INSTITUT FÜR INFORMATIK
Lehr- und Forschungseinheit für
Programmier- und Modellierungssprachen
Oettingenstraße 67, D–80538 München

Ludwig
Maximilians-
Universität
München

Übersetzerbau – Abstrakte Maschinen

François Bry, Norbert Eisinger

Skriptum/Lecture Notes, Sommersemester 2004
http://www.pms.ifi.lmu.de/publikationen

Zusammenfassung

Die Übersetzung einer Programmiersprache bildet höhere Sprachkonstrukte wie etwa Blöcke und Schleifen, die die Programmierung erleichtern, auf die maschinennahen Datenstrukturen und Befehle einer Prozessorsprache ab. Um eine Programmiersprache sowohl unabhängig von einem bestimmten Prozessor implementieren zu können als auch ihre prozedurale Semantik lückenlos zu spezifizieren, wird eine sogenannte abstrakte Maschine definiert. Dabei handelt es sich um eine detaillierte Spezifikation der Implementierung der Programmiersprache in einer idealisierten maschinennahen Sprache.

Dieses Skriptum führt in abstrakte Maschinen für Programmiersprachen der drei Programmierparadigmen imperativ (wie z.B. Pascal), funktional (wie z.B. Haskell) und logisch (wie z.B. Prolog) ein. Für jedes dieser drei Programmierparadigmen werden eine vereinfachte Programmiersprache, eine abstrakte Maschine für diese Sprache, sowie die Übersetzung der Programmiersprache in die Befehle der zugehörigen abstrakten Maschine eingeführt.

Danksagung

Dieses Skriptum ist gegenüber einer früheren Version vom WS 2001/02 in einigen – nicht in allen – Teilen stark überarbeitet. Die äußere Form sollte aber nicht darüber hinwegtäuschen, dass auch die vorliegende Version kein Lehrbuch ist, sondern sich noch in der Entwicklung befindet.

Wir danken zahlreichen Studenten für wertvolle Hinweise, die in die Überarbeitung eingeflossen sind, sowie Sven Panne, der an der Erstellung der Versionen bis zum WS 2001/02 mitgearbeitet und zu ihrem Inhalt und ihrer Gestalt beigetragen hat.

München, im August 2004

François Bry, Norbert Eisinger

Inhaltsverzeichnis

1 Einleitung

Übersetzungstechniken werden nicht nur zur Implementierung von Programmiersprachen verwendet, sondern auch in vielen Softwaresystemen, die eingeschränkte Sprachen einsetzen. Beispiele von solchen eingeschränkten Sprachen sind:

- Befehlssprachen von Betriebssystemen
- Textformatierungs- und Graphikspezifikationssprachen (wie z. B. TEX, LATEX, HTML)
- Hardware-Beschreibungsprachen zur Spezifikation von Hardware-Systemen
- Sprachen, die von Web-Browsern interpretiert werden (wie z. B. CSS)

Darüberhinaus führen Kenntnisse der Übersetzung zu einer besseren Verwendung von Programmiersprachen.

Die Entwicklung des Faches kann wie folgt skizziert werden:

60-er und 70-er Jahre. Übersetzerbau für imperative Programmiersprachen wird von einer Kunst zu einer Technik.

70-er Jahre. Übersetzung funktionaler Programmiersprachen.

80-er Jahre. Übersetzung logischer Programmiersprachen.

90-er Jahre. Übersetzung funktionaler Programmiersprachen mit verzögerter Auswertung (lazy evaluation) und Übersetzung von Typen und Vererbung (bei objektorientierten Programmiersprachen).

1.1 Grundbegriffe

Eine Programmiersprache kann in zwei Weisen implementiert werden: Mittels Interpretation oder mittels Übersetzung.

Interpretation. Ein (korrekter) Interpretierer für eine bestimmte Programmiersprache S bekommt als Eingabe:

- ein Programm P in S
- Eingaben E zu P

Er berechnet daraus eine Ausgabe A zu P oder stößt auf einen Fehler in P. Dabei werden Programm P und Eingabe E zur gleichen Zeit bearbeitet. Es wird kein Ergebnis der Bearbeitung eines Konstruktes von P zur späteren Wiederverwendung behalten. Keine nur aus P gewonnenen Daten werden zur Optimierung verwendet, wie z. B. die Anzahl der deklarierten Variablen oder, ob überhaupt eine Anweisungsfolge ausgeführt werden kann.

Übersetzung. Ziel der Übersetzung ist, eine effizientere Bearbeitung eines Programms als durch seine Interpretation zu ermöglichen. Dafür wird das Programm P zunächst unabhängig von jeglicher Eingabe bearbeitet. Es wird analysiert und in eine Form gebracht, die die effiziente Bearbeitung jeglicher (zulässigen) Eingabe ermöglicht.

Zur Übersetzungzeit: Aus einem Programm P in einer Quellsprache S wird ein Zielprogramm P_M in einer Maschinensprache (oder Assemblersprache) M gewonnen.

Zur Laufzeit: P_M wird mit einer Eingabe E unter Anwendung eines Interpretierers I_M für die Maschinensprache M ausgeführt. Dieser Interpretierer I_M kann die Hardware einer konkreten Maschine sein oder seinerseits durch ein Programm realisiert sein.

Dabei soll gelten: Wenn die Interpretation eines korrekten Programms P mit einer Eingabe E die Ausgabe A liefert, dann liefert die Ausführung des Zielprogramms P_M mit Eingabe E ebenfalls die Ausgabe A. Der Übersetzer soll also eine *bedeutungserhaltende Umwandlung* des Programm P in das Zielprogramm P_M durchführen, wobei die Bedeutung von P durch den Interpretierer der Quellsprache gegeben ist. Das heißt, dass der Interpretierer der Quellspache als Spezifikation für den Übersetzer dient.

Fehlerbehandlung. Fehler der folgenden Arten können im Quellprogramm vorkommen:

Übersetzungszeitfehler. Diese werden nach den entsprechenden Übersetzungsphasen (siehe unten) in lexikalische, syntaktische und semantische Fehler unterschieden. Der Übersetzer soll alle derartigen Fehler im Quellprogramm P erkennen, auch wenn ein Interpretierer das Programm P mit manchen Eingaben korrekt ausführen könnte, weil mit diesen Eingaben die fehlerhafte Stelle im Programm gar nicht zur Ausführung käme.

Laufzeitfehler. Wenn der Interpretierer für die Quellsprache S einen Fehler im Quellprogramm P endeckt, der kein Fehler der obigen Art ist (z. B. Division eines eingelesenen Wertes durch sich selbst bei Eingabe 0), dann soll auch der Interpretierer I_M der Maschinensprache angewandt auf das Zielprogramm P_M einen entsprechenden Fehler melden. Auch für die Fehlerbehandlung dient also der Interpretierer der Quellsprache als Spezifikation für den Übersetzer.

Die deutschsprachige Bezeichnung „Übersetzung“ stellt den Zweck in den Vordergrund, die englischsprachige Bezeichnung „compilation“ (Zusammenfassung, Überprüfung) eher das Mittel zum Zweck.

Abstrakte Maschinen. Anstatt eine Zielsprache zu verwenden, die von einer *konkreten Maschine* (d. h. von einem bestimmten Rechnertyp) abhängt, werden in der Regel Quellprogramme in eine Zielsprache für eine sogenannte *abstrakte Maschine* übersetzt, die für den Sprachtyp besonders geeignet ist. Die abstrakte Maschine kann dann unabhängig vom Übersetzer auf konkreten Maschinen realisiert werden. Dies erleichtert:

- die Implementierung des Übersetzers,
- die Portierung des Übersetzers auf verschiedene Rechnertypen und
- die Überprüfung der Korrektheit der Übersetzung.

Phasen und Läufe. Eine inhaltlich zusammenhängende Teilfunktionalität der Übersetzung wird als *Phase* bezeichnet und üblicherweise durch eine eigene Komponente des Übersetzers realisiert. Die folgende Tabelle zeigt die typischen Phasen.

Die Eingabe des Übersetzers ist ein Quellprogramm dargestellt als eine Folge von Zeichen (character). Daraus müssen zunächst „Wörter“ erkannt werden, also zum Beispiel reservierte Bezeichner der Quellsprache oder auch Teilfolgen von Zeichen wie `>=` die als Notation für ein einziges Symbol dienen (in diesem Fall das Größergleichsymbol). Diese „Wörter“ nennt man Symbole (token). Die Phase, die die Zeichenfolge in eine Symbolfolge übersetzt, heißt lexikalische Analyse.

Ein Bearbeitungsschritt der Übersetzung, bei dem das gesamte Quellprogramm durchlaufen wird (egal ob in der Repräsentation als Zeichenfolge oder in einer der daraus erzeugten Repräsentationen), nennt man einen *Lauf*. Man kann jede Phase als einen eigenen Lauf organisieren, der das Quellprogramm vollständig von der Eingaberepräsentation der Phase in die Ausgaberepräsentation der Phase übersetzt. Dann beginnt eine Phase also erst wenn die vorherige beendet ist. Diese Organisation ist für das Verständnis am einfachsten.

Die Effizienz des Übersetzers kann dadurch verbessert werden, dass man die Phasen miteinander „verzahnt“, so dass insgesamt so wenige Läufe wie möglich durchgeführt werden. Manche Quellsprachen sind bewusst so gestaltet, dass sie in einem einzigen Lauf übersetzt werden können, aber für viele gängige Sprachen ist das nicht möglich.

Übersetzungsphase	**Komponente des Übersetzers**	**Eingabe**	**Ausgabe**
Lexikalische Analyse	Scanner, Lexer oder Symbolentschlüssler	Zeichenfolge	Symbolfolge
Syntaktische Analyse	Parser oder Zerteiler	Symbolfolge	Syntaxbaum
Semantische Analyse	Attributsauswerter	Syntaxbaum	attribuierter Syntaxbaum bzw. Zwischencode
Code-Erzeugung	möglichst Attributsauswerter	attribuierter Syntaxbaum bzw. Zwischencode	Programm in der Zielsprache oder Code
Code-Optimierung		Programm in der Zielsprache oder Code	Programm in der Zielsprache oder Code
Assemblierung	Assembler	Programm in der Zielsprache oder Code	Binärcode

1.2 Literatur

[1] Reinhard Wilhelm und Dieter Maurer: *Übersetzerbau. Theorie, Konstruktion, Generierung.* Springer-Lehrbuch, Springer Verlag, 2. Auflage 1996

[2] Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullman: *Compiler. Principles, Techniques, and Tools.* Addison-Wesley, 1986

[3] Alfred V. Aho, Ravi Sethi und Jeffrey D. Ullman: *Compilerbau.* Addison-Wesley, 1988 (Deutsche Übersetzung von [2])

[4] Niklaus Wirth: *Grundlagen und Techniken des Compilerbaus.* Addison-Wesley, 1996

[5] Simon L. Peyton Jones: *The Implementation of Functional Programming Languages.* Prentice-Hall, 1987

[6] Harold Abelson and Gerald J. Sussman with Julie Sussman: *Structure and Interpretation of Computer Programs.* MIT Press, 1985

[7] Harold Abelson und Gerald J. Sussman mit Julie Sussman: *Struktur und Interpretation von Computerprogrammen.* Springer-Verlag, 1985 (Deutsche Übersetzung von [6])

[8] Hassan Aït Kaci: *Warren's Abstract Machine.* MIT Press, 1991

[9] Matthias Kalle Dalheimer: *Java Virtual Machine — Sprache, Konzept, Architektur.* O'Reilly, 1997

[2] ist das bekannteste Referenzwerk über Algorithmen und Methoden des Übersetzerbaus. Seine Übersetzung [3] in Deutsch mag für den deutschsprachigen Leser leichter zugänglich sein.

Wie diese Vorlesung führt [1] in abstrakte Maschinen ein. Die in [1] beschriebenen abstrakten Maschinen sind jedoch nicht identisch mit den abstrakten Maschinen dieser Vorlesung. Besonders bei der Übersetzung funktionaler Programmiersprachen ist der Unterschied zwischen [2] und dieser Vorlesung wesentlich: Während in dieser Vorlesung eine abstrakte Maschine zur verzögerten (lazy) Auswertung funktionaler Programmiersprachen eingeführt wird, behandelt [1] die Auswertung funktionaler Programmiersprachen in applikativer Reihenfolge.

[4] ist eine leichtverständliche Einführung in die Übersetzung imperativer Programmiersprachen. Wie diese Vorlesung stellt [4] eine vereinfachte Nachahmung der abstrakten Maschine der Programmiersprache Pascal dar.

[5] führt ausführlich in die Implementierung der verzögerten Auswertung funktionaler Programmiersprachen ein. Die in dieser Vorlesung dargestellte abstrakte Maschine für funktionale Programmiersprachen ist ein Nachahmung der in [5] eingeführten Maschine.

[6] führt leichtverständlich in die Übersetzung zur Auswertung funktionaler Programmiersprachen in applikativer Reihenfolge sowie in Grundkenntnisse des Algorithmenentwurfes und der Programmierung ein. [7] ist eine deutsche Übersetzung von [6].

[8] ist das Referenzwerk über die WAM, eine abstrakte Maschine für die Programmiersprache Prolog, der bekanntesten Sprache der logischen Programmierung.

Der Begriff „abstrakte Maschine" hat in den letzten Jahren durch die Programmiersprache Java viel Aufmerksamkeit erhalten. [9] führt in diese Programmiersprache und ihre Übersetzung ein.

1.3 Formale Sprachen, Grammatiken, Automaten

Die lexikalische und syntaktische Analyse verwenden die folgenden Begriffe aus der theoretischen Informatik: Formale Sprachen, Grammatiken, reguläre und kontextfreie Sprachen.

Definition (Alphabet und Formale Sprache). Ein *Alphabet* ist eine nichtleere endliche Menge, deren Elemente *Symbole* genannt werden. Sei Σ ein Alphabet. Dann ist Σ^* die Menge aller *Wörter* (also endlichen Folgen) von Elementen von Σ. Eine *formale Sprache* über Σ ist eine Teilmenge von Σ^*. ■

Notationen. Das leere Wort wird ε geschrieben. Ist $\Sigma = \{a, b\}$, so ist Σ^* die unendliche Menge $\{\varepsilon, a, b, aa, ab, ba, bb, aaa, aab, aba, abb, \ldots\}$ und Σ^+ die unendliche Menge $\Sigma^* \setminus \{\varepsilon\}$. ■

Manche formalen Sprachen (nicht alle, jedoch alle Programmiersprachen!) lassen sich anhand von Grammatiken endlich beschreiben.

Definition (Grammatik, kontextfreie Grammatik, reguläre Grammatik). Eine *Grammatik* ist ein 4-Tupel $G = (V, \Sigma, P, S)$, das folgende Bedingungen erfüllt:

- V ist ein endliche Menge. Ihre Elemente heißen *Variablen* oder *Nichtterminalsymbole* von G.
- Σ ist eine nichtleere endliche Menge mit $V \cap \Sigma = \emptyset$. Sie heißt *Terminalalphabet*, ihre Elemente heißen *Terminalsymbole* von G.
- $P \subseteq (V \cup \Sigma)^+ \times (V \cup \Sigma)^*$ und P ist endlich. Die Elemente von P heißen *Produktionen* oder *Regeln* von G. Eine Produktion $(w_1, w_2) \in P$ wird $w_1 \to w_2$ dargestellt.
- $S \in V$. Dieses ausgezeichnete Nichtterminalsymbol heißt die *Startvariable* oder das *Startsymbol* von G.

Eine Grammatik $G = (V, \Sigma, P, S)$ heißt *kontextfrei*, falls für alle Regeln $w_1 \to w_2 \in P$ gilt: $w_1 \in V$ (d. h., die linke Seite der Regel besteht aus genau einem Nichtterminalsymbol).

Eine Grammatik $G = (V, \Sigma, P, S)$ heißt *rechtslinear*, falls für alle Regeln $w_1 \to w_2 \in P$ gilt: $w_1 \in V$ (d. h., G ist kontextfrei) sowie $w_2 \in \Sigma^*$ oder $w_2 = uA$ mit $u \in \Sigma^*$ und $A \in V$.

Eine Grammatik $G = (V, \Sigma, P, S)$ heißt *linkslinear*, falls für alle Regeln $w_1 \to w_2 \in P$ gilt: $w_1 \in V$ (d. h. G ist kontextfrei) sowie $w_2 \in \Sigma^*$ oder $w_2 = Au$ mit $u \in \Sigma^*$ und $A \in V$.

Eine Grammatik $G = (V, \Sigma, P, S)$ heißt *regulär*, falls sie linkslinear oder rechtslinear ist. ■

Bemerkung. Oft werden reguläre Grammatiken als rechtslineare Grammatiken definiert. Dies widerspricht der obigen Definition nicht, weil jede linkslineare Grammatik in eine äquivalente rechtslineare Grammatik umgewandelt werden kann (Übung!).

Die Definitionen lassen auch ε-Produktionen zu, also Produktionen, deren rechte Seite das leere Wort ist. Jede kontextfreie Grammatik mit ε-Produktionen kann in eine äquivalente kontextfreie Grammatik umgewandelt werden, die höchstens noch eine ε-Produktion $S \to \varepsilon$ für das Startsymbol enthält (ε-Sonderregel, Informatik IV). ■

Beispiel. Der Aufbau von Anweisungen in einer (Pascal-ähnlichen) imperativen Sprache kann wie folgt spezifiziert werden. Der Lesbarkeit halber werden Terminalsymbole zwischen `" "` angegeben. Die Nichtterminalsymbole *Name*, *Bedingung* und *Ausdruck* sind nicht spezifiziert.

Anweisung	$\to$	*Wertzuweisung*
Anweisung	$\to$	*WhileAnweisung*
Anweisung	$\to$	*ProzedurAufruf*
Wertzuweisung	$\to$	*Name* `":="` *Ausdruck*
WhileAnweisung	$\to$	`"while"` *Bedingung* `"do"` *Anweisung*
ProzedurAufruf	$\to$	*Name* `"("` *AusdruckFolge* `")"`
AusdruckFolge	$\to$	*Ausdruck*
AusdruckFolge	$\to$	*AusdruckFolge* `","` *Ausdruck*

■

Beispiel.

$$\begin{array}{lcl}
\textit{Satz} & \rightarrow & \textit{Subjekt Prädikat} \\
\textit{Subjekt} & \rightarrow & \texttt{studenten} \\
\textit{Subjekt} & \rightarrow & \texttt{professoren} \\
\textit{Prädikat} & \rightarrow & \texttt{hören} \\
\textit{Prädikat} & \rightarrow & \texttt{lesen}
\end{array}$$

Hier fangen Variablen mit Großbuchstaben an, Terminalsymbole mit Kleinbuchstaben, und das Startsymbol ist *Satz*. Unter anderem definiert diese Grammatik die folgenden Sätze:

```
professoren lesen
studenten hören
```

Eine andere Schreibweise für die selbe Grammatik ist die folgende, die Alternativen in rechten Seiten von Produktionen zulässt.

$$\begin{array}{lcl}
\textit{Satz} & \rightarrow & \textit{Subjekt Prädikat} \\
\textit{Subjekt} & \rightarrow & \texttt{studenten} \mid \texttt{professoren} \\
\textit{Prädikat} & \rightarrow & \texttt{hören} \mid \texttt{lesen}
\end{array}$$

■

Definition (BNF). Eine kontextfreie Grammatik ist in *Backus-Naur-Form*, kurz BNF, wenn

- es höchstens eine Produktion für jedes Nichtterminalsymbol gibt und
- die rechten Seiten der Produktionen möglicherweise Alternativen enthalten. ■

Jede kontextfreie Grammatik kann in BNF umgewandelt werden.

Sei $G = (V, \Sigma, P, S)$ eine kontextfreie Grammatik. Eine Produktion $A \rightarrow w$ aus P mit $w \in (V \cup \Sigma)^*$ beschreibt, wie aus Wörtern, also Elementen von $(V \cup \Sigma)^*$), durch Ersetzen von A durch w neue Wörter „produziert" werden.

Betrachtet man die aus einem Nichtterminalsymbol „produzierbaren" Wörter, in denen keine weiteren Nichtterminalsymbole vorkommen, kann man jedes Nichtterminalsymbol als Repräsentant für eine Menge von Terminalwörtern auffassen. Ein Terminalsymbol ist ein Symbol, das in solchen Terminalwörtern auftreten kann. Terminalsymbole sind in der Praxis Zeichenfolgen wie etwa `":="` oder `"while"`, und nicht einzelne Zeichen.

Definition. Sei $G = (V, \Sigma, P, S)$ eine kontextfreie Grammatik. Seien $w, w' \in (V \cup \Sigma)^*$.

- *w produziert w'* gemäß G *unmittelbar* oder aus w ist w' gemäß G *unmittelbar ableitbar*, in Zeichen $w \Rightarrow_G w'$, wenn es Wörter u_1, u_2, u_3 und ein Nichtterminalsymbol A gibt, so dass $w = u_1 A u_2$ und $w' = u_1 u_3 u_2$ und $A \rightarrow u_3 \in P$ ist.
- *w produziert w'* gemäß G oder aus w ist w' gemäß G *ableitbar*, in Zeichen $w \Rightarrow_G^* w'$, wenn $w = w'$ ist oder $w \Rightarrow_G w'$ gilt oder es Wörter $v_1, \ldots, v_n$ mit $n \geq 1$ gibt, so dass $w \Rightarrow_G v_1$ und $v_1 \Rightarrow_G v_2$ und $\ldots$ und $v_{n-1} \Rightarrow_G v_n$ und $v_n \Rightarrow_G w'$ gilt. Die Folge w bzw. ww' bzw. $wv_1 \ldots v_n w'$ heißt dann eine *Ableitung* von w' aus w gemäß G.
  Die Relation $\Rightarrow_G^*$ ist also die reflexive und transitive Hülle von $\Rightarrow_G$.

Die durch G definierte *Sprache* $\mathcal{L}(G)$ ist $\{w \in \Sigma^* \mid S \Rightarrow_G^* w\}$.
Eine Sprache $L \subseteq \Sigma^*$ heißt kontextfrei, falls es eine kontextfreie Grammatik G gibt, für die $\mathcal{L}(G) = L$ ist. Eine Sprache $L \subseteq \Sigma^*$ heißt regulär, falls es eine reguläre Grammatik G gibt, für die $\mathcal{L}(G) = L$ ist. ∎

Reguläre Grammatiken reichen für Programmiersprachen nicht aus, weil Programmiersprachen üblicherweise klammerartige Konstrukte enthalten wie `"("`...`")"` oder `"{"`...`"}"` oder `"begin"`...`"end"`, die beliebig tief ineinander geschachtelt werden dürfen, was mit regulären Grammatiken nicht ausdrückbar ist. Aus der Theoretischen Informatik ist bekannt, dass $L = \{a^n b^n \mid n \in \mathbb{N}\}$ eine kontextfreie Sprache ist, die nicht regulär ist.

Definition (Rekursive Definition). Eine kontextfreie Grammatik G heißt *rekursiv*, wenn sie $n \geq 1$ Produktionen

$$\begin{array}{ll} A_1 \to w_1 & \text{in } w_1 \text{ kommt } A_2 \text{ vor} \\ A_2 \to w_2 & \text{in } w_2 \text{ kommt } A_3 \text{ vor} \\ \vdots & \\ A_n \to w_n & \text{in } w_n \text{ kommt } A_1 \text{ vor} \end{array}$$

enthält. ∎

Rekursive reguläre Grammatiken ermöglichen, beliebige Wiederholungen zu erzeugen.

Beispiel. Aus der Grammatik

$$\begin{array}{l} S \to bA \\ A \to d \mid aA \end{array}$$

werden die folgenden Wörter produziert: $bd, bad, baad, baaad, baaaaad, \ldots$ ∎

Definition (EBNF). Die BNF kann mit den folgenden Notationen erweitert werden:

- Faktorisierung $A \to u(w_1 \mid \ldots \mid w_n)$ steht für $A \to uw_1 \mid \ldots \mid uw_n$
- Option $[w]$ steht für $\varepsilon \mid w$
- Wiederholung $A \to u\{w\}v$ steht für $A \to uA'v \quad A' \to \varepsilon \mid wA'$
  $\{w\}$ steht also für die unendlich vielen Alternativen $\varepsilon \mid w \mid ww \mid www \mid wwww \mid \ldots$
  $\{w\}$ entspricht der Kleene-Stern-Notation $(w)^*$ in regulären Ausdrücken
- „bebecomes“ $A ::= w\ .$ steht für $A \to w$
  wobei in w alle oben angegebenen Notationen zulässig sind.

Diese Erweiterung der BNF wird kurz EBNF genannt. ∎

Bemerkungen.

1. Außer Terminalsymbolen und Nichtterminalsymbolen können in EBNF-Produktionen also die Symbole $::=\ \ .\ \ |\ \ (\ \ )\ \ [\ \ ]\ \ \{\ \ \}$ vorkommen. Diese sind Metasymbole der Sprache EBNF und keine Objektsymbole der durch die Grammatik definierten Objektsprache, in der ja nur Terminalsymbole der Grammatik vorkommen können.
2. Wenn keine expliziten ε-Produktionen vorkommen, kann das leere Wort trotzdem mit Produktionen der Gestalt $A \to [w]$ oder $A \to \{w\}$ ableitbar sein. ∎

2 Einführung in die syntaktische Analyse: Implementierung eines LL(1)-Parsers

Die Grundsymbole einer Programmiersprache bestehen in der Regel aus Zeichenfolgen statt aus einzelnen Zeichen, z. B. `BEGIN`, `END`, `if`, `package`, `>=` usw. Eine Komponente des Übersetzers, genannt Symbolentschlüssler (auch Scanner oder Lexer oder Tokenizer), wird eingesetzt, um aus einer Folge von Zeichen (character) die Symbole (token) der Programmiersprache zu erkennen, die aus einem oder mehreren Zeichen zusammengesetzt sind. Die entsprechende Phase der Übersetzung wird lexikalische Analyse genannt. Gegenüber der Terminologie der Theoretischen Informatik ergibt sich somit eine leichte Verschiebung:

Theoretische Informatik	Übersetzerbau
—	Zeichen (character)
Element von Σ: Zeichen oder Symbol	Symbol (token)
Element von Σ^*: Wort oder Satz	Satz, Programm oder Programmteil

Um den Aufbau von Symbolen aus Zeichen zu definieren, reichen reguläre Grammatiken aus, so dass die lexikalische Analyse im wesentlichen auf Implementierungen von endlichen Automaten beruht.

Aufgabe der syntaktischen Analyse ist die Erkennung der Struktur eines Satzes, also einer Folge von Symbolen (token). Diese Erkennung wird *Parsing* genannt. Die Algorithmen, die das Parsing durchführen, heißen *Parser*.

Das Parsing ist eine in der Regel komplizierte – manchmal sogar unmögliche – Aufgabe. Ihre Komplexität hängt von der Art der Produktionen ab, die die Syntax spezifizieren. Parser-Algorithmen sind für viele Klassen von Sprachen bekannt. Je allgemeiner sie anwendbar sind, desto weniger effizient sind sie.

Um den Aufbau von Programmen aus Symbolen zu definieren, sind reguläre Grammatiken normalerweise zu schwach. Eigentlich sind auch kontextfreie Grammatiken dafür zu schwach, weil Spracheigenschaften wie „Jede Variable, die in einem Ausdruck vorkommt, muss deklariert sein“ damit nicht ausgedrückt werden können. Da aber für Grammatiken vom Typ 1 (kontextsensitiv) oder gar Typ 0 keine effizienten Parser-Algorithmen existieren, beschränkt man die Definition der „Syntax“ einer Sprache auf das was mit kontextfreien Grammatiken ausdrückbar ist und behandelt alle nicht kontextfreien Spracheigenschaften in der sogenannten „semantischen“ Analyse, also in der nächsten Übersetzungsphase.

Diese eher pragmatisch begründete Abgrenzung der syntaktischen Analyse hat zur Folge, dass die syntaktische Analyse im Übersetzerbau sich nur mit kontextfreien Grammatiken beschäftigt und die Parser letztlich auf Varianten von Kellerautomaten beruhen.

2.1 LL(1)-Parser

In diesem Abschnitt wird ein Parser, LL(1)-Parser genannt, eingeführt, der relativ einfache Sprachen behandeln kann, die sogenannten LL(1)-Sprachen. Dank seiner Effizienz wird er in der Praxis häufig eingesetzt: Zur Erkennung eines Satzes braucht er eine Zeit, die höchstens proportional zur Länge des Satzes ist ($O(n)$ wobei n die Länge des Satzes/Programms ist, also die Anzahl der Symbole).

Wir betrachten in diesem Kapitel nur kontextfreie Sprachen. Der Einfachheit halber nehmen wir an, dass die kontextfreien Grammatiken in BNF sind. Für jedes Nichtterminalsymbol gibt es also höchstens eine Produktion, in der dieses Nichtterminalsymbol auf der linken Seite vorkommt.

Man kann die Klasse der LL(1)-Sprachen als Teilklasse der kontextfreien Sprachen formal definieren. Statt mit dieser Definition zu beginnen, werden wir im folgenden die Vorgehensweise des LL(1)-Parsers betrachten und erst anschließend bestimmen, welche Sprachen dieser Parser erkennt.

Die Prinzipien des LL(1)-Parsers können in vier Schlagwörtern zusammengefasst werden:

- **top-down:** Ausgehend vom Startsymbol der Grammatik werden die Produktionen „in Vorwärtsrichtung“ angewandt. Der Syntaxbaum des Satzes wird also von seiner Wurzel zu seinen Blättern hin aufgebaut.
- **left-to-right:** Die Eingabe wird ein Mal von links nach rechts durchlaufen. Das hat beim top-down-Parsing zur Folge, dass eine Linksableitung aufgebaut wird (d. h., Produktionen werden immer auf das am weitesten links stehende Nichtterminalsymbol angewandt).
- **one symbol lookahead:** Jeder Schritt beim Aufbau des Syntaxbaums hängt nur vom gegenwärtigen Knoten im Teil-Syntaxbaum und einem einzigen Symbol ab, nämlich dem nächsten zu lesenden Symbol.
- **no backtracking:** Kein Schritt darf rückgängig gemacht werden. Wenn zwischen mehreren anwendbaren Schritten entschieden werden muss, darf es also nicht passieren, dass die Entscheidung in eine „Sackgasse“ führen kann.

Beispiel 2.1 (Erfolgreiche Anwendung des LL(1)-Parsers).

$$
\begin{array}{lcl}
S & \rightarrow & AB \\
A & \rightarrow & a \mid c \\
B & \rightarrow & b \mid d
\end{array}
$$

Diese Grammatik definiert die Sprache, die aus den Sätzen ab, ad, cb und cd besteht. Bei Eingabe des Satzes cd geht ein LL(1)-Parser so vor:

Eingabe	Ableitung	anwendbar	Erklärung
cd	S	$S \rightarrow AB$	Ausgangspunkt
cd	AB	$A \rightarrow a$, $A \rightarrow c$	Nächstes Symbol der Eingabe erzeugbar? Ja, mit $A \rightarrow c$, also:
cd	cB	pop	Terminal-Anfang stimmt, noch abzuleiten:
d	B	$B \rightarrow c$, $B \rightarrow d$	Nächstes Symbol der Eingabe erzeugbar? Ja, mit $B \rightarrow d$, also:
d	d	pop	Terminal-Anfang stimmt, noch abzuleiten:
			Alles leer, fertig.

■

Die mit „Ableitung“ überschriebene Spalte kann man sich als Kellerspeicher vorstellen, der in diesem Fall nach links wächst. Wenn das oberste Element des Kellers ein Terminalsymbol ist, dann muss es das gleiche sein wie das nächste Symbol der Eingabe. In diesem Fall wird jeweils das oberste Symbol aus dem Keller entfernt („pop“) und das erste Symbol der Eingabe überlesen.

Die Vorgehensweise nach den oben genannten vier Prinzipien führt also in diesem Beispiel zum Erfolg. Der Erfolg ist aber nicht mit jeder Grammatik garantiert:

Beispiel 2.2 (LL(1)-Parser, Problem mit gemeinsamen Anfängen von Alternativen).

$$\begin{array}{lcl} S & \to & A \mid B \\ A & \to & cA \mid a \\ B & \to & cB \mid b \end{array}$$

Es gilt $S \Rightarrow B \Rightarrow cB \Rightarrow ccB \Rightarrow ccb$. Anwendung des LL(1)-Parsers auf ccb:

Eingabe	Ableitung	anwendbar	Erklärung
ccb	S	$S \to A$, $S \to B$	Nächstes Symbol der Eingabe erzeugbar? Ja, mit beiden Alternativen. Betrachtet man nur das Symbol c, sind beide gleichwertig. ($\star$) Entscheidung: $S \to A$
ccb	A	$A \to cA$, $A \to a$	Nächstes Symbol der Eingabe erzeugbar? Ja, mit $A \to cA$, also:
ccb	cA	pop	Terminal-Anfang stimmt, noch abzuleiten:
cb	A	$A \to cA$, $A \to a$	Nächstes Symbol der Eingabe erzeugbar? Ja, mit $A \to cA$, also:
cb	cA	pop	Terminal-Anfang stimmt, noch abzuleiten:
b	A	$A \to cA$, $A \to a$	Nächstes Symbol der Eingabe erzeugbar? Nein, Entscheidung ($\star$) war eine Sackgasse!

■

Um aus einer Sackgasse wie im obigen Beispiel wieder herauszukommen, müsste der LL(1)-Parser das Prinzip verletzen, keine Schritte rückgängig zu machen. Um die Sackgasse von vornherein zu vermeiden, müsste er das Prinzip verletzen, nur das jeweils nächste Symbol zu berücksichtigen.

Beispiel 2.3 (LL(1)-Parser, Problem mit Linksrekursion). Linksrekursive Produktionen haben die Gestalt $A \to \ldots |Aw| \ldots$ oder in EBNF die Gestalt $A \to [Aw]$ oder $A \to \{Aw\}$. Sie verhindern eine top-down Syntaxanalyse:

$$S \quad \to \quad a \mid Sb$$

Es gilt $S \Rightarrow Sb \Rightarrow Sbb \Rightarrow abb$. Anwendung des LL(1)-Parsers auf abb:

Eingabe	Ableitung	anwendbar	Erklärung
abb	S	$S \to a$, $S \to Sb$	Nächstes Symbol der Eingabe erzeugbar? Ja, mit beiden Alternativen. ($\star$) Entscheidung: $S \to a$
abb	a	pop	Terminal-Anfang stimmt, noch abzuleiten:
bb			Entscheidung ($\star$) war eine Sackgasse!
			Bei Wahl der anderen Alternative:
abb	S	$S \to a$, $S \to Sb$	($\star\star$) Entscheidung: $S \to Sb$
abb	Sb	$S \to a$, $S \to Sb$	Situation wie ($\star\star$) also gleiche Entscheidung:
abb	Sbb	$S \to a$, $S \to Sb$	Situation wie ($\star\star$) also gleiche Entscheidung:
abb	$Sbbb$	$S \to a$, $S \to Sb$	Situation wie ($\star\star$) also gleiche Entscheidung:
abb	$Sbbbb$	$S \to a$, $S \to Sb$	Situation wie ($\star\star$) also gleiche Entscheidung:
abb	$\ldots$	$\ldots$	

In solchen Fällen kann der LL(1)-Parser also sogar in eine Sackgasse laufen, die unendlich lang ist. Um diese Sackgasse zu vermeiden, müsste er alle Symbole bis zum Ende der Eingabe berücksichtigen. ■

Wir wollen die Grammatiken charakterisieren, bei denen eine Vorausschau über das erste Symbol hinaus nicht nötig ist.

Definition (LL(1)-Hilfsfunktion *first*). Sei $G = (V, \Sigma, P, S)$ eine Grammatik, $w \in (V \cup \Sigma)^*$.

$$\mathit{first}(w) := \{\, a \in \Sigma \mid \exists w' \;\; w \Rightarrow^* aw' \,\} \cup \begin{cases} \{\varepsilon\} & \text{falls } w \Rightarrow^* \varepsilon \\ \emptyset & \text{sonst} \end{cases}$$

Also ist $\mathit{first}(w)$ die Menge aller Terminalsymbole, mit denen eine Folge beginnen kann, die aus w ableitbar ist. Falls aus w auch das leere Wort ε ableitbar ist, wird es zu der Menge $\mathit{first}(w)$ hinzugenommen, obwohl es kein Terminalsymbol ist. ∎

Die folgende Bedingung muss von einer Grammatik erfüllt werden, damit der LL(1)-Parser angewandt werden kann:

Definition (LL(1)-Bedingung, Teil 1). Für jede Produktion $A \to w_1 \mid w_2 \mid \ldots \mid w_n$ muss gelten

$$\mathit{first}(w_i) \cap \mathit{first}(w_j) = \emptyset \text{ für alle } i, j \in \{1, \ldots, n\} \text{ mit } i \neq j.$$ ∎

Die Grammatik in Beispiel 2.2 enthält eine Produktion $S \to A \mid B$, für deren Alternativen gilt $c \in \mathit{first}(A)$ und $c \in \mathit{first}(B)$. Also ist die obige Bedingung verletzt. Die Bedingung erreicht somit das Ziel, diese Grammatik auszuschließen, die der LL(1)-Parser nicht bearbeiten kann.

Beispiel 2.4 (Notwendigkeit von ε in der Definition von *first*).

$$\begin{array}{lcl} S & \to & A \mid B \\ A & \to & \varepsilon \\ B & \to & \varepsilon \end{array}$$

Im ersten Schritt kann der LL(1)-Parser keine eindeutige Entscheidung zwischen den beiden Alternativen treffen. Deshalb sollte die Bedingung so sein, dass sie von dieser Grammatik verletzt wird. Nach Definition ist $\mathit{first}(A) = \{\varepsilon\}$ und $\mathit{first}(B) = \{\varepsilon\}$, so dass die Bedingung verletzt ist. Wäre *first* so definiert, dass ε nicht darin vorkommen könnte, wären in diesem Fall beide Mengen leer, also auch der Durchschnitt leer und damit die Bedingung erfüllt. ∎

Die obige Bedingung hat offensichtlich den Zweck, den Determinismus zu sichern, also das Prinzip „no backtracking“ zu ermöglichen. Wenn sich die *first*-Mengen von Alternativen überschneiden dürften, wäre aufgrund des nächsten Eingabesymbols allein nicht zu entscheiden, welche gewählt werden soll. Die Bedingung reicht aber für den Determinismus noch nicht aus:

Beispiel 2.5 (LL(1)-Parser, Problem mit ε-Produktion).

$$\begin{array}{lcl} S & \to & Aab \\ A & \to & a \mid \varepsilon \end{array}$$

Diese Grammatik erfüllt die LL(1)-Bedingung, Teil 1, weil $\mathit{first}(a) = \{a\}$ und $\mathit{first}(\varepsilon) = \{\varepsilon\}$ disjunkt sind. Es gilt $S \Rightarrow Aab \Rightarrow ab$. Anwendung des LL(1)-Parsers auf ab:

Eingabe	Ableitung	anwendbar	Erklärung
ab	S	$S \to Aab$	Ausgangspunkt
ab	Aab	$A \to a,\ A \to \varepsilon$	Nächstes Symbol der Eingabe erzeugbar? Ja, und zwar mit beiden(!) Alternativen. $(\star)$ Entscheidung: $A \to a$
ab	aab	pop	
b	ab		ungleiche Symbole, $(\star)$ war eine Sackgasse!

Das Problem liegt darin, dass $\mathit{first}(A)$ das Symbol a enthält, das aber auch unmittelbar auf A folgen kann, und dass A in das leere Wort übergehen kann. ■

Die restlichen Definitionen dienen dazu, diesen Fall auszuschließen.

Definition (LL(1)-Hilfsfunktion *follow*). Sei $G = (V, \Sigma, P, S)$ eine Grammatik und $A \in V$.

$$\mathit{follow}(A) = \{\, a \in \Sigma \mid \exists u, v \in (V \cup \Sigma)^* \quad S \Rightarrow^* uAav \,\}$$

Also ist $\mathit{follow}(A)$ die Menge aller Terminalsymbole, die in den aus S ableitbaren Folgen unmittelbar auf A folgen können. Man beachte, dass ε nicht in $\mathit{follow}(A)$ vorkommen kann. ■

Definition (LL(1)-Bedingung, Teil 2). Für jedes Nichtterminalsymbol $A \in V$ mit $A \Rightarrow^* \varepsilon$ muss gelten

$$\mathit{first}(A) \cap \mathit{follow}(A) = \emptyset$$

■

Definition (LL(1)-Grammatik, LL(1)-Sprache. Eine kontextfreie Grammatik, die die LL(1)-Bedingungen Teil 1 und Teil 2 erfüllt, heißt LL(1)-Grammatik. Eine Sprache, die von einer LL(1)-Grammatik erzeugt wird, heißt LL(1)-Sprache. ■

Die Bezeichnung „LL(1)“ bedeutet: Eingabe von Links nach rechts durchlaufen, wobei eine Linksableitung aufgebaut wird und um jeweils 1 einziges Symbol vorausgeschaut werden darf, um Entscheidungen zu treffen.

Wir haben bereits gesehen, dass linksrekursive Produktionen eine top-down Syntaxanalyse verhindern. Muss also nicht eine 3. Bedingung hinzukommen, die linksrekursive Produktionen verbietet? Man kann zeigen (Übung!), dass unter wenigen, natürlichen Zusatzannahmen bereits durch die LL(1)-Bedingungen Teil 1 und Teil 2 ausgeschlossen wird, dass linksrekursive Produktionen vorkommen können. Ferner gilt:

Satz. Für jede Sprache, die durch eine LL(1)-Grammatik definiert ist, und für jede Symbolfolge erkennt der LL(1)-Parser, ob die Symbolfolge zur Sprache gehört oder nicht. ■

Es gibt viele Parser, die anders vorgehen als der LL(1)-Parser. Zunächst kann die Vorausschau um ein Symbol verallgemeinert werden zu einer Vorausschau um k Symbole für eine natürliche Zahl k. Man spricht dann von LL(k)-Grammatiken und LL(k)-Parser. Andere Parser bauen den Syntaxbaum nicht von der Wurzel zu den Blättern hin auf (top-down), sondern von den Blättern zur Wurzel (bottom-up). Sie durchlaufen die Eingabe auch von links nach rechts, aber dabei wird eine Rechtsableitung aufgebaut. Das sind sogenannte LR(1)-Parser oder allgemeiner LR(k)-Parser. Zusätzlich gibt es auch allgemeine Parser, die auf jegliche kontextfreie Grammatiken anwendbar sind, z. B. die Earley- und Cocke-Younger-Kasami-Algorithmen. Sie werden aber zur Übersetzung von Programmiersprachen aus Effizienzgründen kaum eingesetzt.

Bemerkung. Wenn eine Grammatik die LL(1)-Bedingung nicht erfüllt, kann man oft eine äquivalente LL(1)-Grammatik konstruieren, die die selbe Sprache erzeugt. Es wäre aber falsch, zu denken, dass dies immer ohne Folgen für die Sprache wäre. Betrachten wir die Sprache der Differenzen über Ziffern:

$$\begin{array}{lll} \textit{Differenz} & \rightarrow & \textit{Ziffer} \;\mid\; \textit{Differenz}\; \texttt{"-"}\; \textit{Ziffer} \\ \textit{Ziffer} & \rightarrow & \texttt{"0"} \mid \texttt{"1"} \mid \dots \mid \texttt{"9"} \end{array}$$

Die erste Produktion ist linksrekursiv, so dass der LL(1)-Parser nicht anwendbar ist. Mit einer anschaulichen Transformation erhalten wir eine LL(1)-Grammatik für die selbe Sprache:

$$\begin{array}{lll} \textit{Differenz} & \rightarrow & \textit{Ziffer}\; \textit{RestDifferenz} \\ \textit{RestDifferenz} & \rightarrow & \varepsilon \;\mid\; \texttt{"-"}\; \textit{Differenz} \\ \textit{Ziffer} & \rightarrow & \texttt{"0"} \mid \texttt{"1"} \mid \dots \mid \texttt{"9"} \end{array}$$

Mit beiden Grammatiken gilt $\textit{Differenz} \Rightarrow^* \texttt{8 - 3 - 2}$. Der Syntaxbaum gemäß der ersten Grammatik entspricht der Klammerung $(8 - 3) - 2$, der Syntaxbaum gemäß der zweiten Grammatik entspricht der Klammerung $8 - (3 - 2)$, so dass bei der Auswertung der Ausdrücke unterschiedliche Ergebnisse entstehen.

Wäre nur das Wortproblem zu lösen, wären die beiden Grammatiken gleichwertig. Aber in der Syntaxanalyse müssen ja auch die Syntaxbäume aufgebaut werden. ■

2.2 Implementierung eines LL(1)-Parsers

In diesem Abschnitt werden wir die Methode des „rekursiven Abstiegs“ kennenlernen, mit der ein LL(1)-Parser in einfacher und systematischer Weise implementiert werden kann. Dazu betrachten wir zunächst eine weitere Notation für die Produktionen von kontextfreien Grammatiken.

2.2.1 Syntaxgraphen

Die bereits eingeführte EBNF erlaubt folgende Konstrukte in kontextfreien Grammatiken.

0. Produktion: $A ::= w$.
   Jedes Nichtterminalsymbol kommt in höchstens einer Produktion links vor.

Die rechte Seite w kann mit folgenden Konstrukten gebildet werden, wobei $w_1, w_2, \dots, w_n$ wiederum mit diesen Konstrukten gebildet werden können:

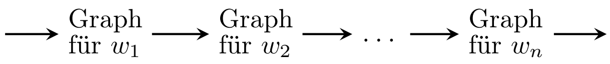
1. Sequenz: $w_1 w_2 \dots w_n$
2. Alternativen: $w_1 \mid w_2 \mid \dots \mid w_n$
3. Option: $[\, w_1 \,]$
4. Wiederholung: $\{\, w_1 \,\}$
5. Nichtterminalsymbol: A
6. Terminalsymbol: `a`

Ein *Syntaxgraph* (auch *Syntaxdiagramm* genannt) ist ein Graph mit zwei Arten von Knoten, der eine Eingangskante und eine Ausgangskante besitzt. Die Eingangskante ist die einzige beschriftete Kante. Die Kanten sind gerichtet und können jeweils auch mehrere Knoten miteinander verbinden. In der Sprechweise der Graphentheorie handelt es sich also um sogenannte Hypergraphen.

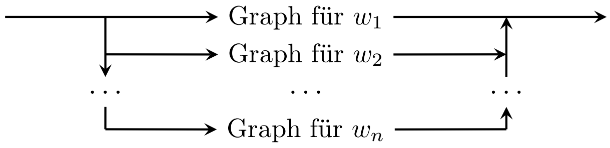
Die folgenden Übersetzungsregeln geben an, wie die EBNF-Notation in die Notation der Syntaxgraphen übersetzt wird.

Übersetzungsregel 2.2.1-0 (Produktion).
Graph für $A ::= w$. ist

Übersetzungsregel 2.2.1-1 (Sequenz).
Graph für $w_1 w_2 \dots w_n$ ist



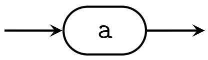
Übersetzungsregel 2.2.1-2 (Alternativen).
Graph für $w_1 \mid w_2 \mid \dots \mid w_n$ ist



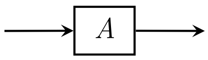
Übersetzungsregel 2.2.1-3 (Option).
Graph für $[\, w_1 \,]$ ist

Übersetzungsregel 2.2.1-4 (Wiederholung).
Graph für $\{\, w_1 \,\}$ ist

Übersetzungsregel 2.2.1-6 (Terminalsymbol)
Graph für `a` ist



Übersetzungsregel 2.2.1-5 (Nichtterminalsymbol).
Graph für A ist



Zu jeder Produktion wird also ein Syntaxgraph erzeugt, dessen Eingangskante mit der linken Seite der Produktion beschriftet ist. Jede aus diesem Nichtterminalsymbol ableitbare Folge von Terminalsymbolen entspricht einer „endlichen Reise“ durch den zugehörigen Syntaxgraphen von der Eingangskante zur Ausgangskante. Die bei dieser „Reise“ durchlaufenen Terminalsymbole werden einfach aufgesammelt. Jedes durchlaufene Nichtterminalsymbol entspricht einem „Aufruf“ des Syntaxgraphen für dieses Nichtterminalsymbol. Nach einer vollständigen „Reise“ durch den aufgerufenen Syntaxgraphen wird die „Reise“ an der Aufrufstelle fortgesetzt.

Wegen dieser anschaulichen „Reise“-Metapher werden Syntaxgraphen gelegentlich auch als „Eisenbahndiagramme“ bezeichnet.

Beispiel (vereinfachte Klassendeklaration in Java). Die Syntax von Klassendeklarationen in der Programmiersprache Java ist durch folgende Grammatik spezifiziert. Terminalsymbole sind hier zwischen `" "` angegeben.

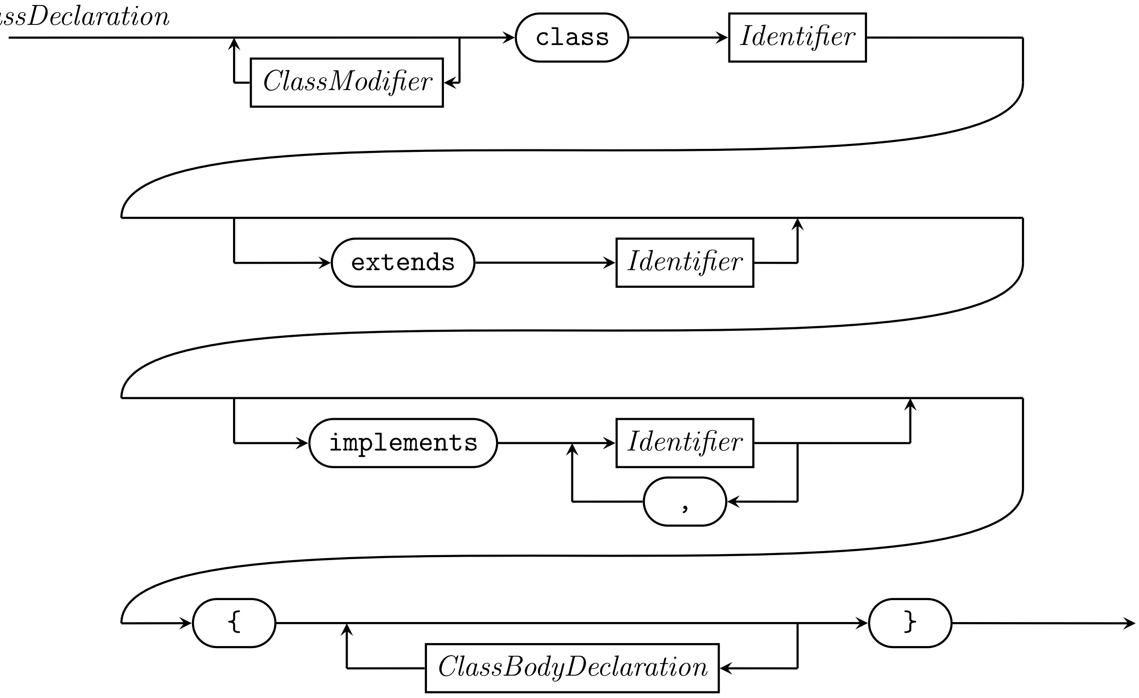
ClassDeclaration ::= { *ClassModifier* } `"class"` *Identifier*
[`"extends"` *Identifier*]
[`"implements"` *Identifier* { `","` *Identifier* }]
`"{"` { *ClassBodyDeclaration* } `"}"` .

ClassModifier ::= `"public"` | `"protected"` | `"private"`
| `"abstract"` | `"static"` | `"final"` | `"strictfp"` .

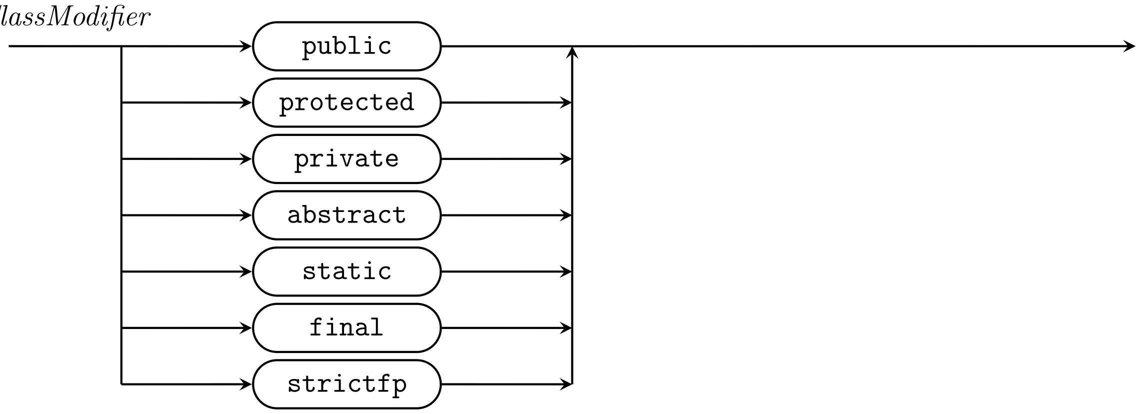
ClassBodyDeclaration ::= ...

Durch Anwendung der obigen Übersetzungsregeln entstehen daraus folgende Syntaxgraphen:

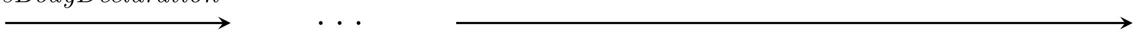
ClassDeclaration



ClassModifier



ClassBodyDeclaration



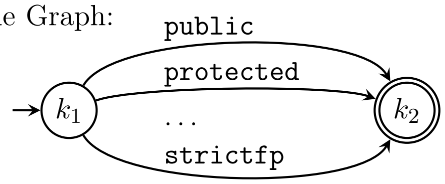
Unter der Annahme dass `name1` und `name2` und `name3` aus *Identifier* ableitbar sind, sind gemäß dieser Syntaxgraphen unter anderem folgende Folgen von Terminalsymbolen syntaktisch zulässige Klassendeklarationen:

```
class name1 { }
private class name1 extends name2 { }
public abstract class name1 implements name2 , name3 { }
```
■

Übung. Arithmetische Ausdrücke bestehend aus Zahlen, arithmetischen Infix-Operatoren mit den üblichen Präzedenzregeln und Klammern mittels Syntaxgraphen spezifizieren. ■

Rein äußerlich erinnern Syntaxgraphen an die graphischen Darstellungen verschiedener Automatenmodelle der Theoretischen Informatik. Der Zusammenhang kann mit Hilfe eines Begriffs aus der Graphentheorie beschrieben werden. Der *duale Graph* zu einem gegebenen Graph wird gebildet, indem man jede Kante als Knoten darstellt und umgekehrt. Der duale Graph des Syntaxgraphen für *ClassModifier*, in dem wir die Eingangskante k_1 und die Ausgangskante k_2 nennen, ist somit der folgende Graph:

ClassModifier



Markiert man, wie hier geschehen, den Knoten für die Eingangskante des Syntaxgraphen mit einem kleinen Pfeil und den Knoten für die Ausgangskante des Syntaxgraphen durch doppelte Umrandung, entsteht das Zustandsübergangsdiagramm eines endlichen Automaten. Ein Syntaxgraph, der nur Terminalknoten enthält, ist also einfach ein endlicher Automat in dualer Darstellung.

Der duale Graph eines Syntaxgraphen, in dem auch Nichtterminalknoten vorkommen, entspricht dem Zustandsübergangsdiagramm eines neuen Automatentyps. Es enthält auch Kanten, die mit einem Nichtterminalsymbol beschriftet sind. Eine solche Kante repräsentiert einen Aufruf des Automaten für dieses Nichtterminalsymbol. Das gedachte Ablaufmodell des neuen Automatentyps muss dafür sorgen, dass bei einem solchen Aufruf der „Rücksprungzustand" abgespeichert wird, also der Zustand im aufrufenden Automaten, mit dem die Bearbeitung fortgesetzt wird, wenn der aufgerufene Automat fertig ist. Da der aufgerufene Automat seinerseits Automaten aufrufen kann, ist für die Verwaltung der „Rücksprungzustände" ein Kellerspeicher erforderlich. Dadurch geht der neue Automatentyp über endliche Automaten hinaus und erreicht die Mächtigkeit von Kellerautomaten.

In einem Syntaxgraphen kann man an Stelle eines Nichtterminalknotens einfach den Syntaxgraphen für dieses Nichtterminalsymbol einsetzen. Wenn man diese Einsetzung in einer Menge von Syntaxgraphen zum Beispiel für jeden *ClassModifier*-Knoten durchführt, wird der Syntaxgraph für *ClassModifier* nirgends mehr benutzt und damit überflüssig. Bei manchen Mengen von Syntaxgraphen kann man auf diese Weise alle Nichtterminalknoten beseitigen, so dass nur noch Terminalknoten vorkommen und das verbleibende Syntaxdiagramm einem endlichen Automaten entspricht.

