{(i)} \quad \curvearrowright \quad w \in [\![\alpha_i]\!]$$

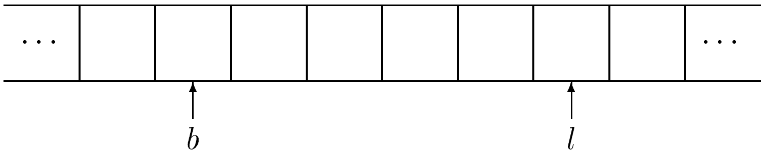
   und

   $$w \notin \bigcup_{j=1}^{i-1} [\![\alpha_j]\!]$$

   $q \in Q$ heißt **produktiv** $:\curvearrowright \quad \exists w \in \Sigma^* : \quad \overline{\delta}(q, w) \in F$. P ist die Menge der produktiven Zustände, also $F \subseteq P$. Die Aufgabe ist nun, P zu bestimmen.

3. Erweitere $\mathfrak{A}$ zu einem Backtrack-DFA $\mathfrak{B}$ mit Ausgabe
   Die Idee dabei ist, ein Einweg-Leseband mit 2 Köpfen zu verwenden:

- einen Backtrack-Kopf b zur Markierung eines Matches
- ein Look-Ahead-Kopf l zur Bestimmung des längsten Matches



Die Konfigurationsmenge von $\mathfrak{B}$ sei gegeben in folgender Form:

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

Dementsprechend ist eine Anfangskonfiguration für $w \in \Sigma^*$ wie folgt aus: $(N, q_0 w, \varepsilon)$.

Es ergeben sich folgende Transitionen (der Einfachheit halber gelte: $q' := \delta(q, a)$):

(a) **normal mode:** Match suchen

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

(b) **backtrack mode:** längsten Match suchen

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

(c) **Eingabeende:**

- $(N, q, W) \vdash$ Ausgabe: $W \cdot lexerr$ falls $q \in P \setminus F$
- $(S, q, W) \vdash$ Ausgabe: $W \cdot S$ falls $q \in F^{(i)}$
- $(S, vaq, W) \vdash (N, q_0 va, W)$ falls $q \in P \setminus F$

Dann gilt für $w \in \Sigma^*$:

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

Der **Zeitaufwand** liegt in der Größenordnung $O(|w|^2)$ für den worst-case.

Beispiel: Sei $\alpha_1 = abc$, $\alpha_2 = (abc)^* d$ und $\Sigma = \{a, b, c, d\}$. Dann erfordert $w = (abc)^m$ $O(m^2)$ Schritte.
Eine Verbesserung ist durch die Tabular-Methode[3] in Linearzeit.[4]

[3] vgl. KMP für String-Pattern-Matching

[4] Weiterführende Literatur: Th. Reps: "Maximal-Munch"-Tokenization in Linear Time, ACM-TOPLAS 20 (1998), S. 259-273

Beispiel für Pascal

Teil 1: $\boxed{\textbf{1-2}}$ (Symbolklassen)
Teil 2: Anweisungen

$$
\begin{array}{rcl}
\texttt{statement} & = & \underline{\texttt{if}}\ \texttt{expr}\ \underline{\texttt{then}}\ \texttt{statement}\ | \\
 & & \underline{\texttt{if}}\ \texttt{expr}\ \underline{\texttt{then}}\ \texttt{statement}\ \underline{\texttt{else}}\ \texttt{statement} \\
\texttt{expr} & = & \texttt{term}\ \underline{\texttt{relop}}\ \texttt{term}\ |\ \texttt{term} \\
\texttt{term} & = & \underline{\texttt{id}}\ |\ \underline{\texttt{num}}
\end{array}
$$

Annahme für die lexikalische Analyse:
Die Lexeme eines `statement` sind durch <u>`blanks`</u> getrennt.

$\boxed{\textbf{1-6}}$

Eingabe: $\underbrace{333}_{lm} E + B$

Besondere Rolle von <u>`blanks`</u> (White space):

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

Definition 1.3 *Ein **Sieber** ist ein Programm zur Beseitigung überflüssiger Lexeme wie Blanks, Kommentare, Pragmas, etc. .*

Attributberechnung

Symbolklassen mit mehreren Lexemen erfordern eine zusätzliche Attributberechnung.

- Dezimalzahl $\rightsquigarrow$ Binärzahl (`install_num`)
- Relationale Operatoren: `LT`,`GT`,`EQ`, ...
- Bezeichner: Prüfung auf Gleichheit von Bezeichnern! Verwendung einer Lexem-Tabelle (LexTab, `install_id`); Eintrag eines Bezeichners in LexTab, Zeiger als Attribut.

Alternative Behandlung von Schlüsselwörtern

Zunächst werden die Schlüsselwörter als Bezeichner behandelt und die LexTab mit eben diesen Schlüsselwörtern initialisiert. Die Attributberechnung (`install_id`) liefert statt $(\texttt{ID}, \bullet\!\!\rightarrow LexTab)$ einfach $(\texttt{IF},)$.

1.1.6 Automatische Scannergenerierung mit Lex

Lex ist ein Unix-Tool, welches nach folgendem Schema arbeitet:

$$\begin{array}{ccccc} \texttt{scan.l} & \overset{lex}{\rightsquigarrow} & \texttt{lex.yy.c} & \overset{C-Compiler}{\rightsquigarrow} & \texttt{a.out} \\ \text{(LexSpec)} & & \text{(C-Programm für Scanner)} & & \text{(ausführbarer Scanner)} \end{array}$$

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

Aufbau einer Lex-Spezifikation

Eine Lex-Spezifikation ist wie folgt aufgebaut:

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

Die letzten beiden Zeilen sind dabei optional.

Definitionen:

1. Direkter C-Code
2. Substitutionen (reguläre Definitionen) der Form

$$\begin{array}{cc} & \vdots \\ \texttt{Name} & \texttt{regexpr}. \\ & \vdots \end{array}$$

3. Startzustände (hier nicht vorgestellt)

Regeln:

$$\texttt{muster} \quad \{\texttt{action}\}$$

Dabei hat `muster` folgenden Aufbau:

$$\texttt{regexpr}_1 \quad [/\texttt{regexpr}_2]$$

Eine Zeichenkette paßt zu `muster`, wenn sie $\texttt{regexpr}_1$ matcht und gleichzeitig ein Lookahead $\texttt{regexpr}_2$ matcht. Der Ausdruck `action` ist C-Code zur Berechnung von Token und Attribut.

Finden der passenden Regel

- longest match
- first match
- keine passende Regel: Zeichen wird ausgegeben. Dies kann mit dem regulären Ausdruck ". | `\n`" abgefangen werden.

Token werden als Integer-Werte kodiert. **Attribute** über globale Variablen weitergeleitet; eine vordefinierte Variable dafür ist `yylval`.

Fileends: Aufruf einer benutzerdefinierten Prozedur

```
int    yywrap(void)
```

Die Rückgabe 1 hat dabei zur Folge, daß Lex eine 0 zurückliefert, was gleichbedeutend ist mit "kein Token / Ende der Tokensequenz".

$\boxed{\textbf{1-7}}$ $\boxed{\textbf{1-8}}$

Kapitel 2

Syntaktische Analyse

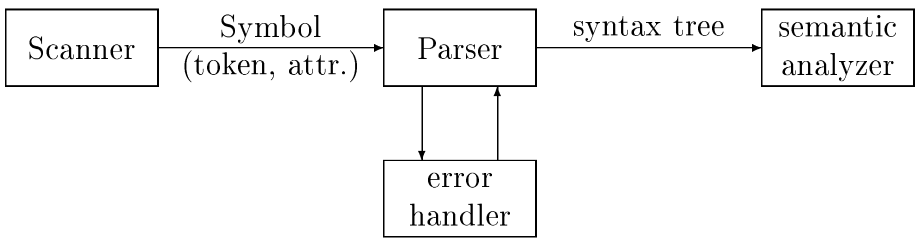
Die Aufgabe der syntaktischen Analyse ist die Zerlegung der Symbolfolgen, die der Scanner ausgibt, in sytaktische Einheiten bzw. Behandlung syntaktischer Fehler.

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

Beachte dabei: es erfolgt eine Schachtelung syntaktischer Einheiten in eine Baumstruktur, im Unterschied zur linearen Symbolfolge.

Definition 2.1 *Ein **Parser** ist ein Programm für die syntaktische Analyse.*

Schnittstellen:



Die **Beschreibung der syntaktischen Struktur** erfolgt durch kontextfreie Grammatiken, die **Erkennung und Analyse** wiederum durch Kellerautomaten mit Ausgabe.

Das Problem dabei ist die deterministische Simulation.

Im allgemeinen Fall: beliebige kontextfreie Grammatik (CFG) wird simuliert mit dem Tabularverfahren von Cocke, Younger und Kasami mit $O(n^3)$-Zeit und $O(n^2)$-Platz.

In Programmiersprachen: spezielle CFG, deren Analyse durch einen deterministischen Kellerautomaten mit "input-look-ahead" bei linearem Platz- und Zeitbedarf geschieht. Dabei gibt es 2 mögliche Techniken:

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

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

2.0.7 Kontextfreie Grammtiken

Für kontextfreie Grammatiken $\mathcal{G} = \langle N, \Sigma, P, S \rangle \in CFG(\Sigma)$gelten im weiteren Verlauf die folgenden Konventionen:

- $A, B, C, ... \in N$ Nichterminalsymbole
- $a, b, c, ... \in \Sigma$ Terminalsymbole
- $S \in N$ Startsymbol
- $u, v, w, ... \in \Sigma^*$ Terminalwörter
- $\alpha, \beta, \gamma, ... \in \mathcal{X}^*$ Satzformen ($\mathcal{X} = N \cup \Sigma$)
- $A \to \alpha := (A, \alpha) \in P$ Regeln, Produktion

Definition 2.2 *Die* ***Ableitungsrelation*** $\Rightarrow \subseteq (\mathcal{X}^*)^2$ *ist definiert durch:*

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

Gilt außerdem $\alpha_1 \in \Sigma^*$ *bzw.* $\alpha_2 \in \Sigma^*$, *so* $\alpha \Rightarrow_l \beta$ *bzw.* $\alpha \Rightarrow_r \beta$.

Man spricht von einem Ableitungsschritt bzw. Links- oder Rechtsableitungsschritt.

Die erzeugten Sprache ist definiert als

$$\begin{aligned} L(\mathcal{G}) & := \{w \in \Sigma^* \mid S \stackrel{*}{\Rightarrow} w\} \\ & = \{w \in \Sigma^* \mid S \stackrel{*}{\Rightarrow}_l w\} \\ & = \{w \in \Sigma^* \mid S \stackrel{*}{\Rightarrow}_r w\} \end{aligned}$$

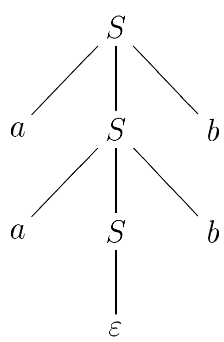
Beispiel:

$$\mathcal{G}: \qquad S \to aSb \mid \varepsilon$$

Das ergibt folgende Ableitung:

$$S \Rightarrow aSb \Rightarrow a^2Sb^2 \Rightarrow ...$$

Ableitungsbaum:



Der Ableitungsbaum repräsentiert die syntaktische Struktur des abgeleiteten Wortes.

Definition 2.3 *$\mathcal{G} \in CFG(\Sigma)$ heißt **eindeutig**, wenn es zu jedem ableitbaren Wort genau **einen** Ableitungsbaum gibt.*

Folgerung 2.4 *$\mathcal{G}$ ist eindeutig, genau dann wenn es für jedes $w \in L(\mathcal{G})$ genau eine Linksableitung (bzw. Rechtsableitung) gibt.*

Definition 2.5 *$\mathcal{G} \in CFG(\Sigma)$ heißt mehrdeutig, wenn $\mathcal{G}$ nicht eindeutig ist.*

$\boxed{\textbf{2-1}}$

2.0.8 l-Analyse, r-Analyse

Die Darstellung von l- bzw. r-Analysen geschieht durch Nummernfolgen. Es gelte: $|P| = p$, $[p] := \{1, \ldots, p\}$, $\pi[p] \to P$ ist bijektiv. Außerdem gelte die Vereinfachung $\pi_i := \pi(i)$.

$$wA\alpha \overset{i}{\Rightarrow}_l w\gamma\alpha \text{ bzw. } wA\alpha \overset{i}{\Rightarrow}_r w\gamma\alpha \text{ falls } \pi_i = A \to \gamma$$

Für $z = i_1, \ldots, i_n \in [p]^+$ gilt:

$$\alpha \overset{z}{\Rightarrow}_l \beta \quad :\curvearrowright \quad \alpha \overset{i_1}{\Rightarrow}_l \alpha_1 \overset{i_2}{\Rightarrow}_l \ldots \overset{i_n}{\Rightarrow}_l \alpha_n = \beta$$

für passende $\alpha_1, \ldots, \alpha_{n-1}$ und $\alpha \overset{\varepsilon}{\Rightarrow}_l \alpha$ (ein "leerer" Ableitungsschritt)

Definition 2.6 *z heißt **l-Analyse** von $\alpha :\curvearrowright S \overset{z}{\Rightarrow}_l \alpha$. z heißt **r-Analyse** von $\alpha :\curvearrowright S \overset{z}{\Rightarrow}_r \alpha$.*

$\boxed{\textbf{2-2}}$

Syntaxanalyse

Sei $\mathcal{G} \in CFG(\Sigma), w \in \Sigma^*$ gegeben. Es erfolgt die Berechnung einer l/r-Analyse, falls $w \in L(\mathcal{G})$, anderfalls die Bestimmung syntaktischer Fehler. Die Generalvoraussetzung dabei: $\mathcal{G} \in CFG$ ist reduziert, d.h. für jedes $A \in N$ gibt es $\alpha, \beta \in \mathcal{X}^*$ und $w \in \Sigma^*$, so daß $S \overset{*}{\Rightarrow} \alpha A \beta$ (A erreichbar) und $A \overset{*}{\Rightarrow} w$ (A produktiv).

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

Definition 2.7 *Der **Top-Down-Analyseautomat** von $\mathcal{G} \in CFG$ (kurz: $NTA(\mathcal{G})$) besteht aus:*

- *Eingabealphabet: Σ*
- *Kelleralphabet: $\mathcal{X} := N \cup \Sigma$*
- *Ausgabealphabet: $[p]$*
- *Konfigurationsmenge: $\Sigma^* \times \mathcal{X}^* \times [p]^*$ (Kellerspitze links !)*

- *Transitionen:*
  - *Ableitungsschritt:* $(w, A\alpha, z) \vdash (w, \beta\alpha, zi)$ *falls* $\pi_i : A \to \beta$
  - *Vergleichsschritt:* $(aw, a\alpha, z) \vdash (w, \alpha, z)$ *für alle* $a \in \Sigma$
- *Anfangskonfiguration für* $w \in \Sigma^*$*:* (w, S, ε)
- *erfolgreiche Endkonfiguration:* $(\varepsilon, \varepsilon, z)$ *falls* $w \in L(\mathcal{G})$

Das Zustandsalphabet entfällt, da der Automat mit einem Kontrollzustand arbeitet.

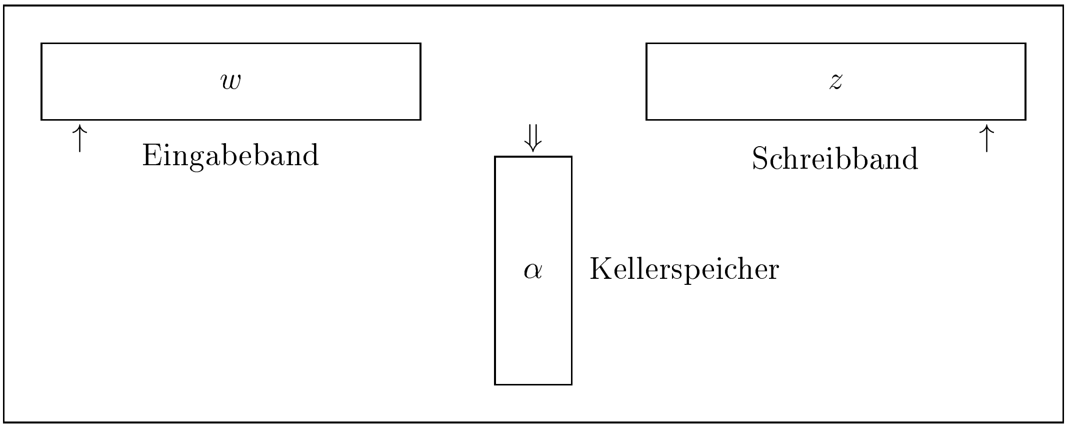


Abbildung 2.1: Schema des $NTA(\mathcal{G})$

Beachte: Der $NTA(\mathcal{G})$ ist nicht-deterministisch, wegen $A \to \beta \mid \gamma$.

$\boxed{\textbf{2-3}}$

Satz 2.8 *Der* $NTA(\mathcal{G})$ *berechnet l-Analysen, d.h.* $(w, S, \varepsilon) \overset{*}{\vdash} (\varepsilon, \varepsilon, z) \quad :\curvearrowright \quad z$ *ist l-Analyse von* w*.*

Beweis 2.9 *Es müssen 2 Richtungen bewiesen werden.*

- *"$\curvearrowright$": Der Automat arbeitet korrekt.*

$$(w, \alpha, y) \vdash (\varepsilon, \varepsilon, yz) \quad \curvearrowright \quad \alpha \overset{z}{\Rightarrow}_l w$$

  Beweis dieser Aussage mittels Induktion über $k := |z|$*.*

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

– $k \hookrightarrow k+1$: *Es sei* $z = iz'$, $\alpha = uA\beta$, $w = uv$, $\pi_i = A \to \gamma$. *Es gilt:*

$$\begin{array}{rcl} (w,\alpha,y) = (uv, uA\beta, y) & \overset{*}{\vdash} & (v, A\beta, y) \\ & \vdash & (v, \gamma\beta, yi) \\ I.V. & \overset{*}{\vdash} & (\varepsilon, \varepsilon, yiz') \end{array}$$

nach Induktionsvoraussetzung also $\gamma\beta \overset{z'}{\Rightarrow}_l v$. *Damit folgt die Behauptung wegen* $\alpha = uA\beta \overset{i}{\Rightarrow}_l u\gamma\beta \overset{z'}{\Rightarrow}_l uv = w$, *also:* $\alpha \overset{z}{\Rightarrow}_l w$

- "$\curvearrowleft$": *Zeige Rückrichtung der Behauptung. Beweis mittels Induktion über* $k = |z|$.

– $k = 0$: $\alpha \overset{\varepsilon}{\Rightarrow}_l w \quad \curvearrowright \quad \alpha = w$. *Es ergibt sich:* $(w,\alpha,y) = (w,w,y) \overset{*}{\vdash} (\varepsilon,\varepsilon,y)$.

– $k \hookrightarrow k+1$: $\alpha = vA\beta \overset{i}{\Rightarrow}_l v\gamma\beta \overset{z'}{\Rightarrow}_l w = vu$ *mit* $\pi_i = A \to \gamma$. *Außerdem gilt* $v\gamma\beta \overset{z'}{\Rightarrow}_l vu$ *wegen* $\gamma\beta \overset{z'}{\Rightarrow}_l u$ *hergeleitet aus* $v\gamma\beta$ *und* $w = vu$.

$$\begin{array}{rcl} (w,\alpha,y) = (vu, vA\beta, y) & \overset{*}{\vdash} & (u, A\beta, y) \\ & \vdash & (u, \gamma\beta, yi) \\ I.V. & \overset{*}{\vdash} & (\varepsilon, \varepsilon, yiz') \end{array}$$

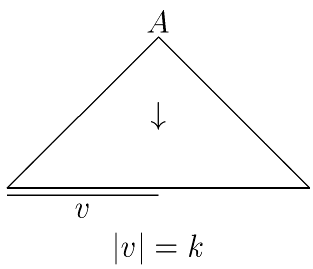
Dabei wurde die Induktionsvoraussetzung auf $\gamma\beta \overset{z'}{\Rightarrow}_l u$ *angewendet.*

□

Das Ziel ist nun, den Nicht-Determinismus des $NTA(\mathcal{G})$ durch **k-look-ahead** auf der Eingabe zu beseitigen. ($k \in \mathbb{N}$)

Definition 2.10 (<u>first</u>-Mengen) *Sei* $\mathcal{G} \in CFG, \alpha \in \mathcal{X}^*$ *und* $k \in \mathbb{N}$. *Wir definieren* $\underline{first}_k(\alpha) \subseteq \Sigma^*$ *durch*

$$\underline{first}_k(\alpha) := \{v \in \Sigma^* \mid \exists w \in \Sigma^* : \alpha \overset{*}{\Rightarrow} vw, |v| = k\} \cup \{v \in \Sigma^* \mid \alpha \overset{*}{\Rightarrow} v, |v| < k\}$$



Folgerung 2.11 *Es ergeben sich folgende Folgerungen:*

1. $\underline{first}_k(\alpha) \neq \emptyset$, *weil* $\mathcal{G}$ *reduziert ist.*

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

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

4. $v \in \underline{first}_k(\alpha) \quad \leftrightarrow \quad \exists x \in \Sigma^* : \; \alpha \overset{*}{\Rightarrow}_l x, \; \{v\} = \underline{first}_k(x)$

Der Wunsch ist nun, die A-Regeln durch v festzulegen $\rightsquigarrow$ LL(k)-Grammatiken, d.h. "lesen der Eingabe von links nach rechts mit k-look-ahead, Berechnung einer Linksanalyse".

