1. Einleitung

In der Informatik werden zur Modellierung komplexer Sachverhalte häufig **Graphen** eingesetzt. Sie begegnen uns in nahezu allen Disziplinen der Informatik - und auch in anderen Anwendungsgebieten, z.B. der Biologie, der Chemie, ... Als Beispiele lassen sich Petrinetze, Syntaxdiagramme, Netzpläne, SADT-Diagramme, abstrakte Syntaxgraphen, Graphen zur Beschreibung von Bildstrukturen, Entity-Relationship-Diagramme, ... anführen.

So wie String- oder Baumgrammatiken dazu verwendet werden, den Aufbau von Zeichenketten oder Bäumen zu beschreiben, läßt sich die Struktur komplexer Graphen formal mit Hilfe von **Graphgrammatiken** definieren. Eine Graphgrammatik ist - ähnlich wie eine "normale" Grammatik - ein System von Regeln - die auch als Produktionen bezeichnet werden - , die aus einem Axiom - auch als Startgraph bezeichnet - alle Graphen einer bestimmten Klasse erzeugen. Dabei beschreibt eine Produktion die Ersetzung einer linken Seite durch eine rechte Seite. Sowohl die linke als auch die rechte Seite sind Graphen; eine Produktion ist auf einen Graphen anwendbar, wenn die linke Seite in ihm vorkommt.

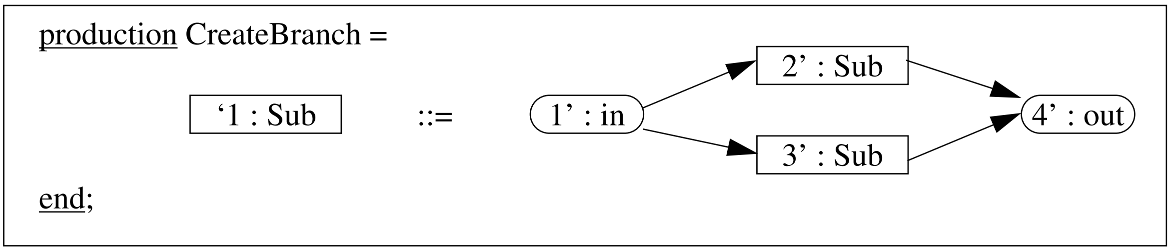


Abb. 1.1: Beispiel für eine Graphproduktion

Ein Beispiel für eine Graphproduktion zeigt Abb. 1.1. Die linke Seite besteht aus einem Knoten, der einen Teil eines Syntaxdiagramms repräsentiert. Dieser Knoten wird durch einen Graphen ersetzt, der eine Verzweigung im Syntaxdiagramm darstellt. Wie wir später noch sehen werden, lassen sich mit Hilfe dieser und anderer Produktionen Syntaxdiagramm-Graphen erzeugen.

Graphgrammatiken sind u.a. in der **Biologie** eingesetzt worden, um das Wachstum von Moosen zu beschreiben. Die Hauptanwendungsgebiete liegen aber bisher auf dem Gebiet der Informatik. So wurden Graphgrammatiken etwa zur Beschreibung der Semantik von Programmiersprachen benutzt. Weitere Anwendungsgebiete liegen beispielsweise im Bereich der **Mustererkennung**, wo es darum geht, die Struktur von Bildern (Röntgenaufnahmen, Satellitenfotos, etc.) zu analysieren. Darüber hinaus stellt auch die Bildsynthese ein potentielles Anwendungsgebiet dar. Die Liste der Anwendungsgebiete läßt sich nahezu beliebig fortsetzen.

Eng verwandt mit den Graphgrammatiken sind die **Graphersetzungssysteme**. Während diese Begriffe in der Literatur oft synonym gebraucht werden, werden wir zwischen ihnen unterscheiden:

- Wann immer der generative Aspekt - d.h. die Angabe eines Systems zur Erzeugung oder Erkennung von Graphen einer bestimmten Klasse - im Vordergrund steht, wird von **Graphgrammatiken** die Rede sein.
- Dagegen verwenden wir den Begriff **Graphersetzungssystem**, wenn wir Transformationsprozesse beschreiben und uns dabei i.w. nicht dafür interessieren, ob die betrachteten Graphen "terminalen" oder "nichtterminalen" Charakter haben.

Wo immer dieser Unterschied irrelevant ist, wird im folgenden von **Graphregelsystemen** die Rede sein.

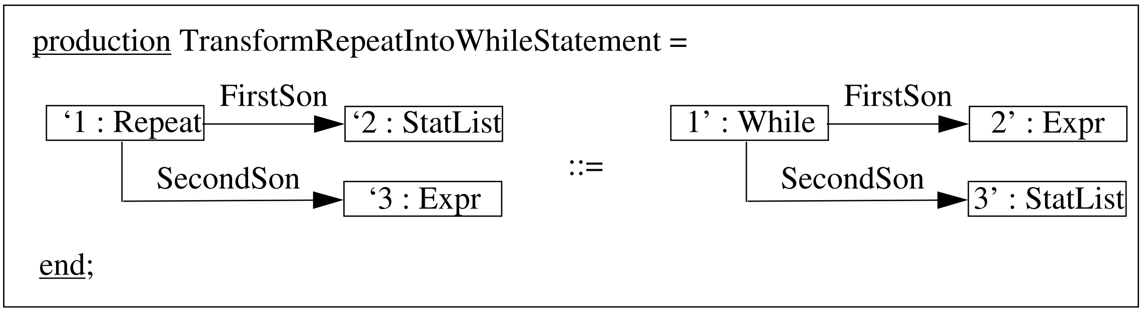


Abb. 1.2: Beispiel für eine Graphersetzung

Graphersetzungssysteme lassen sich als Verallgemeinerungen von Termersetzungssystemen auffassen, die vor allem aus dem Compilerbau bekannt sind. Mit Hilfe von Graphersetzungssystemen lassen sich beispielsweise **strukturbezogene Editoren** spezifizieren; die Wirkungsweise von Kommandos wie Erzeugen, Löschen und Modifizieren von Teilstrukturen wird dabei auf Graphersetzungen zurückgeführt. Als weiteres Anwendungsgebiet läßt sich die **Zwischencode-Optimierung** im Compilerbau anführen (z.B. Herausziehen gemeinsamer Teilausdrücke).

Ein Beispiel für eine Graphersetzung ist in Abb. 1.2 zu sehen. Sie beschreibt die Transformation einer "REPEAT"-Anweisung, die aus einer Anweisungsliste und einer nachfolgenden Abbruchbedingung besteht, in eine "WHILE"-Anweisung, bei der die Reihenfolge von Abbruchbedingung und Anweisungsliste vertauscht ist. Eine solche Operation ist typisch für einen syntaxgesteuerten Editor, der intern Programme mittels abstrakter Syntaxgraphen präsentiert und diese in textueller Form dem Benutzer anzeigt.

Diese Ausarbeitung soll sowohl in die **Theorie** als auch in die **Anwendungen** von Graphregelsystemen einführen. Aus dieser Zielsetzung ergibt sich, daß zum einen in ausreichendem Maße theoretische Grundlagen zu behandeln sind, zum anderen aber auch prakti-

sche Anwendungen nicht vernachlässigt werden dürfen. Um den Umfang der Darstellung in erträglichem Rahmen zu halten, wird ein Überblick über das Gebiet vermittelt, der es dem interessierten Leser ermöglichen soll, sich bei Bedarf in Teilgebiete vertiefend einzuarbeiten.

Betrachtet man die Literatur zu Graphregelsystemen, die sich seit der Entstehung dieses Forschungsgebiets Ende der 60er Jahre angesammelt hat, so stößt man auf eine Fülle verschiedener Ansätze zur **Formalisierung**. Ein wesentliches Ziel dieser Ausarbeitung besteht darin, einen Überblick über die Hauptrichtungen zu vermitteln, in die sich diese Ansätze einordnen lassen. Dabei werden Beziehungen zwischen diesen Hauptrichtungen aufgezeigt; es wird aber aber nicht versucht, eine "Metatheorie" zu entwickeln, die einen einheitlichen Rahmen bietet, innerhalb dessen sich die bisher entwickelten Ansätze beschreiben lassen.

Das Studium der Literatur zeigt weiterhin, daß die Behandlung der Theorie die Beschäftigung mit ihren **Anwendungen** bisher klar dominiert. Dies ist sicherlich ein wichtiger Grund dafür, daß Graphregelsysteme bisher relativ wenig verbreitet sind und gegenüber anderen Ansätzen wie z.B. attributierte Grammatiken und Termersetzungssystemen noch immer als Exoten erscheinen. Daher werden in dieser Ausarbeitung die verschiedenen Ansätze insbesondere unter dem Blickwinkel ihrer praktischen Anwendbarkeit studiert.

Ein interessantes Anwendungsgebiet für Graphregel-, insbesondere Graphersetzungssysteme stellen **Softwareentwicklungsumgebungen** dar. Die Werkzeuge einer Softwareentwicklungsumgebung operieren intern auf komplexen Datenstrukturen, die sich mit Hilfe von Graphen mehr oder weniger hervorragend modellieren lassen. Mit Hilfe von Graphersetzungssystemen lassen sich die Operationen, die von den Werkzeugen einer Umgebung ausgeführt werden, in natürlicher Weise spezifizieren. Dies gilt insbesondere für die Operationen, die ein syntaxgesteuerter Editor ausführt: Softwaredokumente wie Anforderungsdefinitionen, Architekturen, Modulimplementationen etc. lassen sich intern durch Graphen darstellen, Veränderungen von Graphen (z.B. Erzeugen einer Wertzuweisung, Ändern eines Bezeichners etc.) werden mit Hilfe von Graphersetzungen beschrieben.

Im Rahmen des **IPSEN-Projekts** werden Graphersetzungssysteme seit Jahren dazu benutzt, um die einer Softwareentwicklungsumgebung zugrundeliegenden Datenstrukturen und die auf ihnen ausführbaren Operationen auf einem hohen Abstraktionsniveau zu spezifizieren. Der zugrundeliegende Kalkül - und auch seine sprachliche Einkleidung, s.u. - ist im Laufe der Jahre immer weiter entwickelt worden. Den vorläufigen Endpunkt dieser Entwicklung stellt die Sprache **PROGRESS** dar (**PRO**grammierte **GR**aph-**E**rsetzungs-**S**ystem-**S**pezifikation). PROGRESS ist eine Spezifikationssprache, die als Hilfsmittel für den Entwerfer einer Softwareentwicklungsumgebung gedacht ist (sich aber auch in anderen Bereichen ein-

setzen läßt). Als solche steht sie in Konkurrenz zu anderen Ansätzen wie z.B. attributierten Grammatiken oder Entity-Relationship-Ansätzen.

Obwohl die vorliegende Ausarbeitung keine Einführung in PROGRESS darstellt, sondern einen Überblick über das gesamte Gebiet der Graphregelsysteme zu geben versucht, werden wir PROGRESS immer wieder begegnen. Zum einen werden sukzessive Konzepte eingeführt, die sich großenteils auch in PROGRESS wiederfinden; zum anderen wird in der gesamten Ausarbeitung eine PROGRESS-ähnliche Syntax benutzt, um die Konzepte in einer einheitlichen Notation zu präsentieren. Deshalb ist es wichtig, im folgenden zwei Ebenen deutlich auseinanderzuhalten:

- Auf der Ebene der **sprachlichen Einkleidung** geht es um syntaktische Fragen, d.h. um den Aufbau einer formalen Sprache, mit deren Hilfe sich Graphregelsysteme in adäquater Weise notieren lassen.
- Auf der Ebene des zugrundeliegenden **Kalküls** geht es um die Semantik von Graphersetzungen, die - je nach Ansatz - mit unterschiedlichen formalen Hilfsmitteln beschrieben wird.

Die Gliederung der Ausarbeitung orientiert sich an den Kalkülen, auf denen Graphregelsysteme beruhen. Jedem Kalkül - oder besser gesagt: jeder Gruppe von Kalkülen - ist ein eigenes Kapitel gewidmet. In allen Kapiteln wird durchgängig eine an PROGRESS angelehnte sprachliche Einkleidung gewählt. Was reales und was Pseudo-PROGRESS ist, ist dabei nicht von entscheidender Bedeutung. Desgleichen stellen wir die Frage, auf welchem Kalkül PROGRESS selbst basiert, zunächst einmal zurück und konzentrieren uns stattdessen auf die wesentlichen praktischen und theoretischen Eigenschaften der vorgestellten Kalküle.

Hinsichtlich der Kalküle lassen sich drei Gruppen unterscheiden (wobei die Grenzen wie immer fließend sind), die hier nur in Kürze charakterisiert werden sollen:

- **Mengentheoretische Ansätze**: In ihnen werden Graphen durch Mengen von Knoten und Kanten beschrieben, und die Wirkung von Graphregeln wird mit Hilfe von mengentheoretischen Operationen wie Differenz, Vereinigung etc. beschrieben.
- **Kategorientheoretische Ansätze**: Im Mittelpunkt der theoretischen Fundierung steht der Begriff der Kategorie, d.h. einer totalen und assoziativen zweistelligen Verknüpfung. Die Wirkung einer Graphregel wird mit Hilfe von kategorientheoretischen Begriffen definiert.
- **Logikorientierte Ansätze**: Hier werden sowohl Graphen als auch Graphregeln mit Hilfe von Formelmengen dargestellt, und die Ausführung einer Graphregel wird als die Manipulation einer Formelmenge aufgefaßt.

Diese Ansätze unterscheiden sich nicht nur hinsichtlich ihrer theoretischen Fundierung, sondern auch bezüglich ihrer praktischen Anwendbarkeit. Einige wesentliche Eigenschaften seien hier angeführt:

- Mengentheoretische Ansätze sind zum einen intuitiv am leichtesten verständlich und gestatten es zum anderen, recht komplexe Graphregeln formal zu beschreiben.
- Kategorientheoretische Ansätze erschließen sich schwerer dem intuitiven Verständnis und sind auch hinsichtlich der mit ihnen beschreibbaren Graphregeln stärker eingeschränkt. Die dort gemachten Einschränkungen haben aber andererseits den Vorteil, daß Operationen auf Graphregeln wie z.B. Invertieren und Komponieren wohldefiniert sind. Letzteres erleichtert beispielsweise die formale Beschreibung nebenläufiger Prozesse.
- Logikorientierte Ansätze gestatten es, komplexe Regeln über den Aufbau von Graphen zu formulieren, die beispielsweise festlegen, welche Arten von Knoten durch welche Arten von Kanten verbunden werden dürfen und wie Attribute verschiedener Knoten zueinander in Beziehung stehen. Solche Regeln bilden das Schema eines Graphregelsystems (in Analogie zum Schema einer Datenbank) und dienen dazu, die Korrektheit der Graphregeln zu überprüfen sowie in einem konkreten Graphen die Werte gewisser Knotenattribute nicht-prozedural durch gerichtete Gleichungen festzulegen.

2. Mengentheoretischer Ansatz

In diesem Kapitel wird ein Ansatz zur Definition von Graphregelsystemen dargestellt, der auf der Mengentheorie basiert. Dieser Ansatz ist intuitiv leichter verständlich als der kategorientheoretische und der logikbasierte Ansatz, die in den folgenden Kapiteln behandelt werden.

Das vorliegende Kapitel ist folgendermaßen gegliedert: In Abschnitt 2.1 werden zunächst einige Grundbegriffe eingeführt, die im folgenden häufig benötigt werden. In Abschnitt 2.2 werden basierend auf diesen Begriffen kontextfreie Graphgrammatiken definiert. Abschnitt 2.3 befaßt sich mit Verfahren, mit deren Hilfe sich die syntaktische Struktur von Graphen analysieren läßt. In Abschnitt 2.4 werden Attribute eingeführt, die dazu dienen, zusätzliche Informationen an Knoten und Kanten darzustellen. In Abschnitt 2.5 werden Graphgrammatiken betrachtet, die nicht kontextfrei sind; ferner werden Graphersetzungssysteme eingeführt. Abschnitt 2.6 geht darauf ein, wie sich die Anwendung von Graphregeln durch Kontrollstrukturen steuern läßt. Abschnitt 2.7 gibt schließlich eine Übersicht über einige wesentliche Literaturquellen zum vorliegenden Kapitel.

2.1 Grundbegriffe

In diesem Abschnitt werden einige Grundbegriffe definiert, die im folgenden häufig benötigt werden.

Zunächst ist zu klären, was wir unter einem Graphen verstehen wollen. Wir werden in diesem Kapitel - und auch im weiteren Verlauf - **gerichtete Graphen** betrachten, deren Knoten und Kanten markiert sind. Markierungen geben dabei "Typen" von Objekten und von Beziehungen zwischen Objekten wieder. Unmarkierte bzw. ungerichtete Graphen lassen sich als Spezialfälle markierter bzw. gerichteter Graphen auffassen (nur eine Knoten- bzw. Kantenmarkierung; zu jeder Kante existiert auch deren "Inverse"); diese Spezialfälle werden im folgenden nicht explizit behandelt.

Def. 2.1.1 Gerichtete, markierte Graphen

Ein gerichteter, markierter Graph (im folgenden auch kurz als Graph bezeichnet) über zwei Mengen $\mathcal{L}_V$, $\mathcal{L}_E$ ist ein Tripel

$$G = (\mathcal{V}, \mathcal{E}, l)$$

mit folgenden Komponenten:

(1) $\mathcal{V}$ ("vertices") ist eine (endliche) Menge von Knotenbezeichnern.

(2) $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{L}_E \times \mathcal{V}$ ist eine Menge von Kanten ("edges"), wobei $\mathcal{L}_E$ eine (endliche) Menge von Kantenmarkierungen ("edge labels") ist.

(3) $l : \mathcal{V} \to \mathcal{L}_V$ ist die Knotenmarkierungsfunktion; dabei ist $\mathcal{L}_V$ eine (endliche) Menge von Knotenmarkierungen ("vertex labels"). ■

Man beachte, daß Kanten als Tripel definiert sind. Zwischen je zwei Knoten können also nicht mehrere gleich markierte und gerichtete Kanten verlaufen.

Im folgenden werden mit G, G', G", ... Graphen bezeichnet. Für die Benennung ihrer Komponenten gilt zudem: $G = (\mathcal{V}, \mathcal{E}, l)$, $G' = (\mathcal{V}', \mathcal{E}', l')$ etc.

Def. 2.1.2 Menge gerichteter, markierter Graphen

Seien $\mathcal{L}_V$, $\mathcal{L}_E$ zwei endliche Mengen (Knoten- bzw. Kantenmarkierungen). Dann wird die Menge aller markierten Graphen über $\mathcal{L}_V$ und $\mathcal{L}_E$ mit $\mathcal{G}(\mathcal{L}_V,\mathcal{L}_E)$ bezeichnet. ■

Bsp. 2.1.3 Wohlstrukturierte Syntaxdiagramme

Mit Hilfe von Syntaxdiagrammen lassen sich kontextfreie Grammatiken definieren. Wir betrachten hier nur wohlstrukturierte Syntaxdiagramme, die sich aus Teildiagrammen zusammensetzen, die genau einen "in"- und genau einen "out"-Knoten besitzen. Neben den "in"- und "out"-Knoten enthalten Syntaxdiagramme noch weitere Knoten, die Sprachkonstrukte repräsentieren. In Syntaxdiagrammen gibt es nur eine Art von Kanten, die die Reihenfolge von Sprachkonstrukten festlegen und daher als "next"-Kanten bezeichnet werden. Die Kantenmarkierung wird in der graphischen Darstellung der Einfachheit halber weggelassen.

Das folgende Beispieldiagramm (Abb. 2.1) zeigt die Grundelemente Sequenz, Verzweigung und Schleife, aus denen sich wohlstrukturierte Syntaxdiagramme i.a. zusammensetzen. Es ist analog zu dem regulären Ausdruck

(A|B)(CD(A|B))*

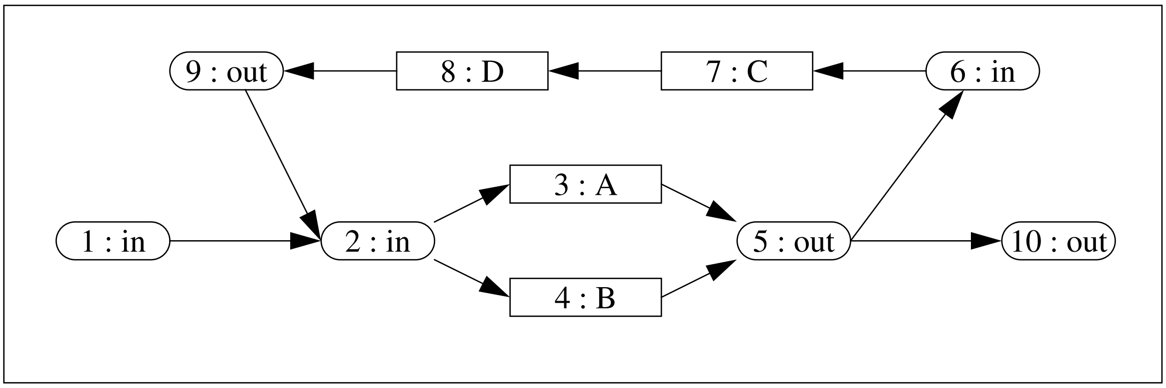


Abb. 2.1: Beispiel für ein wohlstrukturiertes Syntaxdiagramm

Formal läßt sich der obige Graph folgendermaßen beschreiben:

(1) $\mathcal{V} = \{1, ..., 10\}$
(2) $\mathcal{L}_E = \{next\}$
(3) $\mathcal{E} = \{(1,next,2), (2,next,3), (2,next,4), ... \}$
(4) $\mathcal{L}_V = \{in, out, A, B, C, D\}$
(5) $l = \{(1,in), (2,in), (3,A), ... \}$ ■

Die im folgenden definierten Begriffe werden benötigt, um den Effekt von Graphregeln formal beschreiben zu können.