Das kann natürlich nicht für jede Menge von Syntaxgraphen möglich sein. Nicht möglich ist es bei direkter oder indirekter Rekursion. Zum Beispiel kann man im obigen Syntaxgraph für *ClassDeclaration* den *ClassBodyDeclaration*-Knoten beseitigen, indem man den entsprechenden Syntaxgraphen dafür einsetzt. Dieser enthält einen *ClassDeclaration*-Knoten, weil im

Rumpf einer Java-Klasse wiederum Klassen deklariert werden können. Versucht man jetzt, diesen *ClassDeclaration*-Knoten zu beseitigen, indem man das Syntaxdiagramm dafür einsetzt, kommt beim Einsetzen ein *ClassDeclaration*-Knoten hinzu, der auch noch zu beseitigen wäre. Man kann also nicht sämtliche Nichtterminalknoten durch Einsetzen beseitigen.

Interpretation der LL(1)-Bedingung in Syntaxgraphen. Teil 1 der LL(1)-Bedingung bedeutet, dass bei Verzweigungen aufgrund des nächsten Eingabesymbols eindeutig bestimmbar sein muss, mit welchem Zweig die „Reise" fortgesetzt wird. Teil 2 der LL(1)-Bedingung bedeutet, dass bei Syntaxgraphen, die man durchlaufen kann, ohne einen Terminalknoten zu berühren, eindeutig sein muss, ob das nächste Eingabesymbol zu diesem Syntaxgraphen gehört oder zum nächsten. Diese beiden Bedingungen sind für gegebene Syntaxgraphen leicht überprüfbar.

2.2.2 Konstruktion eines LL(1)-Parsers aus Syntaxgraphen

Die am einfachsten zu implementierende Parsing-Methode trägt den Namen „rekursiver Abstieg (recursive descent)". Sie geht von einer Menge von Syntaxgraphen aus, die die LL(1)-Bedingung erfüllt.

Die Grundidee dabei ist, jeden Syntaxgraphen in eine parameterlose Funktion zu übersetzen, die die möglichen Reisen durch diesen Syntaxgraphen implementiert und somit Programmstücke erkennen kann, die aus dem Nichtterminalsymbol an der Eingangskante ableitbar sind. Diese Funktionen werden in folgendes Rahmenprogramm eingebettet:

```
type symbol = ende | ··· ;
symbol sym;

symbol nextSymbol() {
  // liefert bei jedem Aufruf das jeweils nächste Symbol
}

void error(int nummer) {
  // Erzeugung einer Fehlermeldung und Abbruch
}

⟨Code für die Syntaxgraphen, für jeden eine Funktion⟩

void main() {
  sym := nextSymbol();
  S();                          // Startsymbol S
  if (sym != ende) error(0);
}
```

Dabei sei `symbol` ein Aufzählungstyp (enumeration type), der für jedes Terminalsymbol einen Wert zur Verfügung stellt sowie einen zusätzlichen Wert `ende`. Die Variable `sym` vom Typ `symbol` ist global für die zu definierenden Funktionen und wird von diesen gelesen und geändert.

Die Funktion `nextSymbol()` ist die Schnittstelle zur lexikalischen Analyse. Wiederholte Aufrufe dieser Funktion entsprechen einem Durchlauf durch die Folge der Symbole von links nach rechts. Wird die Funktion aufgerufen, wenn kein Eingabezeichen mehr verfügbar ist, liefert sie das Symbol `ende`.

Die Fehlerfunktion realisiert eine primitive Form der Fehlerbehandlung.

Das Hauptprogramm initialisiert die Variable `sym` und ruft die Funktion für das Startsymbol der Grammatik auf. Nach diesem Aufruf muss die gesamte Eingabe durchlaufen worden sein.

Der Code für die Syntaxgraphen wird durch folgende Übersetzungsregeln rekursiv erzeugt, wobei $G, G_1, G_2, \ldots, G_n$ für Teilgraphen stehen. Die Nummerierung entspricht der Nummerierung der Übersetzungsregeln im vorigen Abschnitt.

Übersetzungsregel 2.2.2-0 (Produktion).

Code für $\xrightarrow{A} G \longrightarrow$ ist

```
void A() {
  ⟨Code für Graph G⟩
}
```

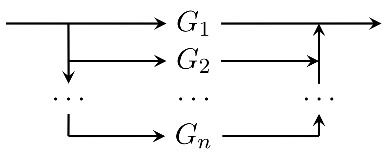
Übersetzungsregel 2.2.2-1 (Sequenz).

Code für $\longrightarrow G_1 \longrightarrow G_2 \rightarrow \ldots \rightarrow G_n \longrightarrow$ ist

```
⟨Code für Graph G1⟩;
⟨Code für Graph G2⟩;
  ...
⟨Code für Graph Gn⟩;
```

Übersetzungsregel 2.2.2-2 (Alternativen).

Code für

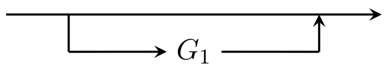


ist

```
case
  (sym ∈ ⟨Firstmenge von Graph G1⟩)
           ⟨Code für Graph G1⟩;
  (sym ∈ ⟨Firstmenge von Graph G2⟩)
           ⟨Code für Graph G2⟩;
  ...
  (sym ∈ ⟨Firstmenge von Graph Gn⟩)
           ⟨Code für Graph Gn⟩;
else error(...)
```

Übersetzungsregel 2.2.2-3 (Option).

Code für



ist

```
if (sym ∈ ⟨Firstmenge von Graph G1⟩) {
  ⟨Code für Graph G1⟩
}
```

Übersetzungsregel 2.2.2-4 (Wiederholung).

Code für

ist

```
while (sym ∈ ⟨Firstmenge von Graph G1⟩) {
  ⟨Code für Graph G1⟩
}
```

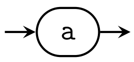
Übersetzungsregel 2.2.2-5 (Nichtterminalsymbol).

Code für $\rightarrow \boxed{A} \rightarrow$ ist

```
A()
```

Übersetzungsregel 2.2.2-6 (Terminalsymbol)

Code für



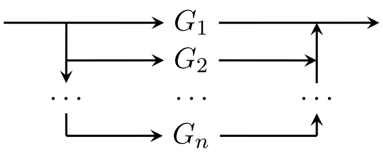
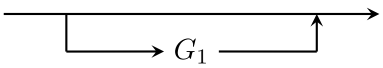
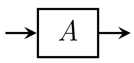
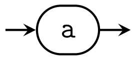
ist

```
if (sym==a) sym:=nextSymbol()
            else error(...)
```

Dazu muss jetzt noch definiert werden, was die Firstmengen von Syntaxgraphen sind. Diese müssen bei der Erzeugung des Codes an den Stellen ⟨Firstmenge von Graph …⟩ eingesetzt werden und können nicht etwa durch einen Funktionsaufruf zur Laufzeit des Parsers berechnet werden. Eine solche Funktion bräuchte ja einen Teilgraph eines Syntaxgraphen als Parameter, also keinen Wert eines Typs, der im Parsing-Programm zur Verfügung stünde.

Wir übertragen einfach die Definition der Hilfsfunktion *first* auf Syntaxgraphen (also eine mathematische Funktion, keine Funktion im Sinn eines Unterprogramms der Implementierungssprache).

Definition (Firstmenge von Syntaxgraphen).

Syntaxgraph G	Firstmenge von Graph G, notiert $\mathit{first}(G) =$
$\rightarrow G_1 \rightarrow G_2 \rightarrow \ldots \rightarrow G_n \rightarrow$	falls eine der folgenden Mengen ε nicht enthält, dann die erste die ε nicht enthält: $\mathit{first}(G_1)$ $\Big(\mathit{first}(G_1) \setminus \{\varepsilon\}\Big) \cup \mathit{first}(G_2)$ … $\Big((\mathit{first}(G_1) \cup \ldots \cup \mathit{first}(G_{n-2})) \setminus \{\varepsilon\}\Big) \cup \mathit{first}(G_{n-1})$ falls jede der obigen Mengen ε enthält, dann: $\Big((\mathit{first}(G_1) \cup \ldots \cup \mathit{first}(G_{n-1})) \setminus \{\varepsilon\}\Big) \cup \mathit{first}(G_n)$
	$\mathit{first}(G_1) \cup \mathit{first}(G_2) \cup \ldots \cup \mathit{first}(G_n)$
	$\{\varepsilon\} \cup \mathit{first}(G_1)$
	$\{\varepsilon\} \cup \mathit{first}(G_1)$
	$\mathit{first}(G_A)$ für den Syntaxgraphen G_A, dessen Eingangskante mit A beschriftet ist.
	{a}

∎

Die Übersetzungsregeln für Terminalsymbole und Nichtterminalsymbole sind die Basisfälle der rekursiven Übersetzung von Syntaxgraphen in Code für den Parser. Es ist bemerkenswert, dass die übrigen Übersetzungsregeln den möglichen Strukturen von Syntaxgraphen (und damit letztlich den EBNF-Konstrukten) genau die typischen Konstrukte zur Ablaufsteuerung in imperativen Programmiersprachen zuordnen: Sequenzialisierung, Fallunterscheidung, einseitige Bedingung, Iteration, Unterprogramme.

Angewandt auf die Syntaxgraphen für Klassendeklarationen in Java ergeben die Übersetzungsregeln folgenden Code:

Beispiel (Parser-Funktionen für Klassendeklaration in Java).

```
void ClassModifier() {
  case
    (sym ∈ {public}) {
      if (sym==public) sym:=nextSymbol() else error(...);
    }
    (sym ∈ {protected}) {
      if (sym==protected) sym:=nextSymbol() else error(...);
    }
    ...
    (sym ∈ {strictfp}) {
      if (sym==strictfp) sym:=nextSymbol() else error(...);
    }
  else error(...);
}
```

```
void ClassDeclaration() {
  while (sym ∈ {public,protected,...,strictfp}) {
    ClassModifier();
  }
  if (sym==class) sym:=nextSymbol() else error(...);
  Identifier();
  if (sym ∈ {extends}) {
    if (sym==extends) sym:=nextSymbol() else error(...);
    Identifier();
  }
  ...
}
```

■

Offensichtlich kann der durch Anwendung der Übersetzungsregeln entstehende Code noch vereinfacht werden. In der Funktion `ClassModifier` können alle `if`-Anweisungen durch ihren `then`-Teil ersetzt werden, weil der Gleichheitstest jeweils in einem Fall des `case` steht, in dem der Gleichheitstest nie falsch sein kann. Das ist aber nicht immer so. Zum Beispiel ist der Test `(sym==class)` in der Funktion `ClassDeclaration` notwendig, um einen Fehler melden zu können, wenn das Symbol `class` fehlt.

In der bisherigen Form entsteht durch die Übersetzungsregeln ein Parser, der nur das Wortproblem löst. Wenn die Eingabe gemäß der Grammatik nicht korrekt ist, bricht der Parser mit einer Fehlermeldung ab. Wenn die Eingabe korrekt ist, durchläuft der Parser die gesamte Eingabe ohne einen Fehler zu melden, aber ohne ein sonstiges Ergebnis zu liefern.

Damit der Parser im Erfolgsfall auch den Syntaxbaum liefert, müssen die Übersetzungsregeln um sogenannte *Strukturanknüpfungen* erweitert werden. Das sind Code-Fragmente, die beim Parsing Datenstrukturen aufbauen. Die einfachste Variante besteht darin, den Parser-Funktionen einen Ergebnistyp `knoten` zu geben und jede Funktion einen neuen Knoten zurückliefern zu lassen, dessen Kinder die Ergebnisse der aufgerufenen Funktionen sind. Auf diese Weise entsteht ein Baum von Knoten, der die Aufrufstruktur der Parser-Funktionen repräsentiert, und diese entspricht ja nach Konstruktion genau dem Syntaxbaum.

2.2.3 Ein LL(1)-Parser-Generator

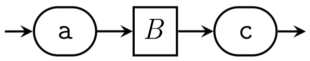
Mit den Ergebnissen des vorigen Abschnitts kann aus Syntaxgraphen automatisch ein LL(1)-Parser gewonnen werden. Dazu fehlt nur noch eine Datenstruktur zur Repräsentation von Syntaxgraphen. Der LL(1)-Parser-Generator ist dann einfach eine Implementierung der Übersetzungsregeln aus dem vorigen Abschnitt. Eingabe des LL(1)-Parser-Generators ist eine Datenstruktur, die eine Menge von Syntaxgraphen repräsentiert, Ausgabe ist der Programmcode des LL(1)-Parsers.

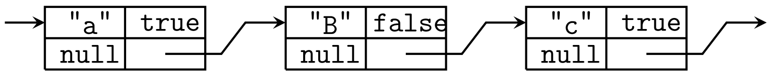
Eine einfache Datenstruktur für Syntaxgraphen basiert auf einem Verbundtyp `knoten` mit vier Komponenten (der Typ `knoten*` bezeichne dabei den Verweistyp, dessen Werte Verweise (pointer) auf Werte des Typs `knoten` sind):

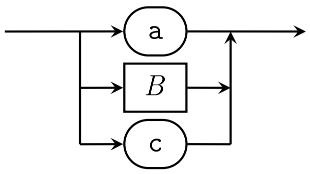
`string beschriftung`	`boolean terminal`
`knoten* alternative`	`knoten* nachfolger`

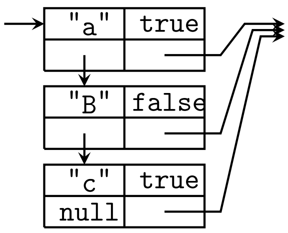
Jeder Knoten eines Syntaxgraphen wird durch einen solchen Verbund repräsentiert. Die Komponente `terminal` gibt an, ob der Knoten ein Oval (Terminalknoten) oder ein Rechteck (Nichtterminalknoten) ist, die Komponente `beschriftung` ist selbsterklärend. Die Komponente `nachfolger` repräsentiert die von diesem Knoten ausgehende Kante, zeigt also auf den nächsten Knoten. Die Komponente `alternative` zeigt auf Alternativen des Knoten im Syntaxgraphen, sofern vorhanden.

Die folgenden Beispiele sollen zeigen, wie mit dieser Datenstruktur Syntaxgraphen repräsentiert werden können.

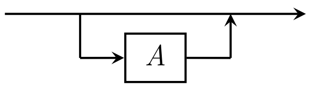


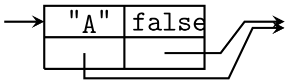




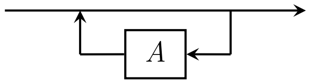


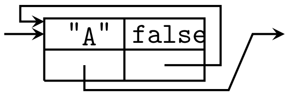
Entweder den Knoten mit Beschriftung `"a"` durchlaufen und dann fortsetzen, oder alternativ dazu den Knoten mit Beschriftung `"B"` durchlaufen und dann fortsetzen, oder alternativ dazu den Knoten mit Beschriftung `"c"` durchlaufen und dann fortsetzen.





Entweder den Knoten mit Beschriftung `"A"` durchlaufen und dann fortsetzen, oder alternativ dazu gleich fortsetzen.





Entweder den Knoten mit Beschriftung `"A"` durchlaufen und dann wieder entscheiden, oder alternativ dazu gleich fortsetzen.

Für diese Datenstrukturen kann man die Übersetzungsregeln aus dem vorigen Abschnitt, ergänzt um geeignete Strukturanknüpfungen, programmieren.

2.3 Bootstrapping des LL(1)-Parsers und Ansatz des „stepwise refinement“

Der Parser-Generator aus dem letzten Abschnitt hat als Eingabe eine Datenstruktur, mit der eine Menge von Syntaxgraphen repräsentiert wird, die die LL(1)-Bedingung erfüllt. Das ist keine passende Eingabe für ein Programm, da die Datenstruktur ja nicht direkt eingetippt werden kann, sondern durch Konstrukte der Implementierungssprache erzeugt werden muss.

Die bisherigen Hilfsmittel reichen aber aus, um eine bequemere Lösung zu erreichen. Man muss sie nur mit einer Technik der Metaprogrammierung zusammensetzen.

Zunächst kann man feststellen, dass die erweiterte Backus-Naur-Form (EBNF) selbst eine formale Sprache ist. Sie kann durch folgende Meta-Grammatik definiert werden.

Meta-Grammatik für EBNF.

$$
\begin{array}{lcl}
\mathit{GRAMMATIK} & ::= & \mathit{PRODUKTION}\ \{\mathit{PRODUKTION}\}\ . \\
\mathit{PRODUKTION} & ::= & \mathit{NT}\ \texttt{"::="}\ \mathit{AUSDRUCK}\ \texttt{"."}\ . \\
\mathit{AUSDRUCK} & ::= & \mathit{TERM}\ \{\texttt{"|"}\ \mathit{TERM}\}\ . \\
\mathit{TERM} & ::= & \mathit{FAKTOR}\ \{\mathit{FAKTOR}\}\ . \\
\mathit{FAKTOR} & ::= & \mathit{NT} \\
 & | & T \\
 & | & \texttt{"("}\ \mathit{AUSDRUCK}\ \texttt{")"} \\
 & | & \texttt{"["}\ \mathit{AUSDRUCK}\ \texttt{"]"} \\
 & | & \texttt{"\{"}\ \mathit{AUSDRUCK}\ \texttt{"\}"}\ . \\
\mathit{NT} & ::= & \ldots \\
T & ::= & \ldots
\end{array}
$$

An den Stellen mit den drei Punkten stehen jeweils Alternativen von *Terminalsymbolen* der Meta-Grammatik. Die Meta-Grammatik-Terminalsymbole auf der rechten Seite von NT sind in der erzeugten Grammatik Nichtterminalsymbole, die Meta-Grammatik-Terminalsymbole auf der rechten Seite von T sind auch in der erzeugten Grammatik Terminalsymbole. ■

Man beachte den Unterschied zwischen $::=$ und `"::="` in der Meta-Produktion für *PRODUKTION*, zwischen $|$ auf der rechten Seite von *FAKTOR* und `"|"` auf der rechten Seite von *AUSDRUCK*, zwischen $\{$ auf der rechten Seite von *TERM* und `"{"` auf der rechten Seite von *FAKTOR*, sowie die analogen Unterschiede bei einigen weiteren Symbolen.

Übung. Man überprüfe, dass unter natürlichen Annahmen über die Stellen mit den drei Punkten die Meta-Grammatik eine LL(1)-Grammatik ist. ■

In einem ersten Schritt wandelt ein Mensch die Meta-Grammatik mit den Übersetzungsregeln aus Abschnitt 2.2.1 in eine Menge von Syntaxgraphen um, und baut dann für diese Syntaxgraphen die Datenstrukturen gemäß Abschnitt 2.2.3 auf. Durch diesen Schritt entsteht eine Datenstruktur-Repräsentation der Meta-Grammatik, die der LL(1)-Parser-Generator gemäß Abschnitt 2.2.3 als Eingabe verarbeiten kann – die Meta-Grammatik erfüllt ja die LL(1)-Bedingung.

Aus dieser Eingabe erzeugt der LL(1)-Parser-Generator den Programmcode eines LL(1)-Parsers. Mit der Standardvariante des LL(1)-Parser-Generators enthält der generierte Programmcode Strukturanknüpfungen zum Aufbau eines Syntaxbaums. Der generierte LL(1)-Parser kann also Grammatiken aus der Sprache EBNF in ihre Syntaxbäume gemäß der Meta-Grammatik übersetzen.

Jetzt wird eine zweite Variante des LL(1)-Parser-Generators erstellt, die sich lediglich darin unterscheidet, dass sie andere Strukturanknüpfungen erzeugt. Diese bauen zu einer Grammatik nicht den Syntaxbaum gemäß der Meta-Grammatik auf, sondern die Datenstruktur gemäß Abschnitt 2.2.3, mit der Syntaxgraphen repräsentiert werden. Tasächlich ist diese Datenstruktur gar nicht so sehr verschieden von einer Datenstruktur, die den Syntaxbaum der Grammatik repräsentieren würde.

Diese zweite Variante des LL(1)-Parser-Generators wird dann auf die manuell erzeugte Datenstruktur-Repräsentation der Meta-Grammatik angewandt. Ergebnis des ersten Schritts ist somit der Programmcode eines LL(1)-Parsers, der Grammatiken aus der Sprache EBNF in die Datenstruktur-Repräsentation ihrer Syntaxgraphen übersetzen kann.

Im zweiten Schritt ist eine LL(1)-Grammatik G gegeben, die in EBNF geschrieben ist. Sie wird mit dem LL(1)-Parser aus dem ersten Schritt in die Datenstruktur-Repräsentation ihrer Syntaxgraphen übersetzt, die dann als Eingabe für den LL(1)-Parser-Generator aus Abschnitt 2.2.3 dienen, und zwar dieses Mal in der Standardvariante.

Ergebnis des zweiten Schritts ist also der Programmcode eines LL(1)-Parsers, der Wörter aus $\mathcal{L}(G)$ in ihre Syntaxbäume gemäß G übersetzen kann.

Schrittweise Verfeinerung (stepwise refinement). Der in diesem Kapitel beschriebene Ansatz ist unter anderem dadurch charakterisiert, dass er einmal entwickelte Hilfsmittel allmählich erweitert, um sie wiederverwenden zu können. Ausgangspunkt war ein Parser, der nur das Wortproblem lösen kann, indem er mit einer Fehlermeldung abbricht oder die gesamte Eingabe fehlerfrei durchläuft. Dieser wurde um Strukturanknüpfungen erweitert, damit er Syntaxbäume erzeugen kann. Von diesen Strukturanknüpfungen wurden Varianten entwickelt, damit er Datenstruktur-Repräsentationen von Syntaxgraphen erzeugen kann, die in Verbindung mit der Technik der Metaprogrammierung das Bootstrapping des Parsers ermöglichen.

Im obigen zweiten Schritt wurde mit der Standardvariante des LL(1)-Parser-Generators gearbeitet. Man kann statt dessen Varianten mit anderen Strukturanknüpfungen entwickeln, die keine Syntaxbäume aufbauen, sondern andere Datenstrukturen. Für einfache Sprachen kann durch diese schrittweise Verfeinerung des LL(1)-Parsers der gesamte Übersetzer gewonnen werden.

3 Übersetzung imperativer Programmiersprachen

3.1 Konzepte imperativer Programmiersprachen

Imperative Programmiersprachen spiegeln die von-Neumann-Rechnerarchitektur wider. Unter einer Berechnung versteht man in diesem Modell eine Folge von zustandsverändernden Aktionen. Imperative Programmiersprachen besitzen in der Regel folgende Konstrukte:

Variablen (imperativer Programmiersprachen)

- Eine Variable (eines Typs) ist ein Name für eine Speicherzelle. Der Inhalt der Speicherzelle ist ein Wert (des gleichen Typs), also ein Datenobjekt. Der Inhalt der Speicherzelle zu einem Zeitpunkt heißt *Wert der Variablen* zu diesem Zeitpunkt.
- Die Werte aller Variablen zu einem Zeitpunkt (also die Inhalte ihrer Speicherzellen) stellen den *Zustand* der Programmausführung dar.
- Der Wert einer Variablen kann durch zustandsverändernde Aktionen geändert werden, z. B. durch eine Wertzuweisung.
- Variablen können lokal zu einer Prozedur sein. Bei jedem Aufruf der Prozedur werden Speicherzellen für die lokalen Variablen der Prozedur reserviert. Am Ende der aufgerufenen Prozedur werden die Speicherzellen wieder freigegeben.
- Pro Aufruf einer Prozedur entsteht eine neue Inkarnation aller Variablen der Prozedur. Existieren zu einem Zeitpunkt mehrere Inkarnationen der selben Variablen, z. B. bei rekursiven Prozeduren, sind die Speicherzellen dafür also nicht die selben.

Die letzten beiden Punkte legen die Verwaltung der Variablen mittels eines Kellers nahe.

Ausdrücke

werden aus Konstanten, Namen (z. B. von Variablen) und Operatoren entsprechend ihrer Typen zusammengesezt.
Jeder Ausdruck hat einen Wert, der bei der Programmausführung bestimmt wird. Dieser Wert hängt im allgemeinen vom Zustand der Programmausführung ab.

Anweisungen

lösen Aktionen aus, die den Zustand der Programmausführung verändern können. Dazu gehören insbesondere die Wertzuweisung und Ein-/Ausgabe-Anweisungen.
Eine Anweisung hat im Gegensatz zu einem Ausdruck keinen Wert. Einige imperative Sprachen erlauben allerdings, gewisse Anweisungen auch als Ausdrücke zu verwenden, die dann doch einen Wert haben.

Explizite Angabe des Kontrollflusses.

Die Reihenfolge, in der Anweisungen ausgeführt werden, wird explizit durch das Programm gesteuert. Mittel dazu sind Konstrukte für:

- Sequenzialisierung: meist Hintereinanderschreiben mit Semikolon als Trennzeichen
- Verzweigung: `if...then`, `if...then...else`, `case`
- definite Iteration `for...from...to...`
- indefinite Iteration: `while`, `repeat...until`.

In älteren oder maschinennäheren imperativen Programmiersprachen kommen meist noch Anweisungen für Sprünge (`goto`) hinzu.

Prozeduren
: sind Unterprogramme, die aufgerufen werden können, so dass nach ihrer Ausführung der Kontrollfluss zur Aufrufstelle zurückkehrt.
Eine Prozedur kann *formale Parameter* haben, deren Werte bei den Aufrufen der Prozedur durch die *aktuellen Parameter* gegeben sind und bei jedem Aufruf andere sein können.
Eine Prozedur kann einen Wert als Ergebnis liefern. In diesem Fall kann sie auch in Ausdrücken aufgerufen werden. Prozeduren, die ein Ergebnis liefern, werden manchmal Funktionen genannt, aber die Terminologie Prozedur/Funktion ist im Bereich der imperativen Programmiersprachen ziemlich uneinheitlich.

Blockstruktur.
: Eine syntaktische Einheit, in der Vereinbarungen (insbesondere von Variablen) vorkommen können, heißt ein *Block*. Moderne imperative Sprachen erlauben (je nach Sprache mehr oder weniger stark eingeschränkte) Schachtelungen von Blöcken, für die das Überschattungsprinzip gilt.

Strukturierte Datentypen
: zur Aggregation von Datenobjekten: *Felder* (arrays), *Verbunde* (records).

Dynamische Datenstrukturen
: mit Verweistypen für Zeiger (pointer) zum Aufbau von graph-ähnlichen Netzen von Datenobjekten.

Beispiele imperativer Programmiersprachen: FORTRAN, COBOL, ALGOL 60, ALGOL 68, Pascal, Modula-2, Ada, C. Beispiele objektorientierter Programmiersprachen mit imperativem Kern: Simula, Modula 3, Ada 95, C++, Java.

3.2 Die imperative Programmiersprache *I*

I ist ein stark vereinfachtes Pascal (oder Modula), um die Übersetzung des Kontrollflusses eines imperativen Programms zu illustrieren. *I* verfügt über:

- Wertzuweisung
- Ein-/Ausgabe-Anweisung
- Sequenzialisierung
- Verzweigung (einseitiges `if`)
- indefinite Iteration (`while`)
- Prozeduren ohne Parameter und ohne Ergebnis, aber mit Blockschachtelung
- einziger Datentyp: ganze Zahlen mit einigen Operationen und Relationen, keine strukturierten Datentypen, keine Verweistypen

Die Syntax von *I* ist durch folgende Grammatik definiert:

Programm	::=	`"PROGRAM"` *Name* `";"` *Block* `"."` .
Block	::=	[`"CONST"` *Name* `"="` *Zahl* { `","` *Name* `"="` *Zahl* } `";"`] [`"VAR"` *Name* { `","` *Name* } `";"`] { `"PROCEDURE"` *Name* `";"` *Block* `";"` } *Anweisung* .

Anweisung	::=	*Name* ":=" *Ausdruck*
	|	"CALL" *Name*
	|	"READ" *Name*
	|	"WRITE" *Ausdruck*
	|	"BEGIN" *Anweisung* { ";" *Anweisung* } "END"
	|	"IF" *Bedingung* "THEN" *Anweisung*
	|	"WHILE" *Bedingung* "DO" *Anweisung* .
Bedingung	::=	*Ausdruck* ("=" | "<" | ">") *Ausdruck*
	|	"NOT" *Bedingung* .
Ausdruck	::=	["+" | "−"] *Term* { ("+" | "−") *Term* } .
Term	::=	*Faktor* { ("∗" | "/") *Faktor* } .
Faktor	::=	*Name* | *Zahl* | "(" *Ausdruck* ")" .

Die Definitionen der ganzen Zahlen und der Namen werden vorausgesetzt, das heißt, *Zahl* und *Name* werden als Terminalsymbole betrachtet. Sie sind reguläre Strukturen, die während der lexikalischen Analyse ermittelt werden können.

Diese Grammatik für I ist eine LL(1)-Grammatik. Man kann leicht überprüfen, dass ε aus keinem Nichtterminalsymbol ableitbar ist, so dass Teil 2 der LL(1)-Bedingung erfüllt ist. Wenn man die EBNF in BNF umwandelt, ist auch einfach nachzurechnen, dass die Grammatik Teil 1 der LL(1)-Bedingung erfüllt, vorausgesetzt, dass kein *Name* ein reserviertes Symbol wie `CALL` ist. Folglich setzen wir diese Bedingung im weiteren voraus.

Übung. Syntaxgraphen für I angeben. ■

Beispiel. Ein I-Programm zur Berechnung der Potenz zweier ganzer Zahlen (die zweite nicht-negativ) und der Fakultät einer nicht-negativen ganzen Zahl:

```
PROGRAM IBeispiel1;
  VAR a,b,pot, n,fak;

  PROCEDURE potenz;
    VAR y;
  BEGIN
    pot := 1;   y := b;
    WHILE NOT y < 1 DO BEGIN
      pot := pot*a;   y := y-1
    END
  END;

  PROCEDURE fakultaet;
  IF n > 1 THEN BEGIN
    fak := fak*n;   n := n-1;   CALL fakultaet
  END;

BEGIN
  READ a;   READ b;     CALL potenz;      WRITE pot;
  READ n;   fak := 1;   CALL fakultaet;   WRITE fak
END.
```
■

Übung. Weitere I-Programme schreiben. ■

3.3 Ein Parser für I

3.3.1 Vereinfachung des automatisch generierten Parsers

Der gemäß Abschnitt 2.2.3 generierte LL(1)-Parser für I kann noch vereinfacht werden: Die obige Grammatik für I hat die Eigenschaft, dass bei jeder Verzweigung im Syntaxgraphen höchstens eine Alternative mit einem Nichtterminalsymbol A beginnt. Daher kann spekulativ einfach angenommen werden, dass immer wenn die mit Terminalsymbolen beginnenden Alternativen nicht möglich sind, die mit A beginnende Alternative verfolgt werden muss. Der Test, ob das nächste vom Symbolentschlüssler gelieferte Symbol in $\mathit{first}(A)$ enthalten ist, kann dann entfallen. Wenn das Symbol nicht zu $\mathit{first}(A)$ gehört, wird der Fehler in der Funktion für A erkannt, sobald darin das erste Terminalsymbol verarbeitet wird. Die nach der Übersetzungsregel für Alternativen erzeugten Programmteile des Parsers werden üblicherweise auf diese Art vereinfacht.

Manchmal wird aber im Gegenteil der gewonnene LL(1)-Parser komplizierter gemacht, um geringfügige Abweichungen von einer LL(1)-Grammatik abzufangen, z. B. kann das Rücksetzverbot in einigen beschränkten Fällen aufgehoben werden.

3.3.2 Behandlung von Vereinbarungen

Zusätzlich zur syntaktischen Analyse gemäß der kontextfreien Grammatik muss überprüft werden, ob alle in Anweisungen vorkommenden Namen vereinbart (deklariert) worden sind. Dazu wird eine sogenannte *Symboltabelle* benutzt. Die Namen werden in die Tabelle eingetragen, sobald sie in einer Vereinbarung auftreten. Kommt ein Name in einer Anweisung vor, dann wird er in der Tabelle gesucht. Wird er dort nicht gefunden, fehlt eine Vereinbarung dafür.

In ähnlicher Weise lassen sich unerwünschte mehrfache Vereinbarungen des gleichen Namens erkennen.

Jede Vereinbarung gehört zu einem Block. In I ist ein Block jeweils als Rumpf des Hauptprogramms und der Prozeduren erlaubt, und in jedem Block können wieder Prozeduren vereinbart werden, so dass Blockschachtelung möglich ist. Eine einzige Symboltabelle für alle Blöcke des Programms reicht aus. Da die in einem Block vereinbarten Namen lokal zu diesem Block sind, werden sie am Ende der Analyse des Blocks aus der Tabelle entfernt.

Die Überprüfung von Vereinbarungen in I kann also durch eine einfache Erweiterung des Parsers und somit als Teil der syntaktischen Analyse behandelt werden, obwohl die Bedingungen dafür nicht in der kontextfreien Grammatik spezifiziert sind – und auch nicht spezifizierbar sind.

Eine Symboltabelle ist eine Standardkomponente jedes Übersetzers. Für Sprachen, die weniger einfach sind als I, das heißt, für alle praktischen Sprachen, kann die Überprüfung von Vereinbarungen, Typen, usw. aber nicht vom Parser übernommen werden. Statt dessen ist eine weitere Phase notwendig, die semantische Analsye. Sie wird oft zeitlich mit der syntaktischen Analyse verzahnt, um einen zusätzlichen Lauf zu vermeiden.

Bemerkung. Die Bedingung, dass alle verwendeten Namen vereinbart sein müssen und nicht mehrfach vereinbart sein dürfen, ist mit einer kontextfreien Grammatik nicht ausdrückbar. Die oben erläuterte Technik ermöglicht also, einen Parser für eine kontextfreie Grammatik zu einem Parser für eine gewisse Art von nicht-kontextfreien Grammatiken zu erweitern. ∎

3.3.3 Behandlung syntaktischer Fehler

Stößt der Parser auf einen Fehler, so ist es wünschenswert,

- diesen zu melden und
- die syntaktische Analyse trotzdem fortzusetzen.

Die Fortsetzung der syntaktischen Analyse erfordert, dass Annahmen über die Art des Fehlers gemacht werden. Diese Annahmen ermöglichen es, einen Teil des Programms zu überspringen (z. B. bis zum nächsten `END`), um ab der erreichten Stelle die Syntaxanalyse fortzusetzen und evtl. fehlende Symbole einzufügen.

Gute Annahmen zu machen ist schwer, und zu jeder Annahme gibt es Fehler, für die sie völlig falsch sind. Welche Annahmen sinnvoll sind, hängt davon ab, welche Fehler häufig vorkommen oder als wahrscheinlich angesehen werden. In Sprachen wie Pascal, Modula oder *I* werden `";"` häufiger vergessen als `"+"` oder `"-"`. Eine einfache Programmiersprache erleichtert nicht nur die Programmierung, sondern auch die Behandlung syntaktischer Fehler.

Einige heuristische Prinzipien zur vernünftigen Behandlung von syntaktischen Fehlern:

- Die Sprache muss zur regelmäßigen Nutzung von reservierten Symbolen zwingen, die wie `BEGIN`, `IF` oder `WHILE` selten vergessen werden.
- Da der Parser rekursiv konzipiert wurde, ist es wünschenswert, dass das Ende eines möglicherweise fehlerhaften Programmteils, wie z. B. einer Prozedur, möglichst richtig geraten werden kann. Dazu sind `BEGIN`-`END`- und ähnliche Klammerungs-Konstrukte günstig. Der Parser muss so erweitert werden, dass er bei jedem rekursiven Aufruf „weiß“, welche Symbole nach diesem Aufruf zulässig sind.
- Trennsymbole können hinzugefügt werden, um das Ende von besonders wichtigen Programmteilen, wie etwa Prozeduren, zu kennzeichnen.
- Häufige Fehler (wie das Vergessen eines `";"` in Pascal oder Modula) können durch eine Änderung der Grammatik in manchen Fällen zugelassen werden. Jedoch sollten in solchen Fällen Warnungen ausgegeben werden.

3.4 Die abstrakte Maschine *MI* für *I*

Die lexikalische, syntaktische und semantische Analyse können unabhängig von der Zielsprache spezifiziert werden, in die die Programme übersetzt werden sollen. Diese Unabhängigkeit ist sinnvoll und wünschenswert.

Für den weiteren Aufbau eines Übersetzers muss aber die Zielsprache berücksichtigt werden. Diese ist meistens eine Maschinensprache. Damit hängen die weiteren Phasen des Übersetzers von der Zielmaschine ab.

Um von einer realen Zielmaschine so unabhängig wie möglich zu sein und um den Übersetzer einfach zu halten, wird eine abstrakte Zielmaschine *MI* konzipiert, in die *I*-Programme übersetzt werden. Diese abstrakte Maschine kann dann auf realen Maschinen interpretiert (man sagt auch *emuliert*) werden. Abstrakte Maschinen können als Formalisierung (durch Algorithmen) von Prozessoren angesehen werden.

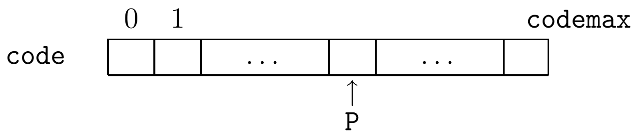
MI ist eine Nachahmung der abstrakten Maschine für Pascal und besteht aus:

- 2 Speichern:
  - 1 Programmspeicher `code` und
  - 1 Datenspeicher `store`.
- 4 Registern:
  - 1 Instruktionsregister `I` und
  - 3 Adressregistern:

    `P` = Programmzähler für den nächsten Befehl in `code`,
    `T` = Top des Kellers in `store`,
    `B` = Basis des aktuellen Segments in `store`.
- Der Sprache S_{MI} (das heißt, den Befehlen von *MI*)

3.4.1 Der Programmspeicher `code`

Der Programmspeicher `code` ist ein Feld der Länge `codemax+1` indiziert von 0 bis `codemax`. Es enthält ein Programm P_{MI} in der Sprache S_{MI}, das vom Übersetzer erzeugt wurde. Während der Ausführung des Programms P_{MI} durch die *MI*-Maschine verändert sich der Inhalt des Programmspeichers `code` nicht. Jeder Befehl (Instruktion, instruction) der Sprache S_{MI} belegt eine Zelle im Programmspeicher `code`. Das Instruktionsregister `I` enthält den Befehl, der gegenwärtig ausgeführt wird. Der Programmzähler `P` markiert, wo anschließend fortgesetzt wird. Der nächste auszuführende Befehl ist also jeweils der Inhalt von `code[P]`.

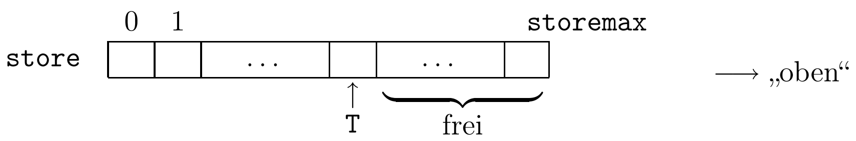


Vor der Ausführung eines Befehls wird der Inhalt des Programmzählers `P` jeweils um 1 hochgezählt. Der Haupt-Instruktionszyklus der abstrakten Maschine *MI* lautet also:

```
P:=0;   I:=code[P];
while (I≠HLT) { P:=P+1;   Befehl in I ausführen;   I:=code[P]; }
```

3.4.2 Der Datenspeicher `store`

Der Datenspeicher `store` ist ein Feld der Länge `storemax+1` indiziert von 0 bis `storemax`. Es wird als Keller verwaltet, der die Speicherzellen für Variablen sowie für Zwischenergebnisse von Berechnungen enthält. Das Register `T` dient als Kellerpegel. Sein Inhalt ist stets die Adresse der obersten belegten Kellerzelle. Die nächste freie Zelle im Datenspeicher ist also jeweils `store[T+1]`.



Manche Befehle „kellern“ den Inhalt einer (explizit adressierten) Zelle von `store` und verlängern dabei den Keller (`T:=T+1; store[T]:=store[i]`). Andere Befehle „entkellern“ den Inhalt der oberen Kellerzelle in eine (explizit adressierte) Zelle von `store` und verkürzen dabei den Keller (`store[i]:=store[T]; T:=T-1`). Weitere Befehle verändern den Inhalt von mehreren Kellerzellen. Genau betrachtet wird also in `store` kein „reiner Keller“ implementiert, weil nicht nur auf dessen oberste Zelle zugegriffen wird.

3.4.3 Die Prozedursegmente

Bei jedem Aufruf einer Prozedur entsteht eine eigene Inkarnation aller lokalen Variablen dieser Prozedur. Deshalb können die Speicherzellen für diese lokalen Variablen nicht zur Übersetzungszeit fest zugeordnet werden, sondern müssen zur Laufzeit bei jedem Aufruf der Prozedur belegt werden.

Statt von der Ausführung einer aufgerufenen Prozedur spricht man kürzer von einer Inkarnation dieser Prozedur. Der Speicherplatz, der einer Prozedur-Inkarnation für die lokalen Variablen und für andere Zwecke zur Verfügung steht, heißt *Prozedursegment* oder kurz *Segment* (*procedure segment* oder auch *activation record*). Das Segment einer Prozedur-Inkarnation muss belegt werden, wenn sie (zur Laufzeit) beginnt, und kann wieder freigegeben werden, wenn sie beendet ist.

Für zwei beliebige Inkarnationen gilt entweder, dass die eine zeitlich (zur Laufzeit) endet bevor die andere beginnt, oder dass die, die früher beginnt, später endet als die andere. Deshalb können die Segmente nach dem Kellerprinzip (LIFO: last in, first out) verwaltet werden. Prozedursegmente werden folglich einfach am oberen Ende des Kellers im Datenspeicher `store` angelegt.

Die Adresse der ersten Zelle, die zu einem Segment gehört, nennt man die *Basisadresse* des Segments. Das Register `B` wird benutzt, um die Basisadresse des obersten Segments zu speichern. Das oberste Segment belegt also jeweils die Zellen `store[B]` bis `store[T]`.

Ein Prozedursegment stellt den gesamten Speicherplatz zur Verfügung, den eine Inkarnation benötigt. Dazu gehört Speicherplatz für einige Verwaltungsdaten, für die lokalen Variablen und für Zwischenergebnisse von Berechnungen. Der Speicherplatz für die Verwaltungsdaten heißt auch *Segmentdeskriptor*.

Für jede Prozedur ist zur Übersetzungszeit bekannt und zur Laufzeit unveränderlich, wie viele Speicherzellen für den Segmentdeskriptor und für die lokalen Variablen benötigt werden. Die Anzahl der Speicherzellen für Zwischenergebnisse ändert sich dagegen zur Laufzeit. Dieser Teil des Segments wird wiederum kellerartig verwaltet. Man nennt ihn auch den *lokalen Keller* des Segments. Es ist klar, dass dieser veränderliche Teil des Segments am oberen Ende gespeichert werden sollte. In *MI* hat ein Prozedursegment folgenden Aufbau:

	Deskriptor					
…	DL	RA	SL	lokale Variablen	lokaler Keller	⟷ …
	↑ B				↑ T	

- DL (dynamic link) ist der Verweis auf den *dynamischen Vorgänger*, also die Basisadresse des Segments der Inkarnation, in der die aktuelle aufgerufen wurde. Das ist jeweils das unmittelbar vorhergehende Segment im Datenspeicher `store`. Diese Adresse wird gebraucht, um bei Beendigung der aktuellen Inkarnation das aktuelle Segment „abräumen“ zu können, das heißt, die aufrufende Inkarnation zur aktuellen machen zu können.

- RA (return address) ist die *Rückkehradresse*, also die Adresse des Befehls im Programmspeicher `code`, der unmittelbar auf den Prozeduraufruf folgt. Diese Adresse wird gebraucht, um bei Beendigung der aktuellen Inkarnation den Programmzähler `P` so besetzen zu können, dass die Programmausführung an der Aufrufstelle fortgesetzt wird.
- SL (static link) ist der Verweis auf den *statischen Vorgänger*, also die Basisadresse des Segments der Inkarnation, in der die aktuelle deklariert wurde. Das ist jeweils irgend ein vorhergehendes Segment im Datenspeicher `store`, aber nicht unbedingt das unmittelbar vorhergehende. Diese Adresse wird gebraucht, um während der Lebenszeit der aktuellen Inkarnation auf ihre nicht-lokalen Variablen zugreifen zu können. Diese Notwendigkeit wird im Folgenden erläutert.

Wenn Spracheigenschaften hinzukommen, über die die Sprache I nicht verfügt, sind zusätzliche Zellen im Segmentdeskriptor erforderlich, zum Beispiel für den Rückgabewert einer Funktion (siehe Abschnitt 3.7.2).

Notwendigkeit des statischen Verweises. Betrachten wir den folgenden Fall. Eine Prozedur P ruft eine Prozedur Q auf, die lokal zu P ist. Die aufgerufene Prozedur Q greift auf eine Variable x zu, die in P vereinbart ist, und ruft sich außerdem rekursiv auf.

```
PROCEDURE P;
   VAR x;

   PROCEDURE Q;
   BEGIN ... WRITE x; ... CALL Q; ... END;

BEGIN
    ... CALL Q; ...
END;
```

Die Variable x ist nicht-lokal zur Prozedur Q. Man sagt auch, dass x eine globale Variable für Q ist. Wenn in einer Programmiersprache nicht-lokale Variablen möglich sind, muss geklärt werden, nach welchen Regeln der Zugriff auf diese Variablen erfolgt. Man spricht von *Sichtbarkeitsregeln.* Die geläufigste Sichtbarkeitsregel ist das Überschattungsprinzip für die Blockschachtelung (ein Block ist in der Sprache I nur als Rumpf des Hauptprogramms und von Prozeduren erlaubt, deren Vereinbarungen aber geschachtelt werden dürfen):

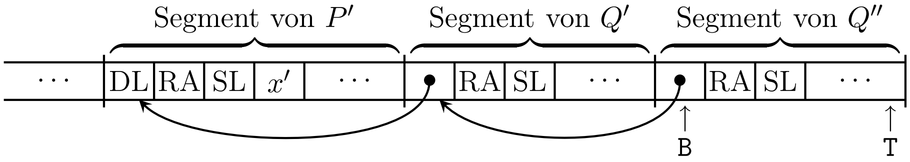
> Enthält ein Block eine Vereinbarung eines Namens N (zum Beispiel für eine Variable), so ist der Name N überall in diesem Block sichtbar, mit Ausnahme von darin geschachtelten Blöcken, die selbst eine Vereinbarung des Namens N enthalten.

Da diese Regel sich nach dem statischen Programmtext richtet und nicht nach der dynamischen Ausführung des Programms, wird sie *statische Sichtbarkeitsregel* genannt. (Einige Programmiersprachen ermöglichen durch besondere Konstrukte (`own` in ALGOL 60, `STATIC` in PL/1 und C), dass Inkarnationen von Variablen, die lokal zu einer Prozedur sind, die Inkarnation dieser Prozedur überleben. Dies wird im Folgenden nicht behandelt.)

Betrachten wir die Ausführung des obigen Programms zu einem Zeitpunkt, an dem Q sich ein Mal rekursiv aufgerufen hat. Um besser auf die Inkarnationen der aufgerufenen Prozeduren Bezug nehmen zu können, benennen wir sie unterschiedlich.

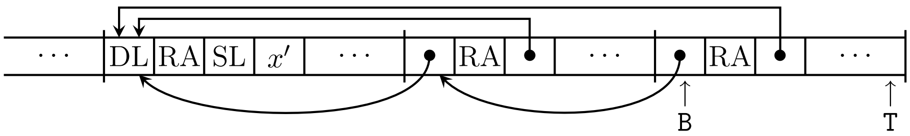
Aufrufstelle	aufgerufene Prozedur	Inkarnation	
irgendwo	P	P'	Anlegen einer Inkarnation x' von x
Anweisungsteil von P	Q	Q'	Zugriff auf die Inkarnation x' von x
Anweisungsteil von Q	Q	Q''	Zugriff auf die selbe Inkarnation x' von x

Zu diesem Zeitpunkt enthält der Datenspeicher `store` also die Segmente für die Inkarnationen P', Q' und Q'', wobei x' eine Speicherzelle im Segment von P' ist und P' der dynamische Vorgänger von Q' und Q' der dynamische Vorgänger von Q'' ist:



Zur Übersetzungszeit steht fest, dass x' die vierte Speicherzelle im Segment von P' ist, aber nicht, was die Basisadresse dieses Segments ist. Wenn von Q' aus auf x' zugegriffen werden muss, ist diese Basisadresse der Verweis auf den dynamischen Vorgänger, aber von Q'' aus liegt x' im dynamischen Vorvorgänger. Um auf x' zuzugreifen, müsste jede weitere Inkarnation von Q eine andere Anzahl von dynamischen Vorgängerverweisen verfolgen. Diese Anzahl kann nicht zur Übersetzungszeit bestimmt werden.

Das Problem wird durch den Verweis SL auf den statischen Vorgänger gelöst. Der statische Vorgänger von Q' und auch von Q'' ist P', und zwar weil Q innerhalb von P vereinbart ist. Zur Übersetzungszeit steht fest, dass von jeder Inkarnation von Q genau ein statischer Vorgängerverweis verfolgt werden muss, um zu dem Segment mit den lokalen Variablen von P zu kommen:



Die Verweise auf die statischen Vorgänger werden wie folgt ermittelt:

Zur Übersetzungszeit wird für jeden Block die *Blockschachtelungstiefe* bestimmt. Dazu muss nur beim Durchgang durch den Programmtext ein Zähler bei jedem Blockanfang inkrementiert und bei jedem Blockende dekrementiert werden.
Jeder Prozeduraufruf `CALL P` kommt in einem Block vor. Die zugehörige Vereinbarung der Prozedur `P` kommt im selben oder einem umfassenden Block vor. Die Differenz der Blockschachtelungstiefen dieser beiden Blöcke ist die „Stufe“ s, die dem Prozeduraufruf zugeordnet wird.
Im obigen Beispiel ist damit dem Aufruf `CALL Q` im Anweisungsteil von `P` die Stufe 0 zugeordnet, dem Aufruf `CALL Q` im Anweisungsteil von `Q` die Stufe 1.

Zur Laufzeit wird aus der Stufe eines Prozeduraufrufs der Wert SL für das Segment der Inkarnation berechnet, die durch den Aufruf neu entsteht:

- Für Stufe $s = 0$ ist SL der Inhalt von `B`. Der statische Vorgänger der neu entstehenden Inkarnation ist also die aufrufende Inkarnation.
  Der Fall $s = 0$ liegt vor, wenn von P' aus Q' neu entsteht.
- Für Stufe $s \geq 1$ erhält man den Wert SL, indem man ausgehend vom aktuellen Segment, dessen Basisadresse in Register `B` zugänglich ist, die Kette der Verweise auf die statischen Vorgänger s Schritte weit durchläuft. Das dort beginnende Segment gehört dem statischen Vorgänger der neu entstehenden Inkarnation.
  Der Fall $s = 1$ liegt vor, wenn von Q' aus Q'' neu entsteht.

Die Adressen der Speicherzellen von Variablen werden damit wie folgt ermittelt:

Zur Übersetzungszeit wird für jede Vereinbarung einer Variablen festgelegt, welche Adresse relativ zum Anfang des Prozedursegments ihre Inkarnationen haben werden. Diese Adresse wird *Offset* (Verschiebung) genannt.
Zu jedem Vorkommen einer Variablen in Anweisungen wird die Stufe s so berechnet wie für Prozeduraufrufe, also als Differenz der Blockschachtelungstiefen der zugehörigen Vereinbarung und des Vorkommens der Variablen.
Auf diese Weise ist jedem Vorkommen eine *Relativadresse* mit zwei Bestandteilen zugeordnet: einer Stufe s und einem Offset.

Zur Laufzeit wird aus einer Relativadresse bestehend aus einer Stufe s und einem Offset die *Absolutadresse* wie folgt ermittelt. Ausgehend vom aktuellen Segment, dessen Basisadresse in Register `B` zugänglich ist, wird die Kette der Verweise auf die statischen Vorgänger s Schritte weit durchlaufen. Die Summe aus der Basisadresse des erreichten Segments und dem Offset ergibt die gesuchte Absolutadresse.

3.4.4 Die Befehle der abstrakten Maschine *MI*

Zur Laufzeit müssen im Datenspeicher `store` Werte der folgenden Typen abgespeichert werden: Zahl, Wahrheitswert, Codeadresse (Index für `code`) und Datenadresse (Index für `store`). Es wird angenommen, dass jeder dieser Werte genau eine Speicherzelle in `store` benötigt, z. B. ein Wort. Diese Annahme ist für abstrakte Maschinen üblich. Für konkrete Maschinen werden dagegen oft „kleine Werte“, wie etwa Wahrheitswerte oder Zeichen, zu mehreren in ein Wort „gepackt“. Andere Werte, wie etwa Gleitpunktzahlen, benötigen eventuell sogar mehr als ein Wort.

Im Programmspeicher `code` werden die Befehle des übersetzten Programms gespeichert. Jeder dieser Befehle hat eine der folgenden Formen:

Befehlscode Argument$_1$ Argument$_2$
Befehlscode Argument
Befehlscode

Es wird angenommen, dass jeder Befehl genau eine Speicherzelle in `code` benötigt, egal ob er null, ein oder zwei Argumente hat und egal, von welchem Typ diese Argumente sind. Diese Annahme ist ebenfalls für abstrakte Maschinen üblich. Für konkrete Maschinen kann eine Tabelle benutzt werden, die die Größe jedes Befehls angibt, so dass der Programmzähler `P` um diese Größe erhöht wird, um die Adresse des nächsten Befehls zu erhalten.

Im Folgenden werden die Befehlscodes durch mnemonische Bezeichner wie etwa `HLT` oder `LOD` dargestellt. Die eigentlichen Befehlscodes sind aber Binärzahlen. Die Übersetzung der mnemonischen Darstellung in die binäre Darstellung ist unmittelbar. Sie kann Befehl für Befehl erfolgen, weil die Befehlssprache der abstrakten Maschine *MI* wie jede Maschinensprache *linear* ist, das heißt, keinerlei Klammerungskonstrukte oder sonstige Schachtelungen enthält.

Zu jedem *MI*-Befehl ist eine Folge von einfachen Anweisungen angegeben, die präzise spezifiziert, welche Wirkung der Befehl auf die Inhalte von Registern und Speicherzellen von *MI* hat. Diese „einfachen Anweisungen“ sind in erster Linie als Kommunikationsmittel für Menschen gedacht und nicht als Muster für die effizienteste Implementierung der beschriebenen Wirkung. Die Idee ist aber, dass diese „einfachen Anweisungen“ Konstrukte verwenden, die in Befehlssätzen konkreter Maschinen üblicherweise zur Verfügung stehen, und dass jeder *MI*-Befehl mit möglichst wenigen Befehlen einer konkreten Maschine realisiert werden kann.

RESET. Initialisierung.

`RST`	`B:=0;   store[B+0]:=0;   store[B+1]:=0;   store[B+2]:=0;   T:=B+2`

Normalerweise sollte jedes *MI*-Programm mit diesem Befehl beginnen. Damit er aber auch funktioniert, wenn er nicht in der Programmspeicherzelle `code[0]` steht, initialisiert er den Programmzähler `P` nicht. Der Programmzähler `P` wird im Haupt-Instruktionszyklus initialisiert, das Instruktionsregister `I` kommt sowieso nur darin vor und muss ebenfalls nicht von diesem Befehl initialisiert werden.

Die ersten drei Zellen des Datenspeichers `store` bilden den Segmentdeskriptor im Segment des Hauptprogramms. Dieser wird zwar nirgends benötigt, aber so hat dieses Segment den gleichen Aufbau wie alle anderen Segmente.

LOAD. Inhalt einer relativ adressierten Zelle auf den Keller laden.

`LOD s i`	`T:=T+1;   store[T]:=store[`*base*(s)`+i]`

`s` ist eine Stufe, `i` ist ein Offset, beide Argumente sind also natürliche Zahlen. Zusammen repräsentieren sie eine Relativadresse wie oben beschrieben. Die Notation *base*(s) steht für die Datenadresse nach s Schritten in der Kette der Verweise auf die statischen Vorgänger. Man kann sie durch folgende Funktionsdefinition spezifizieren (SL ist jeweils die 3. Zelle im Segment, hat also die Adresse Basisadresse+2):

```
Datenadresse base(int s) {
  Datenadresse A:=B;
  loop (s) A:=store[A+2];
  return A;
}
```

Da aber der Wert des Arguments `s` schon zur Übersetzungszeit bekannt ist, sollte diese Spezifikation zur Übersetzungszeit „entfaltet“ (substituiert) werden:

`LOD 0 i`	`T:=T+1;   store[T]:=store[B+i]`
`LOD 1 i`	`T:=T+1;   A:=store[B+2];   store[T]:=store[A+i]`
`LOD 2 i`	`T:=T+1;   A:=store[B+2];   A:=store[A+2];   store[T]:=store[A+i]`
`LOD 3 i`	`T:=T+1;   A:=store[B+2];   A:=store[A+2];   A:=store[A+2];   store[T]:=store[A+i]`

...