Definition 2.12 *Sei $\mathcal{G} \in CFG$ und $k \in \mathbb{N}$. $\mathcal{G} \in LL(k)$ genau dann, wenn für alle Ableitungen der Form*

$$\begin{tikzcd}
 & & w\beta\alpha \arrow[r, Rightarrow, "*", "l"'] & wx \\
S \arrow[r, Rightarrow, "*", "l"'] & wA\alpha \arrow[ur, Rightarrow, "l"'] \arrow[dr, Rightarrow, "l"'] & & \\
 & & w\gamma\alpha \arrow[r, Rightarrow, "*", "r"'] & wy
\end{tikzcd}$$

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

Bemerkung 2.13 *Dazu zwei Bemerkungen:*

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

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

Lemma 2.14 *$\mathcal{G} \in LL(k) \leftrightarrow$ Für alle Linksableitungen der Form*

$$\begin{tikzcd}
 & & w\beta\alpha \\
S \arrow[r, Rightarrow, "*", "l"'] & wA\alpha \arrow[ur, Rightarrow, "l"'] \arrow[dr, Rightarrow, "l"'] & \\
 & & w\gamma\alpha
\end{tikzcd}$$

mit $\beta \neq \gamma$ gilt: $\underline{first}_k(\beta\alpha) \cap \underline{first}_k(\gamma\alpha) = \emptyset$

Beweis 2.15 *Es müssen erneut beide Richtungen gezeigt werden:*

- *Definition $\curvearrowright$ Lemma:*
  Angenommen, die Definitionseigenschaft und die Lemmavoraussetzung gelten, aber $v \in \underline{first}_k(\beta\alpha) \cap \underline{first}_k(\gamma\alpha)$.

  Dann muß $\beta\alpha \overset{*}{\Rightarrow}_l x$ und $\gamma\alpha \overset{*}{\Rightarrow}_l y$ sein mit $\underline{first}_k(x) = \underline{first}_k(y)$. Also nach Definition $\beta = \gamma$, was ein Widerspruch ist.

- *Lemma* $\curvearrowright$ *Definition:*
  Angenommen, es gelten die Lemmaeigenschaft und die Voraussetzung der Definitionseigenschaft und $\beta \neq \gamma$. *Daraus folgt* $\underline{first}_k(\beta\alpha) \cap \underline{first}_k(\gamma\alpha) = \emptyset$.

  Dies führt aber sofort zu einem Widerspruch, weil $\underline{first}_k(x) \subseteq \underline{first}_k(\beta\alpha)$.

$\square$

Das Problem dabei ist die Abhängigkeit der look-ahead-Menge vom Rechtskontext α. Das Ziel ist die Bestimmung der look-ahead-Menge aus Regel allein. Die Idee ist dann, mögliche Rechtskontexte zu vereinigen.

Definition 2.16 (follow-Menge) *Sei* $\mathcal{G} \in CFG$, $A \in N$ *und* $k \in N$. *Wir definieren* $\underline{follow}_k(A) \subseteq \Sigma^*$ *durch*

$$\underline{follow}_k(A) := \{v \in \Sigma^* \mid S \stackrel{*}{\Rightarrow}_l wA\alpha, v \in \underline{first}_k(\alpha)\}$$

2.1.1 Der Fall $k = 1$

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

Satz 2.17 *Für* $\mathcal{G} \in CFG$ *(*$\mathcal{G}$ *reduziert) gilt:* $\mathcal{G} \in LL(1) :\curvearrowleft$ *für alle Regelpaare* $A \to \beta|\gamma$ *mit* $\beta \neq \gamma$ *folgt:*

$$\underline{first}_1(\beta\underline{follow}_1(A)) \cap \underline{first}_1(\gamma\underline{follow}_1(A)) = \emptyset$$

Zur Vereinfachung der Schreibweise schreibt man auch: $\underline{fi} := \underline{first}_1$ und $\underline{fo} := \underline{follow}_1$.

Definition 2.18 $\underline{la}(A \to \beta) := \underline{fi}(\beta\underline{fo}(A))$ *heißt* ***look-ahead-Menge*** *von* $A \to \beta$.

Beachte dabei:

- $\underline{fi}(\alpha) \subseteq \Sigma_\varepsilon (\Sigma_\varepsilon := \Sigma \cup \varepsilon)$
- $\underline{fo}(A) \subseteq \Sigma_\varepsilon$
- $\beta\underline{fo}(A) \subseteq \mathcal{X}^*$
- für $\Gamma \subseteq \mathcal{X}^*$ ist $\underline{fi}(\Gamma) := \bigcup_{\alpha\in\Gamma} \underline{fi}(\alpha)$
- $\underline{la}(A \to \beta) = \underline{fi}(\beta\underline{fo}(A)) \subseteq \Sigma_\varepsilon$

Eigenschaften:

- $\varepsilon \in \underline{fi}(\beta\underline{fo}(A)) \curvearrowleft \beta \stackrel{*}{\Rightarrow} \varepsilon$ und $\varepsilon \in \underline{fo}(A)$
- $a \in \underline{fi}(\beta\,\underline{fo}(A)) \curvearrowleft a \in \underline{fi}(\beta)$ oder $[\beta \stackrel{*}{\Rightarrow} \varepsilon$ und $a \in \underline{fo}(A)]$

Satz 2.19 *Für* $\mathcal{G} \in CFG$ *gilt:* $\mathcal{G} \in LL(1) :\curvearrowleft \underline{la}(A \to \beta) \cap \underline{la}(A \to \gamma) = \emptyset$ *für alle Regelpaare* $A \to \beta|\gamma$ $(\beta \neq \gamma)$

Beweis 2.20 *Es müssen zwei Richtungen gezeigt werden:*

- *"$\curvearrowright$"*
  $\mathcal{G}$ ist $LL(1)$: Angenommen, es gibt $A \to \beta \mid \gamma$ mit $\beta \neq \gamma$ und $c \in \underline{fi}(\beta\underline{fo}(A)) \cap \underline{fi}(\gamma\underline{fo}(A))$ für ein passendes $c \in \Sigma_\varepsilon$.

  Daraus ergeben sich nun 2 Fälle:

  - *1. Fall: $c = \varepsilon$,*
    also $\varepsilon \in \underline{fo}(A) \curvearrowright S \overset{*}{\Rightarrow}_l wA\alpha$ und $\alpha \overset{*}{\Rightarrow} \varepsilon$. Außerdem: $\beta \overset{*}{\Rightarrow}_l \varepsilon$ und $\gamma \overset{*}{\Rightarrow}_l \varepsilon$.
    Das ist aber ein Widerspruch zu $LL(1)$, weil

$$\begin{tikzcd} & & w\beta\alpha \\ S \Rightarrow_l^* & wA\alpha \arrow[ur, Rightarrow, "*", "l"'] \arrow[dr, Rightarrow, "*", "l"'] & \\ & & w\gamma\alpha \end{tikzcd}$$

    mit $\varepsilon \in \underline{fi}(\beta\alpha) \cap \underline{fi}(\gamma\alpha)$.
  - *2. Fall: $c = a \in \Sigma$*
    Es folgt:
    1. *$a \in \underline{fi}(\beta) \cap \underline{fi}(\gamma)$*
    2. *$a \in \underline{fi}(\beta), \gamma \overset{*}{\Rightarrow}_l \varepsilon$ und $a \in \underline{fo}(A)$*
    3. *$a \in \underline{fi}(\gamma), \beta \overset{*}{\Rightarrow}_l \varepsilon$ und $a \in \underline{fo}(A)$*
    4. *$\beta \overset{*}{\Rightarrow}_l \varepsilon, \gamma \overset{*}{\Rightarrow}_l \varepsilon$ und $a \in \underline{fo}(A)$.*

    Aus (1) ergibt sich, da $\mathcal{G}$ reduziert ist, eine Ableitung der Form:

$$\begin{tikzcd} & & w\beta\alpha \\ S \overset{*}{\Rightarrow}_l & wA\alpha \arrow[ur, Rightarrow, "l"'] \arrow[dr, Rightarrow, "l"'] & \\ & & w\gamma\alpha \end{tikzcd}$$

    mit $a \in \underline{fi}(\beta\alpha) \cap \underline{fi}(\gamma\alpha)$, also ein Widerspruch zu $\mathcal{G} \in LL(1)$, da $\beta \neq \gamma$ ist.

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

$$\begin{tikzcd} & & w\beta\alpha \\ S \overset{*}{\Rightarrow}_l & wA\alpha \arrow[ur, Rightarrow, "l"'] \arrow[dr, Rightarrow, "l"'] & \\ & & w\gamma\alpha \end{tikzcd}$$

mit $a \in \underline{fi}(\alpha)$.

Für (2) folgt: $a \in \underline{fi}(\beta)$, *also* $a \in \underline{fi}(\beta\alpha)$, $\gamma \overset{*}{\Rightarrow} \varepsilon$, *also* $a \in \underline{fi}(\gamma\alpha)$.

Analog in (3) und (4).

- ”$\curvearrowleft$”
  Es gilt folgende Ableitung:

$$\begin{tikzcd}[column sep=small] & & w\beta\alpha \\ S \overset{*}{\Rightarrow}_l & wA\alpha \arrow[ur, Rightarrow, "l"'] \arrow[dr, Rightarrow, "l"] & \\ & & w\gamma\alpha \end{tikzcd}$$

mit $\beta \neq \gamma$. *Nach Voraussetzung gilt:* $\underline{fi}(\beta\underline{fo}(A)) \cap \underline{fi}(\gamma\underline{fo}(A)) = \emptyset$. *Da* $\underline{fi}(\beta\alpha) \subseteq \underline{fi}(\beta\underline{fo}(A))$ *und entsprechend* $\underline{fi}(\gamma\alpha) \subseteq \underline{fi}(\gamma\underline{fo}(A))$ *ist, folgt daraus:*

$$\underline{fi}(\beta\alpha) \cap \underline{fi}(\gamma\alpha) = \emptyset$$

□

2.1.2 Berechnung der la-Menge

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

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

Lemma 2.21 *Die Mengen $\underline{fi}(\alpha)$ für $\alpha \in \mathcal{X}^*$ und $\underline{fo}(A)$ für $A \in N$ sind die kleinsten unter den Regeln von (1)-(3) abgeschlossenen Teilmengen von Σ_ε.*

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

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

Bei dieser Grammatik ergeben sich dann folgende Mengen:

- fi- und fo-Mengen:

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

- la-Mengen:

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

Daraus folgt, daß $\mathcal{G}'_{AE} \in LL(1)$, weil die la-Mengen durchschnittfremd sind.

Ein einfacher **LL(1)-Test** ist damit, die la-Mengen zu berechnen und die Alternativen auf Disjunktheit zu überprüfen.

2.1.3 Der deterministische Top-Down-Analyseautomat für $\mathcal{G} \in LL(1)$

Der deterministische Top-Down-Analyseautomat (kurz: $DTA(\mathcal{G})$) verwirklicht folgende Idee: die Zugehörigkeit des Eingabesymbol zu einer la-Menge steuert die Regelauswahl. Man hat 1-look-ahead auf der Eingabe.

Dazu sind folgende Modifikationen des Ableitungsschritts eines $NTA(\mathcal{G})$ notwendig:

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

Folgerung 2.22 *Der Analyseautomat hat eine deterministische Arbeitsweise.*

Beachte: Das Eingabesymbol wird bei Ableitungsschritten nicht gelöscht.

Die Darstellung des $DTA(\mathcal{G})$ durch die **Analysetabelle von $\mathcal{G}$ (<u>action</u>-Funktion von** $\mathcal{G}$):

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

ist definiert durch:

- $\underline{act}(x, A) := (\alpha, i)$ falls $\pi_i = A \to \alpha$ und $x \in \underline{la}(\pi_i)$
- $\underline{act}(a, a) := \underline{pop}$
- $\underline{act}(\varepsilon, \varepsilon) := \underline{accept}$
- $\underline{act}(x, X) := \underline{error}$

$\boxed{\textbf{2-4}}$ $\boxed{\textbf{2-5}}$

Parserkonstruktion mit der Top-Down-Methode

Sei $\mathcal{G} \in CFG(\Sigma)$. Dann verläuft die Parserkonstruktion in folgenden Schritten:

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

Das **Problem** dabei: mehrdeutige Analysetabellen, d.h. $\mathcal{G} \notin LL(1)$.

Transformation in LL(1)

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

1. Beseitigung von Linksrekursionen
2. Links-Faktorisieren

Die Verwendung erfolgt in Parser-erzeugten Systemen.

Aber vorsicht: Transformationen erhalten zwar die Äquivalenz, i.A. aber nicht die syntaktische Struktur, d.h der Ableitungsbaum ändert sich.

2.1.4 Beseitigung von Linksrekursionen

Definition 2.23 *$\mathcal{G} \in CFG$ heißt **linksrekursiv** $:\curvearrowleftright$ $\exists A \in N, \alpha \in \mathcal{X}^* :\ A \overset{+}{\Rightarrow} A\alpha$*

Folgerung 2.24 *$\mathcal{G}$ linksrekursiv $\curvearrowright$ $\mathcal{G}$ ist keine LL(k)-Grammatik ($\forall k \in \mathbb{N}$)*

Der **Grund** ist folgender: wenn ein DTA $A \overset{+}{\Rightarrow} A\alpha$ simuliert, so bleibt der Eingabekopf stehen. D.h. gleicher look-ahead und somit eine Schleife! Es ist also keine Ableitung der Form

$$S \overset{*}{\Rightarrow}_l wA\beta \overset{+}{\Rightarrow}_l wA\alpha\beta \overset{*}{\Rightarrow}_l wv$$

simulierbar.

Beispiel: Die folgende Grammatik $\mathcal{G}_{AE}$ ist linksrekursiv.

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

Der **LL(1)-Test** ergibt dann folgendes:

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

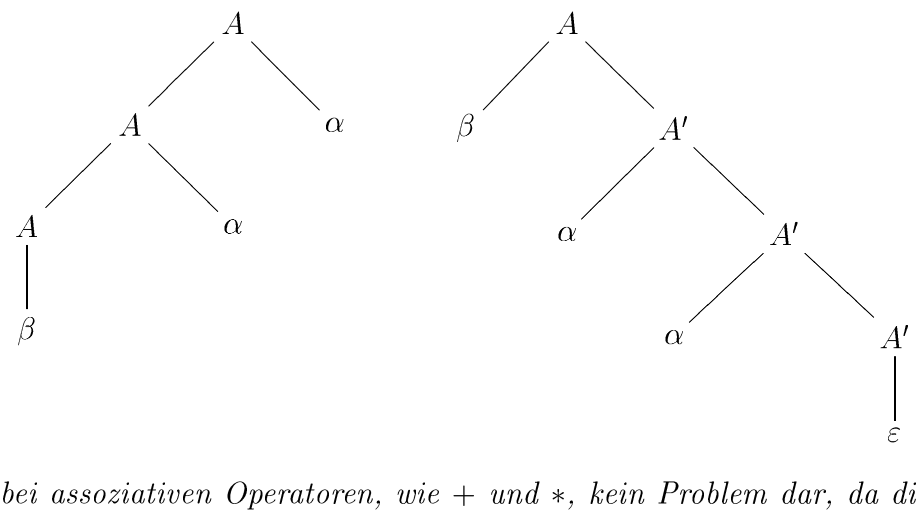
Das bestätigt: $\mathcal{G}_{AE}$ ist keine LL(1)-Grammatik.

Spezialfall: *direkte* Linksrekursion beseitigt man wie folgt: $A \rightarrow A\alpha \mid \beta$, $\beta \neq A\ldots$ und $\alpha \neq \varepsilon$ werden durch

$$\begin{array}{lcl} A & \rightarrow & \beta A' \\ A' & \rightarrow & \alpha A' \mid \varepsilon \end{array}$$

ersetzt.

Folgerung 2.25 *$\mathcal{L}(\mathcal{G})$ ist unverändert, hat jedoch eine geänderte syntaktische Struktur.*



Das stellt bei assoziativen Operatoren, wie $+$ und $*$, kein Problem dar, da die Semantik invariant ist.

Beispiel: Die Beseitigung direkter Linksrekursionen in der Grammatik $\mathcal{G}_{AE}$ ergibt die LL(1)-Grammatik $\mathcal{G}'_{AE}$.

Der allgemeine Fall: indirekte Linksrekursionen:

$$\begin{array}{c} A \to A_1\alpha_1 \mid \ldots \\ A_1 \to A_2\alpha_2 \mid \ldots \\ \vdots \\ A_n \to A\alpha \mid \ldots \end{array}$$

Die Idee ist, die Rekursionen durch Transformation in die Greibach-Normalform (GNF) zu beseitigen. Die GNF besteht nur aus Regeln der Form $A \to aB_1 \ldots B_n$, wobei $B_i \neq S$, oder $S \to \varepsilon$. Dabei ist zu beachten, daß eine Beseitigung von Linksrekursionen nicht notwendigerweise eine LL(1)-Grammatik ergibt! Zwar ist jedes $\mathcal{G} \in CFG$ äquivalent in GNF transformierbar, aber es gilt:

$$\mathcal{L}(LL(k)) \subsetneqq \mathcal{L}(LL(k+1)) \subsetneqq \mathcal{L}(DPDA) \subsetneqq \mathcal{L}(PDA) = CFL$$

Komplexität der LL(1)-Analyse
$\mathcal{G} \in LL(1) \curvearrowright \mathcal{G}$ ist nicht linksrekursiv. Der $DTA(\mathcal{G})$ mit Eingabe von $w \in \Sigma^*$ macht

- $|w|$ Vergleichsschritte zuzüglich eines Erkennungsschrittes
- maximal $|N|$ aufeinanderfolgende Ableitungsschritte.

D.h. der Automat hat maximal $|N| \cdot (|w|+1)$ Transitionen und eine maximale Kellerlänge $max\{|\alpha| \mid A \to \alpha \in P\} \cdot |N| \cdot (|w|+1)$, also **linearen Platz- und Zeitbedarf**.

2.1.5 Links-Faktorisieren

Es sei folgende Beispielgrammatik gegeben:

$$\begin{array}{rcl} statement & \to & \underline{if}\ cond\ \underline{then}\ statement\ \underline{else}\ statement \ \mid \\ & & \underline{if}\ cond\ \underline{then}\ statement \end{array}$$

Die Idee ist nun, die Entscheidung zu verschieben, bis eine Alternative erkennbar ist, d.h. $A \to \alpha\gamma \mid \alpha\beta$ ersetzen durch $A \to \alpha A'$ und $A' \to \beta \mid \gamma$.

2.1.6 Top-Down-Analyse mit rekursiven Prozeduren

Die Idee ist, statt der expliziten Kellerbenutzung den Laufzeitkeller durch Einsatz rekursiver Prozeduren implizit zu nutzen. Der Vorteil ist laut Wirth die leichte Programmierung. Der Spezialfall: $\mathcal{G} \in LL(1)$.