Def. 2.1.4 Teilgraph

Seien G, G' $\in \mathcal{G}(\mathcal{L}_V,\mathcal{L}_E)$. Dann heißt G Teilgraph von G' (in Zeichen : $G \subset G'$) $\Leftrightarrow$

(1) $\mathcal{V} \subseteq \mathcal{V}'$
(2) $\mathcal{E} \subseteq \mathcal{E}'$
(3) $l = l'\,|_{\mathcal{V}}$ (d.h. $\forall\, v \in \mathcal{V}: l(v) = l'(v)$) ■

Def. 2.1.5 Untergraph

Es sei $G \subset G'$. Dann heißt G Untergraph von G' (in Zeichen : $G \subseteq G'$) $\Leftrightarrow$
G enthält alle Kanten aus $\mathcal{E}'$, die Knoten aus $\mathcal{V}$ verbinden:
$\mathcal{E} = \mathcal{E}'\,|_{\mathcal{V}}$ (d.h. $\forall\, e' \in \mathcal{E}' : s(e') \in \mathcal{V} \wedge t(e') \in \mathcal{V} \Rightarrow e' \in \mathcal{E}$)
Dabei seien $s, t : \mathcal{E} \rightarrow \mathcal{V}$ Funktionen, die Quell- bzw. Zielknoten einer Kante liefern ("source" bzw. "target"). ■

Def. 2.1.6 Homomorphismus

Seien G, G' $\in \mathcal{G}(\mathcal{L}_V,\mathcal{L}_E)$. Ferner sei $h : \mathcal{V} \rightarrow \mathcal{V}'$. Die Funktion h heißt (Graph-)Homomorphismus von G nach G' (in Zeichen: $G \tilde{\rightarrow} G'$) $\Leftrightarrow$
h erhält die Markierungen von Knoten sowie Endknoten, Orientierung und Markierung von Kanten) :

(1) $\forall\, v \in \mathcal{V}: l'(h(v)) = l(v)$
(2) $\forall\, v_1, v_2 \in \mathcal{V}\ \forall\, el \in \mathcal{L}_E : (v_1,el,v_2) \in \mathcal{E} \Rightarrow (h(v_1),el,h(v_2)) \in \mathcal{E}'$ ■

Def. 2.1.7 Isomorphismus

Es sei $h : G \tilde{\rightarrow} G'$. h heißt Isomorphismus von G nach G' (in Zeichen: $G \tilde{\tilde{\rightarrow}} G'$) $\Leftrightarrow$

(1) $h : \mathcal{V} \rightarrow \mathcal{V}'$ ist injektiv und surjektiv.
(2) $h^{-1} : \mathcal{V}' \rightarrow \mathcal{V}$ ist ein Homomorphismus von G' nach G.

Ferner heißen G und G' isomorph (in Zeichen: $G \cong G'$) $\Leftrightarrow$
es existiert ein Isomorphismus von G nach G'. ■

Isomorphe Graphen sind also strukturgleich, d.h. sie stimmen bis auf die Knotenbezeichner überein. Man beachte, daß die Bedingung (2) sicherstellt, daß G' keine Kanten enthält, die nicht als Bilder von Kanten aus G auftreten. Die erste Forderung genügt nicht alleine, da h^{-1} i.a. die Forderung (2) aus Def. 2.1.6 verletzt.

Def. 2.1.8 Differenz, Vereinigung und disjunkte Vereinigung von Graphen

Seien G, G' $\in \mathcal{G}(\mathcal{L}_V,\mathcal{L}_E)$. Ferner seien die Knotenmarkierungsfunktionen l und l' miteinander konsistent, d.h.

$l|_{\mathcal{V}\cap\mathcal{V}'} = l'|_{\mathcal{V}\cap\mathcal{V}'}$

Dann ist die Differenz von G und G' folgendermaßen definiert
(in Zeichen : G'' = G \ G'):

(1) $\mathcal{V}'' = \mathcal{V} \setminus \mathcal{V}'$

(2) $\mathcal{E}' = \mathcal{E}|_{\mathcal{V}''}$

(3) $l'' = l|_{\mathcal{V}''}$

Analog gilt für die Vereinigung (G'' = G $\cup$ G'):

(4) $\mathcal{V}'' = \mathcal{V} \cup \mathcal{V}'$

(5) $\mathcal{E}'' = \mathcal{E} \cup \mathcal{E}'$

(6) $l'' = l \cup l'$ (d.h. $l''(v'') = l(v'')$, falls $v'' \in \mathcal{V}$, $l''(v'') = l'(v'')$ sonst)

Falls $\mathcal{V} \cap \mathcal{V}' = \emptyset$, spricht man von disjunkter Vereinigung (G'' = G $\oplus$ G'). ■

2.2 Kontextfreie Graphgrammatiken

In diesem Abschnitt wenden wir uns einer einfachen Klasse von Graphgrammatiken zu, die als kontextfrei bezeichnet wird. Betrachten wir zunächst **kontextfreie Produktionen** für **Zeichenketten**: Ein Beispiel für eine solche Produktion ist

A ::= BC,

wobei A ein Nichtterminalsymbol und B, C beliebige Symbole seien. Wird diese Produktion auf die Zeichenkette

DAE

angewendet, so entsteht daraus

DBCE.

Eine **kontextfreie Graphproduktion** beschreibt nun analog dazu die Ersetzung eines Knotens, der mit einem Nichtterminalsymbol markiert ist, durch einen Teilgraphen. Eine zu obiger Produktion analoge Graphproduktion ist in Abb. 2.2 dargestellt. Dabei wird eine Zeichenkette durch einen Graphen repräsentiert, der für jedes Symbol einen Knoten enthält und dessen Kanten die lineare Anordnung der Symbole wiedergeben.

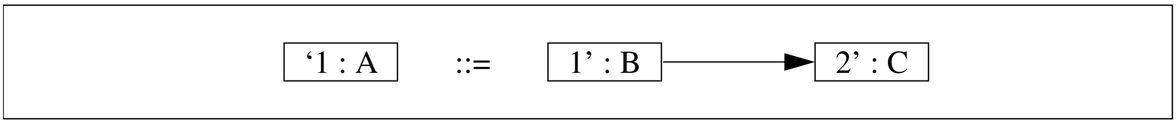


Abb. 2.2: Beispiel für eine kontextfreie Graphproduktion

Die **Anwendung einer Graphproduktion** auf einen sogenannten **Wirtsgraphen** wird in Abb. 2.3 illustriert. Folgende Schritte sind zu unterscheiden:

(1) Der zu ersetzende Knoten mit Markierung A wird **identifiziert** (Knoten 2).

(2) Der zu ersetzende Knoten wird **gelöscht**.

(3) Ein zur rechten Seite isomorpher Untergraph wird **eingefügt** (Knoten 4 und 5 sowie die zwischen ihnen verlaufende Kante).

(4) Der eingefügte Untergraph wird in den Wirtsgraphen **eingebettet**, d.h. es werden Kanten von den "neuen" Knoten des Untergraphen zu "alten" Knoten des Wirtsgraphen gezogen (Kanten von 1 nach 4 bzw. von 5 nach 3).

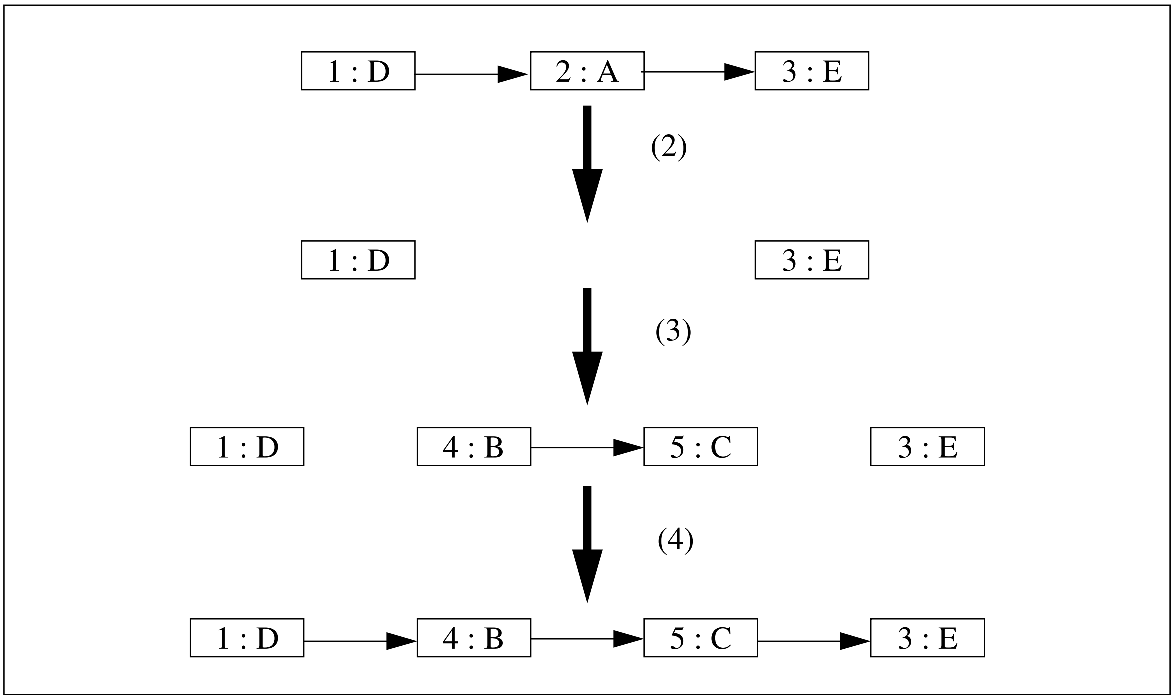


Abb. 2.3: Anwendung einer Graphproduktion

Ein wesentlicher Unterschied gegenüber der Anwendung von Zeichenketten-Produktionen besteht darin, daß die **Einbettung** der rechten Seite in den Wirtsgraphen i.a. nicht a

priori klar ist, sondern explizit geregelt werden muß. I.a. hängt die Einbettung von dem Kontext ab, in dem der zu ersetzende Knoten steht (Kanten von 1 nach 2 bzw. von 2 nach 3 im obigen Beispiel). Dieser Kontext muß daher ermittelt werden, bevor die linke Seite aus dem Wirtsgraphen entfernt wird.

Mit Hilfe einer **Einbettungsüberführungregel** wird beschrieben, wie die rechte Seite in den Wirtsgraphen einzubetten ist. Es gibt viele verschiedene Ansätze, um Einbettungsüberführungen zu spezifizieren. Im folgenden gehen wir auf einige davon ein. Dabei beschränken wir uns hier aber zunächst auf **1-Kontext-Überführungen**. D.h. daß für die Einbettung nur die Knoten relevant sind, die unmittelbar mit dem zu ersetzenden Knoten durch Kanten verbunden sind. Es sei darauf hingewiesen, daß es auch mächtigere Ansätze gibt, bei denen Verbindungen zu Knoten hergestellt werden, die vom zu ersetzenden Knoten beliebig weit entfernt sind. Wir kommen darauf später noch zurück.

Um Einbettungsüberführungen zu definieren, gibt es zwei Möglichkeiten:

(1) **Globale Definition**: Für die gesamte Graphgrammatik gibt es eine Einbettungsüberführungsregel, die für alle Graphproduktionen gilt.

(2) **Lokale Definition**: Jeder Graphproduktion ist eine eigene Einbettungsüberführungsregel zugeordnet.

Wir diskutieren zunächst die globale Definition, die in einfachen Fällen ausreicht. Danach gehen wir auf die lokale Definition ein, die differenziertere Möglichkeiten bietet.

Def. 2.2.1 Terminale und nichtterminale Knotenmarkierungen

Es sei $\mathcal{N}_V \oplus \mathcal{T}_V = \mathcal{L}_V$ eine disjunkte Zerlegung der Knotenmarkierungsmenge. Dann bezeichnen wir Elemente aus $\mathcal{N}_V$ als nichtterminale und Elemente aus $\mathcal{T}_V$ als terminale Knotenmarkierungen. ■

Analog könnte man auch zwischen nichtterminalen und terminalen Kantenmarkierungen unterscheiden. Auf diese Unterscheidung wird hier verzichtet, weil sie im Spezialfall der kontextfreien Graphgrammatiken nicht sinnvoll erscheint. Kanten erscheinen ja nicht auf der linken Seite; ihre Ersetzung kann also nur indirekt über die Einbettungsüberführungsregel spezifiziert werden. Wir behandeln daher Kantenmarkierungen als terminale Markierungen, weisen aber darauf hin, daß bei beliebig gestalteter linker Seite eine Unterscheidung zwischen terminalen und nichtterminalen Kantenmarkierungen sinnvoll sein kann.

Def. 2.2.2 Kontextfreie Graphproduktion ohne Einbettungsüberführungsregel

Eine kontextfreie Graphproduktion (über $\mathcal{L}_V$ und $\mathcal{L}_E$) ohne Einbettungsüberführungsregel (im folgenden auch kurz als "Graphproduktion" bezeichnet) ist ein Paar

p = (L,R),

wobei folgendes gilt:

(1) Die rechte Seite R ist ein Graph über $\mathcal{L}_V$ und $\mathcal{L}_E$.

(2) Die linke Seite L ist ein Graph über $\mathcal{N}_V$ und $\emptyset$, der genau einen Knoten und keine Kante enthält. ■

Bsp. 2.2.3 Produktionen zur Erzeugung von Syntaxdiagrammen

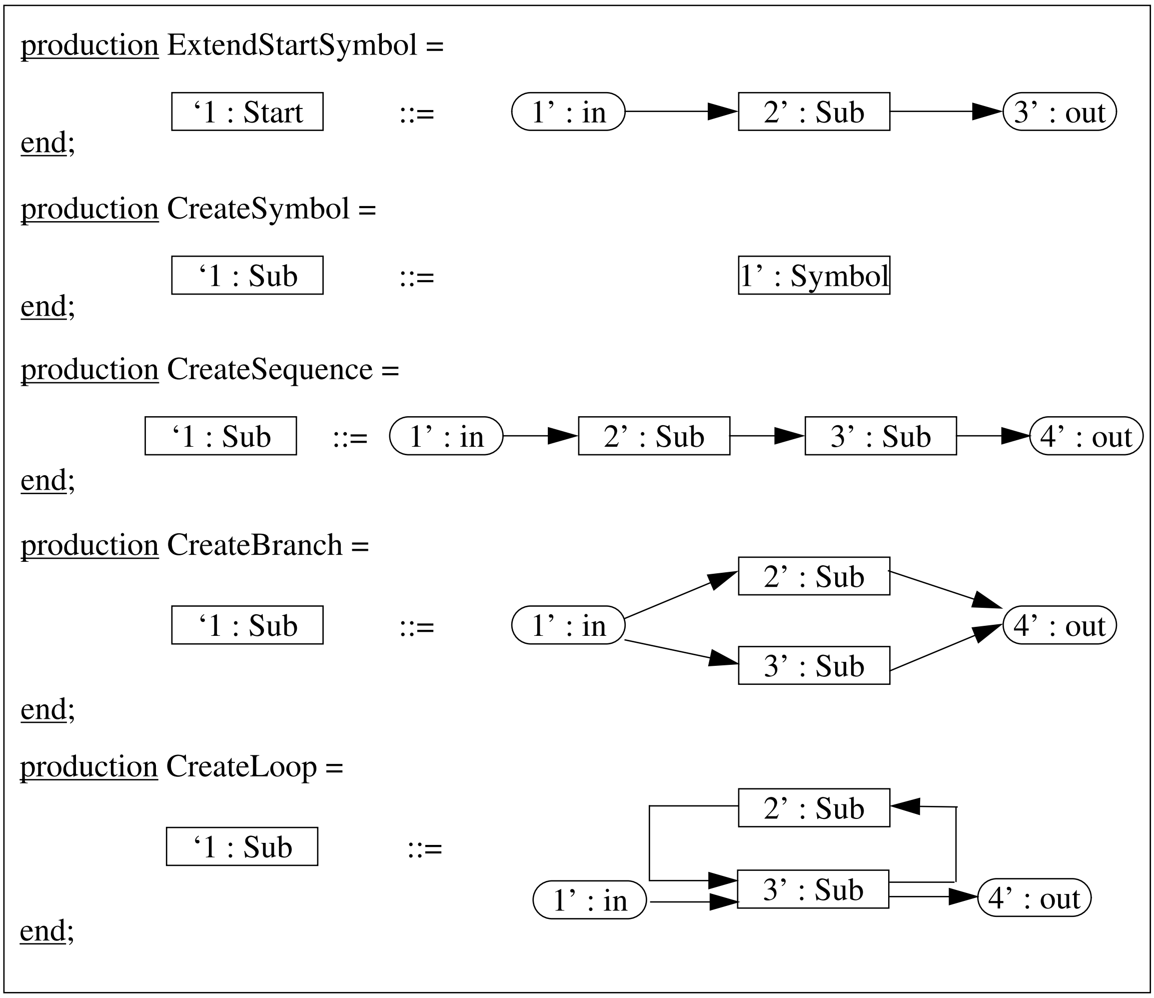


Abb. 2.4: Beispiel für kontextfreie Graphproduktionen

Die Produktionen aus Abb. 2.3 dienen dazu, den Aufbau wohlstrukturierter Syntaxdiagramme zu beschreiben (s. auch Bsp. 2.1.3 auf Seite 7). Man beachte, daß jede Produktion einen zusammenhängenden Untergraphen mit genau einem Eingangs- und genau einem

Ausgangsknoten erzeugt (die im Falle der Produktion "CreateSymbol" zusammenfallen). Ferner sei vermerkt, daß auf die Angabe von (trivialen) Produktionen, die Knoten mit der Markierung "Symbol" durch mit "A", "B", ... markierte Terminalknoten ersetzen, verzichtet wurde. ■

Wenden wir uns nun der Definition von Einbettungsüberführungen zu: Im Falle der globalen Definition wird die Einbettungsüberführung anhand von Markierungen und Richtungen definiert. Folgende Faktoren können dabei eine Rolle spielen:

(1) die Markierung des Kontextknotens,

(2) die Richtung und Markierung einer Kante, die zwischen einem Kontextknoten und dem zu ersetzenden Knoten verläuft,

(3) die Markierung des zu ersetzenden Knotens,

(4) die Markierung eines ersetzenden Knotens und

(5) die Richtung und Markierung einer Kante, die zwischen einem Kontextknoten und einem ersetzenden Knoten erzeugt wird.

Um all diesen Faktoren Rechnung zu tragen, wird eine Einbettungsüberführung folgendermaßen definiert:

Def. 2.2.4 Einbettungsüberführungsregel

Sei $IO = \{\mathrm{I},\mathrm{O}\}$, wobei "I" und "O" für "In" bzw. "Out" stehen.
Eine (globale) Einbettungsüberführungsregel ist eine Relation

$$\mathcal{EM} \subseteq \mathcal{L}_\mathrm{V} \times IO \times \mathcal{L}_\mathrm{E} \times \mathcal{N}_\mathrm{V} \times \mathcal{L}_\mathrm{V} \times IO \times \mathcal{L}_\mathrm{E},$$

wobei die Komponenten eines Tupels nacheinander folgende Bedeutung haben:

(1) Markierung des Kontextknotens
(2) Richtung der alten Kante
(3) Markierung der alten Kante
(4) Markierung des alten Knotens
(5) Markierung des neuen Knotens
(6) Richtung der neuen Kante
(7) Markierung der neuen Kante. ■

Bsp. 2.2.5 Einbettungsüberführungsregel für Syntaxdiagramme

Im Falle der Graphproduktionen zur Erzeugung von Syntaxdiagrammen (Bsp. 2.2.3) muß durch die Einbettungsüberführungsregel lediglich ausgedrückt werden, daß Kanten, die vor der Ersetzung in "Sub"-Knoten hinein- bzw. aus ihnen herausliefen, nach der Ersetzung

in Knoten mit der Markierung "in" oder "Symbol" hinein- bzw. aus Knoten mit der Markierung "out" oder "Symbol" herauslaufen:

$\mathcal{EM}$ = { (Sub, I, next, Sub, in, I, next),
(in, I, next, Sub, in, I, next)
(out, I, next, Sub, in, I, next),
(Symbol, I, next, Sub, in, I, next),
(Sub, I, next, Sub, Symbol, I, next),
(in, I, next, Sub, Symbol, I, next)
(out, I, next, Sub, Symbol, I, next),
(Symbol, I, next, Sub, Symbol, I, next),
. . . . . } ■

Das obige Beispiel zeigt bereits, daß die Möglichkeiten, die Def. 2.2.4 bietet, oft nicht ausgeschöpft werden. Im Beispiel bleiben Kantenmarkierung und -orientierung erhalten, und die Markierung des Kontextknotens spielt keine Rolle. Daher erscheint es sinnvoll, in solchen Fällen vereinfachte Notationen einzuführen. Wir kommen darauf später noch zurück.