STORE. Oberste Kellerzelle entfernen, Inhalt in eine relativ adressierte Zelle speichern.

`STO s i`	`store[`*base*(s)`+i]:=store[T];   T:=T-1`

analog zum Befehl `LOD`.

INCREMENT. Segment erweitern.

`INC i`	`T:=T+i`

`i` ist eine natürliche Zahl.

LITERAL. Zahl auf den Keller laden.

`LIT n`	`T:=T+1;   store[T]:=n`

`n` ist eine natürliche Zahl.

JUMP. Unbedingter Sprung.

`JMP a`	`P:=a`

`a` ist eine Codeadresse.

JUMP-ON-TRUE, JUMP-ON-FALSE. Bedingter Sprung.

`JOT a`	`if (store[T]==true ) {P:=a};   T:=T-1`
`JOF a`	`if (store[T]==false) {P:=a};   T:=T-1`

`a` ist eine Codeadresse.

CALL. Prozeduraufruf.

`CAL s a`	`T:=T+1;   store[T+0]:=B;   store[T+1]:=P;   store[T+2]:=`$base(s)$`;` `B:=T;   T:=B+2;   P:=a`

`s` ist eine Stufe, also eine natürliche Zahl, `a` ist eine Codeadresse. Die Notation $base(s)$ steht für die Datenadresse nach s Schritten in der Kette der Verweise auf die statischen Vorgänger, wie bei `LOD` und `STO`. Das ist der Wert SL für die neu entstehende Inkarnation der aufgerufenen Prozedur, also die Basisadresse des Segments ihres statischen Vorgängers.

Für die neue Inkarnation wird ein neues Segment mit dem Segmentdeskriptor angelegt.

Der erste *MI*-Befehl der aufgerufenen Prozedur steht in der Programmspeicherzelle `code[a]`, aber der Name dieser Prozedur ist nicht mehr repräsentiert.

RETURN. Rückkehr von einer Prozedur.

`RET`	`P:=store[B+1];   T:=B-1;   B:=store[B]`

Das oberste Segment gehört der endenden Inkarnation und wird abgeräumt. Sein Segmentdeskriptor enthält die Inhalte der *MI*-Register beim Entstehen der Inkarnation, die jetzt wiederhergestellt werden.

OPERATOR. Boole'sche und arithmetische Operatoren.

`OPR ¬`	`store[T]:=not(store[T])`
`OPR =`	`store[T-1]:=(store[T-1]== store[T]);   T:=T-1`
`OPR <`	`store[T-1]:=(store[T-1] < store[T]);   T:=T-1`
`OPR >`	`store[T-1]:=(store[T-1] > store[T]);   T:=T-1`
`OPR +`	`store[T-1]:=(store[T-1] + store[T]);   T:=T-1`
`OPR -`	`store[T-1]:=(store[T-1] - store[T]);   T:=T-1`
`OPR *`	`store[T-1]:=(store[T-1] * store[T]);   T:=T-1`
`OPR /`	`store[T-1]:=(store[T-1] / store[T]);   T:=T-1`

Die obersten Kellerzellen (so viele wie die Stelligkeit des Operators) werden entfernt, ihre Inhalte verknüpft, und das Ergebnis auf den Keller geladen.

READ. Eingabe lesen und auf den Keller laden.

`REA`	`T:=T+1;   store[T]:=read()`

WRITE. Oberste Kellerzelle entfernen, Inhalt ausgeben.

`WRI`	`write(store[T]);   T:=T-1`

HALT. Ausführung von *MI*-Befehlen beenden.

`HLT`	Siehe Haupt-Instruktionszyklus

3.5 Code-Erzeugung für *I*

Die Übersetzungsphase der Code-Erzeugung hat als Eingabe den abstrakten Syntaxbaum eines *I*-Programms, das von den vorherigen Phasen der Übersetzung bereits auf Korrektheit überprüft wurde. Ausgehend vom konkreten Syntaxbaum gemäß der Grammatik der Sprache entsteht der abstrakte Syntaxbaum einerseits durch Weglassen von Teilen, die nach der syntaktischen Analyse gar nicht mehr gebraucht werden (z. B. die Symbole `IF` und `THEN` in einer bedingten Anweisung), andererseits durch Hinzufügen von Ergebnissen der semantischen Analyse (z. B. Typinformation).

In diesem Abschnitt wird keine Datenstruktur für den abstrakten Syntaxbaum verwendet, sondern jeweils eine textuelle Beschreibung wie in der Grammatik für *I*. Die textuelle Beschreibung soll aber einfach als Schreibweise für eine geeignete Datenstruktur verstanden werden, die den entsprechenden Teilbaum des abstrakten Syntaxbaums repräsentiert:

Bedingte Anweisung der Form `IF` *Bedingung* `THEN` *Anweisung*	steht für	*BedingteAnweisung* *Bedingung* *Anweisung*

Für die Erzeugung der Relativadressen von Speicherzellen verwendet das Code-Erzeugungs-Programm zwei Zähler und eine kellerartig verwaltete Liste:

`Tiefenzähler` zum Festhalten der Blockschachtelungstiefe des *I*-Programmstücks, für das jeweils gerade *MI*-Code erzeugt wird.

Initialisiert mit 0. Wird bei jedem Eintritt in einen Block (d. h., beim Beginn der Bearbeitung eines Teilbaums mit Wurzel *Block* im Syntaxbaum) inkrementiert, bei jedem Verlassen eines Blocks dekrementiert.

`Namenzähler` zur Berechnung der Offsets von Konstanten und Variablen.

Wird bei jedem Eintritt in einen Block mit der Länge des Segmentdeskriptors für diesen Block initialisiert (bisher also stets 3) und nach jedem Namen in einer Konstanten- oder Variablenvereinbarung dieses Blocks inkrementiert (würden Werte verschiedener Typen unterschiedlich viele Zellen im Datenspeicher `store` benötigen, würde das Inkrement vom Typ abhängen).

`Symboltabelle` zum Festhalten der Werte dieser Zähler für jede Vereinbarung.

Beim Bearbeiten der Vereinbarungen eines Blocks werden Einträge in die Symboltabelle hinzugefügt, die beim Verlassen des Blocks wieder entfernt werden. Somit pulsiert die Symboltabelle kellerartig.

Für jeden Namen N in einer Konstanten- oder Variablenvereinbarung wird ein Tripel (N, t, i) gekellert, wobei t der aktuelle Stand des Tiefenzählers und i der aktuelle Stand des Namenszählers ist.

Für jeden Namen N einer im Block vereinbarten Prozedur wird ein Tripel (N, t, a) gekellert, wobei t der aktuelle Stand des Tiefenzählers ist (also vor der Erhöhung beim Eintritt in den Block der Prozedur) und a die aktuelle Codeadresse ist (der nächste erzeugte *MI*-Befehl kommt also in `code[a]`).

Berechnung der Relativadressen. Für jedes Vorkommen eines Namens N im Anweisungsteil eines Blocks wird die Symboltabelle von oben her nach dem ersten Tripel (N, t, x) durchsucht, das diesen Namen enthält. Die Stufe dieses Vorkommens ist $s = \texttt{Tiefenzähler} - t$, da in der Tabelle die Blockschachtelungstiefe der Vereinbarung gespeichert wurde, die für dieses Vorkommen gültig ist. Handelt es sich um ein Vorkommen des Namens N in einem Prozeduraufruf `CALL` N, so ist x die Codeadresse a des ersten *MI*-Befehls der Prozedur, in allen anderen Fällen ist x das Offset i der Relativadresse.

Im Folgenden setzen wir drei Funktionen *stufe*(N), *offset*(N) und *codeadresse*(N) als Bestandteil des Code-Erzeugungs-Programms voraus, die zu einem Namen N die entsprechenden Werte in der beschriebenen Weise liefern.

Bemerkung. Die Symboltabelle wird üblicherweise effizienter realisiert als oben beschrieben. Man kann für jeden im *I*-Programm vorkommenden Namen einen eigenen Keller verwenden (und beim Austritt aus den Blöcken richtig aktualisieren). Dann ist der relevante Eintrag für einen Namen immer das oberste Element im zugehörigen Keller. Um von einem Namen schnell zum zugehörigen Keller zu kommen, kann man eine Hash-Tabelle verwenden.

Im Übrigen beginnt der Aufbau der Symboltabelle mit Hilfe von Hash-Tabellen meistens bereits in der lexikalischen Analyse. Im Syntaxbaum ist ein Name dann durch einen Knoten repräsentiert, der keinen String enthält, sondern einen Index in die Symboltabelle, so dass alle Vorkommen des gleichen Namens den selben Index haben. Dieser Index führt zu einem Eintrag in der Symboltabelle, in dem neben dem String auch der Verweis auf den zugehörigen Keller gespeichert ist.

Das alles beeinflusst die Effizienz, aber nicht die prinzipielle Vorgehensweise zur Bestimmung der Relativadressen. ∎

Übersetzungsregel (Programm).

Code für	`PROGRAM` *Name* ; *Block* .	ist	`RST`
			⟨Code für *Block*⟩
			`HLT`

Übersetzungsregel (Block).

Code für	*Konstantenvereinbarung* ; *Variablenvereinbarung* ; *Prozedurvereinbarungen* ; *Anweisung*	ist	⟨Code für *Konstantenvereinbarung*⟩ ⟨Code für *Variablenvereinbarung*⟩ `JMP` L ⟨Code für *Prozedurvereinbarungen*⟩ L: ⟨Code für *Anweisung*⟩

Am Anfang der Übersetzung mit dieser Regel wird zunächst `Tiefenzähler` inkrementiert und `Namenzähler` initialisiert, am Ende der Übersetzung mit dieser Regel werden alle Tripel aus der Symboltabelle entfernt, deren Tiefe gleich `Tiefenzähler` ist, danach wird `Tiefenzähler` dekrementiert.

`L` ist die Codeadresse, an der der Code für den Anweisungsteil des Blocks beginnt. Der Sprungbefehl sorgt dafür, dass zur Laufzeit der Code für die Prozedurvereinbarungen übersprungen wird. Der Sprungbefehl entfällt, wenn der Block keine Prozedurvereinbarung enthält.

Die Codeadresse `L` ist an der Stelle, an der der Sprungbefehl erzeugt wird, noch nicht bekannt, sondern erst nach der Code-Erzeugung für alle vereinbarten Prozeduren. Dieses Problem der Vorwärtssprünge kann auf verschiedene Weisen behandelt werden. Am verbreitetsten sind die folgenden Techniken.

Code-Erzeugung in zwei Läufen: Als Zieladressen von Vorwärtssprüngen werden zunächst symbolische Marken wie `L` eingetragen. Während der weiteren Code-Erzeugung wird Buch geführt, welchen tatsächlichen Codeadressen diese symbolischen Marken entsprechen. In einem zweiten Lauf durch den erzeugten Code werden die symbolischen Marken durch die tatsächlichen Adressen ersetzt.

Fixup: (Instandsetzen) Der erzeugte Code wird in eine direkt adressierbare Datenstruktur gespeichert, die im Hauptspeicher gehalten werden kann, etwa ein Feld (array). Dadurch ist ein nachträgliches Hinzufügen von Zieladressen möglich, sobald diese bekannt sind.

Übersetzungsregel (Variablenvereinbarung).

Code für	`VAR` $Name_1$ `,` ..., $Name_n$	ist	`INC` n

Bei jedem der n Namen wird das entsprechende Tripel in die Symboltabelle gespeichert, nach jedem Namen wird `Namenzähler` inkrementiert.

Übersetzungsregel (Konstantenvereinbarung).

Code für	`CONST` $Name_1$ `=` $Zahl_1$ `,` ..., $Name_n$ `=` $Zahl_n$	ist	`INC` n `LIT` $\langle$Code für $Zahl_1\rangle$ `STO 0` i_1 ... `LIT` $\langle$Code für $Zahl_n\rangle$ `STO 0` i_n

Bei jedem der n Namen wird das entsprechende Tripel in die Symboltabelle gespeichert, nach jedem Namen wird `Namenzähler` inkrementiert. Damit sind die Offset-Werte $i_1, \ldots, i_n$ mittels $\mathit{offset}(Name_1), \ldots, \mathit{offset}(Name_n)$ verfügbar oder direkt als Wert von `Namenzähler` an der jeweiligen Stelle.

Auf den ersten Blick stellt sich die Frage, warum für die Zahlen Code erzeugt werden soll. Aber in einer Konstantenvereinbarung wie `CONST c = 299792458` in einem *I*-Programm ist die *Zahl* zunächst einfach eine Zeichenreihe `"299792458"`, die in die Repräsentation des entsprechenden Werts eines Zahltyps der Zielmaschine übersetzt werden muss. Von dieser Zielmaschine hängt unter anderem die größte darstellbare natürliche Zahl und die Genauigkeit von Gleitpunktzahlen ab.

Um Zahlen korrekt übersetzen zu können, muss also zumindest die Zahldarstellung der konkreten Maschine bekannt sein. Man kann die Übersetzung in der Phase der Code-Erzeugung durchführen, wenn man in Kauf nimmt, dass der Übersetzer an eine spezielle konkrete Maschine (oder Klasse von konkreten Maschinen) angepasst ist. Soll der Übersetzer unabhängig von den konkreten Maschinen bleiben, muss die Erzeugung von Code für Zahlen in die abstrakte Maschine verlagert werden. Man kann mit der Abstraktion so weit gehen, einfach die Zeichenreihe aus dem *I*-Programm in den entsprechenden *MI*-Befehl zu übernehmen. Oder man erweitert die abstrakte Maschine um Befehle zur Umwandlung von Zeichenreihen in Zahldarstellungen, die dann auf den konkreten Maschinen entsprechend zu realisieren sind.

Die Übersetzungsregel für Konstantenvereinbarungen belegt für jede Konstante eine Speicherzelle wie für eine Variable und übersetzt Vorkommen der Konstanten in Zugriffe auf diese Speicherzelle. Man kann Vorkommen der Konstanten auch direkt in die Zahl aus der Konstantenvereinbarung übersetzen und so Speicherplatz und Rechenzeit zur Laufzeit einsparen.

Diese Vorgehensweise ist aber problematisch, wenn die Quellsprache im Gegensatz zu I in einer Konstantenvereinbarung nicht nur eine Zahl erlaubt, sondern einen konstanten Ausdruck: `CONST pi = 4.0*atan(1.0)`. In diesem Beispiel wäre es einerseits ungünstig, jedes Vorkommen der Konstanten in den Code des Ausdrucks zu übersetzen, andererseits kann der Wert des Ausdrucks nicht zur Übersetzungszeit bestimmt werden, ohne Annahmen über die Zahldarstellung der konkreten Maschine und über die Eigenschaften der verwendeten Implementierung der Arcustangens-Funktion vorauszusetzen.

Übersetzungsregel (Prozedurvereinbarungen).

Code für	`PROCEDURE` $Name_1$ `;` $Block_1$ `;` ... `PROCEDURE` $Name_n$ `;` $Block_n$	ist	⟨Code für $Block_1$⟩ `RET` ... ⟨Code für $Block_n$⟩ `RET`

Bei jedem der n Namen wird das entsprechende Tripel mit der aktuellen Codeadresse `a` (für den nächsten zu erzeugenden *MI*-Befehl) in die Symboltabelle gespeichert. Nach jedem Namen wird `Namenzähler` inkrementiert. Dagegen bleibt `Tiefenzähler` in dieser Regel unverändert (er wird mit der Regel für die Übersetzung jedes Blocks erhöht und dann wieder erniedrigt).

Übersetzungsregel (Wertzuweisung).

Code für	*Name* `:=` *Ausdruck*	ist	⟨Code für *Ausdruck*⟩ `STO` s i

wobei $s = stufe(Name)$ und $i = offset(Name)$ ist.

Übersetzungsregel (Prozeduraufruf).

Code für	`CALL` *Name*	ist	`CAL` s a

wobei $s = stufe(Name)$ und $a = codeadresse(Name)$ ist.

Übersetzungsregel (Eingabe).

Code für	`READ` *Name*	ist	`REA` `STO` s i

wobei $s = stufe(Name)$ und $i = offset(Name)$ ist.

Übersetzungsregel (Ausgabe).

Code für	`WRITE` *Ausdruck*	ist	⟨Code für *Ausdruck*⟩ `WRI`

Übersetzungsregel (Anweisungsfolge).
Der Code für eine Anweisungsfolge ist die Folge der Codes für die Anweisungen.

Übersetzungsregel (Bedingte Anweisung).

Code für	`IF` *Bedingung* `THEN` *Anweisung*	ist	⟨Code für *Bedingung*⟩ `JOF` L ⟨Code für *Anweisung*⟩ L:

Hier entsteht ein Vorwärtssprung, der mit einer der erwähnten Techniken behandelt werden kann. Verzweigungen mit `if...then...else` oder `case`, die es in I nicht gibt, werden ähnlich übersetzt.

Übersetzungsregel (Wiederholungsanweisung).

Code für	`WHILE` *Bedingung* `DO` *Anweisung*	ist	`JMP` L_2 L_1: ⟨Code für *Anweisung*⟩ L_2: ⟨Code für *Bedingung*⟩ `JOT` L_1

Der unbedingte Sprung am Anfang ist ein Vorwärtssprung, der mit einer der erwähnten Techniken behandelt werden kann. Für den Rückwärtssprung am Ende reicht dagegen eine Hilfsvariable im Code-Erzeugungs-Programm.

Falls die While-Schleife n Mal durchlaufen wird, wird mit dieser Übersetzung $(n+1)$ Mal ein Sprungbefehl ausgeführt. Mit der folgenden Übersetzung, die auch korrekt ist, wird dagegen $(2n+1)$ Mal ein Sprungbefehl ausgeführt. Man beachte, dass ein bedingter Sprung in jedem Fall einen Befehlstakt beansprucht, egal ob er eine Änderung des Programmzählers bewirkt (also wirklich „springt“) oder nicht.

Code für	`WHILE` *Bedingung* `DO` *Anweisung*	ist	L_1: ⟨Code für *Bedingung*⟩ `JOF` L_2 ⟨Code für *Anweisung*⟩ `JMP` L_1 L_2:

Eine (in I nicht vorhandene) `repeat...until`-Anweisung könnte wie folgt übersetzt werden:

Code für	`REPEAT` *Anweisung* `UNTIL` *Bedingung*	ist	L_1: ⟨Code für *Anweisung*⟩ ⟨Code für *Bedingung*⟩ `JOF` L_1

Sonstige Wiederholungsanweisungen werden ähnlich übersetzt.

Übersetzungsregel (Bedingung oder Ausdruck).

Code für	*Name*	ist	`LOD` s i
Code für	*Zahl*	ist	`LIT` ⟨Code für *Zahl*⟩
Code für	α *Ausdruck*	ist	⟨Code für *Ausdruck*⟩ `OPR` α
Code für	*Ausdruck*$_1$ β *Ausdruck*$_2$	ist	⟨Code für *Ausdruck*$_1$⟩ ⟨Code für *Ausdruck*$_2$⟩ `OPR` β

wobei $s = \mathit{stufe}(\mathit{Name})$ und $i = \mathit{offset}(\mathit{Name})$ ist, für den Code von Zahlen die Bemerkungen von oben gelten, α ein einstelliger und β ein zweistelliger Boole'scher oder Vergleichs- oder arithmetischer Operator ist, der sowohl in I als auch in MI vorkommt. Die arithmetischen Vorzeichen kommen in I vor, aber nicht in MI. Für sie lautet die Regel:

Code für	$+$ *Ausdruck*	ist	⟨Code für *Ausdruck*⟩
Code für	$-$ *Ausdruck*	ist	`LIT` ⟨Code für 0⟩ ⟨Code für *Ausdruck*⟩ `OPR` $-$

Die Sprache I enthält keine zweistelligen Boole'schen Operatoren für Konjunktion und Disjunktion. Die obige Übersetzungsregel bewirkt, dass immer beide Operanden eines zweistelligen Operators β ausgewertet werden. Wenn die Quellsprache Operatoren für Konjunktion und Disjunktion mit dieser Semantik anbietet und die abstrakte Maschine um entsprechende Operatoren erweitert wird, kann die selbe Übersetzungsregel dafür verwendet werden.

Die meisten Programmiersprachen definieren aber Konjunktion und Disjunktion so, dass der zweite Operand nicht ausgewertet wird, wenn nach der Auswertung des ersten Operanden bereits das Gesamtergebnis des Boole'schen Ausdrucks feststeht. Das ist praktisch, um undefinierte Fälle abzufangen: `WHILE n > 0 AND 1/n < x DO n := n-1`. Wenn `n` den Wert 0 hat, wird der Teilausdruck `1/n < x` nicht ausgewertet und damit die Division durch Null vermieden. Konjunktion und Disjunktion mit dieser Semantik werden statt mit der obigen Regel in geeignete Verschachtelungen von bedingten Ausdrücken bzw. bedingten Anweisungen umgewandelt, für die dann Code erzeugt wird.

Bei allen Varianten der Übersetzungsegel für einen Ausdruck sind die Bemerkungen vom Anfang dieses Abschnitts relevant: die textuelle Darstellung auf der linken Seite einer Übersetzungsregel steht in Wirklichkeit für den entsprechenden Syntaxbaum. Klammerungen sowie Präzedenzen und Assoziativitätseigenschaften der Operatoren wurden beim Aufbau des Syntaxbaums berücksichtigt und spielen hier keine Rolle mehr.

Die Übersetzungsregel entspricht einer Linearisierung des Syntaxbaums in Postordnung, also der Postfixform des Ausdrucks. Das ist kein Zufall, sondern ein allgemeines Prinzip: die Postfixform von Ausdrücken entspricht genau der Befehlsfolge für ihre Auswertung auf einer Kellermaschine. Das folgende Beispiel soll das Prinzip illustrieren:

Beispiel.

Ausdruck `((3 + 5) + 2) * (12 + (9 * 2)))`

Postfixform `3 5 + 2 + 12 9 2 * + *`

MI-Befehlsfolge:

Befehl	Veränderung des Kellers bei Ausführung				
	...				
`LIT 3`	...	3			
`LIT 5`	...	3	5		
`OPR +`	...	8			
`LIT 2`	...	8	2		
`OPR +`	...	10			
`LIT 12`	...	10	12		
`LIT 9`	...	10	12	9	
`LIT 2`	...	10	12	9	2
`OPR *`	...	10	12	18	
`OPR +`	...	10	30		
`OPR *`	...	300			

Nach der Ausführung dieser Befehlsfolge ist der Keller um eine einzige Zelle größer als vor der Ausführung, und diese oberste Kellerzelle enthält den Wert des Ausdrucks. ■

Die Anwendung der beschriebenen Übersetzungsregeln auf das Beispiel-*I*-Programm vom Anfang dieses Kapitels ergibt das folgende *MI*-Programm. Die Kommentare und Trennlinien sollen nur das Verständnis des Codes erleichtern.

	a	code[a]	Kommentar
`Tiefenzähler` = 0	0	RST	
`Tiefenzähler` = 1	1	INC 5	VAR a,b,pot, n,fak
	2	JMP 37	
`Tiefenzähler` = 1			PROCEDURE potenz
`Tiefenzähler` = 2	3	INC 1	VAR y
Symboltabelle	4	LIT 1	
$(y, 2, 3)$	5	STO 1 5	pot := 1
$(\mathit{potenz}, 1, 3)$	6	LOD 1 4	
$(\mathit{fak}, 1, 7)$	7	STO 0 3	y := b
$(n, 1, 6)$	8	JMP 17	
$(\mathit{pot}, 1, 5)$	9	LOD 1 5	Anweisung von WHILE
$(b, 1, 4)$	10	LOD 1 3	
$(a, 1, 3)$	11	OPR *	
	12	STO 1 5	pot := pot*a
	13	LOD 0 3	
	14	LIT 1	
	15	OPR -	
	16	STO 0 3	y := y-1
	17	LOD 0 3	Bedingung von WHILE
	18	LIT 1	
	19	OPR <	
	20	OPR ¬	
	21	JOT 9	WHILE NOT y < 1
	22	RET	
`Tiefenzähler` = 1			PROCEDURE fakultaet
`Tiefenzähler` = 2	23	LOD 1 6	
Symboltabelle	24	LIT 1	
$(\mathit{fakultaet}, 1, 23)$	25	OPR >	
$(\mathit{potenz}, 1, 3)$	26	JOF 36	IF n > 1
$(\mathit{fak}, 1, 7)$	27	LOD 1 7	
$(n, 1, 6)$	28	LOD 1 6	
$(\mathit{pot}, 1, 5)$	29	OPR *	
$(b, 1, 4)$	30	STO 1 7	fak := fak*n
$(a, 1, 3)$	31	LOD 1 6	
	32	LIT 1	
	33	OPR -	
	34	STO 1 6	n := n-1
	35	CAL 1 23	CALL fakultaet
	36	RET	
`Tiefenzähler` = 1	37	REA	
Symboltabelle	38	STO 0 3	READ a
$(\mathit{fakultaet}, 1, 23)$	39	REA	
$(\mathit{potenz}, 1, 3)$	40	STO 0 4	READ b
$(\mathit{fak}, 1, 7)$	41	CAL 0 3	CALL potenz
$(n, 1, 6)$	42	LOD 0 5	
$(\mathit{pot}, 1, 5)$	43	WRI	WRITE pot
$(b, 1, 4)$	44	REA	
$(a, 1, 3)$	45	STO 0 6	READ n
	46	LIT 1	
	47	STO 0 7	fak := 1
	48	CAL 0 23	CALL fakultaet
	49	LOD 0 7	
	50	WRI	WRITE fak
	51	HLT	

Implementierung des Code-Erzeugungs-Programms. Die beschriebenen Übersetzungsregeln können im Fall der Sprache I direkt in die entsprechenden Funktionen des Parsers integriert werden. Das ist, wie auch schon bei der semantischen Analyse für I, eine Anwendung des Prinzips der schrittweisen Verfeinerung (stepwise refinement) aus Abschnitt 2.3. Dazu müssen die Strukturanknüpfungen Datenstrukturen aufbauen, die die Symbole der Grammatik zusammen mit Zusatzinformation repräsentieren können. Ein Knoten für die Vereinbarung einer Variablen kann beispielsweise einen Verweis in die Symboltabelle enthalten, wo der Schachtelungstiefen-Keller zur Bestimmung der Relativadresse repräsentiert ist. Ein Knoten für ein Vorkommen einer Variablen kann die Relativadresse bestehend aus Stufe und Offset enthalten. Bei Knoten für Konstanten käme noch die jeweilige Zahl dazu.

Wie schon bei der semantischen Analyse gilt aber, dass die Phase der Code-Erzeugung für praktische Sprachen zu komplex ist, um in den Parser integriert werden zu können.

3.6 Speicherbelegung für sonstige konkrete Datentypen

In diesem Abschnitt werden Erweiterungen der abstrakten Maschine *MI* angesprochen, die für weniger einfache Programmiersprachen als I notwendig sind.

3.6.1 Speicherbelegung für statische Felder

Die meisten imperativen Programmiersprachen erlauben die Vereinbarung von n-dimensionalen Feldern, wobei für jede der n Dimensionen eine andere Anzahl von Indexwerten spezifiziert werden kann. Manche Sprachen erzwingen, dass in jeder Dimension die Indizierung bei 0 beginnt, andere erlauben in jeder Dimension die Angabe einer Untergrenze a und einer Obergrenze e zum Beispiel in der Form:

`VAR f: ARRAY [`a_1`..`e_1`, ..., `a_n`..`e_n`]`

Sind die Werte $a_1, e_1, \ldots, a_n, e_n$ zur Übersetzungszeit bekannt, nennt man das Feld *statisch*.

Für die Speicherbelegung muss die (gedachte) n-dimensionale Anordnung der Feldelemente zunächst linearisiert werden. Die Feldelemente werden dann in aufeinanderfolgenden Speicherzellen in der Reihenfolge dieser Linearisierung gespeichert.

Eine mögliche Linearisierung erhält man, wenn man mit dem Feldelement `f[`a_1`, ..., `a_n`]` beginnt und die Indexwerte schrittweise erhöht, und zwar umso schneller, je weiter rechts sie stehen. Für ein dreidimensionales Feld vom Typ `ARRAY [0..23, 0..59, 0..59]` zur Repräsentation von Uhrzeiten erhält man diese Linearisierung, wenn man eine Digitaluhr auf Mitternacht stellt und dann so lange vorstellt, bis sie eine Sekunde vor Mitternacht anzeigt.

Für eine zweidimensionale Matrix, zum Beispiel `VAR m: ARRAY [-2..1, 3..5]`, liefert diese Linearisierung die Feldelemente zeilenweise von links nach rechts, weshalb man auch von der „zeilenweisen" Linearisierung spricht:

```
m[-2,3]   m[-2,4]   m[-2,5]
m[-1,3]   m[-1,4]   m[-1,5]
m[ 0,3]   m[ 0,4]   m[ 0,5]
m[ 1,3]   m[ 1,4]   m[ 1,5]
```

Für ein n-dimensionales Feld `VAR f: ARRAY [`a_1`..`e_1`, ..., `a_n`..`e_n`]` wird Speicherplatz mit dem *MI*-Befehl `INC` belegt. Dazu muss die Anzahl der Speicherzellen berechnet werden, die für die Feldelemente benötigt werden. Unter der Annahme, dass jedes Feldelement eine Speicherzelle im Datenspeicher `store` benötigt, ist diese Anzahl

$$\mathit{größe}(f) = \prod_{j=1}^{n} (e_j - a_j + 1)$$

Die Relativadresse bestehend aus $\mathit{stufe}(f)$ und $\mathit{offset}(f)$ wird für ein Feld berechnet wie für Variablen mit skalarem Typ. In der Speicherzelle mit dieser Relativadresse ist der Wert des Feldelements `f[`a_1`, ..., `a_n`]` gespeichert. Für ein Feldelement `f[`i_1`, ..., `i_n`]` muss zum Offset noch ein Wert addiert werden, der von den Indizes abhängt: $\mathit{offset}(f) + \mathit{rg}_f(i_1, \ldots, i_n)$.

Um diesen „Rang" eines Feldelements `f[`i_1`, ..., `i_n`]` zu ermitteln, betrachten wir zunächst den Sonderfall, dass $a_1 = \ldots = a_n = 1$ ist. Dann ist

$$\begin{array}{rclcccc}
\mathit{rg}_f(i_1, \ldots, i_n) & = & (i_1 - 1) \cdot & e_2 & \cdot\, e_3 \cdot \ldots \cdot & e_{n-1} & \cdot & e_n \\
& + & & (i_2 - 1) & \cdot\, e_3 \cdot \ldots \cdot & e_{n-1} & \cdot & e_n \\
& \vdots & & & \ldots & & & \vdots \\
& + & & & & (i_{n-1} - 1) & \cdot & e_n \\
& + & & & & & & (i_n - 1)
\end{array}$$

Der Summand in der ersten Zeile der Gleichung berücksichtigt die Feldelemente

$$\begin{array}{lcl}
\texttt{f[}\ \ 1\texttt{,}\ 1\texttt{,}\ldots\texttt{,}1\texttt{]} & \ldots & \texttt{f[}\ \ 1\texttt{,}\ e_1\texttt{,}\ldots\texttt{,}e_n\texttt{]} \\
\texttt{f[}\ \ 2\texttt{,}\ 1\texttt{,}\ldots\texttt{,}1\texttt{]} & \ldots & \texttt{f[}\ \ 2\texttt{,}\ e_1\texttt{,}\ldots\texttt{,}e_n\texttt{]} \\
 & \ldots & \\
\texttt{f[}i_1 - 1\texttt{,}1\texttt{,}\ldots\texttt{,}1\texttt{]} & \ldots & \texttt{f[}i_1 - 1\texttt{,}e_1\texttt{,}\ldots\texttt{,}e_n\texttt{]}
\end{array}$$

das heißt, der Summand führt zum Anfang des $(n-1)$-dimensionalen Unterfelds, in dem das gesuchte Feldelement liegt. Die Summanden in den weiteren Zeilen der Gleichung ergeben sich in analoger Weise. Mit der üblichen Konvention, dass das „leere Produkt" (dessen Untergrenze größer ist als die Obergrenze) den Wert 1 hat, gilt also für den genannten Sonderfall

$$\mathit{rg}_f(i_1, \ldots, i_n) = \sum_{k=1}^{n} \left((i_k - 1) \cdot \prod_{j=k+1}^{n} e_j \right)$$

Der Wert für den allgemeinen Fall mit Untergrenzen $a_1, \ldots, a_n$, die verschieden von 1 sein können, ergibt sich daraus durch eine leichte Anpassung:

$$\mathit{rg}_f(i_1, \ldots, i_n) = \sum_{k=1}^{n} \left((i_k - a_k) \cdot \prod_{j=k+1}^{n} (e_j - a_j + 1) \right)$$

Dieser Wert muss zum Offset addiert werden. Dabei sind die Werte $i_1, \ldots, i_n$ der Indizes erst zur Laufzeit bekannt, alle anderen Werte in dieser Formel aber bereits zur Übersetzungszeit. Deshalb bietet sich folgende Umformung an:

$$\begin{aligned}
\mathit{offset}(f) + \mathit{rg}_f(i_1, \ldots, i_n) \quad = \quad & \left[\mathit{offset}(f) - \sum_{k=1}^{n} \left(a_k \cdot \prod_{j=k+1}^{n} (e_j - a_j + 1) \right) \right] \\
& + \sum_{k=1}^{n} \left(i_k \cdot \prod_{j=k+1}^{n} (e_j - a_j + 1) \right)
\end{aligned}$$

Den Anteil in eckigen Klammern nennt man $\mathit{offset}_0(f)$, das *fiktive Offset* des Felds, im Gegensatz zu $\mathit{offset}(f)$, dem *tatsächlichen Offset*. Das tatsächliche Offset ist die Adresse des Feldelements `f[`a_1`, ..., `a_n`]`, das fiktive Offset ist die Adresse des Feldelements `f[0, ..., 0]`. Wenn der Indexwert 0 in jeder der n Dimensionen erlaubt ist, ist das fiktive Offset die Adresse eines existierenden Feldelements, andernfalls ist es nur eine rein rechnerische Bezugsgröße.

Das fiktive Offset kann bereits zur Übersetzungszeit berechnet werden. Außerdem kann man zur Übersetzungszeit für jedes k das Produkt, mit dem i_k in der unteren Summe multipliziert wird, zu einer Konstanten c_k auswerten. Zur Laufzeit ist dann nur noch zu berechnen:

$$\mathit{offset}_0(f) + c_1 \cdot i_1 + \ldots + c_n \cdot i_n$$

3.6.2 Speicherbelegung für dynamische Felder

Ein Feld heißt *dynamisch*, wenn die Unter- und Obergrenzen seiner Indizes nicht vollständig zur Übersetzungszeit bekannt sind. Das ist möglich, wenn in der Vereinbarung des Felds als Grenzen nicht nur Zahlen angegeben werden dürfen, sondern Ausdrücke, in denen auch Variablen vorkommen können.

Imperative Programmiersprachen unterscheiden sich darin, welche Einschränkungen für solche Ausdrücke gelten. Zum Beispiel können dynamische Felder auf Fälle beschränkt sein, wo die Felder als formale Parameter einer Prozedur auftreten und die Indexgrenzen ebenfalls formale Parameter der Prozedur sind, so dass die Grenzen beim Prozedureintritt (nach Auswertung der aktuellen Parameter) bekannt sind. Manche Sprachen erlauben dynamische Felder ohne solche Einschränkungen.

Aber praktisch immer gilt, dass die Grenzen sich zur Laufzeit nicht mehr ändern können, und dass die Anzahl n der Dimensionen zur Übersetzungszeit bekannt ist. Die folgenden Überlegungen gehen von diesen beiden Annahmen aus.

In jedem Fall ist die Anzahl der benötigten Speicherzellen für ein dynamisches Feld erst zur Laufzeit bekannt, und damit ist ab dem zweiten derartigen Feld nicht einmal mehr die Anfangsadresse zur Übersetzungszeit bestimmbar.

Deshalb wird für jedes dynamische Feld im Segment ein Felddeskriptor gespeichert, dessen Größe nur von der Anzahl n der Dimensionen des Felds abhängt, aber nicht von den Unter- und Obergrenzen der Indizes. Der Felddeskriptor enthält

- eine Speicherzelle zur Speicherung des fiktiven Offsets $\mathit{offset}_0(f)$,
- eine Speicherzelle zur Speicherung der Größe $\mathit{größe}(f)$,
- für jedes $j \in \{1, \ldots, n\}$ drei Speicherzellen zur Speicherung der Untergrenze a_j und Obergrenze e_j und der „Spanne“ $d_j = e_j - a_j + 1$, also der Anzahl der erlaubten Indexwerte in der j-ten Dimension.

Dabei wird wie bisher vorausgesetzt, dass jeder dieser Werte in eine einzige Speicherzelle passt. Der Felddeskriptor belegt also $2+3n$ Speicherzellen, deren Anzahl zur Übersetzungszeit bekannt ist, deren Inhalt aber zur Übersetzungszeit noch unbekannt ist.

Wenn zur Laufzeit das Feld erzeugt wird, sind die Werte $a_1, e_1, \ldots, a_n, e_n$ sowie die nächste freie Adresse im Datenspeicher `store` bekannt. Zu diesem Zeitpunkt können also die Werte $\mathit{offset}_0(f)$, $\mathit{größe}(f)$, $d_1, \ldots, d_n$ berechnet und zusammen mit $a_1, e_1, \ldots, a_n, e_n$ im Deskriptor gespeichert werden, zusätzlich wird der Kellerpegel `T` um $\mathit{größe}(f)$ erhöht. Alle Werte im Felddeskriptor ändern sich während der Lebenszeit des Felds nicht mehr, deshalb müssen sie nur ein Mal für jedes Feld berechnet werden.

Für jeden Zugriff auf ein Feldelement `f[`i_1`, ..., `i_n`]` kann dann die Adresse aus den Indexwerten und den im Deskriptor gespeicherten Werten berechnet werden.

Um die Anzahl der arithmetischen Operationen, die dafür zur Laufzeit ausgeführt werden müssen, möglichst gering zu halten, hat sich die Umwandlung der vorigen Formel nach dem Horner-Schema bewährt: $\mathit{offset}_0(f) + [(\ldots((i_1 \cdot d_2 + i_2) \cdot d_3 + i_3) \cdot d_4 + \ \ldots) \cdot d_n + i_n]$.

Zur Berechnung der Adresse von Feldelementen werden aus dem Deskriptor nur $\mathit{offset}_0(f)$ und die Spannen d_j benötigt. Die übrigen Werte im Deskriptor werden benötigt, um zur Laufzeit die Einhaltung der Indexgrenzen überprüfen zu können und um den vom Feld belegten Speicher wieder freigeben zu können, wenn die Lebenszeit des Felds endet.

3.6.3 Speicherbelegung für Verbunde

Verbunde (records) werden ähnlich wie statische Felder angelegt. Die folgenden Unterschiede sind aber zu beachten:

- Die Berechnung der Größe eines Verbunds ist etwas komplizierter als bei einem Feld, weil die Komponenten von unterschiedlichem Typ sein können.
- Die Varianten-Verbunde von Sprachen wie Pascal oder Modula (siehe unten) benötigen die bereits eingeführte Übersetzung von bedingten Anweisungen.
- Manche Sprachen lassen zu, dass die Namen von Verbund-Komponenten auch außerhalb des Verbunds als Namen von (anderen) Variablen vereinbart werden. In dem Fall muss der Parser zwischen den beiden Verwendungen gleicher Bezeichner unterscheiden können.

Beispiel eines Varianten-Verbunds à la Modula:

```
TYPE Koordinaten =
     RECORD CASE Art: Koordinatenart
            OF   kartesisch:           x, y:     REAL
             |   polar:                r, theta: REAL
            END
     END
```

Werte dieses Typs können mit drei Speicherzellen dargestellt werden: die erste enthält einen der Werte `kartesisch` oder `polar`, die anderen enthalten entweder die Werte von `x` und `y` oder die Werte von `r` und `theta`.

Varianten-Verbunde haben einen ähnlichen Zweck wie Unterklassen in objektorientierten Programmiersprachen.

3.6.4 Speicherbelegung für Zeiger und dynamische Datenstrukturen

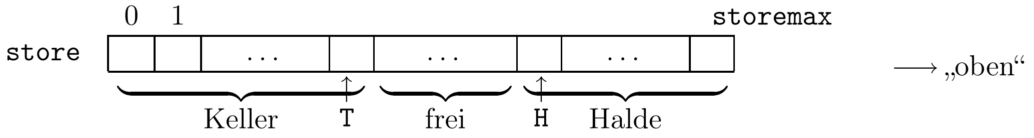
Bisher sind wir davon ausgegangen, dass jedes Datenobjekt, das zur Laufzeit entsteht, durch eine Vereinbarung bekannt gemacht wird. Die Vereinbarung ordnet dem Datenobjekt einen Namen zu, dem zur Übersetzungszeit eine oder mehrere Speicherzellen im Keller des Datenspeichers `store` zugeordnet werden können.

Imperative Programmiersprachen erlauben darüber hinaus oft die Erzeugung von sogenannten anonymen Datenobjekten, denen kein Name im Programmtext entspricht. Der Zugriff auf solche anonymen Datenobjekte ist mit Hilfe von Verweistypen möglich, deren Werte „Verweise“ oder „Zeiger“ (pointer) auf Datenobjekte sind. Wenn diese Verweistypen für Variablen und Komponenten von Feldern und Verbunden erlaubt sind, können damit Netze von miteinander verketteten anonymen Datenobjekten aufgebaut werden, für die nur gewissen „Einstiegspunkten“ Namen durch Vereinbarungen zugeordnet sind.

Die Anzahl der anonymen Datenobjekte, die von diesen „Einstiegspunkten“ aus erreichbar sind, kann sich während der Laufzeit stark verändern. Obendrein kann ihre Lebenszeit die Lebenszeit der Inkarnation überdauern, in der sie erzeugt werden. Beide Gründe sprechen dagegen, anonyme Datenobjekte in den Prozedursegmenten im Keller des Datenspeichers `store` zu speichern.

Die Sprache I sei um eine Anweisung `CREATE` *Name* erweitert, mit der einer Variablen ein Verweis auf ein neues anonymes Datenobjekt zugewiesen wird. Diese Anweisung entspricht `ALLOCATE` in PL/1, `new` in Pascal, `malloc` in C. Für die Freigabe des Speicherplatzes für ein anonymes Datenobjekt kann entweder eine Anweisung wie `FREE` in PL/1, `dispose` in Pascal, `free` in C hinzugenommen werden, oder es wird wie in Java der Implementierung überlassen, den Speicherplatz von anonymen Objekten freizugeben, auf die nicht mehr zugegriffen werden kann. Eine solche automatische Freigabe vom Speicher wird *Speicherbereinigung* (garbage collection) genannt.

Zur Speicherung von dynamisch angelegten anonymen Objekten benutzt die *MI*-Maschine einen neuen Speicherbereich namens *Halde* (heap). Die Halde wird wie der Keller im Datenspeicher `store` angelegt, aber am anderen Ende. Ein neues Adressregister `H` hat als Inhalt die Adresse der ersten Zelle der Halde. Im Befehl `RST` wird es mit `storemax + 1` initialisiert.



Der Befehlssatz der Maschine *MI* wird um einen Befehl `NEW` erweitert. Bei seiner Ausführung wird erwartet, dass die obersten beiden Speicherzellen des Kellers wie folgt belegt sind:

`store[T-1]`	Adresse für den Zeiger auf das zu erzeugende Datenobjekt
`store[T]`	Größe des zu erzeugenden Datenobjekts

`NEW`	`if (T+store[T] < H) {H:=H-store[T];  store[store[T-1]]:=H;  T:=T-2}` `else error("store overflow")`

Zusätzlich müssen alle Befehle, die den Kellerpegel `T` um einen Wert i erhöhen, um den Test `T+`i`<H` erweitert werden. Aber das ist genau genommen keine Erweiterung, sondern wäre auch in der bisherigen Form der abstrakten Maschine schon nötig mit dem Test `T+`$i \leq$ `storemax`.

Ob nun in der Variante mit oder ohne Halde, die Maschine kann so modifiziert werden, dass dieser Test nicht von jedem kellernden Befehl durchgeführt werden muss. Voraussetzung dafür ist, dass zur Übersetzungszeit zu jedem Prozedursegment eine Obergrenze für seine Größe ermittelt werden kann. Für die Sprache I ist das möglich, weil der lokale Keller im Segment nur für die Auswertung von Ausdrücken gebraucht wird und die maximale Anzahl der dabei belegten Kellerzellen anhand der Postfixform des Ausdrucks vorberechnet werden kann.

Der *MI*-Befehl `CAL` erhält dann als drittes Argument diese (zur Übersetzungszeit bekannte) Obergrenze für die aufgerufene Prozedur. Die *MI*-Maschine verwendet ein weiteres Adressregister `E` (extreme stack), dessen Inhalt die Adresse ist, die das oberste Segment maximal erreichen kann. Bei der Ausführung des `CAL`-Befehls wird `E` mit Hilfe des dritten Arguments passend für das neu erzeugte Segment besetzt und der bisherige Inhalt von `E` so im Segmentdeskriptor gesichert, dass er bei Ausführung des Befehls `RET` restauriert werden kann. Wenn der `CAL`-Befehl testet, dass `E<H` für den neuen Wert von `E` gilt, können alle anderen Befehle, die den Kellerpegel `T` erhöhen, ohne die entsprechenden Tests ausgeführt werden.

3.7 Prozeduren

3.7.1 Statische vs. dynamische Sichtbarkeitsregeln

Im Abschnitt über die Prozedursegmente wurde bereits erläutert, dass für Programmiersprachen, die nicht-lokale Variablen erlauben, die Regeln für den Zugriff auf diese Variablen festgelegt werden müssen. Wenn sich diese Regeln nur nach dem statischen Programmtext richten, nennt man sie *statische* oder *lexikalische Sichtbarkeitsregeln*, wenn sie von der dynamischen

Ausführung des Programms abhängen, nennt man sie *dynamische Sichtbarkeitsregeln.* Das Überschattungsprinzip für die Blockschachtelung ist eine weit verbreitete statische Sichtbarkeitsregel.

Eine dynamische Sichtbarkeitsregel, die einigen Dialekten funktionaler Programmiersprachen zugrunde lag, besagt, dass ein Name sich immer auf die zuletzt erzeugte Inkarnation eines Datenobjekts dieses Namens bezieht, deren Lebenszeit noch nicht abgelaufen ist.

Das folgende Beispiel illustriert den Unterschied.

```
PROGRAM P;
   VAR x;

   PROCEDURE R;
   BEGIN  WRITE x  END;

   PROCEDURE Q;
      VAR x;
   BEGIN  x:=2;  CALL R  END;

BEGIN
   x:=1;  CALL Q
END.
```

Bei der Ausführung des Programms P entsteht eine Inkarnation P' mit einer Inkarnation der Variablen x, die den Wert 1 hat. Beim Aufruf der Prozedur Q entsteht eine Inkarnation Q' mit einer Inkarnation der Variablen x, die den Wert 2 hat. Schließlich entsteht noch eine Inkarnation R', die auf ein nicht-lokales x zugreift.

Die zuletzt erzeugte Inkarnation von x ist die von Q' mit Wert 2. Mit der obigen dynamischen Sichtbarkeitsregel gibt das Programm den Wert 2 aus.

Nach dem Überschattungsprinzip für die Blockschachtelung ist die im Hauptprogramm vereinbarte Variable x in der Prozedur Q überschattet, aber überall sonst ist sie sichtbar, insbesondere in der Prozedur R. Mit dieser statischen Sichtbarkeitsregel gibt das Programm den Wert 1 aus.

Die dynamische Sichtbarkeitsregel ist einfach zu implementieren. Wenn in den Prozedursegmenten auch die Namen der Variablen repräsentiert sind, braucht man nur die Kette der dynamischen Vorgänger zurückzuverfolgen, bis man ein Segment erreicht, in dem der gesuchte Name vorkommt. Verweise auf die statischen Vorgänger werden überhaupt nicht benötigt.

Die Verweise auf die statischen Vorgänger in den Prozedursegmenten und die Berechnung der Stufen von Prozeduraufrufen, mit deren Hilfe diese Verweise zur Laufzeit bestimmt werden, war nur dazu erforderlich, die statische Sichtbarkeitsregel der Sprache I zu implementieren.

Obwohl dynamische Sichtbarkeitsregeln einfacher zu implementieren sind, besteht mittlerweile allgemeiner Konsens darüber, dass sie nicht wünschenswert sind. Um die Wirkung der Prozedur R zu verstehen, muss man mit einer dynamischen Sichtbarkeitsregel alle Stellen im Programm berücksichtigen, an denen die Prozedur aufgerufen wird. In einem großen Programm kann es sehr viele solche Stellen geben. Mit einer statischen Sichtbarkeitsregel braucht man dagegen nur die eine Stelle im Programm zu berücksichtigen, an der die Prozedur vereinbart ist.

Für imperative Programmiersprachen waren von Anfang an statische Sichtbarkeitsregeln üblich. Andere Paradigmen haben früher auch dynamische Sichtbarkeitsregeln verwendet, sind aber für die modernen Sprachen dieser Paradigmen zu statischen Sichtbarkeitsregeln übergegangen.

Zum Abschluss sei noch ein komplexeres Beispiel gegeben, an dem man den Unterschied zwischen statischer und dynamischer Sichtbarkeitsregel studieren kann:

Beispiel.

```
PROGRAM P;
   VAR x;

   PROCEDURE Q;

         PROCEDURE R;
            VAR x;
         BEGIN  x:=3;  WRITE x;  CALL Q  END;

   BEGIN
      IF x>0 THEN BEGIN  x:=0;  CALL R  END
   END;

BEGIN
   x:=1;  WRITE x;  CALL Q;  WRITE x
END.
```

Die Ausgabe ist im einen Fall 1, 3, 0, im anderen Fall 1, 3, 3, 3, 3,... ■

3.7.2 Speicherbelegung für Prozedursegmente

Für weniger einfache Sprachen als *I* muss der Segmentdeskriptor neben DL (dynamic link), RA (return address), und SL (static link) noch folgende Daten enthalten:

- Ist die Prozedur eine Funktion, dann wird am Anfang des Segments eine Speicherzelle FV (function value) für den Rückgabewert der Funktion angelegt. FV belegt unbedingt die erste Speicherzelle des Segments. So kann der Befehl `RET` bei Beendigung einer Inkarnation der Funktion das Segment ab der zweiten Speicherzelle „abräumen“, aber den Rückgabewert auf dem Keller lassen. Dies ermöglicht eine Weiterverarbeitung dieses Werts in einem zusammengesetzten Ausdruck oder durch einen Sprung- oder sonstigen Befehl ohne Sonderbehandlung.
- Wenn die Maschine das Adressregister `E` (extreme stack) zur effizienteren Erkennung von Speicherüberläufen verwendet, kommt eine Zelle EP (extreme stack pointer) hinzu, die den Wert des Registers `E` für den dynamischen Vorgänger des Segments enthält.
- Wenn die Prozedur formale Parameter hat, wird für jeden davon eine Speicherzelle angelegt, in die beim Aufruf die Werte der aktuellen Parameter gespeichert werden.

Ob man auch die lokalen Variablen zum „Segmentdeskriptor“ zählt, ist eine Frage des Sprachgebrauchs. Der Segmentdeskriptor ist der Teil des Segments, in dem Verwaltungsdaten gespeichert sind. Die Parameter kann man als solche ansehen, weil sie beim Anlegen des Segments besonders behandelt werden müssen. Sobald das Segment angelegt ist, unterscheiden sich die formalen Parameter aber nicht von lokalen Variablen, so dass es künstlich wäre, die einen zum Deskriptor zu zählen und die anderen nicht.

Unabhängig von der Terminologie hat ein Segment jetzt folgenden Aufbau:

...	FV	DL	RA	SL	EP	Parameter	lokale Variablen	lokaler Keller	⟷	...
	↑ B							↑ T		↑ E

Die Anzahl der Speicherzellen bis einschließlich zu den lokalen Variablen ist für jede Prozedur fest und zur Übersetzungszeit bekannt. Wenn die Sprache dynamische Felder erlaubt, liegen die Felddeskriptoren im Bereich der lokalen Variaben, aber die Speicherzellen für die Feldelemente in einem Bereich zwischen den lokalen Variablen und dem lokalen Keller. Die Anzahl der Speicherzellen für die Feldelemente ist zwar nicht zur Übersetzungszeit bekannt, aber sobald sie zur Laufzeit feststeht, ändert sich diese Anzahl nicht mehr.

3.7.3 Wert- vs. Referenzparameter

Viele imperative Programmiersprachen bieten mehr als einen Mechanismus zur Parameterübergabe an.

Bei der *Wertübergabe* werden die formalen Parameter wie lokale Variablen behandelt, die beim Prozeduraufruf mit den Werten der aktuellen Parameter initialisiert werden. Mit diesem Mechanismus ist es nicht möglich, eine Prozedur zu programmieren, die die Werte der aktuellen Parameter verändert.

Um zu ermöglichen, eine solche Wirkung auf die aktuellen Parameter zu programmieren, gibt es andere Mechanismen der Parameterübergabe, von denen die *Referenzübergabe* am weitesten verbreitet ist. Bei diesem Mechanismus stehen die formalen Parameter für die selben Speicherzellen, für die die aktuellen Parameter stehen. Enthält der Anweisungsteil der Prozedur eine Zuweisung an einen formalen Parameter, wirkt diese also auch auf den aktuellen Parameter.

Die Implementierung der Wertübergabe erfordert keine anderen Techniken als die bisher behandelten. Die Referenzübergabe wird so implementiert, dass die Speicherzellen der formalen Parameter im Prozedursegment mit den Adressen der aktuellen Parameter initialisiert werden statt mit den Inhalten der Speicherzellen mit diesen Adressen. Die semantische Analyse sorgt dann dafür, dass auf die Vorkommen eines formalen Parameters im Anweisungsteil ein Dereferenzierungsoperator angewandt wird. Die Befehle der abstrakten Maschine müssen also unter anderem um solche Dereferenzierungsoperatoren erweitert werden.

In welchen Fällen implizite Dereferenzierungsoperatoren hinzugefügt werden, hängt von der Programmiersprache ab. In Pascal und Modula muss in einer Prozedurvereinbarung für jeden formalen Parameter angegeben werden, welcher der beiden Mechanismen dafür verwendet werden soll, so dass aus dieser Angabe hervorgeht, ob Vorkommen dieses Parameters implizit dereferenziert werden oder nicht. In Java ist für jeden Typ festgelegt, welcher Mechanismus für einen formalen Parameter dieses Typs verwendet wird, so dass die Frage der Dereferenzierung durch die Analyse der Typen beantwortet wird. In C ist die Implementierung der Parameterübergabe im Gegensatz zu diesen Sprachen nicht transparent, sondern muss mit Adress- und Dereferenzierungsoperatoren im Quellprogramm programmiert werden, so dass gar keine implizite Dereferenzierung vorkommt.

3.7.4 Endrekursion

Ein rekursiver Aufruf einer Prozedur P heißt *endrekursiv* (oder auch *restrekursiv*), wenn er die letzte Anweisung vor der Rückkehr aus der Prozedur P ist. Dabei bezieht sich „letzte“ Anweisung nicht auf die Stelle im Programmtext, sondern auf den Kontrollfluss. In Verzweigungen kann also jeder Zweig einen endrekursiven Prozeduraufruf enthalten. Im Fall einer Funktion bedeutet endrekursiv, dass der Aufruf auch nicht Argument eines Operators oder einer Funktion ist.

Wird ein endrekursiver Aufruf zur Laufzeit ausgeführt, so geschieht das in einer Inkarnation P' und hat zur Folge, dass eine neue Inkarnation P'' entsteht. Oberhalb des Segments von P' wird somit das Segment von P'' gekellert. Wenn die Inkarnation P'' beendet und ihr Segment wieder abgräumt ist, passiert in der Inkarnation P' nichts weiter, als dass ihr Segment ebenfalls abgeräumt wird.

Es ist also gar nicht notwendig, das Segment von P' während der Lebensdauer von P'' aufzubewahren. Man kann es abräumen, bevor das Segment von P'' angelegt wird. Nun haben aber die Segmente der beiden Inkarnationen der selben Prozedur den gleichen Aufbau. Statt zuerst das Segment von P' abzuräumen und danach das Segment von P'' (mit völlig gleichen Verwaltungsdaten!) anzulegen, kann man auch gleich das Segment von P' für P'' wiederverwenden.

