Skript zur Compilertechnik

Daniel Scheibler

Michael Henke

Peter Bachmann

Februar/März 2004

Inhaltsverzeichnis

Kapitel 1

Ein- und Überblick

1.1 Zur Motivation

Ein **Compiler** ist ein *Programm*, das

Programme aus einer *Quellsprache* in **bedeutungsäquivalente** Programme einer *Zielsprache* transformiert.

Die *Geschichte* der Compilertechnik ist eng verbunden mit der Geschichte der Programmiersprachen:

- Zunächst wurde in die höheren Programmiersprachen (etwas unbescheiden mit AUTOCODEs bezeichnet) als Konstrukte aufgenommen, was nach dem aktuellen Stand der Erkenntnis in die Maschinensprachen transformierbar war. So entstand 1955 die Sprache FORTRAN, nachdem die dazu erforderliche Compilertechnik geklärt war (FORTRAN=FORmula TRANslator).
- Mit zunehmenden Erkenntnisfortschritt und der praktischen sowie theoretischen Fundierung der Compilertechnik wurden oft die Programmiersprachen entwickelt, ohne sich um die erforderlichen Compiler Gedanken zu machen (ALGOL 68, ADA, ...).
- Die enge Wechselwirkung zwischen Aufbau und Wirkungsweise einer Programmiersprache und der zugehörigen Compiler legt es trotzdem nahe, die Entwicklung einer Sprache und der Compiler zu verflechten (Der Erfolg von PASCAL hat darin eine wesentliche Begründung).

Kenntnisse und Fertigkeiten zur Compilertechnik sind für den Informatiker *nützlich*, weil

- es das Verständnis für interne Abläufe bei Programmiersprachen fördert, was zu einem effizienteren Programmierstil führen kann;
- Algorithmen der Compilertechnik auch in anderen Gebieten (z.B. Datenanalyse) anwendbar sind;
- Spezialsprachen in vielen Einzelanwendungen Einsatz finden und dort Compiler erfordern;
- höhere Universalsprachen ständig Veränderungen erfahren (C $\rightarrow$ C++ $\rightarrow$ C#) und mancher Informatiker vielleicht an solchen Entwicklungen beteiligt sein wird.

1.2 Sprachen, deren Syntax und Semantik

1.2.1 Semiotik

Sprache = Instrument zur Kommunikation und zum Denken.

Die Grundelemente einer Sprache sind die *Zeichen*, die auf unterschiedliche Weise (optisch, akustisch,...) dargestellt und wahrgenommen werden können.

Semiotik = Lehre von den Zeichen mit den Aspekten

- **Syntax** = Lehre vom Satzbau,
- **Semantik** = Lehre von der Bedeutung,
- **Pragmatik** = Lehre vom sprachlichen Handeln.

Wir betrachten speziell die Syntax und Semantik von Programmiersprachen und ignorieren deren Pragmatik. Die vorbereitende **Formalisierung** betreiben wir nur im notwendigen Umfang und Abstraktionsgrad. Für mehr Details siehe auch GTI-Skript, Abschnitt 1.6.

Grundlegende Mengen sind:

- eine höchstens abzählbare Menge Σ von Zeichen, auch *Alphabet* genannt, und
- die Menge Σ^* von *Zeichenketten* über einem Alphabet Σ, wobei jedes Element $\alpha \in \Sigma^*$ eine endliche Folge von Zeichen aus dem Alphabet ist, d.h.

$$\alpha = \alpha(0) \dots \alpha(n-1) \quad \text{mit } n \in \mathbb{N} \text{ und } \alpha(i) \in \Sigma \text{ für } i \in \{0, \dots, n-1\}.$$

  Es heißt dann n die *Länge* von α. Die einzige Zeichenkette der Länge 0 wird mit ε bezeichnet.
  Man beachte weiter, daß Σ^* einelementig oder unendlich ist und wenn Σ abzählbar, dann auch Σ^*.
  Statt von Zeichenketten werden wir auch von *Worten* über dem Alphabet Σ sprechen.
  Es sei $\Sigma^+ := \Sigma^* - \{\varepsilon\}$, d.h. jedes Wort aus Σ^+ ist nicht leer!

Benötigte **Operationen** sind:

$\circ : \Sigma^* \times \Sigma^* \to \Sigma^*$, *Verkettung* genannt, wobei gilt:

$$\alpha \circ \beta := \alpha\beta,$$

d.h. die Verkettung zweier Zeichenketten entsteht einfach durch Hintereinandersetzen, wodurch man berechtigt ist, das Operationszeichen wegzulassen! Man beachte, daß die Verkettung assoziativ ist und daß gilt:

$$\varepsilon\alpha = \alpha\varepsilon = \alpha.$$

$|.| : \Sigma^* \to \mathbb{N}$, die die *Länge* einer Zeichenkette bestimmt, d.h.

$$|\varepsilon| = 0, \qquad |\alpha\beta| = |\alpha| + |\beta|.$$

$\leftarrow : \Sigma^* \to \Sigma^*$, *Invertierung* genannt, wobei gilt:

$$\overleftarrow{\varepsilon} = \varepsilon, \ \overleftarrow{a\alpha} = \overleftarrow{\alpha}\, a.$$

- $\bullet : \wp(\Sigma^*) \times \wp(\Sigma^*) \to \wp(\Sigma^*)$, das *Komplexprodukt* von Wortmengen, definiert durch

$$A \bullet B := \{\alpha\beta \mid \alpha \in A \wedge \beta \in B\}.$$

Über einer Wortmenge Σ^* ist die (partielle) *Präfixordnung* $\sqsubseteq$ definiert durch

$$\alpha \sqsubseteq \gamma \;:\Leftrightarrow\; \exists\beta : \alpha\beta = \gamma.$$

Es sei

$$\sqsubset := \sqsubseteq - = .$$

Eine **formale Sprache** ist eine Menge $L \subseteq \Sigma^*$. Wir nennen die Menge L deswegen *formal*, weil wir uns zunächst nur dafür interessieren, ob ein Wort α aus Σ^* auch aus der Menge L ist. Die Syntax ist (abstrakt gesehen) die charakteristische Funktion

$$syn : \Sigma^* \to \{0, 1\}$$

von L, definiert durch:

$$syn(\alpha) = 1 \quad :\Leftrightarrow \quad \alpha \in L$$

Letztlich interessieren aber nicht nur die zugelassenen Texte (Worte) einer Sprache, sondern deren **Bedeutungen**. Mit B sei die Menge der Bedeutungen bezeichnet, dann ist die Semantik eine Funktion, die jedem Wort der Sprache seine Bedeutung zuordnet, d.h.

$$sem : L \to B.$$

1.2.2 Kontextfreie Grammatiken

In der Chomsky-Hierarchie der Grammatiken (Siehe GTI-Skript, Abschnitt 5) greift die Compilertechnik auf den Typ 2, auch kontextfreie Grammatiken genannt, zurück (Siehe GTI-Skript, Abschnitt 5.3).

Zur Darstellung solcher Grammatiken wird die *Backus-Naur-Form* in verschiedenen Ausprägungen genutzt. Wir wollen hier folgende Konvention befolgen:

- *Metasymbole* werden in kursive Zeichenketten, bestehend aus Buchstaben und der Unterstreichung gesetzt,
- *Grundsymbole* werden in Gänsefüßchen (`"`) geklammert,
- zur Trennung von linker und rechter Seite einer Regel wird die Zeichenkombination `::=` benutzt,
- Regeln mit der gleichen linken Seite werden zusammengefaßt, wobei die verschiedenen rechten Seiten, Alternative genannt, durch einen senkrechten Strich (|) getrennt werden.

Beispiel PRILAN

Konventionen:

- Mit Σ bezeichnen wir die Menge aller Grundsymbole.
- Mit M bezeichnen wir die Menge aller Metasymbole.
- Mit $L(A) := \{\alpha \mid \alpha \in \Sigma^* \wedge A \to^* \alpha\}$ bezeichnen wir die Menge der aus dem Metasymbol A ableitbaren Wörter aus Σ^*.
- Mit $|[A]|$ bezeichnen wir die Anzahl der Alternativen zum Metasymbol A.

- Mit $[A, i]$ bezeichnen wir die $i-$te Alternative zum Metasymbol A, es ist dann $|[A, i]|$ deren Länge.
- Mit $[A, i, j]$ bezeichnen wir das $j-$te Element in der $i-$ten Alternative zum Metasymbol A.

Zur Einsparung von Regeln werden in der BNF weiterhin Erweiterungen genutzt:

Statt der beiden Regeln

$$\begin{array}{lll} A & ::= & \alpha_1 \mid \ldots \mid \gamma_1 B \gamma_2 \mid \ldots \mid \alpha_m \\ B & ::= & \beta_1 \mid \ldots \mid \beta_n \end{array}$$

führen wir unter Nutzung der geschweiften Klammern {} ein:

$$A \quad ::= \quad \alpha_1 \mid \ldots \mid \gamma_1 \{\beta_1 \mid \ldots \mid \beta_n\} \, \gamma_2 \mid \ldots \mid \alpha_m$$

Statt der beiden Regeln

$$\begin{array}{lll} A & ::= & \alpha_1 \mid \ldots \mid \gamma_1 B \gamma_2 \mid \ldots \mid \alpha_m \\ B & ::= & \varepsilon \mid \beta_1 \mid \ldots \mid \beta_n \end{array}$$

führen wir unter Nutzung der eckigen Klammern [ ] ein:

$$A \quad ::= \quad \alpha_1 \mid \ldots \mid \gamma_1 \, [\beta_1 \mid \ldots \mid \beta_n] \, \gamma_2 \mid \ldots \mid \alpha_m$$

Statt der beiden Regeln

$$\begin{array}{lll} A & ::= & \alpha_1 \mid \ldots \mid \gamma_1 B \gamma_2 \mid \ldots \mid \alpha_m \\ B & ::= & C \mid C \; B \end{array}$$

führen wir ein:

$$A \quad ::= \quad \alpha_1 \mid \ldots \mid \gamma_1 C^+ \gamma_2 \mid \ldots \mid \alpha_m$$

Schließlich erlauben wir die Konstruktion

$$[\, \beta_1 \mid \ldots \mid \beta_n]^+$$

zu ersetzen durch

$$[\, \beta_1 \mid \ldots \mid \beta_n]^*$$

oder durch

$$\{\, \beta_1 \mid \ldots \mid \beta_n\}^*$$

Beispiel BALAN

Zu **beachten** ist aber, daß jede der Alternativen mit der entsprechenden linken Seite eine separate Regel beschreibt!

Interessant ist, daß man unter Nutzung dieser Erweiterungen mit nur einem Metasymbol genau die *regulären Wortmengen* (Siehe GTI-Skript, Abschnitt 2.3) beschreiben kann.

Wichtig für die Compilertechnik ist, daß zu jeder Ableitung eines Wortes ein *Syntaxbaum* σ (Siehe wieder GTI-Skript, Definition 5.16, Satz 5.20) gehört.

Beispiel für Syntaxbaum.

Wie wir später sehen werden, ist im Syntaxbaum die Angabe der Grundsymbole überflüssig, wenn man in jedem Knoten zusätzlich vermerkt, welche Regel, d.h. welche Alternative, bei der Ableitung genutzt wurde. Einen solchen Syntaxbaum nennen wir *abstrakt*.

Definition 1.2.1 *Ein abstrakter Syntaxbaum zu einer kontextfreien Grammatik ist ein markierter Baum $\sigma : B \to M \times I\!N$, wobei für jede Adresse $\alpha \in B$ mit $\sigma(\alpha) = (A, n)$ gilt:*

- *n ist die Nummer einer Alternativen des Metasymbols A und*
- *wenn $[A, n] = \gamma_0 A_1 \gamma_1 A_2 \dots A_k \gamma_k$, wobei $A_i \in M, \gamma_i \in \Sigma^*$, so hat der Knoten α genau k Söhne mit den Markierungen $\sigma(\alpha.i) = (A_i, n_i)$.*

Beispiel für abstrakten Syntaxbaum.

Wir betrachten zwei *Linearisierungen* von abstrakten Syntaxbäumen, das heißt die Transformation in eine Folge

$$\sigma(\alpha_0) \dots \sigma(\alpha_n)$$

von Knotenmarkierungen:

- Für die Präfixlinearisierung $pre(\sigma) = \sigma(\alpha_0) \dots \sigma(\alpha_n)$ gilt $\alpha_i \sqsubset_{lex} \alpha_{i+1}$ und
- für die Postfixlinearisierung $post(\sigma) = \sigma(\alpha_0) \dots \sigma(\alpha_n)$ gilt $\alpha_i \sqsubset_{ilex} \alpha_{i+1}$.

Hierbei ist $\sqsubset_{lex}$ die lexikographische Ordnung, definiert durch

$$\alpha \sqsubset_{lex} \beta \;:\Leftrightarrow\; \alpha \sqsubset \beta \text{ oder } \alpha = \gamma.m.\alpha' \text{ und } \beta = \gamma.n.\beta' \text{ und } m < n$$

und $\sqsubset_{ilex}$ ist die inverse lexikographische Ordnung, definiert durch

$$\alpha \sqsubset_{ilex} \beta \;:\Leftrightarrow\; \beta \sqsubset \alpha \text{ oder } \alpha = \gamma.m.\alpha' \text{ und } \beta = \gamma.n.\beta' \text{ und } m < n.$$

Beispiel für Linearisierungen.

Man überlegt sich leicht, daß

$$pre(\sigma) = \sigma(\varepsilon) pre(\sigma/0) \dots pre(\sigma/n_\varepsilon), \quad post(\sigma) = post(\sigma/0) \dots post(\sigma/n_\varepsilon)\sigma(\varepsilon),$$

wobei n_ε die Anzahl der Söhne des Wurzelknotens ist.

Als **Aufgabe** verbleibt die Entwicklung von Algorithmen, die aus den Linearisierungen den abstrakten Syntaxbaum rekonstruieren.

1.2.3 Sprachtransformation aus algebraischer Sicht

Eine (heterogene) **Algebra** besteht aus

- einem *System von Mengen* (Trägermengen genannt) und
- einer Menge von *Operationen* (Funktionen) über den Trägermengen.

Die **Signatur** (S, F) einer Algebra besteht aus

- einer Menge S (von Sorten) und
- einer paarweisen disjunkten Mengenfamilie $F := \{F_\omega \mid \omega \in S^+\}$ (von Funktionssymbolen), d.h. aus $F_\omega = F_{\omega'}$ folgt $\omega = \omega'$.

Für $f \in F_{s_1 \ldots s_n s}$ heißt

- $s \in S$ die *Sorte* und
- $s_1 \ldots s_n \in S^*$ die *Arität* von f.

Eine **Interpretation** I macht eine Signatur zur Algebra, indem

- jeder Sorte $s \in S$ eine Menge s^I und
- jedem Funktionssymbol $f \in F_{s_1 \ldots s_n s}$ eine Operation $f^I : s_1^I \times \ldots \times s_n^I \to s^I$

zugeordnet wird.

Zu jeder kontextfreien Grammatik G konstruieren wir uns eine Algebra $\underline{G}$, indem

- eine Signatur erzeugt wird mit
  - $S := M$, d.h. die Metasymbole der Grammatik sind die Sorten, und
  - $F_{A_1 \ldots A_n A} := \{(A, i) \mid A \in M \wedge i$ ist die Nummer einer Alternativen zu $A\}$ falls $[A, i] = \gamma_0 A_1 \gamma_1 A_2 \ldots A_n \gamma_n$.
- und eine Interpretation G festlegt, dass
  - $A^G = L(A)$ für jedes Metasymbol $A \in M$ sowie
  - $(A, i)^G : A_1^G \times \ldots \times A_n^G \to A^G$ mit $(A, i)^G(\beta_1, \ldots, \beta_n) = \gamma_0 \beta_1 \gamma_2 \ldots \gamma_{n-1} \beta_n \gamma_n$.

Beispiel der Signatur und Algebra von PRILAN.

Ein **Homomorphismus** $\varphi : \underline{A} \to \underline{B}$ ist eine Abbildung zwischen den Algebren $\underline{A}$ und $\underline{B}$ **gleicher Signatur**, so dass

- zu jeder Sorte s eine Funktion $\varphi_s : s^A \to s^B$ existiert und
- für jedes Operationssymbol $f \in F_{s_1 \ldots s_n s}$ gilt:

$$\varphi_s(f^A(a_1, \ldots, a_n)) = f^B(\varphi_{s_1}(a_1), \ldots, \varphi_{s_n}(a_n)).$$

Beispiel für modifizierte PRILAN-Algebra.

Beispiel für Homomorphismus.

Achtung: Zwischen Algebren $\underline{A}$ und $\underline{B}$ zur gleichen Signatur kann es

- mehrere Homomorphismen oder auch
- keinen Homomorphismus

geben.

Beispiel für Algebren, zwischen denen keine Homomorphismen existieren.

Die **Termalgebra** $\underline{T}$ zur Signatur (S, F) wird durch die Interpretation T erzeugt, indem

- für $s \in S$ die Menge s^T die kleinste Menge ist, die der Gleichung

$$s^T = F_s \cup \{t_1 \dots t_n f \mid f \in F_{s_1 \dots s_n s} \wedge t_1 \in s_1^T \dots t_n \in s_n^T\}$$

  genügt und
- für $f \in F_{s_1 \dots s_n s}$ gilt: $f^T(t_1, \dots, t_n) = t_1 \dots t_n f$.

Beispiel für Terme zur PRILAN-Signatur.

Die Termalgebra zur Signatur (S, F) hat die Eigenschaft, dass zu jeder Algebra $\underline{A}$ der gleichen Signatur genau ein Homomorphismus $\underline{\varphi} : \underline{T} \to \underline{A}$ gehört.
Dieser Homomorphismus wird für $f \in F_{s_1 \dots s_n s}, t_1 \in s_1^T, \dots, t_n \in s_n^T$ eindeutig beschrieben durch:

$$\varphi_s(t_1 \dots t_n f) := f^A(\varphi_{s_1}(t_1), \dots, \varphi_{s_n}(t_n)).$$

In einem solchen Homomorphismus $\underline{\varphi} : \underline{T} \to \underline{A}$ repräsentiert

- ein Term t eine *Rechenvorschrift* und
- das homomorphe Bild $\varphi(t)$ das *Ergebnis der Rechnung.*

Beispiel für homomorphe Bilder von Termen.

Wir bezeichnen als

- **abstrakte Syntax** die Termalgebra $\underline{T}$ zur Signatur einer kontextfreien Grammatik G und als
- **konkrete Syntax** den eindeutig bestimmten Homomorphismus $syn : \underline{T} \to \underline{G}$.

Wir stellen folgenden **wichtigen** Sachverhalt fest:

Die Postfixlinearisierung eines abstrakten Syntaxbaumes ist ein Term, dessen homomorphes Bild ein Satz der Sprache ist, d.h.

$$syn(post(\sigma)) \in L(S),$$

falls S das Satzsymbol der Grammatik ist.

Beispiel für Term als Postfixlinearisierung des abstrakten Syntaxbaumes.

Zur abstrakten Beschreibung der durch einen Compiler zu realisierenden Transformation gehen wir etwas idealisierend davon aus, daß sowohl für die Quellsprache als auch für eine Zielsprache Grammatiken Q bzw. Z existieren, die die gleiche Signatur (S, F) erzeugen. Die konkrete Syntax dazu bezeichnen wir mit $syn_Q : \underline{T} \to \underline{Q}$ und $syn_Z : \underline{T} \to \underline{Z}$, wobei $\underline{T}$ die Termalgebra zur Signatur (S, F) ist.

Die durch einen Compiler vorzunehmende Transformation ist nun ein Homomorphismus $compile : \underline{Q} \to \underline{Z}$, für den gilt $syn_Q \circ compile = syn_Z$ bzw. $compile = syn_Q^{-1} \circ syn_Z$.

Um der Forderung zu genügen, daß das zum Quellprogramm q transformierte Programm $compile(q)$ bedeutungsäquivalent zu q ist, bauen wir zur Signatur der Grammatiken (S, F) auch eine *Bedeutungsalgebra* $\underline{B}$ auf, indem wir

- als Trägermengen *Bedeutungen* (was das auch sein mag) und
- als Operationen Funktionen über den Bedeutungen

zuordnen.

Beispiel für Bedeutungsalgebra zu PRILAN

Jetzt existiert wieder ein eindeutig bestimmter Homomorphismus $sem : \underline{T} \to \underline{B}$ und Homomorphismen $sem_Q : \underline{Q} \to \underline{B}$ sowie $sem_Z : \underline{Z} \to \underline{B}$. Zur Absicherung der äquivalenten Bedeutung von $compile(q)$ und q fordern wir:

$$syn_Q \circ sem_Q = sem = syn_Z \circ sem_Z.$$

Unter Bezugnahme auf das folgende Diagramm kann man, algebraisch vornehm, dies dadurch ausdrücken, daß man die *Kommutativität des Diagramms* fordert.