Def. 2.2.6 Kontextfreie Graphgrammatik mit globaler Einbettungsüberführungsregel

Eine kontextfreie Graphgrammatik mit globaler Einbettungsüberführungsregel ist ein Tupel

$gr = (\mathcal{L}_V, \mathcal{L}_E, G_S, \mathcal{P}, \mathcal{EM})$

mit folgender Bedeutung:

(1) $\mathcal{L}_V = \mathcal{N}_V \oplus \mathcal{T}_V$ ist eine endliche Menge von nichtterminalen bzw. terminalen Knotenmarkierungen.
(2) $\mathcal{L}_E$ ist eine endliche Menge von (terminalen) Kantenmarkierungen.
(3) G_S ist der Startgraph, d.h. ein kantenloser Graph, der genau einen Knoten enthält, der mit dem Startsymbol $S \in \mathcal{N}_V$ markiert ist.
(4) $\mathcal{P}$ ist eine endliche Menge von kontextfreien Graphproduktionen über $\mathcal{L}_V$ und $\mathcal{L}_E$.
(5) $\mathcal{EM}$ ist eine globale Einbettungsüberführungsregel. ■

Bsp. 2.2.7 Graphgrammatik für wohlstrukturierte Syntaxdiagramme

Eine Graphgrammatik für wohlstrukturierte Syntaxdiagramme läßt sich folgendermassen definieren:

(1) $\mathcal{L}_V = \mathcal{N}_V \oplus \mathcal{T}_V$ mit $\mathcal{N}_V$ = {Start,Sub,Symbol}, $\mathcal{T}_V$ = {A,B,C,D,...}
(2) $\mathcal{L}_E$ = {next}

(3) $G_S = \{ \{1\}, \emptyset, \{(1,\text{Start})\} \}$

(4) $\mathcal{P}$ ist die Menge von Graphproduktionen aus Bsp. 2.2.3, wobei für jedes $X \in \mathcal{T}_V$ eine Produktion "CreateX" mit der Gestalt wie "CreateSymbol" existiert.

(5) $\mathcal{EM}$ ist die Einbettungsüberführungsregel aus Bsp. 2.2.5. ■

Die folgenden Definitionen zielen darauf ab, die Anwendung einer Produktion auf einen Graphen zu beschreiben. Dabei sind die Definitionen so allgemein gehalten, daß sie auch für Produktionen gelten, die nicht kontextfrei sind (beliebige Graphen als linke Seiten).

Def. 2.2.8 Anwendbarkeit einer Produktion

Es sei p = (L,R) eine Produktion, G ein Graph. p heißt anwendbar auf G $\Leftrightarrow$
Es gibt einen Untergraphen $G' \subseteq G$ mit $L \cong G'$.
G' wird dann ein Vorkommen von L in G genannt. ■

Im Falle kontextfreier Graphproduktionen bedeutet dies, daß der Nichtterminalknoten der linken Seite in G vorkommen muß, wobei Schlingen (also Kanten mit identischen Quell- und Zielknoten) verboten sind. Die Einschränkung auf Untergraphen soll sicherstellen, daß nicht versehentlich eine Produktion an einer Stelle angewendet wird, wo zusätzliche Kanten zwischen den Knoten des Vorkommens der linken Seite verlaufen. Der Graph, der ersetzt wird, sieht dann u.U. völlig anders aus als die linke Seite. Es sei aber bereits darauf hingewiesen, daß es auch Argumente dafür gibt, nur das Vorkommen eines Teilgraphen zu verlangen. Wir kommen darauf später noch zurück.

Def. 2.2.9 Einbettung eines Teilgraphen

Es sei $G \subset G'$. Die Einbettung von G in G' (in Zeichen: $\mathcal{C}(G,G')$, wobei $\mathcal{C}$ für "Kontext" steht) ist folgendermaßen definiert:

(1) $\mathcal{C}(G,G') \subset (\mathcal{V}' \setminus \mathcal{V}) \times \mathit{IO} \times \mathcal{L}_E \times \mathcal{V}$, wobei die Komponenten eines Tupels nacheinander folgende Bedeutung haben:
(1.1) Kontextknoten (im 1-Kontext von G)
(1.2) Orientierung der Kontextkante (I steht für "einlaufend", O für "auslaufend")
(1.3) Markierung der Kontextkante
(1.4) Knoten von G

(2) $\mathcal{C}(G,G') = \{(v',I,el,v) \mid v' \in (\mathcal{V}' \setminus \mathcal{V}) \wedge el \in \mathcal{L}_E \wedge v \in \mathcal{V} \wedge (v',el,v) \in \mathcal{E}'\} \cup$
$\{(v',O,el,v) \mid v' \in (\mathcal{V}' \setminus \mathcal{V}) \wedge el \in \mathcal{L}_E \wedge v \in \mathcal{V} \wedge (v,el,v') \in \mathcal{E}'\}$ ■

Def. 2.2.10 Ableitbarkeit vermöge einer Graphproduktion

Es sei gr = $(\mathcal{L}_V,\mathcal{L}_E,G_S,\mathcal{P},\mathcal{EM})$ eine Graphgrammatik; $p \in \mathcal{P}$; $G,G' \in \mathcal{G}(\mathcal{L}_V,\mathcal{L}_E)$. Dann heißt G' ableitbar aus G vermöge p (in Zeichen: G ~p~> G') $\Leftrightarrow$

(1) p ist anwendbar auf G.

(2) Sei G_L ein Vorkommen der linken Seite L von p, $\mathcal{C}(G_L,G)$ die Einbettung von G_L in G. Dann gibt es Graphen G_1, G_2, G_3 mit folgenden Eigenschaften:
(2.1) $G_1 = G \setminus G_L$ (Löschen der linken Seite).
(2.2) $G_2 = G_1 \oplus G_R$ (Einfügen der rechten Seite). Dabei sei $G_R \cong R$.
(2.3) $G_3 = (\mathcal{V}_2, \mathcal{E}_2 \cup \mathcal{C}, l_2)$, wobei $\mathcal{C}$ folgendermaßen definiert ist:

$$
\begin{aligned}
\mathcal{C} = {} & \mathcal{C}_I \cup \mathcal{C}_O \text{ mit} \\
\mathcal{C}_I = {} & \{(v_1,el',v_{GR}) \mid v_1 \in \mathcal{V}_1 \wedge el' \in \mathcal{L}_E \wedge v_{GR} \in \mathcal{V}_{GR} \wedge \\
& \quad \exists\, v_{GL} \in \mathcal{V}_{GL}\ \exists\, IO \in \mathcal{IO}\ \exists\, el \in \mathcal{L}_E : \\
& \quad (v_1,IO,el,v_{GL}) \in \mathcal{C}(G_L,G) \wedge \\
& \quad (l_1(v_1),IO,el,l_{GL}(v_{GL}),l_{GR}(v_{GR}),I,el') \in \mathcal{EM}\} \\
\mathcal{C}_O = {} & \{(v_{GR},el',v_1) \mid v_1 \in \mathcal{V}_1 \wedge el' \in \mathcal{L}_E \wedge v_{GR} \in \mathcal{V}_{GR} \wedge \\
& \quad \exists\, v_{GL} \in \mathcal{V}_{GL}\ \exists\, IO \in \mathcal{IO}\ \exists\, el \in \mathcal{L}_E : \\
& \quad (v_1,IO,el,v_{GL}) \in \mathcal{C}(G_L,G) \wedge \\
& \quad (l_1(v_1),IO,el,l_{GL}(v_{GL}),l_{GR}(v_{GR}),O,el') \in \mathcal{EM}\}
\end{aligned}
$$

(3) $G' \cong G_3$. ■

Aus dieser Definition folgt insbesondere, daß G' bis auf Isomorphie eindeutig bestimmt ist, wenn die Anwendungsstelle vollständig fixiert ist (d.h. G_L und der Isomorphismus von L nach G_L gegeben sind).

Die Einbettung der rechten Seite in den Wirtsgraphen wird folgendermaßen berechnet (s. Punkt (2.3) der obigen Definition): Ziehe eine mit el' markierte Kante vom Kontextknoten v_1 zum Knoten der rechten Seite v_{GR} (analog im Falle auslaufender Kanten!), falls v_1 mit einem Knoten v_{GL} des Vorkommens der linken Seite durch eine Kante mit der Markierung el verbunden war und die Einbettungsüberführungsregel eine "geeignete" Komponente enthält. Dabei sind neben der Richtung und Markierung der Kanten auch die Markierungen der beteiligten Knoten zu berücksichtigen.

Betrachtet man die obige Definition, so wird deutlich, daß deren Komplexität i.w. durch die Einbettungsüberführung verursacht wird. Die Definition ist deshalb so komplex, weil sie alle i.a. denkbaren Fälle abdecken muß. In vielen praktisch vorkommenden Situationen treten jedoch nur einfache Spezialfälle auf, wie das folgende Beispiel zeigt:

Bsp. 2.2.11 Anwendung einer Graphproduktion

In Abb. 2.5 wird die Graphproduktion "CreateBranch" aus Abb. 2.3 auf den Knoten 2 angewendet. Nach dem Löschen der linken und dem Einfügen der rechten Seite wird deren Einbettung folgendermaßen ermittelt:

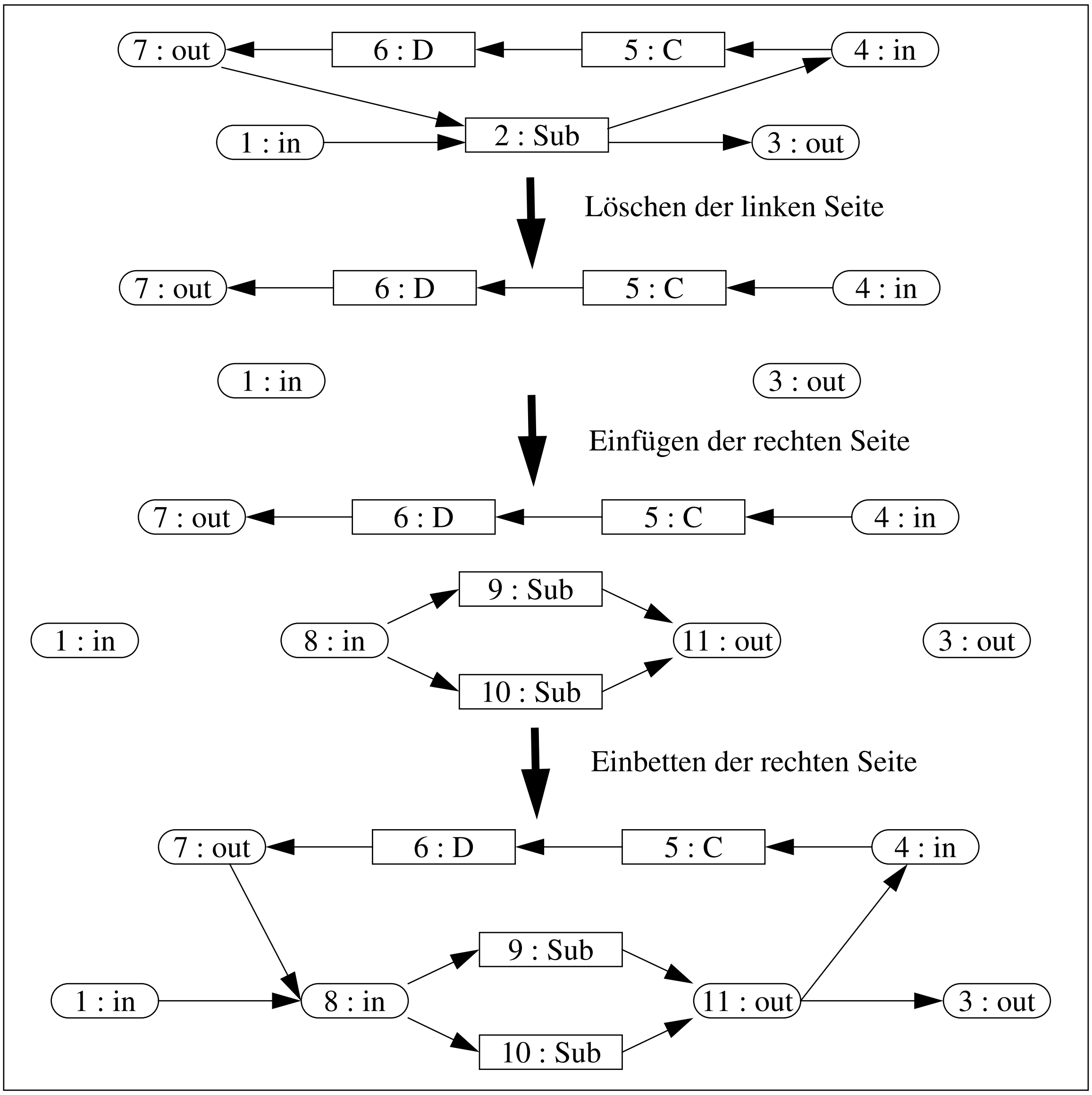


Abb. 2.5: Beispiel für die Anwendung einer Graphproduktion

(1) $\mathcal{C}(G_L,G) = \{(7,I,next,2), (1,I,next,2), (4,O,next,2), (3,O,next,2)\}$

(2) Betrachtet man die Relation $\mathcal{EM}$ aus Bsp. 2.2.5, so erkennt man, daß diese nur richtungs- und kantenmarkierungserhaltende Komponenten enthält. Folgende Komponenten sind hier relevant:

$\mathcal{EM}$ = { (out, I, next, Sub, in, I, next)
(in, I, next, Sub, in, I, next),
(in, O, next, Sub, out, O, next),
(out, O, next, Sub, out, O, next) . . . }

(3) Die Einbettung $\mathcal{C}$ der rechten Seite ergibt sich damit zu:
$\mathcal{C}$ = {(7,next,8), (1,next,8), (11,next,4), (11,next,3)} ■

Def. 2.2.12 Ableitbarkeit

Es sei gr = ($\mathcal{L}_V$,$\mathcal{L}_E$,G_S,$\mathcal{P}$,$\mathcal{EM}$) eine (kontextfreie) Graphgrammatik, G,G' ∈ $\mathcal{G}$($\mathcal{L}_V$,$\mathcal{L}_E$).
G' heißt ableitbar aus G (G ~> G') ⇔ ∃ p ∈ $\mathcal{P}$: G ~p~> G'.
Ferner wird der reflexive und transitive Abschluß von ~> mit $\overset{*}{\leadsto}$ bezeichnet. ■

Def. 2.2.13 Graphsatzformen

Sei gr = ($\mathcal{L}_V$,$\mathcal{L}_E$,G_S,$\mathcal{P}$,$\mathcal{EM}$) eine (kontextfreie) Graphgrammatik. Dann ist die Menge der von gr erzeugten Graphsatzformen folgendermaßen definiert:
$\mathcal{GS}$ (gr) = { G ∈ $\mathcal{G}$ ($\mathcal{L}_V$,$\mathcal{L}_E$) | $G_S \overset{*}{\leadsto}$ G } ■

Def. 2.2.14 Graphsprache

Sei gr wie oben. Dann ist die von gr erzeugte Sprache folgendermaßen definiert:
$\mathcal{L}$ (gr) = { G ∈ $\mathcal{G}$ ($\mathcal{T}_V$,$\mathcal{L}_E$) | G ∈ $\mathcal{GS}$ (gr) } ■

Bisher haben wir Graphgrammatiken mit globaler Einbettungsüberführung betrachtet. Eine andere Möglichkeit besteht darin, Einbettungsüberführungen **lokal** zu definieren, d.h. den einzelnen Graphproduktionen zuzuordnen:

Def. 2.2.15 Kontextfreie Graphproduktion mit Einbettungsüberführung

Eine kontextfreie Graphproduktion mit Einbettungsüberführung ist ein Tripel
p = (L,R,$\mathcal{EM}$),
wobei folgendes gilt:

(1) Die rechte Seite R ist ein Graph über $\mathcal{L}_V$ und $\mathcal{L}_E$.
(2) Die linke Seite L ist ein Graph über $\mathcal{N}_V$ und ∅, der genau einen Knoten und keine Kante enthält.
(3) $\mathcal{EM}$ ist die (unten noch zu definierende) Einbettungsüberführungsregel. ■

Alles weitere läßt sich analog definieren (Graphgrammatik, Ableitbarkeit, Graphsatzformen, Sprache). Auf eine Ausführung dieser Definitionen soll hier verzichtet werden. Wir beschränken uns im folgenden darauf, zu diskutieren, wie sich Einbettungsüberführungen lokal definieren lassen.

Eine Möglichkeit besteht darin, die Definition der globalen Einbettungsüberführung (s. Def. 2.2.4) geeignet abzuwandeln. Dabei kann (im Falle kontextfreier Produktionen) auf die Angabe der Markierung des Knotens der linken Seite verzichtet werden. Das folgende Beispiel illustriert diese Art der Definition lokaler Einbettungsüberführungen, wobei eine PROGRESS-ähnliche Notation verwendet wird.

Bsp. 2.2.16 Lokale Einbettungsüberführungsregel

Die Produktion in Abb. 2.6 beschreibt die Erzeugung einer Sequenz in einem Syntaxdiagramm (vgl. Bsp. 2.2.3 und Bsp. 2.2.5). Dabei bezeichnen in der Einbettungsüberführungsregel "<-" und "->" ein- bzw. auslaufende Kanten. ■

```
production CreateSequence =

    '1 : Sub    ::=   (1' : in) ---> 2' : Sub ---> 3' : Sub ---> (4' : out)

  embedding
    copy <-next- Sub   as <-next- to in,    <-next- in       as <-next- to in,
         <-next- out   as <-next- to in,    <-next- Symbol   as <-next- to in,
         -next-> Sub   as -next-> to out,   -next-> in       as -next-> to out,
         -next-> out   as -next-> to out,   -next-> Symbol   as -next-> to out;

end;
```

Abb. 2.6: Graphproduktion mit lokaler Einbettungsüberführungsregel

Diese **Notation** der Einbettungsüberführungsregel ist noch recht **umständlich**, da in diesem Fall Kantenrichtung und -markierung erhalten bleiben und die Markierung des Kontextknotens keine Rolle spielt. Mit Hilfe entsprechender "Default"-Vereinbarungen kann man die Notation im Beispiel folgendermaßen vereinfachen:

copy <-next- to in, -next-> to out;

Vergleichen wir **lokale** und **globale Einbettungsüberführungsregeln**, so stellen wir folgendes fest:

(1) Globale Einbettungsüberführungsregeln sind vor allem geeignet, wenn in allen Produktionen i.w. die gleichen einfachen Regeln gelten.

(2) Lokale Einbettungsüberführungsregeln bieten differenziertere Ausdrucksmöglichkeiten, da ein- und auslaufende Kanten in verschiedenen Produktionen unterschiedlich behandelt werden können.

(3) Lokale Einbettungsüberführungsregeln sind in der Regel leichter lesbar, weil der Sinn einer Einbettungsüberführungsregel oft erst bei Betrachtung einer konkreten Produktion deutlich wird.

In PROGRESS wurde den **lokalen Einbettungsüberführungsregeln** der Vorzug gegeben. Dabei wurde allerdings hinsichtlich der Identifikation der Knoten der rechten Seite (und auch der linken Seite bei mehrknotigen Ersetzungen) ein anderer Weg beschritten als der oben angedeutete: Statt Knoten implizit durch ihre Markierung zu identifizieren, werden sie **explizit** angesprochen. Dies führt im obigen Beispiel zu folgender Formulierung der Einbettungsüberführungsregel:

copy <-next- to 1’, -next-> to 4’;

Die explizite Identifikation von Knoten ist zum einen leichter lesbar, zum anderen bietet sie differenziertere Ausdrucksmöglichkeiten, wenn auf einer Seite der Produktion mehrere Knoten mit der gleichen Markierung auftreten. Letzteres zeigt das folgende Beispiel:

Bsp. 2.2.17 Explizite Benennung von Knoten in einer Einbettungsüberführungsregel

production CreateSequence =
‘1 : Sub ::= 1’ : Sub → 2’ : Sub
embedding copy <-next- to 1’, -next-> to 2’;
end;

Abb. 2.7: Explizite Benennung von Knoten in der Einbettungsüberführung

Die Produktion "CreateSequence" erzeugt einen Untergraphen, der zusammenhängend ist und genau eine Quelle bzw. Senke hat. Diese Eigenschaft bleibt auch dann erhalten, wenn man den "in"- und den "out"-Knoten wegläßt, so daß die rechte Seite zwei "Sub"-Knoten und eine sie verbindende Kante umfaßt (Abb. 2.7). Die Einbettungsüberführungsregel legt fest, daß einlaufende Kanten in den Knoten 1’ gelenkt werden und auslaufende Kanten aus dem Knoten 2’ herauslaufen. Man beachte, daß im Falle der impliziten Identifikation sich die Knoten 1’ und 2’ nicht unterscheiden ließen, da sie gleich markiert sind. ■

Man beachte, daß die Bezugnahme auf konkrete Knoten in Hinsicht auf die Klasse der erzeugbaren Sprachen keinen Gewinn bringt. Die explizite Identifikation von Knoten in Einbettungsüberführungsregeln läßt sich simulieren, indem man zusätzliche Nichtterminalsymbole (z.B. "Sub_1" und "Sub_2") und entsprechende Zwischenproduktionen einführt. Analog lassen sich auch lokale durch globale Einbettungsüberführungsregeln simulieren. Somit ist die Auswahl eines der hier vorgestellten Mechanismen "nur" eine pragmatische Frage.

2.3 Syntaxanalyse für kontextfreie Graphgrammatiken

In vielen Anwendungen, insbesondere in der Mustererkennung, stellt sich das Problem, die **syntaktische Struktur** eines Graphen zu **analysieren**: Für einen Graphen G und eine Grammatik gr ist die Frage, ob $G \in \mathcal{L}(gr)$ gilt, zu beantworten und im Erfolgsfall eine Ableitung vom Startgraphen G_S nach G zu konstruieren.