Da der lokale Keller bei einem endrekursiven Aufruf sowieso leer ist, erreicht man diese Wiederverwendung im wesentlichen, indem man den Aufruf nicht in einen `CAL`-Befehl übersetzt, sondern in einen `JMP`-Befehl.

Wenn die Prozedur keine Parameter und keine lokalen Variablen hat, wird statt eines Befehls `CAL 1 a`, der durch die Übersetzungsregel entstehen würde, der Befehl `JMP a` erzeugt. Alles, was der `CAL`-Befehl außer der Änderung des Programmzählers bewirken würde, ist nämlich noch aus dem Segment von P' vorhanden.

Hat die Prozedur keine Parameter, aber lokale Variablen, wird der Befehl $\mathtt{JMP\ a}'$ erzeugt mit $\mathtt{a}' = \mathtt{a} + 1$. Dadurch wird der `INC`-Befehl übersprungen, mit dem der Code für die Prozedur beginnt und das Segment für die lokalen Variablen erweitert würde, für die das Segment aber bereits Speicherzellen enthält. Dass die Speicherzellen der Variablen noch die Werte der Inkarnation P' enthalten, stört nicht, da die Inkarnation P'' sowieso von undefinierten Inhalten ausgehen muss.

Bei Prozeduren mit Parametern werden zusätzlich die Werte der aktuellen Parameter, egal ob Adressen oder Inhalte von Speicherzellen, in die Speicherzellen der formalen Parameter kopiert und so die für P' gültigen Werte mit den für P'' gültigen überschrieben.

Diese Optimierung heißt *Endrekursions-Optimierung* (oder *Restrekursions-Optimierung,* tail recursion optimization). Wird sie eingesetzt, ensteht aus einem Quellprogramm, das eine Wiederholung durch Rekursion (mit endrekursiven Aufrufen) ausdrückt, praktisch der gleiche Code wie aus einem Quellprogramm, das die gleiche Wiederholung mit `while` oder einem anderen Iterationskonstrukt ausdrückt.

Die Optimierung ist also sehr mächtig. Da Rekursion in funktionalen und logischen Programmiersprachen eine zentrale Rolle spielt, trägt die Optimierung wesentlich zur effizienten Übersetzung von Sprachen dieser Paradigmen bei. Für imperative Sprachen wird sie dagegen nicht immer implementiert, obwohl sie dafür ebenfalls nützlich ist.

Eine Verallgemeinerung der Endrekursions-Optimierung namens *tail call optimization* überträgt die Idee der Wiederverwendung von Segmenten auf nicht-rekursive Aufrufe.

4 Übersetzung funktionaler Programmiersprachen

4.1 Konzepte funktionaler Programmiersprachen

Die Hauptunterschiede zwischen imperativen und funktionalen Programmiersprachen liegen in der Auslegung der Variablen und in der Spezifikation des Kontrollflusses.

Auslegung der Variablen. In imperativen Programmiersprachen sind Variablen Namen für Speicherzellen, in funktionalen Sprachen für Werte. In imperativen Programmiersprachen können die Werte von Variablen, also die Inhalte ihrer Speicherzellen, mittels Zuweisungen geändert werden. Die Wertzuweisung widerspricht dem Grundprinzip der funktionalen Programmiersprachen, obwohl sie in einigen funktionalen Programmiersprachen vorhanden ist.

Spezifikation des Kontrollflusses. Der Kontrollfluss wird in imperativen Programmen explizit durch Anweisungen spezifiziert. In funktionalen Programmen wird er implizit bestimmt. Jeder funktionalen Programmiersprache liegt ein Auswertungsprinzip zugrunde: *Auswertung in applikativer Reihenfolge*, *Auswertung in normaler Reihenfolge* oder *verzögerte Auswertung.*

Ein weiterer, weniger wesentlicher Unterschied ist, dass die imperativen Programmiersprachen Ausdrücke und Anweisungen, die funktionalen Programmiersprachen nur Ausdrücke aufweisen.

Die funktionale Programmierung wurde mit der Programmiersprache Lisp, die wie FORTRAN anfangs der 60er Jahre definiert wurde, eingeführt. Seit Ende der 70er und Anfang der 80er Jahre sind in deutlichem Abstand zu Lisp mehrere andere funktionale Programmiersprachen definiert worden.

Einige funktionale Programmiersprachen und ihre Abstammung:

```
ca. 1960        1965          1975           1985              1990/1999
    Lisp -----> ISWYM -------> ML ------> Standard ML ---> Haskell/Haskell 98
     | |                                                     ^  ^
     | |                                                     |  |
     | +------> SASL ---------------------> Miranda ---------+  |
     |                                                          |
     +-----------------------> Scheme --------------------------+
                               |
                               +--------> Common Lisp
                                             |
                                             v
                                            CLOS
```

Die Programmiersprache ISWYM (I See What You Mean) wurde nicht implementiert.

Es wird zwischen *reinen* und *nicht-reinen* funktionalen Programmiersprachen unterschieden. Rein ist eine funktionale Programmiersprachen ohne Zuweisungen sowie ohne Sprachkonstrukte, die eine Wertänderung als Nebeneffekt haben. Lisp und ML sind nicht rein, Miranda und Haskell dagegen sind rein. Reine funktionale Programmiersprachen haben den Vorteil, einfach formalisierbar zu sein, was oft das Verständnis von Programmen und ihre (automatische oder nicht-automatische) Überprüfung erleichtert. Mit nicht-reinen funktionalen Programmiersprachen lassen sich jedoch einige Verfahren einfacher und natürlicher darstellen als mit rein funktionalen Programmiersprachen.

Moderne funktionale Programmiersprachen bieten folgende Konzepte an:

- Funktionsdefinitionen mittels Gleichungen oder λ-Abstraktion
- Funktionen höherer Ordnung
- Strukturierte (oder algebraische) Typen und Mustererkennung
- Parametrisierte Typen und Polymorphismus

4.1.1 Funktionsdefinitionen mittels Gleichungen oder λ-Abstraktion

In einer funktionaler Programmiersprache werden Prozeduren mittels Funktionsdefinitionen spezifiziert. Funktionsdefinitionen sind entweder *Gleichungen,* wie z. B.

```
f(x1, x2, x3) = x1 + (x2 * x3)
```

im Folgenden in einem nicht weiter formaliserten Pseudocode als

```
f x1 x2 x3 = x1 + (x2 * x3)
```

geschrieben oder λ-*Abstraktionen,* d. h. Spezifikationen von anonymen (d. h. namenlosen) Funktionen, wie z. B.

```
lambda x1 x2 x3 . x1 + (x2 * x3)
```

Diese λ-Definition definiert die Funktion, die `(x1, x2, x3)` auf `x1 + (x2 * x3)` abbildet.

Rekursive Funktionsdefinitionen sind zulässig. Sie sind für die Turing-Vollständigkeit einer funktionalen Sprache notwendig. Die folgende Funktion ist rekursiv:

```
f x  = if   x > 1
       then x * f (x - 1)
       else 1
```

Ein weiteres Beispiel für rekursive Funktionen ist das folgende:

```
g y = if   y > 0
      then y + h y
      else 1
h x = x * g (x - 1)
```

Man sagt, dass `g` und `h` *simultan-rekursive Funktionen* sind.

Eine Definition, in der links nur ein Bezeichner ohne Argumente vorkommt, heißt *Wertdefinition.* Manche funktionalen Sprachen erlauben das Argument `()` für die Definition von 0-stelligen Funktionen, wenn diese von Wertdefinitionen unterschieden werden sollen.

Ein funktionales Programm besteht in einer endlichen Menge von Funktionsdefinitionen. In funktionalen Programmiersprachen sind auch *lokale Funktionsdefinitionen* möglich, d. h. Funktionsdefinitionen, die nur innerhalb der Definition einer anderen Funktion gelten.

Aus einer Funktion `f` einer Stelligkeit $n \geq 2$ können Funktionen einer Stelligkeit m (mit $1 \leq m < n$) definiert werden. Zum Beispiel führt die folgende Funktionsdefinition

```
f x y = x + y          (*  f :: Zahl x Zahl ---> Zahl  *)
```

zur Funktionsdefinition

```
lambda y . f 1 y       (*  (lambda y . f 1 y) :: Zahl ---> Zahl  *)
```

Die Funktion `lambda y . f 1 y` könnte man `inc1` (Inkrementieren mit 1) benennen.

Dieses Beispiel ist Vorbild eines allgemeinen Prinzips: Die λ-Abstraktion ermöglicht, durch die Auswertung eines Teils der Argumente einer mehrstelligen Funktion eine andere Funktion mit geringeren Stelligkeit zu definieren. Da sich diese Möglichkeit für die Programmierpraxis als nutzlich erwiesen hat, lassen funktionale Programmiersprachen die Auswertung von einem Teil der Argumente einer Funktion zu. Im Gegensatz zu imperativen Programmiersprachen, ist also in einer funktionalen Programmiersprache beim Aufruf einer Funktion die Anzahl der übergebenen Parameter nicht immer gleich. Die Übersetzung funktionaler Programmiersprache trägt dieser Tatsache Rechnung.

4.1.2 Funktionen höherer Ordnung

Unter *Funktionen höherer Ordnung* versteht man Funktionen, die als Argumente oder als Ergebnis Funktionen haben.

Ein traditionelles und wichtiges, in der Praxis nützliches Beispiel einer Funktion höherer Ordnung ist die Funktion `map` (d. h. „abbilden"), die als Argumente eine Funktion `f` und eine Liste `[a1, ..., an]` erhält, und als Ergebnis die Liste `[f(a1), ..., f(an)]` liefert. Eine Definition der `map`-Funktion ist:

```
map function list = if   leer list
                    then []
                    else cons (function (first list))
                              (map function (rest list))
```

wobei `[]` die leere Liste bezeichnet, und `cons` die Funktion zum Einfügen eines Elements am Anfang einer Liste. Der Ausdruck `map quadrat [0,1,2,3]` hat also den Wert `[0,1,4,9]`.

4.1.3 Strukturierte Typen und Mustererkennung

Strukturierte Datentypen. Moderne funktionale Programmiersprachen geben dem Programmierer die Möglichkeit, *Aufzählungs-* sowie *strukturierte Typen* mittels *Typdefinitionen* zu spezifizieren. Anstatt von Typen spricht man auch von *Datentypen*, anstatt von strukturierten Typen von *algebraischen Typen*, anstatt von Typdefinitionen, von *Typvereinbarungen.* Typen, die in einem Programm definiert werden, können in diesem Programm wie die vordefinierten (oder konkreten) Typen der Programmiersprache, z. B. die Zahlen, benutzt werden.

Ein Typ ist eine (endliche oder unendliche) Menge von Werten versehen mit Funktionen, die *Operationen* genannt werden, um diese Werte zu bearbeiten. Ein Beispiel stellt der Typ „Liste" dar, der die folgenden Operatoren besitzt: Ein erster Operator zum Test auf eine leere Liste, ein zweiter Operator zum Einfügen eines Elementes an den Anfang einer Liste, ein dritter Operator zum Zugriff auf das erste Element einer Liste und ein vierter Operator zum Entfernen des ersten Elements der Liste.

Die folgende Typdefinition in einem nicht weiter formalisierten informellen Pseudo-Code spezifiziert einen Aufzählungstyp:

```
Werktag  :: Montag | Dienstag | Mittwoch | Donnerstag  | Freitag
```

Die folgende Typdefinitionen spezifizieren strukturierte Typen:

```
       Paar :: ZahlPaar Zahl Zahl | BoolPaar Bool Bool
BinBaum 'x :: Blatt 'x | InKnoten (BinBaum 'x) (BinBaum 'x)
```

Die Variable `'x` nimmt ihre Werte über Typen, sie wird Typvariable genannt. Die Schreibweise `'`*Name* für eine Typvariablen stammt aus der Programmiersprache ML. Sie ermöglicht, Typvariablen von sonstigen Variablen zu unterscheiden.

Typsynonym. Zusätzlich zu den Typdefinitionen geben viele funktionalen Programmiersprachen die Möglichkeit zur Spezifikation von *Typsynonymen* wie etwa:

```
Arbeitstag = Werktag
      Zahl = Integer
```

Mustererkennung. Um die Definition von Funktionen über strukturierten Typen — wie etwa `Paar` oder `BinBaum 'x` — zu erleichtern, bieten moderne funktionale Programmiersprachen die *Mustererkennung* (pattern matching) an, womit z. B. die Function `map` vom vorangehenden Abschnitt:

```
map function list = if   leer list
                    then []
                    else cons (function (first list))
                              (map function (rest list))
```

auch wie folgt spezifiziert werden kann:

```
map function []    = [];
map function (f:r) = cons (function f)
                          (map function r)
```

In einer solchen Definition muss die Gleichungsmenge alle möglichen Fälle abdecken. Das Finden eines zum aktuellen Parameter des auszuwertenden Ausdrucks passenden Musters erfolgt durch die sequenzielle Suche der Gleichungsmenge von oben nach unter. Ein zum aktuellen Parameter passendes gefundene Muster schließt weitere Gleichungen aus.

Bemerkung. Die Bereicherung einer funktionalen Programmiersprache um die Mustererkennung erhöht ihre Ausdrucksfähigkeit nicht, sondern verbessert lediglich die Bequemlichkeit der Programmierung in dieser Sprache. ■

4.1.4 Parametrisierte Typen und Polymorphismus

Mit Hilfe von *Typvariablen* lassen sich *parametrisierte Typen* definieren, wie z. B. die vorangehende Definition des Typs `BinBaum 'x`, die Binärbäume bezüglich eines beliebigen Typs `'x` definiert.

Parametrisierte Typen ermöglichen die Definition von *polymorphen Funktionen,* d. h. von Funktionen, die auf Objekten unterschiedlicher Typen anwendbar sind. Die folgende Funktion, die die Anzahl von Knoten eines Binärbaums berechnet, ist z. B. polymorph:

```
knotenzahl ::  BinBaum 'x -> Zahl     (* Typ der Funktion knotenzahl *)

knotenzahl baum = if   blatt baum     (* Def. der Funktion knotenzahl *)
                  then 1
                  else 1 + knotenzahl (linkerteil baum)
                         + knotenzahl (rechterteil baum)
```

wobei drei Operatoren, `blatt`, `linkerteil` und `rechterteil`, für den polymorphen Typ `BinBaum` angenommen werden. Typdefinitionen für diese Operatoren können z. B. wie folgt gegeben werden:

```
     blatt  :: BinBaum 'x -> Bool
 linkerteil :: BinBaum 'x -> BinBaum 'x
rechterteil :: BinBaum 'x -> BinBaum 'x
```

Moderne funktionale Sprachen erkennen möglichst zur Übersetzungszeit automatisch die Typen, die für für gegebene Funktionsdefinition möglich sind, und überprüfen, ob die in einem Programm angegebenen Typen möglich sind. Man spricht von *Typinferenz, Typprüfung* und *wohltypisierten* Programme. Zur Typinferenz und -prüfung wird ein *Typsystem* verwendet, d. h. ein Regelwerk, das Bestandsteil der Spezifikation der Programmiersprache ist.

4.2 Auswertungsansätze und Sichtbarkeitsregeln

Ein funktionales Programm wird dadurch ausgeführt, dass ein Ausdruck wie etwa `quadrat (quadrat 3)` ausgewertet wird. Die Auswertung eines Ausdrucks besteht aus einer Folge von Reduktionen. Eine Reduktion ist die Ersetzung eines sogenannten *reduzierbaren Ausdrucks* (reducibe expression, kurz *redex*) durch seine Definition. Die Auswertung eines komplexen Ausdrucks ist beendet, sobald der Ausdruck keinen reduzierbaren Teilausdruck mehr enthält.

Die Auswertung von funktionalen Programmen wird dadurch bestimmt, in welcher Reihenfolge die Teilausdrücke, auch Komponenten genannt, von zusammengesetzten Ausdrücke ausgewertet werden. Da zusammengesetzte Ausdrücke durch Funktionsanwendung enstehen, bestimmt die Reihenfolge der Auswertung der Komponenten A_1 und A_2 einer Funktionsanwendung A_1 A_2 die Art der Auswertung.

Es gibt zwei Grundansätze: die Auswertung in *applikativer Reihenfolge* (*innermost evaluation*) und die Auswertung in *normaler Reihenfolge* (*outermost evaluation*). Ein dritter Ansatz, die *verzögerte Auswertung* (*lazy evaluation*) hat die Vorteile der Auswertung in normaler Reihenfolge ohne ihre Nachteile.

4.2.1 Auswertung in applikativer Reihenfolge

Zur Auswertung einer Anwendung A_1 A_2 wird zunächst A_2 (gewöhnlich von links nach rechts) ausgewertet und dieser Wert wird an A_1 zu dessen Auswertung übergeben.

Betrachten wir die folgende Funktionsdefinition zur Berechnung des Quadrats einer Zahl:

```
quadrat x = x * x
```

Die Auswertung von `quadrat (quadrat 3)` in applikativer Reihenfolge besteht aus den folgenden Schritte:

```
 quadrat (quadrat 3)
   quadrat (3 * 3)
      quadrat 9
       9 * 9
         81
```

Der Vorteil dieses Auswertungsansatzes liegt darin, dass das Argument A_2 nur einmal ausgwertet wird, wie auch immer die Gestalt des definierenden Ausdrucks von A_1 sein mag.

Der Nachteil dieses Auswertungsansatzes liegt darin, dass Argumente auch dann ausgewertet werden, wenn ihre Werte nicht benötigt werden, wie z. B. bei der Auswertung von `const1 (fak c)` im folgenden Beispiel:

```
const1 x = 1
   fak x = if x = 1
           then 1
           else x * f (x - 1)
```

Ist der Wert von `c` eine große natürliche Zahl, dann werden bei einer Auswertung in applikativer Reihenfolge viele unnötige Auswertungen durchgeführt. Ist der Wert von `c` eine negative Zahl, dann terminiert die Auswertung von `fak c` nicht, folglich terminiert auch die Auswertung in applikativer Reihenfolge von `const1 fak c` nicht, obwohl der Wert dieses Ausdrucks, `1`, offensichtlich zu ermitteln ist.

4.2.2 Auswertung in normaler Reihenfolge

Zur Auswertung einer Anwendung A_1 A_2 wird zunächst A_1 (gewöhnlich von links nach rechts) ausgewertet, d. h. der Ausdruck A_1 wird durch seinen definierenden Ausdruck ersetzt, wobei A_2 den aktuellen Parameter liefert.

Die Auswertung von `quadrat (quadrat 3)` in normaler Reihenfolge besteht also aus den folgenden Schritten:

```
   quadrat (quadrat 3)
(quadrat 3) * (quadrat 3)
    (3 * 3) * (quadrat 3)
          9 * (quadrat 3)
          9 * (3 * 3)
          9 * 9
            81
```

Es ist bemerkenswert und charakteristisch für diese Auswertungsart, dass die Auswertung von `quadrat (quadrat 3)` in normaler Reihenfolge zur mehrfachen Auswertung von Teilausdrücken, hier zur doppelten Auswertung von `(quadrat 3)`, führt.

Ebenfalls charakteristisch für die Auswertung in normaler Reihenfolge ist, dass in unnötige Auswertungen von Teilausdrücken nicht stattfinden. Die Auswertung in normaler Reihenfolge von `const 1 (fak 1000)` mit dem Programm

```
const1 x = 1
   fak x = if x = 1
           then 1
           else x * (f x - 1)
```

führt zum Beispiel nicht zur Auswertung von `fak 1000`.

Die Vor- und Nachteile der Auswertung in normaler Reihenfolge sind symmetrisch zu denen der Auswertung in applikativer Reihenfolge. Während der Auswertung in normaler Reihenfolge wird ein Argument nur bei Bedarf ausgewertet, es kann aber mehrmals ausgewertet werden kann.

4.2.3 Verzögerte Auswertung

Die verzögerte Auswertung (lazy evaluation) kann als eine Verbesserung der Auswertung in normaler Reihenfolge angesehen werden, die die mehrfache Auswertung von Argumenten vermeidet. Dies wird dadurch erreicht, dass das mehrfache Auftreten eines Arguments erkannt und durch das Einsetzen von einem einzigen Vorkommen des definierenden Ausdrucks und die Nutzung von Zeigern auf dieses Vorkommen geteilt wird.

Die verzögerte Auswertung von `quadrat (quadrat 3)` besteht zum Beispiel aus den folgenden Schritten:

```
      quadrat (quadrat 3)

         _________
        /         \
   (quadrat 3) *   .

             ____
            /    \
           __     \
          / \      \
       (3 * .) *   .

               __
              / \
              9 * .

               81
```

Es ist bemerkenswert, dass die Auswertungen in applikativer und normaler Reihenfolge eines Ausdrucks in der Sprache dieses Ausdrucks dargestellt werden können, wogegen die Darstellung der verzögerten Auswertung auf Zeigern beruht, die in der Sprache des ausgewerteten Ausdrucks nicht vorhanden sind. Die verzögerte Auswertung wird unten im Abschnitt 4.4.3 (Auswertung eines Ausdrucks als Graphreduktion) präzise formalisiert.

4.2.4 Sichtbarkeitsregeln

Zu einem Auswertungsansatz gehört auch die Auswahl einer Sichtbarkeitsregel. Zur Erläuterung des Begriffes betrachten wir das folgende Beispiel:

```
let a fun x = x * (fun 1) ;  (* lokale Definitionen *)
          x = 0 ;
        f y = x * y
in a f 2
```

1. Gilt die Definition `x = 0` innerhalb der lokalen Definitionen und der definierenden Ausdrücke `x * (fun 1)` und `x * y`, dann ist der Wert des Ausdrucks `a f 2` wie folgt:

```
a f 2 = a fun x          wobei fun = f und x = 2
      = 2 * (f 1)
      = 2 * (0 * 1)      weil x = 0
      = 0
```

2. Eine andere (ebenfalls sinnvolle) Betrachtungsweise ist die folgende:

```
a f 2 = a fun x          wobei fun = f und x = 2
      = 2 * (f 1)
      = 2 * (2 * 1)      weil x = 2
      = 4
```

Offensichtlich sind in beiden Fälle verschiedenen Regeln angewendet worden, um die Bindung von `x` im Ausdruck `a fun x` zu bestimmen. Im ersten Fall wurde die Bindung von `x` statisch anhand der im Programm vorhandenen Gleichungen bestimmt. Im zweiten Fall wurde die Bindung von `x` dynamisch anhand des auszuwertenden Ausdrucks bestimmt.

Man spricht im ersten Fall von einer *statischen oder lexikalischen Sichtbarkeitsregel* und von *statischer oder lexikalischer Bindung.* Im zweiten Fall spricht man von einer *dynamischen Sichtbarkeitsregel* und von einer *dynamischen Bindung.*

Die statische Sichtbarkeitsregel ermöglicht eine Bindung zur Übersetzungszeit, daher der Name statische Bindung. Die dynamische Sichtbarkeitsregel dagegen verlangt, dass die Bindung erst zur Laufzeit ermittelt wird.

Welche Sichtbarkeitsregel soll gewählt werden? Wenn an der deklarativen Sicht der funktionalen Programmierung festgehalten wird, nachdem ein funktionales Programm den Kontrollfluss nicht explizit spezifiziert sondern vielmehr „zeitlose“ Gleichungen, dann ist die statische Sichtbarkeitsregel vom Vorteil.

Mit der statischen Sichtbarkeitsregel ist nämlich der Wert eines Vorkommens eines Ausdrucks A immer der gleiche, ganz gleich, in welchem Kontext (oder „Referenzbereich“) sich dieses Vorkommen befindet. Man sagt, dass die Programmiersprache *referentiell transparent* ist. Mit der dynamischen Sichtbarkeitsregel ist eine funktionale Programmiersprache nicht referentiell transparent.

Im vorangehenden Beispiel, bedeutet die statische Sichtbarkeitsregel, dass die zwei folgenden Programme äquivalent sind:

```
let a fun x = x * (fun 1) ;        let a fun x = x * (fun 1) ;
          x = 0 ;                            z = 0 ; (* z statt x*)
        f y = x * y                        f y = z * y
in a f 2                           in a f 2
```

Unter dynamischer Bindung sind sie es nicht.

4.2.5 Geltungsbereiche von Variablen bei statischer Sichtbarkeitsregel: Informelle Betrachtung

Wird die statische Sichtbarkeitsregel gewählt, wie es in der funktionalen Programmierung heute üblich ist, dann muss der Geltungsbereich einer Variablendefinition festgelegt werden. Im Folgenden wird eine solche Festlegung unter vielen betrachtet.

Betrachten wir einen ersten Fall:

let x = A_1 in A_2

Die Definition von `x` soll in A_1 sowie A_2 gelten.

Beispiel.

```
let x = 1 in x + x
```

Der Wert des Ausdrucks ist 2. ∎

Der Fall von mehereren lokalen Definitionen muss berücksichtigt werden:

`let x = 1 ;`	A_1
`    y = 2 * x`	A_2
`in  x + y`	A_3

Der Geltungsbereich von `x` umfasst A_1, A_2 und A_3. Nach diesem Prinzip ergibt sich, dass der Wert des folgenden Ausdrucks 4 ist:

```
let x = 1 in let y = x + 1 in y * y
```

Was ist aber der Wert des folgenden Ausdrucks?

$$\texttt{let x = }\underset{A_1}{\texttt{1}}\ \texttt{ in let x = }\underset{A_2}{\texttt{x + 1}}\ \texttt{ in }\underset{A_3}{\texttt{x * x}}\qquad(\star)$$

Der innere `let`-Ausdruck wird als neue Definition von `x` betrachtet, die die alte „überschattet“. Der Geltungsbereich des inneren `x` umfasst A_2 und A_3 und überschattet dort das äußere `x`. In anderen Worten kann diese Definition durch die Folgende ersetzt werden:

```
let x = 1  in let y = y + 1 in y * y
```

Der Wert des Ausdrucks $(\star)$ ist also undefiniert, weil `x` in A_2 keinen definierten Wert hat.

Die Semantik von geschatelten `let`-Ausdrücke kann auch anders definiert werden z. B. so, dass der Ausdruck $(\star)$ den Wert 4 erhält.

Geltungsbereiche von Variablen in Definitionen von Funktionen einer Stelligkeit größer oder gleich 1 werden ähnlich definiert:

$$\begin{array}{l}\texttt{let } \texttt{x}_1 \texttt{ = } A_1 \texttt{ ; } \ldots \texttt{ ; } \texttt{x}_{i-1} \texttt{ = } A_{i-1} \texttt{ ; } \texttt{x}_i \texttt{ = } A_i \texttt{ ; } \texttt{x}_{i+1} \texttt{ = } A_{i+1} \texttt{ ; } \ldots \texttt{ ; } \texttt{x}_n \texttt{ = } A_n \\ \texttt{in } A_{n+1}\end{array}$$

Der Geltungsbereich von $\texttt{x}_i$ $(i = 1, \ldots, n)$ ist $A_1, \ldots, A_{i-1}, A_i, A_{i+1}, \ldots, A_n, A_{n+1}$, solange keine neue Defintionen von $\texttt{x}_i$ in den A_j $(j = 1, \ldots, n+1)$ vorkommt.

Man sagt, dass eine Variable `x` in einem Ausdruck *frei vorkommt,* wenn dieser Ausdruck sich nicht im Geltungsbereiches einer Definition von `x` befindet.

Eine formale Definition von Geltungsbereiche wird im Abschnitt 4.3.2 „Geltungsbereiche von Variablen in F“ gegeben.

4.3 Die funktionale Programmiersprache F

Wie im Kapitel 2 mit der imperativen Programmiersprache I bedienen wir uns einer einfache funktionale Programmiersprache, F genannt, die die Hauptmerkmale funktionaler Programmiersprachen aufweist, um das Prinzip der Übersetzung einer funktionaler Programmierpsrachen an einem konkreten Beispiel erläutern zu können.

F weist die folgenden Einschränkungen auf:

- Lokale Definitionen sind in F möglich, jedoch eingeschränkt auf Definitionen von Ausdrücken ohne Parameter (Wertedefinitionen).
- F ermöglicht nicht die Definition von Aufzählungs- und strukturierten Typen. Folglich bietet F die Mustererkennung nicht an.
- Die λ-Abstraktion fehlt in F und F kann keine Funktionen höherer Ordnung behandeln.
- Die Übersetzung von F implementiert die Endrekursion nicht.

Ein F-Programm ist eine endliche Menge von Gleichungen, die eine besondere Gleichung (Hauptgleichung) der Gestalt `main = Ausdruck` enthält. `main` ist der (einzige) auszuwertende Ausdruck.

Bei Auswertung in applikativer Reihenfolge gibt es einen subtilen Unterschied zwischen Wertdefinitionen und Definitionen von 0-stelligen Funktionen, der bei verzögerter Auswertung entfällt. Da F auf der verzögerten Auswertung beruht, unterscheidet F wie Miranda oder Haskell nicht zwischen den beiden Arten von Definitionen. Andere Programmiersprachen wie etwa ML unterscheiden dazwischen. Sie verwenden Notationen wie `val x = quadrat 3` für Wertdefinitionen und `fun f() = quadrat 3` für die Definition von 0-stelligen Funktionen.

In F wird kein Unterschied zwischen rekursiven und nichtrekursiven lokalen Definitionen gemacht, die in manchen funktionalen Programmiersprachen wie etwa ML mittels `val` bzw. `val rec` unterschieden werden. Der Unterschied ist wohl für die Übersetzung vom Belang. Da aber F sowieso nur Wertdefinitionen als lokale Definitionen erlaubt, wäre diese Unterscheidungsmöglichkeit nicht sehr nützlich.

4.3.1 Syntax von F

Programm ::= *Definition* ";" { *Definition* ";"} .
Definition ::= *Variable* {*Variable*} "=" *Ausdruck* .
Lokaldefinitionen ::= *Lokaldefinition* { ";" *Lokaldefinition* } .
Lokaldefinition ::= *Variable* "=" *Ausdruck* .
Ausdruck ::= "`let`" *Lokaldefinitionen* "`in`" *Ausdruck*
    | "`if`" *Ausdruck* "`then`" *Ausdruck* "`else`" *Ausdruck*
    | *Ausdruck BinärOp Ausdruck*
    | *UnärOp Ausdruck*
    | *Ausdruck Ausdruck*
    | "(" *Ausdruck* ")"
    | *AtomarerAusdruck* .
BinärOp ::= "&" | "|" | "==" | "<" | "+" | "−" | "∗" | "/" .
UnärOp ::= "`not`" | "−" .
AtomarerAusdruck ::= *Variable* | `Zahl` | `Wahrheitswert` .
Variable ::= `Name` .

Die Definition der Syntax der Namen, der (ganzen) Zahlen und der Wahrheitswerte "`true`" und "`false`" wird vorausgesetzt, so dass `Name`, `Zahl` und `Wahrheitswert` hier als Terminalsymbole behandelt werden. Sie werden während der lexikalischen Analyse erkannt und überprüft.

Präzedenzen und Assoziativitätsregeln. Der Einfachheit halber spezifiziert die obige Grammatik keine Präzedenz und keine Assoziativitätsregeln für die Operatoren. Die Operatorenpräzedenz wird mit der folgenden Tabelle definiert, wobei ein Operator umso stärker bindet, je höher seine Präzedenz ist.

<table>
<tr><th>Präzedenz</th><th>Assoziativität</th><th>Operator</th></tr>
<tr><td>8</td><td>linksassoziativ</td><td>Funktionsanwendung</td></tr>
<tr><td>7</td><td>linksassoziativ</td><td>∗</td></tr>
<tr><td>7</td><td>undefiniert[1])</td><td>/</td></tr>
<tr><td>6</td><td>linksassoziativ</td><td>+</td></tr>
<tr><td>6</td><td>undefiniert[1])</td><td>binäres −</td></tr>
<tr><td>5</td><td>undefiniert</td><td>unäres −</td></tr>
<tr><td>4</td><td>undefiniert[1])</td><td>==</td></tr>
<tr><td>4</td><td>undefiniert[1])</td><td><</td></tr>
<tr><td>3</td><td>undefiniert</td><td><code>not</code></td></tr>
<tr><td>2</td><td>rechtsassoziativ</td><td>&</td></tr>
<tr><td>1</td><td>rechtsassoziativ</td><td>|</td></tr>
</table>

[1]: Mehrdeutige Ausdrücke der Form $A - B - C$ und $A/B/C$ sowie Ausdrücke der Form $A\ op_1\ B\ op_2\ C$ mit $op_1, op_2 \in \{==, <\}$ sind nicht zulässig. In solchen Fällen muss geklammert werden.

Die Präzedenzen und Assoziativitäten können durch eine Verfeinerung der vorangehenden Grammatik ausgedruckt werden. Die obige Tabelle ist aber für den Leser verständlicher.

Beispiel eines F-Programms.

```
bool x = x == true | x == false;
   f x = if bool x | x < 0
         then 1
         else x * f (x - 1);
  main = f 6;
```

■

Wohltypisiertheit. Ein F-Programm muss weitere Bedingungen erfüllen, die die vorangehende Grammatik nicht spezifiziert:

1. Jedes F-Programms enthält genau eine Funktionsdefinition der Gestalt:

   `main = Ausdruck`

2. F-Programme müssen wohltypisiert sein.

Wie kann die Wohltypisiertheit definiert werden, wenn kein Typsystem vereinbart wurde?

Einige offensichtlichen Bedingungen lassen sich leicht überprüfen, obwohl ihre formale Spezifikation nicht notwendigerweise unmittelbar ist, wie zum Beispiel:

- Die Auswertung der Bedingung eines if-Ausdrucks muss einen Boole'schen Wert liefern.
- Die Auswertung des Arguments eines arithmetischen Ausdrucks muss eine Zahl liefern.
- Die Anzahl der aktuellen Parameter einer Funktionsanwendung darf nicht größer sein als die Anzahl der formalen Parameter in der zugehörigen Funktionsdefinition.

Solche Bedingungen sind nicht immer einfach zu formalisieren. Wir versuchen gar keine für die Praxis zufriedenstellende Spezifikation dieser Bedingung zu geben, und verlangen einfach, dass während der Auswertung eines F-Programms nie auf Funktionsanwendungen gestoßen wird, die eine unpassende Anzahl von aktuellen Parametern oder Parametern eines falschen Typs haben.

Für die Praxis ist eine solche Annahme höchst unzufriedenstellend, weil sie nicht syntaktisch, also ohne das Programm auszuführen, überprüfbar ist. In der Praxis werden syntaktisch entscheidbare und zur Übersetzungszeit prüfbare ausreichende Bedingungen der Wohltypisiertheit verlangt. Dies ist aber für den Zweck, den F hier dient, nicht nötig.

Rekursive Funktionsdefinitionen. Die Lokaldefinitionen eines Ausdrucks der Gestalt

$$\texttt{let } V_1 \texttt{ = } A_1 \texttt{ ; } \ldots \texttt{ ; } V_n \texttt{ = } A_n \texttt{ in } A$$

heißen *rekursiv*, wenn es $i_1, \ldots, i_{k+1}$ in $\{1, \ldots, n\}$ gibt, so dass

- $i_j \neq i_m$ für $j \neq m$, und
- für $j = 1, \ldots, k$ kommt $V_{i_{j+1}}$ frei im Ausdruck A_{i_j} vor, und
- V_{i_1} kommt frei im Ausdruck $A_{i_{k+1}}$ vor.

Informell bedeutet diese Definition, dass die lokalen Definitionen einen Zyklus enthalten: Die Definition von V_{i_1} bezieht sich auf V_{i_2}, usw., und die Definition von $V_{i_{k+1}}$ bezieht sich auf V_{i_1}.

Bemerkungen.

1. In F wird kein Unterschied zwischen rekursiven und nichtrekursiven Lokaldefinitionen gemacht.
2. Die Definition von *Lokaldefinition* ist eingeschränkter als die Definition von *Definition* weil die erste im Gegenteil zur zweiten keine Parameter zulässt.

■

4.3.2 Geltungsbereiche von Variablen in F

Nachdem die Syntax von F eingeführt worden ist, ist es möglich den Geltungsbereich einer Variablen formal zu definieren.

Bemerkung. Der Unterschied zwischen „Vorkommen einer Variable" in einem Ausdruck und „Variable" ist wichtig. Betrachen wir das folgende Beispiel:

	`let x = 1 in let x = 2 in x * x`
Vorkommen der Variable `x`:	v_1 v_2 v_3 v_4

Im ersten Ausdruck überschattet der innere Vorkommen von `x` den äußeren Vorkommen der selben Variable. Die Bedeutung dieses Ausdrucks ist also:

```
let x = 1 in let y = 2 in y * y
```

■

Definition.

- Sei A ein F-Ausdruck (im Sinne der Grammatik von F). Ein Teil B von A heißt *Teilausdruck* von A, wenn B Teil von A (möglicherweise gleich A) und ein Ausdruck (im Sinne der Grammatik von F) ist.
- Sei A der Ausdruck

  $\texttt{v}_1 \ldots \texttt{v}_{i-1}$ `x` $\texttt{v}_{i+1} \ldots \texttt{v}_n$ = A_1

oder

let $\mathtt{v}_1$ = B_1 ; ... ; $\mathtt{v}_{i-1}$ = B_{i-1} ; x = B_i ; $\mathtt{v}_{i+1}$ = B_{i+1} ; ... ; $\mathtt{v}_n$ = B_n
in A_1

Jedes Vorkommen der Variable x in einem Teilausdruck E von A (einschließlich $E = A$) heißt *gebunden* in diesem Teilausdruck E.

- Ein Vorkommen einer Variable, das in einem Ausdruck A vorkommt, heißt *frei* in A, wenn es in A nicht gebunden ist.
- Ein Ausdruck, in dem keine Variable frei vorkommt, heißt *geschlossen.*

■

Sprachen, die wie z. B. Scheme, letrec- von let-Ausdrücken unterscheiden, interpretieren manche Ausdrücke in verschiedenen Weise.

```
(* Keine F-Definitionen! *)
f x = let    y = x ; x = 5 in y;
g x = letrec y = x ; x = 5 in y;
```

Im ersten Fall hat der Ausdruck `x`, mit dem `y` definiert wird, den Wert des formalen Parameters `x`. Der Wert von `f 1` ist also `1`. Eine solche Behandlung ist sinnvoll, weil sie eine effiziente Übersetzung ermöglicht. Sie ermöglicht aber keine rekursiven lokalen Definitionen.

Im zweiten Fall hat der Ausdruck `x`, mit dem `y` definiert wird, den Wert der lokalen Definition, d. h. `x = 5`. Der Wert von `g 1` ist also `5`.

In F werden lokalen Definitionen wie im Falle des letrec-Ausdrucks im vorangehenden Beispiel behandelt. Das heißt, dass das folgende F-Programm den Wert `5` liefert.

```
(* F-Definitionen *)
f x = let y = x ; x = 5 in y;
main = (f 1)
```

Definition. Seien $A, A_0, A_1, \ldots, A_n$ F-Ausdrücke.

1. Die Menge FreiVar(A) der in einem F-Ausdruck A frei vorkommenden Variablen ist wie folgt definiert:

FreiVar(A) $= \emptyset$ falls A eine Zahl oder ein Boole'scher Wert ist
FreiVar(A) $= \{A\}$ falls A eine Variable ist
FreiVar((A)) = FreiVar(A)
FreiVar(op A) = FreiVar(A) falls op ein unärer Operator ist
FreiVar(A_1 op A_2) = FreiVar(A_1) $\cup$ FreiVar(A_2)
falls op ein binärer Operator ist
FreiVar(**if** A_1 **then** A_2 **else** A_3) = FreiVar(A_1) $\cup$ FreiVar(A_2) $\cup$ FreiVar(A_3)
FreiVar($A_1 A_2$) = FreiVar(A_1) $\cup$ FreiVar(A_2)
FreiVar(let $\mathtt{x}_1$ = A_1 ; ... ; $\mathtt{x}_n$ = A_n in A_0) $= (\bigcup_{i=0}^{i=n} \text{FreiVar}(A_i)) \setminus \{\mathtt{x}_1, \ldots, \mathtt{x}_n\}$
FreiVar($\mathtt{x}_1$ = A_1 ; ... ; $\mathtt{x}_n$ = A_n) $= (\bigcup_{i=1}^{i=n} \text{FreiVar}(A_i)) \setminus \{\mathtt{x}_1, \ldots, \mathtt{x}_n\}$

2. Die Menge GebVar(A) der in einem F-Ausdruck A gebunden vorkommenden Variablen ist definiert:

$$\begin{array}{lll}
\mathrm{GebVar}(A) & = \emptyset & \text{falls } A \text{ eine Zahl oder ein Boole'scher Wert ist} \\
\mathrm{GebVar}(A) & = \emptyset & \text{falls } A \text{ eine Variable ist} \\
\mathrm{GebVar}((A)) & = \mathrm{GebVar}(A) & \\
\mathrm{GebVar}(op\ A) & = \mathrm{GebVar}(A) & \text{falls } op \text{ ein unärer Operator ist} \\
\mathrm{GebVar}(A1\ op\ A2) & = \mathrm{GebVar}(A1) \cup \mathrm{GebVar}(A2) & \\
 & & \text{falls } op \text{ ein binärer Operator ist}
\end{array}$$

$$\mathrm{GebVar}(\texttt{if } A_1 \texttt{ then } A_2 \texttt{ else } A_3) = $$
$$\mathrm{GebVar}(A_1) \cup \mathrm{GebVar}(A_2) \cup \mathrm{GebVar}(A_3)$$
$$\mathrm{GebVar}(A1A2) = \mathrm{GebVar}(A_1) \cup \mathrm{GebVar}(A_2)$$
$$\mathrm{GebVar}(\texttt{let x}_1 \texttt{ = } A_1 \texttt{ ; ... ; x}_n \texttt{ = } A_n \texttt{ in } A_0) = (\textstyle\bigcup_{i=0}^{i=n} \mathrm{GebVar}(A_i)) \cup \{\texttt{x}_1, \ldots, \texttt{x}_n\}$$
$$\mathrm{GebVar}(\texttt{x}_1 \texttt{ = } A_1 \texttt{ ; ... ; x}_n \texttt{ = } A_n) = (\textstyle\bigcup_{i=1}^{i=n} \mathrm{GebVar}(A_i)) \cup \{ \texttt{x}_1, \ldots, \texttt{x}_n\}$$

■

4.3.3 Ein Parser für F

Die Grammatik von F muss zunächst so verfeinert werden, dass die Operatorenpräzedenzen und -assoziativitäten von der Grammatik bestimmt werden. Wenn sie dann noch so umgewandelt wird, dass sie die LL(1)-Bedingung erfüllt, kann ein LL(1)-Parser für F gemäß Abschnitt 2.2.3 generiert werden. Ähnlich wie für die imperative Programmiersprache I sind Abweichungen notwendig, um der Tatsache Rechnung zu tragen, dass F nicht kontextfrei ist.

4.4 Auswertung von F-Programmen

Die Syntax einer funktionalen Programmierpsrachen bestimmt den Auswertungsansatz nicht. Für F werden die verzögerte Auswertung und die statische Sichtbarkeitsregel (statische Bindung) gewählt.

Die verzögerte Auswertung hat die Vorteile der Auswertung in normaler Reihenfolge (das Vermeiden von unötigen Auswertungen von Argumente und besseres Terminierungsverhalten) ohne ihren Nachteil (die wiederholte Auswertung von Argumente).

Die statische Sichtbarkeitsregel stellt die referentielle Transparenz der Programmiersprache sicher.

4.4.1 Currying (Darstellung von mehrstelligen Funktionen mittels mehererer einstelliger Funktionen)

Der Einheitlichkeit halber werden mehrstellige Funktionen durch meherere einstellige Funktionen dargestellt, ohne dass dabei die Stelligkeit der Funktionen geändert wird oder verloren geht. Das Prinzip dieser Repräsentation wird zunächst anhand eines Beispiels verdeutlicht:

Die 3-stellige Funktion

```
summe:  Zahl x Zahl x Zahl --->   Zahl
         n      m      p   ---> n + m + p
```

kann als die folgende 1-stellige Funktion angesehen werden

```
summe: Zahl ---> (   Zahl x Zahl --->    Zahl    )
        n              m     p         n + m + p
```

d. h. als eine Funktion, die Zahlen auf 2-stelligen Funktionen Zahl x Zahl ---> Zahl abbildet. Werden wiederum diese 2-stelligen Funktionen nach dem gleichen Prinzip dargestellt, ergibt sich die folgende Repräsentation der 3-stellige Funktion summe

```
summe: Zahl ---> ( Zahl ---> ( Zahl --->    Zahl    ) )
        n            m            p       n + m + p
```

was, unter der Annahme, dass der Operator ---> rechtsassoziativ ist, wie folgt angegeben wird

```
summe: Zahl ---> Zahl ---> Zahl --->    Zahl
        n         m         p       n + m + p
```

Die Darstellung von mehrstelligen Funktionen mittels meherere einstelligen Funktionen wird „currying“ (nach dem Logiker Curry) oder „schönfinkeln“ (nach dem Logiker Schönfinkel) genannt. Für $n \geq 1$ wird also eine n-stellige Funktion

```
f: D1 x D2 x ... x Dn ---> B
```

wie folgt dargestellt:

```
f: D1 ---> D2 ---> ... ---> Dn ---> B
```

4.4.2 Darstellung von Funktionsanwendungen als Graphen

Unter der Anahme, dass (Dank des Curryings) mehrstellige Funktionen mittels 1-stelliger Funktionen dargestellt werden, können Funktionsanwendungen als Graphen dargestellt werden.

Eine (1-stellige) Anwendung A1 A2 wird wie folgt dargestellt:

```
     APP
    /   \
  A1     A2
```

APP steht für Anwendung (application).

Betrachten wir die folgende Definition aus einem F-Programm:

```
quadrat x = x * x
```

und den Ausdruck quadrat (quadrat 3). Der Einheitlichkeit halber schreiben wir (vorübergehend) das Produkt wie eine gewöhnliche Funktionen als Präfix-Operation:

```
quadrat x = * x x
```

Für den Ausdruck quadrat (quadrat 3) wird vom F-Parser der folgende Graph erzeugt:

```
          APP
         /   \
  quadrat     APP
             /   \
      quadrat     3
```

Die Graphen zur Darstellung von Funktionsanwendungen können wie folgt spezifiziert werden:

- Mittels Currying werden mehrstellige Funktionen als 1-stellige Funktionen dargestelt.
- Knoten haben den Bezeichner (tag) APP.
- Blätter sind Basiswerte (Zahlen oder Wahrheitswerte), vordefinierte Funktionen (wie +, *, usw.) oder Variablen.

Offenbar sind Graphen, die dieser Definition entsprechen, Bäume.

Beispiele.

1. Graph zur Darstellung des Ausdrucks `(+ 5 (* 4 3))`:

```
              APP
          /         \
      APP             APP
     /   \           /   \
    +     5       APP     3
                 /   \
                *     4
```

2. Graph zur Darstellung des Ausdrucks `(+ (- 5) (* 4 3))`:

```
              APP
          /         \
      APP             APP
     /   \           /   \
    +    APP      APP     3
         / \     /   \
        -   5   *     4
```

■

4.4.3 Auswertung eines F-Ausdrucks als Graphreduktion

Für die Programmiersprache F wurden ausgewählt:

1. die verzögerte Auswertung, und
2. die statische Sichtbarkeitsregel (statische Bindung).

Die verzögerte Auswertung besteht in einer Verbesserung der Auswertung in normaler Reihenfolge (outermost evaluation), d. h. als nächsten zu reduzierenden Ausdruck wird immer der äußerste Ausdruck gewählt. Die Verbesserung besteht darin, dass die sich aus der Auswertung eines äußeren Ausdrucks ergebenden mehrfachen Vorkommen eines gleichen inneren Ausdrucks mittels Zeiger geteilt werden. Diese Teilung von Teilausrücken führt dazu, dass aus den Bäumen des vorhangehenden Abschnitts Graphen entstehen, die keine Bäume mehr sind.

Die verzögerte Auswertung wird anhand von „Graphreduktionen" formal spezifiziert.

Betrachten wir die Funktionsdefinition:

```
quadrat x = x * x
```

Der definierende Ausdruck `x * x` dieser Definition wird wie folgt als Graph dargestellt:

```
         APP
       /     \
    APP       \
   /   \_______ x
  *
```

Betrachten wir nun den Graph, der den Ausdruck `quadrat (quadrat 3)` darstellt:

```
    ===>  APP
         /   \
quadrat     APP  <= W
           /   \
    quadrat     3
```

Der äußerste Ausdruck `quadrat (quadrat 3)`, dessen Wurzel mit `===>` kennzeichnet ist, wird durch den Graph, der der definierende Ausdruck `x * x` von `quadrat x` darstellt, ersetzt. Dabei wird der formale Parameter `x` in diesem definierenden Ausdruck durch den Graph mit Wurzel `W` ersetzt, der das Argument `quadrat 3` von `quadrat (quadrat 3)` darstellt. So ergibt sich der folgende Graph:

```
     APP
    /   \
 APP     \
 /   \___ APP <***
*        /   \
  quadrat     3
```

Der nächste Reduktionsschritt kann nicht wie der vorherige durchgeführt werden, weil `*` eine vordefinierte Funktion ist. Die Anwendung einer vordefinierten Funktion ist erst dann möglich, wenn ihre alle Argumente ausgewertet worden sind.

Der nächster Reduktionsschritt ist also die Reduktion der mit `<***` gekennzeichneten Funktionsanwendung. Ähnlich wie vorher ergibt sich der folgende Graph:

```
      APP
     /   \
  APP     \
  /  \____ APP  <***
*         /   \
       APP     \
      /   \____ 3
    *
```

Da `*` 2-stellig ist, betrifft die nächste Reduktion die mit `<***` gekennzeichnete Funktionsanwendung. (Wäre `*` 1-stellig, dann betreffe dieser Schritt den tiefsten APP-Knoten.) Aus dieser Reduktion ergibt sich der Graph:

```
      APP    <===
     /   \
  APP     \
  /  \____ 9
*
```

Da `*` 2-stellig ist, wird nun die mit `<===` gekennzeichnete Anwendung reduziert, was den Wert `81` liefert.

Freilegung eines Anwendungsrückgrates. Der nächste zu reduzierende Teil eines Graphen kann wie folgt ermittelt werden:

1. Ausgehend von der Wurzel des Graphen wird der am weitesten links stehende Ast bis zum Blatt durchlaufen. Dieser Ast wird „Rückgrat der (Funktions-)Anwendung" geannt. Das Blatt ist zwangsläufig mit einer Funktion `f` mit Stelligkeit $n \geq 0$ gekennzeichnet.

2. Das Anwendungsrückgrat wird vom seinem Blatt `f` wieder n Schritte hochgelaufen, wobei n die Stelligkeit von `f` ist.

Dies wird „Freilegung des Anwendungsrückgrates" (unwinding the spine) genannt.

Ein Ausdruck `f A1 A2 A3`, wobei `f` eine 2-stellige Funktion ist, wird z. B. wie folgt dargestellt:

```
                 APP
                /   \
    ===>  APP        A3
         /   \
      APP      A2
     /   \
   f       A1
```

Da `f` 2-stellig ist, ist die mit `===>` gekennzeichnete Anwendung die nächstzureduzierende. Wäre `f` 1-stellig, wäre die nächstzureduzierende Anwendung die unterste, wäre `f` 3-stellig, wäre sie die oberste.

Dieses Verfahren lässt sich einfach mit einem Keller implementieren.

Wir erinnern daran, dass geschatelte Ausdrücke klammerfrei dargestellt werden können, wenn eine Postfix-Notation der Operatoren gewählt wird. (S. 3.5.1 „Übersetzung von Ausdrücke“), wie zum Beispiel:

```
      (a + b) * c    --->    a b + c *
(a * b) + (c - d)    --->    a b * c d - +
```

Wie machen uns diese Tatsache zunutze, um Anwendungen in einem Keller darzustellen. Betrachten wir z. B. wieder den Ausdruck `f A1 A2 A3`, in dem `f` eine 2-stellige Funktion ist. Er wird wie folgt in einem Keller klammerfrei dargestellt:

```
        Keller
       +------+
       |   ------------------->  APP
       +------+                 /   \
       |   --------------->  APP      A3
       +------+             /   \
       |   ----------->  APP      A2
       +------+         /   \
top -> |   ------->   f       A1
       +------+
```

Die verzögerte Auswertung wird dadurch verwirklicht, dass wiederholte Teilausdrücke – wie etwa `* 3 3` im vorangehenden Beispiel – durch die Angabe einer gemeinsamen Adressen geteilt werden.

Das Verfahren zur „Freilegung des Rückgrates“ kann durch eine Suche durch die Kellerzellen implementiert werden, wenn vorausgesetzt wird, dass der Zugriff nicht nur auf die oberste Kellerzelle möglich ist. (Die selbe Annahme wurde bereits bei der Spezifikation der abstrakten Maschine MI gemacht.)

Werte. In einem Ausdruck können auch Werte – in F, Zahlen oder Wahrheitswerte – vorkommen. Werte können nicht als Funktionsanwendung dargestellt werden. Zu ihrer Darstellung in (Graphen-)Ausdrücke führen wir eine neue Art von Knoten, die VAL-Knoten:

```
VAL T A
```

wobei `T` der Typ (`Zahl` oder `Bool`) des Wertes `A` ist.

4.5 Die abstrakte Maschine MF für F

MF ist ähnlich wie MI konzipiert. Zwei prinzipielle Unterschiede zwischen den Programmiersprachen I und F, die dabei berücksichtigt werden müssen, sind die folgenden:

1. Die von der abstrakten Maschine MI bearbeiteten dynamischen Strukturen sind bereits in der Quellsprache I explizit aufgebaut. (Die funktionale Sprache F hat keine Zeiger und ermöglicht deswegen keinen expliziten Aufbau von dynamischen Strukturen. Funktionsanwendungen sind aber dynamische Strukturen, die in MF ähnlich wie die dynamische Strukturen von I in einer Halde (heap) dargestellt werden.)

2. Die abstrakte Maschine MI legt die Ergebnisse von Berechnungen so weit wie möglich in den Keller. Die Halde wird zur Speicherung von Daten verwendet, deren Größe im voraus nicht festgelegt werden kann. Da Programmausdrücke in funktionaler Sprachen ebenfalls einer im voraus nicht zu ermittelnde Größe haben, werden sie in der Halde dargestellt. Der Keller enthält lediglich einen Verweis (Zeiger) auf diese Ausdrücke.

In der Spezifikation der abstrakten Maschine MF werden Keller und Halde nicht als zwei Teile eines einzigen Datenspeichers angesehen, sondern als zwei Speicher. Der im Abschnitt 3.6.4 „Speicherbelegung für Zeiger und dynamische Datenstrukturen“ eingeführte Test zur Erkennung von Speicherüberläufen muss aber auch für MF durchgeführt werden. Die Spezifikation dieses Testes ist nicht Teil der abstrakten Maschine MF. Die Durchführung dieses Testes wird der Implementierung von MF auf einer konkreten Maschine überlassen.

MF besteht aus:

- 4 Speichern:
  - 1 Programmspeicher (code)
  - 1 Keller (stack)
  - 1 globale Umgebung (global)
  - 1 Halde (heap)
- 3 Registern:
  - 1 Instruktionsregister I (instruction)
  - 2 Adressregistern:

    T (top)
    P (program counter)
- der Sprache von MF (d. h. die Befehle von MF)

Außer wenn P explizit geändert wird – in den Befehle `Reset`, `Reduce` und `Return` –, wird bei jedem Befehl den Wert von P um 1 erhöht (`P := P + 1`).

Der Hauptzyklus der abstrakten Maschine MF lautet genau wie für die abstrakte Maschine *MI*:

```
P:=0;   I:=code[P];
while (I≠Halt) { P:=P+1;   Befehl in I ausführen;   I:=code[P]; }
```

4.5.1 Die Speicher

Der Programmspeicher code. Der Programmspeicher code wird wie bei der abstrakten Maschine MI verwendet. Sein Inhalt wird zur Laufzeit nicht verändert.

Der Keller stack. Im Speicher stack, der als Keller verwaltet wird, werden Verweise (Zeiger) auf die auszuwertenden Ausdrücke gespeichert. Bis auf diese Indirektion werden Ausdrücke ähnlich wie arithmetische und Boole'sche Ausdrücke mit der Maschine MI klammerfrei und postfix im Keller gespeichert.

Der Programmspeicher global. Der Programmspeicher global wird zur Speicherung der Definitionen von globalen Funktionen verwendet. Eine Funktionsdefinition heißt „global", wenn sie nicht in einem let-Ausdruck vorkommt.

Eine globale Funktionsdefinition wird durch eine `DEF`-Zellen der folgenden Gestalt repräsentiert:

```
DEF  f  N   C-Adr1
```

wobei `f` eine Funktion einer Stelligkeit `N` und `C-Adr1` die Codeadresse der Übersetzung des definierenden Ausdruck in der Definition von `f` ist.