1. Analyseverfahren durch rekursiven Abstieg ohne <u>la</u>-Mengen ("Recursive descent parser")
   Methode: $A \in N \mapsto \mathtt{A}()$ parameterlose Prozedur zur Simulation eines Ableitungsschritts.
   Annahme: Alternativen sind durch Eingabesymbol unterscheidbar
   Eingabe: `sym` als Variable für das Eingabesymbol, `nextsym` zum Lesen des nächsten Eingabesymbols.
   Ausgabe: `print(i)` zur Ausgabe der Regelnummer

   $\boxed{\textbf{2-6}}$ $\boxed{\textbf{2-6a}}$

$$\left.\begin{array}{l} A \to B \mid C \\ B \to b \\ C \to c \end{array}\right\} \text{erfordert Benutzung von } \underline{\text{la}}\text{-Mengen}$$

2. Zusätzliche Verwendung der <u>la</u>-Mengen. Der Vorteil ist die bessere Kontrolle der Regelanwendungen und die frühere Fehlererkennung.

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

Die 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 (rechte Seite durch linke ersetzen)

$\rightsquigarrow$ **Shift-Reduce-Verfahren**

$\boxed{\textbf{2-8}}$

Definition 2.26 *Der nichtdeterministische Bottom-Up-Analyseautomat von $\mathcal{G} \in CFG$ (Bezeichnung: $NBA(\mathcal{G})$) ist wie folgt definiert:*

- *Eingabealphabet:* Σ
- *Kelleralphabet:* $\mathcal{X}$
- *Ausgabealphabet:* $[p]$
- *Konfigurationsmenge:* $\mathcal{X}^* \times \Sigma^* \times [p]^*$ *(Kellerspitze rechts!)*
- *Transitionen:*
  - *Shift-Schritt:* $(\alpha, aw, z) \vdash (\alpha a, w, z)$ *für alle* $a \in \Sigma$

- *Reduce-Schritt:* $(\beta\alpha, w, z) \vdash (\beta A, w, zi)$ *falls i-te Regel* $\pi_i = A \to \alpha$
- *Anfangskonfiguration:* $(\varepsilon, w, \varepsilon)$ *für ein* $w \in \Sigma^*$

Satz 2.27 *Der* $NBA(\mathcal{G})$ *berechnet gespiegelte Rechtsanalysen, d.h. für ein Wort* $w \in \Sigma^*$ *und* $z \in [p]^*$ *gilt:*

$$z \text{ ist Rechtsanalyse von } w \quad \Leftrightarrow \quad (\varepsilon, w, \varepsilon) \overset{*}{\vdash} (S, \varepsilon, \overleftarrow{z})$$

Nicht-Determinismus:

1. Shift- oder Reduce-Schritt ?
2. Reduce-Schritt: wo ist der linke Henkelrand ? ($A \to \alpha$, dabei bezeichne α den Henkel)
3. Reduce-Schritt: linke Regelseite ?
4. Analyseende ?

Das Ziel ist es, den Nicht-Determinismus durch k-look-ahead auf der Eingabe zu beseitigen (LR(k)-Grammatiken).

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

Folgerung 2.28 (S', ε, z) *ist Endkonfiguration, die nicht erneut in eine weitere Endkonfiguration übergehen kann.*

$\boxed{\textbf{2-9}}$

Beseitigung des restlichen Nichtdeterminismus durch LR(k)-Grammatiken

Definition 2.29 (LR(k)-Grammatiken) *Sei* $\mathcal{G} \in CFG$, *startsepariert durch* $S' \to S, k \in \mathbb{N}$. *Dann gilt:* $\mathcal{G} \in LR(k) \Leftrightarrow$ *für alle Rechtsableitungen der Form*

$$\begin{tikzcd}
 & \alpha A w \arrow[r, Rightarrow, "r"'] & \alpha\beta w \\
S \arrow[ur, Rightarrow, "*", "r"'] \arrow[dr, Rightarrow, "*", "r"'] & & \\
 & \alpha' A' w' \arrow[r, Rightarrow, "r"'] & \alpha\beta v
\end{tikzcd}$$

mit $\underline{first}_k(w) = \underline{first}_k(v)$ *gilt:* $\alpha' = \alpha, \quad A' = A, \quad w' = v.$

Folgerung 2.30 *Der Bottom-Up-Analyseautomat kann mit k-look-ahead auf der Eingabe die Transitionen entscheiden.*

2.2.1 LR(0)-Grammatiken

Mit $k = 0$ erfolgt die Entscheidung immer ohne look-ahead, also allein durch den Kellerinhalt $\alpha\beta$. Es erfolgt eine Abstraktion endlicher Informationen aus $\alpha\beta$, welche für die Entscheidung ausreicht.

Definition 2.31 (LR(0)-Auskünfte, LR(0)-Mengen) *Sei $\mathcal{G} \in CFG$ mit $S' \to S$ und $S' \overset{*}{\Rightarrow}_r \alpha A w \Rightarrow \alpha\beta_1\beta_2 w$. Dann heißt $[A \to \beta_1 \cdot \beta_2]$ eine **LR(0)-Auskunft für** $\alpha\beta_1$. Für $\gamma = \mathcal{X}^*$ bezeichnet $\underline{\mathrm{LR}(0)}(\gamma)$ die Menge aller LR(0)-Auskünfte für γ, die sogenannte **LR(0)-Menge von** γ.*

$\boxed{\textbf{2-10}}$

Folgerung 2.32 *Daraus läßt sich folgern:*

1. *$\underline{LR(0)}(\gamma)$ ist endlich.*
2. *$\underline{LR(0)}(\mathcal{G}) := \{\underline{LR(0)}(\gamma) \mid \gamma \in \mathcal{X}^*\}$ ist endlich.*
3. *$[A \to \beta_1 \cdot] \in \underline{LR(0)}(\gamma)$ zeigt die Reduktionsmöglichkeit $(\alpha\beta_1, w, z) \vdash (\alpha A, w, zi)$ für $\pi_i = A \to \beta_1$ und $\gamma = \alpha\beta_1$.*
4. *$[A \to \beta_1 \cdot \beta_2] \in \underline{LR(0)}(\gamma)$ mit $\beta_2 \neq \varepsilon$ bedeutet Shift-Möglichkeit wegen unvollständigem Henkel.*
5. *$\mathcal{G} \in LR(0) \quad \curvearrowright \quad$ die $\underline{\text{LR(0)}}$-Mengen enthalten keine widersprüchlichen Auskünfte.*

2.2.2 Berechnung der $\underline{\textbf{LR(0)}}$-Mengen einer Grammatik

Satz 2.33 *$\mathcal{G} \in CFG$ mit $S' \to S$. Sei $\mathcal{G}$ ferner reduziert. Dann gilt:*

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

$\boxed{\textbf{2-11}}$ zeigt, keine widersprüchlichen Auskünfte, also ist $\mathcal{G} \in \underline{LR(0)}$-Grammatik.

Die $\underline{\text{goto}}$-Funktion

Wenn $\mathcal{G}$ eine LR(0)-Grammatik ist, so folgt daraus, daß $\underline{LR(0)}(\gamma)$ die Shift/Reduce-Entscheidung für den Bottom-Up-Analyseautomaten mit Kellerinhalt γ liefert. Dann ist $\underline{LR(0)}(\mathcal{G})$ das neue Kelleralphabet (statt $\mathcal{X}$)!

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

Die $\underline{\text{goto}}$-Funktion $\underline{goto}(I, X) : \underline{LR(0)}(\mathcal{G}) \times \mathcal{X} \to \underline{LR(0)}(\mathcal{G})$ ist definiert durch:

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

Rückblick

$\boxed{\textbf{2-9}}$

$\mathcal{G} \in LR(0) \quad \curvearrowright \quad$ für alle

$$\begin{tikzcd}
 & \alpha A w \arrow[r, Rightarrow, "r"'] & \alpha\beta w \\
S \arrow[ur, Rightarrow, "*", "r"'] \arrow[dr, Rightarrow, "*", "r"'] & & \\
 & \alpha' A' w' \arrow[r, Rightarrow, "r"'] & \alpha\beta v
\end{tikzcd}$$

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

- $[A \to \beta\cdot] \in \underline{LR(0)}(\alpha\beta)$
- $[A \to \beta_1 \cdot \beta_2] \in \underline{LR(0)}(\alpha\beta_1)$

Sei beispielsweise folgende Ableitung gegeben: $S' \Rightarrow_r S \Rightarrow_r A \Rightarrow a$. Dann gilt:
$\underline{LR(0)}(\varepsilon) : \quad \alpha = \varepsilon, \quad \beta_1 = \varepsilon$

$$\begin{array}{rcl}
S' \to S & \curvearrowright & [S' \to \cdot S] \in \underline{LR(0)}(\varepsilon) \\
S \to A & \curvearrowright & [S \to \cdot A] \in \underline{LR(0)}(\varepsilon) \\
A \to a & \curvearrowright & [A \to \cdot a] \in \underline{LR(0)}(\varepsilon)
\end{array}$$

$\boxed{\textbf{2-10}}$ $\boxed{\textbf{2-11}}$ $\boxed{\textbf{2-12}}$

Berechnung der $\underline{\text{LR(0)}}$-Mengen und der $\underline{\text{goto}}$-Funktion

Die Berechnung der $\underline{\text{LR(0)}}$-Mengen und der $\underline{\text{goto}}$-Funktion erfolgt durch Potenzmengen-kostruktion nichtdeterministischer Automaten.

$\boxed{\textbf{2-13}}$

Sei $\mathcal{G} \in CFG$, $\mathcal{G}$ startsepariert mit $S' \to S$. Dann wird ein $\mathfrak{A}(\mathcal{G}) \in NFA_\varepsilon$ wie folgt konstruiert:

- Zustandsmenge: $Q := \{[A \to \beta_1 \cdot \beta_2] \mid A \to \beta_1\beta_2 \in \mathcal{G}\}$
- Eingabealphabet: $\mathcal{X} := N \cup \Sigma$
- Anfangszustand: $q_0 := [S' \to \cdot S]$
- Endzustandsmenge ist irrelevant $(F = Q)$.
- Transitionsfunktion $\delta : Q \times \mathcal{X}_\varepsilon \to \wp(Q)$ mit:

$$\begin{aligned} \delta([A \to \beta_1 \cdot X\beta_2], X) &\ni [A \to \beta_1 X \cdot \beta_2] \\ \delta([A \to \beta_1 \cdot B\beta_2], \varepsilon) &\ni [B \to \cdot\gamma] \quad \text{falls } B \to \gamma \end{aligned}$$

2-13

Potenzmengenkonstruktion nach Thompson

Kostruktion von $\widehat{\mathfrak{A}(\mathcal{G})} = \left\langle \widehat{Q}, \mathcal{X}, \widehat{\delta}, \widehat{q_0}, \emptyset \right\rangle \in DFA$.
Sei hierzu die erweiterte Transitionsfunktion $\overline{\delta} : \wp(Q) \times \mathcal{X}^* \longrightarrow \wp(Q)$ mit $\overline{\delta}(T, \varepsilon) := \varepsilon(T)$ und $\overline{\delta}(T, wa) := \varepsilon\left(\bigcup_{q \in \overline{\delta}(T,w)} \delta(q, a)\right)$.

- $\widehat{Q} := \{\overline{\delta}(\{[S' \to \cdot S]\}, \alpha) \mid \alpha \in \mathcal{X}^*\}$
- $\widehat{q_0} = \varepsilon(\{[S' \to \cdot S]\})$
- $\widehat{\delta} : \widehat{Q} \times \mathcal{X} \longrightarrow \widehat{Q}$ mit $\widehat{\delta}(T, X) := \overline{\delta}(T, X)$

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

2-14

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

Als Hilfsmittel dienen die Menge $\underline{LR(0)}(\mathcal{G})$ und die goto-Funktion.

Die action-Funktion von $\mathcal{G}$ gibt die Shift-Reduce-Entscheidung an. Sie ist wie folgt definiert:

$$\underline{act} : \underline{LR(0)}(\mathcal{G}) \longrightarrow \{\underline{shift}, \underline{red_i}, \underline{accept}, \underline{error} \mid i \in [p]\}$$

Dabei ist act(I) wie folgt definiert:

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

Die action-Funktion ist bei LR(0) eindeutig.

Die Funktionen act und goto bilden die **LR(0)-Analysetabelle von $\mathcal{G}$**. Diese Tabelle bestimmt den **LR(0)-Analyseautomaten** $DBA(\mathcal{G})$:

- Eingabealphabet: Σ
- Kelleralphabet: $\Gamma := \underline{LR(0)}(\mathcal{G})$
- Ausgabealphabet: $\Delta := [p] \cup \{0\} \cup \{\underline{error}\}$
- Konfigurationsmenge: $\Gamma^* \times \Sigma \times \Delta^*$
- Transitionen:
  - Shift-Schritt:
  $$(\alpha I, aw, z) \vdash (\alpha I I', w, z) \quad \text{falls } \underline{act}(I) = \underline{shift} \text{ und } \underline{goto}(I, a) = I'$$
  - Reduce-Schritt[1]:
  $$(\alpha I, aw, z) \vdash (\widetilde{\widetilde{\alpha} I} I', w, zi) \quad \text{falls } \underline{act}(I) = \underline{redi}$$
  Außerdem muß gelten: $\pi_i = A \to X_1, \ldots, X_n$, $\widetilde{\widetilde{\alpha} I} = \alpha{\cdot}I \mid n$ und $I' = \underline{goto}(\widetilde{I}, A)$.
  - Accept-Schritt:
  $$(I_0 I, \varepsilon, z) \vdash (\varepsilon, \varepsilon, z0) \quad \text{falls } \underline{act}(I) = \underline{accept}$$
  - Fehlererkennung:
  $$(\alpha I, w, z) \vdash (\varepsilon, \varepsilon, z \cdot \underline{error})$$
- Anfangskonfiguration: (I_0, w, ε) mit $I_0 = \underline{LR(0)}(\varepsilon)$ für ein $w \in \Sigma^*$

Folgerung 2.34 *Wenn $\underline{LR(0)}(\mathcal{G})$ konfliktfrei ist, also die action-Funktion eindeutig ist , so arbeitet der LR(0)-Analyseautomat deterministisch und es gilt für $w \in \Sigma^*$ und $z \in [p]^*$:*

- $(I_0, w, \varepsilon) \overset{*}{\vdash} (\varepsilon, \varepsilon, z0) \quad \curvearrowright \quad \overleftarrow{z}$ *ist r-Analyse von w.*
- $(I_0, w, \varepsilon) \overset{*}{\vdash} (\varepsilon, \varepsilon, z \cdot \underline{error}) \quad \curvearrowright \quad w \notin L(\mathcal{G})$.

Beispiel:
Sei $\mathcal{G}$ gegeben wie in $\boxed{\textbf{2-12}}$. $w = aac$ bestimmt die folgende Berechnung:

$$\begin{array}{rcl}
(I_0, aac, \varepsilon) & \vdash & (I_0 I_4, ac, \varepsilon) \vdash (I_0 I_4 I_4, c, \varepsilon) \\
& \vdash & (I_0 I_4 I_4 I_6, \varepsilon, \varepsilon) \vdash (I_0 I_4 I_4 I_8, \varepsilon, 6) \\
& \vdash & (I_0 I_4 I_8, \varepsilon, 65) \vdash (I_0 I_3, \varepsilon, 655) \\
& \vdash & (I_0 I_1, \varepsilon, 6552) \vdash (\varepsilon, \varepsilon, 65520) \\
& \curvearrowright & 02556
\end{array}$$

Kontrolle für 02556:
$$S' \xrightarrow{0} S \xrightarrow{2} C \xrightarrow{5} aC \xrightarrow{5} aaC \xrightarrow{6} aac$$

[1]Bemerkung zur Notation: $wa_1 \ldots a_n \mid n = w$, d.h. der Ausdruck ohne die letzten n Stellen.

2.2.4 SLR(1)-Analyse

SR(0) bedeutet simple look-ahead mit einem Symbol. (Beispiel: $\boxed{\textbf{2-15}}$) Dabei treten Konflikte auf. Man unterscheidet folgende Konflikttypen:

- Reduce-Reduce-Konflikt (RR)
- Shift-Reduce-Konflikt (SR)

Im Beispiel finden sich nur SR-Konflikte.

Das Ziel soll sein, die Konflikte durch das Eingabesymbol zu beseitigen.

Beobachtung:

1. $[A \to \beta_1 \cdot a\beta_2] \in \underline{LR(0)}(\alpha\beta_1) \quad \curvearrowright \quad S \overset{*}{\Rightarrow}_r \alpha A w \Rightarrow_r \alpha\beta_1 a \beta_2 w$
   Also: shift nur bei Eingabe von a.
2. $[A \to \beta\cdot] \in \underline{LR(0)}(\alpha\beta) \quad \curvearrowright \quad S \overset{*}{\Rightarrow}_r \alpha A a w \Rightarrow_r \alpha\beta a w \quad \curvearrowright \quad a \in \underline{fo}(A)$
   Also: reduce mit $A \to \beta$ nur, falls $a \in \underline{fo}(A)$.

Für das Beispiel von $\boxed{\textbf{2-15}}$ gilt dann:

I_1: shift bei Eingabe von $+$
accept bei Eingabe von \$ (rechter Eingaberand, entspricht dem leeren Wort)

I_2: shift bei Eingabe von $*$
reduce bei Eingabe von $\underline{fo}(E) = \{\$, +,)\}$

I_9: shift bei Eingabe von $*$
reduce bei Eingabe von $\$, +,)$

Damit sind die Konflikte beseitigt. Ein weitere Vorteil ist die frühere Fehlererkennung durch genauere Kontrolle der Aktion.

SLR(1)-action-Funktion

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

$$\underline{act} : \underline{LR(0)}(\mathcal{G}) \times (\Sigma \cup \{\$\}) \longrightarrow \{\underline{shift}, \underline{redi}, \underline{accept}, \underline{error} \mid 1 \leq i \leq r\}$$

sei definiert durch:

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

Definition 2.35 $\mathcal{G} \in SLR(1) \quad :\curvearrowright \quad \underline{act}(I, a)$ *ist eindeutig.*

$\boxed{\textbf{2-16}}$

2.2.5 LR(1)-Analyse

Nicht immer sind Konflikte über follow-Mengen lösbar. Als Beispiel siehe **2-17**. Eine Verfeinerung der LR(0)-Auskünfte kann durch Mitführen der möglichen look-ahead-Symbole erreicht werden.

Definition 2.36 (LR(1)-Auskünfte und -Mengen für $\mathcal{G} \in CFG$) *Wir definieren:*

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

Insgesamt gilt: $\underline{LR(1)}(\mathcal{G}) := \{\underline{LR(1)}(\gamma) \mid \gamma \in \mathcal{X}^*\}$.

2-17

Berechnung der LR(1)-Mengen

Modifikation der Berechnung von $\underline{LR(0)}(\mathcal{G})$ unter Berücksichtigung des Rechtskontextes. Im folgenden gelte $x \in \Sigma \cup \{\$\}$.

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

2-18

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

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

$$\underline{act} : \underline{LR(1)}(\mathcal{G}) \times (\Sigma \cup \{\$\}) \longrightarrow \{\underline{shift}, \underline{red i}, \underline{accept}, \underline{error} \mid 1 \leq i \leq r\}$$

sei definiert durch:

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

Definition 2.37 $\mathcal{G} \in LR(1) \quad :\curvearrowright \quad \underline{act}(I, x)$ *ist eindeutig (konfliktfrei).*

2-19 **2-20**

2.2.6 LALR(1)-Analyse

Beseitigung von Entscheidungskonflikten nach LR(1) ist zu aufwendig. So ergibt sich zum Beispiel:

	für obiges Beispiel	ALGOL 60
$\|\underline{LR(0)}(\mathcal{G})\|$	11	~ 100
$\|\underline{LR(1)}(\mathcal{G})\|$	15	~ 1000

Beobachtung dabei: mögliche Informationsredundanz bei $\underline{LR(1)}(\mathcal{G}_{L/R})$.

Definition 2.38 *$I_1, I_2 \in \underline{LR(1)}(\mathcal{G})$ heißen **LR(0)-äquivalent**, $I_1 \sim_0 I_2$ falls die "LR(0)-Anteile" von I_1 und I_2 sind gleich.*

In unserem obigem Beispiel gilt zum Beispiel:

$$I'_4 \sim_0 I'_{11},\ I'_5 \sim_0 I'_{12},\ I'_7 \sim_0 I'_{13},\ I'_8 \sim_0 I'_{10}$$

Folgerung 2.39 $|\underline{LR(1)}(\mathcal{G})/\sim_0| = |\underline{LR(0)}(\mathcal{G})|$

Oft können LR(0)-äquivalente LR(1)-Informationen vereinigt werden, ohne daß die Konfliktlösbarkeit verloren geht.