Es ist erheblich schwieriger, Graphen zu analysieren, als eine Syntaxanalyse für Zeichenketten durchzuführen. Analyseverfahren für Zeichenketten nutzen die lineare Anordnung der Zeichen aus, während eine solche Ordnung auf den Knoten eines Graphen i.a. nicht gegeben ist. Zeichenketten werden von links nach rechts analysiert, wobei Präfixe des Eingaberests dazu benutzt werden, Expansions- bzw. Reduktionsentscheidungen zu treffen. In Graphen sind dagegen die Begriffe "links", "rechts" und "Präfix" sinnlos.

Insbesondere ist es schwierig, Graphen mit vertretbarem Aufwand zu analysieren. Bisher gibt es nur wenige Ansätze, in denen dies gelungen ist. Wir beschränken uns hier auf die **Präzedenzanalyseverfahren,** die es gestatten, Graphen in linearer Zeit zu analysieren (linear bezüglich der Anzahl der Knoten und Kanten).

Der Name "Präzedenzanalyse" reflektiert die Analogie zu der Präzedenzanalyse für Zeichenketten, die insbesondere im Bereich von Ausdrücken in Programmiersprachen eingesetzt wird. Die grundlegende Idee besteht darin, Präzedenzen zwischen Grammatiksymbolen zu definieren. Diese Präzedenzen werden typischerweise dazu benutzt, um den Operatorenvorrang auszudrücken, d.h. um die Analyse so zu steuern, daß Operatoren höherer Priorität vor den Operatoren niedrigerer Priorität ausgewertet werden.

Die **Präzedenzanalyse** für **Zeichenketten** ist ein Bottom-up-Analyseverfahren, dem folgendes Prinzip zugrunde liegt: Gegeben sei ein zu analysierendes Wort

$$w = x_1 \ldots x_n,$$

wobei die x_i lexikalische Einheiten (d.h. Terminalsymbole der kontextfreien Syntax) sind. Zwischen die Terminalsymbole werden gemäß einer Präzedenztafel **Präzedenzrelationen** $<$, $\equiv$ und $>$ mit folgender Interpretation geschrieben:

(1) $<$ kennzeichnet den Anfang eines **Henkels** (d.h. einer zu reduzierenden rechten Seite).

(2) $\equiv$ kennzeichnet aufeinanderfolgende Symbole einer rechten Seite.

(3) $>$ kennzeichnet das Ende einer rechten Seite.

Dies führt zu einem Wort

$$w' = \ < x_1\ r_1 \ldots x_{n-1}\ r_{n-1}\ x_n >,$$

wobei $r_i \in \{ <, \equiv, > \}$. Nun wird von links beginnend nach dem ersten Teilwort

$$w'' = \ < x_i \equiv \ldots \equiv x_j >$$

gesucht. Entfernt man daraus die Präzedenzrelationen, so erhält man die rechte Seite einer Produktion

$A ::= x_i \ldots x_j.$

Nun wird reduziert, d.h. die rechte Seite durch die linke ersetzt und das Verfahren iterativ fortgesetzt.

Die **Präzedenzanalyse** für **Graphen** verläuft analog. Die Analogie wird deutlich, wenn wir Zeichenketten wie zu Beginn von Abschnitt 2.2 (s. Abb. 2.3 auf Seite 10) als Graphen darstellen. Die Präzedenzrelationen werden den Kanten zugeordnet. Es wird ein Untergraph G_R gesucht, der folgende Bedingungen erfüllt:

(1) Allen Kanten von G_R ist die Relation $\equiv$ zugeordnet.

(2) Einlaufende Kanten sind mit $<$ gekennzeichnet.

(3) Auslaufende Kanten tragen die Kennzeichnung $>$.

Ordnet man die Knoten eines Graphen dreidimensional an, so kann man die Suche nach einem Henkel als die Suche nach einem Hochplateau interpretieren, das nur über aufsteigende Kanten erreicht und über absteigende wieder verlassen werden kann. Abb. 2.8 veranschaulicht diese "Maximum-Charakterisierung" für den einfachen Fall der Zeichenketten-Graphen; hier reicht eine zweidimensionale Darstellung aus.

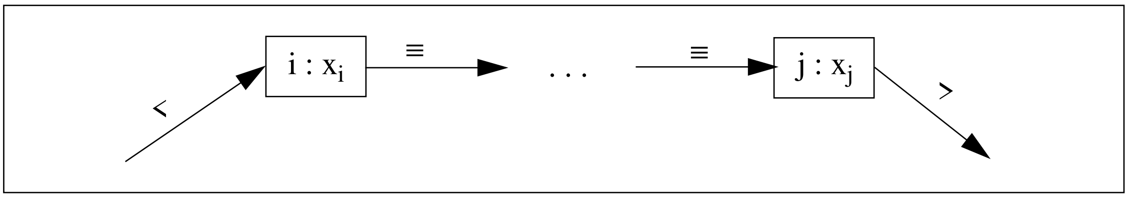


Abb. 2.8: Charakterisierung eines Henkels

Im folgenden stellen wir nun eine der verschiedenen Varianten von Präzedenzanalyseverfahren vor. Die hier dargestellte Variante erhebt nicht den Anspruch, optimal zu sein (z.B. hinsichtlich der Klasse der analysierbaren Sprachen oder der Laufzeit- bzw. Speicherplatzeffizienz); die wesentlichen Ideen der Präzedenzanalyse lassen sich aber anhand dieses Verfahrens gut demonstrieren.

Um die Präzedenzanalyse anwenden zu können, muß die zugrundeliegende Graphgrammatik bestimmte Bedingungen erfüllen. Eine Grammatik, die diesen Bedingungen genügt, nennen wir **Präzedenz-Graphgrammatik**. Welche Bedingungen im einzelnen gelten müssen, wird im folgenden diskutiert. Anschließend wird als Resümee der Begriff der Präzedenz-Graphgrammatik formal definiert.

Die Präzedenzanalyse läßt sich nur anwenden, wenn folgende **Zusammenhangs-Eigenschaften** erfüllt sind:

(1) Der Startgraph ist zusammenhängend (dies steckt schon in der Definition einer kontextfreien Graphgrammatik, s. Def. 2.2.6 auf Seite 14).

(2) Alle rechten Seiten von Graphproduktionen sind zusammenhängend.

(3) Die Einbettungsüberführungsregeln garantieren, daß für einen Nichtterminalknoten eingesetzte rechte Seiten niemals isoliert sind (s.u.).

Bedingungen (1), (2) und (3) zusammen garantieren, daß Graphsatzformen zusammenhängend sind. Somit können Henkel durch lokales Suchen ermittelt werden, wobei man an einem beliebigen Knoten startet. Ferner garantiert Bedingung (2), daß die Henkelsuche abgebrochen werden kann, wenn ein zusammenhängendes Hochplateau gefunden wurde.

Um die **Sackgassenfreiheit** der Analyse zu erreichen, müssen insbesondere folgende Bedingungen erfüllt sein:

(1) Es darf keine Graphproduktionen mit zueinander isomorphen rechten Seiten geben.

(2) Graphproduktionen müssen eindeutig umkehrbar sein.

Während Stringproduktionen stets eindeutig umkehrbar sind, wenn der Henkel identifiziert ist, muß die **eindeutige Umkehrbarkeit** von **Graphproduktionen** explizit gefordert werden. Zwar ist die einzusetzende linke Seite stets eindeutig bestimmt, ihre Einbettung läßt sich jedoch nicht immer eindeutig rekonstruieren. Dies gilt beispielsweise, wenn im Zuge der Einbettungsüberführung Kanten gelöscht werden.

Bsp. 2.3.1 Mehrdeutige Umkehrung einer Graphproduktion

Es seien $t \in \mathcal{T}_V$, $A \in \mathcal{N}_V$, $el_1, el_2 \in \mathcal{L}_V$. In Abb. 2.9 sind angegeben:

(1) eine Graphproduktion, deren Einbettungsüberführungsregel beschreibt, daß (nur) einlaufende Kanten mit der Markierung el_1 übernommen werden,

(2) ein zu reduzierender Graph G,

(3) verschiedene Möglichkeiten der Reduktion von Knoten 2. ■

Folgende Bedingung ist hinreichend, um die eindeutige Umkehrbarkeit einer Graphproduktion zu gewährleisten (**Kantenvererbung**): Alle Kontextkanten des zu ersetzenden Nichtterminalknotens werden auf einen oder mehrere Knoten der rechten Seite übertragen, wobei Richtung und Markierung erhalten bleiben. Um dies zu erreichen, wird die Gestalt von Einbettungsüberführungsregeln eingeschränkt (wir betrachten hier lokale Einbettungsüberführungsregeln mit expliziter Knotenidentifikation):

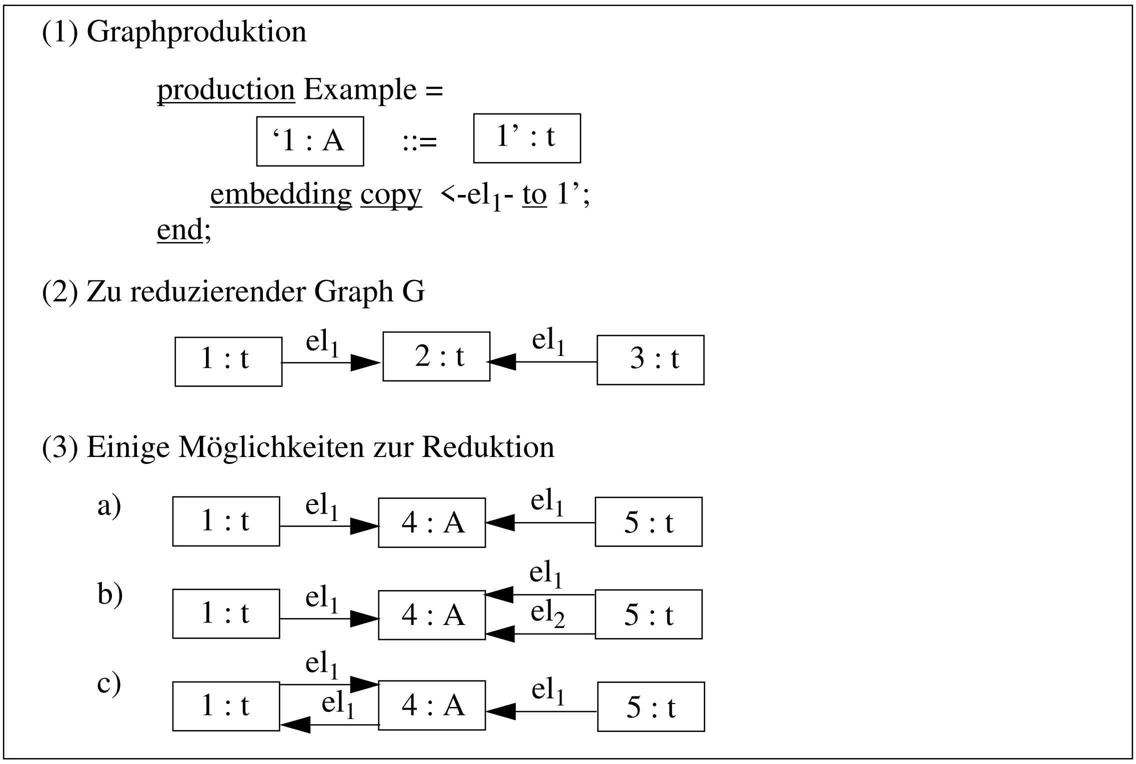


Abb. 2.9 Nicht eindeutige Umkehrbarkeit einer Graphproduktion

Def. 2.3.2 Kontextfreie Graphproduktion mit elementarer Einbettung

Eine kontextfreie Graphproduktion mit elementarer Einbettungsüberführungsregel ist ein Tripel

$p = (L, R, \mathcal{EM})$,

wobei L und R wie üblich definiert sind und für $\mathcal{EM}$ folgendes gilt:

$\mathcal{EM} \subseteq IO \times \mathcal{L}_E \times \mathcal{V}_R$,

wobei die Komponenten eines Tripels folgende Bedeutung haben:

(1) Richtung der Kontext- und der zu erzeugenden Kante,

(2) Markierung der Kontext- und der zu erzeugenden Kante,

(3) Knoten der rechten Seite, der als Quell- bzw. Zielknoten der zu erzeugenden Kante fungiert. ■

Def. 2.3.3 Monotone Einbettungsüberführungsregel

Eine elementare Einbettungsüberführungsregel $\mathcal{EM}$ heißt monoton $\Leftrightarrow$ es werden keine Kanten gelöscht:

$$\forall\ \mathrm{IO} \in \mathit{IO}\ \forall\ \mathrm{el} \in \mathcal{L}_E\ \exists\ v_R \in \mathcal{V}_R : (\mathrm{IO}, \mathrm{el}, v_R) \in \mathcal{EM}\ \blacksquare$$

Bsp. 2.3.4 Graphproduktionen mit monotoner Einbettungsüberführungsregel

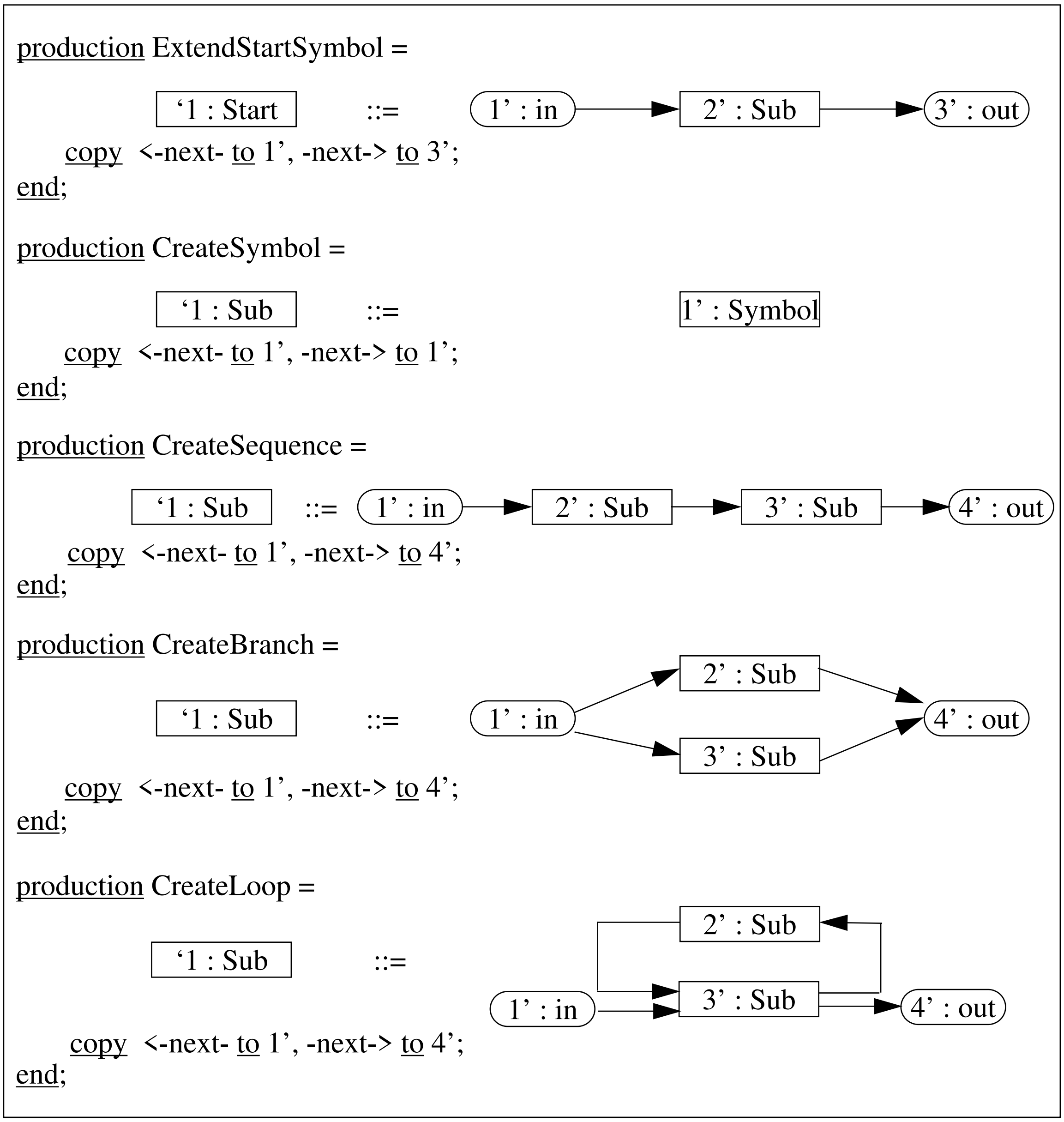


Abb. 2.10 : Beispiel für monotone Graphproduktionen

Abb. 2.3 zeigt Produktionen zur Erzeugung von Syntaxdiagrammen (s. auch Abb. 2.3 auf Seite 10). Der Einfachheit halber wird "Symbol" hier als Terminalsymbol behandelt. Man beachte, daß nur die Graphproduktion "ExtendStartSymbol" überflüssige Einbettungs-überführungskomponenten enthält, die eingefügt werden mußten, um die Monotonie-Eigenschaft sicherzustellen. ■

Eine Graphgrammatik, für die sich die Präzedenzanalyse einsetzen läßt, muß insbesondere **präzedenz-konfliktfrei** sein. Dies bedeutet, daß sich jeder Kante eine Präzedenz eindeutig zuordnen läßt. Wie wir sehen werden, wird neben den bisher erwähnten Präzedenzrelationen $\equiv$, $<$ und $>$ noch eine weitere benötigt, die wir mit $<>$ bezeichnen (näheres dazu s.u.).

Def. 2.3.5 Präzedenzrelationen (informelle Fassung)

Die Relationen
$\equiv, <, >, <> \; \subseteq \mathcal{L}_V \times \mathcal{L}_E \times \mathcal{L}_V$
werden als Präzedenzrelationen bezeichnet. ■

Die Präzedenz einer Kante hängt also von ihrer Markierung und von der Markierung der Quell- und Zielknoten ab. Im einzelnen haben die Präzedenzrelationen folgende Bedeutung (sl, el, tl seien die Markierungen des Quellknotens, der Kante und des Zielknotens):

(1) (sl, el, tl) $\in \equiv \Leftrightarrow$ es gibt eine rechte Seite, die eine mit el markierte Kante enthält, die mit sl und tl markierte Knoten verbindet.

(2) (sl, el, tl) $\in < \Leftrightarrow$ el-Kanten zwischen sl- und tl-Knoten treten als Kanten auf, die in Henkel hineinlaufen (aber nicht aus Henkeln herauslaufen).

(3) (sl, el, tl) $\in > \Leftrightarrow$ el-Kanten zwischen sl- und tl-Knoten treten als Kanten auf, die aus Henkeln herauslaufen (aber nicht in Henkel hineinlaufen).

(4) (sl, el, tl) $\in <> \Leftrightarrow$ el-Kanten zwischen sl- und tl-Knoten kommen sowohl als einlaufende als auch als auslaufende Kanten von Henkeln vor.

Für eine Kante e in einer zu reduzierenden Graphsatzform bedeutet dies folgendes:

(1) e hat die Präzedenz $\equiv \Leftrightarrow$ Quell- und Zielknoten sind im selben Schritt zu reduzieren.

(2) e hat die Präzedenz $< \Leftrightarrow$ der Zielknoten muß vor dem Quellknoten reduziert werden.

(3) e hat die Präzedenz $> \Leftrightarrow$ der Quellknoten muß vor dem Zielknoten reduziert werden.

(4) e hat die Präzedenz $<> \Leftrightarrow$ Quell- und Zielknoten können in beliebiger Reihenfolge reduziert werden.

Die **Präzedenzrelationen** werden anhand der vorgegebenen Graphgrammatik **berechnet**. Im Falle der Relation ≡ ist die Berechnung trivial: man inspiziere alle rechten Seiten und nehme für jede Kante (v, el, v’) das Tripel (l(v), el, l(v’)) in die Relation ≡ auf.

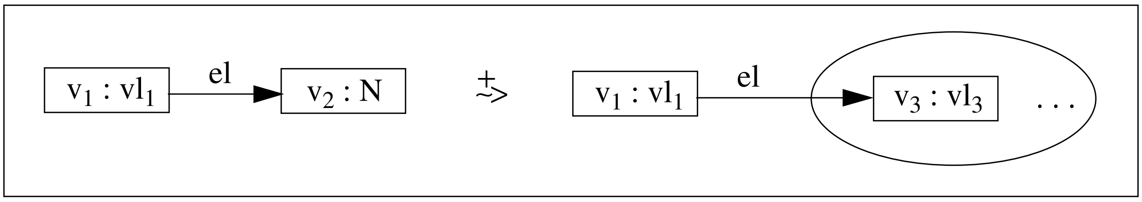


Abb. 2.11: Zur Berechnung von <

Die Relation < läßt sich folgendermaßen berechnen (für > läßt sich analog argumentieren): Man betrachte alle Kanten auf rechten Seiten mit nichtterminalen Zielknoten (Abb. 2.11). Wird ein solcher Zielknoten v_2 expandiert, so werden alle Kontextkanten auf die Knoten des für v_2 eingesetzten Untergraphen vererbt. Ist (v_1, el, v_3) eine solche Kontextkante, so nehme man (vl_1, el, vl_3) in die Relation < auf. Man beachte, daß bei der Berechnung nichtleere Ableitungen beliebiger Länge zu berücksichtigen sind.