Die `DEF`-Zellen werden verkettet, so dass die globale Umgebung eine Liste ergibt. Die globale Umgebung wird zur Übersetzungszeit erstellt und zur Laufzeit durchsucht, um die Stelligkeit und die Codeadresse einer gegebenen Funktion`f` zu ermitteln. Zur Laufzeit wird die globale Umgebung nicht geändert. Die globale Umgebung kann also an einem Ende des Speicherbereichs (store) untegebracht werden, in dem sich sowohl der Keller stack wie die Halde heap sich befinden.

Zur Festellung der Codeadresse der `DEF`-Zellen wird die Code-Erzeugung in zwei Läufen oder die Fixup-Technik angewendet.

Die Halde heap. Die Halde dient zur Speicherung der Anwendungsgraphen, die anhand von `APP`-Zellen dargestellt werden, die zwei Haldenadressen von `APP`-Zellen, Variablen oder Werte angeben:

```
APP  H-Adr1   H-Adr2
```

Eine solche `APP`-Zelle stellt die folgende Struktur dar:

```
        APP
       /   \
     A1     A2
```

Zur Darstellung von Werten werden `VAL`-Zellen verwendet:

```
VAL  Typ   Wert
```

wobei `Typ` entweder `Zahl` oder `Bool` ist (bzw. eine numerische Codierung davon, z. B. 0 für `Zahl` und 1 für `Bool`) und `Wert` der darzustellende Wert ist (bzw. eine numerische Codierung davon, z. B. 0 für `false` und 1 für `true`).

Die abstrakten Maschinen MiniMF und MF. Die Übersetzung einer Vereinfachung der Programmiersprache F, MiniF genannt, wird zunächst behandelt. MiniF unterscheidet sich von F dadurch, dass let-Ausdrücke und vordefinierte Funktionen in MiniF nicht vorkommen. Die abstrakte Maschine MiniMF für MiniF wird zunächst eingeführt. Danach wird sie zur abstrakten Maschine MF für die Behandlung von F-Programmen erweitert.

4.5.2 Die Befehle der abstrakten Maschine MiniMF

Im Folgenden werden die folgende Prozeduren angenommen:

`address(f)`: liefert die Adresse der Zelle `(DEF f  N  C-Adr1)` in der globalen Umgebung, die die Stelligkeit `N` und die Code-Adresse `C-Adr1` der Übersetzung des definierenden Ausdrucks von `f` angibt.

`add2arg(Adr)`: liefert die Haldenadresse `H-Adr2`, falls `Adr` die Haldenadresse von `(APP H-Adr1 H-Adr2)` ist.

`new(Knotentyp, A, B)`: bildet einen Knoten vom Typ `Knotentyp = APP` (Funktionsanwendung) oder `Knotentyp = VAL` mit Feldern `A` und `B` und liefert die Adresse des kreierten Knotens. (Dabei wird ein Haldenregister verwaltet.)

`typ(Adr)`: liefert den Typ `Zahl` oder `Bool` eines Haldenelements `(VAL Typ Wert)` mit Adresse `Adr`.

Jedes Argument eines Befehls hat einen der folgenden Typen (die Typen werden nicht von der abstrakten Maschine bearbeitet, sondern dienen nur dem besseren Verständnis der Befehle):

- F: Funktion einer Stelligkeit $n \geq 1$.
- T: Typ der Sprache F, d. h. „Zahl“ oder „Bool“.
- W: Wert eines der Typen der Sprache F, d. h. eine Zahl oder ein Wahrheitswert.
- N: natürliche Zahl.

Reset. Initialisierung.

```
Reset         T := -1;
```

Der Programmzähler P und das Instruktionsregisters I werden im Haupt-Instruktionszyklus der abstrakten Maschine initialisiert.

Push function. Zum Laden der Adresse (in der globalen Umgebung) der DEF-Zelle einer Funktion auf den Keller:

```
Pushfun f     T := T + 1;
              stack[T] := address(f)
```

`f` vom Typ F.

Push value. Zum Laden der Adresse eines Wertes auf den Keller:

```
Pushval t w   T := T + 1;
              stack[T] := new(VAL, t, w)
```

`w` vom Typ W. `t` vom Typ T (`t` ist der Typ von `w`).

Push parameter. Um die Adresse eines Arguments der Anwendung einer Funktion auf den Keller zu bringen, wenn die Adresse der Funktionsanwendung in der oberen Kellerzelle liegt:

```
Pushparam n   T := T + 1;
              stack[T] := add2arg(stack[T - n - 2])
```

`n` vom typ N.

Der Befehl `Pushparam` wird wie folgt verwendet. Man betrachte die folgende Definition einer 2-stelligen Funktion `g f x = f x`. Die Haldenadresse der Parameter `f` und `x` von `g` werden wie folgt auf den Keller gebracht. Zunächst wird der 2. Parameter `x` behandelt, anschließend der 1. Parameter `f`, was der klammerfreien Postfix-Darstellung von zusammengesetzten Ausdrücken entspricht.

Der Ausgangspunkt ist wie folgt, d. h. der Graph zur Darstellung vom Ausdruck `g f x` wurde bereits aufgebaut:

```
             stack
             +------+
             | Adr1 |
             +------+
             |   --------------------> APP
             +------+                 /   \
             |   --------------->  APP      x
             +------+             /   \
             |   ----------->    g     f
             +------+
    T  -> | Adr2 |
             +------+
```

Wir werden später sehen, dass `Adr2` eine Codeadresse (Rückkehradresse) ist. `Adr1` stellt eine beliebige Adresse dar.

Zum laden der Haldenadresse des 2. Parameters `x` wird der Befehl `Pushparam 2` durchgeführt:

```
             stack
             +------+
             | Adr1 |
             +------+
             |   --------------------> APP
             +------+                 /   \
             |   --------------->  APP      x
             +------+             /   \     ^
             |   ----------->    g     f    |
             +------+                       |
             | Adr2 |                       |
             +------+                       |
    T  -> |   --------------------------+
             +------+
```

Zum laden der Haldenadresse des 1. Parameters `f` wird der Befehl `Pushparam 2` durchgeführt. Die Zahl `2` ergibt sich wie folgt:

- `1`, weil es sich um den 1. Parameter handelt,
- erhöht um `1`, weil bereits 1 Parameter, nämlich `x` behandelt wurde. (Siehe Übersetzungsschema für Funktionsanwendungen auf Seite 78, Unterschied zwischen Pos(x) und Pos+1(x).)

Es ergibt sich das folgende:

```
      stack
      +------+
      | Adr1 |
      +------+
      |   --------------------> APP
      +------+                 /   \
      |   --------------->  APP      x
      +------+             /   \     ^
      |   ----------->    g     f    |
      +------+                  ^    |
      | Adr2 |                  |    |
      +------+                  |    |
      |   ---------------------------+
      +------+                  |
T  -> |   ----------------------+
      +------+
```

Make Application Node. Zum Aufbau eines Anwendungsknotens in der Halde

```
Makeapp       stack[T - 1] := new(APP,stack[T],stack[T-1]);
              T := T - 1
```

Die 2 oberen Kellerzellen werden durch eine einzige Zelle ersetezt, die die Adresse der Wurzel des kreierten Anwendungsgraph enthält.

Nachdem die Haldenadresse aller Parameter einer Funktionsanwendung unter Anwendung des Befehls `Pushparam` auf den Keller gebracht wurden, wird der Befehl `Makeapp` durchgeführt. Betrachtet man also das Beispiel `g f x = f x`, so gibt das vorangehendes Bild den Ausgangspunkt zu einer Durchführung von `Makeapp`. Danach ergibt sich:

```
      stack
      +------+
      | Adr1 |
      +------+
      |   --------------------> APP
      +------+                 /   \
      |   --------------->  APP      x
      +------+             /   \     ^
      |   ----------->    g     f    |
      +------+                  ^    |
      | Adr2 |                  |    |
      +------+                  |    |
T  -> |   -------------> APP    |    |
      +------+          /   \   |    |
                       |     |  |    |
                       +--------+    |
                             |       |
                             +-------+
```

Slide. Um den Keller um n Zellen unterhalb der zwei obersten Zelle abzuräumen:

```
Slide n       stack[T - n - 1] := stack[T - 1];
              stack[T - n] := stack[T];
              T := T - n
```

`n` vom Typ N.

Der Befehl `Slide 3` wird angewandt, um aus dem letzten Zustand des vorangehenden Beispiels den folgenden Zustand zu bilden. Der Parameter `3` entspricht der Tiefe des Anwendungsgraphen, der beseitigt wird, das heißt, 1 plus Stelligkeit der Funktion des Anwendungsgraphen.

```
            stack
            +------+
            | Adr1 |
            +------+
            | Adr2 |
            +------+
     T  ->  |   -------------> APP
            +------+          /   \
                             |     |
                             f     x
```

Reduce. Zur Reduktion eines Graphen:

```
Reduce        P := P - 1;
              case  heap[stack[T]]  of
                    (APP Adr1 Adr2) :  T := T + 1;
                                       stack[T] := Adr1          |
                    (DEF f N Adr)   :  T := T + 1;
                                       stack[T] := P + 1;
                                       P := Adr                  |
                    (VAL Typ Wert)  :  P := stack[T - 1];
                                       stack[T - 1] := stack[T] ;
                                       T := T - 1
              end
```

Der erste Fall ist die Freilegung des Anwendungsrückgrats. Dabei bleibt der Wert von P unverändert, bis das Blatt erreicht wird. Der zweite Fall entspricht den Befehl **CALL** von MI. Er trifft zu, wenn das Blatt des Rückgrates erreicht wird und eine Funktion ist. Die Rückkehradresse wird gesichert, bevor es auf die Übersetzung der Definiton dieser Funktion gesprungen wird. Der dritte Fall trifft nach Beendigung der Auswertung zu und entspricht einen Rücksprung.

Im zweiten Fall wird nicht überprüft, ob genügend, d. h. `N`, (Adressen von) Argumente auf dem Keller liegen. Unter der bereits erwähnte Annahme, dass das F-Programm wohltypisiert ist, trifft es zu. Manche abstrakte Maschinen prüfen dies zur Laufzeit.

Return. Zum Rückkehr zur gesicherten Codeadresse:

```
Return        P := stack[T - 1];
              stack[T - 1] := stack[T];
              T := T - 1
```

Die Haldenadresse des Ergebnisses der bisher geführten Funktionsanwendungen liegt in der oberen Kellerzelle, die Rückkehradresse davor.

Halt. Zum Halten der MiniMF-Maschine.

```
Halt
```

4.6 Code-Erzeugung für MiniMF

Im Folgenden bezeichnet „Code“ das Zielprogramm in der Sprache von MiniMF, das sich aus der Übersetzung eines MiniF- (später F-Programm) ergibt.

Initialisierung. Der Anfang von Code besteht in der folgenden Befehlfolge:

```
Reset
Pushfun main
Reduce
Halt
```

gefolgt von der Befehlfolge, die sich aus der Übersetzung der Funktionsdefinitionen – einschließlich von der Funktion main – des Quellprogramms ergibt.

Übersetzung einer Funktionsdefinition. Die Definition `f` $\mathtt{x}_1 \ldots \mathtt{x}_n$ `= Ausd` einer n-stelligen Funktion `f` wird wir folgt übersetzt:

1. Am Ende des (bereits erzeugten) Codes wird

   L: $\text{Üb}_{\text{Def}}(\mathtt{Ausd}, \text{Pos}, n)$

   hinzugefügt, wobei Pos die Liste $\{\mathtt{x}_1 \to 1, \ldots, \mathtt{x}_n \to n\}$ und L die Adresse ist, wo im Code hinzugefügt wird).

2. In die Globale Umgebung wird die Zelle:

   `DEF f` n L

   hinzugefügt. Ein Verkettungsverweis – in Form einer Umgebungsadresse – kommt am Ende der DEF-Zelle hinzu, so dass die globale Umgebung eine Liste bildet.

Die Liste Pos $= \{\mathtt{x}_1 \to 1, \ldots, \mathtt{x}_n \to n\}$ definiert die Positionen der formalen Parameter der Funktionsdefinition. Pos wird auch *lokale Umgebung* genannt. Präziser wird eine Funktion Pos wie folgt aus der oben spezifizierten Liste definiert:

$\text{Pos}(\mathtt{x}) = i$ falls $x \to i$ der erste Eintrag der Liste Pos ist, dessen linker Teil `x` ist.

Dabei wird nur der erste zutreffende Eintrag berücksichtigt, damit bei let-Ausdrücke das gewünschte Überschatten erfolgt.

Im folgendem bezeichnet Pos$+i$ die Funktion, die wie folgt definiert ist:

$\text{Pos}{+}i(\mathtt{x}) := \text{Pos}(\mathtt{x}) + i.$

Übersetzungsschema Üb_{Def}. Zur Übersetzung des definierenden Ausdrucks (d. h. des rechten Teils) A der Definition einer Funktion mit Stelligkeit n und Positionsfunktion *Pos*:

$\text{Üb}_{\text{Def}}(A, \text{Pos}, n) \quad := \quad \text{Üb}_{\text{Kons}}(A, \text{Pos})$
`Slide` $n+1$
`Reduce`
`Return`

Übersetzungsschema Üb_{Kons}. Zur Konstruktion eines Graphen in der Halde zur Repräsentation eines Ausdrucks:

$$
\begin{array}{rcll}
\text{Üb}_{\text{Kons}}(A_1\ A_2, \text{Pos}) & := & \text{Üb}_{\text{Kons}}(A_2, \text{Pos}) & \\
 & & \text{Üb}_{\text{Kons}}(A_1, \text{Pos}+1) & \\
 & & \texttt{Makeapp} & \\
\text{Üb}_{\text{Kons}}(w, \text{Pos}) & := & \texttt{Pushval}\ \ t\ \ w & w \text{ Basiswert, } t \text{ Typ } (\texttt{Zahl} \\
 & & & \text{oder } \texttt{Bool}) \text{ von } w \\
\text{Üb}_{\text{Kons}}(x, \text{Pos}) & := & \texttt{Pushparam}\ \ \text{Pos}(x) & x \text{ formaler Parameter} \\
\text{Üb}_{\text{Kons}}(f, \text{Pos}) & := & \texttt{Pushfun}\ \ f & f \text{ Funktion}
\end{array}
$$

Die Funktionsanwendung ist linksassoziativ, d. h. $A_1\ A_2\ A_3$ steht für $((A_1\ A_2)A_3)$. Es folgt also aus den vorangehenden Definitionen:

$$
\begin{array}{rcl}
\text{Üb}_{\text{Kons}}(A_1\ A_2\ A_3, \text{Pos}) & := & \text{Üb}_{\text{Kons}}(A_3, \text{Pos}) \\
 & & \text{Üb}_{\text{Kons}}(A_1\ A_2, \text{Pos}+1) \\
 & & \\
 & := & \text{Üb}_{\text{Kons}}(A_3, \text{Pos}) \\
 & & \text{Üb}_{\text{Kons}}(A_2, \text{Pos}+1) \\
 & & \text{Üb}_{\text{Kons}}(A_1, \text{Pos}+2)
\end{array}
$$

So wird die Postfix-Darstellung von Funktionsanwendungen erhalten.

Beispiel. Betrachten wir das folgende MiniF-Programm.

```
main = second 1 2
second x y = y
```

Der Lesbarkeit halber werden eine Codeadresse i als ci und eine Haldenadresse j als hj dargestellt. Die globale Umgebung global wird am Anfang der Halde gelegt, so dass DEF-Zellen Haldenadresse erhalten.

Das vorangehende Programm wird wird folgt übersetzt:

```
code                           global (als Teil von heap)

c0:  Reset
c1:  Pushfun main
c2:  Reduce
c3:  Halt

c4:  Pushval Zahl 2            h0: DEF main 0 c4 h1
c5:  Pushval Zahl 1
c6:  Pushfun second
c7:  Makeapp
c8:  Makeapp
c9:  Slide 1
c10  Reduce
c11: Return

c12: Pushparam 2               h1: DEF second 2 c12 h2
c13: Slide 3
c14: Reduce
c15: Return
```

Ausführung:

```
Register            stack     heap

I: Reset
T: -1
P: c1

I: Pushfun main     s0: h0    h0: DEF main 0 c4
T: s0
P: c2

I: Reduce           s0: h0    h0: DEF main 0 c4
T: s1               s1: c3
P: c4

I: Pushval Zahl 2   s0: h0    h0: DEF main 0 c4
T: s2               s1: c3
P: c5               s2: h2    h2: VAL Zahl 2

I: Pushval Zahl 1   s0: h0    h0: DEF main 0 c4
T: s3               s1: c2
P: c6               s2: h2    h2: VAL Zahl 2
                    s3: h3    h3: VAL Zahl 1

I: Pushfun second   s0: h0    h0: DEF main 0 c4
T: s4               s1: c3
P: c7               s2: h2    h2: VAL Zahl 2
                    s3: h3    h3: VAL Zahl 1
                    s4: h1    h1: DEF second 2 c12

I: Makeapp          s0: h0    h0: DEF main 0 c4
T: s3               s1: c3                               h4:APP
P: c8               s2: h2    h2: VAL Zahl 2               /   \
                    s3: h4    h4: APP h1 h3     h1:second  h3:1


I: Makeapp          s0: h0    h0: DEF main 0 c4              h5:APP
T: s2               s1: c3                                     /   \
P: c9               s2: h5    h5: APP h4 h2              h4:APP  h2:2
                                                           /   \
                                                h1:second  h3:1
I: Slide 1          s0: c3
T: s1               s1: h5    h5: APP h4 h2
P: c10

I: Reduce           s0: c3
T: s2               s1: h5    h5: APP h4 h2
P: c10              s2: h4    h4: APP h1 h3

I: Reduce           s0: c3
T: s3               s1: h5    h5: APP h4 h2
P: c10              s2: h4    h4: APP h1 h3
                    s3: h1    h1: DEF second 2 c12

I: Reduce           s0: c3
T: s4               s1: h5    h5: APP h4 h2
P: c12              s2: h4    h4: APP h1 h3
                    s3: h1    h1: DEF second 2 c12
                    s4: c11
```

```
I: Pushparam 2      s0: c3
T: s5               s1: h5    h5: APP h4 h2
P: c13              s2: h4    h4: APP h1 h3
                    s3: h1    h1: DEF second 2 c12
                    s4: c11
                    s5: h2    h2: VAL Zahl 2

I: Slide 3          s0: c3
T: s2               s1: c11
P: c14              s2: h2    h2: VAL Zahl 2

I: Reduce           s0: c3
T: s1               s1: h2    h2: VAL Zahl 2
P: c11

I: Return           s0: h2    h2: VAL Zahl 2
T: s0
P: c3

I: Halt             s0: h2    h2: VAL Zahl 2
```

■

Sobald die obere Kellerzelle einen Verweis (in Form einer Haldenadresse) auf einen Wert, d. h. auf eine `VAL`-Zelle, enthält, werden nur noch Rückkehradressen nacheinander besucht bis der Befehl **Halt** erreicht wird. Die abstrakte Maschine für MiniF kann also wie folgt optimiert werden:

- Im Befehl **Reduce** wird die Rückkehradresse nicht gerettet.
- Der Befehl **Return** ist im Übersetzungsschema Üb_{Def} nicht nötig.

Diese Optimierung ist aber nur für MiniF gültig. Zur korrekten Behandlung von vordefinierten Funktionen (wie etwa $+, *, /$, usw.) braucht aber die abstrakte Maschine MF für F sowohl die Rettung der Rückkehradresse im Befehl **Reduce** als auch den Befehl **Return** im Übersetzungsschema Üb_{Def}.

4.6.1 Die abstrakte Maschine MF zur Behandlung von vordefinierten Funktionen

Um die Maschine für MiniF um die Behandlung von F-Programmen zu erweitern, werden zwei neue Arten von Haldenzellen eingeführt, der Befehl **Reduce** der abstrakten Maschine MiniMF durch drei neue Befehle ersetzt und drei weitere Befehle hinzugefügt.

Die neuen Haldenzellen haben die Gestalt:

```
IND  Adr

PRE  Op
```

Dabei bezeichnt `Adr` eine Haldenadresse und `Op` entweder die vordefinierte dreistellige Funktion `if` oder eine vordefinierte zweistellige Funktion, d. h. &, |, ==, <, $+$, $-$, $*$, $/$, oder die vordefinierte einstellige Funktion `not`.

Der Befehl **Reduce** der abstrakten Maschine MiniMF für MiniF wird durch die Befehle **Unwind**, **Call** und **Return** ersetzt. Der Befehl **Unwind** entspricht dem Fall `(APP ...)` von **Reduce**, der Befehl **Call** dem Fall `(DEF ...)`, und der Befehl **Return** dem Fall `(VAL ...)`. Zusätzlich werden die Befehle **Pushpre**, **Update** und **Operator** eingeführt.

Anwendung einer zweistelligen vordefinierten Funktion. Im Folgenden betrachten wir den Fall eines Produktes, d. h. der zweistelligen vordefinierten Funktion `*`. Der Ausgangspunkt ist der gleiche wie für Funktionen, die im Programm definiert sind und wurde mit dem Befehl **Unwind** hergestellt:

```
          stack
          +------+
          |   -------------------->  APP
          +------+                   /   \
          |   ---------------->  APP   Arg2
          +------+              /   \
     T -> |   -------------->  *   Arg1
          +------+
```

Der Befehl **Call** wird ausgeführt. Wie bei Funktionen, die im Programm definiert sind, wird dabei die Rückkehradresse auf dem Keller gespeichert. Zusätzlich wird die vordefinierte Funktion (auch Operator genannt) auf dem Keller gespeichert:

```
          stack
          +------+
          |   -------------------->  APP
          +------+                   /   \
          |   ---------------->  APP   Arg2
          +------+              /   \
          |   -------------->  *   Arg1
          +------+
          | C-Adr|    Rückkehradresse
          +------+
     T -> |  *   |
          +------+
```

Mit dem **Pushparam 2** wird die Haldenadresse des ersten Arguments `Arg1` auf den Keller geladen. Wegen der Kellerzelle, die mit `*` belegt ist, muss `Pushparam 2` statt `Pushparam 1` verwendet werden.

```
          stack
          +------+
          |   -------------------->  APP
          +------+                   /   \
          |   ---------------->  APP   Arg2
          +------+              /   \
          |   -------------->  *   Arg1
          +------+                  ^
          | C-Adr|                  |
          +------+                  |
          |  *   |                  |
          +------+                  |
          |   ----------------------+
          +------+
```

Die Befehle **Unwind** und **Call** werden nacheinander ausgeführt, um das erste Argument auszuweren. Danach liegt der die Haldenaddresse des Wertes des ersten Argument auf dem Keller. Im folgenden wird angenommen, dass dieser Wert `3` ist. Damit der Ausdruck `Arg1` nicht wieder ausgewertet wird, falls er in anderen Ausdrücke vorkommt, wird seine Haldenzelle in eine `IND`-Zelle umgewandelt, die auf den Wert von `Arg1` zeigt:

```
stack
+------+
|   ------------------->  APP
+------+                 /   \
|   --------------->  APP   Arg2
+------+             /   \
|   ------------->  *   IND
+------+                 |
| C-Adr|                 |
+------+                 |
|  *   |                 |
+------+                 v
|   -----------> VAL Zahl 3
+------+
```

Mit dem Befehl **Pushparam 4** wird die Haldenadresse des zweiten Arguments `Arg2` auf den Keller geladen. Wegen der zwei zusätzlichen Kellerzellen, die mit `*` bzw. der Adresse der Haldenzelle `(VAL Zahl 3)` belegt sind, muss `Pushparam 4` statt `Pushparam 2` verwendet werden. Die Befehlsequenz **Unwind Call** berechnet den Wert des zweiten Argument `Arg2` und legt die Adresse der Haldenzelle seines Wertes auf den Keller. Im Folgenden wird angenommen, dass dieser Wert 2 ist. Um die Wiederholung der Auswertungen von `Arg2` zu verhindern, wird die Haldenzelle von `Arg2` in eine `IND`-Zelle umgewandelt, die auf seinen Wert zeigt:

```
stack
+------+
|      |
+------+
|   ------------------->  APP
+------+                 /   \
|   --------------->  APP    IND
+------+             /   \     |
|   ------------->  *   IND    |
+------+                 |     |
| C-Adr|                 |     |
+------+                 |     |
|  *   |                 |     |
+------+                 v     |
|   -----------> VAL Zahl 3    |
+------+                       |
|   -----------> VAL Zahl 2 <-+
+------+
```

Der Befehl **Operator 2** wird ausgeführt. Das Argument `2` ist die Stelligkeit des Operatoren `*`. Der Operator, der die Kellerzelle mit Adresse `T - 2` angibt, wird auf die zwei oberen Kellerzelle angewendet. Das Ergebnis dieser Anwendung wird in eine neue Haldenzelle gespeichert, deren Adresse auf den Keller gebracht wird. Dazu werden Kellerzellen beseitigt, die nicht mehr benötig werden:

```
stack
+------+
|      |
+------+
|   -------------------------->  APP
+------+                        /   \
| C-Adr|                     APP   IND --> VAL Zahl 2
+------+                    /   \
|   -----> VAL Zahl 6      *    IND -----> VAL Zahl 3
+------+
```

Durch eine Indirektion, d. h. eine `IND`-Zelle, weist der Befehl **Update Op** der Haldenadresse des urspünglichen Ausdrucks die Adresse des soeben berechneten Werts zu:

```
stack
+------+
|      |
+------+
|   ----------> IND
+------+         |
| C-Adr|         |
+------+         v
|   -----> VAL Zahl 6
+------+
```

Audrücke, die eine einstellige vordefinierte Funktion oder die dreistellige vordefinierte Funktion `if` enthalten, werden ähnlich ausgewertet.

Definition der neuen Befehle. Die neuen Befehle werden unter Verwendung der Funktion `value`, die zunächst spezifiziert wird, wie folgt definiert.

`value(heap[Adr1])`	:=	`value(heap[Adr2])`	falls `heap[Adr1]` = `(IND Adr2)`,
		`heap[Adr1]`	andernfalls.

Unwind. Zur Freilegung des Rückgrates.

```
Unwind       P := P - 1;
             if   value(heap[stack[T]]) = (APP Adr1 Adr2)
             then T := T + 1;
                  stack[T] := Adr1
             else P := P + 1
```

So lange auf Kellerzellen gestoßen wird, die auf `APP`-Zellen zeigen, wird der Befehl `Unwind` durchgeführt. Der Befehl `Unwind` hat also die gleiche Wirkung wie der Fall `(APP Adr1 Adr2)` von `Reduce`.

Call. Zum Aufruf einer Funktion.

```
Call         case value(heap[stack[T]]) of
                  (VAL T V)     : P := P + 1
                  |
                  (DEF f N Adr) : T := T + 1;
                                  stack[T] := P;
                                  P := Adr
                  |
                  (PRE Op 2)    : T := T + 2;  (* Binaere Operatoren *)
                                  stack[T - 1] := P;
                                  stack[T] := Op;
                                  P := 4
                  |
                  (PRE if 3)    : T := T + 1;  (*    Operator if     *)
```

```
                         stack[T] := P;
                         P := 13
          |
          (PRE Op 1)   : T := T + 2;  (* Unaere Operatoren  *)
                         stack[T - 1] := P;
                         stack[T] := Op;
                         P := 19
      end
```

`Call` kommt in einem Zielprogramm immer nach `Unwind` vor. Bei der Durchführung von `Call` ist folglich `heap[stack[T]]` weder eine `APP`- noch eine `IND`-Zelle, sondern eine `VAL`-, eine `DEF`- oder eine `PRE`-Zelle. Weder Keller noch Halde werden von `Call` verändert, wenn `heap[stack[T]]` eine `VAL`-Zelle ist.

Return.

```
Return        P := stack[T - 1];
              stack[T - 1] := stack[T];
              T := T - 1
```

`Return` hat also die gleiche Wirkung wie der Fall `(DEF f N Adr)` von `Reduce`.

Push predefined function. Zum laden einer vordefinierten Funktion `Op` auf den Keller.

```
Pushpre Op    T := T + 1;
              stack[T] := new(PRE Op arity(Op))
```

Die Funktion `new` wird erweitert, um auch Haldenknoten der Gestalt `(PRE Op n)` erzeugen zu können.

Update. Kellerverwaltung bei vordefinierten Funktionen und Änderung einer Haldenzelle.

```
Update Arg    case Arg of
              op:  Aux := stack[T - 2]; (*      Operatoren       *)
                   stack[T - 2] := stack[T - 1]; (* Rückkehradresse *)
                   stack[T - 1] := Aux;
                   T := T - 1
              |
              n:                         (* n-stellige Funktionen *)
                   heap[stack[T - n - 2]] := IND heap[stack[T]]
              end
```

Operator. Das Argument `op` = `1`, `2` oder `if` gibt an, ob es sich um einen einstelligen, zweistlligen Operator oder um den If-Operator handelt.

```
Operator op   case op of
                   1:  Op := stack[T - 1];
                       stack[T - 4] := stack[T - 2]; (* Rückkehradresse *)
```

```
                stack[T - 3] := new(VAL Zahl Op stack[T]);
                T := T - 3
            |
            2:  Op := stack[T - 2];
                stack[T - 5] := stack[T - 3]; (* Rückkehradresse *)
                stack[T - 4] :=
                   new(VAL Zahl stack[T] Op stack[T - 1]);
                T := T - 4
            |
            if: stack[T - 5] := stack[T - 2]; (* Rückkehradresse *)
                if heap[stack[T]] = true
                then stack[T - 4] :=
                       add2arg(heap[stack[T - 5]]))
                else stack[T - 4] :=
                       add2arg(heap[stack[T - 6]]);
                     T := T - 4
      end
```

Übersetzungsschemata.
Das Zielprogramm fängt mit der folgenden Befehlsfolge an:

```
 c0: Reset
     Pushfun main
     Call
     Halt

 c4: Pushparam 2  (* Binäroperator *)
     Unwind
     Call
     Pushparam 4
     Unwind
     Call
     Operator 2
     Update op
     Return

c13: Pushparam    (* if *)
     Unwind
     Call
     Operator if
     Update op
     Return

c19: Pushparam 2   (* Unäroperator *)
     Unwind
     Call
     Operator 1
     Update op
     Return
```

Übersetzung einer Funktionsdefinition `f x`$_1 \ldots$ `x`$_n = A$:

$$
\begin{array}{lll}
\text{Üb}_{\text{Def}}(A, Pos, n) & := & \text{Üb}_{\text{Kons}}(\text{A, Pos}) \\
 & & \texttt{Update}\ \ n \\
 & & \texttt{Slide}\ \ n+1 \\
 & & \texttt{Unwind} \\
 & & \texttt{Call} \\
 & & \texttt{Return}
\end{array}
$$

Die Übersetzungsschemata Üb_{Kons}, die im vorangehenden Abschnitt definiert sind, gelten noch.

Es wird angenommen, dass die vordefinierten Funktionen von F, d. h. die Operatoren, (*, +, if, not, usw.) vor der Übersetzung nach MF präfix dargestellt werden. Die Umwandlung der vordefinierten Funktionen von Infix- nach Präfixoperatoren kann während der Syntaxanalyse geleistet werden.

Beispiel. Betrachten wir das folgende F-Programm:

```
main = quadrat (quadrat (* 3 1))
quadrat x = x * x
```

Dieses Programm wird wird folgt übersetzt:

```
 c0: Reset
     Pushfun main
     Call
     Halt

 c4: Pushparam 2  (* Binäroperator *)
     Unwind
     Call
     Pushparam 4
     Unwind
     Call
     Operator 2
     Update op
     Return

c13: Pushparam 2   (* if *)
     Unwind
     Call
     Operator if
     Update op
     Return

c19: Pushparam 2   (* Unäroperator *)
     Unwind
     Call
     Operator 1
     Update op
     Return
```

```
c25: Pushval Zahl 1         h0: DEF main 0 c25
     Pushval Zahl 3
     Pushpre *
     Makeapp
     Makeapp
     Pushfun quadrat
     Makeapp
     Pushfun quadrat
     Makeapp
     Update 0
     Slide 1
     Unwind
     Call
     Return

c39: Pushparam 1            h1: DEF quadrat 1 c39
     Pushparam 2
     Pushpre *
     Makeapp
     Makeapp
     Update 1
     Slide
     Unwind
     Call
     Return
```

Ausführung:

```
Register            stack     heap

I: Reset
T: -1
P: c1

I: Pushfun main     s0: h0    h0: DEF main 0 c25
T: s0                         h1: DEF quadrat 1 c39
P: c2

I: Call             s0: h0    h0: DEF main 0 c25
T: s1               s1: c3    h1: DEF quadrat 1 c39
P: c25

I: Pushval Zahl 1   s0: h0    h0: DEF main 0 c25
T: s2               s1: c3    h1: DEF quadrat 1 c39
P: c26              s2: h2    h2: VAL Zahl 1

I: Pushval Zahl 3   s0: h0    h0: DEF main 0 c25
T: s3               s1: c3    h1: DEF quadrat 1 c39
P: c27              s2: h2    h2: VAL Zahl 1
                    s3: h3    h3: VAL Zahl 3

I: Pushpre *        s0: h0    h0: DEF main 0 c25
T: s4               s1: c3    h1: DEF quadrat 1 c39
P: c28              s2: h2    h2: VAL Zahl 1
                    s3: h3    h3: VAL Zahl 3
                    s4: h4    h4: PRE * 2
```

```
I: Makeapp          s0: h0    h0: DEF main 0 c25
T: s3               s1: c3    h1: DEF quadrat 1 c39
P: c29              s2: h2    h2: VAL Zahl 1
                    s3: h5    h3: VAL Zahl 3           h5: APP
                              h4: PRE * 2                 /   \
                              h5: APP h4 h3          h4: *     3

I: Makeapp          s0: h0    h0: DEF main 0 c25
T: s2               s1: c3    h1: DEF quadrat 1 c39
P: c30              s2: h6    h2: VAL Zahl 1              h6: APP
                              h3: VAL Zahl 3                 /   \
                              h4: PRE * 2              h5: APP    1
                              h5: APP h4 h3               /   \
                              h6: APP h5 h2          h4: *     3

I: Pushfun quadrat  s0: h0    h0: DEF main 0 c25
T: s3               s1: c3    h1: DEF quadrat 1 c39
P: c31              s2: h6    h2: VAL Zahl 1
                    s3: h1    h3: VAL Zahl 3
                              h4: PRE * 2
                              h5: APP h4 h3
                              h6: APP h5 h2

I: Makeapp          s0: h0    h0: DEF main 0 c25
T: s2               s1: c3    h1: DEF quadrat 1 c39       h7: APP
P: c32              s2: h7    h2: VAL Zahl 1              /      \
                              h3: VAL Zahl 3          quadrat  h6: APP
                                                                  /   \
                              h5: APP h4 h3                  h5: APP   1
                              h6: APP h5 h2                     /   \
                              h7: APP h1 h6               h4: *     3

I: Pushfun quadrat  s0: h0    h0: DEF main 0 c25
T: s3               s1: c3    h1: DEF quadrat 1 c39
P: c33              s2: h7    h2: VAL Zahl 1
                    s3: h1    h3: VAL Zahl 3
                              h4: PRE * 2
                              h5: APP h4 h3
                              h6: APP h5 h2
                              h7: APP h1 h6

                                                          h8: APP
I: Makeapp          s0: h0    h0: DEF main 0 c25         /       \
T: s2               s1: c3    h1: DEF quadrat 1 c39  quadrat  h7: APP
P: c34              s2: h8    h2: VAL Zahl 1                 /      \
                              h3: VAL Zahl 3           quadrat  h6: APP
                              h4: PRE * 2                          /  \
                              h5: APP h4 h3                  h5: APP  1
                              h6: APP h5 h2                     /   \
                              h7: APP h1 h6                h4: *     3
                              h8: APP h1 h7

                                                           h8: APP
I: Update 0         s0: h0    h0: DEF main 0 c25          /       \
T: s2               s1: c3    h1: DEF quadrat 1 c39  quadrat  h7: APP
P: c35              s2: h8    h2: VAL Zahl 1                  /      \
                              h3: VAL Zahl 3            quadrat  h6: APP
                              h4: PRE * 2                           /  \
                              h5: APP h4 h3                   h5: APP  1
                              h6: APP h5 h2                      /   \
                              h7: APP h1 h6                 h4: *     3
                              h8: APP h1 h7
```

```
I: Slide 1          s0: c3    h0: DEF main 0 c25
T: s1               s1: h8    h1: DEF quadrat 1 c39
P: c36                        h2: VAL Zahl 1
                              h3: VAL Zahl 3
                              h4: PRE * 2
                              h5: APP h4 h3
                              h6: APP h5 h2
                              h7: APP h1 h6
                              h8: APP h1 h7

I: Unwind           s0: c3    h0: DEF main 0 c25
T: s2               s1: h8    h1: DEF quadrat 1 c39
P: c37              s2: h1    h2: VAL Zahl 1
                              h3: VAL Zahl 3
                              h4: PRE * 2
                              h5: APP h4 h3
                              h6: APP h5 h2
                              h7: APP h1 h6
                              h8: APP h1 h7

I: Unwind           s0: c3    h0: DEF main 0 c25
T: s2               s1: h8    h1: DEF quadrat 1 c39
P: c38              s2: h1    h2: VAL Zahl 1
                              h3: VAL Zahl 3
                              h4: PRE * 2
                              h5: APP h4 h3
                              h6: APP h5 h2
                              h7: APP h1 h6
                              h8: APP h1 h7

I: Call             s0: c3    h0: DEF main 0 c25
T: s3               s1: h8    h1: DEF quadrat 1 c39
P: c39              s2: h1    h2: VAL Zahl 1
                    s3: c38   h3: VAL Zahl 3
                              h4: PRE * 2
                              h5: APP h4 h3
                              h6: APP h5 h2
                              h7: APP h1 h6
                              h8: APP h1 h7

I: Pushparam 1      s0: c3    h0: DEF main 0 c25          h8: APP
T: s4               s1: h8    h1: DEF quadrat 1 c39      /       \
P: c40              s2: h1    h2: VAL Zahl 1        quadrat  h7: APP
                    s3: c38   h3: VAL Zahl 3                /       \
                    s4: h7    h4: PRE * 2             quadrat  h6: APP
                              h5: APP h4 h3                        /   \
                              h6: APP h5 h2                   h5: APP   1
                              h7: APP h1 h6                       /   \
                              h8: APP h1 h7                  h4: *     3

I: Pushparam 2      s0: c3    h0: DEF main 0 c25
T: s5               s1: h8    h1: DEF quadrat 1 c39
P: c41              s2: h1    h2: VAL Zahl 1
                    s3: c38   h3: VAL Zahl 3
                    s4: h7    h4: PRE * 2
                    s5: h7    h5: APP h4 h3
                              h6: APP h5 h2
                              h7: APP h1 h6
                              h8: APP h1 h7
```

```
I: Pushpre *        s0: c3    h0: DEF main 0 c25
T: s6               s1: h8    h1: DEF quadrat 1 c39
P: c42              s2: h1    h2: VAL Zahl 1
                    s3: c38   h3: VAL Zahl 3
                    s4: h7    h4: PRE * 2
                    s5: h7    h5: APP h4 h3
                    s6: h9    h6: APP h5 h2
                              h7: APP h1 h6
                              h8: APP h1 h7
                              h9: PRE * 2
```

```
                                                         h10: APP
I: Makeapp          s0: c3    h0: DEF main 0 c25           /     \
T: s5               s1: h8    h1: DEF quadrat 1 c39   h9: *       \
P: c43              s2: h1    h2: VAL Zahl 1               h8: APP |
                    s3: c38   h3: VAL Zahl 3              /       \|
                    s4: h7    h4: PRE * 2         quadrat   h7: APP
                    s5: h10   h5: APP h4 h3                /      \
                              h6: APP h5 h2       quadrat  h6: APP
                              h7: APP h1 h6                      /   \
                              h8: APP h1 h7                h5: APP   1
                              h9: PRE * 2                     /   \
                             h10: APP h9 h7               h4: *     3
```

```
                                                           h11: APP
                                                               /   \
                                                          h10: APP  \
                                                        /         \  \
I: Makeapp          s0: c3    h0: DEF main 0 c25     h9: *         \ |
T: s4               s1: h8    h1: DEF quadrat 1 c39                | |
P: c44              s2: h1    h2: VAL Zahl 1               h8: APP | |
                    s3: c38   h3: VAL Zahl 3              /       \| |
                    s4: h11   h4: PRE * 2          quadrat   h7: APP
                              h5: APP h4 h3                   /      \
                              h6: APP h5 h2          quadrat  h6: APP
                              h7: APP h1 h6                      /   \
                              h8: APP h1 h7                h5: APP    1
                              h9: PRE * 2                     /   \
                             h10: APP h9 h7               h4: *     3
                             h11: APP h10 h7
```

```
                                                              h8
                                                               |
I: Update 1         s0: c3    h0: DEF main 0 c25         h11: APP
T: s4               s1: h8    h1: DEF quadrat 1 c39          /   \
P: c45              s2: h1    h2: VAL Zahl 1            h10: APP  \
                    s3: c38   h3: VAL Zahl 3          /         \  \
                    s4: h11   h4: PRE * 2          h9: *        | |
                              h5: APP h4 h3                  h7: APP
                              h6: APP h5 h2                 /      \
                              h7: APP h1 h6       quadrat  h6: APP
                              h8: IND h11                        /   \
                              h9: PRE * 2                  h5: APP    1
                             h10: APP h9 h7                   /   \
                             h11: APP h10 h7              h4: *     3
```

```
I: Slide            s0: c3    h0: DEF main 0 c25
T: s2               s1: c38   h1: DEF quadrat 1 c39
P: c46              s2: h11   h2: VAL Zahl 1
                              h3: VAL Zahl 3
                              h4: PRE * 2
                              h5: APP h4 h3
                              h6: APP h5 h2
                              h7: APP h1 h6
                              h8:
                              h9: PRE * 2
                             h10: APP h9 h7
                             h11: APP h10 h7

I: Unwind           s0: c3    h0: DEF main 0 c25
T: s3               s1: c38   h1: DEF quadrat 1 c39
P: c47              s2: h11   h2: VAL Zahl 1
                    s3: h10   h3: VAL Zahl 3
                              h4: PRE * 2
                              h5: APP h4 h3
                              h6: APP h5 h2
                              h7: APP h1 h6
                              h8:
                              h9: PRE * 2
                             h10: APP h9 h7
                             h11: APP h10 h7

I: Unwind           s0: c3    h0: DEF main 0 c25
T: s4               s1: c38   h1: DEF quadrat 1 c39
P: c47              s2: h11   h2: VAL Zahl 1
                    s3: h10   h3: VAL Zahl 3
                    s4: h9    h4: PRE * 2
                              h5: APP h4 h3
                              h6: APP h5 h2
                              h7: APP h1 h6
                              h8:
                              h9: PRE * 2
                             h10: APP h9 h7
                             h11: APP h10 h7

I: Unwind           s0: c3    h0: DEF main 0 c25
T: s4               s1: c38   h1: DEF quadrat 1 c39
P: c47              s2: h11   h2: VAL Zahl 1
                    s3: h10   h3: VAL Zahl 3
                    s4: h9    h4: PRE * 2
                              h5: APP h4 h3
                              h6: APP h5 h2
                              h7: APP h1 h6
                              h8:
                              h9: PRE * 2
                             h10: APP h9 h7
                             h11: APP h10 h7

I: Call             s0: c3    h0: DEF main 0 c25
T: s6               s1: c38   h1: DEF quadrat 1 c39
P: c48              s2: h11   h2: VAL Zahl 1
                    s3: h10   h3: VAL Zahl 3
                    s4: h9    h4: PRE * 2
                    s5: c48   h5: APP h4 h3
                    s6: *     h6: APP h5 h2
                              h7: APP h1 h6
                              h8:
                              h9: PRE * 2
                             h10: APP h9 h7
                             h11: APP h10 h7
```

```
I: Pushparam 2      s0: c3    h0: DEF main 0 c25         h11: APP
T: s7               s1: c38   h1: DEF quadrat 1 c39          /   \
P: c5               s2: h11   h2: VAL Zahl 1             h10: APP \
                    s3: h10   h3: VAL Zahl 3               /      \ \
                    s4: h9    h4: PRE * 2              h9: *       | |
                    s5: c48   h5: APP h4 h3                    h7: APP
                    s6: *     h6: APP h5 h2                     /    \
                    s7: h7    h7: APP h1 h6              quadrat  h6: APP
                              h8:                                  /   \
                              h9: PRE * 2                      h5: APP   1
                             h10: APP h9 h7                       /   \
                             h11: APP h10 h7                  h4: *     3
```

```
I: Unwind           s0: c3    h0: DEF main 0 c25        h11: APP
T: s8               s1: c38   h1: DEF quadrat 1 c39         /   \
P: c5               s2: h11   h2: VAL Zahl 1            h10: APP \
                    s3: h10   h3: VAL Zahl 3             /       \ \
                    s4: h9    h4: PRE * 2             h9: *       | |
                    s5: c48   h5: APP h4 h3                   h7: APP
                    s6: *     h6: APP h5 h2                   /      \
                    s7: h7    h7: APP h1 h6             quadrat  h6: APP
                    s8: h1    h8:                                  /   \
                              h9: PRE * 2                     h5: APP   1
                             h10: APP h9 h7                      /   \
                             h11: APP h10 h7              h4: *     3
```

```
I: Unwind           s0: c3    h0: DEF main 0 c25          h11: APP
T: s8               s1: c38   h1: DEF quadrat 1 c39           /   \
P: c6               s2: h11   h2: VAL Zahl 1              h10: APP \
                    s3: h10   h3: VAL Zahl 3               /       \ \
                    s4: h9    h4: PRE * 2              h9: *        | |
                    s5: c48   h5: APP h4 h3                     h7: APP
                    s6: *     h6: APP h5 h2                     /      \
                    s7: h7    h7: APP h1 h6              quadrat  h6: APP
                    s8: h1    h8:                                   /   \
                              h9: PRE * 2                     h5: APP    1
                             h10: APP h9 h7                      /   \
                             h11: APP h10 h7               h4: *     3
```

```
I: Call             s0: c3    h0: DEF main 0 c25
T: s9               s1: c38   h1: DEF quadrat 1 c39
P: c39              s2: h11   h2: VAL Zahl 1
                    s3: h10   h3: VAL Zahl 3
                    s4: h9    h4: PRE * 2
                    s5: c48   h5: APP h4 h3
                    s6: *     h6: APP h5 h2
                    s7: h7    h7: APP h1 h6
                    s8: h1    h8:
                    s9: c7    h9: PRE * 2
                             h10: APP h9 h7
                             h11: APP h10 h7
```

```
I: Pushparam 1      s0: c3    h0: DEF main 0 c25        h11: APP
T: s10              s1: c38   h1: DEF quadrat 1 c39         /   \
P: c40              s2: h11   h2: VAL Zahl 1            h10: APP \
                    s3: h10   h3: VAL Zahl 3            /       \ \
                    s4: h9    h4: PRE * 2            h9: *       | |
                    s5: c48   h5: APP h4 h3                  h7: APP
                    s6: *     h6: APP h5 h2                 /       \
                    s7: h7    h7: APP h1 h6           quadrat   h6: APP
                    s8: h1    h8:                                 /   \
                    s9: c7    h9: PRE * 2                    h5: APP   1
                   s10: h6   h10: APP h9 h7                     /   \
                             h11: APP h10 h7              h4: *     3
```

```
I: Pushparam 2      s0: c3    h0: DEF main 0 c25
T: s11              s1: c38   h1: DEF quadrat 1 c39
P: c41              s2: h11   h2: VAL Zahl 1
                    s3: h10   h3: VAL Zahl 3
                    s4: h9    h4: PRE * 2
                    s5: c48   h5: APP h4 h3
                    s6: *     h6: APP h5 h2
                    s7: h7    h7: APP h1 h6
                    s8: h1    h8:
                    s9: c7    h9: PRE * 2
                   s10: h6   h10: APP h9 h7
                   s11: h6   h11: APP h10 h7

I: Pushpre *        s0: c3    h0: DEF main 0 c25
T: s12              s1: c38   h1: DEF quadrat 1 c39
P: c42              s2: h11   h2: VAL Zahl 1
                    s3: h10   h3: VAL Zahl 3
                    s4: h9    h4: PRE * 2
                    s5: c48   h5: APP h4 h3
                    s6: *     h6: APP h5 h2
                    s7: h7    h7: APP h1 h6
                    s8: h1    h8:
                    s9: c7    h9: PRE * 2
                   s10: h6   h10: APP h9 h7
                   s11: h6   h11: APP h10 h7
                   s12: h12  h12: PRE * 2

I: Makeapp          s0: c3    h0: DEF main 0 c25        h13: APP
T: s11              s1: c38   h1: DEF quadrat 1 c39    /        \
P: c43              s2: h11   h2: VAL Zahl 1      h12:*  h11:APP \
                    s3: h10   h3: VAL Zahl 3                /   \ \
                    s4: h9    h4: PRE * 2             h10: APP   \ \
                    s5: c48   h5: APP h4 h3               /   \  | |
                    s6: *     h6: APP h5 h2          h9: *     | | |
                    s7: h7    h7: APP h1 h6                h7: APP |
                    s8: h1    h8:                          /      \|
                    s9: c7    h9: PRE * 2           quadrat  h6: APP
                   s10: h6   h10: APP h9 h7                     /   \
                   s11: h13  h11: APP h10 h7              h5: APP   1
                             h12: PRE * 2                    /   \
                             h13: APP h12 h6            h4: *     3


                                                         h14: APP
I: Makeapp          s0: c3    h0: DEF main 0 c25          /     \
T: s10              s1: c38   h1: DEF quadrat 1 c39   h13: APP   \
P: c44              s2: h11   h2: VAL Zahl 1         /         \ \
                    s3: h10   h3: VAL Zahl 3     h12:*  h11:APP \  \
                    s4: h9    h4: PRE * 2                  /   \ \  \
                    s5: c48   h5: APP h4 h3         h10: APP    \ \ |
                    s6: *     h6: APP h5 h2        /      \     | | |
                    s7: h7    h7: APP h1 h6    h9: *      |     | | |
                    s8: h1    h8:                         h7: APP | |
                    s9: c7    h9: PRE * 2                 /      \| |
                   s10: h14  h10: APP h9 h7        quadrat     h6: APP
                             h11: APP h10 h7                      /   \
                             h12: PRE * 2                  h5: APP    1
                             h13: APP h12 h6                  /   \
                             h14: APP h13 h6            h4: *     3
```

```
                                                            h7
                                                             |
I: Update 1         s0: c3    h0: DEF main 0 c25             v
T: s10              s1: c38   h1: DEF quadrat 1 c39     h14: APP
P: c45              s2: h11   h2: VAL Zahl 1           /        \
                    s3: h10   h3: VAL Zahl 3        h13: APP     \
                    s4: h9    h4: PRE * 2       /            \  \
                    s5: c48   h5: APP h4 h    h12:*  h11:APP    \  \
                    s6: *     h6: APP h5 h2          /   \     \  \
                    s7: h7    h7: IND h14         h10: APP \     \ |
                    s8: h1    h8:                /        \ \   | |
                    s9: c7    h9: PRE * 2     h9: *       | |   | |
                   s10: h14  h10: APP h9 h7                h7  h6: APP
                             h11: APP h10 h7                      /   \
                             h12: PRE * 2                   h5: APP   1
                             h13: APP h12 h6                   /   \
                             h14: APP h13 h6              h4: *     3

I: Slide            s0: c3    h0: DEF main 0 c25
T: s8               s1: c38   h1: DEF quadrat 1 c39
P: c46              s2: h11   h2: VAL Zahl 1
                    s3: h10   h3: VAL Zahl 3
                    s4: h9    h4: PRE * 2
                    s5: c48   h5: APP h4 h3
                    s6: *     h6: APP h5 h2
                    s7: c7    h7: IND h14
                    s8: h14   h8:
                              h9: PRE * 2
                             h10: APP h9 h7
                             h11: APP h10 h7
                             h12: PRE * 2
                             h13: APP h12 h6
                             h14: APP h13 h6


                                                          h7
                                                           |
I: Unwind           s0: c3    h0: DEF main 0 c25           v
T: s9               s1: c38   h1: DEF quadrat 1 c39   h14: APP
P: c46              s2: h11   h2: VAL Zahl 1         /        \
                    s3: h10   h3: VAL Zahl 3      h13: APP     \
                    s4: h9    h4: PRE * 2     /            \  \
                    s5: c48   h5: APP h4 h3  h12:*  h11:APP  \  \
                    s6: *     h6: APP h5 h2         /   \     \  \
                    s7: c7    h7: IND h14       h10: APP \     \ |
                    s8: h14   h8:              /        \ \   | |
                    s9: h13   h9: PRE * 2   h9: *       | |   | |
                             h10: APP h9 h7              h7  h6: APP
                             h11: APP h10 h7                    /   \
                             h12: PRE * 2                 h5: APP    1
                             h13: APP h12 h6                 /   \
                             h14: APP h13 h6            h4: *     3

I: Unwind           s0: c3    h0: DEF main 0 c25
T: s10              s1: c38   h1: DEF quadrat 1 c39
P: c46              s2: h11   h2: VAL Zahl 1
                    s3: h10   h3: VAL Zahl 3
                    s4: h9    h4: PRE * 2
                    s5: c48   h5: APP h4 h3
                    s6: *     h6: APP h5 h2
                    s7: c7    h7: IND h14
                    s8: h14   h8:
```

```
                   s9: h13   h9: PRE * 2
                  s10: h12  h10: APP h9 h7
                            h11: APP h10 h7
                            h12: PRE * 2
                            h13: APP h12 h6
                            h14: APP h13 h6

I: Unwind          s0: c3    h0: DEF main 0 c25
T: s10             s1: c38   h1: DEF quadrat 1 c39
P: c47             s2: h11   h2: VAL Zahl 1
                   s3: h10   h3: VAL Zahl 3
                   s4: h9    h4: PRE * 2
                   s5: c48   h5: APP h4 h3
                   s6: *     h6: APP h5 h2
                   s7: c7    h7: IND h14
                   s8: h14   h8:
                   s9: h13   h9: PRE * 2
                  s10: h12  h10: APP h9 h7
                            h11: APP h10 h7
                            h12: PRE * 2
                            h13: APP h12 h6
                            h14: APP h13 h6

I: Call            s0: c3    h0: DEF main 0 c25
T: s12             s1: c38   h1: DEF quadrat 1 c39
P: c4              s2: h11   h2: VAL Zahl 1
                   s3: h10   h3: VAL Zahl 3
                   s4: h9    h4: PRE * 2
                   s5: c48   h5: APP h4 h3
                   s6: *     h6: APP h5 h2
                   s7: c7    h7: IND h14
                   s8: h14   h8:
                   s9: h13   h9: PRE * 2
                  s10: h12  h10: APP h9 h7
                  s11: c48  h11: APP h10 h7
                  s12: *    h12: PRE * 2
                            h13: APP h12 h6
                            h14: APP h13 h6

I: Pushparam 2     s0: c3    h0: DEF main 0 c25
T: s13             s1: c38   h1: DEF quadrat 1 c39
P: c5              s2: h11   h2: VAL Zahl 1
                   s3: h10   h3: VAL Zahl 3
                   s4: h9    h4: PRE * 2
                   s5: c48   h5: APP h4 h3
                   s6: *     h6: APP h5 h2
                   s7: c7    h7: IND h14
                   s8: h14   h8:
                   s9: h13   h9: PRE * 2
                  s10: h12  h10: APP h9 h7
                  s11: c48  h11: APP h10 h7
                  s12: *    h12: PRE * 2
                  s13: h6   h13: APP h12 h6
                            h14: APP h13 h6

I: Unwind          s0: c3    h0: DEF main 0 c25
T: s14             s1: c38   h1: DEF quadrat 1 c39
P: c5              s2: h11   h2: VAL Zahl 1
                   s3: h10   h3: VAL Zahl 3
                   s4: h9    h4: PRE * 2
                   s5: c48   h5: APP h4 h3
                   s6: *     h6: APP h5 h2
                   s7: c7    h7: IND h14
                   s8: h14   h8:
```

```
                     s9: h13   h9: PRE * 2
                    s10: h12  h10: APP h9 h7
                    s11: c48  h11: APP h10 h7
                    s12: *    h12: PRE * 2
                    s13: h6   h13: APP h12 h6
                    s14: h5   h14: APP h13 h6

I: Unwind            s0: c3    h0: DEF main 0 c25
T: s15               s1: c38   h1: DEF quadrat 1 c39
P: c5                s2: h11   h2: VAL Zahl 1
                     s3: h10   h3: VAL Zahl 3
                     s4: h9    h4: PRE * 2
                     s5: c48   h5: APP h4 h3
                     s6: *     h6: APP h5 h2
                     s7: c7    h7: IND h14
                     s8: h14   h8:
                     s9: h13   h9: PRE * 2
                    s10: h12  h10: APP h9 h7
                    s11: c48  h11: APP h10 h7
                    s12: *    h12: PRE * 2
                    s13: h6   h13: APP h12 h6
                    s14: h5   h14: APP h13 h6
                    s15: h4

I: Unwind            s0: c3    h0: DEF main 0 c25
T: s15               s1: c38   h1: DEF quadrat 1 c39
P: c6                s2: h11   h2: VAL Zahl 1
                     s3: h10   h3: VAL Zahl 3
                     s4: h9    h4: PRE * 2
                     s5: c48   h5: APP h4 h3
                     s6: *     h6: APP h5 h2
                     s7: c7    h7: IND h14
                     s8: h14   h8:
                     s9: h13   h9: PRE * 2
                    s10: h12  h10: APP h9 h7
                    s11: c48  h11: APP h10 h7
                    s12: *    h12: PRE * 2
                    s13: h6   h13: APP h12 h6
                    s14: h5   h14: APP h13 h6
                    s15: h4

I: Call              s0: c3    h0: DEF main 0 c25
T: s17               s1: c38   h1: DEF quadrat 1 c39
P: c4                s2: h11   h2: VAL Zahl 1             h7
                     s3: h10   h3: VAL Zahl 3             |
                     s4: h9    h4: PRE * 2            h14: APP
                     s5: c48   h5: APP h4 h3         /        \
                     s6: *     h6: APP h5 h2       h13: APP    \
                     s7: c7    h7: IND h14      /            \  \
                     s8: h14   h8:          h12:*   h11:APP    \  \
                     s9: h13   h9: PRE * 2          /     \     \  \
                    s10: h12  h10: APP h9 h7     h10: APP  \     \ |
                    s11: c48  h11: APP h10 h7     /       \ \   | |
                    s12: *    h12: PRE * 2     h9:*        ||    | |
                    s13: h6   h13: APP h12 h6              h7   h6: APP
                    s14: h5   h14: APP h13 h6                      /   \
                    s15: h4                                  h5: APP   1
                    s16: c7                                     /   \
                    s17: *                                 h4: *     3
```

```
I: Pushparam 2       s0: c3    h0: DEF main 0 c25
T: s18               s1: c38   h1: DEF quadrat 1 c39
P: c5                s2: h11   h2: VAL Zahl 1
                     s3: h10   h3: VAL Zahl 3
                     s4: h9    h4: PRE * 2
                     s5: c48   h5: APP h4 h3
                     s6: *     h6: APP h5 h2
                     s7: c7    h7: IND h14
                     s8: h14   h8:
                     s9: h13   h9: PRE * 2
                    s10: h12  h10: APP h9 h7
                    s11: c48  h11: APP h10 h7
                    s12: *    h12: PRE * 2
                    s13: h6   h13: APP h12 h6
                    s14: h5   h14: APP h13 h6
                    s15: h4
                    s16: c7
                    s17: *
                    s18: h3

I: Unwind            s0: c3    h0: DEF main 0 c25
T: s18               s1: c38   h1: DEF quadrat 1 c39
P: c6                s2: h11   h2: VAL Zahl 1
                     s3: h10   h3: VAL Zahl 3
                     s4: h9    h4: PRE * 2
                     s5: c48   h5: APP h4 h3
                     s6: *     h6: APP h5 h2
                     s7: c7    h7: IND h14
                     s8: h14   h8:
                     s9: h13   h9: PRE * 2
                    s10: h12  h10: APP h9 h7
                    s11: c48  h11: APP h10 h7
                    s12: *    h12: PRE * 2
                    s13: h6   h13: APP h12 h6
                    s14: h5   h14: APP h13 h6
                    s15: h4
                    s16: c7
                    s17: *
                    s18: h3

I: Call              s0: c3    h0: DEF main 0 c25
T: s18               s1: c38   h1: DEF quadrat 1 c39
P: c7                s2: h11   h2: VAL Zahl 1
                     s3: h10   h3: VAL Zahl 3
                     s4: h9    h4: PRE * 2
                     s5: c48   h5: APP h4 h3
                     s6: *     h6: APP h5 h2
                     s7: c7    h7: IND h14
                     s8: h14   h8:
                     s9: h13   h9: PRE * 2
                    s10: h12  h10: APP h9 h7
                    s11: c48  h11: APP h10 h7
                    s12: *    h12: PRE * 2
                    s13: h6   h13: APP h12 h6
                    s14: h5   h14: APP h13 h6
                    s15: h4
                    s16: c7
                    s17: *
                    s18: h3
```

```
I: Pushparam 4      s0: c3    h0: DEF main 0 c25
T: s19              s1: c38   h1: DEF quadrat 1 c39
P: c8               s2: h11   h2: VAL Zahl 1
                    s3: h10   h3: VAL Zahl 3               h7
                    s4: h9    h4: PRE * 2                   |
                    s5: c48   h5: APP h4 h3                 v
                    s6: *     h6: APP h5 h2            h14: APP
                    s7: c7    h7: IND h14                 /    \
                    s8: h14   h8:                    h13: APP  \
                    s9: h13   h9: PRE * 2           /        \  \
                   s10: h12  h10: APP h9 h7    h12:*  h11:APP  \  \
                   s11: c48  h11: APP h10 h7           /    \   \ |
                   s12: *    h12: PRE * 2         h10: APP  |   | |
                   s13: h6   h13: APP h12 h6        /      \|   | |
                   s14: h5   h14: APP h13 h6      h9: *    ||   | |
                   s15: h4                                 h7  h6:APP
                   s16: c7                                       /   \
                   s17: *                                  h5: APP   1
                   s18: h3                                    /   \
                   s19: h2                               h4: *     3
```

```
I: Unwind           s0: c3    h0: DEF main 0 c25
T: s19              s1: c38   h1: DEF quadrat 1 c39
P: c9               s2: h11   h2: VAL Zahl 1
                    s3: h10   h3: VAL Zahl 3
                    s4: h9    h4: PRE * 2
                    s5: c48   h5: APP h4 h3
                    s6: *     h6: APP h5 h2
                    s7: c7    h7: IND h14
                    s8: h14   h8:
                    s9: h13   h9: PRE * 2
                   s10: h12  h10: APP h9 h7
                   s11: c48  h11: APP h10 h7
                   s12: *    h12: PRE * 2
                   s13: h6   h13: APP h12 h6
                   s14: h5   h14: APP h13 h6
                   s15: h4
                   s16: c7
                   s17: *
                   s18: h3
                   s19: h2
```

```
I: Call             s0: c3    h0: DEF main 0 c25
T: s19              s1: c38   h1: DEF quadrat 1 c39
P: c10              s2: h11   h2: VAL Zahl 1
                    s3: h10   h3: VAL Zahl 3
                    s4: h9    h4: PRE * 2
                    s5: c48   h5: APP h4 h3
                    s6: *     h6: APP h5 h2
                    s7: c7    h7: IND h14
                    s8: h14   h8:
                    s9: h13   h9: PRE * 2
                   s10: h12  h10: APP h9 h7
                   s11: c48  h11: APP h10 h7
                   s12: *    h12: PRE * 2
                   s13: h6   h13: APP h12 h6
                   s14: h5   h14: APP h13 h6
                   s15: h4
                   s16: c7
                   s17: *
                   s18: h3
                   s19: h2
```

```
I: Operator 2       s0: c3    h0: DEF main 0 c25
T: s15              s1: c38   h1: DEF quadrat 1 c39
P: c11              s2: h11   h2: VAL Zahl 1
                    s3: h10   h3: VAL Zahl 3
                    s4: h9    h4: PRE * 2
                    s5: c48   h5: APP h4 h3
                    s6: *     h6: APP h5 h2
                    s7: c7    h7: IND h14
                    s8: h14   h8:
                    s9: h13   h9: PRE * 2
                   s10: h12  h10: APP h9 h7
                   s11: c48  h11: APP h10 h7
                   s12: *    h12: PRE * 2
                   s13: h6   h13: APP h12 h6
                   s14: c7   h14: APP h13 h6
                   s15: h15  h15: VAL Zahl 3
```

```
                                                           h7
I: Update op        s0: c3    h0: DEF main 0 c25            |
T: s14              s1: c38   h1: DEF quadrat 1 c39      h14: APP
P: c12              s2: h11                                   /   \
                    s3: h10                              h13: APP  \
                    s4: h9                                  /     \ |
                    s5: c48   h5: APP h4 h3           h12:*      h6:3
                    s6: *     h6: VAL Zahl 3
                    s7: c7    h7: IND h14
                    s8: h14   h8:
                    s9: h13   h9: PRE * 2
                   s10: h12  h10: APP h9 h7             h11:APP
                   s11: c48  h11: APP h10 h7                /  \
                   s12: *    h12: PRE * 2             h10: APP  \
                   s13: c7   h13: APP h12 h6             /    \ |
                   s14: h6   h14: APP h13 h6        h9: *      h7
                             h15:
```

```
I: Return           s0: c3    h0: DEF main 0 c25
T: s13              s1: c38   h1: DEF quadrat 1 c39
P: c7               s2: h11   h2:
                    s3: h10   h3:
                    s4: h9    h4:
                    s5: c48   h5:
                    s6: *     h6: VAL Zahl 3
                    s7: c7    h7: IND h14
                    s8: h14   h8:
                    s9: h13   h9: PRE * 2
                   s10: h12  h10: APP h9 h7
                   s11: c48  h11: APP h10 h7
                   s12: *    h12: PRE * 2
                   s13: h6   h13: APP h12 h6
                             h14: APP h13 h6
                             h15:
```

```
I: Pushparam 4      s0: c3    h0: DEF main 0 c25
T: s14              s1: c38   h1: DEF quadrat 1 c39
P: c8               s2: h11   h2:
                    s3: h10   h3:
                    s4: h9    h4:
                    s5: c48   h5:
                    s6: *     h6: VAL Zahl 3
                    s7: c7    h7: IND h14
                    s8: h14   h8:
```

```
                          s9: h13   h9: PRE * 2
                         s10: h12  h10: APP h9 h7
                         s11: c48  h11: APP h10 h7
                         s12: *    h12: PRE * 2
                         s13: h6   h13: APP h12 h6
                         s14: h6   h14: APP h13 h6
                                   h15:

I: Unwind                 s0: c3    h0: DEF main 0 c25
T: s14                    s1: c38   h1: DEF quadrat 1 c39
P: c9                     s2: h11   h2:
                          s3: h10   h3:
                          s4: h9    h4:
                          s5: c48   h5:
                          s6: *     h6: VAL Zahl 3
                          s7: c7    h7: IND h14
                          s8: h14   h8:
                          s9: h13   h9: PRE * 2
                         s10: h12  h10: APP h9 h7
                         s11: c48  h11: APP h10 h7
                         s12: *    h12: PRE * 2
                         s13: h6   h13: APP h12 h6
                         s14: h6   h14: APP h13 h6
                                   h15:

I: Call                   s0: c3    h0: DEF main 0 c25
T: s14                    s1: c38   h1: DEF quadrat 1 c39
P: c10                    s2: h11   h2:
                          s3: h10   h3:
                          s4: h9    h4:
                          s5: c48   h5:
                          s6: *     h6: VAL Zahl 3
                          s7: c7    h7: IND h14
                          s8: h14   h8:
                          s9: h13   h9: PRE * 2
                         s10: h12  h10: APP h9 h7
                         s11: c48  h11: APP h10 h7
                         s12: *    h12: PRE * 2
                         s13: h6   h13: APP h12 h6
                         s14: h6   h14: APP h13 h6
                                   h15:

I: Operator 2             s0: c3    h0: DEF main 0 c25
T: s10                    s1: c38   h1: DEF quadrat 1 c39
P: c11                    s2: h11   h2:
                          s3: h10   h3:
                          s4: h9    h4:
                          s5: c48   h5:
                          s6: *     h6: VAL Zahl 3
                          s7: c7    h7: IND h14
                          s8: h14   h8:
                          s9: c48   h9: PRE * 2
                         s10: h16  h10: APP h9 h7
                                   h11: APP h10 h7
                                   h12: PRE * 2
                                   h13: APP h12 h6
                                   h14: APP h13 h6
                                   h15:
                                   h16: VAL Zahl 9
```

```
I: Update Op        s0: c3    h0: DEF main 0 c25
T: s9               s1: c38   h1: DEF quadrat 1 c39
P: c12              s2: h11   h2:
                    s3: h10   h3:
                    s4: h9    h4:
                    s5: c48   h5:
                    s6: *     h6:
                    s7: c7    h7: IND h14
                    s8: c48   h8:
                    s9: h14   h9: PRE * 2
                             h10: APP h9 h7          h11:APP
                             h11: APP h10 h7            /   \
                             h12:                  h10: APP  \
                             h13:                      /   \ |
                             h14: VAL Zahl 9      h9: *     h7
                             h15:                            |
                             h16:                         h14: 9
```

```
I: Return           s0: c3    h0: DEF main 0 c25
T: s8               s1: c38   h1: DEF quadrat 1 c39
P: c48              s2: h11   h2:
                    s3: h10   h3:
                    s4: h9    h4:
                    s5: c48   h5:
                    s6: *     h6:
                    s7: c7    h7: IND h14
                    s8: h14   h8:
                              h9: PRE * 2
                             h10: APP h9 h7             h11:APP
                             h11: APP h10 h7              /    \
                             h12:                     h10: APP \
                             h13:                         /   \|
                             h14: VAL Zahl 9          h9: *    h7
                             h15:                               |
                             h16:                           h14: 9
```

```
I: Return           s0: c3    h0: DEF main 0 c25
T: s7               s1: c38   h1: DEF quadrat 1 c39
P: c7               s2: h11   h2:
                    s3: h10   h3:
                    s4: h9    h4:
                    s5: c48   h5:
                    s6: *     h6:
                    s7: h14   h7: IND h14
                              h8: IND h11
                              h9: PRE * 2
                             h10: APP h9 h7
                             h11: APP h10 h7
                             h12:
                             h13:
                             h14: VAL Zahl 9
                             h15:
                             h16:
```

```
I: Pushparam 4      s0: c3    h0: DEF main 0 c25
T: s8               s1: c38   h1: DEF quadrat 1 c39
P: c8               s2: h11   h2:
                    s3: h10   h3:
                    s4: h9    h4:
                    s5: c48   h5:
                    s6: *     h6:
                    s7: h14   h7: IND h14
                    s8: h7    h8:                 h11: APP
                              h9: PRE * 2           /     \
                             h10: APP h9 h7       h10: APP  \
                             h11: APP h10 h7          /    \|
                             h12:                h9: *     h7
                             h13:                           |
                             h14: VAL Zahl 9             h14: 9
                             h15:
                             h16:

I: Unwind           s0: c3    h0: DEF main 0 c25
T: s8               s1: c38   h1: DEF quadrat 1 c39
P: c9               s2: h11   h2:
                    s3: h10   h3:
                    s4: h9    h4:
                    s5: c48   h5:
                    s6: *     h6:
                    s7: h14   h7: IND h14
                    s8: h7    h8:                  h11: APP
                              h9: PRE * 2           /     \
                             h10: APP h9 h7       h10: APP  \
                             h11: APP h10 h7          /    \|
                             h12:                h9: *     h7
                             h13:                           |
                             h14: VAL Zahl 9             h14: 9
                             h15:
                             h16:

I: Call             s0: c3    h0: DEF main 0 c25
T: s8               s1: c38   h1: DEF quadrat 1 c39
P: c10              s2: h11   h2:
                    s3: h10   h3:
                    s4: h9    h4:
                    s5: c48   h5:
                    s6: *     h6:
                    s7: h14   h7: IND h14
                    s8: h7    h8:                  h11: APP
                              h9: PRE * 2           /     \
                             h10: APP h9 h7       h10: APP  \
                             h11: APP h10 h7          /    \|
                             h12:                h9: *     h7
                             h13:                           |
                             h14: VAL Zahl 9             h14: 9
                             h15:
                             h16:
```

```
I: Operator 2       s0: c3    h0: DEF main 0 c25
T: s4               s1: c38   h1: DEF quadrat 1 c39
P: c11              s2: h11   h2:
                    s3: c48   h3:
                    s4: h17   h4:
                              h5:
                              h6:
                              h7: IND h14
                              h8:                        h11: APP
                              h9: PRE * 2                 /      \
                             h10: APP h9 h7            h10: APP  \
                             h11: APP h10 h7            /        \|
                             h12:                  h9: *         h7
                             h13:                                 |
                             h14: VAL Zahl 9                    h14: 9
                             h15:
                             h16:
                             h17: VAL Zahl 81

I: Update op        s0: c3    h0: DEF main 0 c25
T: s3               s1: c38   h1: DEF quadrat 1 c39
P: c12              s2: c48   h2:
                    s3: h11   h3:
                              h4:
                              h5:
                              h6:
                              h7:
                              h8:
                              h9:
                             h10:
                             h11: VAL Zahl 81
                             h12:
                             h13:
                             h14:
                             h15:
                             h16:
                             h17:

I: Return           s0: c3    h0: DEF main 0 c25
T: s2               s1: c38   h1: DEF quadrat 1 c39
P: c48              s2: h11   h2:
                              h3:
                              h4:
                              h5:
                              h6:
                              h7:
                              h8:
                              h9:
                             h10:
                             h11: VAL Zahl 81
                             h12:
                             h13:
                             h14:
                             h15:
                             h16:
                             h17:
```

```
I: Return          s0: c3    h0: DEF main 0 c25
T: s1              s1: h11   h1: DEF quadrat 1 c39
P: c38                       h2:
                             h3:
                             h4:
                             h5:
                             h6:
                             h7:
                             h8:
                             h9:
                            h10:
                            h11: VAL Zahl 81
                            h12:
                            h13:
                            h14:
                            h15:
                            h16:
                            h17:

I: Return         s0: h11    h0: DEF main 0 c25
T: s0                        h1: DEF quadrat 1 c39
P: c3                        h2:
                             h3:
                             h4:
                             h5:
                             h6:
                             h7:
                             h8:
                             h9:
                            h10:
                            h11: VAL Zahl 81
                            h12:
                            h13:
                            h14:
                            h15:
                            h16:
                            h17:

I: Halt
```