Definition 2.40 *$I \in \underline{LR(1)}(\mathcal{G})$ bestimmt eine $\underline{LALR(1)}$-Menge*

$$\bigcup\{I' \in \underline{LR(1)}(\mathcal{G}) \mid I' \sim_0 I\}$$

Es gilt: $|\underline{LALR(1)}(\mathcal{G})| = |\underline{LR(0)}(\mathcal{G})|$

Allerdings enthalten $\underline{\text{LALR(1)}}$-Mengen im Unterschied zu $\underline{\text{LR(0)}}$-Mengen look-ahead-Symbole zur Lösung von Konflikten.

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

Definition 2.41 $\mathcal{G} \in \underline{LALR(1)}(\mathcal{G})$ $:\curvearrowleft$ *$\underline{LALR(1)}$-action-Funktion ist eindeutig.*

$\boxed{\textbf{2-21}}$

In diesem Beispiel ist die $\underline{LALR(1)}(\mathcal{G})$-Menge von $\mathcal{G}_{L/R}$:

$$\underline{LALR(1)}(\mathcal{G}_{L/R}) := \{I_i^{1/0} \mid 1 \leq i \leq 9\}$$

Also ergibt sich:

$$I_i^{(1/0)} := I'_i \text{ für } i = 0, 1, 2, 3, 6, 9$$

Für die übrigen Elemente gilt:

$$\begin{aligned}
I_4^{(1/0)} &= I'_4 \cup I'_{11} : [L \to * \cdot R, = /\$], [R \to \cdot L, = /\$], [L \to \cdot * R, = /\$], [L \to \cdot a, = /\$] \\
I_5^{(1/0)} &= I'_5 \cup I'_{12} : [L \to a \cdot, = /\$] \\
I_7^{(1/0)} &= I'_7 \cup I'_{13} : [L \to * R \cdot, = /\$] \\
I_8^{(1/0)} &= I'_8 \cup I'_{10} : [R \to L \cdot, = /\$]
\end{aligned}$$

$\boxed{\textbf{2-21}}$

Die LR(1)-action-Funktion zeigt, daß beim Übergang zu LALR(()1) keine Konflikte auftreten: $\mathcal{G}_{L/R} \in LALR(1)$.

Die LR(1)-goto-Funktion überträgt sich auf $\underline{LALR(1)}(\mathcal{G})$, weil für LR(1)-Mengen I_1 und I_2 gilt:

$$I_1 \sim_0 I_2 \quad \curvearrowright \quad \underline{goto}(I_1, x) \sim_0 \underline{goto}(I_2, x)$$

Der Grund: der "LR(0)-Kern" von $\underline{LR(1)}(\alpha X)$ ist durch den "LR(0)-Kern" von $\underline{LR(1)}(\alpha)$ und X vollständig bestimmt.

2.3 Bottom-Up-Analyse mehrdeutiger Grammatiken

Es gilt für eine Grammatik $\mathcal{G} \in CFG$:

$$\mathcal{G} \text{ mehrdeutig} \quad \curvearrowleft\!\!\curvearrowright \quad \mathcal{G} \notin LR = \bigcup_{k \in \mathbb{N}} LR(k)$$

Die Mehrdeutigkeit ist ein natürliches Beschreibungmittel zur Vermeidung aufwendiger Klammerung. Die Auflösung erfolgt durch Regeln für Präzedenz und Assoziativität von Operationssymbolen (allgemeiner: von sytaktischen Konstruktionen).

Ein Beispiel dafür:

$$\mathcal{G}^m_{AE}: \quad E \to E + E \mid E * E \mid (E) \mid id$$

Mehrdeutige Grammatiken

Für $\mathcal{G}^m_{AE}$ gilt die Präzedenz (* vor +) und die Assoziativität links. In dem Beispiel ergeben sich dann folgende Konflikte:

- I_1: SLR(1)-lösbar
- I_7, I_8: nicht LR-auflösbar

Das wird in folgender Beispielrechnung sichtbar:

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

2-20

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

$$S \to iSeS \mid iS \mid a$$

2-21

Es ergeben sich daraus 2 Möglichkeiten einen Ausdruck

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

zu klammern.

$$\begin{array}{ll} \underline{if}\ b\ \underline{then}\ (\underline{if}\ b\ \underline{then}\ b)\ \underline{else}\ a & \text{1. Möglichkeit} \\ \underline{if}\ b\ \underline{then}\ (\underline{if}\ b\ \underline{then}\ b\ \underline{else}\ a) & \text{2. Möglichkeit} \end{array}$$

Die zweite Möglichkeit soll hier als die korrekte angenommen werden.

2-22

$\underline{act}(I_4, e) = \underline{shift}$ folgt aus der 2. Möglichkeit.

Kapitel 3

Semantische Analyse, Attributgrammatiken

Das Ergebnis der syntaktischen Analyse ist ein Ableitungsbaum. Die folgenden kontextabhängigen Eigenschaften sind jedoch nicht durch kontextfreie Grammatiken beschreibbar:

- Deklariertheit von Bezeichnern
- Typinformationen

Die Festlegung dieser Eigenschaften geschieht durch:

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

Eine **statische Semantik** sind kontextabhängige, aber laufzeitunabhängige Eigenschaften eines Programms. Die formale Beschreibung erfolgt durch **Attributgrammatiken**.

Die Idee:

$$\text{CFG zzgl. semantischer Regeln} \quad \rightsquigarrow \quad \text{Zusatzinformationen für den Ableitungsbaum}$$

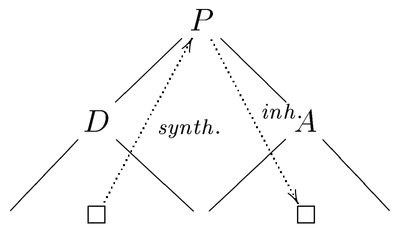
Die semantische Analyse kommt der Attributberechnung gleich. Das Ergebnis der semantischen Analyse ist ein attributierter Ableitungsbaum, der die Grundlage bildet für die anschließende Synthese (Codegenerierung).

3.0.1 Attributgrammatiken

Die Idee: Attribute für $A \in N$, semantische Regeln für ihre Berechnung:

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

Daraus folgt ein beliebiger Informationstransfer im Ableitungsbaum.



Attributwerte: Symboltabellen, Typen, Code, Fehler, ...

Attributgrammatiken haben eine breite Anwendbarkeit, syntaxgerichtete Programmierung. Automatische Attributauswertung erfolgt in Compilergeneratoren. Die Attributierung geht auf Donald Knuth aus dem Jahr 1968 zurück.

$\boxed{\textbf{3-1}}$

Die Attribute berechnen sich nach folgenden Regeln:

$$
\begin{array}{lcl}
\mathcal{G}_B: & B \to 0 & v.0 = 0 \\
& B \to 1 & v.0 = 1 \\
& L \to B & v.0 = v.1 \\
& & l.0 = 1 \\
& L \to LB & v.0 = 2 \cdot v.1 + v.2 \\
& & l.0 = l.1 + 1 \\
& N \to L & v.0 = v.1 \\
& N \to L \cdot L & v.0 = v.1 + \dfrac{v.3}{2^{l.3}}
\end{array}
$$

Je nach Zählung der Terminale und Nichtterminale kann die letzte Zeile auch

$$N \to L \cdot L \quad v.0 = v.1 + \frac{v.2}{2^{l.2}}$$

geschrieben werden. $\mathcal{G}_B$ erzeugt Binärzahlen mit und ohne Punkt, wobei N das Startsymbol ist. Dabei werden die synthetischen Attribute wie folgt verwendet:

- B,N: v (value)
- L: v,l (value,length)

Die semantischen Regeln sind dann Attributgleichungen mit Attributvariablen. Der Index i bezeichnet dabei das i-te Symbol. Das Ziel dabei ist die Bestimmung des Zahlenwertes:

- Attributwerte von v: rationale Zahlen ($A^v = \mathbb{Q}$)
- Attributwerte von l: natürliche Zahlen ($A^l = \mathbb{N}$)

Attributierung von $\mathcal{G}_B$ mit synthetischen und inheriten Attributen
Es wird ein zusätzliches **inherites Attribut** für Bits und Listen verwendet: p (position).

$\boxed{\textbf{3-3}}$

Damit ergeben sich die folgenden Regeln:

$$
\begin{array}{rcl}
\mathcal{G}_B: & B \to 0 & v.0 = 0 \\
& B \to 1 & v.0 = 2^{p.0} \\
& L \to B & v.0 = v.1 \\
& & l.0 = 1 \\
& & p.1 = p.0 \\
& L \to LB & v.0 = v.1 + v.2 \\
& & l.0 = l.1 + 1 \\
& & p.1 = p.0 + 1 \\
& & p.2 = p.0 \\
& N \to L & v.0 = v.1 \\
& & p.1 = 0 \\
& N \to L \cdot L & v.0 = v.1 + v.3 \\
& & p.1 = 0 \\
& & p.2 = -l.2
\end{array}
$$

Berechnung des Wurzelattributs:

- Längen aufwärts bis zur Wurzel
- Positionen abwärts zu den Blättern
- Werte aufwärts zur Wurzel

Definition 3.1 (Attributgrammatik) *Sei $\mathcal{G} = \langle N, \Sigma, P, S\rangle \in CFG$. Sei $\underline{Att}$ eine Menge von **Attributen**, $A = (A^\alpha \mid \alpha \in \underline{Att})$ eine Familie von **Attributwertmengen** und $\underline{att} : \mathcal{X} \to \wp(\underline{Att})$ eine **Attributzuordnung**.*
Sei $\underline{Att} = \underline{Syn} \cup \underline{Inh}$ eine Zerlegung in Teilmengen synthetischer *und* inheriter *Attribute, so daß $\underline{att}$ zerfällt in*

$$\underline{syn} : \mathcal{X} \to \wp(\underline{Syn}) \text{ mit } \underline{syn}(X) = \underline{att}(X) \cap \underline{Syn}$$

und

$$\underline{inh} : \mathcal{X} \to \wp(\underline{Inh}) \text{ mit } \underline{inh}(X) = \underline{att}(X) \cap \underline{Inh}$$

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

$$\underline{IVar}_\pi := \{\alpha.i \mid (i = 0 \text{ und } \alpha \in \underline{syn}(X_0)) \text{ oder } (1 \leq i \leq r \text{ und } \alpha \in \underline{inh}(X_i))\}$$

und

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

*der **Innen-** und **Außenvariablen**.*
*Eine **Attributgleichung von** π (semantische Regel) hat die Form*

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

mit der Eigenschaft $\alpha.i \in \underline{IVar}_\pi, \alpha_1.i_1 \ldots \alpha_n.i_n \in \underline{OVar}_\pi$, *also*

$$f : A^{\alpha_i} \times A^{\alpha_2} \times \ldots \times A^{\alpha_n} \to A^\alpha \text{ und } n \in \mathbb{N}$$

*Sei E_π eine (endliche) Menge von Attributgleichungen, in der jede Innenvariable genau einmal vorkommt, dann heißt $\mathfrak{A}\langle \mathcal{G}, (E_\pi \mid \pi \in P)\rangle$ eine **Attributgrammatik**: $\mathfrak{A} \in AG$.*

3-3

Definition 3.2 (Attributgleichungssystem) *Sei $\mathfrak{A}\langle \mathcal{G}, (E_\pi \mid \pi \in P)\rangle \in AG$. $\mathfrak{A}$ induziert für jeden Ableitungsbaum t von $\mathcal{G}$ ein **Attributgleichungssystem** E_t:*
Sei $\underline{Kn}(t)$ die Menge der Knoten von t. Sie bestimmt die Menge $\underline{Var}_t = \{\alpha.k \mid k \in \underline{Kn}(t)$ markiert durch $X \in \mathcal{X}, \alpha \in \underline{att}(X)\}$ der aktuellen Attributvariablen von t.

3-4

Wird an einem inneren Knoten (kein Blattknoten) $k_0 \in \underline{Kn}(t)$ die Regel $\pi = X_0 \to X_1 \ldots X_i$ angewandt und sind $k_1 \ldots k_i \in \underline{Kn}(t)$ die entsprechenden Nachfolgerknoten, so erhält man das **Attributgleichungssystem** E_{k_0} von k_0 aus E_π durch die Indexsubstitution $(i \mapsto k_i \mid 0 \leq i \leq r)$ bei den Attributvariablen (formale Parameter $\rightsquigarrow$ aktuelle Parameter).
Dann ist $E_t := \bigcup\{E_k \mid k \text{ innerer Knoten von } t\}$.

3-4

Beachte dabei, daß es zu jeder aktuellen Attributvariablen $a.k$, ausgenommen die inheriten der Wurzel und die synthetischen der Blätter genau eine Gleichung $a.k = \ldots$ gibt.

Annahme:

- Keine inheriten Attribute des Startsymbols (ausgenommen bei inkrementeller Übersetzung)
- Synthetische Attribute der Terminale kommen vom Scanner

3.0.2 Lösbarkeit von E_t

E_t kann keine, genau eine oder mehrere Lösungen haben. Betrachte als **Beispiel** die Regeln $A \to uBv$ und $B \to w$. Ferner sei $\alpha \in \underline{syn}(B)$ und $\beta \in \underline{inh}(B)$.
Dann existieren u.a. die folgenden Attributgleichungen: $\beta.i = f(\alpha.i)$ und $\alpha.0 = g(\beta.0)$.
Man erkennt, daß E_t eine **zirkuläre Abhängigkeit** enhält:

$$\left.\begin{array}{l}\beta.k = f(\alpha.k)\\ \alpha.k = g(\beta.k)\end{array}\right\} \to^{g=\underline{id}_{\mathbb{N}}} \beta.k = f(\beta.k)$$

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

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

Folgerung 3.3 *Zirkularitäten sind unerwünscht. Sie entstehen erst in* $E_t = \bigcup_{k \in \underline{IKn}(t)} E_k$*, nicht bereits in* E_π*.*

Definition 3.4 $\mathfrak{A} \in AG$ *heißt* ***zirkulär****: es gibt einen Ableitungsbaum* t *mit zirkulärem Attributgleichungssystem* E_t*, d.h. eine aktuelle Attributvariable hängt von sich selbst ab.*

Satz 3.5 *Es ist entscheidbar, ob* $\mathfrak{A} \in AG$ *zirkulär ist. Das geschieht mit einer Zeitkomplexität von* $O(2^n)$ *mit* $n = |\mathfrak{A}|$*.*

3.0.3 Hilfsmittel für Zirkularität und Attributberechnung

Definition 3.6 *Sei* $\mathfrak{A} \in AG$ *und* t *ein Ableitungsbaum von* $\mathfrak{A}$*.* E_t *bestimmt den* ***Ableitungsgraphen*** DG_t*:*

- *Knotenmenge* $\underline{Kn}(DG_t) := \underline{Var}_t$
- *Kantenmenge* $(x_1, x_2) \in \underline{Kan}(DG_t) \quad :\curvearrowright \quad x_2 = f(\ldots, x_1, \ldots) \in E_t$

Also ist die Kante von x_1 *nach* x_2 *in* DG_t*, wenn* x_2 ***direkt*** *von* x_1 *abhängt.*

3-3 **3-4** **Puzzle**

Attributberechnung

1. E_t als Termersetzungssystem, Top-Down-Berechnung
   Keine Zirkularität $\curvearrowright$ Termination mit eindeutiger Lösung.
2. Bottom-Up-Auflösung von E_t: Variablen durch Werte ersetzen
3. Uniforme Berechnung, unabhängig vom Ableitungsbaum:

(a) Berechnungspläne für $(E_\pi \mid \pi \in P)$ aufstellen[1]

(b) Rekursive Prozeduren/Funktionen für $(E_\pi \mid \pi \in P)^2$: die Idee ist, jeder synthetischen Attributvariablen eine Prozedur/Funktion zuzuordnen mit dem inheriten Attribut als Parameter.

Diese Verfahren haben eine große Bedeutung für die automatische Compilererzeugung.

4. Spezialfälle: SAG, LAG mit Attributberechnung während der Syntaxanalyse.

3.0.4 S-Attributgrammatiken

Definition 3.7 *$\mathfrak{A} \in AG$ heißt S-Attributgrammatik ($\mathfrak{A} \in SAG$), wenn $\underline{Inh} = \emptyset$, also $\underline{Att} = \underline{Syn}$.*

Folgerung 3.8 *Bottom-Up-Berechnung der Attributwerte, Spezialfall: Bestimmung der Wurzelattribute*

Die Idee ist, eine Bottom-Up-Syntaxanalyse durchzuführen mit einer Zwischenspeicherung von Attributwerten im Analysekeller.

I_1	v3
I_7	v2
I_8	v1
I_4	
$\vdots$	$\vdots$

Beim reduce-Schritt werden simultan die Werte der synthetischen Attribute von A aus den Werten unter $v1, v2, v3$ berechnet (Records). Beim accept-Schritt werden die Werte der Wurzelattribute ausgegeben.

Beispiel 1: Stackcode für arithmetische Ausdrücke

Gegeben ist die folgende Grammatik:

$$
\begin{array}{rl|l}
E \to & (E+E) & c.0 = c.2; c.4; ADD \\
& \mid (E*E) & c.0 = c.2; c.4; MULT \\
& \mid id & c.0 = LOAD\ a.1 \\
& \mid num & c.0 = LIT\ z.1
\end{array}
$$

Das Attribut c bezeichnet Stackcode, a bezeichnet eine Adresse und z eine Zahl. Die Werte von $a.1$ sind die lexikalischen Attribute, die mit dem Token vom Scanner übergeben werden: $(id, X)(num, 7)$

[1] Hier sei auf Kennedy/Warren verwiesen.
[2] Hier sei weiter auf Jourdan/Gallie verwiesen.

$$((3 + X) * (Y + 5))$$

$$\Big\downarrow {\scriptstyle Scanner}$$

$$((< num, 3 > + < id, X >) * (< id, Y > + < num, 5 >))$$

Codegenerierung durch Attributsauswertung während der Syntaxanalyse:

1. shift: Aufruf des Scanners durch "nextsym"
   (a) Token $\mapsto$ LR(0)-Menge mit goto
   (b) lexikalisches Attribut ebenfalls auf Analysekeller
2. reduce:
   (a) Reduktion auf dem Analysekeller
   (b) simultane Attributberechnung
3. accept: Wurzelattribut ausgeben

$$\begin{array}{c|l} \texttt{num} & \texttt{3} \\ \texttt{(} & \\ \texttt{(} & \end{array} \rightsquigarrow \begin{array}{c|c} \texttt{id} & \texttt{X} \\ \texttt{+} & \\ \texttt{E} & \texttt{LIT 3} \\ \texttt{(} & \\ \texttt{(} & \end{array} \rightsquigarrow \begin{array}{c|c} \texttt{)} & \\ \texttt{E} & \texttt{LOAD X} \\ \texttt{+} & \\ \texttt{E} & \texttt{LIT 3} \\ \texttt{(} & \\ \texttt{(} & \end{array} \rightsquigarrow \begin{array}{c|l} \texttt{E} & \texttt{LIT 3; LOAD X; ADD;} \\ \texttt{(} & \end{array}$$

$\rightsquigarrow$ Ergebnis: `LIT 3; LOAD X; ADD; LOAD Y; LIT 5; ADD; MULT;`

Beispiel 2: Berechnung des abstrakten Syntaxbaumes

Dazu muß zunächst der Begriff der **konkreten und abstrakten Syntax** erklärt werden. $\pi = A_0 \to w_0 A_1 w_1 \ldots A_r w_r$ repräsentiert ein Operationssymbol F_π vom Typ

$$A_1 \times \ldots \times A_r \to A_0$$

(A_i als Sorte, Typ).

Bemerkung 3.9 *"Verkleben" der Regeln zu einem Ableitungsbaum entspricht der funktionalen Applikation der zugehörigen Operationssymbole.*

Folgerung 3.10 *Der Ableitungsbaum vereinfacht sich zum* ***abstrakten Syntaxbaum*** *(AST). Nur dieser ist für die Übersetzung erforderlich.*

- **konkrete Sytax:** benutzerfreundliche "Mix-fix"-Notation (syntactic sugar) unter Verwendung natürlicher Sprache für Terminalsymbole in CFG.
- **abstrakte Syntax:** darstellungsunabhängige algebraische Struktur.