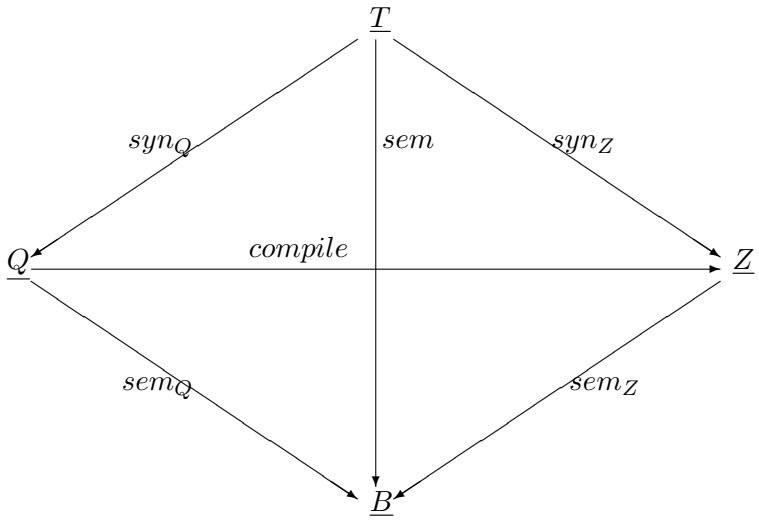


Diagramm für algebraisch idealisierte Sprachtransformation

1.3 Spezifika von Programmiersprachen

1.3.1 Syntaktische Spezifika

Problem: Programmiersprachen sind (leider) kontextsensitiv, aber im allgemeinen nicht kontextfrei!

(Siehe wieder GTI-Skript, Abschnitt 5.2)

Trotzdem wird zur Beschreibung der Syntax eine kontextfreie Grammatik G eingesetzt. Als **Konsequenz** gehören zur durch diese Grammatik definierten Sprache PS mehr Worte als zur Programmiersprache.

Gelöst wird dieses Problem dadurch, dass eine *Kontextalgebra* $\underline{C}$ mit der aus G abgeleiteten Signatur (S, F) aufgebaut wird. Wenn $Programm$ das Satzsymbol von G ist, dann wird $Programm^C := \{0, 1\}$ gesetzt.

Für den eindeutig bestimmten Homomorphismus $context : \underline{T} \to \underline{C}$ gilt dann, dass für jeden Term $t \in Programm^T$ der Sorte $Programm$ das Bild aus $\{0, 1\}$ ist, also $context_{Programm}(t) \in \{0, 1\}$.

Unter der Annahme, dass die Grammatik G *eindeutig* ist

(Siehe wieder GTI-Skript, Definition 7.5)

können wir nun die Menge PS aller syntaktisch korrekten Texte einer Programmiersprache definieren als

$$PS := \{\alpha \mid \exists t \in Programm^T : \ syn_Q(t) = \alpha \wedge context_{Program}(t) = 1\}$$

bzw. als

$$PS := \{\alpha \mid context_{Program}(syn_Q^{-1}(alpha)) = 1\}.$$

Beispiel für Kontextalgebra zu PRILAN.

Hinweis: In der Literatur wird der Homomorphismus

- *context* oft als *statische Semantik* und
- *sem* als *dynamische Semantik* bezeichnet.

Aus hauptsächlich **technologischen Gründen** und zur **besseren Übersichtlichkeit** wird die Struktur der Sprache **aufgeteilt** in

- die **lexikalische Struktur**:
  Das bedeutet, daß der Text in eine Folge von Teiltexten, Morpheme genannt, zerlegt wird. Jedes Morphem muß Element einer regulären Sprache sein
  (Siehe wieder GTI-Skript, Abschnitt 2.3 bzw. Abschnitt 5.4).
  Aber: nicht alle regulären Teiltexte einer Programmiersprache werden als Morpheme betrachtet und abgetrennt, sondern nur **kleinste bedeutungstragende Einheiten** (Für die Details dazu siehe die Aufteilung in Morphemklassen).
- die **kontextfreie Struktur** über der Morphemmenge.

Beispiel für Morpheme bei BALAN.

Konsequenz dieses Vorgehens ist, daß

- Morpheme in der kontextfreien Struktur (Grammatik) als Grundsymbole aufgefaßt werden, der Zugriff auf den konkreten Teiltext über Hilfsfunktionen erfolgt.
- syn_Q^{-1} unterteilt ist in
  - den **Scanner** für die Analyse der lexikalischen Struktur vornimmt und
  - den **Parser** für die Analyse der kontextfreien Struktur.
- Der abstrakte Syntaxbaum bezieht sich auf die kontextfreie Struktur, wobei Blätter auch mit Morphembezeichnern und einen Verweis auf den zugehörigen konkreten Teiltext markiert sein können.

1.3.2 Semantische Spezifika

Bedeutungen bei Programmiersprachen werden modelliert durch

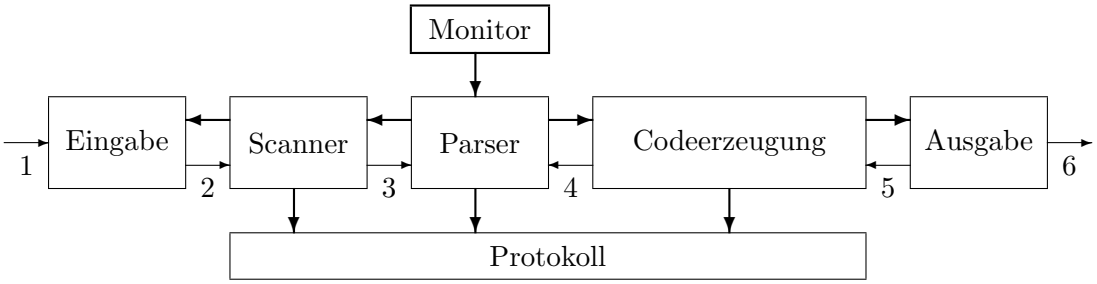
- **formale Automaten** (z.B. ASM) bei **imperativen** Programmiersprachen,
- **Funktionen** bei **funktionalen** Programmiersprachen,
- **Relationen/Prädikate** bei **logischen** Programmiersprachen.

1.4 Implementierung von Compilern

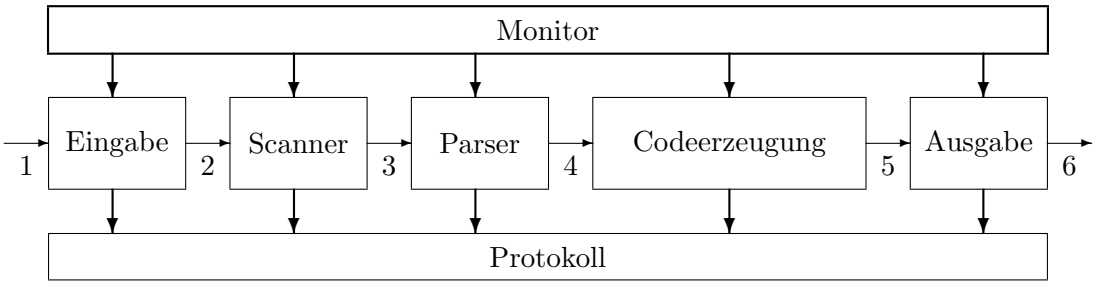
1.4.1 Die modulare Struktur

Man unterscheidet:

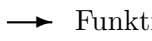
- **Einpaß-Compiler**, bei denen der Zielcode in einem Schritt erzeugt wird und Zwischeninformationen scheibchenweise auf- und abgebaut werden. Der **Monitor** hat hier nur die Aufgabe, Initialisierungen vorzunehmen und dann den **Parser** zu aktivieren, der dann als Master die ihm unterstehenden Module aktiviert, wobei diese wiederrum auf andere Slaves zugreifen.
  Schema:



- **Mehrpaß-Compiler**, bei denen der Zielcode in mehreren Schritten aufgebaut wird,. In jedem Schritt werden Zwischeninformationen komplett über das gesamte Programm aufgebaut und im nächsten Schritt wieder verwertet. Der Monitor hat hier die Funktion eines Masters und ist für die Steuerung des Gesamtablaufes verantwortlich.
  Schema:



Legende:



1	Quellcode	4	Syntaxbaumelemente
2	Zeichen	5	Zwischencode - Fragmente
3	Morpheme	6	Zielcode, Zielprogramm

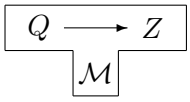
Man bezeichnet

- als **frontend** die Teile Eingabe, Scanner und Parser und
- als **backend** die Teile Codeerzeugung und Ausgabe, wobei die Codeerzeugung auch noch weiter aufgeteilt werden kann.

1.4.2 Implementierungstechnologien

Bootstrapping ist eine Methode, um Compiler mittels Compilern leichter zu erzeugen bzw. auf andere Computer und Sprachen zu übertragen.

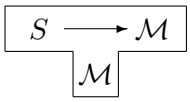
Schema für einen Compiler:



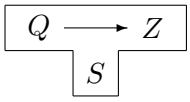
mit

- Q : Quellsprache
- Z : Zielsprache
- $\mathcal{M}$: Sprache in der, der Compiler geschrieben ist
- S : Hilfssprache

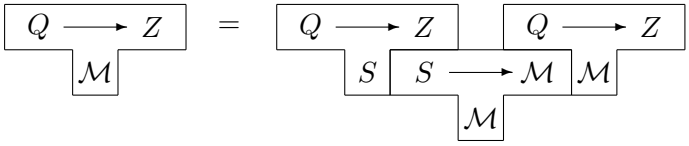
In der 1. Stufe erzeugt man von Hand:



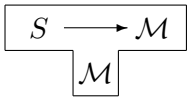
In der 2. Stufe erzeugt man von Hand:



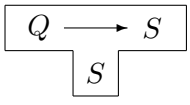
In der 3. Stufe erzeugt man:



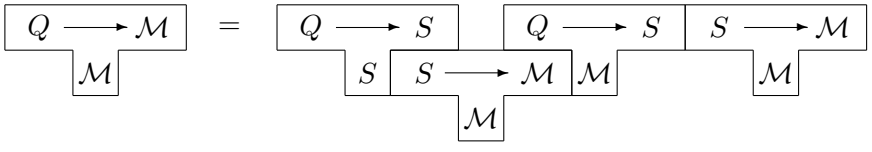
Verwendung von S als Zwischensprache:
In der 1. Stufe erzeugt man von Hand:



In der 2. Stufe erzeugt man von Hand:

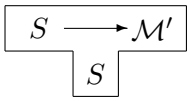


In der 3. Stufe erzeugt man:

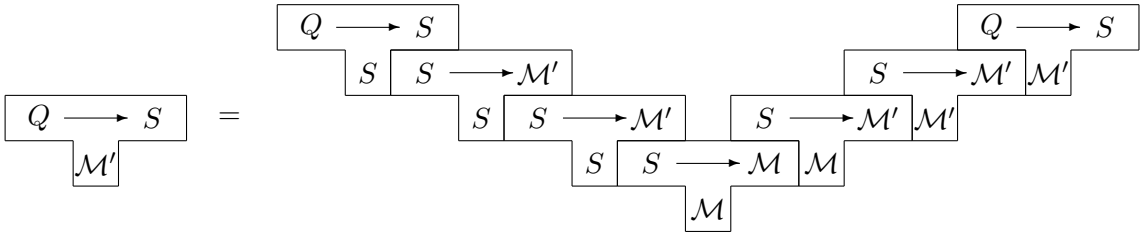


Übertragung auf einen weiteren Computer mit der Maschinensprache $\mathcal{M}'$

In der 1. Stufe erzeugt man von Hand:



In der 2. Stufe erzeugt man:



Compiler-Generatoren erzeugen auf der Grundlage einer möglichst deklarativen Beschreibung von Syntax und Semantik - mehr oder weniger automatisch und vollständig - die entsprechenden Teile des Compilers.
Für die Erzeugung des frontends werden oft die Werkzeuge **lex** und **yacc** benutzt. Eine gute Einführung liefert A GUIDE TO LEX & YACC.

Entwicklungsumgebungen integrieren Compiler und ergänzen diese um

- Editoren,
- Browser,
- Debugger,
- Ressourcen-Assistenten,
- Projekt-Verwaltung,
- on-line Hilfen,
- ... .

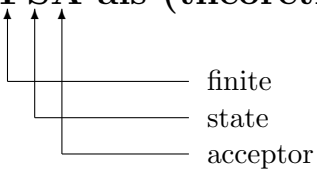
Kapitel 2

Scanning - lexikalische Analyse

Aufgabe ist die Zerlegung des Quelltextes in eine Folge von Morphemen.

Achtung: In der Literatur wird statt von Morphemen auch von **Token** gesprochen!

2.1 Der FSA als (theoretisches) Modell



Für die ausführliche Beschreibung von endlichen Akzeptoren:
siehe wieder das GTI-Skript, Abschnitt 2.2.

Wir betrachten hier eine spezielle Variante, definiert als ein 5-Tupel:

$$FSA = (Z, \Sigma, z_a, F, \delta)$$

mit

- Z - eine **endliche Zustandsmenge**
- Σ - eine höchstens abzählbare Menge, das **Eingabealphabet** (Menge der Grundsymbole),
- $z_a \in Z$ - ein **Anfangszustand**,
- $F \subseteq Z$ - eine Menge von **Endzuständen** und
- $\delta : Z \times \Sigma \to Z$ - die **Überführungsfunktion**.

Die **Arbeitsweise** ist durch den *taktweisen Übergang* zwischen *Situationen*, in denen sich der FSA befinden kann, beschrieben.

Situation: $(z, \alpha) \in Z \times \Sigma^*$

wobei

- z den aktuellen Zustand, in dem sich der FSA befindet und
- α das akuelle Wort an der Eingabe (hier auch Eingabetext genannt)

beschreibt.

Takt: $(z, a\alpha) \mapsto (z', \alpha)$ falls $a \in \Sigma \wedge z' = \delta(z, a)$

Bemerkung: Wir betrachten hier nur den deterministischen und totalen Fall, d.h.

- $\delta(z, a)$ hat als Ergebnis genau einen *Folgezustand* und
- $\delta(z, a)$ ist immer definiert.

Die durch einen FSA **akzeptierte Sprache** $L(FSA)$ ist dann beschrieben als:

$$L(FSA) := \{\alpha \mid (z_a, \alpha) \mapsto^* (z, \varepsilon) \wedge z \in F\}.$$

Es ist hier $\mapsto^*$ der reflexive und transitive Abschluß der *Übergangsrelation*

$$\mapsto \subseteq (Z \times \Sigma^*) \times (Z \times \Sigma^*).$$

2.2 Morphemarten und Zustandsarten

Die Festlegung der zu behandelnden **Morphemklassen** ist nicht starr und hängt unter anderem ab von

- der Struktur der Quellsprache,
- der gewählten Technik für den Parser,
- der gewählten Technik für die Codeerzeugung.

Im allgemeinen unterscheidet man zwischen den folgenden Morphemklassen:

- Bezeichner,
- Zahlenliterale,
- Zeichenkettenliterale,
- Schlüsselwörter,
- Operatoren,
- Trenner (zu beachten ist, daß typgraphische Trenner wie Leerzeichen Tabulator, Newline,... nicht als Morpheme behandelt werden!),
- Kommentare,
- ...

Damit ergeben sich für den Scanner folgende **Aufgaben**:

- Zerlegung des Quelltextes in eine Folge von Morphemen,
- Aufbereitung der einzelnen Morpheme, d.h.
  - Zuordnung eines Codes in Abhängigkeit von der Morphemklasse, z.B.:
    - alle Bezeichner erhalten den gleichen Code,
    - alle Zahlenliterale erhalten den gleichen Code,
    - alle Zeichenkettenliterale erhalten den gleichen Code,
    - jedes Schlüsselwort erhält einen separaten Code,
    - jeder Operator erhält einen separaten Code,
    - jeder Trenner erhält einen separaten Code.
  - Zuordnung eines *Wertes* zu dem Morphem bei gewissen Morphemklassen, z.B.:
    - einen Index des Eintrages in eine Namensliste bei Bezeichnern (der Aufbau einer Namensliste für Bezeichner kann aber auch vom Parser vorgenommen werden, dann wird als Wert die Zeichenkette des Morphems zugeordnet),
    - die interne binäre Repräsentation bei Zahlenliteralen,
    - die interne binäre Repräsentation bei Zeichenkettenliteralen.
- Zuordnung der Positionsinformation im Quelltext.
- Kommentarbehandlung, die im allgemeinen im Überlesen besteht. Allerdings muß eine Spezialbehandlung dann einsetzen, wenn (was ich verachte!) Kommentare für Compilerdirektiven mißbraucht werden.

Man kann **zwei Arten von Morphemen** unterscheiden:

- Morpheme mit **implizitem Abschluß**, bei denen der Abschluß dadurch erkannt wird, daß ein Zeichen auftritt, was nicht zum Morphem gehört. Ein solches Zeichen kann
  - zum nächsten Morphem gehören oder
  - ein typographischer Trenner (Leerzeichen, Tabulator, Newline,..) sein.
- Morpheme mit **explizitem Abschluß**, bei denen der Abschluß durch ein spezielles Zeichen (oder Zeichenkombination), das zum Morphem gehört, erkannt wird.

Als Konsequenz wird folgende **Modifikation des FSA** vorgenommen:

1. Statt für das Erkennen der einzelnen Morpheme separate FSA's zu benutzen wird ein einzelner FSA konzipiert, bei dem der Übergang vom Endzustand in den Anfangszustand automatisch ohne das Akzeptieren einer Eingabe erfolgt.
   Für diesen Übergang wird ein separater Takt durchgeführt, der es erlaubt, spezielle Aktionen zur Aufbereitung der Morpheme auszuführen.
2. Die Menge F der Endzustände wird unterteilt in $F = F_i \cup F_e$ wobei
   - F_i - Endzustände mit implizitem Abschluß sind, bei deren Eintreffen keine Eingabezeichen akzeptiert wird und
   - F_e - Endzustände mit explizitem Abschluß sind, bei deren Eintreffen eine Eingabezeichen akzeptiert wird.

3. Zur Reduktion der Tabelle zur Beschreibung der Überführungsfunktion δ werden Grundsymbole zu Klassen zusammengesfaßt, auf die der FSA immer gleichartig reagiert.

   Beispiel für Klassenbildung bei Grundsymbolen

Damit ergeben sich folgende **Taktarten**:

- $(z, \alpha) \mapsto (z_a, \alpha)$ falls $z \in F$,
- $(z, a\alpha) \mapsto (z', a\alpha)$ falls $\delta(z, a) = z' \in F_i$,
- $(z, a\alpha) \mapsto (z', \alpha)$ sonst.

2.3 Notwendige Dienste

Wie im Abschnitt 1.4.1 bereits angegeben, ist dem Scanner noch eine Eingabe vorgeschaltet. Zur Beschreibung der erforderlichen Dienste beider Module werden Namen vergeben, auf die später zurückgegriffen wird. Natürlich ist die Bereitstellung der Dienste auch von der Gesamtkonzeption abhängig, so daß die hier gemachten Angaben nur eine Minimalversion darstellen.

- **Eingabe** stellt bereit
  - `nextchar` - geht zum nächsten anliegenden Zeichen aus der Eingabe über,
  - `currchar` - liefert das aktuelle anliegende Zeichen,
  - `currclass` - liefert die Klasse des aktuell anliegenden Zeichens,
  - `currpos` - liefert die Quelltextposition der aktuellen Zeichens.
- **Scanner** stellt bereit
  - `nextmor` - geht zum nächsten Morphem über,
  - `mortext` - liefert den Text des aktuellen Morphems,
  - `morcode` - liefert den Code des aktuellen Morphems,
  - `morpos` - liefert die Anfangsposition des aktuellen Morphems,
  - `morval` - liefert den Wert des aktuellen Morphems.

  Ein wichtiges, aber oft unterschätztes, Problem besteht in der Fehlerbehandlung, d.h. in der Angabe der möglichst korrekten Fehlerposition und einer Klassifikation des Fehlers. Für den Scanner ergeben sich zwei Möglichkeiten:

  - **Die unmittelbare Reaktion**, indem der Scanner eine eigene Fehlernachricht erzeugt und einen Abbruch erzwingt.
  - **Die mittelbare Reaktion**, indem der Scanner ein (oder mehrere) *Fehlermorphem* erzeugt und die Arbeit fortsetzt.

  Ich bevorzuge die mittelbare Reaktion, da

  - damit die Fehlerbehandlung weitgehend (bis auf falsche Kontextabhängigkeiten) auf den Parser konzentriert ist und
  - eine bessere Möglichkeit zur Klassifikation des Fehlers und der Stabilisierung des Verfahrens vorhanden ist, da der Parser eine globalere Sicht hat.

2.4 Entwicklungstechnologie

Beim Entwurf und der Implementierung eines Scanners kann man

1. weitgehend von Hand vorgehen, falls es relativ wenige, gut überschaubare Morphemarten gibt oder
2. ein Werkzeug (z.B. lex) benutzen, um den Scanner automatisch zu generieren.

Im **Fall 1** wird zunächst die Überführungsfunktion des Aktzeptors, am günstigsten durch eine Tabelle repräsentiert, entworfen.

Im ersten Schritt klassifiziert man die Eingabezeichen gemäß der zu erwartenden Auswirkung auf die Morphembildung ein. Damit hat man die Kopfzeile der Tabelle für die Überführungsfunktion ermittelt und zugleich Anforderungen an die Dienste der Eingabe fixiert.

Anschließend bestimmt man die erforderlichen Zustände und die Reaktion der Überführungsfunktion. Dabei geht man schrittweise vor, indem vom Anfangszustand startend auf jede mögliche Eingabe die notwendige Reaktion beschreibt, die

- entweder auf einen bereits konzipierten Zustand führt
- oder die Einführung eines neuen Zustandes notwendig macht.