■

4.7 Spracherweiterungen

4.7.1 λ-Abstraktion

Die verzögerte Auswertung verlangt, dass λ-Ausdrücke in normaler Reihenfolge, d. h. „von außen aus“, ausgewertet werden. Dies ist leicht durchzuführen, wenn alle Lamda-Ausdrücke A die folgende Gestalt haben:

$$\mathrm{A} = \texttt{lambda}\ \ \texttt{x}_1\ \ .\ \texttt{lambda}\ \texttt{x}_2 \ldots\ \ .\ \texttt{lambda}\ \texttt{x}_n\ \ .\ \ B$$

mit:

1. $n \geq 0$, und
2. B ist kein λ-Ausdruck, und
3. alle in B vorkommenden λ-Ausdrücke sind ebenfalls dieser Gestalt, und

4. in A kommen keine freie Variablen vor.

Hat A diese Gestalt, dann heißt A ein *Superkombinator.* Ist A ein Superkombinator, dann kann A durch eine (globale) Funktionsdefinition

$\texttt{\$f x}_1 \ldots\ \texttt{x}_n\ B'$

ersetzt werden, wobei `$f` eine neue, im Programm nicht vorkommende Variable ist und B' sich aus dem rekursiven Ersetzen von in B vorkommenden λ-Ausdrücken durch neue (globale) Funktionen ergibt.

Die Umwandlung von λ-Ausdrücken in Superkombinatoren ist möglich. Es handelt sich im Grunde genommen um eine Normalisierung von Programmen. Diese Technik, die hier nicht eingeführt wird, heißt „λ-Lifting“.

4.7.2 Strukturierte Typen

Konstruktoren – wie etwa `cons` – sowie besondere Werte und Typen – wie etwa die leere Liste `nil` und der Typ `Liste` – müssen in die Zielsprache eingeführt werden.

4.8 Speicherbereinigung

Es kann vorkommen, dass Knoten auf der Halde nicht mehr vom Keller oder von DEF-Zellen aus erreichbar sind. In diesem Fall sind sie nutzlos geworden. Im vorangehenden Beispiel der Übersetzung und Ausführung eines F-Programms sind die Inhalte von solchen nutzlos gewordene Haldenzellen nicht angegeben.

Der Speicherplatz in der Halde, der von nicht mehr verwendbaren oder erreichbaren Daten in Anspruch genommen wird, kann prinzipiell für zukünftig aufzubauende Daten wiedergewonnen werden. Methoden die dies ermöglichen werden Speicherbereiningungsmethoden genannt.

Bei der Speicherbereinigung mussen unter anderem folgende Fragen untersucht werden:

1. Vergleich der Kosten und des Gewinns der Speicherbereinigung. Die Speicherbereinigung nimmt Zeit und Ressourcen in Anspruch, die sich lohnen müssen.

2. Kompaktierung. Wenn Speicherplatz für spätere Verwendungen wiedergewonnen wird, tendiert der freie Raum der Halde dazu, in immer kleineren Blöcke zerstückelt zu sein, die zum Anlegen von großen Graphen zu klein sind. In dem Fall ist eine Kompaktierung der Halde nötig.

3. Funktioniert die Methode zur Speicherbereinigung auch mit zyklischen Datenstrukturen?

Die Speicherbereinigung kann auch bei imperativen Programmierspachen notwendig oder wünschenswert sein. Da imperative Programme den Kontrollfluss explizit angeben, bestimmen sie auch die Lebensdauer von dynamischen Datenstrukturen. Das heißt, dass es dem Programmierer im Prinzip möglich ist, den Speicher selber von unnötig gewordenen oder nicht mehr zugreifbaren Datenstrukturen zu bereinigen. Mit einer funktionalen – oder logischen – Programmiersprache ist dies aber nicht mehr möglich, so dass die Speicherbereinigung nur durch das Laufzeitsystem durchgeführt werden kann.

4.9 Zur Übersetzung gemäß der Auswertung in applikativer Reihenfolge

Die Übersetzung gemäß der Auswertung in applikativer Reihenfolge erscheint einfacher als die Übersetzung gemäß der verzögerten Auswertung, weil alle Argumente einer Funktion komplett ausgewertet übergeben werden.

Jeder Funktionsdefinition wird ähnlich wie eine Prozedur einer imperativen Programmiersprache übersetzt. Ein gewaltiger Unterschied liegt aber darin, dass funktionale Programmiersprachen die Auswertung eines Teils der Argumente ermöglichen. Betrachten wir z. B. die folgende Funktionsdefinition:

```
f x y = + x y
```

Die Auswertung von `f 1` liefert eine Funktion, die angewandt auf ein `y` dieses um `1` erhöht. Für jeden Wert von `x` ergibt sich eine unterschiedliche Funktion auf `y`. Eine Funktionsdefinition kann also nicht gleich wie eine Prozedur einer imperativen Programmiersprache übersetzt werden.

Die Lösung liegt in einer Indirektion: der eigentliche Wert der gebundenen Variablen – im obigen Beispiel `x` – werden in einer „Tabelle“ gespeichert. Anstatt eines Wertes wird die Adresse in der Tabelle auf dem Keller gelegt. Die Tabelle, die in der Regel als Liste implementiert ist, heißt „Umgebung“.

Da bei jeder Funktionsanwendung eine Umgebung angelegt wird, werden Umgebungen in eine Liste verkettet, die wie ein Keller bei der Berechnung jeder Funktionsanwendung wächst und bei der Beendigung der Auswertung schrumpft.

Der im diesem Kapitel dargestellte Übersetzungsansatz ist insofern einfacher, dass keine solche Umgebung verwaltet werden müssen. Der Preis dafür ist aber die Freilegung des Rückgrates und das λ-Lifting.

5 Übersetzung logischer Programmiersprachen

5.1 Konzepte logischer Programmiersprachen

Die logische Programmierung beruht auf der Beobachtung, dass Beweise aus einem Fragment der Prädikatenlogik erster Stufe als Programmabläufe interpretiert werden können.

Logische Programmiersprachen unterscheiden sich von imperativen Programmiersprachen ähnlich wie sich funktionale von imperativen Programmiersprachen unterscheiden: Sie kennen keine Wertzuweisung, und der Kontrollfluss wird implizit bestimmt.

Auslegung der Variablen. Variablen logischer Programmiersprachen sind Namen für Ausdrücke. Variablen sind keine Namen für Speicherzellen mit veränderlichem Inhalt, wie es in imperativen Programmiersprachen der Fall wäre. Deshalb gibt es in logischen Programmiersprachen keine Wertzuweisung. Variablen sind aber auch keine Namen für Werte, wie es in funktionalen Programmiersprachen der Fall wäre, weil Ausdrücke in logischen Programmiersprachen normalerweise keine Werte haben.

Spezifikation des Kontrollflusses. Das Berechnungsmodell logischer Programmiersprachen basiert auf logischer Deduktion und nicht auf Aktionen. Deshalb gibt es in logischen Programmiersprachen keine Anweisungen, also auch keine Anweisungen zur Spezifikation des Kontrollflusses. Der Kontrollfluss wird in logischen Programmiersprachen wie in funktionalen Programmiersprachen implizit durch das Auswertungsprinzip bestimmt, das der Sprache zu Grunde liegt. Die Auswertungsprinzipien logischer Programmiersprachen sind aber andere als die funktionaler Programmiersprachen.

Logische Programmiersprachen unterscheiden sich von funktionalen Programmiersprachen vor allem in folgenden Punkten:

Werte von Ausdrücken. Die mathematische Logik grenzt zwei Arten von Ausdrücken voneinander ab: *Formeln* repräsentieren Aussagen, die wahr oder falsch sein können. *Terme* repräsentieren Objekte, über die wahre oder falsche Aussagen gemacht werden können, die aber nicht selbst wahr oder falsch sind. In logischen Programmiersprachen äußert sich diese Abgrenzung darin, welche Ausdrücke welche Werte haben können.

Terme sind Ausdrücke, die aus Termen zusammengesetzt sein können. Terme werden nicht ausgewertet, sie können als ihre eigenen Werte angesehen werden. *Atomare Formeln* oder kurz *Atome* sind andere Ausdrücke, die aus Termen zusammengesetzt sein können. Im Lauf der Programmausführung werden Wahrheitswerte für die atomaren Formeln ermittelt. Atomare Formeln können mit Hilfe von logischen Junktoren zu anderen Arten von Formeln zusammengesetzt werden, zum Beispiel Negation oder Konjunktion.

Relationale Ausdrücke. In der funktionalen Programmierung ist `fakultaet(3)` ein Ausdruck, der zum Wert `6` ausgewertet wird. In der logischen Programmierung wird dieser Zusammenhang durch eine atomare Formel wie etwa `fakultaet(3,6)` dargestellt, für die ein Wahrheitswert ermittelt wird. Die Ausdrücke der logischen Programmierung sind also nicht funktional, sondern relational.

Um den Wert eines Ausdrucks in einem anderen Ausdruck zu verwenden, benutzt man für funktionale Ausdrücke die Schachtelung, zum Beispiel `quadrat(fakultaet(3))`. Für relationale Ausdrücke benutzt man zu diesem Zweck die Konjunktion, zum Beispiel `fakultaet(3,6)` $\wedge$ `quadrat(6,36)`.

Für Terme, die man als „Ergebnis“ ansieht, benutzt man Variablen, die insbesondere zwischen Konjunktionsgliedern geteilt werden können: `fakultaet(3,`y`)` $\wedge$ `quadrat(`y`,`z`)`. Die Auswertung von Formeln mit Variablen ergibt auch die Variablenbindung, für die der Wahrheitswert ermittelt wurde, in diesem Fall etwa `wahr` für $y = 6$, $z = 36$.

Position von „Ergebnissen“ in Ausdrücken. In relationalen Ausdrücken benutzt man Variablen für Terme, die man als „Ergebnis“ ansieht. Das ist nicht etwa nur für den letzten Term einer atomaren Formel möglich, sondern an beliebiger Position. Die atomare Formel `fakultaet(3,`y`)` wird zum Wahrheitswert `wahr` für die Bindung $y = 6$ ausgewertet. Die atomare Formel `fakultaet(`x`,6)` wird zum Wahrheitswert `wahr` für die Bindung $x = 3$ ausgewertet. Im Gegensatz zu funktionalen Ausdrücken geben relationale Ausdrücke also keine „Richtung“ vor.

Mehrere Vorkommen einer Variablen sind auch möglich: `ungerade(x)` $\wedge$ `quadrat(`x`,`x`)` wird zum Wahrheitswert `wahr` für die Bindung $x = 1$ ausgewertet.

Anzahl von „Ergebnissen“ in Ausdrücken. Relationale Ausdrücke können manchmal auf verschiedene Weise und für unterschiedliche Bindungen zum Wahrheitswert `wahr` ausgewertet werden. Zum Beispiel wird `fakultaet(`x`,1)` zum Wahrheitswert `wahr` für die Bindung $x = 0$ und auch zum Wahrheitswert `wahr` für die Bindung $x = 1$ ausgewertet.

Manche Ausdrücke können sogar auf unendlich viele Weisen ausgewertet werden. Zum Beispiel wird `fakultaet(`x`,`y`)` zum Wahrheitswert `wahr` ausgewertet für die Bindung $x = 0$, $y = 1$ und für die Bindung $x = 1$, $y = 1$ und für die Bindung $x = 2$, $y = 2$ und so weiter.

Wenn es mehrere Bindungen für den selben Ausdruck gibt, kann das so organisiert werden, dass sie der Reihe nach jeweils auf Anforderung geliefert werden. Die systematische Suche nach Auswertungsalternativen kann durch ein *Rücksetzverfahren* (backtracking) geleistet werden. Beides ist in Prolog der Fall, gilt aber nicht als notwendiges Merkmal von Sprachen der Logikprogrammierung. Auch das von Prolog verwendete Deduktionsverfahren namens *Resolution* ist nicht zwingend für logische Programmiersprachen.

Die logische Programmierung (oder Logikprogrammierung) entstand Anfang der 1970-er Jahre. Den erste Entwurf der Programmiersprache Prolog (ein Kürzel von PROgrammieren in LOGik) konzipierten Alain Colmerauer und Philippe Roussel an der Universität Marseille. Bob Kowalski von der Universität Edinburgh und dem Imperial College in London gab dieser Programmiersprache ihre Formalisierung in der Logik. An der Universität Edinburgh wurde dann Prolog angewendet und weiterentwickelt. Aus dieser Arbeit entstand durch David H. Warren die abstrakte Maschine, auf der alle Prolog-Systeme heutzutage beruhen, und die WAM (*Warren Abstract Machine*) heißt. Die in diesem Kapitel eingeführte abstrakte Maschine ist eine wesentliche Vereinfachung und grundsätzliche Systematisierung der WAM.

Erst in der ersten Hälfte der 1980-er Jahre erhielt die Logikprogrammierung weltweit eine große Aufmerksamkeit: durch das vom japanischen Industrie- und Handelsministerium MITI geförderte Forschungsprojekt „Computer der fünften Generation“, durch das Tokioter Forschungsinstitut ICOT und durch das im Jahr 1984 gegründete Münchner Forschungszentrum der Europäischen Computer-Industrie ECRC, die bis zum Anfang der 1990-er Jahre Anwendungen, Implementierungen und Ergänzungen der logischen Programmierung untersuchten.

Im Rest dieses Abschnitts führen wir kurz in Begriffe der Logik und automatischen Deduktion ein, die zum Verständnis der logischen Programmierung notwendig sind.

5.1.1 Terme, Atome und Klauseln

Definition. Eine (formale) Sprache der Logik besteht aus folgenden, paarweise disjunkten Symbolmengen:

1. Eine unendliche Menge von *Variablen* $x, y, z, \ldots$
2. Für jedes $n \geq 0$ eine Menge von *Funktionssymbolen* der Stelligkeit n. Um auszudrücken, dass ein Funktionssymbol f die Stelligkeit n hat, wird die Notation f/n benutzt.
   0-stellige Funktionssymbole heißen auch *Konstanten*.
3. Für jedes $n \geq 0$ eine Menge von *Relationssymbolen* (auch *Prädikatssymbole* genannt) der Stelligkeit n. Um auszudrücken, dass ein Relationssymbol p die Stelligkeit n hat, wird die Notation p/n benutzt.
4. Logische Symbole: die Quantoren $\forall$ und $\exists$ und die Junktoren $\neg$, $\wedge$, $\vee$, $\Rightarrow$ und $\Leftrightarrow$. ∎

Bemerkung. Die Variablenmenge muss unendlich sein, damit die Größe der Beweise nicht beschränkt ist. ∎

Definition. Terme, Atomare Formeln, Literale und Klauseln werden wie folgt definiert:

$$\begin{array}{rcl} \textit{Term} & ::= & \textit{Variable} \mid \textit{Funktionssymbol} \;\; [\, \texttt{"("}\; \textit{Term}\; \{\texttt{","}\; \textit{Term}\}\; \texttt{")"} \,]. \\ \textit{Atom} & ::= & \textit{Prädikatssymbol} \;\; [\, \texttt{"("}\; \textit{Term}\; \{\texttt{","}\; \textit{Term}\}\; \texttt{")"} \,]. \\ \textit{Literal} & ::= & [\, \texttt{"}\neg\texttt{"} \,]\; \textit{Atom}. \\ \textit{Klausel} & ::= & \Box \mid \textit{Literal}\; \{\texttt{"}\vee\texttt{"}\; \textit{Literal}\}. \end{array}$$

wobei einem Funktions- bzw. Relationssymbol der Stelligkeit $n \geq 1$ eine geklammerte Sequenz von genau n Termen folgt. Hinter 0-stelligen Symbolen stehen keine Klammern.

Ein Literal, das mit dem Negationszeichen "$\neg$" beginnt, wird *negatives Literal* genannt, ein Literal ohne Negationszeichen wird *positives Literal* genannt.

Die leere Klausel wird $\Box$ notiert. Sie stellt eine Formel dar, die immer falsch ist.

Die Bezeichnung *Formel* ist ein Oberbegriff unter anderem für Atome, Literale und Klauseln, aber nicht für Terme. Die Bezeichnung *Ausdruck* ist ein Oberbegriff für Formeln und Terme.

Ein *Grundterm* ist ein Term, in dem keine Variablen vorkommen. Analog bedeutet *Grundatom*, *Grundliteral*, *Grundklausel* usw., dass keine Variablen darin vorkommen. ∎

Das Wort „Literal“ wird im Bereich der Programmiersprachen für textuelle Darstellungen von konstanten Werten verwendet, in der Logik dagegen für bestimmte Formeln, nämlich atomare Formeln oder negierte atomare Formeln.

Wenn $L_1, \ldots, L_m$ Literale sind, ist $L_1 \vee \ldots \vee L_m$ eine Klausel. Sie ist eine quantorfreie Kurzschreibweise der geschlossenen Formel $\forall x_1 \ldots \forall x_k (L_1 \vee \ldots \vee L_m)$, wobei $x_1, \ldots, x_k$ die Variablen sind, die in den Literalen $L_1, \ldots, L_m$ vorkommen.

Seien $A_1, \ldots, A_p, B_1, \ldots, B_n$ atomare Formeln und $x_1, \ldots, x_k$ die darin vorkommenden Variablen. Dann sind die beiden Formeln $\forall x_1 \ldots \forall x_k (\neg B_1 \vee \ldots \vee \neg B_n \;\; \vee \;\; A_1 \vee \ldots \vee A_p)$ und $\forall x_1 \ldots \forall x_k ([B_1 \wedge \ldots \wedge B_n] \;\Rightarrow\; [A_1 \vee \ldots \vee A_p])$ logisch äquivalent. Deshalb kann man eine Klausel statt als Disjunktion von Literalen auch als Implikation schreiben, deren „wenn“-Seite eine Konjunktion von Atomen und deren „dann“-Seite eine Disjunktion von Atomen ist. In der Logikprogrammierung wird diese Schreibweise von Klauseln in Verbindung mit der Rückwärtsimplikation bevorzugt: $A_1 \vee \ldots \vee A_p \;\Leftarrow\; B_1 \wedge \ldots \wedge B_n$.

Definition. Seien $A_1, \ldots, A_p, B_1, \ldots, B_n$ atomare Formeln, also $\neg B_1 \vee \ldots \vee \neg B_n \vee A_1 \vee \ldots \vee A_p$ eine Klausel mit n negativen und p positiven Literalen. Folgende Bezeichungen und Notationen sind in der Logikprogrammierung üblich.

für $p \geq 0$, $n \geq 0$	Klausel	$A_1 \vee \ldots \vee A_p \Leftarrow B_1 \wedge \ldots \wedge B_n$
	Klauselkopf, Kopf	$A_1 \vee \ldots \vee A_p$
	Klauselrumpf, Rumpf	$B_1 \wedge \ldots \wedge B_n$
für $p \leq 1$	Hornklausel	
für $p = 1$, $n > 0$	Programmklausel, Regel	$A_1 \Leftarrow B_1 \wedge \ldots \wedge B_n$
für $p = 1$, $n = 0$	Programmklausel, Fakt	$A_1 \Leftarrow$
für $p = 0$, $n > 0$	Ziel	$\Leftarrow B_1 \wedge \ldots \wedge B_n$
für $p = 0$, $n = 0$	leere Klausel	$\Leftarrow$

■

Eine Hornklausel ist also eine Klausel mit höchstens einem positiven Literal. Für den leeren Klauselkopf wird manchmal auch $\perp$ geschrieben, für den leeren Klauselrumpf $\top$, für die leere Klausel $\Box$ oder $\perp \Leftarrow \top$.

Logische Programmiersprachen verwenden formale Sprachen der Logik, jedoch mit einigen syntaktischen Abweichungen. In Prolog wird zum Beispiel :- für $\Leftarrow$ geschrieben, Komma für $\wedge$ und Semikolon für $\vee$.

5.1.2 Substitutionen und allgemeinste Unifikatoren

Definition. Sei $\mathcal{T}$ die Menge aller Terme. Eine Substitution ist eine Abbildung $\sigma : \mathcal{T} \to \mathcal{T}$, so dass:

1. für jede Konstante c ist $\sigma(c) = c$.
2. für jedes n-stellige Funktionssymbol f mit $n \geq 1$ und für alle Terme $t_1, \ldots, t_n$ ist $\sigma\Big(f(t_1, \ldots, t_n)\Big) = f(\sigma\Big(t_1\Big), \ldots, \sigma\Big(t_n\Big))$.
3. Die Menge $\{x \mid x \text{ ist Variable und } \sigma(x) \neq x\}$ ist endlich. ■

Bemerkungen.

1. Für die Punkte 1. und 2. sagt man auch, dass eine Substitution ein „Homomorphismus auf $\mathcal{T}$“ ist. Für Punkt 3. sagt man, dass eine Substitution „fast überall identisch“ ist.

2. Da ein Term Klammern als syntaktische Bestandteile haben kann, ist es etwas verwirrend, für die Anwendung einer Substitution auf einen Term ebenfalls eine Schreibweise mit Klammern zu verwenden. In der obigen Definition wurden unterschiedliche Größen für die verschiedenen Arten von Klammern benutzt. Es ist auch üblich, die Anwendung einer Substitution σ auf einen Term t klammerfrei in Postfixnotation zu schreiben, also $t\sigma$ statt $\sigma(t)$.

3. Wegen 1. und 2. bildet eine Substitution alle Bestandteile eines Terms, die keine Variablen sind, auf sich selbst ab. Wegen 3. gibt es zu jeder Substitution σ endlich viele Variablen $x_1, \ldots, x_k$, die nicht auf sich selbst abgebildet werden, alle anderen Variablen werden auf sich selbst abgebildet. Eine Substitution σ kann deshalb dargestellt werden durch eine endliche Menge $\{t_1/x_1, \ldots, t_k/x_k\}$ mit $\sigma(x_i) = t_i$.

   Ein Paar t_i/x_i wird ausgesprochen „t_i anstelle von x_i“.

4. Die Notation t_i/x_i kommt von der englischen Formulierung „to substitute t_i for x_i". Die Übersetzung in andere Sprachen ergibt oft eine Formulierung wie „x_i ersetzen durch t_i". Deshalb wird insbesondere von nicht englischsprachigen Autoren auch die Notation x_i/t_i verwendet, zumal diese der Sichtweise entspricht, dass x_i auf t_i abgebildet wird.

5. Die Notation a/b wird in zwei verschiedenen Kontexten verwendet, zwischen denen allerdings keine Verwechslungsgefahr besteht: zum einen für Substitutionen wie oben, zum anderen für die Stelligkeit von Funktions- oder Prädikatssymbolen. In diesem Kontext wird f/n ausgesprochen: „f mit Stelligkeit n". ∎

Die Anwendung einer Substitution auf einen Term ist definiert und ergibt einen Term. Die Definition wird um Fälle für Relationssymbole und logische Symbole erweitert, und zwar völlig analog zu Funktionssymbolen (Fall 2). Damit ist die Anwendung einer Substitution auf eine Formel auch definiert und ergibt eine Formel.

Beispiel. Sei K die Klausel $\texttt{sympathisch}(x) \Leftarrow \texttt{informatiker}(x) \land \texttt{mag}(x, \texttt{logik})$ und sei $\sigma = \{\texttt{anna}/x\}$ oder genauer, sei σ die durch diese Menge repräsentierte Substitution.

Dann ist $\sigma(K)$ die Klausel $\texttt{sympathisch}(\texttt{anna}) \Leftarrow \texttt{informatiker}(\texttt{anna}) \land \texttt{mag}(\texttt{anna}, \texttt{logik})$.

Die Klausel K repräsentiert die Aussage „alle, die Informatiker sind und die Logik mögen, sind sympathisch", die Klausel $\sigma(K)$ repräsentiert die Aussage „wenn Anna Informatikerin ist und die Logik mag, dann ist sie sympathisch". ∎

Manchmal kann man für zwei verschiedene Ausdrücke eine gut ausgewählte Substitution finden, deren Anwendung auf die beiden Ausdrücke das gleiche Ergebnis liefert.

	Term t_1	Term t_2	Substitution τ
	$g(x,y,c)$	$g(z,f(z),c)$	$\{a/x,\ f(a)/y,\ a/z,\ b/v\}$
$\tau(t_i)$	$g(a,f(a),c)$	$g(a,f(a),c)$	

Es fällt auf, dass die Substitution τ auch Variableneinsetzungen vornimmt, die nicht notwendig sind, um für die beiden Terme das gleiche Ergebnis zu erzielen. Die folgende Substitution erreicht das gleiche Ergebnis mit möglichst wenigen Variableneinsetzungen.

	Term t_1	Term t_2	Substitution σ
	$g(x,y,c)$	$g(z,f(z),c)$	$\{f(x)/y,\ x/z\}$
$\sigma(t_i)$	$g(x,f(x),c)$	$g(x,f(x),c)$	

Die Substitution σ ist „allgemeiner" als τ im folgenden Sinn: man kann die Wirkung von τ auf einen beliebigen Term t auch mit Hilfe von σ erreichen, indem man zuerst σ auf t anwendet und danach noch die Substitution $\vartheta = \{a/x,\ b/v\}$. Für jeden Term t gilt $\tau(t) = \vartheta(\sigma(t))$.

Die folgende Definition formalisiert diese Intuition.

Definition. Seien A_1 und A_2 zwei Ausdrücke (also Terme, Atome, Literale oder andere Formeln). Eine Substitution σ heißt *Unifikator* von A_1 und A_2, wenn $\sigma(A_1) = \sigma(A_2)$ ist. Zwei Ausdrücke heißen *unifizierbar*, wenn es einen Unifikator für sie gibt.

Ein Unifikator σ von A_1 und A_2 heißt *allgemeinster Unifikator* von A_1 und A_2, wenn es für jeden Unifikator τ von A_1 und A_2 eine Substitution ϑ gibt, so dass $\tau = \vartheta \circ \sigma$ ist. ∎

Beispiele. Allgemeinste Unifikatoren von $g(x, y, c)$ und $g(z, f(z), c)$ sind:
$\{f(x)/y,\ x/z\}$
$\{z/x,\ f(z)/y\}$.
Nicht-allgemeinste Unifikatoren von $g(x, y, c)$ und $g(z, f(z), c)$ sind unter anderem:
$\{f(x)/y,\ x/z,\ b/v\}$
$\{a/x,\ f(a)/y,\ a/z\}$
$\{f(v)/x,\ f(f(v))/y,\ f(v)/z\}$
$\{f(f(v))/x,\ f(f(f(v)))/y,\ f(f(v))/z\}$
...
Die Terme $g(x, y, c)$ und $g(z, f(z), f(c))$ sind nicht unifizierbar.
Die Terme $g(x, x, c)$ und $g(f(z), f(f(z)), c)$ sind nicht unifizierbar, insbesondere sind die folgenden Substitutionen keine Unifikatoren (anwenden!) von diesen beiden Termen:
$\{f(z)/x\}$
$\{f(f(z))/x\}$
$\{f(f(z))/x,\ f(z)/z\}$ ∎

Eine Substitution σ heißt *idempotent*, wenn $\sigma \circ \sigma = \sigma$ ist, das heißt, wenn ihre mehrmalige Anwendung genau so wirkt wie ihre einmalige Anwendung. In der Darstellung als Menge von Paaren äußert sich die Idempotenz darin, dass keine Variable, die in einem Paar rechts vorkommt, in einem Paar auch links vorkommt. Von den obigen Beispielen ist die letzte Substitution nicht idempotent, alle anderen sind idempotent.

Eigenschaften.

1. Sind zwei Ausdrücke unifizierbar, dann besitzen sie einen allgemeinsten Unifikator.
2. Verschiedene allgemeinste Unifikatoren von zwei Ausdrücken unterscheiden sich nur durch Umbenennung der Variablen.
3. Alle allgemeinsten Unifikatoren von zwei Ausdrücken sind idempotent. ∎

Die letzte Eigenschaft spielt eine Rolle für Algorithmen zur Berechnung von allgemeinsten Unifikatoren. Wenn dabei eine Substitution entsteht, die eine Variable x auf einen Term abbildet, in dem x vorkommt, dann kann die berechnete Substitution kein allgemeinster Unifikator sein. Der Test, ob ein solcher Fall vorliegt, wird meistens *occur check* genannt.

Definition Eine Substitution ν heißt *Variablenumbenennung*, wenn gilt:

1. Für jede Variable x ist $\nu(x)$ eine Variable.
2. Für alle Variablen $x_1 \neq x_2$ mit $\nu(x_1) \neq x_1$ und $\nu(x_2) \neq x_2$ gilt $\nu(x_1) \neq \nu(x_2)$. ∎

5.1.3 Resolution

Die Resolution ist eine Inferenzmethode, die in vielen Deduktionssystemen eingesetzt wird. Sie wurde Anfang der 1960-er Jahre von John Alan Robinson eingeführt. Die prozedurale Semantik von Sprachen der Logikprogrammierung bezieht sich auf eine Vereinfachung dieser Inferenzmethode, die in diesem Abschnitt eingeführt wird.

Ausgangspunkt ist eine Menge von Programmklauseln, also Fakten und Regeln, sowie ein Ziel. Ein Resolutionsschritt besteht darin, aus einem Ziel zusammen mit jeweils einer Programmklausel ein neues Ziel herzuleiten, eine sogenannte Resolvente.

Für das selbe Ziel kann es mehrere Resolventen geben, die mit verschiedenen Programmklauseln jeweils in einem Resolutionsschritt herleitbar sind. Auf jede dieser Resolventen können wieder Resolutionsschritte anwendbar sein, so dass insgesamt ein Baum von Zielen entsteht, dessen Wurzel das ursprüngliche Ziel ist. Jeder Ast in diesem Baum, der mit der leeren Klausel endet, ist ein Beweis für das Ziel an der Wurzel des Baums.

Resolutionsschritt, Grundfall. Seien $A_1, \ldots, A_n, B_1, \ldots, B_m$ Grundatome.

Ziel		$\Leftarrow A_1 \wedge \ldots \wedge A_n$
Fakt	A_1	$\Leftarrow$
Resolvente		$\Leftarrow A_2 \wedge \ldots \wedge A_n$

oder

Ziel		$\Leftarrow A_1 \wedge \ldots \wedge A_n$
Regel	A_1	$\Leftarrow B_1 \wedge \ldots \wedge B_m$
Resolvente		$\Leftarrow B_1 \wedge \ldots \wedge B_m \wedge A_2 \wedge \ldots \wedge A_n$

Gegeben seien jeweils die beiden Klauseln oberhalb des Strichs, dann darf die Klausel unterhalb des Strichs, die sogenannte Resolvente, hergeleitet werden. Man sieht, dass die Resolvente wegen der zweiten gegebenen Klausel jeweils hinreichend für das Ziel ist.

Resolutionsschritt, allgemeiner Fall. Seien $A_1, \ldots, A_n, B_1, \ldots, B_m, C$ beliebige Atome.

Ziel	$\Leftarrow A_1 \wedge \ldots \wedge A_n$
Fakt	$C \Leftarrow$
Variablenumbenennung ν	so dass keine Variable aus dem Ziel in dem umbenannten Fakt $\nu(C)$ vorkommt
allgemeinster Unifikator σ	von $\nu(C)$ und A_1
Resolvente	$\Leftarrow \sigma(A_2) \wedge \ldots \wedge \sigma(A_n)$

oder

Ziel	$\Leftarrow A_1 \wedge \ldots \wedge A_n$
Regel	$C \Leftarrow B_1 \wedge \ldots \wedge B_m$
Variablenumbenennung ν	so dass keine Variable aus dem Ziel in der umbenannten Regel $\nu(C \Leftarrow B_1 \wedge \ldots \wedge B_m)$ vorkommt
allgemeinster Unifikator σ	von $\nu(C)$ und A_1
Resolvente	$\Leftarrow \sigma(\nu(B_1)) \wedge \ldots \wedge \sigma(\nu(B_m)) \wedge \sigma(A_2) \wedge \ldots \wedge \sigma(A_n)$

Beispiel.

Ziel		$\Leftarrow \texttt{sympathisch}(y)$
Regel	$\texttt{sympathisch}(x)$	$\Leftarrow \texttt{informatiker}(x) \wedge \texttt{mag}(x, y)$
Variablenumbenennung ν	$\{u/x,\ v/y\}$	
allgemeinster Unifikator σ	$\{y/u\}$	
Resolvente		$\Leftarrow \texttt{informatiker}(y) \wedge \texttt{mag}(y, v)$

Die Variablenumbenennung ν ist unabdingbar. Um sich davon zu überzeugen, kann das betrachtete Beispiel mit zwei zusätzlichen Fakten fortgesetzt werden:

$$\begin{array}{ll} \texttt{informatiker}(\texttt{anna}) & \Leftarrow \\ \texttt{mag}(\texttt{anna}, \texttt{logik}) & \Leftarrow \end{array}$$

Aus der obigen Resolvente kann mit zwei weiteren Resolutionsschritten die leere Klausel hergeleitet werden, womit das ursprüngliche Ziel $\Leftarrow \texttt{sympathisch}(y)$ für $y = \texttt{anna}$ bewiesen ist.

Ohne die Variablenumbenennung im ersten Schritt wäre die leere Klausel aber nicht herleitbar, weil die Resolvente dann $\Leftarrow \texttt{informatiker}(y) \wedge \texttt{mag}(y, y)$ gewesen wäre, die mit dem nächsten Resolutionsschritt auf das Ziel $\Leftarrow \texttt{mag}(\texttt{anna}, \texttt{anna})$ zurückgeführt wird, auf das kein weiterer Resolutionsschritt mit den gegebenen Klauseln anwendbar ist.

Die Anwendung einer Variablenumbenennung auf die Programmklausel in einem Resolutionsschritt nennt man *Variablenstandardisierung* (standardisation apart) oder *Rektifikation*.

Die hier eingeführte Variante der Resolution ist in zweierlei Hinsicht ein Sonderfall. Zum einen wurden Resolutionsschritte nur für Hornklauseln definiert. Für Klauseln, die keine Hornklauseln sind, müssten neben Resolutionsschritten noch sogenannte Faktorisierungsschritte hinzukommen (siehe Vorlesung Deduktionssysteme). Zum anderen wurden Resolutionsschritte so definiert, dass immer das erste Atom des Ziels an dem Schritt beteiligt ist. In der (Theorie der) Logikprogrammierung werden Varianten der Resolution betrachtet, bei denen das beteiligte Atom des Ziels durch eine sogenannte Auswahlfunktion bestimmt wird.

5.1.4 Zur Semantik von logischen Programmiersprachen

Die Semantik einer Programmiersprache wird für gewöhnlich in zwei verschiedenen und komplementären Weisen definiert. Die sogenannte prozedurale Semantik legt fest, wie Programme ausgeführt und dabei Ergebnisse „konstruiert“ werden. Sie bildet die Grundlage für Implementierungen der Sprache. Die sogenannte deklarative Semantik spezifiziert, was Ergebnisse eines Programms sind, ohne dabei Bezug auf ein Modell der Ausführung zu nehmen. Sie dient dem einfacheren Verständnis von Programmen und der mathematischen Analyse.

Für imperative Programmiersprachen steht die prozedurale Semantik im Vordergrund, und deklarative Semantiken sind meist nur mit einigem Aufwand definierbar.

Für die Logikprogrammierung ist die prozedurale Semantik durch die Resolutionsmethode gegeben. Eine deklarative Semantik namens „Semantik der minimalen Modelle“ oder „Semantik der kleinsten Herbrand-Modelle“ ergibt sich unmittelbar aus der deklarativen Semantik der mathematischen Logik, der sogenannten Modelltheorie. Darüber hinaus gibt es eine auf Fixpunkten beruhende konstruktive Charakterisierung der minimalen Modelle. Diese Charakterisierung liefert eine dritte Semantik, die sogenannte „Fixpunktsemantik“. Sie ist auch konstruktiv, aber wesentlich einfacher als die prozedurale Semantik, die den Implementierungen zu Grunde liegt.

Eine der schönen Eigenschaften der logischen Programmierung ist, dass alle drei Semantiken äquivalent sind.

5.2 Die logische Programmiersprache L

Die Programmiersprache L entspricht „reinem Prolog“ (pure Prolog). In L gibt es keine Zahlen, d. h., das Prolog-Konstrukt `is` ist in L nicht vorhanden.

5.2.1 Syntax von L

Programm	::=	{ *Programmklausel* } *Ziel* .
Programmklausel	::=	*NichtVariableLTerm* ("." | *Ziel*) .
Ziel	::=	":-" *Literal* { "," *Literal* } "." .
Literal	::=	["not"] *LTerm* .
NichtVariableLTerm	::=	*Name* ["(" *LTerm* {"," *LTerm*} ")"] .
LTerm	::=	*Variable* | *NichtVariableLTerm* .

Zusätzlich gilt:

1. Ein *Name* ist ein Bezeichner ungleich `not`, der mit einem Kleinbuchstaben beginnt (z. B. `x`, `x1`, `studienfach`, `sTUDIENFACH`).
2. Eine *Variable* ist ein Bezeichner, der mit einem Großbuchstaben beginnt (z. B. `X`, `X1`, `Studienfach`, `STUDIENFACH`).

In L wird `:-` für $\Leftarrow$ und Komma für $\wedge$ und `not` für $\neg$ geschrieben. Die Grammatik spezifiziert, dass `not` stärker bindet als `,` und `,` stärker als `:-` und dass Ausdrücke der Form A`:-`B`:-`C oder `not not` A gar nicht gebildet werden können. Assoziativitätsregeln müssten noch in die Grammatik eingebaut werden. Aus Gründen der Verständlichkeit sind statt dessen alle Präzedenz- und Assoziativitätsregeln für die Operatoren in der folgenden Tabelle zusammengefasst, wobei wie in Prolog ein Operator umso stärker bindet, je niedriger seine Präzedenz ist. (Vorsicht: bei F ist es umgekehrt!)

Präzedenz	Assoziativität	Operator
3	undefiniert	`:-`
2	rechtsassoziativ	`,`
1	undefiniert	`not`

Bezeichnungen wie „Regel“, „Fakt“, „Kopf“, „Rumpf“ werden für L übernommen. Man beachte folgende Besonderheiten der Syntax:

1. Ein L-Programm ohne Programmklausel ist erlaubt, aber es enthält immer genau ein Ziel als letzte Klausel. Prolog-Programme enthalten dagegen normalerweise keine Ziele, weil Prolog-Implementierungen Treiberschleifen zur Eingabe von Zielen anbieten.
2. Jede Klausel wird mit einem Punkt abgeschlossen.
3. Fakten werden ohne `:-` geschrieben.
4. `not` kann nur im Rumpf einer Klausel vorkommen und kann nicht geschachtelt werden.
5. L unterscheidet nicht, ob ein *Name* ein Funktionssymbol oder ein Relationssymbol bezeichnet. Folglich kann L zwischen Termen und Atomen lediglich aufgrund ihrer Position unterscheiden. In der Programmklausel

   ```
   p(X, f(Y))   :-   q(X, g(a,Y)),   f(X).
   ```

   wären im Sinn der mathematischen Logik `f(Y)` und `g(a,Y)` Terme, aber `p(X, f(Y))` und `q(X, g(a,Y))` und `f(X)` Atome, also `f` unzulässigerweise sowohl ein Funktionssymbol als auch ein Relationssymbol. Für L ist dagegen jeder dieser Ausdrücke ein *LTerm*, und die doppeldeutige Verwendung von `f` ist zulässig.
6. Ebenfalls im Gegensatz zur Logik erlaubt L, dass ein Symbol mit mehreren Stelligkeiten benutzt wird, wie in `:- p(a), p(X,b).`

Die fehlende Unterscheidung zwischen Termen und Atomen hat zur Folge, dass auch eine Variable an der Stelle eines Atoms stehen kann. L erlaubt wie Prolog, dass ein Literal die Gestalt V oder `not` V mit einer Variablen V hat. In diesem Fall ist das L-Programm nur dann korrekt, wenn zur Zeit der Auswertung des Literals die Variable V an einen nichtvariablen LTerm gebunden ist, der dann als das eigentliche Atom behandelt wird, das ausgewertet wird.