3-6

Das Beispiel auf der Folie zeigt die Grammatik $\mathcal{G}_S$. Wir betrachten in dem Beispiel folgendes kleines Beispielprogramm:

$$\underline{\texttt{begin}}\ \texttt{id := num + id;}\ \underline{\texttt{if}}\ \texttt{id < num}\ \underline{\texttt{then}}\ \texttt{id := num + id}\ \underline{\texttt{end}}$$

Die Aufgabe ist es, den AST während der Bottom-Up-Analyse zu berechnen. Dabei verwenden wir die Methode der **S-Attributgrammatiken**. Die Darstellung des Aufbaus von AST erfolgt durch die Konstruktion von Graphen.

a) **Graphen**
Sei $a \in Adr$ unendliche Menge von Adressen für Speicherplätze (Heap) und Ω Operationsalphabet mit Stelligkeit.

$f^{(i)} \in \Omega = \{\underline{assign}^{(1)}, \underline{seq}^{(2)}, \underline{cond}^{(2)}, \underline{less}^{(2)}, \underline{plus}^{(2)}, \underline{ident}^{(0)}, \underline{number}^{(0)}\}$

$\underline{Kno}$ sei die Menge der Ω-Knoten.

$k \in \underline{Kno} := \{(a, f, a_1, \ldots, a_n) \mid a, a_i Adr, f \in \Omega^{(n)}\}$

$$\begin{array}{c|c|c|c|c|} \cline{2-5} \xrightarrow{a} & f & \bullet & \cdots & \bullet \\ \cline{2-5} \multicolumn{1}{c}{} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{a_1 \downarrow} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{a_n \downarrow} \end{array}$$

Der Graph ist dann die Menge von Knoten mit einer Wurzel.

$$G \in \underline{Graph} := \{(a, K) \mid a \in Adr, K \subseteq \underline{Kno}\}$$

3-7

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

S-Attributierung von $\mathcal{G}_S$:
Wurzeladressen als S-Attribute im Analysekeller mitführen, Heap anstelle eines Ausgabebandes.

3-7a

Beispiel 3: Typberechnung für arithmetische Ausdrücke

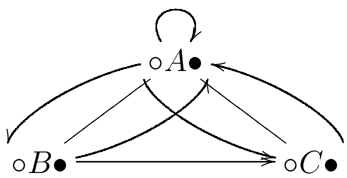
$\underline{Typ} := \{int, real\}$ mit $\max : \underline{Typ}^2 \longrightarrow \underline{Typ}$ bezüglich $int < real$.

3-7a

3.0.5 L-Attributgrammatiken

Definition 3.11 $\mathfrak{A} = \langle \mathcal{G}, (E_\pi | \pi \in P)\rangle \in AG$ *ist eine L-Attributgrammatik* ($\mathfrak{A} \in LAG$) $:\Leftrightarrow$ *für jede Attributgleichung* $\alpha.i = f(\ldots \beta.j \ldots)$ *gilt:* $\alpha \in \underline{Inh} : \beta \in \underline{Syn} \Rightarrow j < i$.

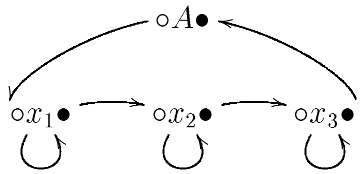
Beispiel: (DG_π)



Folgerung 3.12 *Eine* $\mathcal{G} \in LAG$ *ist nicht zirkulär. Attributberechnung in „depth-first, left-to-right-order“, Baumreise mit zwei Knotenbesuchen.*

1. *top-down (inherite Attribute)*
2. *bottom-up (synthetische Attribute)*

Auswertungsreihenfolge:



Syntaxanalyse mit L-Attributauswertung

Sei $\mathfrak{A} = \langle \mathcal{G}, (E_\pi | \pi \in P)\rangle \in LAG$ mit $\mathcal{G} \in LL(1)$. **Ziel** ist die Erweiterung der Top-Down-Analyse zur Berechnung eines synthetischen Wurzelattributs. Die **Methode** ist die Expansion von $A \in N$ auf den Analysekeller derart, daß eine spätere Reduktion möglich ist.

Kelleralphabet:
$\bigcup_{\pi \in P} (\underline{LR(0)}_\pi(\mathcal{G}) \times \underline{Val}_\pi)$ mit $\underline{LR(0)}_\pi(\mathcal{G}) := \{[A \to \alpha \cdot \beta] \mid A \to \alpha\beta = \pi\}$ und $\underline{Val}_\pi = \{\underline{val}_\pi \mid \underline{val}_\pi : \underline{Var}_\pi \dashrightarrow \text{"Wertmenge"}\}$

Bei der **Arbeitsweise** betrachten wir drei Fälle:

1. "Expand"-Schritt mit Berechnung inheriter Attribute

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

$\Rightarrow$

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

Dabei ist $B \to \beta$ wg. $\mathcal{G} \in LL(1)$ durch den Eingabe-look-ahead bestimmt.

$\underline{val}$ belegt die inheriten Attributvariablen von B nach Attributgleichungen für $A \to X_1 \ldots X_{i-1}B\gamma$. Sei $\alpha \in \underline{inh}(B)$ und $\alpha.i = t \in E_{A \to X_1 \ldots \gamma}$, so $\underline{val}'(\alpha.0) = \widehat{\underline{val}}(t)$.

$\widehat{\underline{val}}(t)$ ist dabei die homomorphe Fortsetzung von $\underline{val}$ auf Terme. (Kurz: Einsetzen und Ausrechnen)

2. "Match"-Schritt

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

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

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

$\underline{val}''$ erweitert $\underline{val}$ um die synthetischen Attribute von B. Dabei ist $\underline{val}''(\alpha.i) = \widehat{\underline{val}'}(t)$, falls $\alpha.0 = t \in E_{B \to \beta}$, ansonsten gilt $\underline{val}''(\alpha.i) = \underline{val}(\alpha.i)$.

3-8 **3-9**

3.0.6 Anwendung von LAG

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

$$\begin{array}{rcll} P & \to & DL; SL & \text{Programm} \\ DL & \to & V|V; DL & \text{Deklarationsliste} \\ SL & \to & S|S; SL & \text{Statementliste} \\ V & \to & a|b|\dots & \text{Variable} \\ S & \to & V := V & \text{Statement} \end{array}$$

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

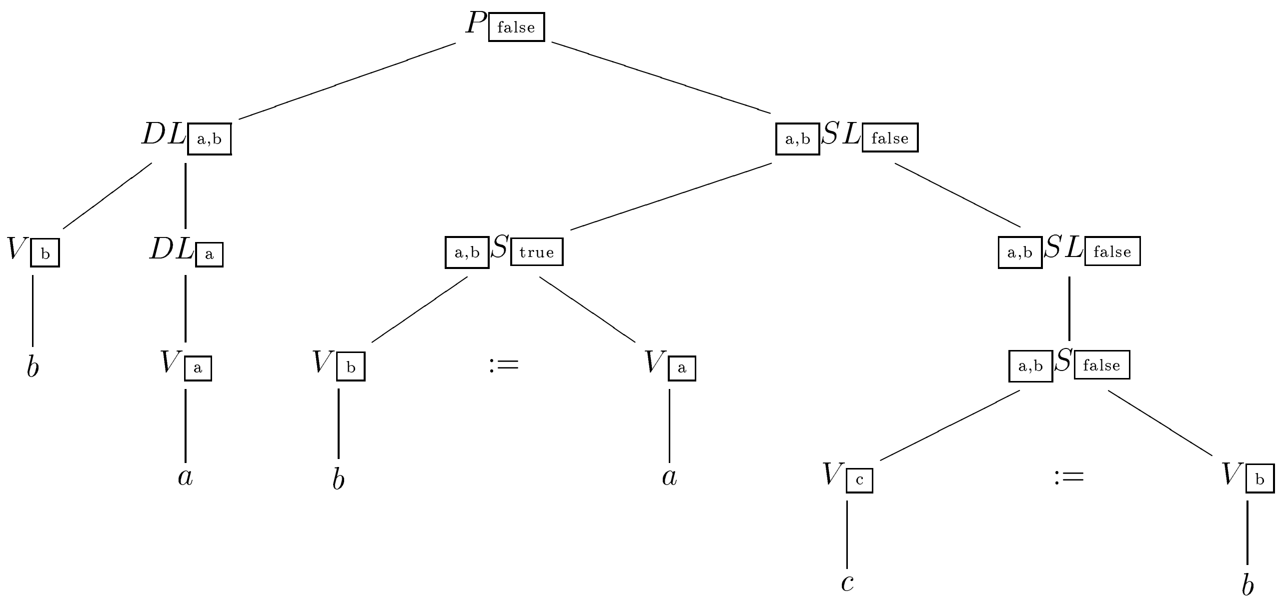
Attribute:

syn	v	deklarierte/benutzte Variable für V mit Werten aus $\{a, b, \dots\}$
syn	dv	deklarierte Variablen für DL, $\subseteq \{a, b, \dots\}$
inh	env	Umgebung für S, SL, $\subseteq \{a, b, \dots\}$
syn	decl	deklariert? für S, SL, $P \in \{\text{true,false}\}$

Attributgleichungen:

$$\begin{array}{ll}
P \to DL; SL & decl.0 = decl.3 \\
 & env.3 = dv.1 \\
DL \to V & dv.0 = \{v.1\} \\
DL \to V; DL & dv.0 = \{v.1\} \cup dv.3 \\
V \to a|\ldots & v.0 = a\ldots \\
SL \to S & env.1 = env.0 \\
 & decl.0 = decl.1 \\
SL \to S; SL & decl.0 = decl.1 \text{ AND } decl.3 \\
 & env.1 = env.0 \\
 & env.3 = env.0 \\
S \to V := V & decl.0 = v.1 \in env.0 \text{ AND } v.3 \in env.0
\end{array}$$

Attributierter Syntaxbaum:



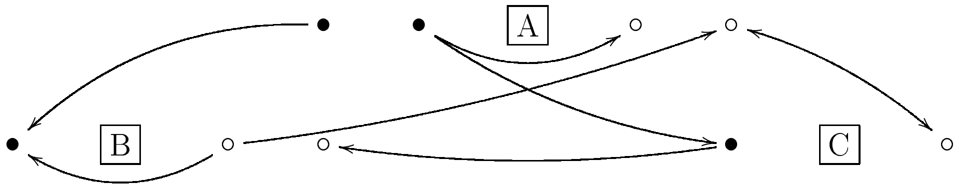
3.0.7 Zirkularitätstest für Attributgrammatiken

Sei $\mathfrak{A} = \langle \mathcal{G}, (E_\pi | \pi \in P) \rangle \in AG$. Eine Regel $\pi = X_0 \to X_1 \ldots X_r$ bestimmt ihren **Abhängigkeitsgraphen** DG_π mit der Knotenmenge $\underline{Kno}(DG_\pi) := \underline{OKno}(DG_\pi) \cup \underline{UKno}(DG_\pi)$. Die **Ober-** und die **Unterknoten** sind im einzelnen wie folgt definiert:

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

$$(X_1, X_2) \in \underline{Kan}(DG_\pi) :\curvearrowright X_2 = f(\ldots X_1 \ldots) \in E_\pi$$

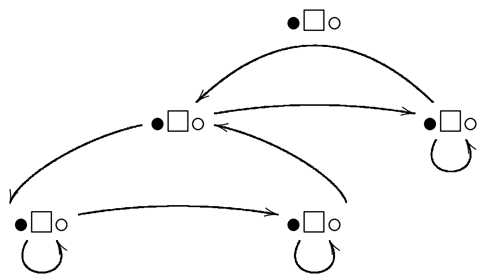
Dazu ein Beispiel: DG_π mit $\pi = A \to BC$.



Beachte dabei:

$$\underline{Var}_\pi = \underline{UKno}(DG_\pi) \dot{\cup} \underline{OKno}(DG_\pi) = \underline{OVar}_\pi \dot{\cup} \underline{IVar}_\pi$$

Insbesondere gilt: $\underline{Kan}(DG_\pi) \subseteq \underline{OVar}_\pi \times \underline{IVar}_\pi$, also keine Zirkularität in DG_π.
Das Problem dabei ist, daß beim Verkleben der DG_π zu DG_t für einen Ableitungsbaum t Schleifen auftreten können.
Dabei stellt man folgende Beobachtung an: Eine Schleife in DG_π bestimmt einen Teilgraphen der Form



also eine Regel π mit verbindbaren Unterknoten DG_π. Eine Verbindung gehört immer zu einem Nichtterminalsymbol und verläuft von einer inheriten zu einer synthetischen Attributvariable.

Definition 3.13 (Attributabhängigkeit) *Sei $A \in N$, $\alpha \in \underline{inh}(A)$,$\alpha' \in \underline{syn}(A)$. Dann heißt α'* ***von α unterhalb A abhängig*** *(Bezeichnung: $\alpha \overset{A}{\curvearrowright} \alpha'$), wenn ein Ableitungsbaum t mit Wurzel A und Wurzelknoten k existiert, so daß in DG_t ein Pfad von $\alpha.k$ nach $\alpha'.k$ führt:*

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

Man beachte dabei, daß Paare abhängiger Attribute von verschiedenen Ableitungsbäumen mit gleicher Wurzel getrennt werden müssen, da man sonst unzulässige Abhängigkeitskombinationen erhält (Fehler: Knuth !).

Puzzle

Definition 3.14 (Attributabhängigkeitsmenge) *Sei t ein Ableitungsbaum mit Wurzel A. Es sei:*

$$D(A,t) := \{(\alpha, \alpha') \mid \alpha \overset{A}{\curvearrowright} \alpha' \textit{ in } t\}$$

Außerdem sei:

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

Bei der induktiven Bestimmung benutzen wir folgende Bezeichnung:

Definition 3.15 *Für* $\pi = A_0 \to w_0 A_1 w_1 \dots A_r w_r$ *und* $D_i \subseteq inh(A_i) \times \underline{syn}(A_i)$ *definieren wir* $D[\pi; D_1, \dots, D_r] \subseteq \underline{inh}(A_i) \times \underline{syn}(A_i)$ *durch:*

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

Lemma 3.16 *Die Attributabhängigkeitssysteme* $\mathcal{D}(A)$ *mit* $A \in N$ *sind induktiv bestimmt durch die folgenden Punkte:*

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

Beweis 3.17 *Induktion über die Struktur der Ableitungsbäume*

Beachte, daß für $D \in \mathcal{D}(A)$ gilt: $D \subseteq \underline{inh}(A) \times \underline{syn}(A)$. Daher bricht der Berechnungsprozeß nach endlich vielen Schritten ab.

Folgerung 3.18 (Zirkularitätstest) $\mathfrak{A} \in AG$ *ist zirkulär, genau dann wenn es* $\pi = A_0 \to w_0 A_1 w_1 \dots A_r w_r$, $\alpha.k \in \underline{UKno}(DG_\pi)$ *und* $D_i \in \mathcal{D}(A_i)$ $(1 \leq i \leq r)$ *gibt, so daß* $(\alpha.k, \alpha.k) \in \underline{trans}(\underline{Kan}(DG_\pi) \cup \bigcup_{i=1}^{r} \{(\beta.i, \beta'.i) \mid (\beta, \beta') \in D_i\})$.

Die Komplexität ist $n = |\mathfrak{A}|$, $T(n)$ Zeit zur Entscheidung der Zirkularität: $2^{\frac{c \cdot n}{log(n)}} \leq T(n) \leq 2^{d \cdot n^2}$

Stark-nichtzirkuläre Grammatiken

Definition 3.19 $\mathfrak{A} \in AG$ *heißt stark-nichtzirkulär, wenn der Zirkularitättest mit* $D_S(A)$ *anstelle von Mengen* $D(A) \in \mathcal{D}$ *keine Zirkularität ergibt.*

Das Resultat ist ein Test in Polynomzeit. $D_S(A) := \bigcup\{D(A) \mid D(A) \in \mathcal{D}(A)\}$.

3.0.8 Funktionale Auswertung von Attributgrammatiken

Ziel ist die Berechnung der synthetischen Wurzelattribute eines Ableitungsbaumes.

Sei $\mathfrak{A} = \langle \mathcal{G}, (E_\pi \mid \pi \in P) \rangle \in AG$ mit $\underline{inh}(S) = \underline{inh}(a) = \underline{syn}(a) = \emptyset$ für $a \in \Sigma$.

Für $A \in N$ sei $T_A := \{t \mid t \text{ Ableitungsbaum mit Wurzel } A\}$. $t \in T_A$ bestimmt $\pi = A \to w_0 A_1 w_1 \dots A_r w_r$ und $t_i \in T_{A_i}$ $(1 \leq i \leq r)$.

Bezeichnung: $t = [\pi; t_1, \dots, t_r]$ mit dem Spezialfall $r = 0$, falls $\pi = A \to w$.

Die Beobachtung dabei: Der Wert eines $\alpha \in \underline{syn}(A)$ ist bestimmt durch $t \in T_A$ und die Werte von $\underline{inh}(A) = \{\beta_1, \dots, \beta_r\}$:

$$f_A^\alpha : T_A \times V^{\beta_1} \times \dots \times V^{\beta_r} \to V^\alpha$$

Sei $\mathfrak{A} = \langle \mathcal{G}, (E_\pi \mid \pi \in P) \rangle \in AG$, $A \in N, \alpha \in \underline{syn}(A)$, $f_A^\alpha : T_A \times V^{\beta_1} \times \ldots \times V^{\beta_r} \to V^\alpha$.

Frage: Wie bestimmt $(E_\pi \mid \pi \in P)$ die Funktionen f_A^α?

Die Definition dieser Attributfunktion erfolgt durch Rekursion über den Ableitungsbaum und Fallunterscheidung nach der Wurzelproduktion mit Hilfe der E_π.

Der Einfachheit halber sei für alle $A \in N$: $\underline{syn}(A) = \{\alpha\}$ und $\underline{inh}(A) = \{\beta\}$, außerdem gelte $k \leq 2$ (gewährleistet z.B. durch Chomsky-Normalform).

$$E_{A \to A_1 A_2} : \begin{array}{lcl} \alpha.0 & = & f(\alpha.1, \alpha.2, \beta.0) \\ \beta.1 & = & g(\alpha.1, \alpha.2, \beta.0) \\ \beta.2 & = & h(\alpha.1, \alpha.2, \beta.0) \end{array}$$

Für die Attributfunktionen folgt daraus:

$$\begin{array}{lcl} f_A^\alpha(t, v) & = & \underline{\texttt{if}}\ t = [A \to A_1 A_2; t_1, t_2]\ \underline{\texttt{then}} \\ & & \quad f(f_{A_1}^{\alpha_1}(t_1, x_1), f_{A_2}^{\alpha_2}(t_2, x_2), v) \\ & & \underline{\texttt{whererec}} \\ & & \quad x_1 = g((f_{A_1}^{\alpha_1}(t_1, x_1), f_{A_2}^{\alpha_2}(t_2, x_2), v) \\ & & \quad x_2 = h((f_{A_1}^{\alpha_1}(t_1, x_1), f_{A_2}^{\alpha_2}(t_2, x_2), v) \\ & & \underline{\texttt{if}}\ t = [A \to \ldots] \\ & & \vdots \\ & & \underline{\texttt{if}}\ t = [A \to \ldots] \end{array}$$

Wichtig: Die Auswertung muß mit nicht-strikter Semantik erfolgen, also mit verzögerter Auswertung der Funktionsaufrufe nach dem "leftmost-outermost"-Prinzip. Liegt eine Zirkularität vor, so kommt es bei der Berechnung der inheriten Werte x_1 und x_2 zu einer Rekursion mit nicht-terminierender Berechnung.