Bei jedem Zustand ist zu entscheiden, ob man es sich um einen Endzustand (impliziter oder expliziter) oder nicht handelt.
Beispiel für Steuertabelle eines Scanners

Nachdem die Steuerfunktion in Form einer Tabelle ermittelt ist, erfolgt die Implementierung der Dienste `nextmor, morcode, morpos, morval,...` durch

- ein Programmfragment, etwa einem in eine Iteration eingebetteten Entscheidungsbaum
  Beispiel für einen hand-implementierten Scanner oder
- einen fest programmierten Akzeptor, der über die Tabelle gesteuert wird.
  Beispiel für einen tabellen-gesteuerten Akzeptor

Im **Fall 2** ist man an die durch das Werkzeug vorgegebenen Technologie gebunden.

Bei *lex* hat man den Weg beschritten, daß der Scanner als C-Programmfragment beschrieben wird, in das deklarative Anteile eingebaut sind, die dann durch einen Generator in C-Code aufgelöst werden. Das Grundprinzip ist:

- Die einzelnen Morpheme (Token genannt) werden über reguläre Ausdrücke beschrieben. In den regulären Ausdrücken können *Wildcards* eingebaut werden, so daß ein regulärer Ausdruck als ein Muster (pattern) fungiert, das mit dem entsprechenden Morphem angeglichen (matched) wird.
- Jedem Morphem kann man eine Aktion (in Form einer C-Anweisung) zuordnen.
- Es existieren einige globale, vordefinierte Variable und Funktionen. Dort können Morpheminformationen abgelegt und Standardaktionen ausgelöst werden. Zum Beispiel die Variable `yytext`, die einen Zeiger auf den Text des aktuell ermittelten Morphems als Wert besitzt. Die Variable `yylval` (entpricht unserem `morval`) dient dazu einen Wert für das aktuelle Morphem aufzunehmen. Die Funktion `int yylex(void)` veranlaßt die Analyse des nächsten Morphems (entspricht unserem `nextmor`).
- Weitere Variable und Funktionen können nach Bedarf hinzugefügt werden.

- Der Aufruf kann für jedes Morphem einzeln (über `yylex`) oder für den gesamten Quelltext insgesamt (durch eine eigene `main`-Funktion) erfolgen.

Durch dieses Konzept ist `lex` ein sehr flexibles Werkzeug, das auch für andere Aufgaben (z.B. Datenanalyse) eingesetzt werden kann. Allerdings muß der Nutzer mit der Sprache C vertraut sein und die Konventionen von *lex* genau beachten.

Beispiel für Morphemdefinition in lex

Kapitel 3

Parsing

3.1 Grundstrategien

Gegeben sind:

- eine kontextfreie Grammatik mit dem Satzsymbol S und
- ein Text $\alpha \in L(S)$.

Gesucht sind:

- der Syntaxbaum σ mit $\sigma(\varepsilon) = S$ und $\sigma^*(\varepsilon) = \alpha$, falls dieser existiert
  oder sonst
- eine Fehlermitteilung.

Das allgemeine Vorgehen ist eine *Versuch und Irrtum* (*trial and error*) Strategie.

3.1.1 Top-down Strategie

- Ausgehend vom Satzsymbol S wird versucht, eine Ableitung $S \rightarrow^* \alpha$ zu konstruieren.
- Bei der Ableitung wird jeweils das am weitesten links stehende Metasymbol ersetzt. Man spricht von einer *links-kanonischen Ableitung*.
- Die Auswahl einer Alternativen erfolgt stur in einer festen Reihenfolge, z.B. von links nach rechts.
- Beim Erkennen einer *Sackgasse* (Irrtum!) wird die Ableitung bis zur letzten möglichen Verzweigung zurückgegangen (*back-tracking*).

Eine **Sackgasse** zu α
ist eine Folge von Ableitungsschritten

$$\alpha_r \rightarrow \alpha_{r+1} \rightarrow \ldots \rightarrow \alpha_{r+k}$$

mit

- $S \rightarrow^* \alpha_r \rightarrow^* \alpha$ und
- $\forall i : 1 \leq i \leq k \;\Rightarrow\; \alpha_{r+i} \not\rightarrow^* \alpha$.

Beispiel für top-down-Analyse

Probleme sind:

- die Vermeidung des Einlaufens in Sackgassen durch das frühzeitige Erkennen
  (Offenbar ist man in einer Sackgasse, falls der terminale Präfix von α_{r+i}, d.h. der maximale Präfix, der kein Metasymbol enthält, kein Präfix von α ist.)
  und damit verbunden
- die Linksrekursivitäten in der Grammatik, d.h. Metasymbole A mit der Eigenschaft, daß Ableitungen der Form $A \rightarrow^* A\alpha$ existieren, die das rechtzeitige Erkennen von Sackgassen praktisch unmöglich machen.

3.1.2 Bottom-up Strategie

- Ausgehend von α wird versucht, eine *Rückwärtsableitung* $\alpha \leftarrow^* S$ zu konstruieren, indem die Regeln rückwärts angewendet werden. Das bedeutet, falls eine Teilzeichenkette mit einer Alternative eines Metasymbols übereinstimmt, kann diese durch das Metasymbol ersetzt werden.
- Es wird immer versucht, die am weitesten links stehende Teilzeichenkette zu ersetzen,
- Beim Erkennen einer Sackgasse wird die Rückwärtsableitung bis zur letzten möglichen Verzweigung zurückgegangen.

Eine **Sackgasse** zu α
ist eine Folge von Rückwärts-Ableitungsschritten

$$\alpha_r \leftarrow \alpha_{r+1} \leftarrow \ldots \leftarrow \alpha_{r+k}$$

mit

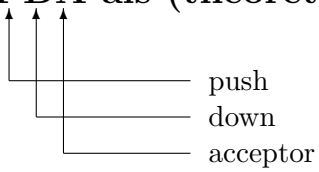
- $\alpha \leftarrow^* \alpha_r \leftarrow^* S$ und
- $\forall i : 1 \leq i \leq k \;\Rightarrow\; \alpha_{r+i} \not\leftarrow^* S.$

Beispiele für bottom-up-Analyse

Probleme sind:

- die sehr komplizierte Sackgassenerkennung und
- die komplizierte Organisation des back-tracking.

3.2 Der PDA als (theoretisches) Modell



Für die ausführliche Beschreibung von endlichen Akzeptoren
siehe wieder das GTI-Skript, Abschnitt 6.3.
Wir betrachten hier eine spezielle Variante, definiert als ein 4-Tupel:

$$PDA = (Z, \Sigma, z_a, \delta)$$

mit

- Z - eine **endliche Zustandsmenge**
- Σ - das **Eingabealphabet**,
- $z_a \in Z$ - ein **Anfangszustand**,
- $\delta : Z \times (\Sigma \cup \{\varepsilon\}) \to \wp(Z^*)$ - die **Überführungsfunktion**.

Die **Arbeitsweise** ist durch den *taktweisen Übergang* zwischen *Situationen*, in denen sich der PDA befinden kann, beschrieben.

Situation: $(\gamma, \alpha) \in Z^* \times \Sigma^*$

wobei

- γ die aktuelle Belegung des *stack* (auch *Stapel, Keller* genannt) ist, in dem sich der PDA befindet und
- α das akuelle Wort an der Eingabe (hier auch Eingabetext genannt)

beschreibt.

Takte:

(1) $(\gamma z, a\alpha) \mapsto (\gamma\gamma', a\alpha)$ falls $z \in Z \wedge a \in \Sigma \wedge \gamma' \in \delta(z, \varepsilon)$

(2) $(\gamma z, a\alpha) \mapsto (\gamma\gamma', \alpha)$ falls $z \in Z \wedge a \in \Sigma \wedge \gamma' \in \delta(z, a)$

Bemerkung: Unser PDA arbeitet nichtdeterministisch, wobei in einer Situation

- in der Taktarten (1) und (2) mehrere verschiedene Kellerbelegungen möglich sind und
- sowohl Taktart (1) als auch (2) anwendbar sein können.

Die durch einen PDA **akzeptierte Sprache** $L(PDA)$ ist dann beschrieben als:

$$L(FSA) := \{\alpha \mid (z_a, \alpha) \mapsto^* (\varepsilon, \varepsilon)\}.$$

Einen PDA, der von der kontextfreien Grammatik $G = (V, \Sigma, S, P)$ erzeugten Sprache $L(S)$ akzeptiert, erhalten wir durch die Festlegungen

- $Z := V$,
- $z_a := S$,
- $\delta(A, \varepsilon) := \{\overleftarrow{\alpha} \mid (A, \alpha) \in P\}$ für $A \in M$ und
- $\delta(a, a) := \{\varepsilon\}$ für $a \in \Sigma$.

Dieses Vorgehen entspricht der **top-down Strategie!**

Beispiel für einen top-down PDA

Folgende **Modifikationen** des PDA dienen der technischen Vereinfachung und der Möglichkeit, einen Syntaxbaum zu erzeugen, was ja ein Resultat des Scanners sein soll:

- Jeder Takt kann von den obersten i Kellerelementen abhängen.
- Zusätzlich entsteht eine Ausgabe.
- Statt die Takte durch die Überführungsfunktion δ zu steuern, werden die möglichen Taktübergänge explizit angegeben.

Situation: $(\gamma, \alpha : \kappa) \in Z^* \times \Sigma^* \times (M \times \mathbb{N})^*$

wobei

- γ die aktuelle Belegung des Kellers ist,
- α das akuelle Wort an der Eingabe ist und
- κ der linearisierte, bisher aufgebaute **abstrakte** Syntaxbaum ist.

Die Trennung der letzten Komponente durch ein Kolon ist nur aus Gründen der Übersichtlichkeit so gewählt.

Jetzt ist der **top-down** PDA zur kontextfreien Grammatik $G = (V, \Sigma, S, P)$ durch folgende Takte beschrieben:

$$(1) \quad (\gamma A, \varphi : \kappa) \quad \mapsto \quad (\gamma \overleftarrow{[A, i]}, \varphi : \kappa(A, i)) \quad \text{falls } A \in M$$

$$(2) \quad (\gamma a, a\varphi : \kappa) \quad \mapsto \quad (\gamma, \varphi : \kappa) \quad \text{falls } a \in \Sigma$$

Man nennt:

- einen Takt der Art (1) einen **reduce**-Takt und
- einen Takt der Art (2) einen **accept**-Takt.

Für $\alpha \in L(S)$ gilt:

$$(S, \alpha : \varepsilon) \mapsto^* (\varepsilon, \varepsilon : \kappa)$$

wobei κ die Präfixlinearisierung des abstrakten Syntaxbaumes ist.

Beispiel für modifizierten top-down PDA mit Erzeugung des abstrakten Syntaxbaumes

Für den **bottom-up** PDA wählen wir die Takte:

$$\begin{array}{lllll} (1) & (\gamma[A,i], \varphi : \kappa) & \mapsto & (\gamma A, \varphi : \kappa(A,i)) & \text{falls } A \in M \\ (2) & (\gamma, a\varphi : \kappa) & \mapsto & (\gamma a, \varphi : \kappa) & \text{falls } a \in \Sigma \end{array}$$

Für $\alpha \in L(S)$ gilt:

$$(\varepsilon, \alpha : \varepsilon) \mapsto^* (S, \varepsilon, \kappa)$$

wobei κ die Postfixlinearisierung des abstrakten Syntaxbaumes ist.

Beispiel für modifizierten bottom-up PDA mit Erzeugung des abstrakten Syntaxbaumes

3.3 Analyse mit back-tracking

Die beiden vorgestellten $PDA's$ arbeiten *nichtdeterministisch*, müssten also raten, welche der möglichen Takte auszuführen sind, um eine erfolgreiche Taktfolge zu erhalten, falls eine solche existiert. Das Raten ist aber in der Praxis nicht möglich. Es muss durch Suchmechanismen ersetzt werden, die im *Lösungsraum* nach gewissen Strategien die Lösung suchen.

Bei uns ist der Lösungsraum ein *Ableitungsbaum* (nicht mit dem Syntaxbaum verwechseln!), dessen Knoten mögliche Situationen sind. Die Söhne eines Knotens $(\gamma, \varphi : \kappa)$ sind alle die Situationen $(\gamma', \varphi' : \kappa')$, für die Takte $(\gamma, \varphi : \kappa) \mapsto (\gamma', \varphi' : \kappa')$ existieren.

Beispiel für Ableitungsbaum

Eine sichere Suchstrategie ist die *Breitensuche*, bei der im Suchbaum, startend mit der Wurzel, zunächst alle Knoten mit dem gleichen Abstand zur Wurzel durchmustert werden bis man auf die Lösung stößt. Existiert eine Lösung, so wird diese nach endlich vielen Suchschritten erreicht.

Die Breitensuche ist aufwendig zu organisieren, da man viele Zwischeninformationen aufbewahren muss. Deshalb benutzt man oft die *Tiefensuche*, bei der startend bei der Wurzel die Knoten ihrer lexikographischen Ordnung gemäß durchmustert werden (das entspricht der Präfixlinearisierung eines Baumes). Pfade im Baum, auf denen keine Lösung liegt, enden in Sackgassen!
Bei der Tiefensuche wird eine Lösung nach endlich vielen Suchschritten erreicht, falls keine unendlichen Pfade existieren.

Beispiel für Breiten- und Tiefensuche

Wir erweitern jetzt den top-down Akzeptor um back-tracking-Mechanismen, die es erlauben, beim Erkennen einer Sackgasse zu einem möglichen Verzweigungspunkt zurückzulaufen und eine neue Variante zu probieren. Dieser Akzeptor arbeitet deterministisch und bestimmt den Syntaxbaum nach dem Prinip der Tiefensuche. Da für die bottom-up-Strategie das back-tracking aufwendig zu implementieren ist, demonstrieren wir das nur für die top-down-Strategie.

3.3.1 Volles back-tracking

Um back-tracking Mechanismen einzubauen, werden zwei Arten von Situationen unterschieden:

- **brute-force**-Situationen, in denen *Vorwärtstakte* ausgeführt werden, um die Tiefensuche voranzutreiben. Diese Situationen werden mit einem f indiziert.
- **back-track**-Situationen, in denen *Rückwärtstakte* ausgeführt werden, um aus Sackgassen herauszufinden. Diese Situationen werden mit einem b indiziert.

Eine weitere hilfreiche Modifikation ist die Erzeugung des vollen Syntaxbaumes während der brute-force-Takte. Das vereinfacht beim back-tracking die Rekonstruktion von Eingabe und Keller.

Der **back-tracking-top-down-Akzeptor** ist durch die folgenden Takte beschrieben:

$$
\begin{array}{lllll}
(1) & (\gamma A, \varphi : \kappa)_f & \mapsto & (\gamma \overleftarrow{[A,1]}, \varphi : \kappa(A,1))_f & \text{falls } A \in M \\
(2) & (\gamma a, a\varphi : \kappa)_f & \mapsto & (\gamma, \varphi : \kappa a)_f & \text{falls } a \in \Sigma \\
(3) & (\gamma, \varphi : \kappa)_f & \mapsto & (\gamma, \varphi : \kappa)_b & \text{falls } \gamma = \varepsilon \\
 & & & & \vee\ (\gamma = \gamma' a \wedge a \in \Sigma \wedge a \not\sqsupseteq \varphi) \\
(4) & (\gamma, \varphi : \kappa a)_b & \mapsto & (\gamma a, a\varphi : \kappa)_b & \text{falls } a \in \Sigma \\
(5) & (\gamma \overleftarrow{[A,i]}, \varphi : \kappa(A,i))_b & \mapsto & (\gamma A, \varphi : \kappa)_b & \text{falls } i = |[A]| \\
(6) & (\gamma \overleftarrow{[A,i]}, \varphi : \kappa(A,i))_b & \mapsto & (\gamma \overleftarrow{[A,i+1]}, \varphi : \kappa(A,i+1))_f & \text{falls } i < |[A]|
\end{array}
$$

Beispiel für back-tracking top-down Akzeptor

Problem: Linksrekursivitäten in der Grammatik lassen Sackgassen nicht erkennen.

Eine **mögliche Lösung** für das Problem der Linksrekursivitäten ist deren **Elimination**. Zwei Verfahren bieten sich an:

- Das Umwandeln in **Rechtsrekursivitäten** der Form $A \to^* \alpha A$.
  Das verlangt eine Transformation in der Grammatik, was die Signatur verändert und sich auf Bedeutungen auswirken kann (Auswertungsreihenfolge von Operatoren können sich ändern)!
  Beispiel für back-tracking top-down Akzeptor ohne Linksrekursivitäten
- Das Umwandeln in **Zyklen** der Form $A \to \{\alpha\}^+$.
  Das erfordert gewisse Tricks beim Parsing (beim LF(k)-Verfahren behandelt), um trotzdem den ursprünglich angestrebten Syntaxbaum zu erhalten.

3.3.2 Analyse mit fast-back

Das **vereinfachende Prinzip** beim fast-back ist:

Erfolgreiche Teilableitungen werden nicht angezweifelt!

Beim back-tracking wird also in solche erfolgreichen Teilableitungen nicht mehr hineingegangen.

Beispiel für fast-back-Analyse

Der **Vorteil** besteht darin, daß

- beim back-tracking Informationen aus dem Syntaxbaum nicht benötigt werden, wodurch man wieder nur den abstrakten Syntaxbaum aufbauen braucht und
- die Rekonstruktion der Eingabe entfällt, indem man nur Positionsinformationen mitführt.

Die fast-back Analyse arbeitet nicht für jede Grammatik korrekt!

Beispiel für keine fast-back-Grammatiken

Definition 3.3.1 *Eine kontextfreie Grammatik hat die fast-back-Eigenschaft,*

$$\textit{falls aus}\quad S \to^* \varphi A\beta \begin{cases} \varphi\alpha\beta \to^* \varphi\psi\beta \\ \varphi\alpha'\beta \to^* \varphi\psi\psi'\beta \to^* \varphi\psi\psi'\chi \end{cases} \quad \textit{und } \alpha \neq \alpha' \quad \textit{folgt, daß} \quad \beta \to^* \psi'\chi;$$

wobei: $S, A \in M,\ \varphi, \psi, \psi', \chi \in \Sigma^*,\ \alpha, \alpha', \beta \in V^*.$

Satz 3.3.1 ***Hinreichende Bedingung*** *für die fast-back-Eigenschaft ist, daß für jedes Metasymbol* $A \in M$*, gilt:*

$$\textit{Aus}\quad A \begin{cases} \alpha \to^* \psi \\ \alpha' \to^* \overline{\psi} \end{cases} \quad \textit{und } \alpha \neq \alpha' \quad \textit{folgt, daß} \quad \psi \sqcap \overline{\psi};$$

wobei: $A \in M,\ \psi, \overline{\psi} \in \Sigma^*,\ \alpha, \alpha', \in V^*.$

Beweis:
Die Prämisse in der Definition 3.3.1 ist nie erfüllt, deshalb ist die fast-back-Eigenschaft immer gesichert. ■

Folgerung 3.3.2 *Falls die hinreichende Bedingung aus Satz 3.3.1 erfüllt ist, dann ist die Grammatik eindeutig.*

Beweis:
Klar ■

Zur **Überprüfung der hinreichenden Bedingung** führen wir für $k \in \mathbb{N}$ folgende Funktionen ein:

- $pre_k : V^* \to V^*$ mit $pre_k(\alpha) := \begin{cases} \alpha & \text{falls } |\alpha| \leq k \\ \alpha' & \text{falls } \alpha = \alpha'\alpha'' \wedge |\alpha'| = k \end{cases}$
- $First_k : V^* \to \wp(\Sigma^*)$ mit $First_k(\varphi) := \{pre_k(\alpha) \mid \varphi \to^* \alpha \wedge \alpha \in \Sigma^*\}$.
- $Pre : \wp(V^*) \to \wp(V^*)$ mit $Pre(\mathcal{A}) := \{\alpha' \mid \alpha' \sqsubseteq \alpha \wedge \alpha \in \mathcal{A}\}$.
- $Pre_k : \wp(V^*) \to \wp(V^*)$ mit $Pre_k(\mathcal{A}) := \{pre_k(\alpha) \mid \alpha \in \mathcal{A}\}$.

Nun gilt folgendes

Lemma 3.3.3 *Falls für alle $A \in M$ und für alle Paare $i, j \in \mathbb{N}$ mit $1 \leq i \neq j \leq |[A]|$ ein $k \in \mathbb{N}$ existiert mit*

$$First_k([A,i]) \cap Pre(First_k([A,j])) = \emptyset,$$

so hat die Grammatik die fast-back-Eigenschaft.

Beweis:
Klar nach Satz 3.3.1 ■

Die **Berechnung der $First_k(A)$-Mengen** beruht auf folgenden

Lemma 3.3.4 *Eigenschaften:*

- $First_k(a) = \{a\}$ *für* $a \in \Sigma$,
- $First_k(\varepsilon) = \{\varepsilon\}$,
- $First_k(x_1 \ldots x_n) = Pre_k(First_k(x_1) \bullet \ldots \bullet First_k(x_n))$ *für* $x_i \in V$,
- $First_k(A) := \bigcup\{First_k([A,i] \mid 1 \leq i \leq |[A]|\}$ *für* $A \in M$.