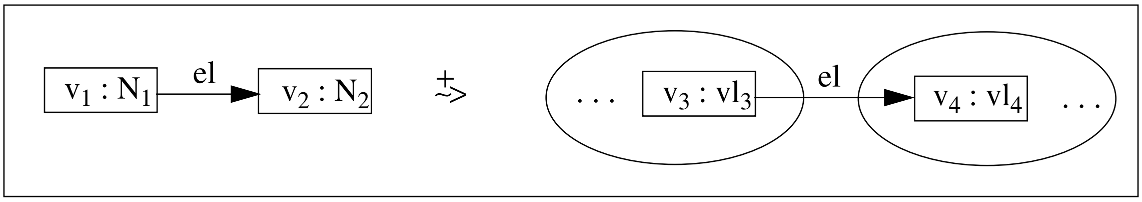


Abb. 2.12: Zur Berechnung von <>

Für die Berechnung von <> sind solche Kanten rechter Seiten relevant, deren Quell- und Zielknoten nichtterminal sind (Abb. 2.12). Werden beide Knoten expandiert, so entstehen Kanten, die Knoten der dafür eingesetzten Untergraphen verbinden. Ihnen ist die Präzedenz <> zuzuordnen, weil sie in beliebiger Reihenfolge reduziert werden können. Man beachte, daß es i.a. nicht gleichgültig ist, in welcher Reihenfolge reduziert wird. Die Unabhängigkeit von der Reihenfolge ist der Tatsache zu verdanken, daß die Einbettungsüberführungsregeln eine sehr einfache Gestalt besitzen (Monotonie-Eigenschaft).

Um die Relationen <, > und <> aus der Grammatik zu berechnen, führen wir zwei Hilfsrelationen

$$In, Out \subseteq \mathcal{N}_V \times \mathcal{L}_E \times \mathcal{L}_V$$

mit folgender Bedeutung ein:

(1) (N, el, vl) $\in$ *In* $\Leftrightarrow$ es gibt eine Produktion, die einen N-Knoten auf der linken Seite hat und eine einlaufende el-Kante auf einen vl-Knoten der rechten Seite überträgt.

(2) (N, el, vl) $\in$ *Out* $\Leftrightarrow$ es gibt eine Produktion, die einen N-Knoten auf der linken Seite hat und eine auslaufende el-Kante auf einen vl-Knoten der rechten Seite überträgt.

Def. 2.3.6 Hilfsrelationen *In*, *Out*

Sei (hier und im folgenden)

$$gr = (\mathcal{L}_V, \mathcal{L}_E, G_S, \mathcal{P})$$

eine kontextfreie Graphgrammatik mit monotonen Einbettungsüberführungsregeln. Dann sind die Relationen *In*, *Out* folgendermaßen definiert:

(1) $$In = \{ (N, el, vl) \mid (N, el, vl) \in \mathcal{N}_V \times \mathcal{L}_E \times \mathcal{L}_V \ \wedge \ \exists\, p \in \mathcal{P} : p = (L, R, \mathcal{EM}) \wedge l_L(v_L) = N \ \wedge \ \exists\, v_R \in V_R : l_R(v_R) = vl \ \wedge \ (I, el, v_R) \in \mathcal{EM} \}$$

(2) $$Out = \{ (N, el, vl) \mid (N, el, vl) \in \mathcal{N}_V \times \mathcal{L}_E \times \mathcal{L}_V \ \wedge \ \exists\, p \in \mathcal{P} : p = (L, R, \mathcal{EM}) \wedge l_L(v_L) = N \ \wedge \ \exists\, v_R \in V_R : l_R(v_R) = vl \ \wedge \ (O, el, v_R) \in \mathcal{EM} \} \ \blacksquare$$

Die Relationen *In*, *Out* beschreiben nur die Vererbung von Kanten über einen Ableitungsschritt. Die Vererbung über beliebig viele Ableitungsschritte läßt sich mit Hilfe eines transitiven Abschlusses formalisieren, der - zusammen mit weiteren Operationen - wie folgt definiert wird:

Def. 2.3.7 Verkettung, Potenz, transitiver Abschluß, Inverse

Seien

$$R, R_1, R_2 \subseteq \mathcal{L}_V \times \mathcal{L}_E \times \mathcal{L}_V.$$

(1) Die Verkettung von R_1 und R_2 ist folgendermaßen definiert:
$$R_1 \circ R_2 = \{ (vl, el, vl') \mid \exists\, vl'' \in \mathcal{L}_V : (vl, el, vl'') \in R_1 \ \wedge \ (vl'', el, vl') \in R_2 \}$$

(2) Für die Potenz gilt:
$$R^1 = R \ \text{ bzw. } \ R^{i+1} = R^i \circ R \ \ (i > 0)$$

(3) Der transitive Abschluß ist wie folgt definiert:
$$R^+ = R^1 \cup R^2 \cup R^3 \ldots$$

(4) Schließlich gilt für die Inverse:
$$R^{-1} = \{ (vl', el, vl) \mid (vl, el, vl') \in R \} \ \blacksquare$$

Nach diesen Vorbereitungen sind wir nun in der Lage, die Präzedenzrelationen auszurechnen und den Begriff der Präzedenz-Graphgrammatik zu definieren:

Def. 2.3.8 Präzedenz-Graphgrammatik

Es sei gr = ($\mathcal{L}_V$, $\mathcal{L}_E$, G_S, $\mathcal{EM}$) eine kontextfreie, monotone Graphgrammatik. gr heißt Präzedenz-Graphgrammatik ⇔

(1) Alle rechten Seiten von Produktionen sind zusammenhängend.

(2) Es gibt keine Produktionen mit zueinander isomorphen rechten Seiten.

(3) gr ist präzedenz-konfliktfrei, d.h. die folgenden Präzedenzrelationen sind paarweise disjunkt:

$$(3.1)\ \equiv\ = \{\ (vl, el, vl')\ |\ (vl, el, vl') \in \mathcal{L}_V \times \mathcal{L}_E \times \mathcal{L}_V\ \wedge\ \exists\, p \in \mathcal{P} : p = (L, R, \mathcal{EM})\ \wedge\ \exists\, v, v' \in \mathcal{V}_R : l_R(v) = vl \wedge l_R(v') = vl' \wedge (v, el, v') \in \mathcal{E}_R\ \}$$

$$(3.2)\ <\ =\ (\ \equiv\ \circ\ In^+\ )$$

$$(3.3)\ >\ =\ (\ (Out^+)^{-1}\ \circ\ \equiv\ )$$

$$(3.4)\ <>\ =\ (\ (Out^+)^{-1}\ \circ\ \equiv\ \circ\ In^+\ )\ \blacksquare$$

Die Formeln für die Präzedenzrelationen <, > und <> lassen sich folgendermaßen begründen:

(1) Ein Tripel (vl_1, el, vl_3) charakterisiert eine einlaufende Kante, wenn eine Kante einer rechten Seite existiert, die durch (vl_1, el, N) charakterisiert ist, und einlaufende el-Kanten von N-Knoten auf vl_3-Knoten vererbt werden (s. Abb. 2.11 auf Seite 27). Dieser Sachverhalt wird gerade durch die Verkettung von ≡ und In^+ wiedergegeben.

(2) Analog läßt sich für auslaufende Kanten argumentieren. Dabei ist zunächst die Inverse von Out^+ zu bilden, weil Quellknoten auslaufender Kanten jeweils durch die letzten Komponenten von Out^+-Tripeln charakterisiert werden. Schließlich ist $(Out^+)^{-1}$ noch mit ≡ zu verketten.

(3) Kanten, die sowohl in rechte Seiten ein- als auch aus ihnen auslaufen können, lassen sich folgendermaßen charakterisieren: Man betrachte Kanten auf rechten Seiten, die zwischen Nichtterminalsymbolen verlaufen (s. Abb. 2.12 auf Seite 27). Mittels In^+ bzw. $(Out^+)^{-1}$ gehe man ziel- bzw. quellseitig zu den Markierungen ersetzender Knoten über, auf die sich aus- bzw. einlaufende Kanten vererben. Das verbindende Glied zwischen diesen beiden Relationen stellt die Relation ≡ dar.

Ausgehend von dieser Bedeutung der Präzedenzrelationen läßt sich nun der Begriff des Henkels folgendermaßen definieren:

Def. 2.3.9 Henkel

Gegeben seien eine Präzedenz-Graphgrammatik gr = ($\mathcal{L}_V$, $\mathcal{L}_E$, G_S, $\mathcal{P}$), ein Graph $G \in \mathcal{G}(\mathcal{L}_V, \mathcal{L}_E)$, und ein Untergraph $G_R \subseteq G$. Dann heißt G_R Henkel $\Leftrightarrow$ es existiert eine Produktion $p \in \mathcal{P}$ mit p = (L, R, $\mathcal{EM}$), so daß folgendes gilt:

(1) $G_R \cong R$ (Daraus folgt, daß allen Kanten von G_R die Präzedenz $\equiv$ zugeordnet ist)

(2) Allen einlaufenden Kanten ist die Präzedenz < oder <> zugeordnet:
$\forall\, v \in \mathcal{V}_G \setminus \mathcal{V}_{GR}\ \ \forall\, el \in \mathcal{L}_E\ \ \forall\, v_{GR} \in \mathcal{V}_{GR}$:
$(v, el, v_{GR}) \in \mathcal{E} \Rightarrow (l(v), el, l(v_{GR})) \in (\, < \,\cup\, <> \,)$

(3) Allen auslaufenden Kanten ist die Präzedenz > oder <> zugeordnet:
$\forall\, v_{GR} \in \mathcal{V}_{GR}\ \forall\, el \in \mathcal{L}_E\ \ \forall\, v \in \mathcal{V}_G \setminus \mathcal{V}_{GR}$:
$(v_{GR}, el, v) \in \mathcal{E} \Rightarrow (l(v_{GR}), el, l(v)) \in (\, > \,\cup\, <> \,)$ ■

Bsp. 2.3.10 Präzedenzrelationen für Syntaxdiagramme

Die Präzedenzrelationen für die Graphgrammatik zur Erzeugung von Syntaxdiagrammen (zugehörige Produktionen s. Bsp. 2.3.4) sehen folgendermaßen aus:

(1) $\equiv$ = { (in, next, Sub), (Sub, next, out), (Sub, next, Sub) }

(2) In = { (Start, next, in), (Sub, next, Symbol), (Sub, next, in) }

(3) $In^+ = In$

(4) Out = { (Start, next, out), (Sub, next, Symbol), (Sub, next, out) }

(5) $Out^+ = Out$

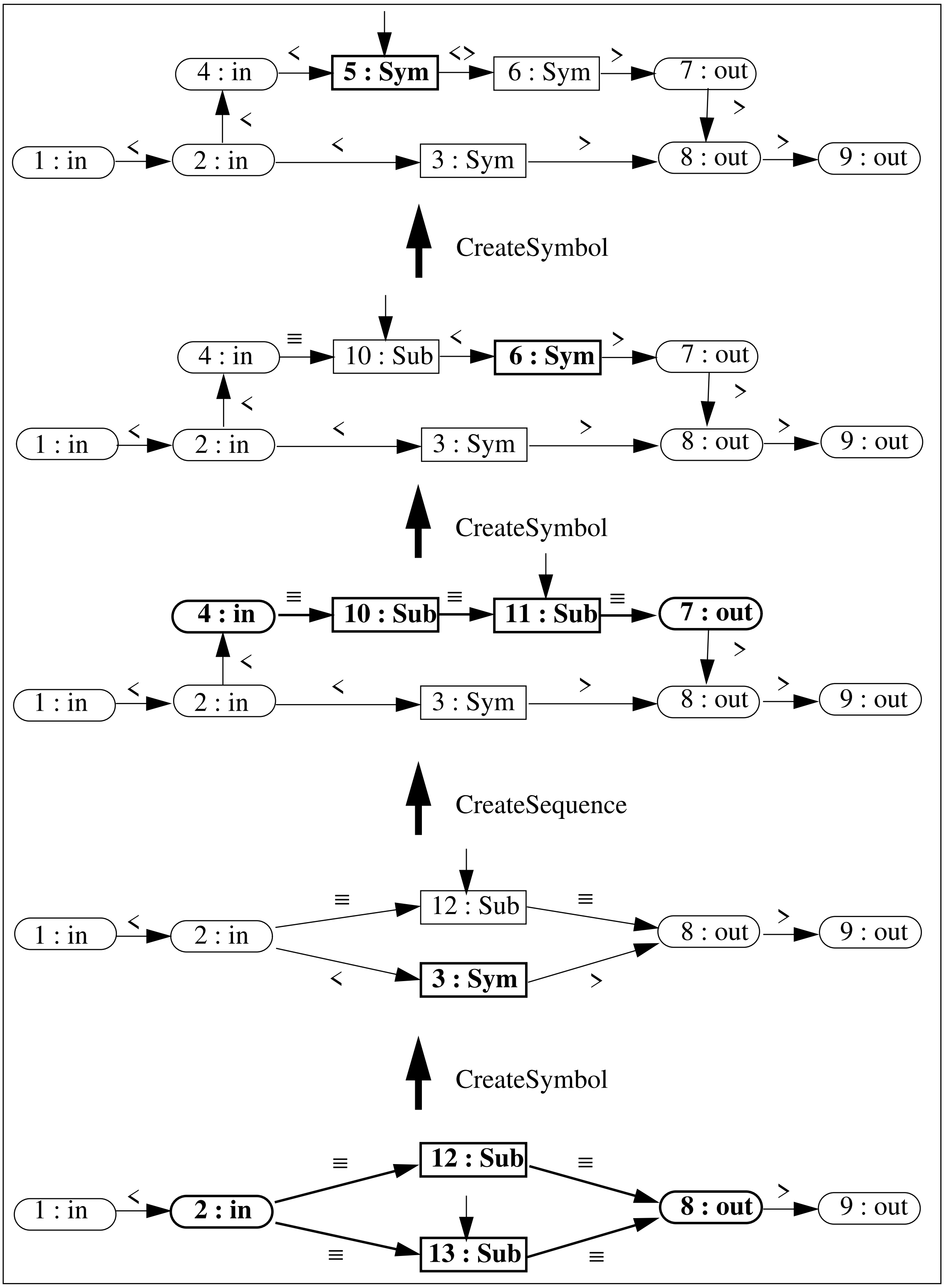
(6) $(Out^+)^{-1}$ = { (out, next, Start), (Symbol, next, Sub), (out, next, Sub) }

(7) $< \;=\; \equiv \circ In^+$
= { (in, next, Symbol), (in, next, in), (Sub, next, Symbol), (Sub, next, in) }

(8) $> \;=\; (Out^+)^{-1} \circ \equiv$
= { (Symbol, next, out), (Symbol, next, Sub), (out, next, out), (out, next, Sub) }

(9) $<> \;=\; (Out^+)^{-1} \circ \equiv \circ In^+ \;=\; > \circ In^+ \;=\; (Out^+)^{-1} \circ <$
= { (Symbol,next,Symbol), (Symbol,next,in), (out,next,Symbol), (out,next,in) }

Die Relationen ≡, <, > und <> sind paarweise disjunkt. Ferner sind die Einbettungsüberführungsregeln monoton, die rechten Seiten sind zusammenhängend, und es gibt keine Produktionen mit zueinander isomorphen rechten Seiten. Also handelt es sich hier um eine Präzedenz-Graphgrammatik. ■


CreateSymbol
CreateSymbol
CreateSequence
CreateSymbol
1 : in
2 : in
3 : Sym
4 : in
5 : Sym
6 : Sym
7 : out
8 : out
9 : out
10 : Sub
11 : Sub
12 : Sub
13 : Sub

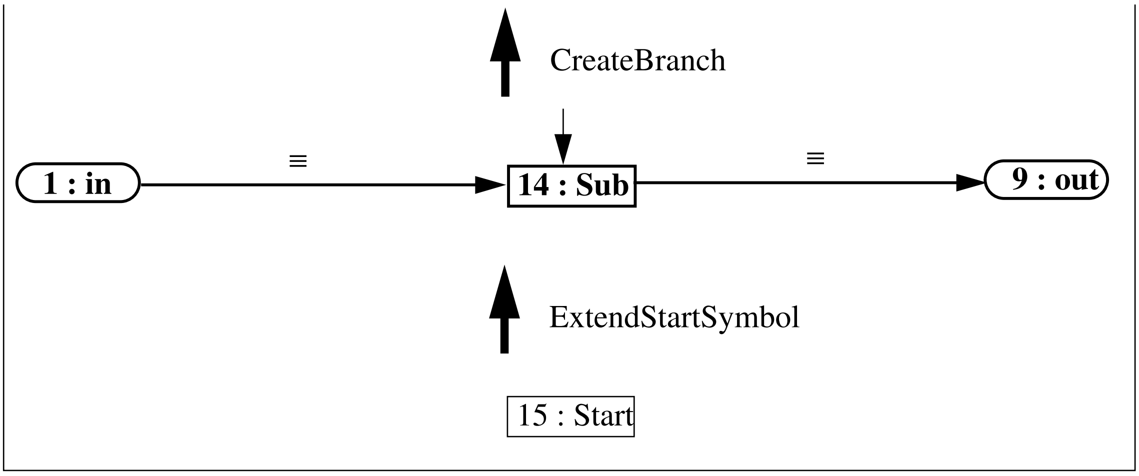


Abb. 2.13: Beispiel für eine Präzedenzanalyse

Zum Abschluß dieses Abschnitts geben wir nun noch ein Beispiel für die Durchführung einer Präzedenzanalyse an.

Bsp. 2.3.11 Anwendung der Präzedenzanalyse

Abb. 2.13 zeigt ein Beispiel für die Reduktion eines Syntaxdiagramms. Die entsprechenden Graphproduktionen sind in Bsp. 2.3.4 auf Seite 25, die Präzedenzrelationen in Bsp. 2.3.10 auf Seite 30 angegeben. Die Knotenmarkierung "Symbol" ist aus Darstellungsgründen mit "Sym" abgekürzt. Der Knoten, an dem die Suche nach einem Henkel gestartet wird, ist jeweils mit einem Pfeil gekennzeichnet. Nach einer Reduktion dient jeweils die eingesetzte linke Seite als Startpunkt für die Suche. Der Henkel selbst ist jeweils durch Fettdruck hervorgehoben. ■

Das in diesem Abschnitt vorgestellte Verfahren zur Präzedenzanalyse erhebt nicht den Anspruch, in irgendeiner Hinsicht "optimal" zu sein. Insbesondere lassen sich die Bedingungen, die in Def. 2.3.8 auf Seite 29 an eine Präzedenz-Graphgrammatik gestellt wurden, weiter abschwächen. Wir gehen hierauf nicht weiter ein, sondern verweisen auf die einschlägige Literatur (s. dazu Abschnitt 2.7 auf Seite 55).

2.4 Attributierte Graphgrammatiken

Im bisherigen Verlauf dieses Kapitels haben wir gerichtete Graphen zugrunde gelegt, deren Knoten und Kanten markiert sind. Weder den Knoten noch den Kanten ließen sich ausser ihrer Markierung weitere Informationen zuordnen. Dies hat sich bereits bei der Modellierung von Syntaxdiagrammen als unbefriedigend erwiesen (s. Bsp. 2.2.7 auf Seite 14): Sprachkonstrukte wie Deklaration, Zuweisung etc. werden in Syntaxdiagrammen durch

Knoten repräsentiert, denen bisher die entsprechende Zeichenkette als Markierung zugeordnet werden mußte (es sei denn, es wurde von diesen Zeichenketten völlig abstrahiert, wie dies in Abschnitt 2.3 geschehen ist). Der Nachteil dieser Modellierung liegt auf der Hand: Die Gemeinsamkeiten der Knoten, die Sprachkonstrukte repräsentieren, lassen sich durch die Knotenmarkierung nicht ausdrücken. I.w. muß für jede Knoteninstanz eine eigene Markierung eingeführt werden. Dies widerspricht der Vorstellung, daß die Markierung den Typ eines Knotens widerspiegelt.

Dieses Problem läßt sich dadurch lösen, daß man **Knotenattribute** einführt. Dies erlaubt es beispielsweise, ein Sprachkonstrukt in einem Syntaxdiagramm durch einen Knoten vom Typ "Symbol" zu repräsentieren, dem als Attribut eine Zeichenkette zugeordnet ist. Diese Vorgehensweise wird auch im Compilerbau verwendet, um die lexikalische von der kontextfreien Syntax zu trennen oder die statische Semantik in Form von Attributwerten festzuhalten. Grob gesprochen dient die Einführung von Attributen in Graphen dazu, eine Trennung von **Struktur**- und **Wertinformation** zu ermöglichen: Strukturen werden durch markierte Knoten und gerichtete, markierte Kanten ausgedrückt, während mit Hilfe von Attributen Werte dargestellt werden.

Auf eine formale Einführung von Knotenattributen (d.h. eine Erweiterung der Definitionen von Graphen, Graphproduktionen etc.) wollen wir an dieser Stelle verzichten. Statt dessen gehen wir anhand eines einfachen Beispiels darauf ein, wie Produktionen zur Manipulation attributierter Graphen in PROGRESS notiert werden.