Spezialfall: $\mathfrak{A} \in LAG$

$$E_{A \to A_1 A_2} : \begin{array}{lcl} \alpha.0 & = & f(\alpha.1, \alpha.2, \beta.0) \\ \beta.1 & = & g(\beta.0) \\ \beta.2 & = & h(\alpha.1, \beta.0) \end{array}$$

Die Attributfunktionen aus dem obigen Beispiel vereinfachen sich dann dahingehend, daß der <u>`where`</u>-Teil so aussieht:

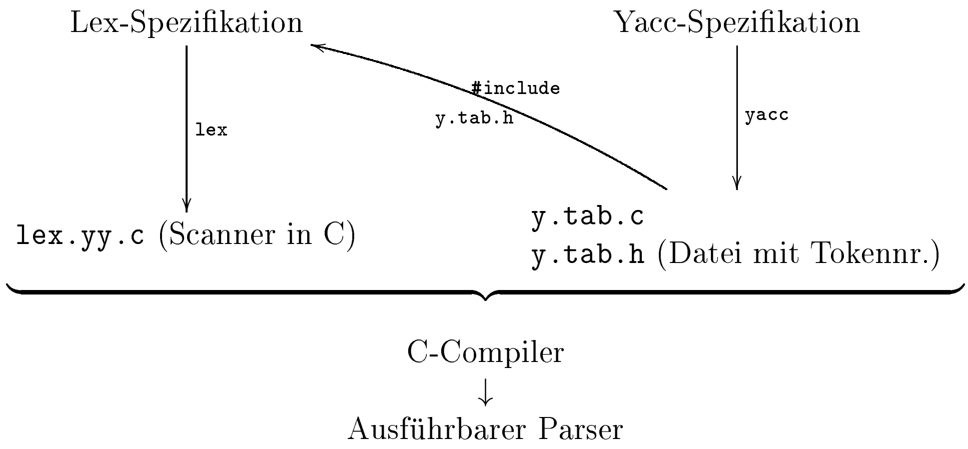
$$\begin{array}{l} \underline{\texttt{where}} \\ \quad \begin{array}{lcl} x_1 & = & g(v) \\ x_2 & = & h((f_{A_1}^{\alpha_1}(t_1, x_1), v) \end{array} \end{array}$$

Nach dem Einsetzen ergibt sich dann:

$$\begin{array}{lcl} f_A^\alpha(t, v) & = & \underline{\texttt{if}}\ t = [A \to A_1 A_2; t_1, t_2]\ \underline{\texttt{then}} \\ & & f(f_{A_1}^{\alpha_1}(t_1, g(v)), f_{A_2}^{\alpha_2}(t_2, h((f_{A_1}^{\alpha_1}(t_1, g(v)), v)), v) \end{array}$$

3.1 Automatische Parsergenerierung mit Yacc

Dieser Abschnitt schließt die Attributauswertung mit ein. `yacc` ist eine Abkürzung und bedeutet: Yet another compiler compiler.
Das **System** schaut schematisch etwa so aus:



3.1.1 Aufbau einer Yacc-Spezifikation

Yacc-Spezifikationen haben eine ähnliche Struktur wie Lex-Spezifikationen, sie sind in drei Teile gegliedert, jeweils durch `%%` getrennt.

1. **Definitionen**

   - von Token, z.B. `token NAME NUMBER`
     Beachte: Nichtdeklarierte Token sind möglich, indem man sie bei der Benutzung in Hochkommata einschließt: '+','=',...
   - C-Code für Definitionen (Datenstrukturen, Includes), wie folgt:
     `%{code ... %}`

2. **Regeln**

   - CFG und Attributierung (Anweisungen, diese müssen nicht tatsächlich zur Attributberechnung eingesetzt werden, sondern können beliebige C-Funktionen sein).
     Beispiel:

     ```
     stmt : NAME '=' expr
          | expr                     {printf("=%d\n",$1)}

     expr : expr '+' NUMBER          {$$ = $1 + $3}
          | NUMBER                   {$$ = $1}
     ```

     Beim letzten Fall ist `$1` ein lexikalisches Attribut, das durch den Aufruf des Scanners mit `nextsym()` gewonnen wird.

3. **C-Funktionen**
   zur direkten Übernahme in `y.tab.c` und zur Verwendung in Attributfunktionen.

Zu obigen Beispiel würde folgende Lex-Spezifikation passen:

```
%{
#include "y.tab.h"
extern int yylval; /* lexikalischer Ausdruck fuer nextsym */
%}
%%
[0-9]+     {yylval = atoi(yytext); return NUMBER;}
[a-zA-Z]+  {return NAME;}
[ \t]      /* ignore whitespace */
\n         {return 0;} /* \n hier als EOF */
.          {return yytext[0];} /* ASCII-Wert als Token zurueck */
```

Weitere Möglichkeiten:

- mehrere synthetische Attribute können durch C-Strukturen realisiert werden
- inherite Attribute lassen sich durch C-Funktionen berechnen, Zugriff auf vorher berechnete Werte ist mit `$0,$-1,$-2,`... möglich.
- Zur Behandlung von Mehrdeutigkeiten kann man Präzedenzen definieren (Punkt- vor Strichrechnung, dangling else).

Kapitel 4

Übersetzung in Zwischencode

Die Codeerzeugung kann man schematisch wie folgt aufteilen: PS $\overset{trans}{\Longrightarrow}$ Z $\overset{code}{\longrightarrow}$ MC.

- front-end: <u>trans</u> erzeugt maschinenunabhängigen Zwischencode für eine abstrakte Stackmaschine.
- back-end: <u>code</u> erzeugt Maschinencode.

Der Vorteil dieser Zerlegung: Z ist maschinenunabhängig, woraus Portabilität, Transparenz und Codeoptimierung folgt.

Beispiel: P-Code von Pascal, JAVA Virtual Maschine (JVM)

$\boxed{\textbf{4-1}}$ $\boxed{\textbf{4-2}}$ $\boxed{\textbf{4-3}}$ $\boxed{\textbf{4-4}}$

4.1 Übersetzung von Ausdrücken

In diesem Abschnitt soll auf die Übersetzung von Ausdrücken, Anweisungen, Blöcken und Prozeduren eingegangen werden.

Zur Verdeutlichung dient eine Beispielprogrammiersprache BPS (Pascal ohne Datenstrukturen und ohne Prozedurparameter).

- arithmetische und bool'sche Ausdrücke mit strikter und nicht-strikter Semantik
- Kontrollstrukturen: Sequenz, Verzweigung, Iteration
- Blöcke und Prozeduren: lokale und globale Objekte, variable Umgebung, dynamische Speicherverwaltung mit Laufzeitkeller

In den unterschiedlichen Programmiersprachen gibt es unterschiedliche Prozedurkonzepte:

- FORTRAN:
  Unterprogramme, nicht geschachtelt, keine Rekursion $\rightsquigarrow$ statische Speicherverwaltung, Speicherbedarf zur Übersetzungszeit bekannt

- C:
  rekursive Prozeduren, nicht geschachtelt $\leadsto$ dynamische Speicherverwaltung, der Speicherbedarf erst zur Laufzeit bekannt, keine statischen Verweise auf Laufzeitkeller

- ALGOL-Familie (Pascal, ALGOL60, Modula, Oberon):
  geschachtelte, rekursive Prozedurdeklarationen $\leadsto$ dynamische Speicherverwaltung mit statischen Verweisen

In unserem Beispiel soll nun gelten:

- keine Datenstrukturen, keine Prozedurparametzer

- Datentypenbeschränkung auf Integer

Syntax von BPS

Im folgenden gelte stets: $n \geq 1$.

$\underline{\texttt{Int}}$:	$\texttt{Z}$		`% Z ist ein Bezeichner`
$\underline{\texttt{Ide}}$:	$\texttt{I}$		`% I ist ein Bezeichner`
$\underline{\texttt{AExp}}$:	$\texttt{E}$	`::=`	$\texttt{Z} \mid \texttt{I} \mid (\texttt{E}_1 + \texttt{E}_2) \mid \ldots$
$\underline{\texttt{BExp}}$:	$\texttt{BE}$	`::=`	$(\texttt{E}_1 < \texttt{E}_2) \mid \underline{\texttt{not}}\ \texttt{BE} \mid (\texttt{BE}_1\ \underline{\texttt{and}}\ \texttt{BE}_2) \mid (\texttt{BE}_1\ \underline{\texttt{or}}\ \texttt{BE}_2)$
$\underline{\texttt{Cmd}}$:	Γ	`::=`	`I:=E | I() |`
			$\underline{\texttt{begin}}\ \Gamma_1; \ldots; \Gamma_\texttt{n}\ \underline{\texttt{end}} \mid$
			$\underline{\texttt{if}}\ \texttt{BE}\ \underline{\texttt{then}}\ \Gamma_1\ \underline{\texttt{else}}\ \Gamma_2 \mid$
			$\underline{\texttt{while}}\ \texttt{BE}\ \underline{\texttt{do}}\ \Gamma$
$\underline{\texttt{Decl}}$:	Δ	`::=`	$\Delta_c \Delta_v \Delta_p$
	Δ_c	`::=`	$\varepsilon \mid \underline{\texttt{const}}\ \texttt{I}_1 = \texttt{Z}_1, \ldots, \texttt{I}_\texttt{n} = \texttt{Z}_\texttt{n};$
	Δ_v	`::=`	$\varepsilon \mid \underline{\texttt{var}}\ \texttt{I}_1, \ldots, \texttt{I}_\texttt{n};$
	Δ_p	`::=`	$\varepsilon \mid \underline{\texttt{proc}}\ \texttt{I}_1, \texttt{B}_1; \ldots; \underline{\texttt{proc}}\ \texttt{I}_\texttt{n}, \texttt{B}_\texttt{n};$
$\underline{\texttt{Block}}$:	$\texttt{B}$	`::=`	$\Delta\Gamma$
$\underline{\texttt{Prog}}$:	$\texttt{P}$	`::=`	$\underline{\texttt{in/out}}\ I_1, \ldots, I_n; B.$

Semantik von BPS

- Bezeichner einer Deklaration müssen paarweise verschieden sein

- ein im Anweisungsteil Γ eines Blocks $\Delta\Gamma$ auftretender Bezeichner muß deklariert sein, und zwar in Δ oder in der Deklarationsliste eines $\Delta\Gamma$ umschließenden Blocks

- Mehrfachdeklarationen eines Bezeichners auf verschiedenen Niveaus sind möglich: die "innere" Deklaration ist für ein Auftreten gültig

- **static scopes:** Beim Aufruf einer Prozedur ist die Deklarationsumgebung und nicht die Aufrufumgebung gültig.

$\boxed{\textbf{4-6}}$ $\boxed{\textbf{4-7}}$ $\boxed{\textbf{4-8}}$ $\boxed{\textbf{4-9}}$ $\boxed{\textbf{4-10}}$ $\boxed{\textbf{4-11}}$

Wichtig sind 2 Grundbegriffe:

1. Umgebung
2. Zustandsraum

$\boxed{\textbf{4-7}}$ $\boxed{\textbf{4-8}}$

Für die `while`-Schleife sähe das dann beispielsweise so aus:

$$F(x) = \underline{if}\ p(x)\ \underline{then}\ F(g(x))\ \underline{else}\ x$$

4.1.1 Zwischencode für BPS

Zur Erzeugung des Zwischencode benutzen wir exemplarisch eine abstarkte Maschine (AM) mit Daten- und Prozedurkeller.

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

AM-Befehlssatz

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

AM-Befehlssemantik

Die Befehlssemantik ist allgemein für jeden AM-Befehl wie folgt definiert:

$$[\![B]\!] : ZR \dashrightarrow ZR$$

Das ergibt dann im einzelnen:

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

- $[\![\texttt{LT}]\!](m, d : z_1 : z_2, p) := (m + 1, d : b, p)$ mit $b = \begin{cases} 1 & \text{falls } z_1 < z_2 \\ 0 & \text{sonst} \end{cases}$
- $[\![\texttt{AND}]\!](m, d : b_1 : b_2, p) := (m + 1, d : b_1 \wedge b_2, p)$ mit $b_1, b_2 \in \{0, 1\}$
- $[\![\texttt{OR}]\!](m, d : b_1 : b_2, p) := (m + 1, d : b_1 \vee b_2, p)$ mit $b_1, b_2 \in \{0, 1\}$
- $[\![\texttt{NOT}]\!](m, d : b, p) := (m + 1, d : \neg b, p)$ mit $b \in \{0, 1\}$
- $[\![\texttt{JFALSE n}]\!](m, d : b, p) := \begin{cases} (n, d, p) & \text{falls } b = 0 \\ (m + 1, d, p) & \text{falls } b = 1 \end{cases}$
- $[\![\texttt{JMP n}]\!](m, d, p) := (n, d, p)$

Um die Prozedur- und Transportbefehlssemantik näher zu beschreiben, müssen wir erst ein paar Hilfsfunktionen erklären.

Zunächst betrachten wir die **Struktur des Prozedurkellers p**: p zerfällt in Aktivierungsblöcke (Frames) der Form:

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

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

Berechnung des statischen Verweises:

$\boxed{\textbf{4-12}}$

sv liefert die Differenz zwischen Aufruf- und Deklarationsniveau und damit die Länge der Verweiskette. Als **Hilfsfunktion** dient die base-Funktion.

$$\underline{base} : PK \times \mathbb{N} \dashrightarrow \mathbb{N}$$

Diese Hilfsfunktion bestimmt für einen Prozedurkeller bezüglich einer Niveaudifferenz den Beginn der Deklarationsumgebung (als **absolute** Adresse des aktuellen Prozedurkellers). Die Funktion ist rekursiv definiert:

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

Im Beispiel von $\boxed{\textbf{4-12}}$ ergibt sich damit nach dem zweiten Aufruf von A:

$$\begin{array}{rcl}
\underline{base}(p,0) & = & 1 \\
\underline{base}(p,1) & = & \underline{base}(p,0) + p.\underline{base}(p,0) \\
 & & 1 + p.1 \\
 & & 1 + 5 = 6 \\
\underline{base}(p,2) & = & \underline{base}(p,1) + p.\underline{base}(p,1) \\
 & & 6 + p.6 \\
 & & 6 + 5 = 11
\end{array}$$

Es folgt:

$$sv = \underline{base}(p,2) + 2 + 2$$

Dabei werden 2 Stellen für die lokalen Variablen und 2 Stellen für Rücksprungadresse und den dynamischen Verweis verwendet. Nun ergibt sich daraus die Semantik der weiteren Befehle:

- $[\![\texttt{CALL}(\texttt{codeadr}, \texttt{dif}, \texttt{loc})]\!](m,d,p) := (codeadr, d, (\underline{base}(p, dif) + loc + 2) : (loc + 2) : (m+1) : \underbrace{0 : \ldots : 0}_{loc} : p)$

- $[\![\texttt{RET}]\!](m, d, p1 : \ldots : p.t) := \underline{if}\ t \geq 2 + p.2\ \underline{then}\ (p.3, d, p.(2 + p.2) : \ldots : p.t)$

`LOAD(dif, offset)` und `STORE(dif, offset)` laden bzw. speichern Variablen zwischen Datenkeller und Prozedurkeller. Die relative Adressierung erfolgt mit

- der Niveaudifferenz `dif` zwischen Auftreten und Deklaration.

- `offset` als relativer Adresse im Aktivierungsblock.

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

- $[\![\texttt{LOAD}(\texttt{dif}, \texttt{offset})]\!](m,d,p) := (m+1, d : p.[\underline{base}(p, dif) + 2 + offset], p)$

- $[\![\texttt{STORE}(\texttt{dif}, \texttt{offset})]\!](m, d : z, p) := (m+1, d, p[\underbrace{\underline{base}(p, dif) + 2 + offset}_{Adresse}/z])$

- $[\![\texttt{LIT z}]\!](m,d,p) := (m+1, d : z, p)$

AM-Code

AM-Code sind Befehlsfolgen mit aufsteigenden Befehlsmarken:

$$P \in \text{ AM-Code } :\curvearrowright P = a_1 : B_1; \ldots a_p : B_p;$$

mit $a_i \in Adr := \mathbb{N}, a_i = a_1 + i - 1$ und B_i ein AM-Befehl $(1 \leq i \leq p)$. Die Semantik von P ergibt sich durch die Iteration der Befehle gemäß des Befehlszählers BZ. Allgemein ausgedrückt ergibt sich die Funktion:

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

Diese ist wie folgt definiert:

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

4.1.2 Übersetzung von BPS-Programmen in AM-Code

Die Übersetzung erfolgt nach folgender Abbildung:

$$\underline{trans} : \text{ BPS-Prog } \longrightarrow \text{ AM-Code}$$

Dabei diene der AM-Code als Zwischencode. Als Hilfsmittel dabei bedienen wir uns der Symboltabelle:

$$\underline{Tab} := \{st \mid st : \underline{Ide} \dashrightarrow (\{\underline{const}\} \times \mathbb{Z}) \cup (\{\underline{var}\} \times \underline{Lev} \times \underline{Off}) \cup (\{\underline{proc}\} \times \underline{Adr} \times \underline{Lev} \times \underline{Size})\}$$

Variablendeklaration

Das Deklarationsniveau ist $dl \in \underline{Lev} := \mathbb{N}$, das Offset ist $off \in \underline{Off} := \mathbb{N}$.

Beachte: Die Niveaudifferenz zum Auftreten der Variable im Anweisungsteil bestimmt mit dem Offset off den Speicherplatz auf dem Prozedurkeller.

Prozedurdeklaration

Die Startadresse ist $sa \in \underline{Adr}$ des Prozedurcodes, das Deklarationsniveau ist $dl \in \underline{Lev} := \mathbb{N}$ und die Anzahl der lokalen Variablen ist $loc \in \underline{Size} := \mathbb{N}$.

Beachte: Prozedurdeklaration $\leadsto$ Codeerzeugung mit Eintrag von (sa, dl, loc) für den Prozedurbezeichner in der Symboltabelle. Der Prozeduraufruf benötigt (sa, dl, loc) und das Aufrufniveau für den Aktivierungsblock und die Codeauswahl.

Aufbau der Symboltabelle

Die Abbildung

$$\underline{up} : \underline{Decl} \times \underline{Tab} \times \underline{Adr} \times \underline{Lev} \dashrightarrow \underline{Tab}$$

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

$$\begin{aligned} \underline{up}(\Delta_C \Delta_V \Delta_P, st, a, l) \quad := \quad & \underline{if}\ dif\ fid(\Delta_C \Delta_V \Delta_P) \\ & \underline{then}\ \underline{up}(\Delta_P, \underline{up}(\Delta_V, \underline{up}(\Delta_C, st, a, l), a, l), a, l) \end{aligned}$$

- $\underline{up}(\varepsilon, st, a, l) := st$
- $\underline{up}(\underbrace{\underline{const}\ I_1 = z_1; , \ldots, I_n = z_n;}_{\Delta_C}, st, a, l) := st[I_1/(\underline{const}, z_1), \ldots, I_n/(\underline{const}, z_n)]$
- $\underline{up}(\underbrace{\underline{var}\ I_1, \ldots, I_n;}_{\Delta_V}, st, a, l) := st[I_1/(\underline{var}, l, 1), \ldots, I_n/(\underline{var}, l, n)]$
- $\underline{up}(\underbrace{\underline{proc}\ I_1, B_1; \ldots; \underline{proc} I_n, B_n;}_{\Delta_P}, st, a, l) :=$
  $st[I_1/(\underline{proc}, a_1, l, \underline{size}(B_1)), \ldots, I_n/(\underline{proc}, a_n, l, \underline{size}(B_n))]$

Dabei sind $a_1, \ldots, a_n$ symbolische, "freie" Adressen; ihre genaue Berechnung erfodert die Definition einer Codelängenfunktion. $\underline{size} : \underline{Block} \longrightarrow \mathbb{N}$ ermittelt den Speicherbedarf als Anzahl der lokalen Variablen, z.B. $\underline{size}(\Delta_C \underline{var} I_1, \ldots, I_n; \Delta_P \Gamma) := n$.

Anfangstabelle

$P = \underline{in/out}\ I_1, \ldots, I_n; B.$ hat die Semantik $\mathfrak{M}[\![P]\!] : \mathbb{Z}^n \dashrightarrow \mathbb{Z}^n$. Für $(z_1, \ldots, z_n) \in \mathbb{Z}^n$ wählen wir den Anfangszustand $s = (1, \varepsilon, 0 : 0 : 0 : z_1 : \ldots : z_n) \in ZR$ mit 1 als Startadresse des AM-Codes von P, ε als leerer Datenkeller und $0 : 0 : 0 : z_1 : \ldots : z_n$ als I/O-Block auf dem Prozedurkeller mit $sv = dv = ra = 0$ ($ra = 0$ ist Stopadresse).