Beweis:
Klar ■

Folgerung 3.3.5 *Die Mengen $First_k(A)$ für die Metasymbole einer kontextfreien Grammatik ergeben sich als die kleinste Lösung eines Gleichungssystems, das entsprechend der Gleichungen aus Lemma 3.3.4 aufzustellen ist.*

Beweis:
Die *cpo* $(\wp(V^*), \subseteq)$ mit den Operationen Vereinigung, Komplexprodukt und Präfixbildung erfüllt die Voraussetzungen des Fixpunktsatzes von Tarski-Knaster
siehe Algebra-Skript, Abschnitt 2.7,
wobei die leere Menge das Minimum ist, mit dem die Iteration gestartet wird. ■

Beispiel für die Berechnung der $First_k$-Mengen und der Überprüfung der fast-back Eigenschaft.

Für die **Implementierung** ergeben sich **zwei Varianten**:

- **Syntax-gesteuert:** es wird ein allgemeiner Akzeptor implementiert, der über die (aufbereitete) Grammatik gesteuert wird und damit einen Parser realisiert.
- **Syntax-gebunden:** aus der Grammatik wird ein spezieller Parser (eventuell automatisch) erzeugt, der nur für diese Grammatik die Syntaxanalyse vornimmt.

Für einen **syntax-gesteuerten Akzeptor der fast-back-Analyse** wird folgende weitere Modifizierung unseres Akzeptors vorgenommen:

- Im Kellerinhalt wird nur eine "Analyseabsicht" mit den Elementen $(A, i, j, q : s)$ abgelegt. Dabei beschreiben:
  - die ersten drei Komponen das Element $[A, i, j]$ (j-tes Symbol in i-ter Alternative von A),
  - q die aktuelle Quelltextposition,
  - s die aktuelle Position im Syntaxbaum.
- Die zweite und dritte Komponente einer Situation beinhalten jetzt statt der Eingabe und dem Syntaxbaum nur die entsprechenden Positionsangaben. Beim Zurücksetzen dieser Positionsangaben muß der Scanner entsprechend reagieren.
- Es werden folgende *Dienste* genutzt:
  - `nextmor`: bekannte Methode vom Scanner,
  - `morcode`: Variable vom Scanner, die sich auf die aktuelle Eingabeposition bezieht.
  - `outsyn(x)`: setzt `x` in den Syntaxbaum an der Position ein, die in der beim Takt **erreichten** Situation als dritte Komponente vermerkt ist.

Der **fast-back Akzeptor** ist durch die folgenden Takte beschrieben:

(1)	$(\gamma(A,i,j,q:s),k:l)_f$	$\mapsto$	$(\gamma(A,i,j+1,q:s)(B,1,1,k:l),k:l+1)_f$	
	`+outsyn(B,1)`		falls $j \leq \lvert[A,i]\rvert$ und $[A,i,j] = B \in M$	
(2)	$(\gamma(A,i,j,q:s),k:l)_f$	$\mapsto$	$(\gamma(A,i,j+1,q:s),k+1:l)_f$	
	`+nextmor()`		falls $j \leq \lvert[A,i]\rvert$ und $morcode = [A,i,j] \in \Sigma$	
(3)	$(\gamma(A,i,j,q:s),k:l)_f$	$\mapsto$	$(\gamma,k:l)_f$	falls $j > \lvert[A,i]\rvert$
(4)	$(\gamma(A,i,j,q:s),k:l)_f$	$\mapsto$	$(\gamma(A,i,j,q:s),k:l)_b$	
			falls $j \leq \lvert[A,i]\rvert$ und $morcode \neq [A,i,j] \in \Sigma$	
(5)	$(\gamma(A,i,j,q:s),k:l)_b$	$\mapsto$	$(\gamma(A,i+1,1,q:s),q:s)_f$	
	`+outsyn(A,i+1)`		falls $i < \lvert A\rvert$	
(6)	$(\gamma(A,i,j,q:s),k:l)_b$	$\mapsto$	$(\gamma,q:s)_b$	falls $i = \lvert A\rvert$

Hinweise:

- Der Akzeptor startet mit der Situation $((S,1,1,1:1),1:1)$, wobei der Syntaxbaum bereits aus dem Eintrag $(S,1)$ besteht!
- Der Takt (3) wird ausgeführt, falls die Ableitung des Metasymbols A mit der Alternative $[A,i]$ erfolgreich war!

Beispiel für fast-back-Akzeptor

Jetzt läßt sich auch eine Postfixlinearisierung (die ist mitunter günstiger für das backend!) des abstrakten Syntaxbaumes erzeugen, indem folgende Takte verändert werden:

$$
\begin{array}{lll}
(1') & (\gamma(A,i,j,q:s),k:l)_f & \mapsto \quad (\gamma(A,i,j+1,q:s)(B,1,1,k:l),k:l)_f \\
 & & \text{falls } j \leq |[A,i]| \text{ und } [A,i,j] = B \in M \\
(3') & (\gamma(A,i,j,q:s),k:l)_f & \mapsto \quad (\gamma,k:l+1)_f \\
 & \texttt{+outsyn(A,i)} & \text{falls } j > |[A,i]| \\
(5') & (\gamma(A,i,j,q:s),k:l)_b & \mapsto \quad (\gamma(A,i+1,1,q:s),q:s)_f \\
 & & \text{falls } i < |A|
\end{array}
$$

Hinweis:

- Der Akzeptor startet mit der Situation $((S,1,1,1:0),1:0)$, wobei der Syntaxbaum am Anfang leer ist!

Beispiel für fast-back-Akzeptor mit PostfixLinearisierung des Syntaxbaumes

Eine **syntax-gebundenen Parser für die fast-back-Analyse** erhält man über ein **System rekursiver Funktionen**, die als C++-Funktionen beschrieben sind.

Hierbei sei

- `Tmorcode` der Typ von `morcode`,
- `Tmorpos` der Typ von `morpos`, wobei diese Variable durch `nextmor()` inkrementiert wird und
- `Tsynpos` der Typ von `synpos`, einer globalen Variablen, in der die aktuelle Position des aufgebauten Syntaxbaumes gespeichert ist und die von `outsyn(...)` inkrementiert wird.

Erforderliche **Funktionen** sind:

- eine Funktion

```
bool check (Tmorcode m)
{   if(m==morcode) { nextmor(); return 1; }
    return 0;
}
```

- zu jedem Metasymbol A eine Funktion

```
bool check_A (Tmorpos mp, Tsynpos sp)
{   if(alt_1_of_A(mp,sp))  return 1;
    .
    .
    .
    if(alt_n_of_A(mp,sp))  return 1; //n==|[A]|
    return 0;
}
```

- zu jeder Alternativen $x_1 \ldots x_m$ von A eine Funktion

```
bool alt_i_of_A (Tmorpos mp, Tsynpos sp)
{   morpos=mp; synpos=sp; outsyn(A,i);
    if(!ch_1) return 0;
    .
    .
    .
    if(!ch_m)  return 0;
    return 1;
}
```

$$\text{wobei} \qquad \texttt{ch\_i} = \begin{cases} \texttt{check}(x_i) & \text{für } x_i \in \Sigma \\ \texttt{check\_}x_i\texttt{(morpos,synpos)} & \text{für } x_i \in M \end{cases}$$

Hinweis:

- Der Start erfolgt durch Aufruf von `check_S(1,0)`, wobei S das Startsymbol der Grammatik ist.

Beispiel für syntax-gebundenen fast-back Parser

Die Postfixlinearisierung des abstrakten Syntaxbaumes erhält man durch die leichte Modifikation:

```
bool alt_i_of_A (Tmorpos mp, Tsynpos sp)
{   morpos=mp; synpos=sp;
    if(!ch_1) return 0;
    .
    .
    .
    if(!ch_m)  return 0;
    outsyn(A,i);
    return 1;
}
```

3.4 Analyse ohne back-tracking

Um auf back-tracking zu verzichten, muß in jedem Schritt höchstens ein Takt ausführbar sein.

Bei der top-down-Strategie ist im reduce-Takt zu entscheiden, welche der Alternativen des an der Kellerspitze stehenden Metasymbols zu wählen ist.

Bei der bottom-up-Strategie muß zusätzlich zwischen den reduce- und accept-Takten unterschieden werden.

Allen hier vorgesetellten Verfahren ist gemeinsam, daß für die Entscheidungsfindung k Zeichen aus der anliegenden Eingabe herangezogen werden. Man nennt diese k Zeichen die **look-ahead**-Symbole.

3.4.1 Das LF(k)-Verfahren

Das LF(k)-Verfahren nutzt die top-down-Strategie und entscheidet *ausschließlich* auf der Grundlage der k look-ahead-Symbole, welche Alternative in einem reduce-Takt zu wählen ist.

Definition 3.4.1 *Eine kontextfreie Grammatik hat die LF(k)-Eigenschaft (kurz: ist eine LF(k)-Grammatik), falls*

$$\text{aus } S \begin{cases} * \ \varphi A\beta \to \varphi\alpha\beta \to^* \varphi\psi \\ * \ \varphi' A\beta' \to \varphi'\alpha'\beta' \to^* \varphi'\psi' \end{cases} \quad \text{und} \quad pre_k(\psi) = pre_k(\psi') \quad \text{folgt, daß } \alpha = \alpha';$$

wobei: $S, A \in M,\ \alpha, \alpha', \beta, \beta' \in V^*, \varphi, \varphi', \psi, \psi' \in \Sigma^*$.

Folgerung 3.4.1 *Jede LF(k)-Grammatik ist eindeutig.*

Der nächste Satz ist ein wesentlicher Schritt, um die $LF(k)$-Eigenschaft überprüfen zu können.

Satz 3.4.2 *Eine Grammatik hat **genau dann** die LF(k)-Eigenschaft, wenn*

$$\text{aus } S \begin{cases} * \ \varphi A\beta \to \varphi\alpha\beta \\ * \ \varphi' A\beta' \to \varphi'\alpha'\beta' \end{cases} \quad \text{und} \quad First_k(\alpha\beta) \cap First_k(\alpha'\beta') \neq \emptyset \ \text{ folgt, daß } \alpha = \alpha';$$

wobei: $S, A \in M,\ \alpha, \alpha', \beta, \beta' \in V^*, \varphi, \varphi' \in \Sigma^*$.

Beweis:
Der Beweis geht unmittelbar aus der Definition der $First_k$-Funktion hervor! ■

Das verbleibende Problem besteht in der Überprüfung von $First_k(\alpha\beta) \cap First_k(\alpha'\beta') \neq \emptyset$ in obigem Satz. Um dies zu lösen, führen wir ein:

- $Follow_k : M \to \wp(\Sigma^*)$ mit $Follow_k(A) := \bigcup\{First_k(\beta) \mid \exists\varphi \in V^* : S \to^* \varphi A\beta\}$

Für $First_k(\alpha\beta)$ aus dem Satz 3.4.2 ergibt sich damit

$$First_k(\alpha\beta) \subseteq Pre_k(First_k(\alpha) \bullet Follow_k(A))$$

und da sich das Kriterium auf alle möglichen $\beta \in Follow_k(A)$ bezieht, ergibt sich:

Satz 3.4.3 *Eine Grammatik hat **genau dann** die LF(k)-Eigenschaft, wenn für alle Metasymbole $A \in M$ und für alle i, j mit $1 \leq j \neq j \leq |[A]|$ gilt:*

$$Pre_k(First_k([A,i]) \bullet Follow_k(A)) \cap Pre_k(First_k([A,j]) \bullet Follow_k(A)) = \emptyset.$$

Um immer k-look-ahead-Symbole zur Verfügung zu haben, wird eine **zusätzliche Regel** in die Grammatik aufgenommen:

$$S' \rightarrow S \vdash^k,$$

wobei

- S das alte Satzsymbol,
- S' das neue Satzsymbol und
- $\vdash^k$ k Endezeichen sind.

Für S' gilt:

- $Follow_k(S') := \{\varepsilon\}$.

Für alle ursprünglichen Metasymbole $A \in M$ gilt nun das

Lemma 3.4.4 *:*

$$Follow_k(A) = \bigcup\{Pre_k(First_k(\beta) \bullet Follow_k(B)) \mid B \rightarrow \varphi A \beta\}$$

Beweis:
Wegen

$$S' \rightarrow^* \varphi A \beta$$

genau dann, wenn

$$\exists B : S' \rightarrow^* \varphi_1 B \beta_2 \rightarrow \varphi_1 \varphi_2 A \beta_1 \beta_2 \text{ und } \beta = \beta_1 \beta_2$$

gilt:

$$\begin{array}{lcll} Follow_k(A) & = & \bigcup\{ & First_k(\beta) \mid S \rightarrow^* \varphi A \beta\} \\ & = & \bigcup\{ & First_k(\beta_1\beta_2) \mid S \rightarrow^* \varphi_1 B \beta_2 \rightarrow \varphi_1\varphi_2 A \beta_1\beta_2\} \\ & = & \bigcup\{ & Pre_k(First_k(\beta_1) \bullet First_k(\beta_2)) \mid S \rightarrow^* \varphi_1 B \beta_2 \wedge B \rightarrow \varphi_2 A \beta_1\} \\ & = & \bigcup\{ & Pre_k(First_k(\beta_1) \bullet Follow_k(B)) \mid B \rightarrow \varphi_2 A \beta_1\} \end{array}$$

■

Folgerung 3.4.5 *Die Mengen $Follow_k(A)$ für die Metasymbole einer kontextfreien Grammatik ergeben sich als die kleinste Lösung eines Gleichungssystems, das entsprechend der Gleichungen aus Lemma 3.4.4 aufzustellen ist.*

Beispiel für Gleichungssystem für $Follow_k$-Mengen und Überprüfung der LF(k)-Eigenschaft

Beispiele für Grammatiken, die für kein k die LF(k)-Eigenschaft haben

Aus Satz 3.4.3 ergibt sich die

Folgerung 3.4.6 *Wenn eine Grammatik die LF(k)-Eigenschaft hat, so gibt es für jedes $\alpha \in \Sigma^*$ höchstens ein $1 \leq i \leq |[A]|$ mit $\alpha \in Pre_k(First_k([A,i]) \bullet Follow_k(A))$.*

Falls also für die Eingabe φ gilt, daß $pre_k(\varphi) \in Pre_k(First_k([A,i]) \bullet Follow_k(A))$, so muß in einem reduce Takt die Alternative i benutzt werden.

Wir bilden eine **Steuerfunktion** $T : M \times \Sigma^k \to I\!N$ für die Alternativenauswahl, indem

$$T(A,\alpha) = \begin{cases} i & \text{falls } \alpha \in Pre_k(First_k([A,i]) \bullet Follow_k(A)) \\ 0 & \text{sonst} \end{cases}$$

Beispiel für Steuerfunktion

Der **syntax-gesteuerte LF(k)-Akzeptor** ist nun durch folgende Takte beschrieben, wobei wir wieder auf die bereits beim PDA angewendete abstraktere Darstellungsweise benutzen. Als Konsequenz entsteht eine Präfixlinearisierung. Will man eine Postfixlinearisierung erreichen, so hat man analog wie beim fast-back Akzeptor vorzugehen:

$$\begin{array}{lll} (1) & (\gamma A, \varphi : \kappa) & \mapsto \quad (\gamma \overleftarrow{[A,i]}, \varphi : \kappa(A,i)) \\ & & \text{falls } A \in M \wedge i = T(A, pre_k(\varphi)) > 0 \\ (2) & (\gamma a, a\varphi : \kappa) & \mapsto \quad (\gamma, \varphi : \kappa) \\ & & \text{falls } a \in \Sigma \end{array}$$

Beispiel für LF(2)-Analyse

Der **syntax-gebundene LF(1)-Parser** arbeitet analog zum fast-back Parser. Die erforderlichen Funktionen sind hier:

- eine Funktion

```
void check (Tmorcode m)
{   if(m==morcode) nextmor();
    else error("Syntaxfehler");
    //error ist eine Funktion, die den Fehlerabbruch erzwingt
}
```

- zu jedem Metasymbol A eine Funktion

```
void check_A ()
{   switch(T(A,morcode))
    {
       case 1: alt_1_of_A(); return;
       .
       .
       .
       case n: alt_n_of_A(); return; //n==|[A]|
       default: error("Syntaxfehler");
    }
}
```

- zu jeder Alternativen $x_1 \dots x_m$ von A eine Funktion

```
void alt_i_of_A ()
{   outsyn((A,i));
    ch_1; ... ; !ch_m;
}
```

wobei $\texttt{ch\_i} = \begin{cases} \texttt{check}(x_i) & \text{für } x_i \in \Sigma \\ \texttt{check\_}x_i\texttt{()} & \text{für } x_i \in M \end{cases}$

Natürlich kann man die Funktionen `alt_i_of_A()` auch in die Funktionen `check_A()` integrieren!

Beispiel für syntax-gebundenen LF(1) Parser

Zusammenfassung der Schritte für die Entwicklung eines LF(k)-Parsers:

1. Berechnung von $First_k$,
2. Berechnung von $Follow_k$,
3. Berechnung der Steuertabelle T
   (falls diese eindeutig ist, ist die LF(k)-Eigenschaft gesichert).

Durch **Faktorisierung** kann man mitunter eine Grammatik, die nicht $LF(k)$ ist, in eine $LF(k)$-Grammatik transformieren oder auch das k reduzieren!

Regeln der Form

$$A \quad ::= \quad \ldots \ \alpha\beta \mid \alpha\gamma \ \ldots$$

werden transformiert in Regeln der Form

$$\begin{array}{lll} A & ::= & \ldots \ \alpha B \ \ldots \\ B & ::= & \beta \mid \gamma \end{array}$$

Beispiele für Faktorisierung

Beispiel für LF(1) Analyse nach Faktorisierung

Leider führt Faktorisierung nicht immer zum Erfolg, bzw. ist nicht immer anwendbar!

Beispiel für Grammatik, die nicht in LF(k) faktorisiert werden kann

Die **Transformation von Linksrekursivitäten der Form**

$$A \quad ::= \quad B \mid A\,B$$

mit $First_k(B) = \{a1, \ldots, an\}$ in Zyklen der Form

$$A \quad ::= \quad \{B\}^+$$

läßt sich bei syntax-gebundenen LF(k)Parsern durch Austausch der Funktion `check_A` erreichen. Die neue Funktion arbeitet nach dem Prinzip:

```
check_A()
{
    check_B();
    outsyn(A,1);
    do switch(morcode)
       { case a1: case a2: ... case an:
              nextmor();
              check_B;
              outsyn(A,2);
              break;
         default: return;
       } while(1);
}
```

Hierbei ist zu beachten, daß wieder eine **Postfixlinearisierung** erzeugt wird!

Beispiel für Behandlung von Zyklen

3.4.2 Das LL(k)-Verfahren

Das LL(k)-Verfahren nutzt die top-down-Strategie und entscheidet auf der Grundlage der k look-ahead-Symbole *und* des Kellerinhaltes, welche Alternative in einem reduce-Takt zu wählen ist.

Definition 3.4.2 *Eine kontextfreie Grammatik hat die LL(k)-Eigenschaft (kurz: ist eine LL(k)-Grammatik), falls*

$$\text{aus } S \to^* \varphi A\beta \begin{cases} \to \varphi\alpha\beta \to^* \varphi\psi \\ \to \varphi\alpha'\beta \to^* \varphi\psi' \end{cases} \quad \text{und} \quad pre_k(\psi) = pre_k(\psi') \quad \text{folgt, daß } \alpha = \alpha';$$

wobei: $S, A \in M,\ \alpha, \alpha', \beta \in V^*, \varphi, \varphi', \psi, \psi' \in \Sigma^*$.

Folgerung 3.4.7 *Jede LL(k)-Grammatik ist eindeutig.*

Folgerung 3.4.8 *Jede LF(k)-Grammatik ist auch LL(k)-Grammatik.*

Satz 3.4.9 *Eine Grammatik hat* ***genau dann*** *die LL(k)-Eigenschaft, wenn*

$$\text{aus } S \to^* \varphi A\beta \begin{cases} \to \varphi\alpha\beta \\ \to \varphi\alpha'\beta \end{cases} \quad \text{und} \quad First_k(\alpha\beta) \cap First_k(\alpha'\beta) \neq \emptyset \text{ folgt, daß } \alpha = \alpha';$$

wobei: $S, A \in M,\ \alpha, \alpha', \beta \in V^*, \varphi, \varphi' \in \Sigma^*$.

Beweis:
Der Beweis geht unmittelbar aus der Definition der $First_k$-Funktion hervor! ■

Das verbleibende Problem besteht wieder in der Überprüfung von $First_k(\alpha\beta) \cap First_k(\alpha'\beta) \neq \emptyset$ in obigem Satz. Allerdings ist jetzt zu beachten, daß sich der Durchschnitt auf ein gemeinsames β mit $S \to^* \varphi A\beta$ bezieht. Damit können $First_k(\alpha\beta)$ und $First_k(\alpha'\beta)$ nicht mehr über die Menge $Follow_k(A)$ bestimmt werden, da ja darin alle $First_k(\beta)$-Mengen vereinigt sind.