Bsp. 2.4.1 Beispiel für eine attributierte Graphproduktion

```
production CreateSymbol (Name : String) =

          [ ‘1 : Sub ]   ::=   [ 1’ : Symbol ]

     embedding copy <-next-, -next-> to 1’;
     transfer 1’.SymbolName := Name;
end;
```

Abb. 2.14: Beispiel für eine attributierte Graphproduktion

Abb. 2.14 zeigt als Beispiel eine Produktion, die einen Knoten für ein Sprachkonstrukt erzeugt. Der erzeugte Knoten trägt die Markierung "Symbol" und hat ein Attribut "SymbolName", dem im **Transferteil** der Produktion der Wert "Name" zugewiesen wird. Dieser Wert wird der Produktion als **Parameter** übergeben. Dadurch wird vermieden, daß für jede Zeichenkette eine eigene Produktion benötigt wird. I.a. kann eine Produktion zusätzlich noch

einen **Bedingungsteil** enthalten, in dem Bedingungen an die Knotenattribute der linken Seite formuliert werden. Wir kommen darauf später noch zurück. ■

Bisher haben wir nur **eigenständige Attribute** betrachtet, deren Werte nur durch explizite Zuweisungen in Produktionen manipuliert werden dürfen. Ihnen stehen die **abgeleiteten Attribute** gegenüber, deren Werte implizit durch gerichtete Gleichungen festgelegt werden. Als Beispiel läßt sich etwa die Kapitelnumerierung in einer Dokumentation anführen: Ein Knoten, der ein Kapitel repräsentiert, trägt als Attribut eine Nummer, die sich durch Inkrementieren der Nummer ergibt, die dem Knoten für das Vorgängerkapitel zugeordnet ist. Auf die Behandlung abgeleiteter Attribute werden wir in Kapitel 4 eingehen.

Grundsätzlich ist es auch denkbar, nicht nur Knoten-, sondern auch **Kantenattribute** einzuführen. So könnte man beispielsweise in Graphen, die Verwandtschaftsverhältnisse zwischen Personen darstellen, Kanten einführen, die verheiratete Personen miteinander verbinden und denen als Attribut das Datum der Heirat zugeordnet ist. Die Einführung von Kantenattributen bringt jedoch folgende Probleme mit sich:

(1) Der Ansatz, Kanten durch Tripel zu identifizieren, erweist sich als unhandlich. Man müßte dann im Bedingungs- bzw. Transferteil einer Graphproduktion stets auf diese Tripel Bezug nehmen, um Bedingungen an Kantenattribute zu formulieren bzw. ihnen Werte zuzuweisen. Wesentlich bequemer wäre es, wenn Kanten ebenso wie Knoten eigene Bezeichner erhalten. Wir kommen darauf in Kapitel 3 noch zurück.

(2) Die Einbettungsüberführungsregeln müssen so erweitert werden, daß neu erzeugten Einbettungskanten definierte Attributwerte zugeordnet werden. Bei stark eingeschränkten Varianten von Einbettungsüberführungen ist dies unproblematisch. So können z.B. im Falle monotoner Einbettungsüberführungsregeln (s. Def. 2.3.3 auf Seite 25) die Attribute der alten Kanten bei den neu erzeugten Kanten übernommen werden. Im allgemeinen Fall können jedoch Probleme auftreten. So läßt sich i.a. eine neu erzeugte Kante nicht genau einer alten Kante zuordnen, die für ihre Erzeugung verantwortlich ist (Beispiel : Sowohl mit el_1 als auch mit el_2 markierte einlaufende Kanten werden zu el_3-Kanten ummarkiert). Es ist dann unklar, welche Kantenattribute übernommen werden sollen.

2.5 Kontextabhängige Graphgrammatiken/ Graphersetzungssysteme

In diesem Kapitel haben wir bisher nur kontextfreie Graphgrammatiken kennengelernt. In vielen Fällen lassen sich jedoch Graphen einer bestimmten Klasse nicht mit Hilfe einer kontextfreien Graphgrammatik erzeugen, sondern man benötigt Graphproduktionen

mit mehrknotigen linken Seiten. Man spricht dann von **kontextabhängigen Graphproduktionen** bzw. von einer **kontextabhängigen Graphgrammatik**.

Bei Betrachtung der Definitionen aus Abschnitt 2.2 erkennt man, daß sich kontextabhängige Graphproduktionen problemlos einführen lassen: Es ist lediglich der Begriff der Graphproduktion (Def. 2.2.2) so zu erweitern, daß auf der linken Seite ein beliebiger Graph $L \in \mathcal{G}(\mathcal{L}_V, \mathcal{L}_E)$ zugelassen wird (der mindestens einen Nichtterminalknoten enthalten sollte). Alle anderen Definitionen lassen sich unverändert übernehmen, da nirgends die Tatsache benutzt wird, daß auf der linken Seite ein einknotiger kantenloser Graph steht.

Im folgenden geben wir ein Beispiel für eine kontextabhängige Graphgrammatik an.

Bsp. 2.5.1 Kontextabhängige Graphgrammatik für Flußdiagramme

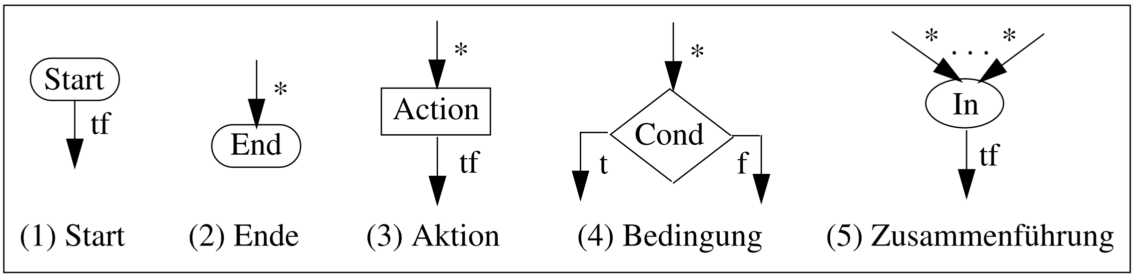


Abb. 2.15: Elemente von Flußdiagrammen

Ein **Flußdiagramm** setzt sich aus folgenden Elementen zusammen (Abb. 2.15):

(1) Es gibt genau einen **Startknoten**, aus dem genau eine mit "tf" markierte Kante ausläuft, die einen unbedingten Kontrollfluß repräsentiert.

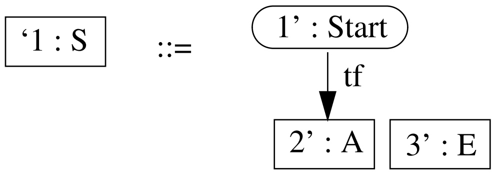
(2) Es gibt genau einen **Endknoten**, in den genau eine beliebig markierte Kante hineinläuft.

(3) Es gibt beliebig viele **Aktionsknoten**, in die jeweils genau eine beliebig markierte Kante hinein- bzw. aus denen jeweils genau eine "tf"-Kante herausläuft.

(4) Es gibt beliebig viele **Bedingungsknoten**, in die jeweils genau eine beliebig markierte Kante hineinläuft. Aus jedem Bedingungsknoten laufen genau zwei Kanten heraus, die mit "t" bzw. "f" markiert sind und bedingte Kontrollflüsse repräsentieren.

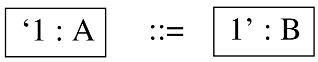
(5) Es gibt beliebig viele **Zusammenführungsknoten**, in die jeweils mindestens zwei beliebig markierte Kanten hineinlaufen bzw. aus denen jeweils genau eine "tf"-Kante herausläuft, die zu einem Aktions-, Bedingungs- oder Endknoten führt.

production ExtendStartSymbol =



end;

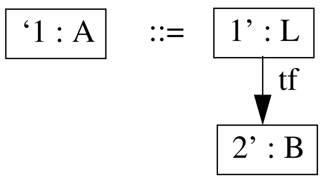
production CreateUnlabeledNode =



embedding copy <-*- from ‘1 to 1’;

end;

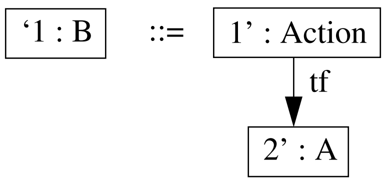
production CreateLabeledNode =



embedding copy <-*- from ‘1 to 1’;

end;

production CreateActionNode (Text : String) =

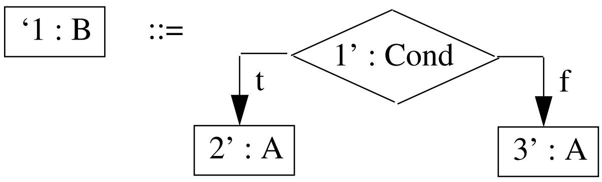


embedding copy <-*- from ‘1 to 1’;

transfer 1’.Text := Text;

end;

production CreateCondNode (Text : String) =



embedding copy <-*- from ‘1 to 1’;

transfer 1’.Text := Text;

end;

production CreateEndNode =

‘1 : B

::= 1’ : End

‘2 : E

embedding copy <-*- from ‘1 to 1’;

end;

production CreateSecondIncomingEdge =

‘1 : A ‘2 : L ::= 1’ : In

embedding copy <-*- from ‘1, ‘2 to 1’, -tf-> from ‘2 to 1’;

end;

production CreateNextIncomingEdge =

‘1 : A ‘2 : In ::= 1’ : In

embedding copy <-*- from ‘1, ‘2 to 1’, -tf-> from ‘2 to 1’;

end;

Abb. 2.16: Graphproduktionen zur Erzeugung von Flußdiagrammen

Abb. 2.16 zeigt Graphproduktionen, mit deren Hilfe sich Flußdiagramme erzeugen lassen. Einige dieser Graphproduktionen sind kontextabhängig. Der Grund dafür besteht darin, daß Flußdiagramme i.a. nicht wohlstrukturiert sind, sondern der Kontrollfluß beliebig strukturiert sein kann ("Spaghetti-Programme").

Betrachten wir nun die Produktionen im einzelnen:

(1) Die Produktion "ExtendStartSymbol" erzeugt den terminalen Startknoten 1’ und einen nichtterminalen Knoten 3’, der später zur Erzeugung des Endknotens benutzt wird. Ferner wird ein Knoten 2’ erzeugt, der der weiteren Strukturerzeugung dient und mit "A" markiert ist. Für "A"-Knoten gilt die folgende Invariante: In einen "A"-Knoten läuft genau eine Kante hinein, und es läuft keine Kante heraus.

(2) Die Produktion "CreateUnlabeledNode" bereitet die Erzeugung eines Aktions-, Bedingungs- oder Endknotens vor, vor dem keine Zusammenführung benötigt wird. Der hier benutzte Begriff "Label" wurde gewählt, um eine Assoziation zu Sprungmarken herzustellen. Man beachte, daß für "B"-Knoten dieselbe Invariante gilt wie für "A"-Knoten.

(3) Analog bereitet die Produktion "CreateLabeledNode" die Erzeugung eines Bedingungs-, Aktions- oder Endknotens vor, vor dem eine Zusammenführung benötigt wird. Für die Zusammenführung wird zunächst ein Nichtterminalknoten erzeugt, in den genau eine Kante hineinläuft.

(4) Mittels "CreateActionNode" und "CreateCondNode" werden Aktions- bzw. Bedingungsknoten erzeugt. Die "A"-Knoten werden jeweils für die weitere Strukturerzeugung benötigt.

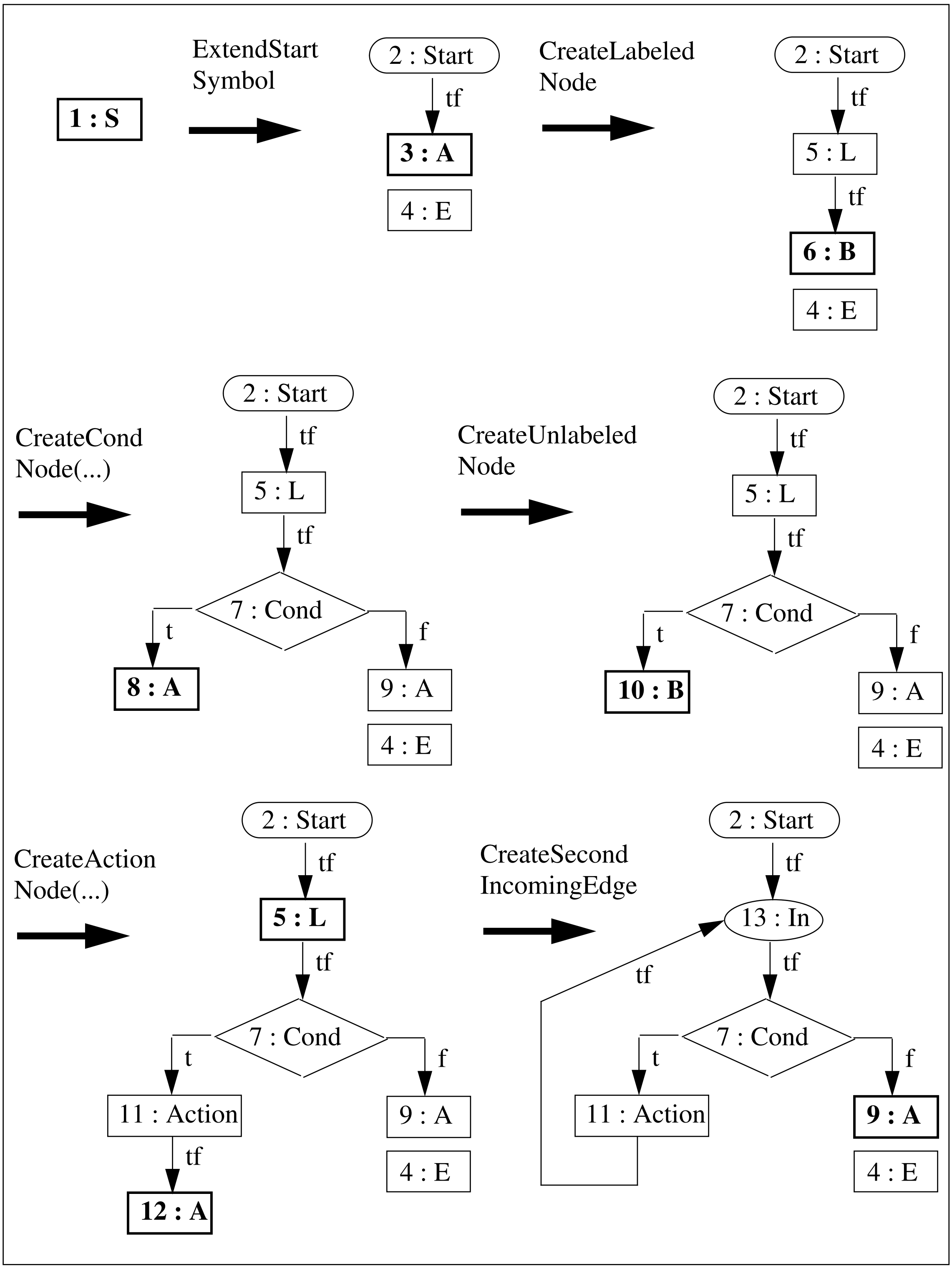
(5) Mittels "CreateEndNode" wird aus einem "B"-Knoten und dem einzigen "E"-Knoten der Endknoten erzeugt.

(6) "CreateSecondIncomingEdge" erzeugt die zweite in eine Zusammenführung hineinlaufende Kante, indem ein "A"- und ein "L"-Knoten miteinander verschmolzen werden. Der Verschmelzungsknoten ist terminal, weil er nun bereits die mindestens erforderliche Anzahl einlaufender Kanten hat.

(7) Mittels "CreateNextIncomingEdge" lassen sich weitere Kanten erzeugen, die in eine Zusammenführung hineinlaufen.

Um das Zusammenspiel der Produktionen zu illustrieren, zeigt Abb. 2.17 die Erzeugung eines einfachen Flußdiagramms. Man beachte, daß die zu ersetzenden linken Seiten jeweils durch Fettdruck gekennzeichnet sind. ■

Während bei Graphgrammatiken der generative Aspekt - d.h. die Angabe eines Systems zur Erzeugung/Erkennung von Graphen einer bestimmten Klasse - im Vordergrund steht, beschreiben **Graphersetzungssysteme** Transformationsprozesse, bei denen die Unterscheidung zwischen nichtterminalen und terminalen Strukturen unwichtig ist. Formal unterscheiden sich Graphersetzungssysteme von Graphgrammatiken lediglich dadurch, daß nicht zwischen nichtterminalen und terminalen Markierungen unterschieden wird. Folglich gibt es auch keinen Unterschied zwischen der Menge der Graphsatzformen und der durch ein Graphersetzungssystem beschriebenen Sprache (alle Graphen, die aus dem Startgraphen durch Anwendung von Graphersetzungsregeln erzeugt werden können). Anstelle des Begriffs "Graphproduktion" verwendet man den Begriff "**Graphersetzungsregel**". Da Graphersetzungen nicht nur dazu dienen, neue Strukturen zu erzeugen, sondern auch bestehende Strukturen zu verändern oder zu löschen, sind sie häufig kontextabhängig, d.h. sie haben häufig mehrknotige linke Seiten.


1 : S
ExtendStart
Symbol
2 : Start
tf
3 : A
4 : E
CreateLabeled
Node
2 : Start
tf
5 : L
tf
6 : B
4 : E
CreateCond
Node(...)
2 : Start
tf
5 : L
tf
7 : Cond
t
f
8 : A
9 : A
4 : E
CreateUnlabeled
Node
2 : Start
tf
5 : L
tf
7 : Cond
t
f
10 : B
9 : A
4 : E
CreateAction
Node(...)
2 : Start
tf
5 : L
tf
7 : Cond
t
f
11 : Action
tf
12 : A
9 : A
4 : E
CreateSecond
IncomingEdge
2 : Start
tf
13 : In
tf
tf
7 : Cond
t
f
11 : Action
9 : A
4 : E

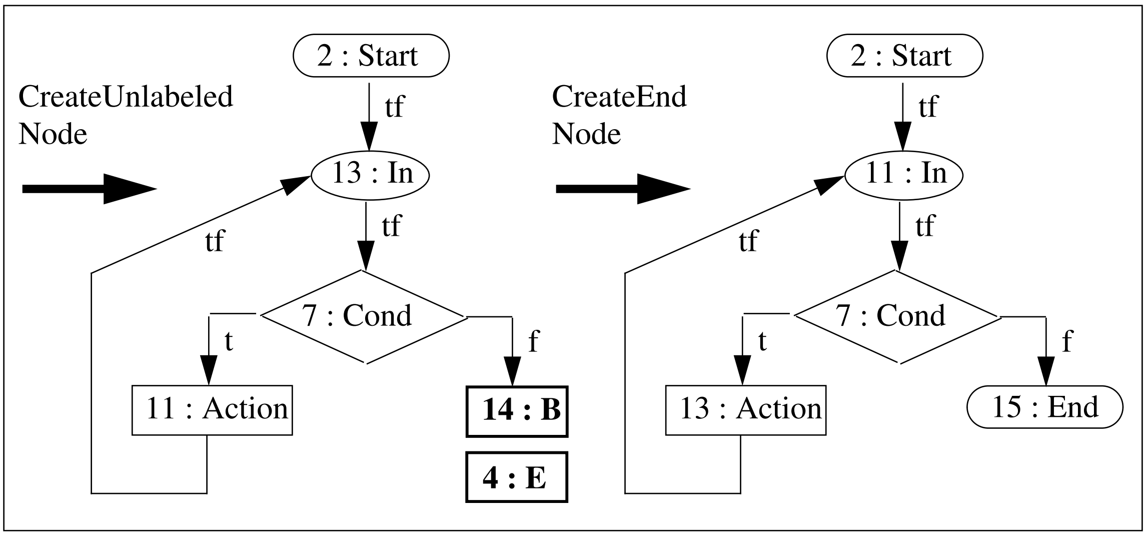


Abb. 2.17: Beispiel für die Erzeugung eines Flußdiagramms

Ein Beispiel für den Einsatz von Graphersetzungssystemen wurde bereits in der Einleitung angegeben. Dort wurde ein Kommando eines strukturbezogenen Editors (Umwandeln einer "REPEAT"- in eine "WHILE"-Anweisung) mit Hilfe einer Graphersetzungsregel spezifiziert (Abb. 1.2 auf Seite 2; man beachte, daß die dort angegebene Graphersetzung insofern unvollständig ist, als weder der Ort der Ersetzung - d.h. das aktuelle Inkrement, auf das sich das Kommando bezieht - noch die Einbettungsüberführungsregel angegeben wurde). Ein weiteres Beispiel für die Anwendung von Graphersetzungsregeln wird im folgenden angegeben.

Bsp. 2.5.2 Zwischencode-Optimierung

Im Compilerbau werden **abstrakte Syntaxgraphen** als Ausgangspunkt für die Codeerzeugung benutzt. Es handelt sich dabei um einen hohen Zwischencode, der wertvolle strukturelle Informationen enthält, die bei der Optimierung von Nutzen sind. Als Beispiel läßt sich das Herausziehen gemeinsamer Teilausdrücke anführen.

Im folgenden betrachten wir arithmetische Ausdrücke, in denen als Operanden Variablenbezeichner und als Operatoren "+", "$*$" und Klammern vorkommen. Abb. 2.18 zeigt als Beispiel die externe und die interne Repräsentation eines Ausdrucks als Text bzw. als abstrakter Syntaxbaum.

Der abstrakte Syntaxbaum aus Abb. 2.18 enthält Redundanzen, d.h. zueinander isomorphe Unterbäume. Diese Redundanzen lassen sich beseitigen, indem sukzessive Knoten

verschmolzen werden. Es entsteht dann ein redundanzfreier abstrakter Syntaxgraph, von dem ausgehend sich optimierter Maschinencode erzeugen läßt.