Die entsprechend initialisierte Anfangstabelle hat daher n Einträge.

$$st_{I/O} = (\underline{var}, 0, j)$$

Weitere Hilfsfunktionen

Weitere Hilfsfunktionen zur Definition der Übersetzung

$$\underline{trans} : \text{ BPS-Prog } \longrightarrow \text{ AM-Code}$$

sind die folgenden:

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

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

4-11 **4-13**

4.1.3 Die Übersetzung

Die Übersetzung wird nun einmal exemplarisch schrittweise gezeigt:

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

Mit `CALL` wird der Hauptblock aufgerufen, `JMP` setzt den Standard-Stop. a_Γ ist Startadresse des Anweisungscodes von Γ in $B = \Delta\Gamma$. Die **Berechnung** erfolgt wie folgt: $a_\Gamma := 1 + \texttt{codelength}(\Delta)$

Blockübersetzung (block translation)

$$\begin{array}{lll} \underline{bt}(\Delta\Gamma, st, a, l) & := & \underline{dt}(\Delta, \underline{up}(\Delta, st, a_1, l), a_1, l) \\ & & \underline{ct}(\Gamma, \underline{up}(\Delta, st, a_1, l), a, l) \\ & & \texttt{a': RET} \end{array}$$

Deklarationsübersetzung (declaration translation)

$$\begin{array}{rll} \underline{dt}(\Delta_C\Delta_V\Delta_P, st, a, l) & := & \underline{dt}(\Delta_P, st, a, l) \\ \underline{dt}(\varepsilon, st, a, l) & := & \varepsilon \\ \underline{dt}(\underline{proc}\ I_1, B_1; \ldots; \underline{proc}\ I_n, B_n;, st, a, l) & := & \underline{bt}(B_1, st, a_1, l+1) \\ & & \vdots \\ & & \underline{bt}(B_n, st, a_n, l+1) \end{array}$$

Beachte: $st(I_d) = (proc, a_j, l+1)$, weil bt die Funktion dt und up mit dem gleichen Adreßparameter aufruft und beide Funktionen aus diesem Parameter in gleicher Weise die Adressen für die Prozedurrümpfe erzeugt.

bt wird mit dem Niveau $l+1$ aufgerufen.

Anweisungsübersetzung (command translation)

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

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

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

$$\begin{array}{lll} \underline{ct}(\underline{if}\ BE\ \underline{then}\ \Gamma_1\ \underline{else}\ \Gamma_2, st, a, l) & := & \underline{sbt}(BE, st, a, l) \\ & & \texttt{a': JFALSE a''} \\ & & \underline{ct}(\Gamma_1, st, a, l) \\ & & \texttt{a''-1: JMP a'''} \\ & & \texttt{a'': } \underline{ct}(\Gamma_2, st, a'', l) \\ & & \texttt{a''': ...} \end{array}$$

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

Ausdrucksübersetzung (expression translation)

$$\begin{array}{rcl} \underline{et}(Z, st, a, l) & := & \underline{\text{if}}\ st(I) = (\underline{const}, z)\ \underline{\text{then}} \\ & & \texttt{a: LIT Z} \end{array}$$

$$\begin{array}{rcl} \underline{et}(I, st, a, l) & := & \underline{\text{if}}\ st(I) = (\underline{var}, dl, off)\ \underline{\text{then}} \\ & & \texttt{a: LOAD}(l - dl, off) \end{array}$$

$$\begin{array}{rcl} \underline{et}(E_1 + E_2, st, a, l) & := & \underline{et}(E_1, st, a, l) \\ & & \underline{et}(E_2, st, a', l) \\ & & \texttt{a'': ADD} \end{array}$$

Beachte: et erzeugt Stackcode, dessen Berechnung den Wert des Ausdrucks auf den Datenkeller liefert.

Boolean-Ausdrucksübersetzung (strict boolean translation)

$$\begin{array}{rcl} \underline{sbt}(E_1 < E_2, st, a, l) & := & \underline{et}(E_1, st, a, l) \\ & & \underline{et}(E_2, st, a', l) \\ & & \texttt{a'': LT} \end{array}$$

$$\begin{array}{rcl} \underline{sbt}(\underline{not}\ BE, st, a, l) & := & \underline{sbt}(BE, st, a, l) \\ & & \texttt{a': NOT} \end{array}$$

$$\begin{array}{rcl} \underline{sbt}(BE_1\ \underline{and}\ BE_2, st, a, l) & := & \underline{sbt}(BE_1, st, a, l) \\ & & \underline{sbt}(BE_2, st, a', l) \\ & & \texttt{a'': AND} \end{array}$$

$$\begin{array}{rcl} \underline{sbt}(BE_1\ \underline{or}\ BE_2, st, a, l) & := & \underline{sbt}(BE_1, st, a, l) \\ & & \underline{sbt}(BE_2, st, a', l) \\ & & \texttt{a'': OR} \end{array}$$

Satz 4.1 (Korrektheit der Übersetzung) *Für jedes Programm* $P \in \underline{Prog}^{(n)}$ *und den Tupeln* $(z_1, \ldots, z_n)$, $(z'_1, \ldots, z'_n) \in \mathbb{Z}^n$ *gilt:*

$$\begin{array}{rl} & \mathfrak{M}[\![P]\!](z_1, \ldots, z_n) = (z'_1, \ldots, z'_n) \\ :\curvearrowleft & \mathfrak{I}[\![\underline{trans}(P)]\!](1; \varepsilon; 0:0:0:z_1:\ldots:z_n) = (0, \varepsilon, 0:0:0:z'_1:\ldots:z'_n) \end{array}$$

Beweis 4.2 *M.Mohnen: A Compiler Correctness Proof for the Static Link Technique by means of Evolving Algebras [Fund. Inform. 29 (1997), S. 257-303]*

$\boxed{\textbf{4-14}}$ $\boxed{\textbf{4-15}}$ $\boxed{\textbf{4-16}}$ $\boxed{\textbf{4-17}}$

4.1.4 Nicht-strikte Variante der Übersetzung

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

Idee: Vererbung von Sprungzielen für boolesche Ergebnisse

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

Hierbei werden 3 Adressen übergeben:

1. freie Anfangsadresse
2. true-Adresse
3. false-Adresse

Nach geeignete Modifikation der Übersetzung von Verzweigung und Iteration ergibt sich:

$$\begin{array}{lll}
\underline{nbt}(E_1 < E_2, st, a, a_t, a_f, l) & := & \underline{et}(E_1, st, a, l) \\
& & \underline{et}(E_2, st, a', l) \\
& & \texttt{a'': LT} \\
& & \texttt{a''+1: JFALSE } \mathtt{a_f} \\
& & \texttt{a''+2: JMP } \mathtt{a_t}
\end{array}$$

$$\underline{sbt}(\underline{not}\ BE, st, a, a_t, a_f, l) \quad := \quad \underline{nbt}(BE, st, a, a_f, a_t, l)$$

$$\begin{array}{lll}
\underline{nbt}(BE_1\ \underline{and}\ BE_2, st, a, a_t, a_f, l) & := & \underline{nbt}(BE_1, st, a, a', a_f, l) \\
& & \texttt{a':}\ \ \underline{nbt}(BE_2, st, a', a_t, a_f, l)
\end{array}$$

$$\begin{array}{lll}
\underline{nbt}(BE_1\ \underline{or}\ BE_2, st, a, a_t, a_f, l) & := & \underline{nbt}(BE_1, st, a, a_t, a', l) \\
& & \texttt{a':}\ \ \underline{nbt}(BE_2, st, a', a_t, a_f, l)
\end{array}$$

Übersetzung von Verzweigung und Iteration

Sprungziele werden an boolesche Ausdrücke weitergegeben.

$$\begin{array}{lll}
\underline{nct}(\underline{if}\ BE\ \underline{then}\ \Gamma_1\ \underline{else}\ \Gamma_2, st, a, l) & := & \underline{nbt}(BE, st, a, a_{\Gamma_1}, a_{\Gamma_2}, l) \\
& & \underline{nct}(\Gamma_1, st, a_{\Gamma_1}, l) \\
& & \texttt{a': JMP a''} \\
& & \underline{nct}(\Gamma_2, st, a_{\Gamma_2}, l) \\
& & \texttt{a'': ...}
\end{array}$$

$$\begin{array}{lll}
\underline{nct}(\underline{while}\ BE\ \underline{do}\ \Gamma, st, a, l) & := & \underline{nbt}(BE, st, a, a_t, a_f, l) \\
& & \underline{nct}(\Gamma, st, a_t, l) \\
& & \texttt{a'': JMP a} \\
& & \mathtt{a_f} = \texttt{a''+1: ...}
\end{array}$$

4.1.5 Prozeduren mit Parametern

Wir erweitern BPS um Wert- und Variablenparameter für Prozeduren.

Syntax: (modifiziert für Deklaration und Aufruf von Prozeduren)

`Ide`:	`I, J, V`		
`Cmd`:	Γ	`::=`	$\ldots \mid I(E_1, \ldots, E_p; V_1, \ldots, V_q)$
`Decl`:	$\ldots \Delta_p$	`::=`	$\varepsilon \mid$ `proc` $\mathtt{I}(\mathtt{I}_1, \ldots, \mathtt{I}_\mathtt{p};$ `var` $\mathtt{J}_1, \ldots, \mathtt{J}_\mathtt{q})\mathtt{B}; \ldots$

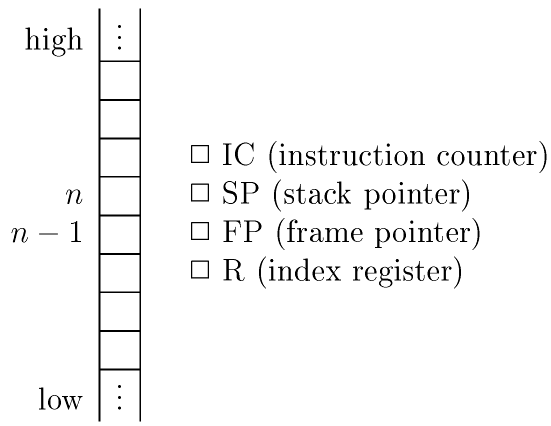
`Decl` wurde um formale Wert- bzw. Variablenparameter ergänzt, `Cmd` wurde für die aktuellen Parameterausdrücke modifiziert.

Semantik:

- **Prozedurdeklarationen:** formale Parameter behandelt als eine in der Umgebung des Programmrumpfes deklarierte Variable, entsprechende Verwendung im Rumpf
- **Prozeduraufruf:** Wertparameter als lokale Variable (neuer Speicherplatz), Variablenparameter durch Zeiger auf entsprechenden Speicherplatz aktualisieren

Zwischencode:
Aktivierungsblöcke brauchen zusätzlichen Speicher für aktuelle Parameter. Statt der Erweiterung des bisherigen Modells führen wir nun ein realistischeres Maschinenmodell ein.

Zustandsraum



Merkmale des Zustandsraumes sind:

- 1 Keller für DK und PK
- Kellerzellen mit festem Adreßraum
- SP zeigt auf Kellerspitze (in diesem Fall unten)
- FP zeigt auf den obersten Aktivierungsblock

- R zur Realisierung der Verweiskettentechnik

Struktur eines Frames:

high	⋮	
	param 1	
	⋮	aktuelle Wert- und Variablenparameter
	param r	
	sl	statischer Verweis auf Deklarationsumgebung
	ra	Rücksprungadresse
$FP \longrightarrow$	dl	dynamischer Verweis auf letzten Frame
	loc 1	
	⋮	lokale Variablen
$SP \longrightarrow$	loc t	
low	⋮	

Befehlssatz

Die arithmetischen und logischen Befehle, sowie die Sprungbefehle gelten wie bisher. Der Datenkeller liegt auf der Stackspitze, der Stackpointer (SP) muß jeweils gesetzt werden. Anstelle von CALL(ca, dif, loc), RET, LOAD(dif, off), STORE(dif, off) werden nun folgende Befehle eingeführt:

- 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>
- 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> ($R \leftarrow < n >$)
- LOAD R, R+2 ($R \leftarrow R + 2$)

Ein- und Ausgabe für $p = \underline{in/out}\ I_1, \ldots, I_n; B.$:

$$
(z_1,\ldots,z_n) \quad \longmapsto \quad
\begin{array}{r|c|l}
 & \vdots & \\ \hline
 & z_1 & \\ \hline
 & \vdots & \text{IO-Parameter als Wertparameter} \\ \hline
 & z_n & \\ \hline
 & 0 & \text{sl} \\ \hline
 & 0 & \text{ra} \\ \hline
SP, FP \longrightarrow & 0 & \text{dl} \\ \hline
IC = \boxed{1} & \vdots &
\end{array}
\quad \longmapsto \quad (z_1,\ldots,z_n)
$$

Zwischencodeerzeugung

Zur Zwischencodeerzeugung muß nun eine Modifikation der Funktion

$$\underline{trans}: \text{ BPS-Prog} \longrightarrow \text{AM-Code}$$

unter Berücksichtigung von

a) Prozeduren mit Parametern

b) einer neuen abstrakten Maschine (neuer Befehlssatz)

vorgenommen werden.

Die Symboltabelle wird um Einträge für Variablenparameter erweitert, Wertparameter werden wie lokale Variablen behandelt.

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

Negative Offsets sind wegen der Position des Frampointers möglich: $\underline{Off} := \mathbb{Z}$

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

Der Aufbau der Symboltabelle mit $\underline{\text{up}}$ erfolgt wie bereits in diesem Kapitel geschildert, allerdings mit negativen Offsets bei lokalen Variablen.

Beachte: Der Eintrag der lokalen Variablen erfolgt durch $\underline{\text{up}}$ aber von Parametern einer Prozedur durch $\underline{\text{dt}}$ (vergleiche auch folgendes Beispiel).

$$
\begin{array}{rcl}
\underline{trans}(\underline{in/out}\ I_1,\ldots,I_n; B.) & := & \texttt{1: PUSH FP} \\
 & & \texttt{2: CALL } \mathtt{a}_\Gamma \\
 & & \texttt{3: JMP 0} \\
 & & \underline{bt}(B, st_{I/O}, a_\Gamma, 1, 0)
\end{array}
$$

4-20 **4-21** **4-22** **4-23**

$$
\begin{array}{rcl}
\underline{dt}(\Delta, st, a, l) & := & \underline{dt}(\Delta_P, st, a, l) \\
\underline{dt}(\varepsilon, st, a, l) & := & \varepsilon \\
\underline{dt}(I(I_1,\ldots,I_p, \underline{var}\ J_1,\ldots,J_q); B; \Delta_P, st, a, l) & := & \underline{bt}(B, st', a_1, l+1, p+q) \\
 & & \underline{dt}(\Delta_P, st, a', l)
\end{array}
$$

Alternative zur Verweiskettentechnik

Eine Alternative zur Verweiskettentechnik stellt die Display-Technik dar. Ihr Unterschied zur Verweiskettentechnik: sie hat einen schnelleren Variablenzugriff, aber einen höheren Speicheraufwand.

- **lokale Displays:** aufgerufene Prozedur trägt alle statischen Verweise in den Frame ein
- **globale Displays:** die statischen Verweise werden global in einem SLA (static link array) gespeichert

4.2 Übersetzung von Datenstrukturen

Datenstrukturen sind Variablen mit Komponenten, also ein strukturierter Zustandsraum. Die abstrakte Maschine hat weiterhin eine lineare Speicherstruktur mit Speicherzellen für atomare Daten. Die **Übersetzungsaufgabe** ist nun, den strukturierten Zustandsraum auf den linearen Speicherbereich abzubilden (Adreßberechnung).

- **statische Datenstrukturen:** Speicherbedarf zur Übersetzungszeit bekannt
- **dynamische Datenstrukturen:** Speicherbedarf laufzeitabhängig

Es wird der Heap (Halde) als zusätzliche Maschinenkomponente benötigt. Außerdem steht die garbage collection (Speicherbereinigung) zur Verfügung.

4.2.1 Statische Datenstrukturen

Gegeben sei die Programmiersprache PSSD (Programmiersprache mit statischen Datenstrukturen), d.h. sie enthält Array (Felder) und Records. Wir haben einen induktiven Aufbau von Typen für Variablen vorliegen, es gibt keine Prozeduren.

Die abstrakte Maschine hat nun einen Hauptspeicher (HS) statt eines Prozedurkellers (also genau einen Frame) mit $ZR := BZ \times DK \times HS$ und $HS = \{h | h : \mathbb{N} \to \mathbb{Z}\}$.

$\boxed{\textbf{4-24}}$ $\boxed{\textbf{4-25}}$

Typsemantik

Ein Typ bezeichent eine Menge. So ist zum Beispiel $\mathfrak{T}[\![int]\!] := \underline{restr.}(\mathbb{Z})$ (bzw. eine im Rechner darstellbare endliche Teilmenge), $\mathfrak{T}[\![real]\!] := \underline{restr.}(\mathbb{R})$, etc.

Typkompabilität: Overloading, Typkonversion

Zuweisungskomplexität:
Das folgende Beispiel würde normalerweise einen type-mismatch-error erzeugen:

$$\begin{array}{l}
\underline{\texttt{type}}\ \texttt{I}_1 = \texttt{T}, \texttt{I}_2 = \texttt{T} \\
\underline{\texttt{var}}\ \texttt{v}_1 : \texttt{I}_1,\ \texttt{v}_2 : \texttt{I}_2 \\
\texttt{v}_1 := \texttt{v}_2
\end{array}$$

- **starkes Typkonzept:** Bezeichnerberücksichtigungen. Der Vorteil dabei ist, daß man Kontrollmöglichkeiten während der Übersetzung durch Typchecker und Sicherheit im Softwaresystem hat.
- **schwaches Typkonzept:** große Kompabilität

Der **Zustandsraum** ist strukturiert (Feld- und Verbundkomponenten):

a			$\cdots$	
	a[0]	a[1]	$\cdots$	a[17]

$\boxed{\textbf{4-26}}$ $\boxed{\textbf{4-27}}$

Erläuterungen zur Symboltabelle:

- Basiswerte benötigen genau 1 Speicherplatz
- $(\underline{type}, \underline{array}, Z_1, Z_2, I, n)$ "array descriptor" (dope vector), mit Z_1, Z_2 als Grenzen des Array, dem Typ I und dem Speicherbedarf n.
- $(\underline{type}, \underline{record}, I_1, I_1', O_1, \ldots, I_n, I_n', O_n, s, n)$ "record descriptor", mit I_i als Selektor, dem Typ I_i', dem Offset O_i und dem Speicherbedarf s. Es liegt ein indizierter "Table-Lookup" vor: $st(I, I_j) := (I_j', O_j)$.
- $(\underline{var}, I, h)$ mit dem Typ I und dem Offset h.

Aufbau einer Symboltabelle

Die Funktion $\underline{\text{up}}$ ist nun wie folgt definiert:

$$\underline{up} : \underline{Decl} \times \underline{Tab} \dashrightarrow \underline{Tab}$$

Die explizite Typschachtelung wird mit neuen Bezeichnern aufgelöst. Der Einfachheit halber nehmen wir an, daß $\Delta = \Delta_C \Delta_T \Delta_V \in \underline{Decl}$ entschachtelt ist, d.h.

1. Die Bezeichner sind paarweise verschieden.
2. Aus $\Delta_T = \underline{type}\ I1 = T_1, \ldots, I_n = T_n$ folgt:
   - $T_i \in \{bool, real, int\}$ **oder**
   - $T_i \in \{I_1, \ldots, I_{i-1}\}$ **oder**
   - $T_i = \underline{array}[Z_1..Z_2]\ of\ I_j$ mit $1 \leq j \leq i-1$ **oder**
   - $T_i = \underline{record}\ S_1 : I_{j_1}, \ldots, S_r : I_{j_r}\ \underline{end}$ mit $1 \leq j_1, \ldots, j_r \leq i-1$
3. $\Delta_V = \underline{var}\ v_1 : I_{j_1}; \ldots; v_s : I_{j_s}$ mit $1 \leq j_1, \ldots, j_s \leq n$

Für geschachtelte Funktionen ist up nun wie folgt deklariert:

- $(\Delta_c\Delta_T\Delta_V, st) := \underline{up}(\Delta_V, \underline{up}(\Delta_T, \underline{up}(\Delta_c, st)))$
- $\underline{up}(\varepsilon, st) := st$
- $\underline{up}(\underline{const}\ I_1 = c_1, \ldots, st) := st[I_1/(\underline{const}, c_1), \ldots]$
- $\underline{up}(\underline{type}\ I = bool;\ drest, st) := \underline{up}(\underline{type}\ drest, st[I/(\underline{type}, bool, 1)])$
- $\underline{up}(\underline{type}\ I = J;\ drest, st) := \underline{up}(\underline{type}\ drest, st[I/st(J)])$

$\boxed{\textbf{4-24}}$ $\boxed{\textbf{4-27}}$

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

Übersetzung von PSSD-Programmen

Zur Übersetzung von PSSD-Programmen definieren wir uns die folgende Hilfsfunktion:

$$\underline{vtyp} : \underline{var} \times \underline{Tab} \dashrightarrow \underline{Ide}$$

Diese Funktion bestimmt für eine Variable bezüglich einer Symboltabelle den zugehörigen Typbezeichner.