Wir bilden deshalb eine Menge von Mengen $\mathcal{F}_k$ mit

$$\mathcal{F}_k(A) := \{First_k(\beta) \mid S \to^* \varphi A\beta\}.$$

Jetzt ergibt sich der

Satz 3.4.10 *Eine Grammatik hat* ***genau dann*** *die LL(k)-Eigenschaft, wenn*
für alle Metasymbole $A \in M$, *für alle* $F \in \mathcal{F}_k(A)$ *und für alle* i, j *mit* $1 \leq i \neq j \leq |[A]|$
gilt:

$$Pre_k(First_k([A,i]) \bullet F) \cap Pre_k(First_k([A,j]) \bullet F) = \emptyset.$$

Für die Berechnung der $\mathcal{F}_k$-Mengen führen wir eine Operation

$$\otimes_k : \wp(V^*) \times \wp(\wp(V^*)) \to \wp(\wp(V^*))$$

ein mit

$$F \otimes_k \mathcal{F} := \{Pre_k(F \bullet F') \mid F' \in \mathcal{F}\}.$$

Für alle Metasymbole $A \in M$ erhält man das

Lemma 3.4.11

$$\mathcal{F}_k(A) = \bigcup\{First_k(\beta) \otimes_k \mathcal{F}_k(B) \mid B \to \varphi A\beta\}$$

Beweis:
Wegen

$$S' \to^* \varphi A\beta$$

genau dann, wenn

$$\exists B : S' \to^* \varphi_1 B\beta_2 \to \varphi_1\varphi_2 A\beta_1\beta_2 \text{ und } \beta = \beta_1\beta_2$$

gilt:

$$\begin{array}{lrcll} & \mathcal{F}_k(A) & = & \{\ First_k(\beta) & \mid S \to^* \varphi A\beta\} \\ & & = & \{\ First_k(\beta_1\beta_2) & \mid S \to^* \varphi_1 B\beta_2 \to \varphi_1\varphi_2 A\beta_1\beta_2\} \\ & & = & \{\ Pre_k(First_k(\beta_1) \bullet First_k(\beta_2)) & \mid S \to^* \varphi_1 B\beta_2 \wedge B \to \varphi_2 A\beta_1\} \\ & & = & \{\ Pre_k(First_k(\beta_1) \bullet F) & \mid F \in \mathcal{F}_k(B) \wedge B \to \varphi_2 A\beta_1\} \\ & & = & \bigcup\{\ First_k(\beta_1) \otimes_k \mathcal{F}_k(B) & \mid B \to \varphi_2 A\beta_1\} \end{array}$$

■

Folgerung 3.4.12 *Die Mengen $\mathcal{F}_k(A)$ für die Metasymbole einer kontextfreien Grammatik ergeben sich als die kleinste Lösung eines Gleichungssystems, das entsprechend der Gleichungen aus Lemma 3.4.11 aufzustellen ist.*

Aus der Festlegung $S' \to S \vdash^k$ und der Tatsache, daß $S' \to^* S'$ die einzige Ableitung nach S' ist, ergibt sich:

$$\mathcal{F}_k(S') = \{\{\varepsilon\}\} \text{ und } \{\vdash^k\} \in \mathcal{F}(S).$$

Beispiel für die Überprüfung der LL(k) Eigenschaft

Die prinzipielle Arbeitsweise des **syntax-gesteuerten LL(k)-Akzeptors** könnte nun durch folgende Takte beschrieben werden:

$$\begin{array}{lll} (1) & (\gamma A, \varphi : \kappa) & \mapsto \quad (\gamma\overleftarrow{[A,i]}, \varphi : \kappa(A,i)) \\ & & \text{falls } A \in M \wedge pre_k(\varphi) \in First_k([A,i]\overleftarrow{\gamma}) \\ (2) & (\gamma a, a\varphi : \kappa) & \mapsto \quad (\gamma, \varphi : \kappa) \\ & & \text{falls } a \in \Sigma \end{array}$$

Hierbei ist allerdings die begleitende Berechnung von $First_k([A,i]\overleftarrow{\gamma})$ für den aktuellen Kellerinhalt γA zu aufwendig. Da es aber für alle möglichen Kellerinhalte nur endlich viele zu berechnende Mengen gibt, werden diese vorab bestimmt. Im Keller wird dann zusätzlich Information darüber mitgeführt, welche der $First_k$-Mengen aktuell zu betrachten sind.

Beobachtung aus dem Beweis zu Lemma 3.4.11: Wenn $F = First_k(\overleftarrow{\gamma}) \in \mathcal{F}_k(A)$ die zum Kellerinhalt γA gehörende Menge ist und $[A,i] = \varphi B\beta$, dann ist $Pre_k(First_k(\beta) \bullet F)$ die zum Kellerinhalt $\gamma\overleftarrow{\beta}$ gehörende Menge.

Konsequenz ist die Transformation der Grammatik:

- Zu jedem Metasymbol A und zu jeder Menge $F \in \mathcal{F}_k(A)$
führen wir die neuen Metasymbole $\langle A, F\rangle$ ein. Es sei M' die Menge der neuen Metasymbole und $V' := M' \cup \Sigma$.

- Zu jeder Regel $A \to \chi_0 A_1 \chi_1 \dots A_n \chi_n$
führen wir die Regeln $\langle A, F\rangle \to \chi_0 \langle A_1, F_1\rangle \chi_1 \dots \langle A_n, F_n\rangle \chi_n$
mit $F \in \mathcal{F}_k(A), F_i \in \mathcal{F}_k(A_i)$ und $F_i = Pre_k(First_k(\chi_i A_{i+1} \dots A_n \chi_n) \bullet F)$ ein.

Beispiel für die transformierte Grammatik

Konvention: Für eine Zeichenkette $\beta \in V'^*$ sei $\overline{\beta}$ die Zeichenkette, die aus β hervorgeht, indem alle Metasymbole $\langle A, F\rangle$ durch A ersetzt werden.

Lemma 3.4.13 *Für jedes Metasymbol A der ursprünglichen Grammatik und jedes Metasymbol $\langle A, F\rangle$ der transformierten Grammatik gilt:*

$$\forall \alpha \in V^* \exists \beta \in V'^* : \alpha = \overline{\beta} \wedge A \to^* \alpha \quad \Leftrightarrow \quad \langle A, F\rangle \to^* \beta$$

und

$$\langle A, F\rangle \to^* \beta \quad \Rightarrow \quad A \to^* \overline{\beta}.$$

Beweis:
Leicht induktiv über die Länge der Ableitung zu führen. ■

Folgerung 3.4.14 *Es gelten:*

- $L(A) = L(\langle A, F\rangle)$ *für alle* $A \in M$ *und alle* $\langle A, F\rangle \in M'$,
- $First_k(\beta) = First_k(\overline{\beta})$ *für alle* $\beta \in V'^*$ *und*
- *die transformierte Grammatik ist zur ursprünglichen äquivalent.*

Zur transformierten Grammatik bilden wir eine **Steuerfunktion** $T : M' \times \Sigma^k \to I\!N$ für die Alternativenauswahl, indem

$$T(\langle A, F\rangle, \alpha) = \begin{cases} i & \text{falls } \alpha \in Pre_k(First_k([A, i]) \bullet F) \\ 0 & \text{sonst} \end{cases}$$

Beispiel für Steuerfunktion zur transformierten Grammatik

Der **syntax-gesteuerte LL(k)-Akzeptor** ist nun durch folgende Takte beschrieben,

$$(1) \qquad (\gamma\langle A, F\rangle, \varphi : \kappa) \quad \mapsto \quad (\gamma\overleftarrow{[\langle A, F\rangle, i]}, \varphi : \kappa(A, i))$$
$$\text{falls } \langle A, F\rangle \in M' \wedge T(\langle A, F\rangle, pre_k(\varphi)) = i > 0$$

$$(2) \qquad (\gamma a, a\varphi : \kappa) \quad \mapsto \quad (\gamma, \varphi : \kappa)$$
$$\text{falls } a \in \Sigma$$

Hinweis: Der Akzeptor startet mit der Situation $(\langle S, \{\vdash^k\}\rangle, \varphi : \varepsilon)$.

Die zur transformierten Grammatik gehörende Steuerfunktion kann (das ist oft so) für zwei Metasymbole $\langle A, i\rangle, \langle A, j\rangle$ mit $i \neq j$ stets gleiche Resultate liefern. Dann kann man diese Metasymbole zu einem Metasymbol $\langle A, ij\rangle$ zusammenfassen. Diese Reduktion hat keinerlei Konsequenzen.
Beispiel für Zusammenfassung von Metasymbolen mit gleichem Verhalten

Weiter kann es auftreten, daß die Steuerfunktion für zwei Metasymbole $\langle A, i\rangle, \langle A, j\rangle$ mit $i \neq j$ zwar verschiedene Resultate liefert, von denen aber eines immer 0 (Fehlerzustand) ist (ähnliches Verhalten). Dann kann man diese Metasymbole auch zu einem Metasymbol $\langle A, ij\rangle$ zusammenfassen, wobei die 0-Resultate überschrieben werden. Die Konsequenz aus dieser Reduktion ist, daß Fehler in der Eingabe erst später erkannt werden können. Man kann weiterhin partiell auch die Anzahl der look-ahead-Symbole reduzieren, wenn das für die Steuerfunktion keine Konflikte bringt.

Beispiel für Zusammenfassung von Metasymbolen mit ähnlichem Verhalten

Lemma 3.4.15 *Wenn in der transformierten Grammatik*

$$\langle S', \{\varepsilon\}\rangle \rightarrow^* \varphi\langle A, F\rangle\beta$$

gilt, so ist

$$F = Follow_k(\langle A, F\rangle).$$

Beweis: (induktiv über die Länge der Ableitung):
Induktionsanfang: Länge der Ableitung ist 0, also

$$\langle S', \{\varepsilon\}\rangle \rightarrow^* \langle S', \{\varepsilon\}\rangle \qquad \text{und} \qquad Follow_k(\langle S', \{\varepsilon\}\rangle) = \{\varepsilon\}.$$

Induktionsschritt: Es sei

$$\langle S', \{\varepsilon\}\rangle \rightarrow^* \varphi\langle A, F\rangle\beta \rightarrow \varphi\varphi'\langle A_i, F_i\rangle\beta'\beta,$$

also

$$\langle A, F\rangle \rightarrow \varphi'\langle A_i, F_i\rangle\beta'$$

und nach Induktionsannahme $F = Follow_k(\langle A, F\rangle)$.
Dann gilt entsprechend der Transformation der Grammatik :

$$F_i = Pre_k(First_k(\overline{\beta'}) \bullet F)$$

und dem Lemma 3.4.4:

$$\begin{aligned} Follow_k(\langle A_i, F_i\rangle) &= \bigcup\{Pre_k(First_k(\beta') \bullet Follow_k(\langle A, F\rangle) \mid \langle A, F\rangle \rightarrow \varphi'\langle A_i, F_i\rangle\beta'\} \\ &= \bigcup\{Pre_k(First_k(\beta') \bullet F) \mid \langle A, F\rangle \rightarrow \varphi'\langle A_i, F_i\rangle\beta'\} \end{aligned}$$

und nach Folgerung 3.4.14

$$\begin{aligned} &= \bigcup\{Pre_k(First_k(\overline{\beta'}) \bullet F) \mid \langle A, F\rangle \rightarrow \varphi'\langle A_i, F_i\rangle\beta'\} \\ &= \bigcup\{F_i \mid \langle A, F\rangle \rightarrow \varphi'\langle A_i, F_i\rangle\beta'\} \\ &= F_i. \end{aligned}$$

Folgerung 3.4.16 *Es ergibt sich:*

- *Der LL(k)-Akzeptor ist ein LF(k)-Akzeptor zur transformierten Grammatik,*
- *zu jeder LL(k)-Grammatik existiert eine äquivalente LF(k)-Grammatik und*
- *jede mit einem LL(k)-Verfahren analysierbare Sprache ist auch mit einem LF(k)-Verfahren analysierbar.*
- *Es gibt Sprachen, die nicht mittels LL(k)-Verfahren analysierbar sind.*

Satz 3.4.17 *Für jedes $k > 0$ existiert eine Sprache, für die eine LL(k) (LF(k)) Grammatik, aber keine LL(k-1) (LF(k-1)) Grammatik existiert.*

Beweis: (Idee)
Wir setzen $L_k := \{a^n \varphi^n \mid n \in I\!N \wedge \varphi \in \{b, b^k c\}\}.$

Eine zugehörige Grammatik ist:

$$\begin{array}{lcl} S & ::= & a\,S\,A \mid \varepsilon \\ A & ::= & b \mid b^k c \end{array}$$

Faktorisierung ergibt für A:

$$\begin{array}{lcl} A & ::= & b\,B \\ B & ::= & b^{k-1} c \mid \varepsilon \end{array}$$

Insgesamt kann A eliminiert werden und man erhält:

$$\begin{array}{lcl} S & ::= & a\,S\,b\,B \mid \varepsilon \\ B & ::= & b^{k-1} c \mid \varepsilon \end{array}$$

Das ist eine LL(k) Grammatik, denn:

$$\begin{array}{lcll} First_k(B) & = & \{\varepsilon, b^{k-1}c\} & \\ \mathcal{F}_k(S) & = & First_k(bB) \otimes_k \mathcal{F}_k(S) \;\cup\; \{\{\vdash^k\}\} & \\ & = & \{b, b^k\} \otimes_k \mathcal{F}_k(S) \;\cup\; \{\{\vdash^k\}\} & \\ & = & \{\{b^i \vdash^{k-i}, b^k\} \mid 1 \le i \le k\} \;\cup\; \{\{\vdash^k\}\} & \text{Iteration ergibt:} \\ \mathcal{F}_k(B) & = & \mathcal{F}_k(S) & \end{array}$$

Für die beiden Alternativen von B gilt:

$$\begin{array}{lcll} \{b^{k-1}c\} \cap \{b^i \vdash^{k-i}, b^k\} & = & \emptyset & \text{für jedes } i \text{ mit } 1 \le i \le k \\ \{b^{k-1}c\} \cap \{\vdash^k\} & = & \emptyset & \text{bzw.:} \end{array}$$

■

Zusammenfassung der Schritte für die Entwicklung eines LL(k)-Parsers:

1. Berechnung von $First_k$,
2. Berechnung von $\mathcal{F}_k(A)$ für alle Metasymbole A,
3. Berechnung der transformierten Grammatik,
4. Berechnung der Steuertabelle T
   (falls diese eindeutig ist, ist die LL(k)-Eigenschaft gesichert).
5. Reduktion von Steuertabelle und Grammatik (falls gewünscht).

Bemerkung: Die Schritte 2,3 und 4 können verzahnt abgearbeitet werden!

3.4.3 Das LR(k)-Verfahren

Das LR(k)-Verfahren nutzt die bottom-up-Strategie und entscheidet auf der Grundlage der k look-ahead-Symbole *und* des Kellerinhaltes, welche Alternative in einem reduce-Takt zu wählen ist.

Zur Erinnerung die Taktarten des bottom-up Akzeptors:

$$\begin{array}{lllll} (1) & (\beta[A,i], \varphi : \kappa) & \mapsto & (\beta A, \varphi : \kappa(A,i)) & \text{falls } A \in M \\ (2) & (\beta, a\varphi : \kappa) & \mapsto & (\beta a, \varphi : \kappa) & \text{falls } a \in \Sigma \end{array}$$

Definition 3.4.3 *Eine kontextfreie Grammatik hat die LR(k)-Eigenschaft (kurz: ist eine LR(k)-Grammatik), falls*

$$\text{aus } S \begin{cases} \overset{*}{\rightarrow} \beta A\varphi \rightarrow \beta\alpha\varphi \\ \overset{*}{\rightarrow} \beta' A'\psi'' \rightarrow \beta\alpha\psi'\psi'' \end{cases} \quad \text{und } pre_k(\varphi) = pre_k(\psi'\psi'') \text{ folgt, daß } A = A',\ \beta = \beta' \text{ und } \psi' = \varepsilon;$$

wobei: $S, A, A' \in M,\ \alpha, \beta, \beta' \in V^*, \varphi, \psi, \psi', \psi'' \in \Sigma^*$.

Folgerung 3.4.18 *Jede LR(k)-Grammatik ist eindeutig.*

Definition 3.4.4 *Zu einer gegebenen kontextfreien Grammatik und einem $k \in \mathbb{N}$ sei eine Funktion $\zeta : V^* \rightarrow \wp(M \times \mathbb{N} \times \mathbb{N} \times \Sigma^k)$ definiert durch:*

$$\zeta(\beta\alpha) := \{(A,i,j,\gamma) \mid S \rightarrow^* \beta A\varphi \wedge \alpha \sqsubseteq [A,i] \wedge j = |\alpha| \wedge \gamma \in First_k(\varphi)\}.$$

Folgerung 3.4.19 *Der bottom-up-Akzeptor kann ausführen*

- *den Takt (1) genau dann, wenn* $(A,i,j,\gamma) \in \zeta(\beta[A,i]) \wedge j = |[A,i]| \wedge \gamma = pre_k(\varphi)$ *und*
- *den Takt (2) genau dann, wenn* $(A,i,j,\gamma) \in \zeta(\beta) \wedge \beta = \beta'\alpha \wedge \alpha\alpha' = [A,i] \wedge j = |\alpha| < |[A,i]| \wedge pre_k(\varphi) \in First_k(\alpha'\gamma)$.

Beweis: Klar nach Definition des bottom-up-Akzeptors. ■

Satz 3.4.20 *Eine Grammatik hat genau dann die LR(k)-Eigenschaft, wenn*

$$\begin{aligned}\forall\chi: \quad & S \to^* \chi\varphi \\ & \wedge\ (A,i,j,\gamma) \in \zeta(\chi) \wedge j = |[A,i]| && \textit{(B1)} \\ & \wedge\ (A',i',j',\gamma') \in \zeta(\chi) \wedge \alpha\alpha' = [A',i'] \wedge j' = |\alpha| \wedge \gamma \in First_k(\alpha'\gamma') && \textit{(B2)} \\ & \Longrightarrow\ (A,i,j,\gamma) = (A',i',j',\gamma').\end{aligned}$$

Beweis:
Angenommen, es ist $(A,i,j,\gamma) \neq (A',i',j',\gamma')$.
Wenn $A = A' \wedge i = i' \wedge j = j'$ folgt wegen $\gamma \in First_k(\alpha'\gamma') = First_k(\gamma') = \{\gamma'\}$, daß $\gamma = \gamma'$!
Also ergibt sich aus der Annahme: $A \neq A' \vee i \neq i' \vee j \neq j'$.
Die Teilkonjunktion (B1) ermöglicht Takt (1), die Teilkonjunktion (B2) ermöglicht Takt (2).
Wenn nun $A \neq A' \vee i \neq i' \vee j \neq j'$, so könnte Takt (1) auf verschiedene Weise ablaufen oder Takt (1) und Takt (2) wären beide möglich. Das ist ein Widerspruch! ■

Es ist praktisch nicht möglich, für jeden Kellerinhalt χ das Resultat $\zeta(\chi)$ zu berechnen. Die Idee des LR(k)-Verfahrens besteht wieder darin, daß man begleitend in jedem Takt an den Keller $\zeta(\chi)$ anhängt. Zu diesem Zweck berechnet man die **endliche** Menge

$$Z := \{\zeta(\chi) \mid S \to^* \chi\varphi\}$$

vorab.

Für $z \in Z$ ergeben sich folgende

Lemma 3.4.21 *Eigenschaften:*

1. *Falls* $(A,i,j,\gamma) \in z \wedge \alpha A'\alpha' = [A,i] \wedge j = |\alpha|$,
   dann gilt $(A',i',0,\gamma') \in z$ *wobei* $\gamma' \in First_k(\alpha'\gamma) \wedge 1 \leq i' \leq |[A']|$.

2. *Falls* $z = \zeta(\chi) \wedge (A,i,j,\gamma) \in z \wedge j < |[A,i]|$,
   dann gilt $(A,i,j+1,\gamma) \in z' = \zeta(\chi[A,i,j+1])$.

Beweis:
1. ergibt sich aus der Existenz einer Ableitung $S \to^* \beta A\varphi \to \beta\alpha A'\alpha'\varphi \to \beta\alpha\varepsilon\delta\alpha'\varphi$,
wobei $\chi = \beta\alpha$ und $[A',i'] = \delta$ zu setzen ist.
2. ergibt sich direkt aus der Definition von z. ■

Um diese Eigenschaften abzusichern, werden folgende **Operationen** eingeführt:

1. $\xi : \wp(M \times I\!N \times I\!N \times \Sigma^k) \to \wp(M \times I\!N \times I\!N \times \Sigma^k)$ mit
   $\xi(z) := z \cup \{(A',i',0,\gamma') \mid (A,i,j,\gamma) \in z \wedge \alpha A'\alpha' = [A,i] \wedge j = |\alpha| \wedge$
   $\gamma' \in First_k(\alpha'\gamma) \wedge 1 \leq i' \leq |[A']|\}$

2. $\delta : Z \times V \to Z$ mit
   $\delta(z,[A,i,j+1]) := \xi(\{(A,i,j+1,\gamma) \mid (A,i,j,\gamma) \in z \wedge j < |[A,i]|\})$.

Folgerung 3.4.22 *Z ist die kleinste Menge, für die gilt:*

$$Z = \{z_o\} \cup \{\delta(z,x) \mid z \in Z \wedge x \in V\},$$

wobei $z_0 = \xi(\{(S',1,0,\varepsilon)\})$.