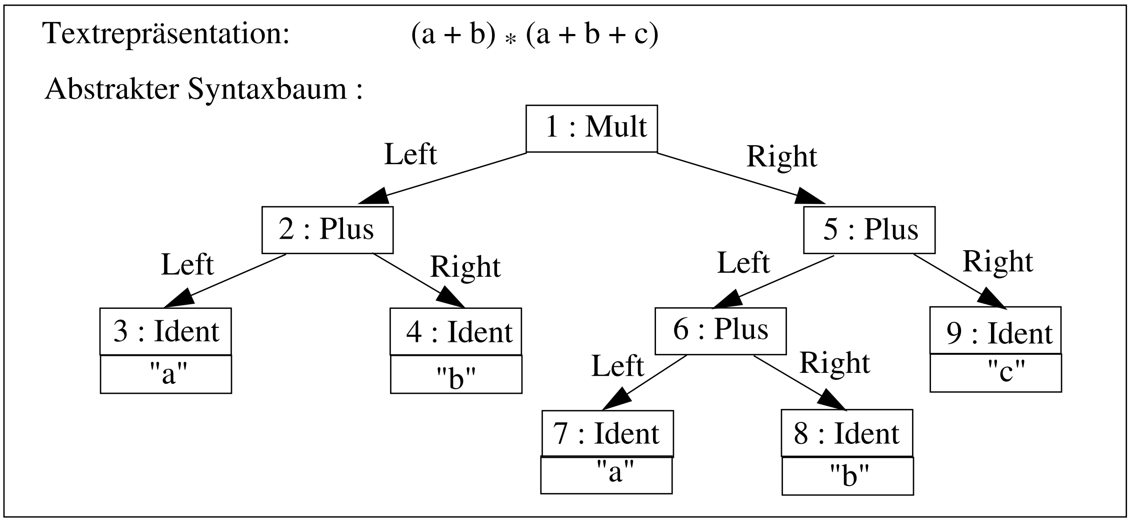


Abb. 2.18: Text- und Baumdarstellung eines Ausdrucks

Abb. 2.19 zeigt zwei Graphersetzungsregeln, mit deren Hilfe sich Redundanzen in abstrakten Syntaxgraphen beseitigen lassen. Es wird vorausgesetzt, daß es sich bei den abstrakten Syntaxgraphen, auf den diese Graphersetzungsregeln angewendet werden, um Darstellungen von jeweils genau einem Ausdruck handelt. Mittels "MergeIdentNodes" werden Bezeichnerknoten verschmolzen, denen dasselbe Namensattribut zugeordnet ist (s. condition-Teil der Graphersetzungsregel). Analog werden mit Hilfe von "MergePlusNodes" zwei Additionsknoten verschmolzen, deren Operanden identisch sind. Man beachte, daß diese Graphersetzungsregel nur anwendbar ist, wenn die Operanden Bezeichner sind. Eine Verallgemeinerung auf beliebige Operanden liegt auf der Hand, läßt sich aber mit den bisher eingeführten Hilfsmitteln nicht formulieren (man beachte, daß man nicht einfach die Markierung "Ident" durch "*" - beliebige Markierung - ersetzen kann, da sich dann die Markierungen der Knoten der rechten Seite nicht festlegen lassen). Wir kommen darauf später noch zurück.

Schließlich zeigt Abb. 2.20, wie sich der abstrakte Syntaxbaum mit Hilfe der Graphersetzungen aus Abb. 2.19 optimieren läßt. Die zu ersetzenden Untergraphen sind jeweils durch Fettdruck gekennzeichnet. ■

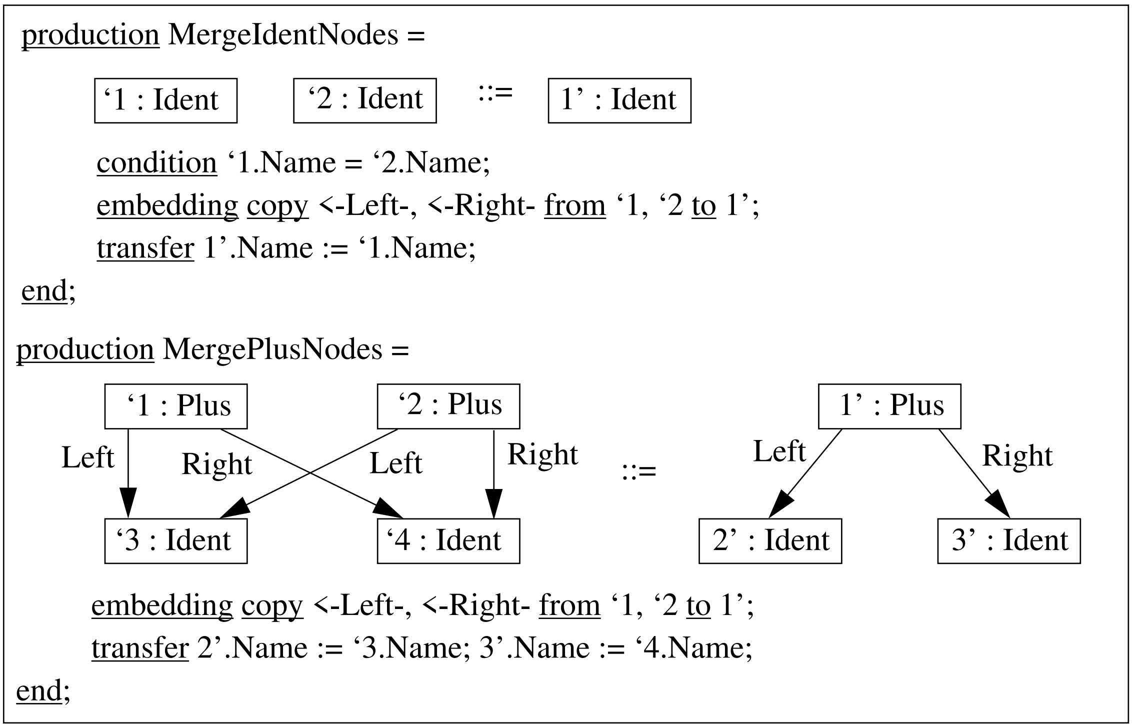
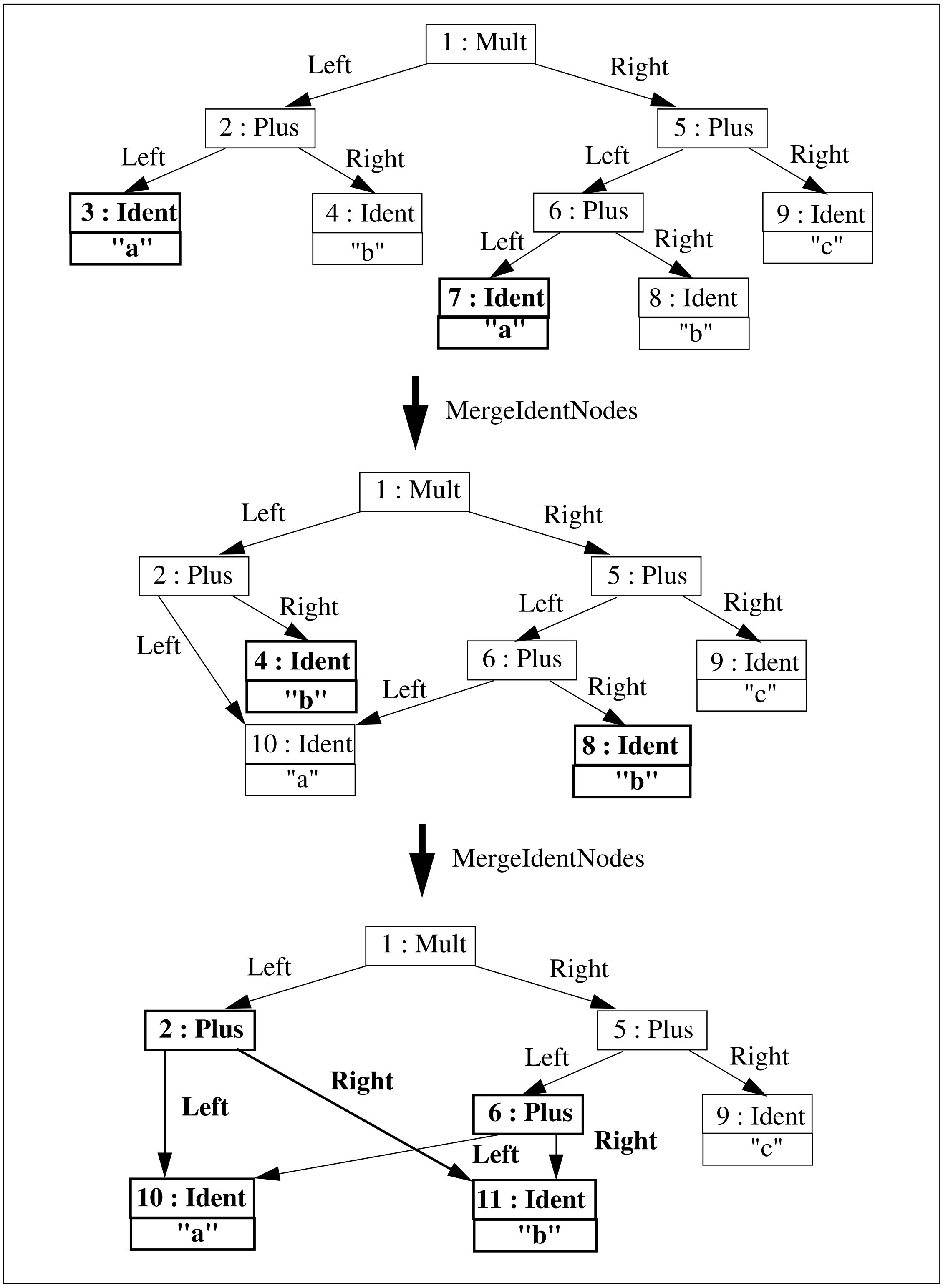


Abb. 2.19: Graphersetzungsregeln zur Optimierung von Ausdrücken

Zum Abschluß dieses Abschnitts setzen wir uns nun noch mit der Frage auseinander, ob der Übergang von kontextfreien zu kontextabhängigen Graphregeln hinsichtlich der **Klasse der Graphsprachen** einen Gewinn bringt. Bei Stringgrammatiken ist dies bekanntermaßen der Fall. Bei Graphgrammatiken ist die Situation komplizierter, da außer der Gestalt der linken und rechten Seiten auch die Art der Einbettungsüberführungsregeln die Klasse der Graphsprachen beeinflußt. Läßt man sehr komplizierte Einbettungsüberführungsregeln zu, so läßt sich zeigen /Na 79/, daß durch den Übergang von kontextfreien zu kontextabhängigen Graphregeln die Klasse der Graphsprachen nicht größer wird: Die Einbettungsüberführungsregeln erlauben es, auf den Kontext eines Knotens zuzugreifen. Daher läßt sich eine kontextabhängige Graphregel durch eine Menge von kontextfreien Graphregeln simulieren. Diese Simulation funktioniert aber nur, wenn die Einbettungsüberführungsregeln hinreichend mächtig sind. Näheres dazu findet der Leser in /Na 79/.


1 : Mult
Left
Right
2 : Plus
5 : Plus
3 : Ident
"a"
4 : Ident
"b"
6 : Plus
9 : Ident
"c"
7 : Ident
"a"
8 : Ident
"b"
MergeIdentNodes
10 : Ident
"a"
MergeIdentNodes
11 : Ident
"b"

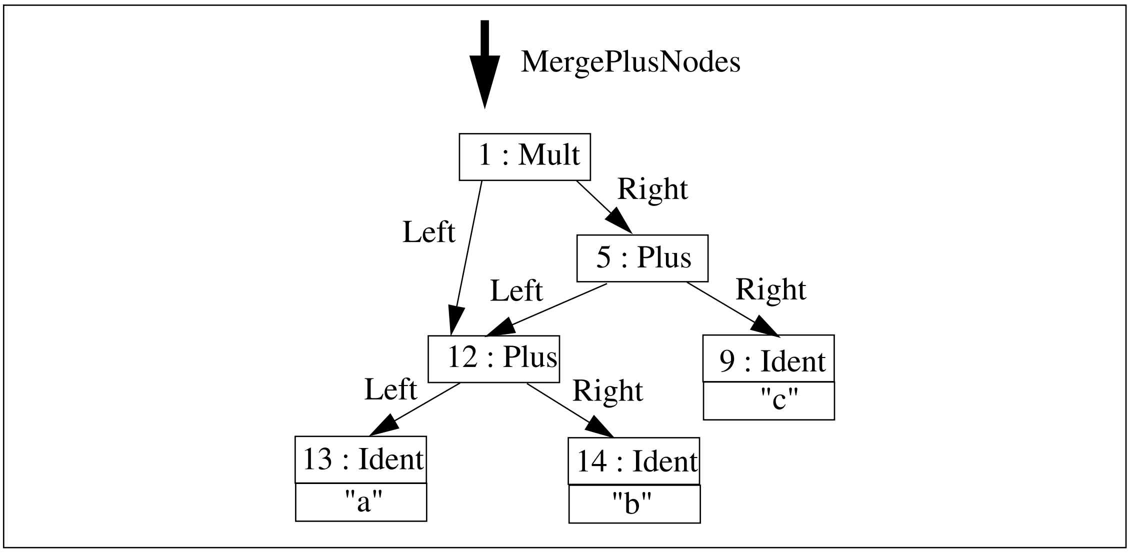


Abb. 2.20: Optimierung eines Ausdrucks

2.6 Programmierte Graphregelsysteme

Im bisherigen Verlauf dieses Kapitels haben wir uns mit Graphregelsystemen befaßt, in denen die Reihenfolge der Anwendung von Graphregeln nicht festgelegt ist. In vielen Fällen erweist es sich jedoch als notwendig oder zumindest nützlich, **Kontrollstrukturen** für Graphregeln einzuführen. Mit Hilfe von Kontrollstrukturen läßt sich beispielsweise ausdrükken, daß zwei Graphregeln nacheinander angewendet werden sollen, daß wahlweise eine von beiden angewendet werden soll, etc. Auf diese Weise entsteht ein **programmiertes Graphregelsystem**.

Natürlich kann man die linken und rechten Seiten von Graphregeln immer so erweitern, daß diese in der gewünschten Reihenfolge nacheinander ausgeführt werden. Diese Vorgehensweise ist jedoch aus softwaretechnischer Sicht mehr oder weniger inakzeptabel. Dies gilt insbesondere dann, wenn man einige wenige Basisoperationen in immer neuen Abfolgen zu verschiedenen komplexen Graphtransformationen kombinieren will. In diesem Fall dürfte es zumindest für den Leser einer Spezifikation ohne explizite Kontrollstrukturen unmöglich sein, aus der Betrachtung einer ungeordneten Menge von Graphregeln das Wissen über ihre Ausführungsreihenfolge wiederzugewinnen.

Kontrollstrukturen sind aus imperativen Programmiersprachen wohlbekannt. Es zeigt sich jedoch, daß derartige Kontrollstrukturen für Graphregelsysteme ungeeignet sind. Daher wurden in PROGRESS andersartige Kontrollstrukturen eingeführt, die auf die Eigenschaften der Elementaroperationen abgestimmt sind. Als Elementaroperationen werden dabei neben

den schon eingeführten Graphregeln auch **Graphtests** benutzt, mit deren Hilfe sich das Vorkommen eines Graphen im Wirtsgraphen überprüfen läßt. Graphtests lassen sich als Regeln auffassen, die eine identische Ersetzung durchführen.

Graphtests und Graphersetzungsregeln haben folgende Eigenschaften, die für den Entwurf von Kontrollstrukturen von großer Bedeutung sind:

(1) **Boolescher Charakter**: Eine Operation ist entweder erfolgreich, oder sie schlägt fehl.

(2) **Atomarer Charakter**: Eine Graphersetzungsregel wird entweder vollständig ausgeführt, oder sie schlägt fehl und läßt den Wirtsgraphen unverändert.

(3) **Nichtdeterministischer Charakter**: Das Ergebnis der Anwendung einer Graphersetzungsregel ist i.a. nicht eindeutig festgelegt. Kommt die linke Seite mehrfach im Wirtsgraphen vor, so wird irgendeines dieser Vorkommen in zufälliger Weise zur Ersetzung ausgewählt.

Von den Kontrollstrukturen wird verlangt, daß sie diese Eigenschaften der Elementaroperationen erhalten. Für alle **zusammengesetzten Operationen** soll also ebenfalls gelten:

(1) Anhand des Erfolgs oder Mißerfolgs einer zusammengesetzten Operation kann im Kontrollfluß verzweigt werden.

(2) Eine zusammengesetzte Operation hat - als ganzes betrachtet - atomaren Charakter.

(3) Eine zusammengesetzte Operation kann mehrere verschiedene Ergebnisse haben.

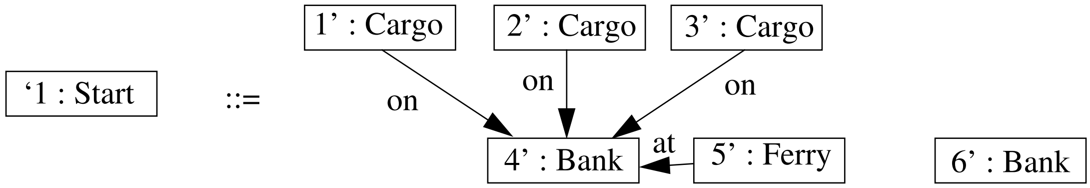
Darüber hinaus muß bei der Definition der Semantik von Kontrollstrukturen insbesondere der **Nichtdeterminismus** der zusammenzusetzenden Operationen berücksichtigt werden: Eine zusammengesetzte Operation ist nur dann erfolglos, wenn sie für alle (!) möglichen Ergebnisse der Operationen erfolglos ist, aus denen sie sich zusammensetzt. Dies bedeutet beispielsweise, daß eine Sequenz zweier Graphersetzungsregeln nicht allein durch eine ungeschickte Auswahl der Anwendungsstelle der ersten Regel zum Scheitern gebracht werden kann.

Bevor wir uns nun mit der Definition von Kontrollstrukturen auseinandersetzen, geben wir ein Beispiel an, das deren Anwendung illustriert.

Bsp. 2.6.1 Das Fährmann-Problem

Das folgende Problem wird häufig in der KI-Literatur behandelt: Ein Wolf, eine Ziege und ein Kohlkopf sollen über einen Fluß transportiert werden. Die dafür verfügbare Fähre ist jedoch so klein, daß pro Fahrt nur ein Frachtgut transportiert werden kann. Aus naheliegenden Gründen darf der Fährmann zudem nicht Wolf und Ziege oder Ziege und Kohlkopf gleichzeitig unbewacht an einem Ufer zurücklassen.

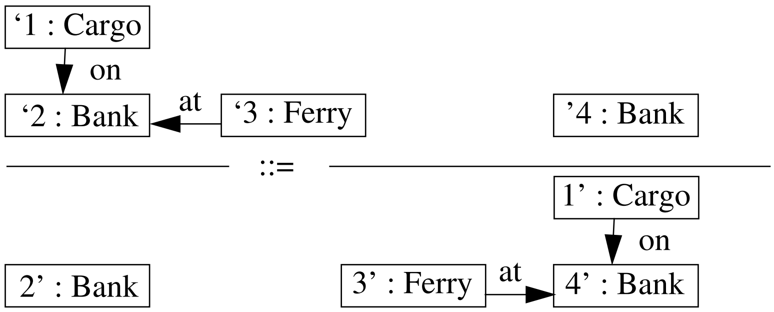
production CreateProblem =



transfer 1’.Id := "Cabbage"; 2’.Id := "Wolf"; 3’.Id := "Goat";
4’.Id := "ThisBank"; 6’.Id := "OtherBank";

end;

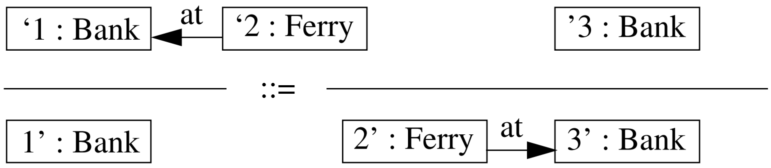
production TransportCargo =



embedding copy <-on- from ‘2 to 2’, <-on- from ‘4 to 4’;
transfer 1’.Id := ‘1.Id; 2’.Id := ‘2.Id; 4’.Id := ‘4.Id;

end;

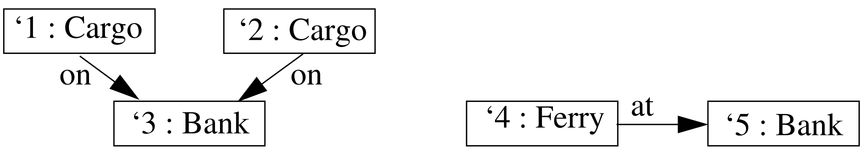
production FerryOver =



embedding copy <-on- from ‘1 to 1’, <-on- from ‘3 to 3’;
transfer 1’.Id := ‘1.Id; 3’.Id := ‘3.Id;

end;

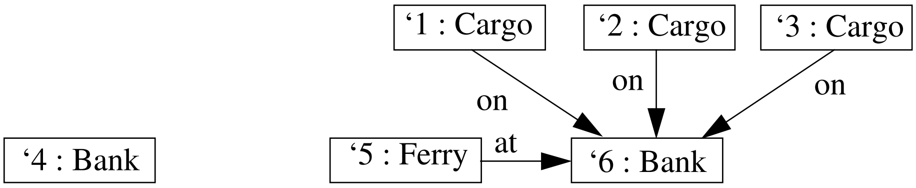
test TwoAloneOnSameBank (FirstId, SecondId : String) =



condition ‘1.Id = FirstId; ‘2.Id = SecondId;

end;

test Success =



condition '6.Id = "OtherBank";
end;

Abb. 2.21: Elementaroperationen für das Fährmann-Problem

Dieses Problem läßt sich mit Hilfe eines programmierten Graphersetzungssystems lösen, dessen Elementaroperationen in Abb. 2.21 dargestellt sind:

(1) Die Ausgangssituation läßt sich durch die rechte Seite der Graphersetzungsregel "CreateProblem" modellieren. Frachtgüter, Flußufer und Fähre werden durch Knoten mit der Markierung "Cargo", "Bank" bzw. "Ferry" dargestellt. In den ersten beiden Fällen dienen Stringattribute zur Unterscheidung der verschiedenen Frachtgüter bzw. Flußufer. "on"- bzw. "at"-Kanten dienen dazu, die Positionen der Frachtgüter bzw. der Fähre darzustellen.