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

Die Übersetzungsfunktionen vt und et:

- vt berechnet auf dem Datenkeller die Anfangsadresse einer Variablen im Hauptspeicher.

- $\underline{et}$ erzeugt den üblichen Datenkeller-Code, wobei jetzt mit `LODI` Variablenwerte geladen werden.

$$\underline{vt} : \underline{var} \times \underline{Tab} \dashrightarrow \underline{Code}$$

$\underline{vt}(I, st)$:= $\underline{\text{if}}$ $st(I) = (\underline{var}, J, k)$ $\underline{\text{then}}$
`LIT k;` (k ist Anfangsadresse der Variablen)

$\underline{vt}(V[E], st)$:= $\underline{\text{if}}$ $\underline{vtyp}(V, st) = I$ $\underline{\text{and}}$ $st(I) = (\underline{type}, \underline{array}, Z_1, Z_2, J, n)$
$\underline{\text{and}}$ $st(J) = (\underline{type}, \ldots, k)$ $\underline{\text{then}}$
$\underline{vt}(V, st)$;
$\underline{et}(E, st)$;
`JAC(Z1, Z2)`
`LIT Z1;`
`SUB;`
`LIT k;`
`MULT;`
`ADD;`

$\underline{vt}(V.I, st)$:= $\underline{\text{if}}$ $\underline{vtyp}(V, st) = J$ $\underline{\text{and}}$ $st(J.I) = (J', O)$ $\underline{\text{then}}$
$\underline{vt}(V, st)$;
`LIT O;`
`ADD;`

Beachte dabei, daß die Offsetberechnung bei Arrayindizes laufzeitabhängig ist. Bei Records werden die Offsets schon in die Symboltabelle eingetragen.

Ausdrucksübersetzung (expression translation):

$$\underline{et} : \underline{Exp} \times \underline{Tab} \dashrightarrow \underline{Code}$$

$\underline{et}(I, st)$:= $\underline{\text{if}}$ $st(I) = (\underline{const}, c)$ $\underline{\text{then}}$
`LIT c;`
$\underline{\text{if}}$ $st(I) = (\underline{var}, J, k)$ $\underline{\text{and}}$ $st(J) = (\underline{type}, bas, 1)$ $\underline{\text{then}}$
`LIT k;`
`LODI;`

$\underline{et}(c, st)$:= `LIT c;`

$\underline{et}(V, st)$:= $\underline{\text{if}}$ $\underline{vtyp}(V, st) = J$ $\underline{\text{and}}$ $st(J) = (\underline{type}, bas, 1)$ $\underline{\text{then}}$
$\underline{vt}(V, st)$;
`LODI;`

$\underline{et}(E_1 + E_2, st)$ wie bereits gehabt

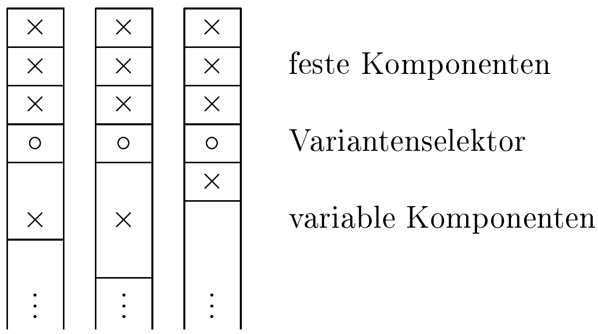
Operationsübersetzung (operation translation):

$$\underline{ot}(V := E, st) \quad := \quad \begin{array}{l} \underline{\text{vt}}(\text{V,st}); \\ \underline{\text{et}}(\text{E,st}); \\ \texttt{STOREI;} \end{array}$$

4.2.2 Dynamische Datenstrukturen

Variante Records

Der Speicherbedarf wird für die größte Variante vorgehalten. Derselbe Bereich wird für verschiedene Varianten vorgesehen.

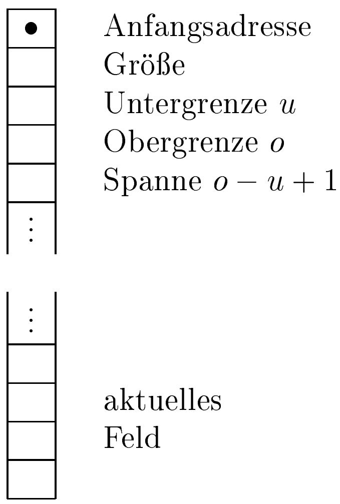


3 Varianten

Dynamische Arrays

Hier sind Felder als formale Prozedurparameter bzw. Felder mit variabler Feldgröße möglich. Die Feldgrenze wird nicht bei der Deklaration, sondern erst beim Aufruf durch aktuelle Parameter festgelegt. Bei der Speicherreservierung ist zu beachten, daß der Bedarf zur Übersetzungszeit unbekannt ist, aber bei Eintritt in die Prozedur bestimmbar ist. Die Implementierung erfolgt ohne Heap.

Die Idee: **indirekte Adressierung** mit Hilfe eines **Felddescriptors**.



Der erste Bereich dabei ist statisch und findet sich im Parameterbereich des Frames, der zweite Teil hingegen beinhaltet das aktuelle Feld und wird dynamisch bei Eintritt in die

Prozedur behandelt. Die Anfangsadresse verweist dabei auf den Speicherbereich des ersten Elementes des Feldes.

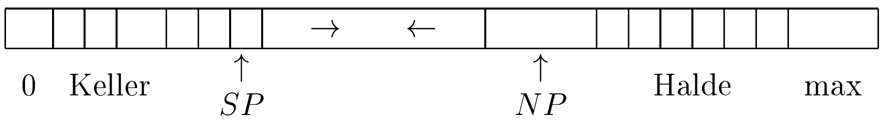
Zeiger und dynamische Speicherbelegung

Zeiger (pointer) sind dynamisch veränderbare Strukturen. Die Erzeugung von Objekten erfolgt dabei nicht durch Deklarationen, sondern durch Anweisungen (z.B. `NEW`).

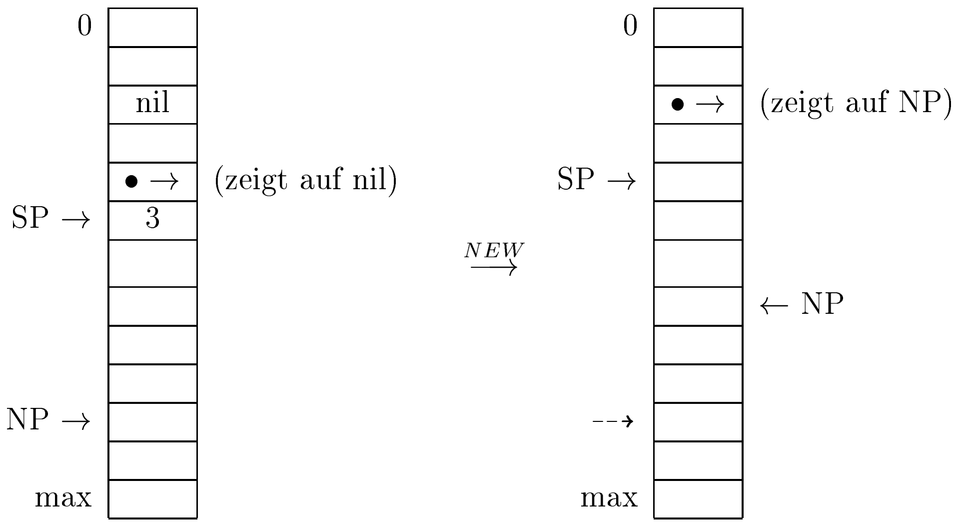
Beispiel: Sei `p : POINTER TO t;` eine Zeigervariable. Dann wird mit der Anweisung `NEW(p);` neuer Speicherplatz Speicherplatz für diese Variable erzeugt.

Die Speicherfreigabe erfolgt nicht automatisch: es sind spezielle Algorithmen für die Speicherbereinigung (garbage collection) notwendig.

Die Kellertechnik ist bei dieser Form nicht mehr ausreichend. Es wird ein neuer Speicherbereich benötigt: der Heap (Halde).



Der `NEW`-Befehl:



EP ist der **extreme Stackpointer**. Er bezeichnet die Oberschranke des Kellers zur Durchführung eines Prozeduraufrufs, bestimmt ferner durch die Größe der Ausdrücke im Prozedurrumpf und ggf. die Größe dynamischer Felder. **D.h. der extreme Stackpointer ist beim Prozedureintritt berechenbar.**

`NEW`-Semantik:

```
if NP- < SP > ≤ EP then
   error("store overflow")
else
   NP := NP-<SP>;
   <<SP-1>> := NP;
   SP := SP-2;
end.
```

Die Übersetzung von `new(p)` erfolgt dann mit folgender Befehlsfolge:

```
LOAD adr(p);
LOAD size(p);
NEW;
```

Kapitel 5

Erzeugung von Maschinencode

5-1 **5-2**

Der **Befehlsvorrat** ist die Anzahl der Maschinenbefehle: $\sim 30{-}100$ abhängig von der Zahl der Operationen und Adressierungsart der Operationen. Die **Befehlsarten** sind arithmetische, logische, Transport-, Sprung, Unterprogramm- und Kommunikationsbefehle.

Maschinenbefehle: (in lesbarer Assemblernotation)

<Resultat> ::== <Operand> op <Operand> | <Operand>
<Operand> ::== R_i | M[<Adr>] | c
<Resultat> ::== R_i | M[<Adr>]

Dabei bezeichnet R_i das Register i, M[<Adr>] eine Hauptspeicherzelle mit der Adresse <Adr> und c eine Konstante.

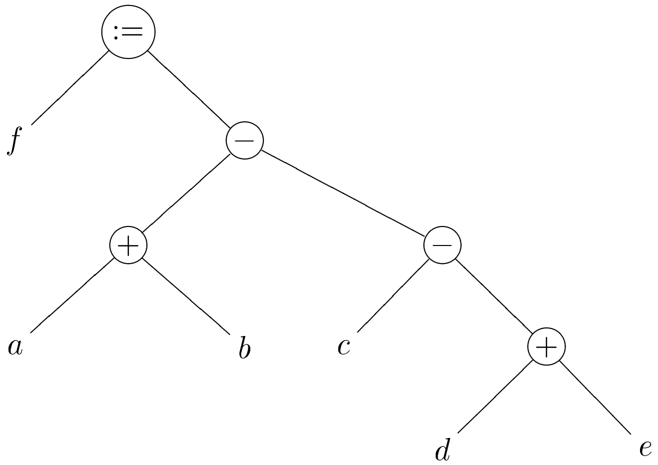
Adressierungsarten:

<Adr> ::== R_i | c | $++R_i$ | $--R_i$ | R_i++ | R_i-- | M[<Adr>] |
<Adr> + <Adr> | <Adr> * c | <Adr>.<size>
<size> ::== L | W | B (Long, Word und Byte)

5-3 **5-4**

Codeerzeugung für Ausdrücke

Beispiel: `f:=(a+b)-(c-(d+e))`



Annahme: Die Zielmaschine arbeitet mit r Registern $R_0, R_1, \ldots, R_{r-1}$.

Dann stehen folgende Befehle zur Verfügung:

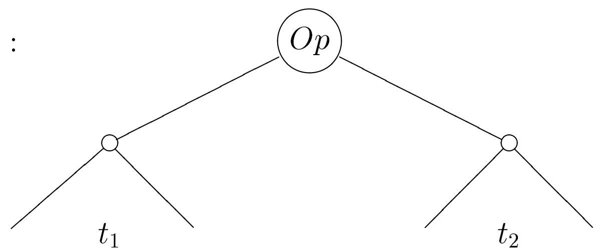
$\mathtt{R_i := M[V]}$
$\mathtt{M[V] := R_i}$
$\mathtt{R_i := R_i\ op\ M[V]}$
$\mathtt{R_i := R_i\ op\ R_j}$

Mögliche Befehlsfolge:	**Kürzere Variante:**
$\mathtt{R_0 := M[a]}$	$\mathtt{R_0 := M[c]}$
$\mathtt{R_0 := R_0 + M[b]}$	$\mathtt{R_1 := M[d]}$
$\mathtt{R_1 := M[d]}$	$\mathtt{R_1 := R_1 + M[e]}$
$\mathtt{R_1 := R_1 + M[e]}$	$\mathtt{R_0 := R_0 - R_1}$
$\mathtt{M[t_1] := R_1}$	$\mathtt{R_1 := M[a]}$
$\mathtt{R_1 := M[c]}$	$\mathtt{R_1 := R_1 + M[b]}$
$\mathtt{R_1 := R_1 - M[t_1]}$	$\mathtt{R_1 := R_1 - R_0}$
$\mathtt{R_0 := R_0 - R_1}$	$\mathtt{M[f] := R_1}$
$\mathtt{M[f] := R_0}$	

Prinzip für Optimierung

Sei folgender Baum vorausgesetzt:

t :



Die Annahme: t_i benötige r_i Register zur Auswertung $(i = 1, 2)$.

Aus dieser Annahme leiten sich nun die folgenden 3 Fälle ab:

- **Fall 1:** $r_1 < r_2 \leq r$
  In diesem Fall kann t mit r_2 Registern ausgewertet werden: erst t_2 mit einem Register für das Ergebnis, dann t_1 mit $r_1 + 1 \leq r_2$.
- **Fall 2:** $r_1 = r_2 \leq r$
  In diesem Fall sind $r_2 + 1$ Register erforderlich.
- **Fall 3:**
  Sind mehr als r Register erforderlich, so erfolgen Zwischenspeicherungen.

Der **Algorithmus** ist in 2 Teile geteilt: die **Markierungsphase** und die **Generierungsphase**.

1. **Markierung:** S-Attributierung, d.h die Knoten werden mit dem Registerbedarf versehen.
$$r(op(t_1, t_2)) = \begin{cases} max(r(t_1), r(t_2)) & \text{falls } r(t_1) \neq r(t_2) \\ r(t_1) + 1 & \text{falls } r(t_1) = r(t_2) \end{cases}$$
   Linke Blätter werden mit 1 gekennzeichnet, rechte Blätter mit 0.

2. **Generierung:** optimaler Code für den Ausgangsbaum.
   Eingabe: Ausdrucksbaum t (mit markiertem Registerbedarf)
   Ausgabe: optimaler Code für t (kürzester Code mit kleinster Registerzahl)

```
var R:Registeradresse;  T:Hauptspeicheradresse;
var RSTACK : Stack of Registeradresse;
var TSTACK : Stack of Hauptspeicheradresse;

proc Code(t : Ausdrucksbaum mit Registerbedarf)
  case t of
   (leaf a,1) :
      output(top(RSTACK) := M[a]);

   op((t1,r1),(leaf a,0)) :
      Code(t1);
      output(top(RSTACK) := top(RSTACK) op M[a]);

   op((t1,r1),(t2,r2)) :
      cases
        - r1 < r2 und r1 < r :
            exchange(RSTACK);
            Code(t2);
            R := pop(RSTACK);
```

```
                Code(t1);
                output(top(RSTACK) := top(RSTACK) op <R>);
                push(RSTACK, R);
                exchange(RSTACK);
            - r1 >= r2 und r2 < r :
                Code(t1);
                R := pop(RSTACK);
                Code(t2);
                output(<R> := <R> op top(RSTACK));
                push(RSTACK, R);
            - r1 >= r und r2 >= r :
                Code(t2);
                T := pop(TSTACK);
                output(M[T] := top(RSTACK));
                Code(t1);
                output(top(RSTACK) := top(RSTACK) op M[T]);
                push(TSTACK, T)
          end
      end
    end
```

Optimalität:

	1 Befehl für jeden inneren Knoten
+	1 Befehl für ein linkes Blatt
+	1 Befehl für solche inneren Knoten,
	deren Unterbäume nicht mit r Registern auswertbar sind

Das **Problem** dabei ist: gemeinsame Teilausdrücke werden mehrfach ausgewertet. Daraus folgt für die Optimalität die **NP-Vollständigkeit**.

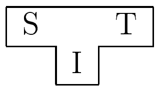
Kapitel 6

Compilerentwicklung, Bootstrapping

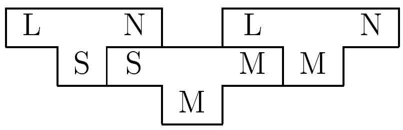
Einen Übersetzungsvorgang kann man mit folgendem Schema beschreiben:

$$\text{Quellsprache S} \xrightarrow{trans} \text{Zielsprache T (Maschine)}$$

Die Implementierungssprache des Compiler trans wird in Sprache I geschrieben. Zur Vereinfachung wählt man die folgende Schreibweise in Form eines T-Diagramms :



Die Grundidee ist, daß Compiler selbst Programme sind, die mit Compilern übersetzt werden können. In T-Diagramm-Schreibweise wird das wie folgt dargestellt:



Eine alternative Schreibweise ergibt sich zu:

$$\begin{array}{ccc} L & & N \\ & S & \end{array} + \begin{array}{ccc} S & & M \\ & M & \end{array} = \begin{array}{ccc} L & & N \\ & M & \end{array}$$

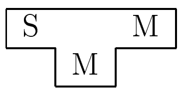
D.h. ein Compiler für $L \rightarrow N$ wird in S geschrieben. Für S existiert ein Compiler auf M. Dieser erzeugt einen lauffähigen Compiler für $L \rightarrow N$ auf M.

Definition 6.1 (Crosscompiler) *Ein Crosscompiler ist ein Compiler, der auf einer Maschine M_1 läuft und Code für eine Maschine M_2 erzeugt.*

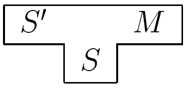
6.0.3 Sukzessive Compilerentwicklung durch "Bootstrapping"

Die Aufgabe ist, eine neue Sprache L auf einer Maschine M zu implementieren.

1. kleine Teilmenge $S \subseteq L$ von Hand implementieren



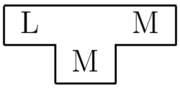
2. größere Teilmenge S' (eventuell auch schon ganz L) mit der Eigenschaft $S \subseteq S' \subseteq L$ in S auf M implementieren



3. Bootstrap-Schritt:

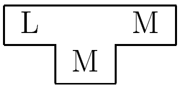
$$\begin{matrix} S' & & M \\ & S & \end{matrix} + \begin{matrix} S & & M \\ & M & \end{matrix} = \begin{matrix} S' & & M \\ & M & \end{matrix}$$

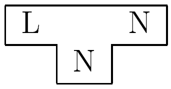
4. weiteres Bootstrapping liefert lauffähigen L-Compiler für M



6.0.4 Compilerportierung

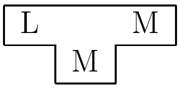
Die Aufgabe ist, einen lauffähigen L-Compiler für eine Maschine M auf eine Maschine N zu portieren.

Gegeben: 

Gesucht: 

Die Methode läuft dann wie folgt:

1. Schreibe den gesuchten Compiler in L:



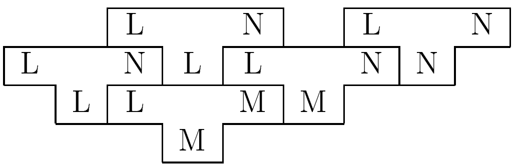
2. Übersetze diesen Compiler aus dem ersten Schritt in einen lauffähigen Crosscompiler:

$$\begin{matrix} L & & N \\ & L & \end{matrix} + \begin{matrix} L & & M \\ & M & \end{matrix} = \begin{matrix} L & & N \\ & M & \end{matrix}$$

3. Übersetze den Compiler aus dem ersten Schritt ein zweites Mal mit dem erzeugten Crosscompiler:

$$\begin{matrix} L & & N \\ & L & \end{matrix} + \begin{matrix} L & & N \\ & M & \end{matrix} = \begin{matrix} L & & N \\ & N & \end{matrix}$$

Dieser ganze Ablauf kann auch als (etwas verwirrendes) T-Diagramm dargestellt werden.



- E N D E -

Index