Damit ist ein Berechnungsverfahren für die Menge Z gegeben!
Beispiel für Berechnung der Menge Z

Zur Menge Z berechnet man nun eine **Steuerfunktion** $T : Z \times \Sigma^k \to ((M \times I\!N \times I\!N) \cup \{\downarrow, \perp\})$ mit

$$T(z,\gamma) := \begin{cases} (A,i,j) & \text{falls } (A,i,j,\gamma) \in z \wedge j = |[A,i]| \\ \downarrow & \text{falls } (A,i,j,\gamma') \in z \wedge \alpha\alpha' = [A,i] \wedge j = |\alpha| < |\alpha\alpha'| \wedge \gamma \in First_k(\alpha'\gamma') \\ \perp & \text{sonst} \end{cases}$$

Beispiel für die Steuerfunktion

Der **syntax-gesteuerte LR(k)-Akzeptor** hat als Kellerelemente Paare $(x,z) \in V \times Z$ und ist durch die folgenden Takte beschrieben:

$$\begin{array}{lll} (1) & (\beta(x,z)\beta'(x',z'),\varphi:\kappa) & \mapsto \quad (\beta(x,z)(A,z''),\varphi:\kappa(A,i)) \\ & \text{falls } T(z',pre_k(\varphi)) = (A,i,|\beta'|+1) \wedge z'' = \delta(z,A) & \\ (2) & (\beta(x,z),a\varphi:\kappa) & \mapsto \quad (\beta(x,z)(a,z''),\varphi:\kappa) \\ & \text{falls } T(z',pre_k(a\varphi)) = \downarrow \wedge z'' = \delta(z,a) & \end{array}$$

Hinweis: Es gilt

$$\varphi \in L$$

genau dann, wenn

$$((S',z_0),\varphi \vdash^k,\varepsilon) \mapsto^* ((S',z_0)(S,z_1),\vdash^k,\kappa),$$

wobei $z_1 = \delta(z_o,S)$ und κ der Syntaxbaum in Postfixlinearisierung ist.

Beispiel für die LR(1) - Analyse

Nachfolgende Eigenschaften sind wichtig, werden aber nur ohne Beweis angegeben.

Satz 3.4.23 *Jede mit einem deterministischen PDA analysierbare Sprache hat eine LR(1) Grammatik.*

Satz 3.4.24 *Es ist nicht entscheidbar, ob eine Sprache mit einem deterministischen PDA analysierbar ist.*

3.4.4 Vereinfachung: LALR und SLR

Das **LALR-Verfahren** (genutzt bei yacc) arbeitet analog zum LR-Verfahren, reduziert aber die Menge Z.

Wir definieren:

- $Kern(z) := \{(A, i, j) \mid \exists \gamma : (A, i, j, \gamma) \in z\}$,
- $[z] := \bigcup\{z' \mid Kern(z) = Kern(z')\},\ Z' := \{[z] \mid z \in Z\}$,
- $\delta'([z], x) := [z']$, falls $\exists \overline{z} \in [z] : z' = \delta(\overline{z}, x)$.

Die Steuerfunktion ist wie für das LR(k)-Verfahren definiert.

Beispiel für Reduktion der Zustände und der Steuerfunktion

Das **SLR-Verfahren** arbeitet analog zum LR-Verfahren, verzichtet aber weitgehend auf look-ahead!

Die Menge $\zeta(\beta\alpha)$ ist jetzt definiert als

$$\zeta(\beta\alpha) := \{(A, i, j) \mid S \to^* \beta A\varphi \to \beta\alpha\alpha'\varphi \wedge \alpha\alpha' = [A, i] \wedge j = |\alpha|\}.$$

Für die Funktionen ξ und δ ergibt sich jetzt:

- $\xi(z) := z \cup \{(A', i', 0) \mid (A, i, j) \in z \wedge \alpha A'\alpha' = [A, i] \wedge j = |\alpha| \wedge 1 \leq i' \leq |[A']|\}$
- $\delta(z, [A, i, j+1]) := \xi(\{(A, i, j+1) \mid (A, i, j) \in z \wedge j < |[A, i]|\})$.

In der Steuerfunktion ist jetzt ein look-ahead-Symbol zu berücksichtigen:

$$T(z, a) := \begin{cases} (A, i, j) & \text{falls } (A, i, j) \in z \wedge j = |[A, i]| \wedge a \in Follow_1(A) \\ \downarrow & \text{falls } (A, i, j) \in z \wedge j < |[A, i]| \wedge a = [A, i, j+1] \\ \bot & \text{sonst} \end{cases}$$

Beispiel für andere Reduktion der Zustände und der Steuerfunktion

Kapitel 4

Kontextprüfung und Codeerzeugung

4.1 Das Grundprinzip

Sowohl bei der Kontextprüfung als auch bei der Codeerzeugung hat man (im Prinzip) eine entsprechende Algebra zu entwickeln.

Dazu ist

- ein System von Trägermengen festzulegen und
- jedes Funktionssymbol durch eine Funktion zu interpretieren.

Die Steuerung des Gesamtablaufs ergibt sich aus den Homomorphismen *context* bzw. syn_Z, d.h. aus der Interpretation des abstrakten Syntaxbaumes. Liegt dieser in Postfixlinearisierung vor, werden die entsprechenden Funktionen der Reihe nach aktiviert.

Das theoretische Konzept der Homomorphismen geht allerdings davon aus, daß alle Operationen der Kontextalgebra beziehungsweise der Zieltextalgebra $\underline{Z}$ Funktionen im mathematischem Sinne, d.h. auch ohne jeglichen Seiteneffekte, sind. Das könnte man - theoretisch - auch so implementieren. Dabei wären alle Informationen zwischen den Funktionen über Parameter zu vermitteln, was einen hohen Organisationsaufwand bedeutet.

Um den praktischen Bedürfnissen Rechnung zu tragen, geht man vom theoretisch sauberen Konzept der Funktionen ab und erlaubt, globale Objekte zu führen, deren Werte durch die den Funktionssymbolen zugeordneten *Programmfunktionen* modifiziert werden. Zu diesen globalen Objekten gehören standardmäßig:

- **Namensliste**, in die Bezeichnungen von Variablen und Typinformationen eingetragen werden. Der Zugriff besteht in Funktionen zum Eintrag, zur Abfrage und zum Löschen von Informationen. Die Art und Struktur der Einträge hängt stark von der Quellsprache ab. Bei Sprachen mit Blockstrukur ist zu berücksichtigen, daß die Namensliste mehrfache Einträge erlauben muß, um lokale Variable behandeln zu können. Für die Behandlung von Beispielen beschränken wir uns auf die drei Funktionen
  - `insert(bezeichnung,typ)` - zum Eintrag einer Bezeichnung mit einer Typinformation,
  - `index(bezeichnung)` - zur Bestimmung des Indizes des Eintrages der Bezeichnung, ist die Bezeichnung nicht in der Namensliste, so soll 0 zurückgegeben werden.
  - `type(index)` - gibt die Typinformation zum Eintrag mit dem entsprechenden Index zurück.

- **Kellerspeicher**, hier **Wertekeller** genannt, in den Zwischeninformationen eingetragen werden, die entsprechend der rekursiven Struktur des Syntaxbaumes zu verwalten sind. Folgende Funktionen sollen für den Zugriff auf den Kellerspeicher vorhanden sein:
  - `push(x)` - kellert `x` ein.
  - `top(i)` - liefert den Inhalt der Kellerzelle mit der relativen Adresse `i` (von der Kellerspitze aus betrachtet).
  - `pop(i)` - löscht die `i`-obersten Zellen aus dem Keller.

4.2 Kontextprüfung

Die wesentlichen Kontextbedingungen ergeben sich aus

- dem **Vereinbarungszwang**, d.h. daß benutzte Objekte vorher zu vereinbaren sind, und
- dem **Typkonzept**, das Rechte auf den Zugriff von Objekten oder deren Komponenten festlegt.

Um zu prüfen, ob ein genutztes Objekt vereinbart ist, wird bei der Vereinbarung ein Eintrag der Objektidentifikation (im allgemeinen der Bezeichnung) in die Namensliste vorgenommen und bei der Nutzung geprüft, ob ein solcher Eintrag vorliegt.

Beispiel für eine einfache Kontextprüfung bei PRILAN

Die Überprüfung der Einhaltung des Typkonzeptes ist komplizierter und hängt davon ab, wie stark das Typkonzept in der Quellsprache ausgebaut ist. Minimal ist zu prüfen, ob die Typen von aktuellen und formalen Parametern kompatibel sind. Da Operatoren auch Funktionen sind, so wird damit auch die Typkompatibilität von Operanden in Ausdrücken abgefangen. Wegen der Überladung von Funktionen muß aus den Typen der aktuellen Parametern (der Operanden) auch der konkrete Funktionsaufruf abgeleitet werden.
Die Typen der einzelnen Parameter (Operanden) werden im Wertekeller abgelegt, nach erfolgter Prüfung der Typkompatibilität wieder ausgekellert und dafür der Resultattyp eingekellert.

Beispiel für die Kontextprüfung bei BALAN

Am aufwendigsten ist die Typbehandlung bei funktionalen Sprachen mit polymorphen Typen (Haskell, Miranda,...). Hier sind Unifikationsalgorithmen einzusetzen, um den allgemeinsten Typ eines Objektes zu bestimmen. Im Rahmen dieser Vorlesung wird darauf nicht eingegangen.

4.3 Codeerzeugung

Wie bereits im Abschnitt 1.4.1 erwähnt, kann man die Codeerzeugung in einem Schritt ausführen oder aber auch in mehrere Schritte zerlegen. Letzteres führt dazu, daß man zunächst in eine **Zwischensprache** übersetzt, aus der dann, vielleicht wieder in mehreren Schritten, der Zielcode erzeugt wird.

4.3.1 Erzeugung einer Zwischensprache

Die Erzeugung einer Zwischensprache hat folgende Vorteile:

- Man kann eine technologisch saubere Trennung verschiedener Aufgaben vornehmen.
- Man hat eine gute Basis für die Zielprogrammoptimierung.
- Man hat eine gute Basis für die Erzeugung unterschiedlichen Zielcodes (*Retargierbarkeit* genannt) .
- Man hat eine gute Basis für die interpretative Abarbeitung.
- Man hat eine gute Basis für die Codeerzeugung während des Programmladens (*just in time compilation* genannt), einer Technik, die immer stärker zum Einsatz kommt.

Die Struktur der Zwischensprache kann sich

- an eine textuelle Repräsentation mehr oder weniger starkt anlehnen, was die Lesbarkeit erhöht oder
- eine interne Datenstruktur sein.

Beispiele für textuelle Repräsentationen sind die bei Pascal, Java und .NET genutzten **Bytecodes**. Aber auch abstraktere Zwischensprachen werden insbesondere dann eingesetzt, wenn man eine umfangreiche Programmoptimierung anschließen möchte. Solche Zwischensprachen bestehen meist aus

- Dreiadreß-Befehlen der Form
  - `x:=y`,
  - `x:=u y`,
  - `x:=y o z`,

  wobei `x,y,z` Variable sind, `u` ein unärer und `o` einen binären Operator repräsentiert.
- Befehlen zur Adreßbildung und
- Befehlen zur Programmverzweigung (Sprünge, Funktionsaufrufe,...).

Wichtig dabei ist:

- In der Zwischensprache ist die Typbehandlung weitgehend abgeschlossen, d.h.
  - für die Variablen wird nur eine Adreßinformation und die Länge des benötigten Speicherbereiches angegeben und
  - die Überladung von Funktionen und Operatoren ist aufgelöst.
- Variable werden unterschieden in:
  - **Programmvariable**, die aus dem Quelltext übernommen sind, bei denen aber Mehrdeutigkeiten in der Bezeichnung (aus dem Blockkonzept resultierend) aufgelöst sind und
  - **temporäre Variable**, die in der Zwischensprache neu eingeführt sind, der Aufnahme von Zwischenresultaten dienen und die im Zielcode vorrangig in Registern zu führen sind.
- Es treten keine expliziten Literale auf, diese werden über Konstantenbezeichnungen realisiert, die in der Zwischensprache wie Programmvariable behandelt, aber nur lesend verwendet werden.
- Die Speicherplanung ist vorgenommen, wobei
  - Datenstrukturen und mehrdimensionale Felder aufgelöst sind, so daß qualifizierte Zugriffe auf Komponenten von Datenstrukturen und Indexrechnungen jetzt explizit über spezielle Adreßoperatoren realisiert werden,
  - Referenzierung und Entreferenzierung über spezielle Operatoren realisiert werden und
  - die Blockstruktur aufgelöst ist, so daß jede Programmvariable eine relative Adresse zum Anfang ihres Blockspeicherbereiches (Siehe auch 4.4) hat.

Die Auflösung der Blockstruktur und der damit verbundenen eindeutigen Bezeichnung von Programmvariablen hat die Konsequenz, daß die ursprünglichen Bezeichnungen der Variablen im Quellprogramm verändert werden. Für die Fehlerbehandlung oder auch für Debugger-Programme ist es notwendig, eine Bezugnahme zur ursprünglichen Bezeichnung zu sichern. Das kann durch das Führen einer Namensliste erfolgen, die in der Ebene der Zwischensprache verfügbar ist. In dieser Namensliste sind alle erforderlichen Informationen über die Programmvariablen abgelegt, für Konstantenbezeichner auch der Wert.

Je nach der Struktur der Quellsprache kann es erforderlich werden, auch die Zwischensprache in mehreren Schritte zu erzeugen.

Beispiel für die Erzeugung der Zwischensprache für BALAN-Ausdrücke

4.3.2 Befehlsplanung und Registerzuordnung

Der Zielcode besteht aus einer Folge von Maschinenbefehlen, die mehr oder weniger intensiv auf Register unterschiedlicher Funktion (zum Beispiel arithmetische Register zur Aufnahme von Operanden und Resultaten, Indexregister zur Aufnahme von Adressen) zugreifen. Bei der Befehlsplanung (in der Literatur oft als *Scheduling* bezeichnet) arbeitet man die Zwischensprache Befehl für Befehl ab und erzeugt dabei die entsprechenden Maschinenbefehle. Dabei sollte es die Zwischensprache erlauben, dies möglichst ohne Vorschau auf weitere Zwischensprachen-Befehle durchzuführen. Natürlich muß man unter Umständen, zum Beispiel bei Vorwärtssprüngen, die Erzeugung des Zielcodes verzögern. Man kann diesen Schritt aber auch dadurch vereinfachen, indem zunächst Assemblercode erzeugt wird. Je nach Funktionalität des verfügbaren Assemblers

kann damit die Berechung von Adressen von Programmvariablen und Einsprungstellen auf den Assembler verlagert werden.

Die Zuordnung von Registern zu den temporären Variablen der Zwischensprache erfolgt im allgemeinen verzahnt mit der Befehlsplanung, da die Verfügbarkeit von Registern von der bisher entstandenen Befehlsfolge abhängt. Stehen ausreichend Register zur Verfügung, so ist die Registerzuordnung unproblematisch. Anderenfalls kann es durch zu frühzeitige Entscheidung über die Registervergabe zu Engpässen kommen, die zusätzliche Transfer-Befehle zwischen Registern oder das Retten von Registerinhalten in Hilfsvariable (in der Literatur auch *Spillcode* genannt) erforderlich macht. Für Befehlsplanung und Registerzuordnung gibt es, je nach Architektur des Zielcodes, mittlerweile ein grosses Spektrum an ausgefeilten Verfahren, deren Einsatz sich aber nur lohnt, wenn man die Erzeugung hoch effizienten Zielcodes erreichen möchte. Wir behandeln zwei dieser Varianten in den Abschnitten 5.2.1 und 5.2.2.

An dieser Stelle soll ein einfaches Verfahren skizziert werden, das unter strikten Wahrung der durch die Zwischensprache festgelegten Abarbeitungsreihenfolge Maschinencode erzeugt. Dabei wird davon ausgegangen, daß

- der Maschinencode aus Zweiadreß-Befehlen besteht, dessen erste Adresse sowohl den ersten Operanden als auch das Resultat adressiert,
- alle arithmetischen Befehle nur auf Register zugreifen, ausschließlich Transferbefehle (Speichern,Laden) auf den Speicher Zugriff haben,
- es keine speziellen Maschinenbefehle für unäre Operatoren gibt,
- in der Zwischensprache nur bei Kopierbefehlen Programmvariable entweder auf
  - der linken (`a:=t`)
  - oder der rechten Seite (`t:=a`) auftreten,
- jede temporäre Variable in der Zwischensprache
  - genau einmal belegt (`t:=...`) und
  - höchstens einmal benutzt (`..:=...t...`) wird,
- eine temporäre Variable ein *Alias* einer Hilfsvariablen sein kann,
- zwischen temporären Variablen und Registern eine Zuordnung bestehen kann,
- ein Register *frei* heißt, wenn es keiner temporären Variablen zugeordnet ist,
- Kommutativität von Operatoren nicht ausgenutzt wird.

Um im Bedarfsfall ein Register `i` zu beschaffen, wird

- ein beliebiges freies Register genommen, falls ein solches existiert
- oder sonst
  - ein durch eine temporäre Variable `t` belegtes Register genommen, dessen Benutzung am weitesten zurückliegt (in der Hoffnung, daß es nicht so bald oder nie mehr benötigt wird),
  - Maschinencode zum Abspeichern des Inhalts von `i` in eine Hilfsvariable `h` geplant wird und
  - die temporäre Variable `t` als ein Alias von `h` gesetzt wird.

Die einzelnen Befehle der Zwischensprache bewirken nun folgende Aktionen:

- `a:=t`
  - Planung eines Maschinenbefehls zur Speicherung des Registers i, dem die temporäre Variable `t` zugeordnet ist, in die Programmvariable `a`.
  - Gebe das Register i frei.
- `t:=a`
  - Beschaffe ein Register `i`.
  - Ordne `i` der temporären Variablen `t` zu.
  - Plane einen Maschinenbefehl zum Laden der Programmvariablen `a` in das Register `i`.
- `t:= u t'`
  Das entspricht dem nächsten Fall, da der unäre Operator `u` über einen arithmetischen Befehl mit binärem Operator realisiert werden muß.
  Aber: ein Register `i` muß der temporären Variablen `t'` zugeordnet sein, denn ein unärer Operator hat höchste Priorität.
- `t:= t' o t''`
  Es ist zu unterscheiden:
  - Wenn die temporäre Variable `t'` ein Alias einer Hilfsvariablen `h` ist, dann
    - beschaffe ein Register `i`,
    - plane einen einen Maschinenbefehl zum Laden der Hilfsvariablen `h` in das Register `i` und
    - ordne das Register `i` der temporären Variablen `t'` zu;
  - Wenn `t''` ein Alias einer Hilfsvariablen `h` ist, dann
    - beschaffe ein Register `j`,
    - plane einen einen Maschinenbefehl zum Laden der Hilfsvariablen `h` in das Register `i`
    - ordne das Register `j` der temporären Variablen `t''` zu.
  - Jetzt sei das Register `i` der temporären Variablen `t'` und `j` der temporären Variablen `t''` zugeordnet und demzufolge
    - plane einen Maschinenbefehl zur Verknüpfung der Inhalte der Register `i` und `j` mittels der binären Operation `o` und Speicherung des Resultats in das Register `i`,
    - ordne das Register `i` jetzt der temporären Variablen `t` zu und
    - gebe das Register `j` frei.

In dieses Verfahren kann man durch weitere Fallunterscheidungen noch Verbesserungen der Befehlsplanung und der Registerzuordnung vornehmen.

Beispiel für einfache Befehlsplanung und Registerzuordnung

4.4 Speicherverwaltung

Alle in einem Programm zu verwaltenden Objekte, d.h. Funktionen und Variable, müssen im Speicher abgelegt werden, d.h. für sie muß ein Speicherbereich reserviert werden und jedes Objekt muß eine entsprechende Adresse zugeordnet erhalten. Dabei versucht man, möglichst viele Vorberechnung zur Compilezeit, d.h. statisch, auszuführen, damit zur Laufzeit, d.h. dynamisch, nur die nicht statisch bestimmbaren Größen zu berechnen sind.

4.4.1 Behandlung von Funktionen

Alle Funktionen bzw. Methoden von Klassen werden beim Linkvorgang im allemeinen zu einem festen Segment zusammengefaßt. In diesem Segment sind die Adressen aller Funktionen fest verankert.

Besonderheiten:

- **Die späte Bindung** bei (virtuellen) Methoden von Klassen erfordert eine indirekte Bezugnahme. Der Aufruf einer virtuellen Methode führt über einen Pointer, der in der entsprechenden Instanz für jede virtuelle Funktion eingerichtet ist.
  Beispiel für die Wirkung der späten Bindung
- **Die just-in-time Compilation** (eine bei .NET verwendete Technologie) verlagert das backend des Übersetzungsvorganges und den Linkvorgang auf den erstmaligen Aufruf einer Funktion. Deshalb muß dann das Segment für Funktionen dynamisch organisiert werden.

4.4.2 Statische Verwaltung von Variablen

Die **Statische Verwaltung** beinhaltet vorbereitende Adreßberechnungen von Variablen und Komponenten von Datenstrukturen, im einzelnen sind das:

- Jedem Typ wird eine Längeninformation zugeordnet, das ist die maximale Größe des erforderlichen Speicherbereichs, um einen beliebigen Wert des Typs aufnehmen zu können.
  - Jeder Grundtyp hat eine feste Länge (in Abhängigkeit von der gewählten Plattform).
  - Zu jedem abgeleiteten Typ wird eine Beschreibung (*dope-Vektor* genannt), berechnet, auf deren Grundlage die Längeninformation sowie die relativen Adressen der Komponenten bestimmt werden können.
- Alle im Block lokal und als automatisch vereinbarten Objekte werden zu einem Blockspeicherbereich (BSB) zusammengefaßt und erhalten eine relative Adresse zum Anfang des BSB.