Bei dieser Bedingung handelt es sich um eine nicht immer syntaktisch überprüfbare Eigenschaft, die durch hinreichende Bedingungen sichergestellt werden kann. Im weiteren werden wir keine Methoden zur Überprüfung dieser Bedingung einführen, jedoch annehmen, dass diese Bedingung immer erfüllt ist.

Ebenfalls wie Prolog verbietet L, dass der Kopf einer Programmklausel ein Variable ist. Dies könnte aber insofern erlaubt werden, als die abstrakte Maschine ML Variablen als Köpfe von Programmklauseln bearbeiten kann.

Nichtlogische Programmkonstrukte wie in Prolog das cut (der `!`-Operator) zur Steuerung des Rücksetzens, `assert` und `retract` zur Änderung des Programms zur Laufzeit oder Ein- und Ausgabeoperationen sind in L nicht vorhanden (L entspricht „reinem Prolog"). Die Negation mit `not` weicht wie in Prolog von der Negation der klassischen Logik ab.

5.2.2 Geltungsbereiche von Variablen in L

In L sind Variablen lokal zu den Programmklauseln oder Zielen, in denen sie vorkommen. Folglich können zwei Programmklauseln oder Ziele, die bis auf eine Variablenumbenennung gleich sind, gegeneinander ausgetauscht werden, ohne die prozedurale Semantik des Programms zu ändern. Dies ist damit konsistent, dass die Resolution eine Variablenstandardisierung eingebaut hat.

Mit der deklarativen Semantik ist die Begründung für die Austauschbarkeit, dass zwei Formeln $\forall x_1 \ldots \forall x_k\, F(x_1, \ldots, x_k)$ und $\forall x'_1 \ldots \forall x'_k\, F(x'_1, \ldots, x'_k)$ äquivalent sind, wenn $F(\ldots)$ eine quantorfreie Formel mit den angegebenen Variablen ist und für $x_i \neq x_j$ auch $x'_i \neq x'_j$ ist.

5.2.3 Ein Parser für L

Die Grammatik für L erfüllt die LL(1)-Bedingung. Wenn noch die Rechtsassoziativität des Operators `,` eingebaut wird ohne die LL(1)-Bedingung zu verletzen, kann ein LL(1)-Parser für L gemäß Abschnitt 2.2.3 generiert werden.

5.2.4 Prozedurale Semantik von negationsfreien L-Programmen

Zunächst wird die prozedurale Semantik von negationsfreien L-Programmen durch einen Algorithmus in einem höheren Pseudocode spezifiziert. Diese Spezifikation wird als Vorgabe zur Entwicklung einer abstrakten Maschine dienen. Die abstrakte Maschine wird danach so erweitert, dass sie auch die Negation behandeln kann.

Die prozedurale Semantik von L wird mit einem (einfachen, rekursiven) Interpretierer definiert, der als Spezifikation der abstrakten Maschine dient.

Ein Interpretierer ohne Rücksetzen für negationsfreie L-Programme.

Eingabe: Ein negationsfreies L-Programm mit `Programmklauseln` und `Ziel`.

Ausgabe: Eine Substitution oder `falsch`.

Die folgenden Funktionen werden verwendet:

`rumpf(X)`: liefert den (eventuell leeren) Rumpf einer Klausel oder eines Ziels `X`.

`pop(S)`: entfernt das erste Element eines Kellers oder einer Sequenz `S` und liefert es.

`append(S1, S2)`: setzt die Sequenz `S1` mit der Sequenz `S2` fort und liefert das Ergebnis.

```
Algorithmus evaluate(Programmklauseln, Ziel);

Variablen R: Rumpf;
          A: Atom;
          sigma, theta, ny: Substitution;
          gescheitert: boolean;

R := rumpf(Ziel);
theta := Identitäts-Substitution;
gescheitert := false;
while not gescheitert und nichtleer(R) do
A := pop(R);
if es können gewählt bzw. berechnet werden:
   - eine Programmklausel K mit Kopf C und (eventuell leerem) Rumpf B,     (*)
   - eine Variablenumbenennung ny von K, so dass ny(K) keine gemeinsame
     Variable mit A und R hat, und
   - einen allgemeinsten Unifikator sigma von A und ny(C)
   then R := append( sigma(ny(B)), sigma(R) ); (* ergibt einen neuen *)
        theta := sigma o theta                 (*     Zielrumpf      *)
   else gescheitert := true
end-if;
if gescheitert
   then return false
   else schränke theta auf die im Ziel vorkommenden Variablen ein;
        return theta
end-while
```

Es kann vorkommen, dass mehrere Programmklauseln an der Stelle `(*)` ausgewählt werden können. Der obige Interpretierer berücksichtigt nur eine Wahl. Das in L angewandte Prinzip zur Berücksichtigung aller Alternativen heißt „Rücksetzen“ (backtracking). Dabei werden die Programmklauseln in der Reihenfolge ausgewählt, in der sie im Programm stehen.

Beispiele.

1.
```
p :- s.
p :- r, q.
q.
r.
:-  p, r.
```

Das erste Atom `p` des Ziels kann mit der ersten und mit der zweiten Programmklausel resolviert werden.

Die Wahl der ersten Programmklausel ergibt die Resolvente `:- s, r.`, mit der die Fortsetzung scheitert, weil keine Programmklausel den Kopf `s` hat.

Rücksetzen führt dazu, dass danach für `p` die zweite Programmklausel gewählt wird und die Resolvente `:- r, q, r.` ergibt. Dieses Ziel führt in drei Resolutionsschritten mit der dritten und vierten Programmklausel zum Erfolg. Das Ergebnis ist die Identitätssubstitution, die durch die leere Variablenbindung repräsentiert wird.

2.
```
p.
p.
q.
:- p, q.
```
   Durch Auswahl der ersten Programmklausel entsteht aus dem Ziel die Resolvente `:- q.`, die mit der dritten Programmklausel zum Erfolg führt.

   Beim Rücksetzen wird für das Atom `p` des Ziels danach die zweite Programmklausel gewählt, was auf die gleiche Weise zum Erfolg führt.

   Es gibt also zwei Erfolge, beide mit der leeren Variablenbindung.

3.
```
p(a).
p(b).
q(b).
:- p(X), q(X).
```
   Durch Auswahl der ersten Programmklausel entsteht die Resolvente `:- q(a).`, mit der die Fortsetzung scheitert, weil kein Klauselkopf mit diesem Atom unifizierbar ist.

   Beim Rücksetzen wird für das Atom `p(X)` des Ziels danach die zweite Programmklausel gewählt, was die Resolvente `:- q(b).` ergibt, die mit der dritten Programmklausel zum Erfolg führt mit der Variablenbindung `X = b`.

Die Beispiele zeigen, dass das Rücksetzen notwendig ist. Wurde eine unpassende Programmklausel ausgewählt, dann kann der Beweis nicht vollendet werden, was durch die Auswahl einer anderen Programmklausel in manchen Fälle behoben werden kann. ■

Die Auswertung eines Zieles kann drei Ausgänge haben:

1. Das Ziel wird (einmal oder mehrmals) bewiesen, oder
2. das Ziel wird nicht bewiesen und die Auswertung terminiert, oder
3. das Ziel wird nicht bewiesen und die Auswertung terminiert nicht.

Ein Interpretierer mit Rücksetzen für negationsfreie L-Programme. Zur Verwaltung der Alternativen bei der Auswahl einer Programmklausel wird mit einem Keller eine Tiefensuche implementiert.

> **Eingabe:** Ein negationsfreies L-Programm mit `Programmklauseln` und `Ziel`.
>
> **Ausgabe:** Eine Substitution oder `falsch`.

Zusätzlich zu den im vorangehenden Abschnitt eingeführten Funktionen werden die folgenden Funktionen und Prozeduren verwendet:

`top(S)`: liefert das erste Element eines Kellers oder einer Sequenz `S`.

`push(S, E)`: legt `E` auf den Keller `S`.

`leer(X)`: liefert `wahr`, falls der Keller oder die Sequenz `X` leer ist, andernfalls liefert `falsch`.

`neu(K)`: ersetzt die Variablen in `K` durch Variablen, die in keiner Datenstruktur vorkommen, und liefert das Ergebnis.

`klausel(n)`: liefert die `n`-te Programmklausel.

`kopf(K)`: liefert den Kopf einer Klausel `K`.

`unify(A, B)`: liefert einen allgemeinsten Unifikator von `A` und `B`, falls es einen gibt, andernfalls liefert `falsch`.

```
Algorithmus evaluate(Programmklauseln, Ziel);

Konstante max: Anzahl der Programmklauseln;
Variablen Z: Rumpf eines Ziels;
          A: Atom;
          Nr:  Position einer Programmklausel im Programm;
          sigma, theta: Substitution;

Keller := (rumpf(Ziel), 1, Identitäts-Substitution);
while not leer(Keller) do
   (Z, Nr, sigma) := top(Keller)
   if leer(Z)
      then schränke sigma auf die im Ziel vorkommenden Variablen ein;
           return sigma;
           repeat (Z, Nr, sigma) := pop(Keller)
              until Nr <= max or leer(Keller)
           end-repeat
      else A := pop(Z);
           theta := false;
           while Nr <= max and theta = false do
              K := neu(klausel(Nr));
              theta := unify(A, K);
              if not theta then Nr := Nr + 1 end-if
           end-while;
           if Nr > max
              then  repeat pop(Keller)
                       until Nr <= max or leer(Keller)
                    end-repeat
              else Nr := Nr + 1
                   push(Keller,
                        (theta([rumpf(K)) | Z]), 1, theta o sigma) )
           end-if
   end-if
```

Der Unifikationsalgorithmus. Die folgenden Funktionen werden verwendet:

`variable(Ausd)`: liefert „wahr“, falls `Ausd` eine Variable ist, andernfalls liefert „falsch“.

`symb(Ausd)`: liefert den Name des äußeren Symbols von `Ausd`.

`arity(Ausd)`: liefert die Stelligkeit vom äußeren Symbol `symb(Ausd)` von `Ausd`.

`arglist(Ausd)`: liefert die Argumentliste von `Ausd`. Beim Aufruf ist `Ausd` keine Variable.

```
Funktion unify(A1, A2);
         (* liefert einen allgemeinsten Unifikator der LTerme A1, A2,
            falls sie unifizierbar sind, andernfalls liefert false *)

Variablen theta: Substitution

theta := Identitäts-Substitution;
if Variable(A1)
   then theta := { A2/A1 }
   else if Variable(A2)
           then theta :=  { A1/A2 }
           else if symb(A1) = symb(A2) and arity(A1) = arity(A2)
                   then theta := unifysequence(arglist(A1),
                                               arglist(A2),
                                               Identitäts-Substitution)
                   else theta := false
                end-if
        end-if
end-if;
return theta



Funktion unifysequence(L1, L2, sigma);

Variablen T1, T2: LTerm;
          theta: Substitution;

if leer(L1) and leer(L2)
   then return sigma
   else T1 := pop(L1); T2 := pop(L2);
        theta := unify(T1, T2);
        if not theta = false
           then return unifysequence(theta(L1), theta(L2), theta o sigma)
           else return false
        end-if
end-if
```

Beispiele.

1. `unify( f(X, X), f(a, b) )`
2. `unify( f(g(X),X), f(Y,b) )`

Der hier beschriebene Unifikationsalgorithmus führt den sogenannten *occur check* nicht durch. Damit ist er formal betrachtet inkorrekt. Es ist üblich – wenn auch nicht immer der Fall –, dass Prolog-Implementierungen diesen Test nicht durchführen und die Verantwortung dem Programmierer überlassen, inkorrekte Berechnungen zu vermeiden.

5.3 Die abstrakte Maschine ML für L

Die abstrakte Maschine ML für L wird stufenweise eingeführt. Zunächst wird eine abstrakte Maschine für ein Fragment MiniL von L definiert. MiniL erlaubt nur nullstellige Prädikatssymbole. Ein Atom in MiniL hat also eine Gestalt wie etwa `a` oder `p`. Dann wird die abstrakte Maschine für MiniL auf das variablenfreie Fragment GroundL von L erweitert. Schließlich wird die Erweiterung auf L-Programme mit Variablen erläutert.

5.3.1 Prinzip der abstrakten Maschine für MiniL

Betrachten wird das folgende Programm:

```
  | 1    2   3
--+-------------
1 | a :- c,   d.
2 | b.
3 | c.
4 |   :- a,   c.
```

Zunächst wird der Rücksetzensregisters B (backtracking flag) initialisiert: `B := false`. Dann wird das erste Atom des (Rumpfes des) Zieles (`a`) auf den Keller gelegt:

`push a`
Dieser Befehl legt einen neuen „Auswahlpunkt“ auf dem Keller:

```
          +-----+
C -> s1 |  1  |   Erste Programmklausel
          +-----+
     s2 | nil |   Adresse des letzten Auswahlpunkts
          +-----+
     s3 | 4.3 |   Rückkehraddresse zur Fortsetzung des Beweises des
          +-----+   Zieles nach Beendigung des Beweises von a.
T -> s4 |  a  |
          +-----+
```

Es ergibt sich also das folgende (unvollständige) Muster für die Code-Erzeugung:

Ziel: `:- a, c` ⟶ Code: `push a   push  c`

Bevor das Zielatom `c` berücksichtigt wird, muss aber das Zielatom `a` bewiesen werden. Dazu werden die Programmklausel aufgerufen. Die Codererzeugung wird also wie folgt verfeinert:

Ziel: `:- a, c` ⟶ Code: `push a   call   push c   call`

Der Befehl `call` ruft die vom Register C gezeigte Klausel, in dem Fall die erste Programmklausel `a :- c, d.` Zunächst wird versucht, den Kopf `a` dieser Klausel mit dem Atom der oberen Kellerzelle zu unifizieren. Gelingt es nicht wird der Rücksetzensregister B auf `true` gesetzt. Andernfalls, wird der Rumpf (`:- a, c`) der Klausel zum neuen Ziel. Die (unvollständige) Code-Erzeugung aus einer Programmklausel ist also wie folgt :

Programmklausel: `a :- c, d` ⟶ Code: `unify a  push c  call  push d  call`

`unify a`
Dieser Befehl ist in dem Fall erfogreich. Da es mehrere Klauseln geben kann, die `a` als Kopf haben, hinterlässt der Befehl `call` das Atom `a` aus dem Ziel auf dem Keller.

`push c`

```
            +-----+
         s1 |  2  |
            +-----+
         s2 | nil |
            +-----+
         s3 | 4.3 |
            +-----+
         s4 |  a  |
            +-----+
    C -> s5 |  1  |   Erste Programmklausel zum Beweis von c
            +-----+
         s6 | s1  |   vorheriger Wert von C
            +-----+
         s7 | 1.3 |   Rückkehraddresse zur Fortsetzung des Beweises des
            +-----+   Zieles nach Beendigung des Beweises von c
         s8 |  c  |
            +-----+
```

`call`
Aufruf des vom C gezeigten Klausel, Erhöhung von stack(C) um 1:

```
            +-----+
         s1 |  2  |
            +-----+
         s2 | nil |
            +-----+
         s3 | 4.3 |
            +-----+
         s4 |  a  |
            +-----+
    C -> s5 |  2  |   Nächste Programmklausel zum Beweis von c
            +-----+
         s6 | s1  |
            +-----+
         s7 | 1.3 |
            +-----+
         s8 |  c  |
            +-----+
```

`unify a`
Da `a` und `c` nicht unifizierbar sind, wird der Rücksetzensregister B auf `true` gesetzt. Ein darauffolgender Befehl `backtrack?` erhöht `stack(C)` um 1 und verringert den Programmzähler P um 2, falls B = `true` ist, um so zum Befehl `call` zurückzukehren. Die erweiterte Übersetzung einesr Klausel ist also:

Klausel: `a :- c, d` $\longrightarrow$ Code: `unify a  backtrack?  push c  call  push d  call`

Der nächstauszuführende Befehl ist also wieder einmal `call`. C wird nochmals um 1 erhöht, weil der Kopf `b` der zweiten Programmklausel mit `c` nicht unifiziert:

```
          +-----+
       s1 |  2  |
          +-----+
       s2 | nil |
          +-----+
       s3 | 4.3 |
          +-----+
       s4 |  a  |
          +-----+
  C -> s5 |  3  |
          +-----+
       s6 | s1  |
          +-----+
       s7 | 1.3 |
          +-----+
       s8 |  c  |
          +-----+
```

Noch einmal wird `call` durchgeführt. Diesmal gelingt die Unifikation, so dass B den Wert `false` zugewiesen wird und der darauffolgende Befehl `backtrack?` keine Wirkung hat. Da die dritte Klausel ein Faktum ist, gibt es keinen Rumpf zu beweisen.

Um den Programmzahler P auf die Zielprogrammadresse, die der Überstzung des Atoms 1.3 entspricht, ist ein `return`-Befehl notwendig. Programmklauseln und Ziele werden also wie folgt übersetzt:

Klausel: `a :- c, d` ⟶
Code: `unify a  backtrack?  push c  call  push d  call  return`
Ziel: `:- a, c, d` ⟶
Code: `unify a  backtrack?  push c  call  push d  call  return`

Der Befehl `return` verwendet einen Register R, so dass stack(R+1) die Rückkehradresse ist. Da es im Allgemeinen andere Klauseln als die dritte Klausel geben kann, womit das Atom des oberen Auswahlpunkt bewiesen werden kann, kann dieser Auswahlpunkt nicht vom Keller beseitigt werden. Hierbei handelt es sich um eine wesentliche Abweichung von der Vorgehensweise, die wir aus den imperativen und funktionalen Programmiersprachen kennengelernt haben.

`return`

```
          +-----+
       s1 |  2  |
          +-----+
       s2 | nil |
          +-----+
       s3 | 4.3 |
          +-----+
       s4 |  a  |
          +-----+
  C -> s5 | nil |
          +-----+
       s6 | s1  |
          +-----+
       s7 | 1.3 |
          +-----+
       s8 |  c  |
          +-----+

   P := 1.3
```

Es hätte aber passieren können, dass das Atom des oberen Auswahlpunkt mit dem Kopf von keiner Klausel unifiziert. In dem Fall wird nach Durchlauf aller Programmklauseln dem Register C den Wert `nil` zugewisen. Für solchen Falle wird der Befehl `backtrack?` auch nach jedem `call`-Befehl aufgerufen. Programmklauseln und Ziele werden also wie folgt übersetzt:

Klausel: `a :- c, d` $\longrightarrow$
Code: `unify a  backtrack?  push c  call  backtrack?  push d  call  backtrack?  return`

Ziel: `:- a, c, d` $\longrightarrow$
Code: `unify a  backtrack?  push c  call  backtrack?  push d  call  backtrack?  return`

Eine Fortführung ist nicht nötig, um das Prinzip der abstrakten Maschine zu vermitteln.

5.3.2 Speicherbereiche, Register und Befehle der abstrakten Maschine ML

Die abstrakte Maschine besteht aus:

- 5 Speicherbereichen:
  - 1 programmspeicher (code)
  - 1 Umgebung (env)
  - 1 Keller (stack)
  - 1 Unifikationskeller (us)
  - 1 Rücksetzenskeller (trail)
- 12 Registern:
  - 1 Befehlregister: I (instruction)
  - 7 Adressregistern:
    - P (program counter): code-Adresse
    - T (top of the stack): stack-Adresse
    - C (last choice point): nil oder stack-Adresse
    - R (return register): nil oder stack-Adresse
    - E (local environement register):nil oder stack-Adresse
    - UP (unification pointer): stack-Adresse
    - UT (top of us): us-Adresse
    - TT (top of trail): trail-Adresse
  - 1 Flaggeregister: B (backtrack flag): Wahrheitswert
  - 3 Zählerregistern:
    - PC (push counter): natürliche Zahl
    - SC (skip counter): natürliche Zahl
    - AC (argument counter): nil oder natürliche Zahl
- einer Sprache mit 6 Befehle: `push`,`unify`, `call`, `return`, `backtrack?` und `prompt`.

Die Speicherbereiche. Im Programmspeicher code wird das Zielprogramm gespeichert. Die Umgebung env dient dazu, auf die Codeadresse der Überzungen der Klauseln zu verweisen. Die Inhalte von code und env werden zur Übersetzungszeit festgelegt und bleiben zur Laufzeit unverändert. Es wäre also möglich und sinnvoll, env vor dem Keller stack im gleichen Speicherbereich zu implementieren, oder sogar durch einen zweiten Lauf über das Programm bzw. Zielprogramm, das Zielprogramm so zu verändern, dass env nicht mehr benötigt wird.

Die Unifikations- und Rücksetzenskeller us und trail werden zur Behandlung der Variablenbindungen benötigt. us ist ein Hilfskeller, der im Falle von L-Programmen mit Variablen zur Unifikation benötigt wird. In trail werden die Variablenbindungen gespeichert, die beim Rücksetzen rückgängig gemacht werden mussen.

Im folgenden wird zunächst MiniL behandelt. Dazu wird lediglich verwendet:

- 3 Speicherbereiche:
  - 1 Programmspeicher `code`
  - 1 Datenspeicher `stack` für den Keller
  - 1 Umgebung `env`
- 6 Register:
  - 1 Instruktionsregister `I`
  - 1 Boole'sches Register `B` zum Anzeigen ob Backtracking notwendig ist
  - 4 Adressregister:

    `T` = Top des Kellers in `stack`
    `C` = Basis des obersten Choice Points in `stack`
    `R` = `stack`-Adresse der `code`-Adresse für den Rücksprung
    `P` = Programmzähler für den *nächsten* Befehl in `code`.

Die Umgebung `env` enthält Daten, mit denen folgende Hilfsfunktionen definiert sind:

$c\_first()$	Codeadresse des ersten Befehls der ersten Programmklausel
$c\_next(c_i)$	Codeadresse des ersten Befehls der $(i+1)$-ten Programmklausel wenn c_i die Codeadresse des ersten Befehls der i-ten Programmklausel ist `nil` wenn die i-te Programmklausel die letzte ist
$c\_goal()$	Codeadresse des ersten Befehls des Ziels
$c\_last()$	Codeadresse des letzten Befehls des Ziels (immer `prompt`)

Im Gegensatz zu den abstrakten Maschinen der früheren Kapitel wird bei jedem Befehl die Änderung des Programmzählers `P` explizit spezifiziert. Der Haupt-Instruktionszyklus der abstrakten Maschine MiniML lautet also:

```
C:=nil;          R:=nil;          T:=-1;          B:=false;
P:=c_goal();     I:=code[P];
while (P≠nil) { Befehl in I ausführen;  I:=code[P]; }
```

Die Befehle.

`push` A.

Legt für ein Atom A einen neuen Auswahlpunkt auf den Keller.

```
stack[T+1]:=c_first();
stack[T+2]:=C;                             (* voriger Auswahlpunkt *)
stack[T+3]:=P+3;                           (* Rückkehradresse      *)
stack[T+4]:=A;                             (* Atom *)
C:=T+1;   R:=C+1;   T:=T+4;   P:=P+1
```

Aufbau eines Auswahlpunkts:

```
       +-----------------------------+
C ->   | nächste noch nicht gewählte |
       |      Programmklausel        |
       |      (Code-Addresse)        |
       +-----------------------------+
R ->   |     letzter Wert von C      |
       |      (Stack-Adresse)        |
       +-----------------------------+
       |      Rückkehradresse        |
       |      (Code-Adresse)         |
       +-----------------------------+
       | Atom aus Rumpf des Ziels    |
       | oder einer Programmklausel  |
       +-----------------------------+
```

Die Rückkehradresse ist `P+3` und nicht `P+1`, weil auf den Befehl `push` immer die Befehle `call` und `backtrack?` folgen, die alle drei gemeinsam ein einziges Atom bearbeiten.

`unify` A.
Überprüft, ob das Atom im obersten Auswahlpunkt mit A unifizierbar ist.

```
B:=(stack[C+3] ≠ A);                    (* nicht unifizierbar *)
P:=P+1
```

`C+3` zeigt auf das Atom im obersten Auswahlpunkt. Hier könnte man `T` statt `C+3` verwenden, weil ein MiniL-Atom immer nur eine Kellerzelle belegt. Da aber L-Atome in der Regel mehrere – und nicht alle gleich viele – Kellerzellen belegen, wäre `T` für Erweiterungen der Maschine nicht mehr korrekt.

`call`.
Springt zum Beginn des Codes der nächsten Programmklausel, die für den obersten Auswahlpunkt noch nicht gewählt wurde, und aktualisiert den obersten Auswahlpunkt.

```
if (stack[C]==nil) {                    (* keine Alternative *)
  B:=true;   P:=P+1
} else {                                (* Aktualisierung    *)
  P:=stack[C];   stack[C]:=c_next(stack[C])
}
```

`return`.
Logisches (aber nicht physikalisches) Entkellern des obersten Auswahlpunkts.

```
P:=stack[R+1];                     (* Rückkehradresse      *)
if (stack[R]≠nil) {
  R:=stack[R]+1          (* R-Wert des vorigen Auswahlpunkts *)
}
```

Bei Beendigung eines Beweises kann der oberste Auswahlpunkt noch nicht physikalisch beseitigt werden, weil ein späteres Atom zum Rücksetzen zwingen kann, das den logisch entkellerten Auswahlpunkt wieder kellert. Das Register `C` zeigt auf den physikalisch obersten Auswahlpunkt, das Register `R` auf den logisch obersten Auswahlpunkt.

Vor der Ausführung eines `return`-Befehls gilt immer `R=C+1`, danach zeigt `R` auf die zweite Zelle des logisch obersten Auswahlpunkts, der unterhalb des physikalisch obersten liegt. Nach einer Sequenz von `return`-Befehlen kommt entweder die Maschine zum Halten, oder es wird ein `push`-Befehl ausgeführt, der die Gleichung `R=C+1` wieder herstellt.

`backtrack?`
Falls der vorherige Befehl (`unify` oder `call`) Rücksetzen verlangt hat, dieses durchführen. Falls der vorherige Befehl kein Rücksetzen verlangt hat, weiter zum nachfolgenden Befehl.

```
if (B) {
  while (stack[C]==nil ∧ stack[R]≠nil) {
    C:=stack[R];   R:=C+1;   T:=C+3   (* phys. entkellern    *)
  }
  if(stack[C]==nil) {                 (* dann stack[R]==nil  *)
    P:=c_last()                       (* prompt              *)
  } else {
    P:=stack[C];
    stack[C]:=c_next(stack[C]);
    B:=false                          (* Rücksetzen beendet  *)
  }
} else {
  P:=P+1                              (* zum nächsten Befehl *)
}
```

Wenn Rücksetzen verlangt wurde, wird so lange der physikalisch oberste Auswahlpunkt entkellert, wie er keine Alternative mehr hat. Falls dadurch der ganze Keller leer wurde, wird zum `prompt`-Befehl verzweigt wobei das Rücksetzregister `B` den Wert `true` behält, was zum Halten der Maschine führt. Falls der Keller nicht leer wurde, ist der dann oberste Auswahlpunkt derjenige, bei dem das Rücksetzen endet.

`prompt`
Abbruch melden und halten bzw. Erfolg melden und je nach Eingabe Rücksetzen auslösen oder halten.

```
if (B) {
  write('no (more) solutions');   P:=nil        (* Halt       *)
} else {
  write('yes. more?');
  if (read()==';') { B:=true;   P:=P-1 }        (* backtrack? *)
  else  P:=nil                                  (* Halt       *)
}
```

Der else-Teil von `prompt` könnte mit einem Aufruf `display` anfangen, der im Fall von MiniL den Beweisbaum, der ja noch im Keller repräsentiert ist, formatiert und ausgibt. Dazu wäre eine Änderung von `call` und `backtrack?` günstig, damit der C-Wert eines Auswahlpunkts nicht schon von `call`, sondern erst beim Rücksetzen mit `backtrack?` aktualisiert wird. Im Falle von L-Programmen mit Variablen sollte `display` auch die Bindungen der im Ziel vorkommenden Variablen ausgeben.

Code-Erzeugung für MiniL.

Üb(:- Sequenz .)	:=	Üb_{Body}(Sequenz) `prompt`
Üb(Atom :- Sequenz .)	:=	Üb_{Head}(Atom) Üb_{Body}(Sequenz) `return`

Dabei wird die Codeadresse der Übersetzung der Klausel in `env` angelegt.

$$\text{Üb(Atom .)} \quad := \quad \text{Üb}_{\text{Head}}\text{(Atom)}$$
$$\texttt{return}$$

Dabei wird die Codeadresse der Übersetzung der Programmklausel in `env` angelegt.

$$\text{Üb}_{\text{Head}}\text{(Atom)} \quad := \quad \texttt{unify}\ \text{Atom}$$
$$\texttt{backtrack?}$$

$$\text{Üb}_{\text{Body}}\text{([Atom | Sequenz])} \quad := \quad \texttt{push}\ \text{Atom}$$
$$\texttt{call}$$
$$\texttt{backtrack?}$$
$$\text{Üb}_{\text{Body}}\text{(Sequenz)}$$

$$\text{Üb}_{\text{Body}}\text{([])} \quad := \quad []$$

Bemerkungen.

1. Der Befehl `prompt` kommt nur ein Mal immer als letzter Befehl vor.
2. Im Schema Üb(:- Sequenz) ist Sequenz nie leer (vergleiche die Grammatik für L). Daraus folgt, dass vor dem einzigen Befehl `prompt` stets der Befehl `backtrack?` kommt.
3. Die Umgebung `env` kann auf verschiedene Weise realisiert werden. Die Sequenz der Codeadressen der Übersetzungen der Programmklauseln und des Ziels reicht aus, weil sie zur Implementierung der Funktion `c_next` wie eine Liste durchsucht werden kann. Im Grunde stellt `env` eine ähnliche Sprungleiste dar, wie sie in einer Übung zur abstrakten Maschine MI (Behandlung von Vorwärtssprüngen) eingeführt wurde. Anstelle dieser „Sprungleiste“ könnte auch eine Modifikation der Fixup-Technik oder ein zweiter Lauf verwendet werden, um die Werte der Hilfsfunktionen zu bestimmen. ■

Beispiel.

Quellprogramm:

```
p :- q.
q :- r.
r.
:- p, r.
```

Zielprogramm:

```
 c0: unify p      (* 1. Klausel *)
 c1: backtrack?
 c2: push q
 c3: call
 c4: backtrack?
 c5: return
 c6: unify q      (* 2. Klausel *)
 c7: backtrack?
 c8: push r
 c9: call
c10: backtrack?
c11: return
```

```
c12: unify r       (* 3. Klausel *)
c13: backtrack?
c14: return

c15: push p        (*    Ziel    *)
c16: call
c17: backtrack?
c18: push r
c19: call
c20: backtrack?
c21: prompt
```

Ausführung:

```
* Initialisierung:
  P := c15;
  T := s0;             (* leerer Keller *)
  C := nil;
  R := nil;
  B := false;

* I := push p        C -> s1: c0     (* 1. Klausel *)
  P := c16           R -> s2: nil
                          s3: c18
                     T -> s4: p

* I := call          C -> s1: c6     (* 2. Klausel *)
  P := c0            R -> s2: nil
                          s3: c18
                     T -> s4: p

* I := unify p       Keller unverändert
  B := false
  P := c1

* I := backtrack?    Keller unverändert
  P := c2

* I := push q              s1: c6    (* 2. Klausel *)
  P := c3                  s2: nil
                           s3: c18
                           s4: p
                      C -> s5: c0    (* 1. Klausel *)
                      R -> s6: s1
                           s7: c5
                      T -> s8: q

* I := call                s1: c6    (* 2. Klausel *)
  P := c0                  s2: nil
                           s3: c18
                           s4: p
                      C -> s5: c6    (* 2. Klausel *)
                      R -> s6: s1
                           s7: c5
                      T -> s8: q

* I := unify p        Keller unverändert
  B := true
  P := c1
```

```
* I := backtrack?          s1: c6    (* 2. Klausel *)
  P := c6                  s2: nil
  B := false               s3: c18
                           s4: p
                      C -> s5: c12   (* 3. Klausel *)
                      R -> s6: s1
                           s7: c5
                      T -> s8: q

* I := unify q        Keller unverändert
  B := false
  P := c7

* I := backtrack?     Keller unverändert
  P := c8

* I := push r              s1: c6    (* 2. Klausel *)
  P := c9                  s2: nil
                           s3: c18
                           s4: p
                           s5: c12   (* 3. Klausel *)
                           s6: s1
                           s7: c5
                           s8: q
                     C ->  s9: c0
                     R -> s10: s5
                          s11: c11
                     T -> s12: r

* I := call                s1: c6    (* 2. Klausel *)
  P := c0                  s2: nil
                           s3: c18
                           s4: p
                           s5: c12   (* 3. Klausel *)
                           s6: s1
                           s7: c5
                           s8: q
                     C ->  s9: c6    (* 2. Klausel *)
                     R -> s10: s5
                          s11: c11
                     T -> s12: r

* I := unify p       Keller unverändert
  B := true
  P := c1

* I := backtrack?          s1: c6    (* 2. Klausel *)
  P := c6                  s2: nil
  B := false               s3: c18
                           s4: p
                           s5: c12   (* 3. Klausel *)
                           s6: s1
                           s7: c5
                           s8: q
                     C ->  s9: c12   (* 3. Klausel *)
                     R -> s10: s5
                          s11: c11
                     T -> s12: r

* I := unify q       Keller unverändert
  B := true
  P := c7
```

```
* backtrack?                s1: c6    (* 2. Klausel *)
  P := c12                  s2: nil
  B := false                s3: c18
                            s4: p
                            s5: c12   (* 3. Klausel *)
                            s6: s1
                            s7: c5
                            s8: q
                      C ->  s9: nil   (* keine naechste Klausel *)
                      R -> s10: s5
                           s11: c11
                      T -> s12: r

* I := unify r        Keller unverändert
  B := false
  P := c13

* I := backtrack?     Keller unverändert
  P := c14

* I := return                s1: c6    (* 2. Klausel *)
  P := c11                  s2: nil
                            s3: c18
                            s4: p
                            s5: c12   (* 3. Klausel *)
                      R ->  s6: s1
                            s7: c5
                            s8: q
                      C ->  s9: nil   (* keine naechste Klausel *)
                            s10: s5
                           s11: c11
                      T -> s12: r

* I := return                s1: c6    (* 2. Klausel *)
  P := c5             R ->  s2: nil
                            s3: c18
                            s4: p
                            s5: c12   (* 3. Klausel *)
                            s6: s1
                            s7: c5
                            s8: q
                      C ->  s9: nil   (* keine naechste Klausel *)
                            s10: s5
                           s11: c11
                      T -> s12: r

* I := return                s1: c6    (* 2. Klausel *)
  P := c18           R ->  s2: nil
                            s3: c18
                            s4: p
                            s5: c12   (* 3. Klausel *)
                            s6: s1
                            s7: c5
                            s8: q
                      C ->  s9: nil   (* keine naechste Klausel *)
                            s10: s5
                           s11: c11
                      T -> s12: r
```

```
* I := push r                s1: c6    (* 2. Klausel *)
  P := c19                   s2: nil
                             s3: c18
                             s4: p
                             s5: c12   (* 3. Klausel *)
                             s6: s1
                             s7: c5
                             s8: q
                             s9: nil   (* keine naechste Klausel *)
                             s10: s5
                            s11: c11
                            s12: r
                       C -> s13: c0    (* 1. Klausel *)
                       R -> s14: s9
                            s15: c21
                       T -> s16: r

* I := call                   s1: c6    (* 2. Klausel *)
  P := c0                    s2: nil
                             s3: c18
                             s4: p
                             s5: c12   (* 3. Klausel *)
                             s6: s1
                             s7: c5
                             s8: q
                             s9: nil   (* keine naechste Klausel *)
                             s10: s5
                            s11: c11
                            s12: r
                       C -> s13: c6    (* 2. Klausel *)
                       R -> s14: s9
                            s15: c21
                       T -> s16: r

* I := unify p         Keller unverändert
  B := true
  P := c1

* I := backtrack?            s1: c6    (* 2. Klausel *)
  P := c6                    s2: nil
  B := false                 s3: c18
                             s4: p
                             s5: c12   (* 3. Klausel *)
                             s6: s1
                             s7: c5
                             s8: q
                             s9: nil   (* keine naechste Klausel *)
                             s10: s5
                            s11: c11
                            s12: r
                       C -> s13: c12   (* 3. Klausel *)
                       R -> s14: s9
                            s15: c21
                       T -> s16: r

* I := unify q         Keller unverändert
  B := true
  P := c7
```

```
* I := backtrack?           s1: c6    (* 2. Klausel *)
  P := c12                  s2: nil
  B := false                s3: c18
                            s4: p
                            s5: c12   (* 3. Klausel *)
                            s6: s1
                            s7: c5
                            s8: q
                            s9: nil   (* keine naechste Klausel *)
                            s10: s5
                           s11: c11
                           s12: r
                      C -> s13: nil   (* keine naechste Klausel *)
                      R -> s14: s9
                           s15: c21
                      T -> s16: r
* I := unify r        Keller unverändert
  B := false
  P := c13
* I := backtrack?     Keller unverändert
  P := c14
* I := return               s1: c6    (* 2. Klausel *)
  P := c21                  s2: nil
                            s3: c18
                            s4: p
                            s5: c12   (* 3. Klausel *)
                            s6: s1
                            s7: c5
                            s8: q
                            s9: nil   (* keine naechste Klausel *)
                      R ->  s10: s5
                           s11: c11
                           s12: r
                      C -> s13: nil   (* keine naechste Klausel *)
                           s14: s9
                           s15: c21
                      T -> s16: r
* I := prompt         Keller unverändert      Ausgabe: more?
  B := true                                    Eingabe:      ;
  P := c20
* I := backtrack?           s1: c6    (* 2. Klausel *)
  P := c12                  s2: nil
  B := false                s3: c18
                            s4: p
                       C -> s5: nil   (* keine naechste Klausel *)
                       R -> s6: s1
                            s7: c5
                       T -> s8: q
* I := unify r         Keller unverändert
  B := true
  P := c13
* I := backtrack?      C -> s1: c12   (* 3. Klausel *)
  P := c6              R -> s2: nil
  B := false                s3: c18
                       T -> s4: p
* I := unify q         Keller unverändert
  B := true
  P := c7
```

```
* I := backtrack?     C -> s1: nil   (* keine naechste Klausel *)
  P := c12            R -> s2: nil
  B := false               s3: c18
                      T -> s4: p
* I := unify r        Keller unverändert
  B := true
  P := c13
* I := backtrack?     C -> s1: nil   (* keine naechste Klausel *)
  P := c21            R -> s2: nil
                           s3: c18
                      T -> s4: p
* I := prompt         Keller unverändert
                                     Ausgabe: no (more) solutions
  stop
```

■

5.3.3 Implementierung der Negation durch Scheitern

Die Negation als Scheitern ist eine im tagtäglichen Leben verbreitete Form der Negation. Sie besagt, dass eine Aussage A als bewiesen betrachten werden kann, sobald A nicht beweisbar ist.

Einerseits ist diese Negationsart sehr anschaulich. Nicht anders interpretieren wir z. B. Fahrpläne (wenn keine Verkehrsverbindung gefunden werden kann, heißt es, dass es keine gibt) oder eine Studentendatenbank (die nichtimmatrikulierte Studenten werden nicht explizit erfasst).

Andererseits weicht die Negation als Scheitern sehr wesentlich von der Negation der (klassischen) mathematischen Logik (dass aus den Gruppenaxiome nicht logisch folgt, dass eine Gruppe kommutativ ist, heißt nicht, dass alle Gruppen nicht kommutativ sind).

Die Suche nach einer ausreichende, allen Fällen abdeckenden Formalisierung der Negations als Scheitern ist seit mehr als ein Jahrzehnt Thema aktiver Forschungen. Hier wollen wir dieses Thema nicht ansprechen und lediglich zeigen, wie üblicherweise die Negation als Scheitern in logischen Programmiersprachen implementiert wird. Es wird darauf hingewiesen, dass der hier dargestellter Ansatz keineswegs zufriedenstellend ist, weil er manche Programme unanschaulich bearbeitet.

Die grundidee zur Implementierung der Negation als Scheitern ist einfach: Zur Auswertung eines negativen Literal not Atom wird Atom ausgewertet. Scheitert diese Auswertung, dann gilt not Atom als bewiesen, andernfalls schweitert die Auswertung von not Atom.

Zur Auswertung eines in einer Negation vorkommenden Atom wird der Übersetungsschema Üb_{Body} wie folgt erweitert:

$\text{Üb}_{\text{Body}}(\ [\ \text{not Atom} \mid \text{Sequenz}\ ]\ )$:= `push not`
`push` Atom
`call`
`backtrack?`
`return neg`
`backtrack?`
$\text{Üb}_{\text{Body}}(\ \text{Sequenz}\ )$

In den Übersetzungsschemata soll „`return`“ durch „`return pos`“ ersetzt werden.
Die Befehle `push`, `return` und `backtrack?` werden wie folgt angepasst:

```
push Arg

     if Arg = not
        then stack(T + 1) := nil
        else stack(T + 1) := c_first
     end;
     stack(T + 2) := C;
     if Arg = not
        then stack(T + 3) := P + 4;
        else stack(T + 3) := P + 3
     end;
     stack(T + 4) := Arg;
     C := T + 1;
     R := C + 1;
     T := T + 4;


return Arg

     if Arg = neg
        then stack(R - 1) := nil;
        B := false (* return neg wird nur mit B = true aufgerufen *)
     end;
     if stack(R) \= nil
        then P := stack(R + 1);
             R := stack(R) + 1
        else P := P + 1
     end;
```

Beim Arg = `neg` wird der obere Auswahlpunkt vom Keller nicht entfernt, sondern die erste Zelle des von R gezeigten Auswahlpunkts auf `nil` gesetzt. Der Fall Arg = `pos` entspruicht dem alten Befehl `return`.

```
  backtrack?

     if B then
             while stack(C) = nil
                   and stack(R) \= nil
             do
                C := stack(R);
                R := C + 1;
                T := C + 3
             end-while;
             if stack(C) = nil
                then if stack(C + 3) = not
                        then P := stack(C + 2)  (* neu *)
                        else P := c_last
                     end
                else P := stack(C);
                     stack(C) := c_next(stack(C));
                     B := false
             end-if
          else P := P + 1
     end-if;
```

Beispiel.

```
p :- not q.          c0: unify p    (* 1. Klausel *)
q.                   c1: backtrack?
:- not p, q.         c2: push not
                     c3: push q
                     c4: call
                     c5: backtrack?
                     c6: return neg
                     c7: backtrack?
                     c8: return pos

                     c9: unify q     (* 2. Klausel *)
                    c10: backtrack?
                    c11: return pos

                    c12: push not    (* Ziel *)
                    c13: push p
                    c14: call
                    c15: backtrack?
                    c16: return neg
                    c17: backtrack?
                    c18: push q
                    c19: call
                    c20: backtrack?
                    c21: prompt
```

```
* Initialisierung
  P := c12;
  T := 0;
  C := nil;
  R := nil;
  B := false;

* I := push not     C -> s1: nil
  P := c13          R -> s2: nil
                         s3: c16
                    T -> s4: not

* I := push p            s1: nil
  P := c14               s2: nil
                         s3: c16
                         s4: not
                    C -> s5: c0
                    R -> s6: s1
                         s7: c16
                    T -> s8: p

* I := call              s1: nil
  P := c0                s2: nil
                         s3: c16
                         s4: not
                    C -> s5: c9
                    R -> s6: s1
                         s7: c16
                    T -> s8: p

* I := unify p      Keller unverändert
  B := false
  P := c1

* I := backtrack?   Keller unverändert
  P := c2
```

```
* I := push not          s1: nil
  P := c3                s2: nil
                         s3: c16
                         s4: not
                         s5: c9
                         s6: s1
                         s7: c16
                         s8: p
                   C ->  s9: nil
                   R -> s10: s5
                        s11: c6
                   T -> s12: not
```

```
* I := push q            s1: nil
  P := c4                s2: nil
                         s3: c16
                         s4: not
                         s5: c9
                         s6: s1
                         s7: c16
                         s8: p
                         s9: nil
                        s10: s5
                        s11: c6
                        s12: not
                   C -> s13: c0
                   R -> s14: s9
                        s15: c6
                   T -> s16: q
```

```
* I := call              s1: nil
  P := c5                s2: nil
                         s3: c16
                         s4: not
                         s5: c9
                         s6: s1
                         s7: c16
                         s8: p
                         s9: nil
                        s10: s5
                        s11: c6
                        s12: not
                   C -> s13: c9
                   R -> s14: s9
                        s15: c6
                   T -> s16: q
```

```
* I := backtrack?  Keller unverändert
  P := c6
```

```
* I := return neg        s1: nil
  B := false             s2: nil
  P := c6                s3: c16
                         s4: not
                         s5: c9
                         s6: s1
                         s7: c16
                         s8: p
                         s9: nil
                   R -> s10: s5
                        s11: c6
                        s12: not
                   C -> s13: nil
                        s14: s9
                        s15: c6
                   T -> s16: q
```

```
* I := return neg        s1: nil
  B := false             s2: nil
  P := c6                s3: c16
                         s4: not
                         s5: c9
                   R ->  s6: s1
                         s7: c16
                         s8: p
                         s9: nil
                        s10: s5
                        s11: c6
                        s12: not
                   C -> s13: nil
                        s14: s9
                        s15: c6
                   T -> s16: q
```

```
* I := return neg        s1: nil
  B := false        R -> s2: nil
  P := c16               s3: c16
                         s4: not
                         s5: nil
                         s6: s1
                         s7: c16
                         s8: p
                         s9: nil
                        s10: s5
                        s11: c6
                        s12: not
                   C -> s13: nil
                        s14: s9
                        s15: c6
                   T -> s16: q
```

```
* I := return neg        s1: nil
  B := false        R -> s2: nil
  P := c17               s3: c16
                         s4: not
                         s5: nil
                         s6: s1
                         s7: c16
                         s8: p
                         s9: nil
                        s10: s5
                        s11: c6
                        s12: not
                   C -> s13: nil
                        s14: s9
                        s15: c6
                   T -> s16: q
* I := backtrack?   Keller unverändert
  P := c18
* I := push q            s1: nil
  P := c19               s2: nil
                         s3: c16
                         s4: not
                         s5: nil
                         s6: s1
                         s7: c16
                         s8: p
                         s9: nil
                        s10: s5
                        s11: c6
                        s12: not
                        s13: nil
                        s14: s9
                        s15: c6
                        s16: q
                   C -> s17: c0
                   R -> s18: s1
                        s19: c21
                   T -> s20: q
* I := call              s1: nil
  P := c0                s2: nil
                         s3: c16
                         s4: not
                         s5: nil
                         s6: s1
                         s7: c16
                         s8: p
                         s9: nil
                        s10: s5
                        s11: c6
                        s12: not
                        s13: nil
                        s14: s9
                        s15: c6
                        s16: q
                   C -> s17: c9
                   R -> s18: s1
                        s19: c21
                   T -> s20: q
```

```
* I := unify p      Keller unverändert
  B := true
  P := c1
* I := backtrack?        s1: nil
  P := c9                s2: nil
  B := false             s3: c16
                         s4: not
                         s5: nil
                         s6: s1
                         s7: c16
                         s8: p
                         s9: nil
                        s10: s5
                        s11: c6
                        s12: not
                        s13: nil
                        s14: s9
                        s15: c6
                        s16: q
                   C -> s17: nil
                   R -> s18: s1
                        s19: c21
                   T -> s20: q
* I := unify q      Keller unverändert
  B := false
  P := c10
* I := backtrack?   Keller unverändert
  P := c11
* I := return            s1: nil
  P := c21          R -> s2: nil
                         s3: c16
                         s4: not
                         s5: nil
                         s6: s1
                         s7: c16
                         s8: p
                         s9: nil
                        s10: s5
                        s11: c6
                        s12: not
                        s13: nil
                        s14: s9
                        s15: c6
                        s16: q
                   C -> s17: nil
                        s18: s1
                        s19: c21
                   T -> s20: q
* I := prompt       Keller unverändert     Ausgabe:  more?
  B := true                                Eingabe:        ;
  P := c20
* I := backtrack?    Keller unverändert
  B := true
  P := c21
* I := prompt        Keller unverändert    Ausgabe: no (more) solutions
  stop
```

■

Bemerkung. Anstatt beim `call` oder beim `return neg` die C-Zelle auf `nil` zu setzen, konnte der ganze Auswahlpunkt gelöscht werden. Damit ginge jedoch der im Keller dargestellten Beweisbaum verloren, was das Verständnis der abstrakten Maschine erschwere. ■

5.3.4 Anpassung der abstrakten Maschine zu GroundL

GroundL ist eine Erweiterung von MiniL, die zusammengesetzte Groundterme zulässt, d. h. GroundL entspricht L ohne Variablen. Der Einfachheit halber wird die Behandlung der Negation als Scheitern zu GroundL nicht angepasst.

Konnte ein zusammengesetzter Term in einer Kellerzelle untergebracht werden, bedürfte die in den vorangehenden Absätze eingeführte abstrakte Maschine nur eine Änderung des Befehls unify (gemäß des im Abschnitt 5.2.4.2 "Der Unifikationsalgorithmusëingeführten Algorithmus). Da aber Kellerzellen eine feste Größe haben, mussen zusammengestezte Terme zerlegt auf den Keller gelegt werden.

Da die abstrakten Maschinen MI und MF beide sich die Postfix-Darstellung von Ausdrücke im Keller bedienen, scheint zunächst diese Darstellung auch für MF angebracht zu sein. Tatsächlich erweist sie sich für GroundL als passend. Leider hat sie einen großen, schwer zu behebenden Nachteil, wenn Programme mit Variablen behandelt werden. Aus diesem Ground werden zusammengesetzte Terme im Präfix-Format auf den Keller gebracht. Dazu werden zusammengeseztet L-Terme wie folgt linearisiert:

```
p(a, f(g(b), c), h(d))  ⟶  p/3  a/0  f/2  g/1  b/0  c/0  h/1  d/0
```

Nullstellige sowie nichtnullstellige Funktionssymbole Name / Arity werden mittels Kellerzellen der Gestalt

```
STR  Name  Arity
```

dargestellt. Ein neues Register UP (unification pointer) wird verwendet, das während der Unifikation auf das gegenwärtig überprüfte Symbol zeigt. Bei erfolgreicher Unifikation dieses Symbol wird UP um 1 erhöht. Die organisatorischen Zellen eines Auswahlpunktes mussen nun getrennt vom Atom auf den Keller angelegt werden, was eine gesonderte Behandlung erfordert.

Der Befehl `push` wird wie folgt angepasst:

```
push  STR  Symbol  Arity

  stack(T + 1) := STR  Symbol  Arity;
  T := T + 1;
  P := P + 1;

push  CHP

  stack(T + 1) := c_first ;
  stack(T + 2) := C;
   C := T + 1;
   R := C + 1;
  UP := C + 6; (* Unifikationsregister gesetzt auf Atomanfang *)
   T := T + 7;
   P := P + 1;
```

Die 4. und 5. Zellen im Auswahlpunkt werden später zur Variablenbehandlung benötigt. Die Kellersadresse des Endes des Atoms wird am Ende des Schreibens des Atoms mit dem Befehl

push EndAtom, der unten definiert wird, auf den Keller angelegt. Da die Länge des Code zum Anlegen eines Atoms auf dem Keller im voraus unbekannt ist, kann erst nach Beendigung des Anlegens des Atoms die Rückkehradresse in die dafür vorgesehene Zelle des Auswahlpunktes angelegt werden. Dies geschieht mit dem Befehl push EndAtom.
Gestalt eines Auswahlpunktes

```
           +-----------------------------+
     C ->  |  Addresse des Code für die  |   \
           |       nächste Klausel       |   |
           +-----------------------------+   |
     R ->  |     letzter Wert von C      |   |
           |       (Stack-Adresse)       |   |
           +-----------------------------+   |
           |      Rückkehradresse        |   |
           |      (Code-Adresse)         |   |
           +-----------------------------+   |
           |  Für eine spätere Nutzung   |    >  Organisatorische Zellen
           |                             |   |
           +-----------------------------+   |
           |  Für eine spätere Nutzung   |   |
           |                             |   |
           +-----------------------------+   |
           |     Endes des  Atoms        |   |
           |       (Stack-Adresse)       |   |
           +-----------------------------+   /
    UP ->  |                             |   \
           |                             |    |
           |                             |    |
           |                             |     >  Darstellung des Atoms
           |                             |    |
           |                             |    |
           |                             |   /
           +-----------------------------+
```

```
  push  EndAtom

    stack(C + 2) := P + 3;     (* Rueckkehradresse *)
    stack(C + 5) := T;         (*     Atomende     *)
    P := P + 1;
```

Da die Kellerzellen bereits beim `push CHP` angelegt worden sind, wird beim `push EndAtom` der Wert des Registers T nicht geändert.

Nun bedarf die Unifikation von Groundtermen nur eine leichte Änderung des Befehls unify. Das Atom-Register zeigt auf die nächst zu unifizierende Kellerzelle:

```
  unify  Symbol / Arity

     if not B
        then B := not ( STR  Symbol  Arity ) = stack(UP);
                 (* not unify((STR  Symbol  Arity), stack(UP)) *)
        UP := UP + 1
     end-if;
     P := P + 1;
```

Zwei Groundterme sind unifizierbar, genau dann, wenn sie identisch sind, d. h. genau dann, wenn ihre lineare Präfix-Darstellungen übereinstimmen. Da die Stelligkeiten sowohl auf dem Keller als auch im Code explizit dargestellt sind, kann es nicht vorkommen, dass erst aufgrund unterschiedlicher Länge ihrer Lineardartstellung sich zwei GroundL-Terme als nicht unifizierbar erweisen.

Der Befehl `backtrack?` muss geringfügig geändert werden, damit beim Rücksetzen das

Unifikationsregister UP auf dem Atomanfang (d. h. auf C + 6) und T auf dem Atomende (d. h. stack(C + 5)) gesetzt werden. Dies geschieht am Anfang der Spezifikation von backtrack? unmittelbar nach der While-Schleife :

```
backtrack?

  if B then
          while stack(C) = nil
                and stack(R) \= nil
          do
              C := stack(R);
              R := C + 1
          end;
            T  := stack(C + 5); (* neu: Ad. des Atomendes   *)
           UP := C + 6;         (* neu: Ad. des Atomanfangs *)
          if stack(C) = nil
             then P := c_last
             else P := stack(C);
                  stack(C) := c_next(stack(C));
                  B := false
          end
       else P := P + 1
    end;
```

Die Übersetzungsschemata werden wie folgt verändert, wobei `lin(Atom)` die Linearisierung von Atom liefert:

Üb(:- Sequenz .)	:=	Üb$_{Body}$(Sequenz) `prompt`
Üb(Atom :- Sequenz .)	:=	Üb$_{\text{Head}}$(Atom) Üb$_{\text{Body}}$(Sequenz) `return`
Üb$_{\text{Head}}$(Atom .)	:=	Üb$_{\text{Unify}}$(`lin`(Atom))
Üb$_{\text{Unify}}$([Exp | Sequenz])	:=	Üb$_{\text{Unify}}$(Exp) `backtrack?` Üb$_{\text{Unify}}$(Sequenz)
Üb$_{\text{Unify}}$([])	:=	[]
Üb$_{\text{Unify}}$(Symbol / Arity)	:=	`unify` `STR` Symbol Arity

Üb$_{\text{Body}}$([Atom | Sequenz])	:=	`push` Atom
		`call`
		`backtrack?`
		Üb$_{\text{Body}}$(Sequenz)

Üb$_{\text{Body}}$([])	:=	[]
Üb$_{\text{Push}}$([Exp | Sequenz])	:=	Üb$_{\text{Push}}$(Exp) Üb$_{\text{Push}}$(Sequenz)
Üb$_{\text{Push}}$([])	:=	[]
Üb$_{\text{Push}}$(Symbol / Arity)	:=	`push` `STR` Symbol Arity

Bemerkungen.