Zur Bestimmung der dope-Vektoren:

- bei Feldern (arrays) erfolgt in der Quellsprache
  - bei der Vereinbarung die Angabe
    - des Typs der einzelnen Komponenten,
    - die untere Grenze u_j und obere Grenze o_j für jede Dimension

    und
  - beim Zugriff für jede Dimension die Angabe eines Indizes i_j.

Wenn n die Dimension des Feldes und l_j die Länge des $n-j$-dimensionalen Teilfeldes ist, dann ist

- l_n die Länge einer Komponente und
- $l_{j-1} = (o_j - u_j + 1) * l_j$.

Für die Adresse einer Komponente ergibt sich dann

$Anfang + \sum_{j=1}^{n}(i_j - u_j) * l_j = Anfang - \sum_{j=1}^{n} u_j * l_j + \sum_{j=1}^{n} i_j * l_j = L + \sum_{j=1}^{n} i_j * l_j$, falls $L = Anfang - \sum_{j=1}^{n} u_j * l_j$.

Als dope-Vektor wählt man deshalb $(L, l_1, \ldots, l_n)$.
Bemerkung: Wenn wie bei C die untere Grenze immer 1 ist, vereinfacht sich die Berechnung des dope-Vektors entsprechend.

- bei Strukturen (structs, records) erfolgt in der Quellsprache
  - bei der Vereinbarung die Angabe
    - die Reihenfolge der Komponenten und
    - für jede Komponente deren Bezeichnung und Typ

    und
  - beim Zugriff die qualifizierte Bezeichnung der Komponente.

  Wenn A_j die Adresse der j-ten Komponente ist, so wählt man als dope-Vektor: $(A_1, \ldots, A_n)$.

4.4.3 Dynamische Verwaltung von Variablen

Die **Dynamische Verwaltung** beinhaltet die Bereitstellung von Speicherbereichen und die Komplettierung der Adreßberechnungen.

4.4.4 Der Stack

Im Stack (ein Kellerspeicher) werden alle Objekte verwaltet, die einer automatischen (entsprechend der Blockstruktur) Erzeugung und Vernichtung unterliegen. Da Blöcke hierarchisch geschachtelt sind, erfolgt die Erzeugung und Vernichtung solcher Objekte immer nach dem last-in-first-out (LIFO) Prinzip. Wir wollen hier davon ausgehen, dass die globale Programmumgebung auch als ein Block angesehen wird, der automatisch beim Programmstart betreten wird.

Es wird

- bei jedem Eintritt in einen Block der Blockspeicherbereich im Stack ausgefaßt und
- beim Austritt aus einem Block der Blockspeicherbereich im Stack freigegeben.

Achtung: Durch Sprünge (`goto,continue,return,break,...`) kann in einem Schritt aus mehreren Blöcken ausgetreten werden. In einem solchen Falle sind im Stack alle entsprechenden Blockspeicherbereiche freizugeben!

4.4.5 Der Heap

Der heap (Halde) dient dazu, die über Allokierungsoperatoren (`new`) erzeugten Objekte aufzunehmen. Dabei ist es im allgemeinen nicht vorhersehbar, in welcher Reihenfolge erzeugte Objekte wieder freigegeben werden.

Der heap ist in eine Folge von Bereichen eingeteilt, von denen jeder entweder mit

- **belegt** gekennzeichnet ist (dann gehört der Bereich zu einem Objekt, das aktuell noch zugreifbar ist und von dem angenommen wird, daß sein Wert noch benötigt wird) oder mit
- **frei** gekennzeichnet ist (dann kann über diesen Bereich frei verfügt werden).

Die heap-Verwaltung verläuft nun folgendermaßen:

- **Am Anfang** ist der gesamte heap ein mit frei markierter Bereich.
- **Bei einer Anforderung zum Allokieren von Speicher** wird der kleinste mit frei markierte Bereich im heap gesucht, der groß genug ist, um der Anforderung zu genügen.
- **Bei der Freigabe eines Bereiches** wird der entsprechende Bereich mit frei markiert. Wenn ein Nachbar dieses Bereiches auch frei ist, dann wird dieser mit dem aktuell freigegebenen Bereich verschmolzen.

Um die Verwaltung effizient zu gestalten, wird eine Liste aller mit frei markierten Bereiche geführt. Das kann etwa durch eine Verkettung aller dieser Bereiche erfolgen.

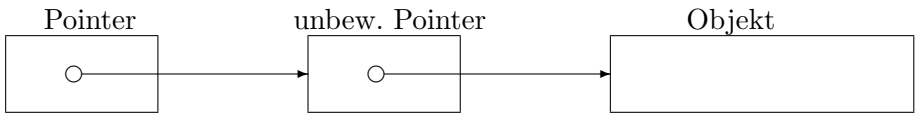
Ein **Problem** entsteht dann, wenn kein freier Bereich groß genug ist, um der Anforderung zu genügen.

Die Lösung entsteht durch die **Kompaktierung**: alle mit belegt markierten Bereiche werden an den Anfang des heap verschoben und der Rest des heap als ein einziger freier Bereich markiert.

Als **Konsequenz** wird eine (aufwendige) **Änderung aller Pointer** auf den heap erforderlich.

Eine **vereinfachende**, aber nicht allgemeingültige **Lösung** besteht darin, daß alle Pointer als indirekte Bezugnahme über einen unbeweglichen Pointer (UP) implementiert sind:

- zu jedem Objekt gehört genau ein unbeweglicher Pointer, dessen Wert auf das Objekt verweist und
- der eigentliche Pointer zeigt auf den unbeweglichen Pointer.



Der **Vorteil** besteht darin, daß bei einer Verschiebung des Objektes nur der Wert des einen unbeweglichen Pointers zu ändern ist, wobei jedes Objekt einen Rückverweis auf seinen unbeweglichen Pointer haben kann. Im Objekt enthaltene Pointer (das ist meist der Fall) brauchen nicht verändert zu werden.

Der **Nachteil** besteht in der indirekten Bezugnahme, was die Pointerarithmetik verbietet. Außerdem müssen die unbeweglichen Pointer in einem speziellen heap verwaltet werden.

Eine **aufwendigere**, aber vollständige **Lösung** muß für die direkte Bezugnahme bei Pointern gewählt werden. Zur Änderung der Pointer müssen alle Pointer besucht werden. Das erfordert ein **Tracing** durch die Pointerketten:

- Man startet mit allen Pointern, die sich im Stack befinden und
- durchsucht und verändert dann alle die Pointer, die sich in Objekten befinden, auf die besuchte Pointer verweisen.

Insgesamt ergeben sich zwei Hauptmethoden:

- **Mark-compact** erfordert drei Durchläufe:
  1. Berechnung und Ablegen von Verschiebeinformation: für jedes Objekt wird die Anzahl der Byte berechnet, um die das Objekt zu verschieben ist und diese Information wird mit ins Objekt aufgenommen.
  2. Tracing durch die Pointerketten und Aktualisierung der Werte.
  3. Verschiebung aller belegten Bereiche im heap.
- **Copying** erfordert nur zwei Durchläufe, indem zwei heaps, ein **from-heap** und ein **to-heap** geführt werden:
  1. Alle belegten Bereiche werden aus dem from-heap in den to-heap kopiert und die neue Adresse wird im Bereich des from-heap vermerkt.
  2. Tracing durch die Pointerketten und Aktualisierung der Werte im to-heap. Anschließend vertauschen from-heap und to-heap die Rollen.

Achtung: Wenn ein belegter Speicherbereich nicht explizit freigegeben wird, aber durch Pointerverlust nicht mehr referierbar ist, so entsteht **Müll** (garbage)!

Lösung durch **Garbage collection** hat zwei Aspekte:

1. **Müllerkennung**, das Objekte (heap-Bereiche) als lebendig markiert, falls diese referierbar sind, in zwei Varianten.
   - Der einfachen, aber nicht vollständigen Form der **Referenz-Zählung**:
     - Für jedes Objekt wird ein Zähler geführt, der die Anzahl der Referenzen angibt. Falls der Zähler größer Null, ist das Objekt lebendig.
     - Bei einer Veränderung eines Pointers (Überschreiben, Kopieren) wird der Zähler entsprechend verändert.
     - Wenn ein Objekt stirbt (Zähler geht auf Null), dann werden alle Zähler, die zu Pointern gehören, die Komponenten des Objektes sind, dekrementiert.

     Achtung: bei Zyklen in Pointerketten werden tote Objekte nicht erkannt.
   - **Mark-sweep** besteht aus
     - Tracing, wie oben beschrieben, wobei alle besuchten Objekte als lebendig markiert werden und
     - Sweeping, wobei alle nicht als lebendig markierten Objekte freigegeben werden.

     Bei Bedarf kann man ein Mark-compact anschließen.
2. **Müllsammlung**, d.h. der Kompaktierung des heap.

Eine Kombination von Müllerkennung und Müllsammlung erfolgt unter Nutzung von Copying und Tracing:

- Beim Tracing werden alle noch nicht kopierten lebendigen Objekte in den to-heap kopiert, dabei wird die neue Adresse im from-heap vermerkt und das Objekt als kopiert gekennzeichnet und die Pointer aktualisiert.
- Falls beim Tracing ein Pointer auf ein schon kopiertes Objekt zeigt, wird nur aktualisiert.

Kapitel 5

Optimierung des Zielcodes

5.1 Überblick

Be der Compilierung entsteht oft ein Verlust an Effizienz des Zielprogramms (gegenüber der Assemblerprogrammierung) bezüglich Speicherbedarf und Laufzeit. Die Gründe sind:

- Der Widerspruch zwischen dem problemorientierten Aufbau der Quellsprache und dem maschinenorientierten Aufbau der Zielsprache.
- Die oft komfortablen Formulierungsmöglichkeiten in den höheren Programmiersprachen verführen zur legeren Programmierung.

Achtung: Eine umfassende Optimierung des Zielcodes ist sehr aufwendig und lohnt sich nur bei

- zeitkritischen Anwendungen mit großen algorithmischen Anteilen und
- wenn Optimierungseffekte nicht durch andere Einflußfaktoren wie starke Interaktion mit dem Betriebssystem oder dem Nutzer oder auch schlechte Wahl der Algorithmen eliminiert werden.

Optimierungsmethoden kann man **einteilen** in

$$\text{Zielcode-abhängige (za)} \quad \Leftrightarrow \quad \text{Zielcode-unabhängige (zu)}$$

bzw. in

$$\text{lokale (l)} \quad \Leftrightarrow \quad \text{globale (g)}$$

Hauptprinzip ist die Optimierung auf der Grundlage einer geeigneten Zwischensprache und zwar durch:

- **Zielcode-abhängig:**
  ausgefeilte Techniken bei der Zielcodeerzeugung,
- **Zielcode-unabhängig:**
  Programmtransformation von der Zwischensprache in die Zwischensprache.

Hauptproblem ist die erforderliche umfassende **Informationssammlung** über das Programm (im wesentlichen über die Analyse des Datenflusses), die beim Aufbau des Syntaxbaumes und der Überprüfung von Kontextbedingungen partiell mit erledigt werden kann.

Konsequenz: wenn eine Optimierungsabsicht besteht, sollte auf die Zwischensprache mit zusätzlicher Information angereichert werden.

Hauptmethoden der Optimierung sind:

- Berechnung des Wertes konstanter Ausdrücke während der Compilierung (zu,l/g).
- Elimination redundanter Berechnungen (dead-code elimination, common-subexpression elimination) (zu,l/g).
- Verlagerung von Berechnungen in Programmteile, die weniger oft durchlaufen werden (code-motion) (zu,g).
- Transformation von iterativen Funktionsaufrufen (strength reduction) (zu,g).
- Optimale Registerbenutzung (za,l/g).
- Optimale Befehlsplanung (za,l).

5.2 Lokale Optimierung

Bei der lokalen Optimierung beschränkt man sich auf die Behandlung sogenannter Basisblöcke. Ein **Basisblock** ist eine Folge von Anweisungen (Befehlen), deren Abarbeitung immer am Beginn des Basisblockes begonnen wird (kein Einsprung in den Basisblock hinein) und am Ende beendet wird (kein vorzeitiger Aussprung). Basisblöcke bestehen deshalb aus Transferoperationen vom und zum Speicher sowie aus arithmetischen Operationen.

5.2.1 Zwischensprachen-gesteuert

Dabei wird die Zwischensprache im Basisblock Befehl für Befehl abgearbeitet.

Wir stellen eine einfache Methode vor, die relativ guten Zielcode erzeugt, wobei die Abarbeitungsreihenfolge der Zwischensprachenbefehle verändert wird.

Voraussetzungen:

- Die Ausführung der Befehle der Zwischensprache verursachen keinerlei Seiteneffekte.
- Temporäre Variable werden für die Zwischensprache nur erzeugt, wenn Zwischenresultate entstehen, für die in der Quellsprache keine Programmvariable vorgesehen sind.
  Beispiel für modifizierte Zwischensprache
- Zwischen Registern und Variablen (sowohl den Programmvariablen als auch den temporären Variablen) besteht eine Zuordnung.
- Ein Register ist **frei**, falls ihm keine Variablen zugeordnet sind.

Wir nennen eine Variable `v` an einem Befehl **lebendig**, falls hinter diesem Befehl `v` benutzt wird ohne dazwischen belegt zu werden.

Zur Beschreibung des Verfahrens werden einige Funktionen eingeführt:

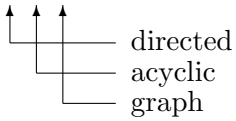
- `var(i)` ist die Menge der dem Register `i` zugeordneten Variablen.
- `ass(v,i)` ordnet die Variable `v` dem Register `i` zu.
- `minreg(v,w)` gibt ein Register zurück, dem nicht die Variable `w` zugeordnet ist und in dem außer der Variablen `v` die wenigsten Variablen zugeordnet sind.
- `fetch(v,w)` liefert ein Register `i` und bewirkt:
  wenn es ein freies Register `i` gibt,
  dann gebe `i` zurück
  und wenn ein Register `j` mit `v`∈`var(j)` existiert,
  dann plane einen Transferbefehl vom Register `j` zum Register `i`
  sonst plane einen Ladebefehl von der Variablen `v` zum Register `i`.
  sonst gebe ein Register `i=minreg(v,w)` zurück
  und plane für alle lebendigen Variablen `u` mit `u`∈`var(i)`, `u`≠ `v`, Speicherbefehle vom Register `i` zu den Variablen `u` (Spillcode)
  und wenn `v`∉`var(i)` dann plane einen Ladebefehl von der Variablen `v` nach dem Register `i`.

Die einzelnen Befehle der Zwischensprache bewirken nun folgende Aktionen:

- `x:=y`
  wenn `x` lebendig ist,
  dann wenn es ein Register `i` gibt mit `y`∈`var(i)`,
  dann führe `ass(x,i)` aus
  sonst sei `i=minreg(y,y)`
  und plane für alle Variable `v`∈`var(i)` einen Speicherbefehl von `i` nach `v`
  und führe `ass(x,i)` aus
  und wenn `y` lebendig ist, dann führe `ass(y,i)` aus
  und plane einen Ladebefehl von `y` nach `i`.
- `x:= u y`
  Das entspricht dem nächsten Fall, da der unäre Operator `u` über einen arithmetischen Befehl mit binärem Operator realisiert werden muß.
- `x:= y o z`
  wenn `x` lebendig ist,
  dann wenn es ein Register `i` gibt mit {`y`}=`var(i)` und (`y` ist nicht lebendig oder `x`=`y`)
  dann nehme `i`
  sonst nehme das `i=fetch(y,y)` und `ass(y,i)`
  und
  wenn es ein Register `j` gibt mit `z`∈`var(j)` und (`z` ist nicht lebendig oder `i`≠`j`)
  dann nehme `j`
  sonst nehme das `j=fetch(z,y)` und `ass(z,j)`
  und plane einen Maschinenbefehl zur Verknüpfung der Inhalte der Register `i` und `j` mittels der binären Operation `o` und Speicherung des Resultats in das Register `i`
  und setze `var(i)`={`x`}
  und wenn `y` nicht lebendig ist dann löse alle Zuordnungen von `y` zu Registern
  und wenn `z` nicht lebendig ist dann löse alle Zuordnungen von `z` zu Registern

Beispiel für optimierten Zielcode

5.2.2 DAG - gesteuert



Zur Definition eines DAG siehe das GTI-Skript, Abschnitt 1.7.

Das Verfahren hat folgende Schritte:

1. Erzeugung eines DAG aus dem Basisblock.
   Erfreuliche Konsequenz: alle redundanten Berechnungen aus dem Basisblock werden eliminiert!
2. Registerplanung mit virtuellem Registersatz.
3. Zuordnung von virtuellen zu realen Registern, dabei, falls erforderlich
   - Einfügen zusätzlicher Hilfskanten in den DAG und
   - Zwischenspeicherung von Registern (Spillcode).
4. Befehlsplanung.

5.2.2.1 Erzeugung des DAG

Der DAG kann

- entweder direkt aus dem Syntaxbaum
- oder aus der Zwischensprache

erzeugt werden. Hier wird letztere Variante vorgeführt.

Folgende **Knotenarten** gibt es im DAG:

- **Load-Knoten**, die **keine Operanden** haben.
- **Store-Knoten**, die **einen Operanden k1** (den Knoten k1) haben.
- **u-Knoten**, die **einen Operanden k1** (den Knoten k1) und den unären Operator **u** haben.
- **o-Knoten**, die **zwei Operanden k1,k2** (die Knoten k1,k2) und den binären Operator **o** haben.

Folgende **Kantenarten** gibt es:

- **Operandenkanten**, die von einem Quell-Knoten auf einen Operanden eines Ziel-Knotens verweisen. Diese Kanten repräsentieren das Resultat des Quell-Knotens, das als ein Operand des Ziel-Knotens verwendet wird. Eine Operandenkante ist durch die Angabe des Operanden in einem Store-, u- oder o-Knoten bereits festgelegt.
- **Hilfskanten**, die zusätzliche Bedingungen für die Reihenfolge der Auswertung der Knoten festlegen.

Zur Erzeugung des DAG wird eine Liste mit Einträgen der Form (v, i, k) geführt, wobei

- v eine Variable,
- $i \in \{L, S\}$ (L für Load, S für Store),
- k ein Knoten ist.

Folgende Funktionen werden benutzt:

- `update(v,i,k)` - modifiziert die Liste, indem
  wenn ein Eintrag $(v, \_, \_)$ in der Liste existiert,
  dann lösche diesen Eintrag
  und füge den Eintrag (v, i, k) in die Liste ein.
- `getnode(v)` - liefert einen Knoten k, indem
  wenn ein Eintrag $(v, \_, k)$ in der Liste existiert,
  dann nehme das `k`,
  sonst nehme `k=insertload(v)` und füge (v, L, k) in die Liste ein.
- `insertload(v)` fügt den Load-Knoten $\mathbf{k} : \mathbf{v} \rightarrow$ zum DAG hinzu und gibt k zurück.
- `insertu(k1,u)` fügt eventuell einen u-Knoten zum DAG hinzu und gibt diesen zurück, indem
  wenn ein u-Knoten $\mathbf{k} : \mathbf{u\ k1}$ existiert,
  dann nehme `k`,
  sonst bilde einen neuen u-Knoten $\mathbf{k} : \mathbf{u\ k1}$ und nehme `k`.
- `inserto(k1,k2,u)` fügt eventuell einen o-Knoten zum DAG hinzu und gibt diesen zurück, indem
  wenn ein o-Knoten $\mathbf{k} : \mathbf{k1\ o\ k2}$ existiert,
  dann nehme `k`,
  sonst bilde einen neuen u-Knoten $\mathbf{k} : \mathbf{k1\ o\ k2}$ und nehme `k`.

Die einzelnen Befehle der Zwischensprache bewirken nun folgende Aktionen:

- `x:=y`
  nehme `k=getnode(y)`
  und `update(x,S,k)`.
- `x:= u y`
  nehme `k1=getnode(y)` und nehme `k=insertu(k1,u)`
  und `update(x,S,k)`.
- `x:= y o z`
  nehme `k1=getnode(y)` und nehme `k2=getnode(z)` und nehme `k=inserto(k1,k2,u)`
  und `update(x,S,k)`.
- `Am Ende des Basisblockes:`
  1. Zu jedem Eintrag (v, S, k) der Liste, für den v eine Programmvariable ist, führe aus:
     füge einen neuen Store-Knoten $\mathbf{l} : \mathbf{k} \rightarrow \mathbf{v}$ ein
     und wenn ein Load-Knoten $\mathbf{m} : \mathbf{v} \rightarrow$ existiert, von dem aus k nicht erreichbar ist,
     dann füge eine Hilfskante von m nach k ein.
  2. Streiche alle Knoten, die keinen Store-Knoten erreichen und damit alle entsprechenden Kanten.

Beispiel für Generierung des DAG

5.2.2.2 Register-Planung mit virtuellem Registersatz

Das **Grundprinzip** besteht in

- Der Bildung einer **Verträglichkeitsrelation** $\sim$ über den Kanten des DAG, wobei
  $e \sim e' \quad :\Leftrightarrow \quad e$ und e' dürfen ein gleiches Register benutzen.
  (Eine Verträglichkeitsrelation ist eine reflexive und symmetrische Relation.)
- Der Bildung der minimalen Anzahl von **Verträglichkeitsklassen**.
  (Das entspricht der minimalen Anzahl erforderlicher virtueller Register.)

Die Relation $\sim$ erhält man mittels der partiellen Ordnung $>$ über den Kanten:

$$(k,l) > (k',l') \;\Leftrightarrow\; l = k' \vee \exists l'' : (l,l'') > (k',l').$$

Nun gilt das

Lemma 5.2.1

$$(k,l) \sim (k',l') \;\Leftrightarrow\; (k,l) < (k',l') \vee (k,l) > (k',l') \vee k = k'.$$

Beweis:
Wenn $(k,l) < (k',l') \vee (k,l) > (k',l')$, dann existieren die den Kanten zugeordneten Operandenwerte nicht gleichzeitig und
wenn $k = k'$, dann werden beide Operandenwerte von der gleichen Operation erzeugt, sind also gleich. ■

Beispiel für Verträglichkeitsklassen

Leider ist die Bildung einer minimalen Anzahl von Verträglichkeitsklasse im allgemeinen NP-vollständig! Der **Ausweg** eröffnet sich, indem man spezielle Eigenschaften der hier vorliegenden Verträglichkeitsrelation ausnutzt!

Neben der Ordnung über den Kanten existiert eine partielle Ordnung $>$ über den Knoten:

$$k > l \;\Leftrightarrow\; (k,l) \text{ ist eine Kante} \;\vee\; \exists k' : k > k' \wedge k' > l.$$

Lemma 5.2.2 *In jeder Verträglichkeitsklasse C existiert eine Kantenfolge*

$$(k_1,l_1) > (k_2,l_2) > \ldots > (k_n,l_n)$$

so, daß für jede Kante $(k,l) \in C$ ein i existiert mit

$$k = k_i \;\wedge\; (l \geq k_{i+1} \vee i = n).$$

Beweis:
Es sei $\alpha = (k_1,l_1) > \ldots > (k_n,l_n)$ eine längste Kette in C und $(k,l) \in C$.

Es muß gelten, daß $k_1 \geq k$,
denn sonst wäre $k > l > k_1$, da (k,l) kompatibel zu (k_1,l_1) ist (beide Kanten liegen in der gleichen Kompatibilitätsklasse C) und man könnte eine Kette $(k,l) > (k_1,l_1) > \ldots > (k_n,l_n)$ bilden, die länger als α ist.
Es sei nun i der größte Index mit $k_i \geq k$, also gilt $(k,l) > (k_{i+1},l_{i+1})$ oder $i = n$. Für $k_i > k$ erhält man mit $(k_1,l_1) > \ldots > (k_i,l_i) > (k,l) > (k_{i+1},l_{i+1}) > \ldots > (k_n,l_n)$ wieder eine Kette, die länger als α ist. Also verbleibt $k_i = k$.
Die Bedingung $l \geq k_{i+1} \vee i = n$ ergibt sich wieder aus der Kompatibilität von (k,l) und (k_{i+1},l_{i+1}). ■

Für die Kompatibilitätsklassen erheben wr nun folgende vereinfachende **Forderung**:

Alle Kanten, die vom gleichen Knoten ausgehen, müssen zur gleichen Klasse gehören, also dem gleichen virtuellem Register zugeordnet werden.

Diese Forderung entspricht der Tatsache, daß das Resultat einer Operation immer in ein Register abgeslegt wird, ist also **keine Einschränkung!**

Die **Konsequenz** dieser Forderung ist, daß man jetzt alle Kanten einer Klasse durch ihre Quellknoten beschreiben kann. Es sei $\widetilde{N}$ die Menge der Knoten im DAG, von denen Kanten abgehen,d.h.

$$\widetilde{N} := \{k \mid \exists l : (k,l) \in E\}$$

(die Menge der Quellknoten von Kanten, E sei die Menge aller Operandenkanten). Wir legen nun fest:

Definition 5.2.1 *Die Ordnung $>$ über den Knoten des DAG induziert eine partielle Ordnung $\succ$ über den Quellknoten, indem*

$$k \succ k' \;:\Leftrightarrow\; k > k' \wedge \forall l : (k,l) \in E \Rightarrow\; l \geq k'.$$

Folgerung 5.2.3 *Die Verträglichkeitsklassen werden durch Ketten in der Ordnung $\succ$ beschrieben.*

Beispiel für induzierte Ordnung

Die erfreuliche **Konsequenz** ist, daß man jetzt Algorithmen zur Bildung einer minimaler Kettenzerlegung einsetzen kann, die polynomialen Aufwand haben.

5.2.2.3 Zuordnung von virtuellen zu realen Registern

Jetzt wird jedem virtuellem Register ein reales Register zugeordet.

Wenn ausreichend reale Register vorhanden sind, ist das kein Problem,

sonst muß die Anzahl von Verträglichkeitsklassen reduziert werden. Das erreicht man durch die Erweiterung der Ordnung $\succ$.

Wir führen für Verträglichkeitsklassen C, C', die ja eine Struktur laut Lemma 5.2.2 haben, folgende Bezeichnungen ein:

- $max\, C = k_1$
- $Min\, C = \{\, l \mid (k_n, l) \in C\}$
- $C > C' \;:\Leftrightarrow\; \forall\, l \in Min\, C : l \geq max\, C'$
- $C \rhd C' \;:\Leftrightarrow\; \forall\, l \in Min\, C :\; max\, C' \not> l$
- $C = C_1 > C_2 > \ldots > C_n \;:\Leftrightarrow\; C = C_1 \cup C_2 \cup \ldots \cup C_n \wedge \forall\, 1 \leq i < n : C_i > C_{i+1}$

Um die Anzahl der Verträglichkeitsklassen zu reduzieren, kombinieren wir Verträglichkeitsklassen mittels Mischen und Splitten.

Mischen kombiniert zwei Klassen

$$C = C_1 > \ldots > C_n \text{ und } C' = C'_1 > \ldots > C'_n \text{ mit } C_1 \rhd C'_1 \rhd C_2 \rhd \ldots \rhd C_n \rhd C'_n$$

zu $C \cup C'$, falls durch das Einfügen folgender Hilfskanten

- $(l, max\, C'_i)$ für alle $l \in Min\, C_i$ und $l \not\geq max\, C'_i$
- $(l, max\, C_{i+1})$ für alle $l \in Min\, C'_i$ und $l \not\geq max\, C_{i+1}$

eine geordnete Folge

$$C_1 > C'_1 > C_2 > \ldots > C_n > C'_n$$

gebildet werden kann.

Beispiel 1 für die Kombination durch Mischen

Beispiel 2 für die Kombination durch Mischen

Splitten zerteilt eine Klasse

$$C = C_1 > C_2 > \ldots C_n$$

in die Klassen C_1 bis C_n, so daß diese dann einzeln mit anderen Klassen gemischt werden können.

Beispiel 1 für die Kombination durch Splitten

Beispiel 2 für die Kombination durch Splitten

Mit Mischen und Splitten kann man nicht immer bis auf wenige (zum Beispiel 2) Klassen reduzieren.

Beispiel für nicht mögliche Reduktion auf zwei Register mittels Mischen und Splitten

Dazu ist **Kantenteilung**, verbunden mit Registerrettung (Spillcode) erforderlich:
Wenn für Klassen C, C' und eine Operandenkante $(k, l) \in C$ gilt, daß

$$k \not> min\, C' \text{ und } \forall\, p \in Min\, C' : l \not> p,$$

dann

- füge für eine Hilfsvariable h den Store-Knoten $\mathbf{k'} : \mathbf{k} \rightarrow \mathbf{h}$ ein,
  dadurch entsteht eine Operandenkante (k, k')
- füge den Load-Knoten $\mathbf{l'} : \mathbf{h} \rightarrow$ ein,
- ersetze im Knoten l den Operanden k durch den Operanden l',
  dadurch entsteht eine Operandenkante (l', l),
- füge die Hilfskante $(k', min\, C')$ ein,
- füge für alle $p \in Min\, C'$ die Hilfskanten (p, l') ein,
- kombiniere C mit C'.

Beispiel für das Einfügen von Spillcode

5.3 Globale Optimierung

5.3.1 Datenfluß-Analyse

Die Datenfluß-Analyse soll die notwendigen Informationen beschaffen, damit globale Optimierungsmethoden angewendet werden können.

Grundlage ist der **Steuergraph** des Programms, bestehend aus

- den Basisblöcken als Knoten,
- den Verzweigungen zwischen den Basisblöcken als Kanten,
- genau einem Basisblock als Startknoten,
- genau einem Basisblock als Endknoten.

Benötigte Informationen an einem Knoten l sind zum Beispiel:

- **reaching definitions**: An einem Knoten k wird eine Variable v belegt und zu l existiert ein Pfad, auf dem v nicht belegt wird.
- **next-use**: an einem Knoten k wird eine Variable v benutzt und von l nach k existiert ein Pfad, auf dem v nicht belegt wird.
  (gibt Hinweis auf Lebendigkeit!)
- **always defined**: Die Variable **v** wird auf jedem Pfad vom Startknoten zu l belegt.
- **reaching copies**: Auf jedem Pfad vom Startknoten zu l liegt der Kopierbefehl `x:=y` und zwischen allen diesen Kopierbefehlen und dem Knoten l wird weder `x` noch `y` belegt.
  (`y` kann eine Variable oder Literal sein!)
- **available expressions**: Auf jedem Pfad vom Startknoten zu l wird der Ausdruck `x o y` berechnet und zwischen allen diesen Berechnungen und dem Knoten l wird weder `x` noch `y` belegt.

5.3.1.1 Grundprinzip

Das Grundprinzip der Datenfluß-Analyse besteht in der iterativen Berechnung der Informationen für jeden Basisblock, wobei von außen auf den Basisblock bei

- der Vorwärtsanalyse von allen unmittelbaren Vorgängern und
- der Rückwärtsanalyse von allen unmittelbaren Nachfolgern

Einflüsse zu berücksichtigen sind.

Es sei

- I eine Menge von *Informationen* (jede Information $i \in I$ kann man sich wieder als Menge von Einzelinformationen vorstellen),
- $in(B) \in I$ die Information am Eintritt des Basisblockes B,
- $out(B) \in I$ die Information am Austritt des Basisblockes B,

Die **Vorwärtsanalyse** berechnet für den Basisblock B:

- $in(B)$ aus $out(P_i)$ von allen unmittelbaren Vorgängern $P_1, \ldots, P_k$ von B mittels einer Funktion $\bigtriangledown : \wp(I) \to I$, indem

$$in(B) = \bigtriangledown\{out(P_1), \ldots, out(P_k)\}.$$

- $out(B)$ aus $in(B)$ und B mittels einer Funktion $\varphi : \mathcal{B} \to (I \to I)$, indem

$$out(B) = \varphi(B)(in(B)).$$

- Zusammengefaßt erhält man für jeden Basisblock B und dessen unmittelbaren Vorgänger $P_1, \ldots, P_k$ die Gleichung:

$$out(B) = \varphi(B)(\bigtriangledown\{out(P_1), \ldots, out(P_k)\}).$$

Die **Rückwärtsanalyse** berechnet für den Basisblock B:

- $out(B)$ aus $in(S_i)$ von allen unmittelbaren Nachfolgern $S_1, \ldots, S_k$ von B mittels einer Funktion $\bigtriangleup : \wp(I) \to I$, indem

$$out(B) = \bigtriangleup\{in(S_1), \ldots, in(S_k)\}.$$

- $in(B)$ aus $out(B)$ und B mittels einer Funktion $\varrho : \mathcal{B} \to (I \to I)$, indem

$$in(B) = \varrho(B)(out(B)).$$

- Zusammengefaßt erhält man für jeden Basisblock B und dessen unmittelbaren Vorgänger $S_1, \ldots, S_k$ die Gleichung:

$$in(B) = \varrho(B)(\bigtriangleup\{in(S_1), \ldots, in(S_k)\}).$$

In beiden Fällen erhält man ein rekursives (im Falle von Schleifen im Steuergraph) Gleichungssystem, das iterativ zu lösen ist.

Falls der Startknoten B keinen unmittelbaren Vorgänger hat (bzw. der Endeknoten keinen Nachfolger) dann gilt dafür bei der Vorwärtsanalyse $out(B) = \varphi(B)(\bigtriangledown(\emptyset))$. Man als einzigen unmittelbaren Vorgänger aber auch einen leeren Basisblock L ansetzen, der alle Variablenbelegungen unverändert läßt. Wenn die zu L gehörende Information ι ist, dann muß gelten:

$$\bigtriangledown(\emptyset) = \iota$$

und dazu analog

$$\bigtriangleup(\emptyset) = \iota.$$

Man kann $\bigtriangledown$ (analog $\bigtriangleup$) mittel eines binären Operators (Funktion) $\triangledown : I \times I \to I$ ($\triangle$: $I \times I \to I$) beschreiben, indem

$$\bigtriangledown\{i\} := i, \quad \bigtriangledown\{i_1, \ldots, i_k\} := i_1 \triangledown (\bigtriangledown\{i_2, \ldots, i_k\}).$$

Für den Operator $\triangledown$ (analog für $\triangle$) sind folgende Eigenschaften zu fordern:

1. $a \triangledown (b \triangledown c) = (a \triangledown b) \triangledown c$ - Assoziativität
2. $a \triangledown b = b \triangledown a$ - Kommutativität
3. $a \triangledown a = a$ - Idempotenz
4. es gibt ein $\top \in I$ mit $a \triangledown \top = a$ für alle $a \in I$

Das Element $\top$ (Top-Element) repräsentiert die *worst-case Information*, mit der die Iteration für alle $out(B)$ bei der Vorwärtsanalyse zu starten ist.

Über der Menge der Informationen I führen wir eine (partielle) Ordnung $\leq \subseteq I \times I$ ein, indem

$$a \leq b \ :\Leftrightarrow \ a \triangledown b = a.$$

In dieser Ordnung bringt die Tatsache, daß $a \leq b$ ist, zum Ausdruck, daß a *schärfere* Aussagen über ein Programm macht, als b. Die Optimierung auf der Grundlage von b kann eventuell umfangreicher erfolgen, als auf der Grundlage von a, aber die Optimierung auf der Grundlage von a ist sicherer. Da für eine beliebige Information $a \in I : a \triangledown \top = a$ gilt, erhält man $a \leq \top$, also repräsentiert $\top$ die schwächste Information.

Schließlich soll die Funktion φ (bzw. ϱ) aus schärferen Informationen auch schärfere Informationen gewinnen. Wir fordern also die Monotonie:

$$a \leq b \ \Rightarrow \ \varphi(a) \leq \varphi(b) \wedge \varrho(a) \leq \varrho(b).$$

5.3.1.2 Schärfe der Analyseergebnisse

Es sei $\beta = B_1 B_2 \ldots B_n$ ein Pfad, der bei der Abarbeitung des Programms durchlaufen wird. Es ergibt sich

$$\begin{aligned} out(B_n) &= \varphi(B_n)(out(B_{n-1})) \\ &= \varphi(B_n)(\varphi(B_{n-1})(out(B_{n-2}))) \\ &\ldots \end{aligned}$$

Wir erweitern $\varphi : \mathcal{B} \to (I \to I)$ auf $\varphi : \mathcal{B}^* \to (I \to I)$, indem

$$\varphi^*(B)(a) := \varphi(B)(a), \ \ \varphi^*(B_1 \ldots B_n)(a) := \varphi^*(B_2 \ldots B_n)(\varphi(B_1)(a)).$$

Wenn ι die zum leeren Basisblock gehörende Information ist, dann beschreibt $\varphi^*(\alpha)(\iota)$ die zum Pfad α gehörende Information!

Es sei nun A die Menge aller Pfade, die bei irgendeiner Abarbeitung des Programms vom Startknoten zum Endknoten durchlaufen werden. Dann ist

$$e = \triangledown\{\varphi(\alpha)(\iota) \mid \alpha \in A\}$$

die exakte Information über das Programm.

Nun ist bekannt (Siehe das GTI-Skript, Abschnitt 3.5), daß die Menge A nicht entscheidbar ist. Deshalb gehen wir der entscheidbaren Obermenge $G \supseteq A$, der Menge aller Pfade im Steuergraph des Programms über.

Wegen $(a \triangledown b) \triangledown a = a \triangledown (a \triangledown b) = (a \triangledown a) \triangledown b = a \triangledown b$ gilt $(a \triangledown b) \leq a$, also

$$\triangledown\{\varphi(\alpha)(\iota) \mid \alpha \in G\} \leq \triangledown\{\varphi(\alpha)(\iota) \mid \alpha \in A\}.$$

Falls B der zum Endeknoten gehörende Basisblock ist, dann ist (bei der Vorwärtsanalyse) $out(B)$ die zum Programm gehörende Information. Diese Information wird durch Iteration des aufgestellten Gleichungssystems gewonnen. Es sei $out^*(B)$ der Fixpunkt.

Satz 5.3.1

$$out^*(B) \leq \bigtriangledown\{\varphi(\alpha)(\iota) \mid \alpha \in G\}.$$

Beweis:
Aus $a \bigtriangledown b \leq a$ und der Monotonie von φ folgt $\varphi(a \bigtriangledown b) \leq \varphi(a)$ und $\varphi(a \bigtriangledown b) \leq \varphi(b)$.

Wenn	$a \leq b$	und	$a \leq c$	, so
	$a \bigtriangledown b = a$		$a \bigtriangledown c = a$	, zusammen
			$a \bigtriangledown b \bigtriangledown c = a$	, also
			$a \bigtriangledown (b \bigtriangledown c) = a$	, also
			$a \leq b \bigtriangledown c$	

und damit

$$\varphi(a \bigtriangledown b) \leq \varphi(a) \bigtriangledown \varphi(b).$$

■

Beispiele für Datenfluß-Analyse: reaching definitions
Beispiele für Datenfluß-Analyse: always defined
Beispiel für Datenfluß-Analyse: next use

5.3.2 Transformationen

Wichtig: Transformationsschritte können die Informationsbasis verändern.

Konsequenz: Datenfluß-Analyse und Transformation ist iterativ zu wiederholen.

5.3.2.1 Berechnung konstanter Ausdrücke zur Compile-Zeit

- **Information:** Liste der Konstanten.
- **Methode:** Berechne alle Ausdrücke, die nur aus Konstanten bestehen und ersetze diese durch das Resultat.
- **Konsequenz:** Falls Belegungen der Form `x:=c` (`c` ist Konstante) entstehen, können auch Ausdrücke mit Variablen konstant werden.

5.3.2.2 Dead code elimination

- **Information:** next-use.
- **Methode:** Falls eine Variable `x` kein next-use hat, kann die Belegung `v:=e` gestrichen werden.
- **Konsequenz:** Für die Variablen in `e` reduziert sich next-use.

5.3.2.3 Copy propagation

- **Information:** reaching copies.
- **Methode:** Falls am Knoten `k` die Variable `x` benutzt wird und `x:=y` ist in reaching copies, dann kann `x` durch `y` ersetzt werden (`y` kann Variable oder Konstante sein).
- **Konsequenz:** Für `x` reduziert sich next-use.

5.3.2.4 Common subexpression elimination

- **Information:** available expressions.
- **Methode:** Falls eine Berechnung `y o z` an `k` in available expressions ist, so gehe zu allen diesen vorangehenden Berechnungen `x:=y o z` und ersetze diese durch `u:=y o z; x:=u` und ersetze an `k` die Berechnung `y o z` durch `u`.
- **Konsequenz:** Neue Kopierbefehle.

5.3.2.5 Code motion

- **Information:** reaching definitions.
- **Methode:** Es sei
  - L eine Menge von Knoten, die auf einem Zyklus liegen,
  - h der Knoten, über den als einziger in L eingetreten wird,
  - `y o z` eine Berechnung in L für die sowohl `y` als auch `z` keine reaching definitions in L haben,

  dann
  - füge einen Knoten h' mit `x:=y o z` hinter h ein und
  - ersetze `x:=y o z` in L durch `x`.
- **Konsequenz:** eventuell entstehen neue Kopierbefehle.

5.3.2.6 Strength reduction

- **Information:** Zyklen der Form:

```
do
{
    a=g(f(x));
    x+=c;
}
while(test(x));
```

- **Methode:**
  Für $f_\Delta(x) = f(x+c) - f(x)$ gilt: $f(x+c) = f(x) + f_\Delta(x)$.
  Wenn die Berechnung von $f_\Delta(x)$ billiger als von $f(x)$ ist, dann transformiert man nach

```
d=f(x);
do
{
    a=g(d);
    d+=f_D(x);
    x+=c;
}
while(test(x));
```

- **Konsequenz:**
  Wir setzen $f_{\Delta^{i+1}}(x) = f_{\Delta^i}(x+c) - f_{\Delta^i}(x)$ mit $f_{\Delta^0}(x) = f(x)$, also gilt:
  $f_{\Delta^i}(x+c) = f_{\Delta^i}(x) + f_{\Delta^{i+1}}(x)$.
  Wenn $f_{\Delta^{n+1}}(x,c)$ eine Konstante `cn`, dann lohnt sich die Transformation in

```
d0=f(x);
d1=f_D1(x);
d2=f_D2(x);
.
.
.
dn=f_Dn(x);
cn=f_Dn(x+c)-dn;
do
{
    a=g(d0);
    d0+=d1;
    d1+=d2;
    .
    .
    .
    dn+=cn;
    x+=c;
}
while(test(x));
```

Beispiel für strength reduction