1. Da die Darstellung des Termes mit der oberen Kellerzelle endet, ist für GroundL die explizite Angabe der Kelleradresse des Atomendes nicht nötig. Nicht so für L, weil nach dem Atom die Variablen des aufgerufenen Programmklausel sowie der sich aus der Unifikation egebenden Teilterme ebenfalls auf den Keller angelegt werden. Der Einfachheit halber werden schon in diesem Abschnitt alle für L notwendigen organisatorischen Zellen spezifiziert.

2. Die der Behandlung der Negation als Scheitern beruht auf das folgende angepasste Übersetzungsschema:

   $\text{Üb}_{\text{Body}}(\ [\ \texttt{not}\ \text{Atom} \mid \text{Sequenz}\ ]\ ) \quad := \quad$ `push not`
   `push CHP`
   $\text{Üb}_{\text{Push}}(\texttt{lin}(\text{Atom}))$
   `push END`
   `call`
   `backtrack?`
   `return neg`
   `backtrack?`
   $\text{Üb}_{\text{Body}}(\ \text{Sequenz}\ )$

   Nur die Zeilen 2 bis 4 des Codes sind neu.

■

5.3.5 Behandlung von Variablen

In diesem Abschnitt wird erläutert, wie die in den vorangehenden Absätze dargestellte abstrakte Maschine zur Behandlung von Variablen angepasst werden kann. Der Einfachheit halber wird die Behandlung der Negation als Scheitern bei L-Programmen mit Variablen nicht angepasst. Das Prinzip dazu ist aber ähnlich wie für GroundL-Programme.

Die Linearisierung von L-Termen mit Variablen erfolgt gleich wie für GroundL-Termen, z. B.:

`p(X, f(g(b), Y), h(Y))` $\longrightarrow$ `p/3  X  f/2  g/1  b/0  Y  h/1  Y`

Darstellung von Variablen auf dem Keller. Variablen werden mittels Speicherzellen der Gestalt

`VAR  Name  Stack-Add`

im Keller dargestellt, wobei der Parameter Stack-Add entweder den Wert `nil` hat, falls die Variable mit keinem anderen Term des Kellers gebunden ist, andernfalls die Kelleradresse eines bereits im Keller dargestellten Termes.

Die Frage stellt sich noch, wie Stackadresse für Variablen festgestellt werden, die in den Köpfe von Programmklauseln vorkommen, d. h. von Variablen die im Code im Befehl `unify` vorkommen.

Dazu bedienen wir uns den bereits für imperativen Programmiersprachen eingeführte Ansatz. Eine Programmklausel kann wie eine Prozedur einer Imperativen Programmiersprache, die Resolution mit einer Programmklausel kann ähnlich wie einen Prozeduraufruf angesehen werden. Inkarnationen aller in der Programmklausel vorkommenden Variablen stellen eine lokale Umgebung dar. Sie sollen bei einer Resolution mit der Klausel auf den Keller angelegt werden.

In der Umgebung werden die Variablennamen angegeben, damit beim Anlegen von Rumpfatome die Kelleradressen dieser Variablen ermittelt werden können. (Die Variablennamen können auch zur Übersetzungszeit durch Positionen in der Umgebung ersetzt werden. Obwohl dies effizienter und für eine echte Implementierung angebracht ist, wird dieser Ansatz der Klarheit halber hier nicht verfolgt.)

Zu jedem Rumpfatom gehöhrt also eine lokale Umgebung, die zuvor auf dem Keller angelegt wurde. Folglich muss im Auswahlpunkt eine organisatorische Zelle zur Speicherung der Kelleradresse dieser Umgebung verwendet werden. Ein neues Register E (environment) zeigt auf die zuletzt auf den Keller angelegte Umgebung:
Gestalt eines Auswahlpunktes

```
            +-----------------------------+
      C ->  |  Adresse des Code für die   |   \
            |       nächste Klausel       |    |
            +-----------------------------+    |
      R ->  |     letzter Wert von C      |    |
            |       (Stack-Adresse)       |    |
            +-----------------------------+    |
            |      Rückkehradresse        |    |
            |      (Code-Adresse)         |    |
            +-----------------------------+     > Organisatorische Zellen
(* neu *)   |    Adresse der dem Atom     |    |
            |  zugehörigen lokalen Umg.   |    |
            +-----------------------------+    |
            |  Für eine spätere Nutzung   |    |
            |                             |    |
            +-----------------------------+    |
            |    Adresse des Atomendes    |    |
            |       (Stack-Adresse)       |   /
            +-----------------------------+
     UP ->  |                             |   \
            |                             |    |
            |                             |    |
            |                             |     >  Darstellung des Atoms
            |                             |    |
            |                             |    |
            |                             |   /
            +-----------------------------+
```

Nach dem Aufruf einer Programmklausel wird auf den Keller unmittelbar nach dem zuletzt angelegten Auswahlpunkt die lokale Umgebung des Aufrufes angelegt:

```
            +-----------------------------+
      E ->  |                             |   \
            |                             |    |
            |                             |    |   Lokale Umgebung der
            |                             |     >     aufgerufenen
            |                             |    |      Programmklausel
            +-----------------------------+    |
            |   EndEnv                    |   /
            +-----------------------------+
            |                             |
            |                             |
```

Die im Ziel vorkommende Variablen belegen die Kellerzellen 1 bis C - 2, wenn C die Adresse des Auswahlpunktes des ersten Zielatomes ist:

```
          +----------------------------+
E = 1 ->  |                            |   \
          |                            |    |
          |                            |    |
          |                            |     >  Lokale Umgebung des
          |                            |    |         Zieles
          +--------------------------- +    |
          |   EndEnv                   |   /
          +----------------------------+
    C ->  |                            |
          |                            |
```

Die von E gezeigte Umgebung ist diejenige, die zur Festellung der Kellersadresse der im Unify-Befehle vorkommenden Variablen benutzt werden soll. E wird mit dem Befel push EndEnv auf die Adresse der zuvor angelegten lokalen Umgebung gesetzt. Zur Feststellung der Kelleradresse einer in einem Push-Befehl vorkommende Variable soll anstatt von E auf die im Auswahlpunkt unter der Adresse C + 3 gespeicherten Kelleradresse zugegriefen werden.

Im übernächsten Abschnitt (5.3.4.4 „ML-Befehle (außer unify)“) wird eine Funktion `s_add` eingeführt, die die zwei Modi `unify` und `push` kennt und dementspechend auf die passende lokale Umgebung zugreift.

Bemerkung. Die Variablen in den (nicht übersetzten) Programmklausel können durch Positionen in der Variablenliste ersetzt werden, was die Aufrufe der Funktion `s_add` überflüssig machen würde. ∎

Rücksetzen mit Variablen und der Rücksetzenskeller. Bei der Unifikation können Variablen gebunden werden. Dies ist z. B. mit dem Follgenden L-Programm der fall:

```
p(Y) :- q(Y).
q(a).
q(b).
:- p(X).
```

Die folgenden Übersetzung dieses L-Programm baut Auswahlpunkte der Gestalt auf, wie sie im letzten Abschnitt eingeführt wurde. (Die Übersetzungsschemata für L-Programme werden im nächsten Abschnitt 5.3.4.3 „Code-Erzeugung für ML“ formal eingeführt.)

```
 c0: push BegEnv    (* 1. Klausel *)
 c1: push VAR Y
 c2: push EndEnv
 c3: unify STR  p  1
 c4: backtrack?
 c5: unify VAR  Y
 c6: backtrack?
 c7: push CHP
 c8: push STR  q  1
 c9: push VAR  Y
c10: push EndAtom
c11: call
c12: backtrack?
c13: return pos
```

```
c14: pushBegEnv    (* 2. Klausel *)
c15: push EndEnv
c16: unify STR  q  1
c17: backtrack?
c18: unify STR  a  0
c19: backtrack?
c20: return pos



c21: pushBegEnv    (* 3. Klausel *)
c22: push EndEnv
c23: unify STR  q  1
c24: backtrack?
c25: unify STR  b  0
c26: backtrack?
c27: return pos



c28: pushBegEnv    (*    Ziel    *)
c29: push VAR X
c30: push EndEnv
c31: pusg CHP
c32: push STR  p  1
c33: push VAR  X
c34: push EndAtom
c35: call
c36: backtrack?
c37: prompt
```

Beim Rücksetzen müssen die Variablenbindungen rückgängig gemacht werden. Betrachten wir den bei der Auswertung des vorangehenden Beispieles aufgebauten Keller:

```
Nach Ausführung des Befehles c35: call

 E ->  s1: VAR X  nil       (* lokale Umgebung des Zieles *)
       s2: EndEnv
 C ->  s3: nil         (* statt c14, vor call war hier c0 *)
 R ->  s4: nil
       s5: c33
       s6: s1
       s7:
       s8: s10
UP ->  s9: STR  p  1
      s10: VAR  X  s1

P :=  c0
```

```
Nach Ausführung des Befehles c11:

       s1: VAR  X  nil   (* lokale Umgebung des Zieles *)
       s2: EndEnv
       s3: nil           (* statt c11 *)
       s4: nil
       s5: c33
       s6: s1
       s7:
       s8: s10
       s9: STR  p  1
      s10: VAR  X  s1
 E -> s11: VAR  Y  s1   (* lokale Umgebung des 1. Aufrufes *)
      s12: EndEnv
 C -> s12: c21          (* statt c0 *)
 R -> s14: s3
      s15: c11
      s16: s11
      s17:
UP -> s18: STR  q  1
      s19: VAR  Y  s11

 P := c12
Nach Ausführung des Befehles c18:

       s1: VAR  X  s20 (* Ad. vom c16: unify STR a 0 angelegt *)
       s2: EndEnv
       s3: nil         (* statt c12 *)
       s4: nil
       s5: c33
       s6: s1
       s7:
       s8: s10
       s9: STR  p  1
      s10: VAR  X  s1
 E -> s11: VAR  Y  s1   (* lokale Umgebung des 1. Aufrufes *)
      s12: EndEnv
 C -> s12: c21          (* statt c0 *)
 R -> s14: s3
      s15: c11
      s16: s11
      s17:
      s18: STR  q  1
      s19: VAR  Y  s11  (* Ad. vom c5: unify VAR Y angelegt *)
UP -> s20: STR  a  0    (* Zelle vom c16: unify STR a 0 angelegt *)
```

Zum Rücksetzen (durch prompt nach einer Eingabe von `";"`) muss die Bindung

```
   s1: VAR  X  s20
```

rückgängig gemacht werden, d. h. diese Kellerzelle muss durch die folgende Kellerzelle ersetzt werden:

```
   s1: VAR  X  nil
```

Zum Rückgängigmachen von Variablenbindungen wird ein Rücksetzenskeller zur Speicherung der „Variablenbindungsspuren“ verwendet. Daher heißt dieser Keller trail (Spur). Das Trail wird wie folgt verwendet: Beim Anlegen einer Variablenbindung (wie etwa s1: VAR X s20), wird die Adresse der betroffenen Kellerzelle (im Beispiel s1) auf dem Trail angelegt. Beim Rücksetzen wird der Trail geräumt, wobei die Variablen, deren Adresse in der geräumten Trail-Zelle liegen auf nil gesetzt werden.

Damit der Rücksetzenskeller synchron mit dem Keller stack wächst und schrümpft, wird die Adresse seiner zuletzt angelegt Zelle (seiner oberen Zelle) in den zuletzt angelegten Auswahlpunkt gespeichert. Dazu wird die lezte noch nicht benutzte reservierte Zelle eines Auswahlpunktes verwendet.

Der Rücksetzenskeller verhält sich also wie ein Zwillingskeller vom stack, der synchron mit stack pulsiert.

Die Frage stellt sich, warum ein zweiter Keller notwendig ist. Der Grund liegt darin, dass bei der unifikation der Keller stack wachsen kann, wie im vorangehenden Beispiel mit dem Befehl c16: unify STR a 0. Es ist also nicht möglich, die Kellerzellen nach dem zuletzt angelegten Auswahlpunkt bzw. nach der zuletzt angelegeten Umgebung dafür die nutzen, die „Variablenbindungspuren“ zu speichern.

Zur Verwaltung des Rücksetzenskeller trail wird ein neues Register TT (top of trail) benutzt.

Bemerkung. Aufgrund der besonderen Gestalt des Kellers sowie des Codes wurde der Befehl c16: unify STR a 0 wie ein Push-Befehl behandelt. Dabei handelt es sich um einen wichtige Aspekt der Unifikation mit Variablen, worauf im Abschnitt 5.3.4.6 „Unifikation“ zurückgekommen wird. ■

Der Unifikationskeller. Betrachten wir folgendes Beispiels eines L-Programms:

```
q(a).
r(b).
p(a, b).
:- q(X), r(Y), p(X, Y).
```

Nach der erfolgreichen Auswertung vom Teilziel `q(X), r(Y)` und dem Anlegen eines Auswahlpunktes für das Atom `p(X, Y)` haben die Keller stack und trail die folgenden Gestalt:

```
 s1: VAR  X  s11
 s2: VAR  Y  s19
 s3: EndEnv
----------------  Auswahlpunkt
 s4: nil          Codeadresse der Üb. der nächsten Klausel
 s5: nil          Kelleradresse des letzten Auswahlpunktes
 s6: c?           Rückkehradresse
 s7: t0           top of trail
 s8: s10          Atomende
 s9: STR  q  1
s10: VAR  X  s1
----------------
s11: STR  a  0
----------------  Auswahlpunkt
s12: nil          Codeadresse der Üb. der nächsten Klausel
s13: s4           Kelleradresse des letzten Auswahlpunktes
s14: c?           Rückkehradresse
s15: t1           top of trail
s16: s18          Atomende
s17: STR  r  1
s18: VAR  Y  s2
s19: STR  b  0
```

```
      ----------------  Auswahlpunkt
  C -> s20: nil         Codeadresse der Üb. der nächsten Klausel
  R -> s21: s12         Kelleradresse des letzten Auswahlpunktes
       s22: c?          Rückkehradresse
       s23: t2          top of trail
       s24: s27         Atomende
 UP -> s25: STR  p  2
       s26: VAR  X  s1
  T -> s27: VAR  Y  s2
       ----------------


       t1: s1
 TT->  t2: s2
```

Nun soll die folgenden Befehlsequenz durchgeführt werden:

```
ci1: unify STR  p  2
ci2: unify STR  a  0
ci3: unify STR  b  0
```

Diese Sequenz gibt die Präfix-Darstellung von zusammengesetzten LTerme wider. Auf dem Keller wurde das p-LTerm ebenfalls in der Präfix-Darstellung angelegt. Die Unifikation kann jedoch nicht wie bei GroundL-Programme synchron zwischen Keller und Code erfolgen, weil ein Nebeneffekt vorherigen Unifikationen Indirektionen im Keller-LTerm sind: Wobei die Kellerzeller s25 beim Befehl ci1 betrachtet werden kann, sollen die Kellerzellen s11 bzw. s19 bei den Befehle ci2 bzw. ci3 berücksichtigt werden, d. h. die Zellen, die sich auch der sogenannten Dereferenzierung der Adresse s26 bzw. s27 ergeben.

Die Derefenzierungsprozedur wird wie folgt spezifiziert:

```
procedure deref(Add: Kelleradresse): Kelleradresse
   var Add2: nil oder Kelleradresse;
begin
   if stack(Add) = STR Symbol Arity
      then return Add
      else let stack(Add) = VAR Symbol Add2;
           if Add2 = nil
              then return Add
              else return deref(Add2)
           end
   end
end
```

Nach der Durchführung des Befehles `ci1` soll also zur Durchführung des Befehles soll also das Unifikationsregister UP auf `deref(s26) = s11` statt `s26` zeigen, und nach dem Befelh ci2 zur Durchführung des Befehles `ci3` auf `deref(s27) = s19` zu zeigen. Um diese „Sprünge" zu bewältigen, wird ein Unifikationskeller us (unification stack) verwendet. Wenn `deref(UP)` $\neq$ `UP`, dann wird zunächst den gegenwärtigen Wert von `UP` 1+ auf den Unifikationskeller gerettet, dann `UP` auf `deref(UP)` gesetzt. Die Anzahl der Unifikationsbefehle bis zum Rücksprung wird mit einem neuen Register `AC` (argument counter) gezählt. AC wird mit der Stelligkeit des zusammengesetzten LTerm initialisiert und bei jedem Argument um 1 verringert und um die Stelligkeit dieses Arguments erhöht.

Der Unifikationskeller stellt also lediglich ein Werkzeug der Unifikation dar. Sie wird also beim Anlegen eines neuen Auswahlpunktes geleert, d. h. das Register `UT` (top of unification stack), das auf ihre obere Zelle zeigt, wird auf 0 gesetzt.

Code-Erzeugung für ML. Wir setzen voraus, dass die Syntaxanalyse eine Darstellung (VarSeq, K) einer Programmklauseln K erzeugt, die die (eventuell leere) Liste VarSeq der in der Programmklausel vorkommenden Variablen liefert. Es ergibt sich folgende Übersetzungsschemata:

Üb(VarSeq, :- Sequenz .)	:=	`push BegEnv`
		Üb_{Env}(VarSeq)
		Üb_{Body}(Sequenz)
		`prompt`

Üb(VarSeq, Atom :- Sequenz .)	:=	`push BegEnv`
		Üb_{Env}(VarSeq)
		Üb_{Head}(Atom)
		Üb_{Body}(Sequenz)
		`return pos`

Üb(VarSeq, Atom .)	:=	`push BegEnv`
		Üb_{Env}(VarSeq)
		Üb_{Head}(Atom)
		`return pos`

Üb_{Env}([Symbol | Sequenz])	:=	`push VAR` Symbol Üb_{Env}(Sequenz)
Üb_{Env}([])	:=	`push EnvEnd`
Üb_{Head}(Atom)	:=	Üb_{Unify}(`lin`(Atom))
Üb_{Unify}([Exp | Sequenz])	:=	Üb_{Unify}(Exp) `backtrack?` Üb_{Unify}(Sequenz)
Üb_{Unify}([])	:=	[]
Üb_{Unify}(Symbol / Arity)	:=	`unify STR` Symbol Arity
Üb_{Unify}(Symbol)	:=	`unify VAR` Symbol

Üb_{Body}([Atom | Sequenz])	:=	`push CHP`
		Üb_{Push}(lin(Atom))
		`push EndAtom`
		`call`
		`backtrack?`
		Üb_{Body}(Sequenz)

Üb_{Body}([])	:=	[]
Üb_{Push}([Exp | Sequenz])	:=	Üb_{Push}(Exp) Üb_{Push}(Sequenz)
Üb_{Push}([])	:=	[]
Üb_{Push}(Symbol / Arity)	:=	`push STR` Symbol Arity
Üb_{Push}(Symbol)	:=	`push VAR` Symbol

ML-Befehle (außer unify). Initialisierung:

```
 P := c_goal;    (* code-Adresse der Übersetzung des Zieles *)

 T := 0;         (* leerer Keller *)
 C := nil;
 R := nil;
 B := false;
TT := 0;         (* leeres trail *)
```

```
push  Arg

  case Arg of
     STR  Symbol  Arity : stack(T + 1) := STR  Symbol  Arity ;
                          T := T + 1                                  |
     VAR  Symbol        : stack(T + 1) := VAR  Symbol  s_add(Symbol, push) ;
                          T := T + 1                                  |
     CHP                : stack(T + 1) := c_first
                          stack(T + 2) := C;
                          stack(T + 4) := E;
                          stack(T + 5) := TT; (* top of trail *)
                          C := T + 1;
                          R := C + 1;
                          UP := C + 6;        (* Atomanfang *)
                          T := T + 6;
                          UT := 0;            (* Unifikationskeller leer *)
                          PC := 0;
                          AC := nil                                   |
     EndAtom            : stack(C + 2) := P + 3;
                          stack(C + 5) := T                           |
     BegEnv             : E := nil                                    |
     EndEnv n           : stack(T + 1) := ( EndEnv );
                          T := T + 1;
                          E := T - n                                  |

  end;
  P := P + 1;
```

Die folgende Funktion wurde verwendet:

`s_add(Symbol, Mode)`: liefert `nil` oder eine Kelleradresse

Variablen: add: nil oder Kelleradresse; i: Kelleradresse;

```
begin
   add := nil;
   case Mode of
      unify : i := E                    |
      push  : if C = nil
                 then add = nil
                 else i := stack(C + 3)
              end                       |
   end;
   while stack(i) \= EndEnv and add = nil do
      if stack(i) := VAR  Symbol Address then add := Address end;
      i := i + 1
   end;
   return add
end
```

```
call
  if stack(C) = nil  (* keine Alternative mehr *)
     then B := true;
          P := P + 1
     else P := stack(C);
          stack(C) := c_next(stack(C))
          (* Aktualisierung des Auswahlpunktes *)
     end;


return pos
  P := stack(R + 1);        (* Rückkehradresse *)
  E := stack(R + 2);
  if stack(R) \= nil        (* Es gibt einen vorherigen Auswahlpunkt *)
     then R := stack(R) + 1 (* R-Wert des vorherigen Auswahlpunktes  *)
  end;


backtrack?
  if B then
          while stack(C) = nil
                and stack(R) \= nil
          do
              C := stack(R);
              R := C + 1
          end;
           T  := stack(C + 5);        (*   Adresse des Atomendes  *)
           E  := T + 1;               (* Adresse des lokalen Umg. *)
           UP := C + 6;               (*  Adresse des Atomanfangs *)
           UT := 0;                   (* Unifikationskeller leer *)
           PC := 0;
           AC := nil;
           for i := stack(C + 4) + 1 to i := TT do
              if trail(i) <= T
                 then let stack(trail(i)) = VAR Symbol Add;
                      stack(trail(i)) := VAR Symbol nil
              end
           end;
          TT := stack(C + 4);
          if stack(C) = nil
             then P := c_last
             else P := stack(C);
                  stack(C) := c_next(stack(C));
                  B := false
          end
       else P := P + 1
    end;


prompt
  if B
     then write('no (more) solutions');
                             (* die abstrakte Maschine haelt *)
     else display;
          write('more ?');
          if read = ';'
             then B := true;
                  P := P - 1; (* Ad. von backtrack? vor prompt *)
             else stop
          end
   end;
```

```
Prozedur display:
   var i: Kelleradresse;
begin
   i := 1;
   while stack(i) \= EndEnv do
      let stack(i) = VAR Symbol Add;
      display_term(deref(i)); write(" / "); write(Symbol); write(NewLine)
   end
end

display_term(Add: Kelleradresse):
begin
   case stack(deref(Add)) of
      VAR Symbol nil : write(Symbol)
      |
      STR Symbol Arity : write(Symbol);
                         write("(");
                         display_term(deref(Add)+1);
                         write(")")
      |
   end
end
```

Der Befehl unify. Mit dem Rücksetzen (s. Abschnitt 5.3.1 „Eine abstrakte Maschine für MiniL") stellt die Implementierung der Unifikation in einer Registersprache den eigentlichen Beitrag der logischen Programmierung zum Übersetzerbau.

Zur Spezifikation des Befehls unify gibt es 4 Fälle zu unterscheiden.

Fall 1: Das Argument des Unify-Befehls ist eine eine ungebundene Variable V, d. h. `deref(s_add(V, unify)) = nil`

Fall 2: Das Argument des Unify-Befehls ist eine gebundene Variable V, d. h. `deref(s_add(V, unify))` $\neq$ `nil`

Fall 3: Das Argument des Unify-Befehls ist keine Variable und UP zeigt (direkt oder indirekt) auf eine ungebundene Variable, d. h. stack(deref(UP)) = VAR Symbol nil

Fall 4: Das Argument des Unify-Befehls ist keine Variable und UP zeigt (direkt oder indirekt) auf eine gebundene Variable, d. h. stack(deref(UP)) = STR Symbol Arity

Im 1. Fall soll die mit V gemeinten Variable an der Adresse UP gebunden werden. Zeigt UP auf ein zusammengesetzter LTerm, dann sollen soviel Kellerzellen übersprungen werden wie die Stelligkeit von dieser LTewrm groß ist. Sind einige Argumente vom von UP gezeigten LTerm selber zusammengesetzt, dann muss der Sprung um ihre Stelligkeit erhöht werden. Dazu wird ein Register SC (skip counter) verwendet.

Im 2. Fall mussen 2 auf dem Keller liegenden LTerme unifiziert werden. Dabei können Variablen gebunden werden (d. h. nil durch Kelleradresse in VAR Symbol nil Kellerzeller ersetzt werden), aber keine neuen Zellen auf den Keller gelegt. Zur Unifikation von zwei bereits im Keller vorhandenen LTermen bedienen wir uns eine Prozedur unify, deren Implementierung unter erörtet wird.

Im 3. Fall soll die vom Register UP gezeigten ungebundenen Variablen mit dem in den kommenden Unify-Befehle definierten LTerm gebunden werden. Da dieser LTerm auf dem Keller noch nicht liegt, muss er angelegt werden. Ein Push-Modus wird eingesetzt der bewirkt, dass die kommenden Unify-Befehle die Wirkung von Push-Befehle haben. Zur Feststellung der Dauer des Push-Modus muss lediglich rekursiv die Stelligkeit der im Code vorkommenden LTermen berücksichtigt werden.

Im 4. Fall wird das äußere Symbol nebst Stelligkeit mit dem Symbol nebst Stelligkeit des vom UP gezeigten Symbol verglichen. Bei einer Stelligkeit von mindestens 1 soll gegebenfalls UP auf die Kelleradresse der Argumente der Reihe nach gesetzt werden, nachdem seinem alten Wert in den Unifikationskeller gerettet wurde.

Zur Verwaltung der Sprünge und dem Unifikationskeller werden die Makroinstruktionen `add_AC`, `restore_AC_UP?` und `save_AC_UP?` verwendet.

```
unify Arg

if PC >= 1         (*  Push-Modus   *)
 then
   case Arg of
        STR Symbol Arity : stack(T + 1) := STR Symbol Arity                 |
        VAR Symbol       : stack(T + 1) := VAR Symbol s_add(Symbol, unify) |
   end-case;
   T := T + 1;
   PC := PC - 1 + arity(Arg)
 else case Arg of (* NonPush-Modus *)
          VAR Symbol       : if deref(s_add(Symbol, unify)) = nil
                              then let stack(deref(s_add(Symbol, unify)) =
                                                            VAR Symbol2 nil;
                                    stack(deref(s_add(Symbol, unify)) :=
                                                            VAR Symbol2  UP;
                                   trail(TT+1):=deref(s_add(Symbol,unify));
                                   TT := TT+1;
                                   SC := arity(stack(UP));
                                   while SC >= 1 do            (*  Skip  *)
                                      UP := UP + 1;
                                      SC := SC - 1 + arity(stack(UP))
                                   end-while
                               else
                                 B := not unify(deref(s_add(Symbol, unify)),
                                                                          UP)
                             end-if;
                             add_AC(-1);
                             restore_AC_UP?;
                             UP := UP + 1
         |
         STR Symbol Arity : if stack(deref(UP)) = VAR Symbol2 nil
                              then stack(deref(UP)) := VAR Symbol2 T+1;
                                   trail(TT + 1) := deref(UP);
                                   TT := TT + 1;
                                   stack(T + 1) := STR Symbol Arity;
                                                             (* push *)
                                   T := T + 1;
                                   PC := Arity (* Push-Modus falls >= 1 *)
                                   add_AC(-1);
                                   restore_AC_UP?;
                                   UP := UP + 1
```

```
                                  else let stack(deref(UP)) =
                                                          STR Symbol2 Arity2;
                                      if Symbol1 \= Symbol2
                                         or Arity \= Arity2
                                        then B := true (* Unif. scheitert *)
                                        else if Arity >= 1
                                                 then add_AC(Arity)
                                                        (* Arity + 1 - 1 *)
                                                 else add_AC(-1)
                                              end;
                                              restore_AC_UP?;
                                              UP := UP + 1;
                                              save_AC_UP?  (*  Sprung?  *)
                                       end-if
                              end-if
         |
       end-case
  end-if;
  P := P + 1;
```

Makroinstructionen:

```
  add_AC(n):   (* n ganze Zahl *)
     if AC \= nil
        then AC := AC + n
     end

  restore_AC_UP?:
     if AC = 0
        then AC := us(UT - 1);
             UP := us(UT);
             UT := UT - 2
     end

  save_AC_UP? :
     if UP <= stack(C + 5) and deref(UP) \= UP and arity(stack(deref(UP))) \= 0
        then us(UT + 1) := AC;
             us(UT + 2) := UP + 1; (* wg. der Indirektion, wo fortgesetzt *)
             UT := UT + 2;         (*           werden soll               *)
             UP := deref(UP);      (* Sprung *)
             AC := 0
      end
```

Funktionen:

```
  deref(Add: Kelleradresse): Kelleradresse
     var Add2: nil oder Kelleradresse;
  begin
     if stack(Add) = STR Symbol Arity
        then return Add
        else let stack(Add) = VAR Symbol Add2;
             if Add2 = nil
                then return Add
                else return deref(Add2)
             end
     end
  end
```

```
  arity(Zelle: Kellerzelle): natürliche Zahl
  begin
     if Zelle = VAR Symbol Add   (*  ohne deref aufzurufen, \.dh  *)
        then return 0            (* ungeachtet des Wertes von Add *)
        else let Zelle = STR Symbol Arity;
             return Arity
     end
  end
```

Die Prozedur unify zur Unifikation von zwei auf dem Keller liegenden LTermen.

```
unify(Add1, Add2: stack-Adresse): Wahrheitswert
   var Weiter: Wahrheitswert;
       Stack : Keller;
       D1, D2: stack-Adresse;
begin
   push(Add1, Stack);
   push(Add2, Stack);
   Weiter := true;
   while Weiter and not leer(Stack) do
      D1 := deref(pop(Stack));
      D2 := deref(pop(Stack));
      if D1 \= D2
         then if stack(D1) = ( VAR V1 nil )
                 then stack(D1) = ( VAR V1 D2 );
                      trail(TT + 1) := D1;
                      TT := TT + 1
                 else if stack(D2) = ( VAR V2 nil )
                         then stack(D2) = ( VAR V2 D1 );
                              trail(TT + 1) := D2;
                              TT := TT + 1
                         else let stack(D1) = STR  Symbol1 Arity1;
                               stack(D2) = STR  Symbol2 Arity2;
                               if Symbol1 \= Symbol 2 or Arity1 \= Arity2
                                  then Weiter := false
                                  else for i := 1 to i := Arity1 do
                                           push(V1 + i, Stack);
                                           push(V2 + i, Stack)
                                       end
                               end
                      end
              end
      end
   end-while
   return Weiter
end
```

Was für einen Hilfskeller soll für die Prozedur unify verwendet werden? Ob ein bereits vorhandener Keller in Frage kommt, hängt davon ab, ob und gegebenfalls wie die Prozedur unify disen Keller verändert.

Die Prozedur unifty kann den Keller stack dadurch verändern, dass sie eine Zelle der Gestalt (VAR Symbol nil) in eine Zelle der Gestalt (VAR Symbol S-Add) mit A-Add $\bar{\text{n}}$il umwandelt. Weder legt sie neue Zellen auf stack an, noch löscht sie manche. Das heißt, sie verändert aber den Wert des Registers T nicht. Es ist also möglich, den für die Prozedur unify notwendigen Keller auf den Keller stack zu legen, d. h. in der Tat, den Keller stack als Hilfskeller der Prozedur unify zu nutzen.

Nach jedem Aufruf der Prozedur unify, muss der Hilfskeller geleert werden. Wird der Keller stack als Hilfskeller verwendet, dann geschieht dies dadurch, dass der Zustand vom Register T vor jedem Aufruf der prozedur unify gerettet wird - dazu empfiehlt sich der Unifikationskeller us -, und nach Beendingung des Aufrufes wiederhergestellt wird.

Wird der Keller stack als Hilfskeller der Prozedur unify verwendet, ergibt sich also folgende Änderung des Befehls unify. Die Zeile

```
B := not unify(deref(s_add(Symbol, unify)), UP)
```

wird ersetzt durch die Sequenz

```
us(UT + 1) := T;
UT := UT + 1;
B := not unify(deref(s_add(Symbol, unify)), UP);
T := us(UT);
UT := UT - 1;
```

Da die Prozedur unify den Unifikationskeller us, der zu Rettung von Stack-Adresse vor Sprünge eingeführt wurde, nicht verändert, ist es auch möglich diesen Keller anstatt von stack als Hilfskeller der Prozedur unify zu nutzen. in diesem Fall sollte dann der Keller stack zu Rettung des Wertes von UT verwendet werden.

Da aber die Prozedur unify aber den Rücksetzenskeller trail bei jeder Variablenbindung verändert, kann dieser Keller als Hilfskeller der Prozedur unify nicht benutzt werden.

5.3.6 Spracherweiterungen

- Once und Cut (!)

  Zur Implementierung von Once oder Cut, die zur Vermeidung des Rücksetzen in einem Klauselrumpf eingesetzt werden können, wird eine ähnliche Technik verwendet wie für die Negation als Scheitern. Beim Cut muss der Skopus ermittelt werden, der sich vom Anfang des Rumpfes, in dem das Cut vorkommt, bis zum Cut ausdehnt.

- Arithmetische Ausdrücke

  Die Programmiersprache Prolog, wovon L eine Vereinfachung ist, bittet den Sprachkonstruckt „is“ zu Berechnung von arithmetischen Ausdrücke. Diese können wie bei einer imperativen Programmiersprachen „auf dem Keller“ berechnet werden.

- If-then-else

  Die Programmiersprache Prolog bittet einen Konstrukt der Gestalt `A -> B ; C` mit der Bedeutung `if A then B else C`. Dieser Konstrukt wird ähnlich wie bei einer imperativen Programmiersprache implementiert.

- assert und retract

  Die Programmiersprache Prolog bittet die Sprachkonstrukte assert und retract zur Änderung des Programms zur Laufzeit. Die in dieser Weise hinzugefügten Regeln oder Fakten mussen Übersetzt werden. Ihre Übersetzungen mussen zu den Übersetzungen der Regeln und Fakten im Code hinzugefügt werden, damit sie berücksichtigt werden. Diese Sequentialisierung stellt einige Probleme, u.a. was die Semantik der Sprache betrifft, die für Prolog erst mitte der 80er Jahren in einem Standard festgesttelt worden sind. (Vorher waren die verschiedenen Impelementierungen von Prolog in dieserr Hinsicht nicht alle gleich.)

5.3.7 Code-Optimierungen

Einige Optimierungen sind offensichtlich und ergeben sich daher, dass die abstrakte Maschine bewusst einfaqch gehalten wurden, in der Hoffnung, sie verständlich zu machen:

- Zum anlegen einer leeren lokalen Umgebung, wäre es effizienter E auf nil zu setzen, anstatt die Befehlsequenz push BegEnv pushEndEnv durchzuführen.
- Wenn C auf nil zeigt, ist es effizienter beim return den oberen Auswahlpunkt zu löschen, anstatt C auf nil zu setzen.
- Nach Beendigung der Auswertung einem negierten Ausdruck, kann ebenfalls der Keller von allen Auswahlpunktes dieses negierten Ausdrucks geräumt werden, da kein Rücksetzen eintreten kann. Beim once oder cut ist ebenfalls solch ein Räumen möglich.

Andere Optimierungen sind weniger unmittelbar:

- Nur manche Variablen einer Programmklausel brauchen auf den Keller angelegt zu werden (s. Übung im Abschnitt 5.3.4.1 „Darstellung von Variablen auf dem Keller“).
- Letzes-Ziel-Optimierung. Unter diesem Namen versteht man die Endrekursion. Im Falle von logischen Programme ist sie nur dann anwendbar, wenn keine weitere Klausel ein Rücksetzen auf das letztes Atom ermöglichen konnte. Bei der Endrekursion wird üblicherweise nur das Prädikatssymbol berücksichtigt, wie etwa in:

  ```
  p( ... ) :- A1, A2, ..., p( ... )
  ```
- Auswahl der nächten Programmklausel. Anstatt einer sequentielle Suche durch alle (Übersetzungen der) Fakten und Regeln, können effizienteren Datenstrukturen wie Schlüssel-Transformationsstruckturen (hasch tables) oder Baumstrukturen eingesetzt werden.

6 Übersetzerentwicklung und -portierung

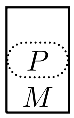
Übersetzer sind ziemlich komplexe Programme. Deshalb ist es sinnvoll, sie in höheren Programmiersprachen zu implementieren. In vielen Fällen ist die Quellsprache eines Übersetzers eine höhere Programmiersprache. In solchen Fällen hat es sich als nützlich erwiesen, die Quellsprache selbst als Implementierungssprache des Übersetzers zu verwenden. Das setzt eine Implementierungsmethode voraus, bei der zunächst ein erster Übersetzer für einen kleinen Kern der Sprache implementiert wird, der dann schrittweise verfeinert wird. Teil der Verfeinerungsschritte ist jeweils eine Übersetzung des Übersetzers „mit sich selbst“. Diese Implementierungsmethode wird Bootstrapping genannt. Eine Variante der Methode erlaubt es, den Übersetzer mit möglichst geringem Aufwand auf verschiedene Plattformen zu übertragen.

Dieses Kapitel führt zunächst T-Diagramme als Illustrationshilfsmittel ein und erläutert dann die Bootstrapping-Methode genauer. Der letzte Abschnitt behandelt den darauf beruhenden systematischen Portierungsansatz.

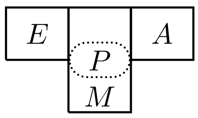
6.1 T-Diagramme

Ein T-Diagramm ist eine graphische Veranschaulichung für Programme, bei denen mehrere Programmiersprachen eine Rolle spielen, wie es insbesondere für Übersetzer der Fall ist. Die Bezeichnung T-Diagramm kommt daher, dass die meisten dieser graphischen Darstellungen eine Gestalt haben, die an den Großbuchstaben T erinnert. Die folgenden Grundmuster von T-Diagrammen sind gebräuchlich:

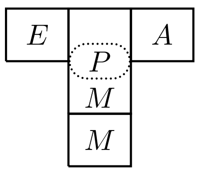
- Programm mit Namen P, das in einer Programmiersprache M geschrieben ist.



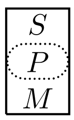
- Programm mit Namen P, das in einer Programmiersprache M geschrieben ist und eine Eingabe E in eine Ausgabe A umwandelt.



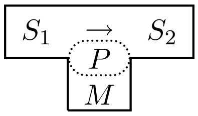
- Ausführung dieses Programms auf einer Maschine, deren Maschinensprache M ist.



- Das Programm P ist ein Interpretierer, der in einer Programmiersprache M geschrieben ist und Programme in einer Programmiersprache S interpretieren kann.

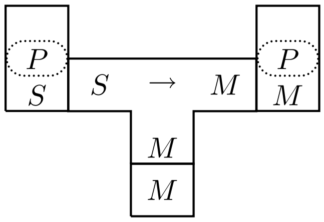


- Das Programm P ist ein Übersetzer, der eine Quellspache S_1 in eine Zielsprache S_2 übersetzt und in der Sprache M implementiert ist.

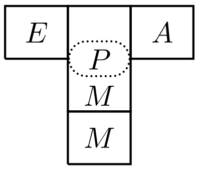


In allen T-Diagrammen kann der Name P des Programms auch weggelassen werden. Man kann T-Diagramme nun wie Dominosteine aneinanderreihen, wenn angrenzende Felder gleich bezeichnet sind. Damit lassen sich viele Übersetzungsfälle sehr anschaulich darstellen.
Beispiele:

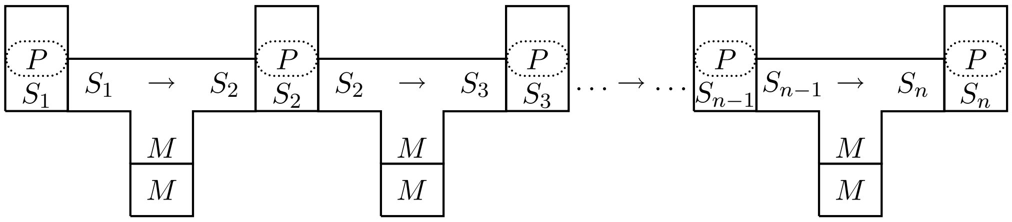
- Seien S und M zwei Programmiersprachen. Übersetzung eines S-Programms P in ein M-Programm mit einem (namenlosen) Übersetzer, der in M geschrieben ist und auf einer M-Maschine läuft:



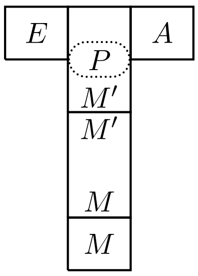
  Anschließende Ausführung des übersetzten Programms auf der Maschine M mit Eingabe E und Ausgabe A:



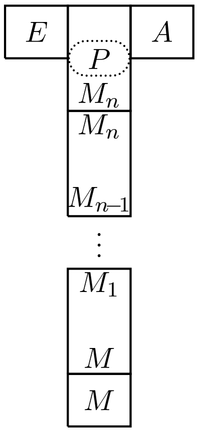
- Mehrstufige Übersetzung eines Programms P:



- Ausführung eines M'-Programms P auf einer M-Maschine mit Hilfe eines (namenlosen) Interpretierers, der in M implementiert ist:

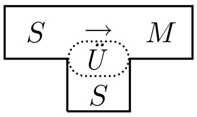


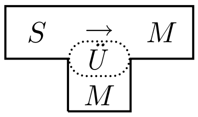
- Mehrstufige Interpretation:



6.2 Bootstrapping zur schrittweisen Übersetzerentwicklung

Gegeben sei eine (höhere) Programmiersprache S, die sich zur Implementierung eines Übersetzers eignet, sowie die Spezifikation einer abstrakten Maschine M, die als Zielsprache für die Übersetzung geeignet ist. Gesucht ist ein Übersetzer $\ddot{U}$ von S nach M, der selbst in S geschrieben ist, sowie eine Version dieses Übersetzers, die in M geschrieben ist.

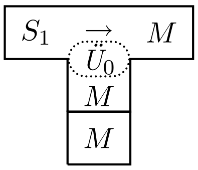




6.2.1 Initialisierung

Zunächst muss die abstrakte Maschine M realisiert werden. Das geschieht durch einen Interpretierer (oder Emulator), dessen Implementierung normalerweise unproblematisch ist.

Der kritischere Teil der Initialisierung besteht darin, einen kleinen Sprachkern S_1 von S zu identifizieren und einen Behelfs-Übersetzer $\ddot{U}_0$ von S_1 nach M zu erstellen, der in M geschrieben ist. Zusammen mit der Implementierung von M steht damit eine erste Übersetzungsmöglichkeit zur Verfügung:

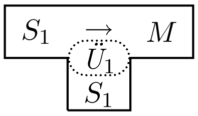


Der Behelfs-Übersetzer wird nur ganz am Anfang benötigt und kann danach weggeworfen werden. Er braucht weder effizient zu sein noch effizienten Code zu erzeugen, er kommt mit einer rudimentären Fehlerbehandlung aus, und er kann in jeder sonstigen Hinsicht minimal sein. Normalerweise wäre es heutzutage auch Purismus, ihn in der maschinennahen Sprache M zu implementieren. Wenn eine andere höhere Programmiersprache zur Verfügung steht, ist es praktischer, ihn damit zu implementieren.

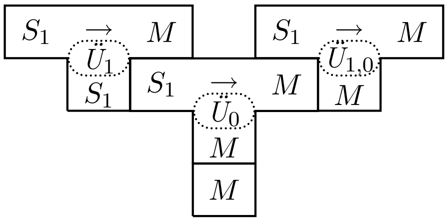
Unter dem Gesichtspunkt der Initialisierung ist es günstig, den Sprachkern S_1 möglichst klein zu wählen, damit der Behelfs-Übersetzer möglichst einfach bleiben kann. Andererseits wird die Programmiersprache S_1 im folgenden Schritt als Werkzeug eingesetzt, das unbequem zu benutzen sein kann, wenn S_1 zu klein gewählt wird.

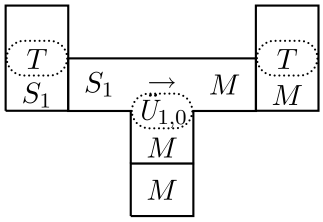
6.2.2 Erster Schritt des Bootstrapping

Mit dem Behelfs-Übersetzer $\ddot{U}_0$ steht eine höhere Programmiersprache, nämlich S_1, zur Verfügung. In dieser Sprache können komplexe Programme geschrieben werden, insbesondere auch Übersetzer. Der erste Schritt des Bootstrapping besteht darin, in der Sprache S_1 einen (völlig neuen und hoffentlich besseren) Übersetzer $\ddot{U}_1$ zu schreiben, der von S_1 nach M übersetzen kann:



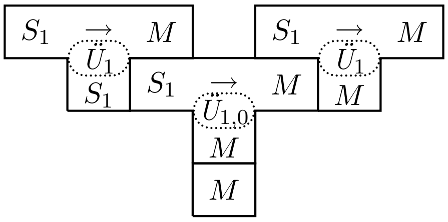
Um den neuen Übersetzer $\ddot{U}_1$ zu testen, kann man ihn mit Hilfe des Behelfs-Übersetzers $\ddot{U}_0$ nach M übersetzen und dann auf Testprogramme T anwenden, deren erzeugter M-Code Aufschluss über die Qualität von $\ddot{U}_1$ gibt:





Das Programm $\ddot{U}_{1,0}$ wurde vom Behelfs-Übersetzer $\ddot{U}_0$ erzeugt und ist deswegen wahrscheinlich nicht so effizient wie möglich. Aber die Qualität von $\ddot{U}_{1,0}$ ist genau die von $\ddot{U}_1$. Es übersetzt S_1-Programme in M-Code von der Qualität und Effizienz, die durch das S_1-Programm $\ddot{U}_1$ implementiert ist.

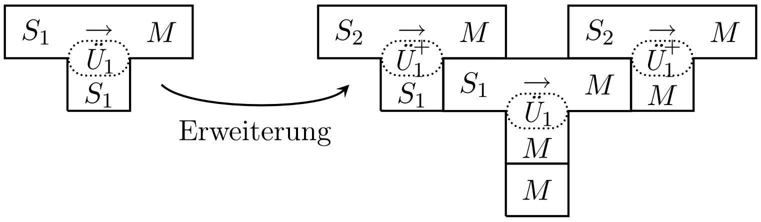
Wenn die Testphase zufriedenstellend abgeschlossen und das Programm $\ddot{U}_{1,0}$ stabil ist, lässt man es statt eines Testprogramms den Übersetzer $\ddot{U}_1$ übersetzen. Da das Programm $\ddot{U}_{1,0}$ selbst durch Übersetzung aus $\ddot{U}_1$ entstanden ist, wird der Übersetzer $\ddot{U}_1$ jetzt sozusagen „mit sich selbst“ übersetzt:



Dadurch erhält man *die* M-Version des Übersetzers $\ddot{U}_1$. Sowohl ihre Laufzeiteffizienz als auch die Qualität der von ihr geleisteten Übersetzung hängen nur noch von der S_1-Version von $\ddot{U}_1$ ab und werden nicht davon beeinflusst, wie gut oder schlecht der Behelfs-Übersetzer $\ddot{U}_0$ war. Der Behelfs-Übersetzer wird von diesem Punkt an nicht mehr benötigt.

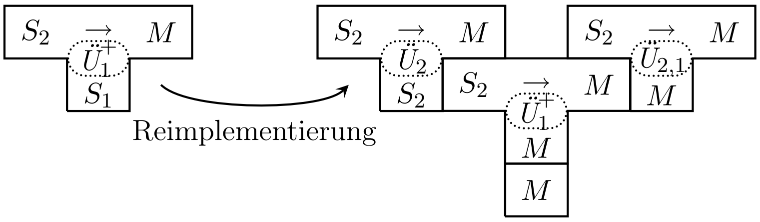
6.2.3 Schrittweise Weiterentwicklung

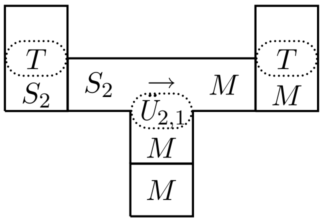
Der Übersetzer $\ddot{U}_1$ liegt nun in einer S_1-Version und einer M-Version vor und kann S_1-Programme nach M übersetzen. Jetzt wird der Sprachkern S_1 von S zu einer größeren Teilsprache S_2 von S erweitert. Dazu erweitert man die S_1-Version von $\ddot{U}_1$ zu einem Übersetzer $\ddot{U}_1^+$, der auch die hinzugefügten Sprachkonstrukte übersetzen kann. Diesen kann man mit der M-Version von $\ddot{U}_1$ übersetzen:



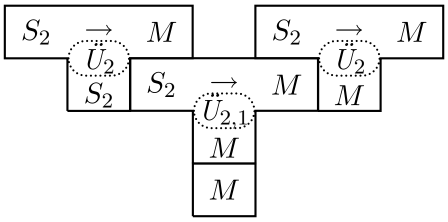
Danach wird der erweiterte Übersetzer $\ddot{U}_1^+$ in der Sprache S_2 reimplementiert als Übersetzer $\ddot{U}_2$. Dabei können die mit S_2 eingeführten neuen Sprachkonstrukte verwendet werden, was möglicherweise zu einer Verbesserung des Übersetzers führt (zum Beispiel wenn er bessere Datenstrukturen verwenden kann, die in der Sprache S_1 noch nicht verfügbar waren).

Um den neuen Übersetzer $\ddot{U}_2$ zu testen, kann man ihn mit Hilfe der M-Version von $\ddot{U}_1^+$ nach M übersetzen und dann auf Testprogramme T anwenden, deren erzeugter M-Code Aufschluss über die Qualität von $\ddot{U}_2$ gibt:





Die Effizienz des Programms $\ddot{U}_{2,1}$ hängt von $\ddot{U}_1^+$ ab, aber es übersetzt S_2-Programme in M-Code von der Qualität und Effizienz, die durch das S_2-Programm $\ddot{U}_2$ implementiert ist. Nach Abschluss der Testphase übersetzt man den Übersetzer $\ddot{U}_2$ mit $\ddot{U}_{2,1}$, also wieder „mit sich selbst“:



Damit hat man eine S_2-Version und eine M-Version des Übersetzers $\ddot{U}_2$, deren Qualität und Effizienz nicht mehr vom Übersetzer $\ddot{U}_1$ abhängen.

Wenn man jeweils den Index 1 durch i und den Index 2 durch $i+1$ ersetzt, erhält man das Schema für beliebig viele weitere Entwicklungsschritte. Diese können sowohl dazu dienen, den Übersetzer $\ddot{U}_i$ auf eine größere Teilsprache von S zu erweitern, als auch dazu, den Übersetzer für die selbe Teilsprache zu verbessern. Die Verbesserung kann zum Beispiel in einem besseren Übersetzungsverfahren bestehen oder darin, dass besserer Code erzeugt wird.

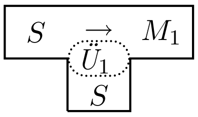
Es muss betont werden, dass am Ende jedes Schritts ein Übersetzer steht, dessen Qualität nicht von seinen Vorgängern (und insbesondere nicht von dem Behelfs-Übersetzer $\ddot{U}_0$) abhängt. Das Bootstrapping kann also durchaus mit einer einfachen oder ungeschickten Implementierung eingeleitet werden. Am Ende der schrittweisen Weiterentwicklung steht trotzdem eine S-Version und eine M-Version des bestmöglichen Übersetzers $\ddot{U}$ für die uneingeschränkte Sprache S.

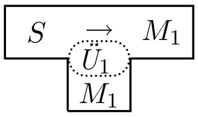
Selbstverständlich ist eine Voraussetzung des Bootstrapping, dass die Programmiersprache S überhaupt zur Implementierung von Übersetzern geeignet ist. Das ist aber bei neueren Programmiersprachen meistens der Fall, und zwar für Sprachen aller Programmierparadigmen.

6.3 Anwendung des Bootstrapping zur Übersetzerportierung

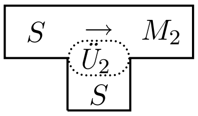
Das Bootstrapping kann auch angewandt werden, um einen Übersetzer von einer Maschine auf eine andere zu portieren. Man spricht dann vom „retargeting".

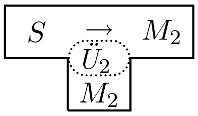
Gegeben seien eine S-Version und eine M_1-Version eines Übersetzers $\ddot{U}_1$, der von S nach M_1 übersetzen kann, sowie eine (abstrakte oder konkrete) Maschine M_2.





Gesucht sind zwei entsprechende Versionen eines Übersetzers von S nach M_2:

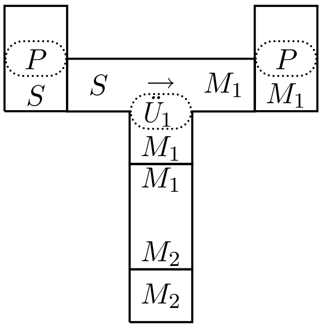


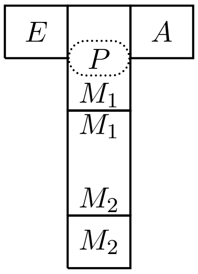


Aufgrund der Komplexität von Übersetzern ist es erstrebendswert, so viel wie möglich aus dem gegebenen Übersetzer $\ddot{U}_1$ zu übernehmen. Dazu eignen sich die Implementierungen der Analysephasen (lexikalische, syntaktische und semantische Analyse) sowie ein Teil der Synthesephasen (wie etwa Transformationen des Zwischencodes), die von der Zielsprache und -maschine unabhängig sind.

6.3.1 Initialisierung

Zunächst muss nur die abstrakte Maschine M_1 auf der gewünschten Zielmaschine M_2 realisiert werden. Das geschieht durch einen Interpretierer (oder Emulator), dessen Implementierung normalerweise unproblematisch ist. Damit ist die Programmiersprache S bereits auf der Zielmaschine M_2 verwendbar:





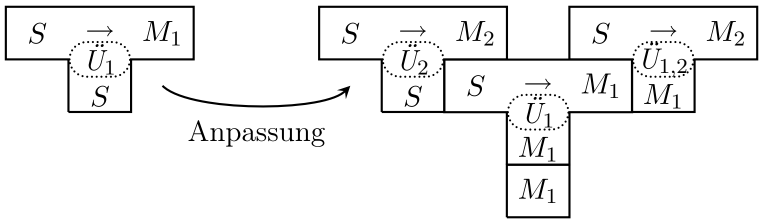
Wenn die Portierung dort erfolgt, wo die Maschine M_1 verfügbar ist, entfällt sogar dieser Initialisierungsschritt.

Wir werden im Folgenden davon ausgehen, dass die Maschine M_1 verfügbar ist, ohne den gegebenenfalls erforderlichen Interpretierer explizit in die Diagramme aufzunehmen.

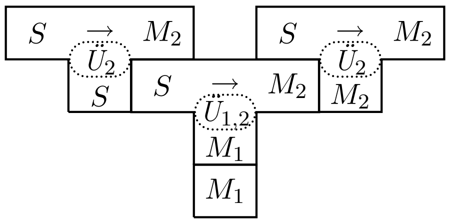
6.3.2 Anpassung des Übersetzers an die neue Zielmaschine

Die S-Version des Übersetzers $\ddot{U}_1$ wird in einen Übersetzer $\ddot{U}_2$ umgeschrieben, der M_2-Code statt M_1-Code erzeugt. Da der Übersetzer $\ddot{U}_1$ (hoffentlich) im Hinblick auf Portierbarkeit strukturiert ist, sind die zu ändernden Programmteile gut vom Rest des Übersetzerprogramms abgekapselt, und da er in der höheren Programmiersprache S geschrieben ist, sind diese Programmteile (hoffentlich) gut verständlich implementiert und leicht zu ändern. Wenn obendrein die Maschinen M_1 und M_2 hinreichend ähnlich sind, zum Beispiel weil sie auf der gleichen Prozessortechnologie beruhen, kann die Anpassung eine sehr einfache Angelegenheit sein.

Damit erhält man die S-Version des Übersetzers $\ddot{U}_2$. Diese kann man dann mit der M_1-Version von $\ddot{U}_1$ übersetzen:



Das Programm $\ddot{U}_{1,2}$ übersetzt S-Programme nach M_2 und kann auf der Maschine M_1 ausgeführt werden, die ja zur Verfügung steht. Also übersetzen wir die S-Version von $\ddot{U}_2$ mit $\ddot{U}_{1,2}$ und damit „mit sich selbst", und erhalten so auch noch die gewünschte M_2-Version des Übersetzers $\ddot{U}_2$:



Die beiden Versionen von $\ddot{U}_2$ benötigen die Maschine M_1 nicht mehr.