(2) Mit Hilfe von "TransportCargo" wird irgendein Frachtgut über den Fluß transportiert.

(3) Mit "FerryOver" gelangt die Fähre ohne Frachtgut an das andere Ufer.

(4) Mit Hilfe des Graphtests "TwoAloneOnSameBank" läßt sich überprüfen, ob sich mindestens zwei bestimmte Frachtgüter unbewacht an einem Ufer befinden (der Aufpasser befindet sich immer dort, wo die Fähre ist). Durch geeignete Parametrisierung lassen sich damit verbotene Situationen darstellen.

(5) Schließlich beschreibt der Graphtest "Success" den gewünschten Zielzustand.

Abb. 2.22 zeigt, wie sich die Elementaroperationen aus Abb. 2.21 anordnen lassen, um das Fährmann-Problem zu lösen. Dabei werden u.a. folgende Sprachmittel verwendet, die später noch genauer erläutert werden:

(1) Eine **Transaktion** klammert eine Folge von Operationen. Sie ist atomar und liefert implizit einen Booleschen Wert zurück, der ihre erfolgreiche oder gescheiterte Durchführung signalisiert.

(2) Eine **Sequenz** ist eine Folge von Operationen, die nacheinander ausgeführt werden. Als Sequentialisierungszeichen wird "&" verwendet.

(3) Eine **Schleife** dient dazu, eine Folge von Operationen zu iterieren. Sie wird durch das Schlüsselwort "loop" gekennzeichnet.

(4) Eine **Alternative** beschreibt eine nichtdeterministische Auswahl und wird durch das Schlüsselwort "or" gekennzeichnet.

```
transaction SolveFerryMansProblem (out NoOfActions : INTEGER)
   (* Lösung des Fährmann-Problems mit einer minimalen Zahl von Aktionen *)
=
   use CurrentNo : INTEGER := 5
   in
         CreateProblem
      &  loop
            when not def (SolutionWith_N_Actions(CurrentNo))
            then CurrentNo := inc(CurrentNo)
         end
      &  NoOfActions := CurrentNo
   end
end;

transaction SolutionWith_N_Actions(NoOfActions : INTEGER)
   (* Suche nach einer Lösung mit höchstens "NoOfActions" Aktionen *)
=
   use I : INTEGER := 0
   in
      loop
         when not Success
         then
               I := inc(I)
            &  (TransportCargo or FerryOver)
            &  not   (TwoAloneOnSameBank("Wolf", "Goat")
                     or TwoAloneOnSameBank("Goat", "Cabbage")
                     or valid I > NoOfActions)
      end
   end
end;
```

Abb. 2.22: Transaktionen zur Lösung des Fährmann-Problems

Die Transaktion "SolveFerryMansProblem" sucht nach einer Lösung des Fährmann-Problems mit einer minimalen Anzahl von Aktionen. Nach der Initialisierung der lokalen Variablen "CurrentNo" wird zunächst mittels "CreateProblem" die Startsituation erzeugt.

Anschließend wird in einer Schleife nach Lösungen mit höchstens "CurrentNo" Schritten gesucht, wobei "CurrentNo" jeweils inkrementiert wird. Die Schleife bricht ab, wenn eine Lösung gefunden ist (d.h. sobald die hinter "when" notierte Bedingung nicht mehr erfüllt ist).

Die Transaktion "SolutionWith_N_Actions" sucht nach einer Lösung mit höchstens "NoOfActions" Aktionen. Die Lösung wird durch Probieren ermittelt. Die Schleife im Rumpf der Transaktion terminiert erfolgreich, sobald der Graphtest "Success" erfolgreich ist. Ist letzteres nicht der Fall, so wird die hinter "then" stehende Operationenfolge ausgeführt: "I" wird inkrementiert sowie irgendeine Aktion nichtdeterministisch ausgewählt und ausgeführt. Man beachte, daß die Aktion "TransportCargo" selbst ebenfalls nichtdeterministisch ist, da nicht festgelegt ist, welche Fracht an Bord zu nehmen ist. Danach wird überprüft, ob eine verbotenene Situation entstanden ist oder die maximal zulässige Anzahl von Aktionen überschritten wurde. Trifft eine dieser Bedingungen zu, so hat die bisherige Sequenz von Aktionen zu einem Mißerfolg geführt. Dann setzt das **Backtracking** ein, d.h. es werden Aktionen zurückgenommen und durch andere ersetzt. Die Schleife schlägt erst dann endgültig fehl, wenn alle möglichen Sequenzen, die höchstens "NoOfActions" Aktionen umfassen, zum Mißerfolg führen. ■

Um die Semantik der Kontrollstrukturen zu beschreiben, werden **Kontrolldiagramme** verwendet. Ein Kontrolldiagramm für eine Transaktion wird konstruiert, indem man diese in ihre Bestandteile zerlegt und für letztere entsprechende Unterdiagramme erzeugt. Kontrolldiagramme ähneln Flußdiagrammen, mit deren Hilfe häufig in konventionellen Programmiersprachen die Semantik von Kontrollstrukturen beschrieben wird. Im Gegensatz zu den Flußdiagrammen erlauben Kontrolldiagramme aber einen nichtdeterministischen Kontrollfluß.

Def. 2.6.2 Kontrolldiagramm

Sei $\mathcal{O}$ eine Menge von Bezeichnern für Elementaroperationen, d.h. Graphtests bzw. Graphersetzungsregeln.
Ein Kontrolldiagramm ist ein Graph

$$D = (\mathcal{V}, \mathcal{E}, l) \in \mathcal{G}(\mathcal{L}_V, \mathcal{L}_E)$$

mit folgenden Eigenschaften:

(1) D läßt sich disjunkt in Unterdiagramme zerlegen:
$D = D_1 \oplus \ldots \oplus D_n$ für ein geeignetes $n > 0$

(2) $\mathcal{L}_V = \{ \underline{\text{start}}, \underline{\text{stop}}, \underline{\text{call}}\ O_1, \ldots, \underline{\text{call}}\ O_k, \underline{\text{call}}\ v_1, \ldots, \underline{\text{call}}\ v_l \}$ mit:
(2.1) start : Markierung des Startknotens eines Unterdiagramms
(2.2) stop : Markierung des Stopknotens eines Unterdiagramms

(2.3) $\underline{\text{call}}\ O_i$ ($O_i \in \mathcal{O}$ mit $1 \le i \le k$) : Aufruf eines Graphtests bzw. einer Graphersetzungsregel

(2.4) $\underline{\text{call}}\ v_j$ (mit $1 \le j \le l$) : Aufruf eines Unterdiagramms. Für v_j muß gelten: $v_j \in \mathcal{V} \wedge l(v_j) = \underline{\text{start}}$

(3) $\mathcal{L}_E = \{\ \underline{\text{abort}}, \underline{\text{commit}}\ \}$ mit:

(3.1) abort : Kontrollfluß im Mißerfolgsfall

(3.2) commit : Kontrollfluß im Erfolgsfall

(4) Für jedes Unterdiagramm $D_i = (\mathcal{V}_i, \mathcal{E}_i, l_i)$ gilt:

(4.1) D_i ist zusammenhängend.

(4.2) Es gibt genau einen Startknoten:
$\exists!\ v_i \in \mathcal{V}_i :\ l_i(v_i) = \underline{\text{start}}$,
und alle Knoten in D_i sind vom Startknoten aus erreichbar

(4.3) Es gibt genau einen Stopknoten:
$\exists!\ v_i \in \mathcal{V}_i :\ l_i(v_i) = \underline{\text{stop}}$,
und von allen Knoten in D_i aus ist der Stopknoten erreichbar

(4.4) Der Startknoten hat keine einlaufenden Kanten:
$\forall\ s_i, t_i \in \mathcal{V}_i\ \forall\ el \in \mathcal{L}_E : (s_i, el, t_i) \in \mathcal{E}_i \Rightarrow l_i(t_i) \neq \underline{\text{start}}$

(4.5) Der Stopknoten hat keine auslaufenden Kanten:
$\forall\ s_i, t_i \in \mathcal{V}_i\ \forall\ el \in \mathcal{L}_E : (s_i, el, t_i) \in \mathcal{E}_i \Rightarrow l_i(s_i) \neq \underline{\text{stop}}$

(4.6) Aus dem Startknoten laufen nur "commit"-Kanten heraus:
$\forall\ s_i, t_i \in \mathcal{V}_i\ \forall\ el \in \mathcal{L}_E : (s_i, el, t_i) \in \mathcal{E}_i \wedge l_i(s_i) = \underline{\text{start}} \Rightarrow el = \underline{\text{commit}}$ ■

Ein Kontrollunterdiagramm besteht also aus jeweils genau einem Start- und Stopknoten sowie beliebig vielen Aufrufknoten. Die Knoten sind durch "commit"- und "abort"-Kanten miteinander verbunden. Dabei dürfen mehrere Kanten derselben Markierung aus dem gleichen Knoten herauslaufen (nichtdeterministischer Kontrollfluß).

Ausgehend von dem in Abschnitt 2.2 definierten Begriff der Ableitbarkeit (Def. 2.2.10 auf Seite 15) wird nun die Ableitbarkeit mittels eines Kontrollunterdiagramms definiert.

Def. 2.6.3 Ableitbarkeit mittels eines Kontrollunterdiagramms

Sei $D = (\mathcal{V}, \mathcal{E}, l)$ ein Kontrolldiagramm, $D' = (\mathcal{V}', \mathcal{E}', l') \subseteq D$ ein Kontrollunterdiagramm von D. Ferner seien G, G' Graphen. Dann heißt G' ableitbar aus G vermöge D' (in Zeichen: G ~D'~> G') $\Leftrightarrow$
Es existieren Graphen $G_1, \ldots, G_n$, Knoten $v_1, \ldots, v_n \in \mathcal{V}'$ und Kantenmarkierungen $el_1, \ldots el_{n-1} \in \{\ \underline{\text{abort}}, \underline{\text{commit}}\ \}$ mit:

(1) $G_1 = G,\ G_n = G'$

(2) $l'(v_1) = \underline{\text{start}},\ l'(v_n) = \underline{\text{stop}}$

(3) $\forall\ 1 \leq i < n : (v_i, el_i, v_{i+1}) \in \mathcal{E}'$, d.h. durch die Knoten und die Kantenmarkierungen wird ein Pfad durch D' beschrieben

(4) $G_2 = G_1$ (d.h. die Abarbeitung des Startknotens hat keine graphverändernde Wirkung)

(5) Für alle $2 \leq i < n$ gilt:

(5.1) el_i = <u>commit</u> $\wedge$ $l'(v_i)$ = <u>call</u> O mit $O \in \mathcal{O} \Rightarrow$
G_{i+1} ist aus G_i ableitbar vermöge der durch O bezeichneten Operation

(5.2) el_i = <u>commit</u> $\wedge$ $l'(v_i)$ = <u>call</u> v mit $v \in \mathcal{V} \Rightarrow$
G_{i+1} ist aus G_i ableitbar vermöge des Kontrollunterdiagramms mit dem Startknoten v

(5.3) el_i = <u>abort</u> $\wedge$ $l'(v_i)$ = <u>call</u> O mit $O \in \mathcal{O} \Rightarrow$
die mit O bezeichnete Operation ist auf G_i nicht anwendbar, und es ist $G_{i+1} = G_i$

(5.4) el_i = <u>abort</u> $\wedge$ $l'(v_i)$ = <u>call</u> v mit $v \in \mathcal{V} \Rightarrow$
es gibt keinen aus G_i vermöge des Kontrollunterdiagramms mit Startknoten v ableitbaren Graphen, und es ist $G_{i+1} = G_i$ ■

Darauf aufbauend läßt sich nun die erfolgreiche Ausführbarkeit eines Kontrollunterdiagramms definieren:

Def. 2.6.4 Erfolgreiche Ausführbarkeit eines Kontrollunterdiagramms

Es seien D, D' und G wie oben. Dann heißt D' erfolgreich ausführbar auf G $\Leftrightarrow$
es gibt einen Graphen G' mit G ~D'~> G'. ■

Im folgenden definieren wir nun die Semantik von Kontrollstrukturen, indem wir entsprechende Kontrolldiagramme angeben. Dabei beschränken wir uns auf diejenigen Sprachkonstrukte, die in der Beispielspezifikation aus Abb. 2.22 auf Seite 48 verwendet wurden, und verweisen ansonsten auf die einschlägige Literatur (s. Abschnitt 2.7). A, A_1, A_2, . . . bezeichnen im folgenden Aufrufe von elementaren Operationen oder Kontrolldiagrammen.

Def. 2.6.5 "not"-Operator

Abb. 2.23 zeigt Syntax und Semantik des "not"-Operators. ■

Syntax: not A (A darf keine graphverändernde Operation sein)

Kontrolldiagramm:

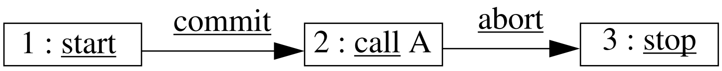


Abb. 2.23: Definition des "not"-Operators

Der "not"-Operator ist dazu gedacht, Bedingungen zu formulieren. Die Forderung, daß A keine graphverändernde Operation sein darf, wird nur erhoben, um die Lesbarkeit von PROGRESS-Spezifikationen zu erhöhen. Man beachte, daß "not A" auch dann seiteneffektfrei wäre, wenn A eine graphverändernde Operation wäre: Entweder schlägt nämlich A fehl und verändert den Wirtsgraphen nicht, oder A ist erfolgreich, und das Kontrollunterdiagramm kann nicht erfolgreich ausgeführt werden. Letzteres impliziert wegen der Atomarität, daß die Effekte von A zurückgesetzt werden.

Will man eine graphverändernde Operation trotz der Vorschrift aus Def. 2.6.5 als Argument des "not"-Operators verwenden, so muß man sie mit Hilfe des Operators "def" seiteneffektfrei machen:

Def. 2.6.6 Der "def"-Operator

Abb. 2.24 zeigt Syntax und Semantik des "def"-Operators. ■

Syntax : def A

Kontrolldiagramm:

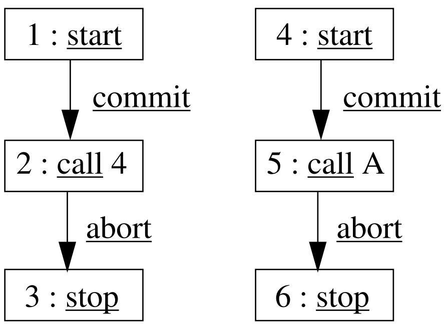


Abb. 2.24: Definition des "def"-Operators

Bei der Abarbeitung des Kontrolldiagramms für den "def"-Operator sind zwei Fälle zu unterscheiden:

(1) A ist erfolgreich. Dann schlägt das rechte Unterdiagramm fehl und läßt den Wirtsgraphen unverändert. Das linke Unterdiagramm wird erfolgreich durchlaufen. Insgesamt wird also "commit" zurückgeliefert, ohne daß der Wirtsgraph verändert wird.

(2) A schlägt fehl. Dann ist das rechte Unterdiagramm erfolgreich, ohne den Wirtsgraphen zu verändern, und das linke Unterdiagramm schlägt fehl.

Def. 2.6.7 "&"-Operator (Sequenz)

Abb. 2.25 zeigt Syntax und Semantik des "&"-Operators. ■

Syntax : A_1 & . . . & A_n

Kontrolldiagramm :

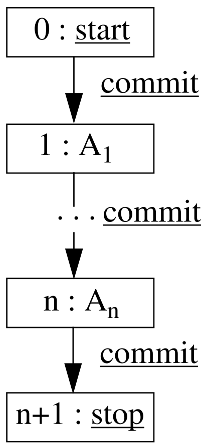


Abb. 2.25: Definition des "&"-Operators

Das Kontrolldiagramm zum "&"-Operator wird nur dann erfolgreich durchlaufen, wenn alle Operationen $A_1, \ldots, A_n$ erfolgreich nacheinander ausführbar sind. Dabei können diese Operationen i.a. unterschiedliche Ergebnisse haben. Falsche Entscheidungen bei der Ausführung der Operationen $A_1, \ldots, A_i$, die eine erfolgreiche Ausführung von A_{i+1} unmöglich machen, führen noch nicht zum Scheitern der Sequenz, sondern das Backtracking setzt ein.

Def. 2.6.8 "or"-Operator (Alternative)

Abb. 2.26 zeigt Syntax und Semantik des "or"-Operators (der Alternative). ■

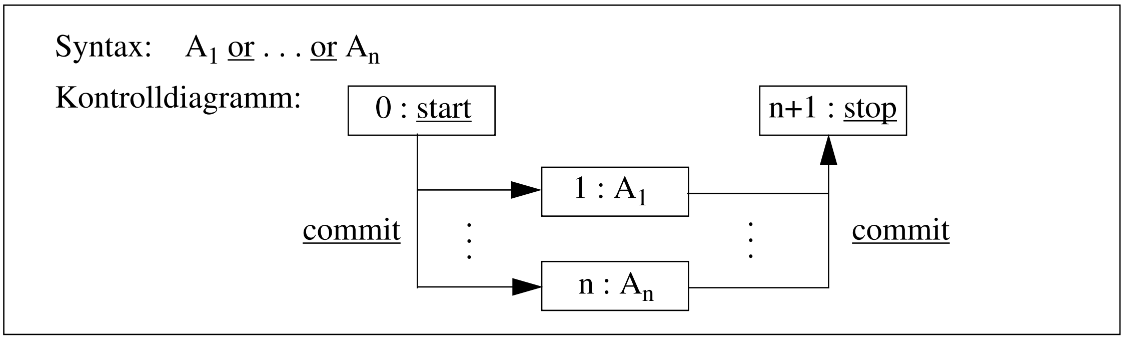


Abb. 2.26: Definition des "or"-Operators

Der "or"-Operator stellt eine nichtdeterministische Auswahl dar. Hier tritt erstmalig der Nichtdeterminismus auf der Ebene der Kontrollstrukturen zutage; bisher steckte er ausschließlich in der Anwendung von Graphersetzungsregeln.

Schließlich verbleibt noch die "loop"-Schleife, die hier in einer gegenüber PROGRESS leicht vereinfachten Form dargestellt wird:

Def. 2.6.9 "loop"-Operator (Schleife)

Abb. 2.27 zeigt Syntax und Semantik des "loop"-Operators. ■

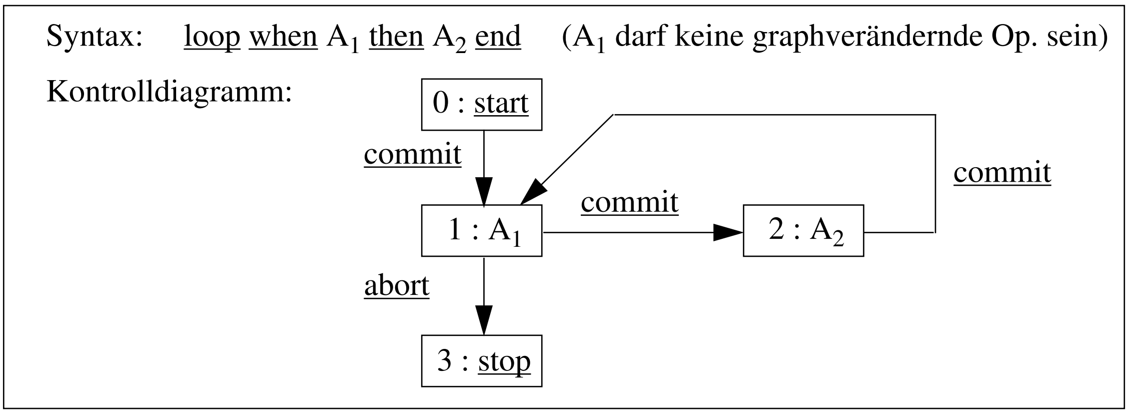


Abb. 2.27: Definition des "loop"-0perators

Die Ausführung einer Schleife kann nur dann scheitern, wenn A_1 erfolgreich ist und A_2 scheitert. Auch hier ist zu beachten, daß alle Möglichkeiten zur Ausführung von A_1 und A_2 in Betracht gezogen werden müssen.

Zum Abschluß dieses Abschnitts greifen wir das Fährmann-Problem noch einmal auf und illustrieren beispielhaft einige mögliche Abläufe.

Bsp. 2.6.10 Das Fährmann-Problem

Die Transaktionen zur Lösung des Fährmann-Problems wurden in Abb. 2.22 auf Seite 48 dargestellt. Bei erstmaligem Aufruf der Transaktion "SolutionWith_N_Actions" hat der Eingabeparameter "NoOfActions" den - nicht ausreichenden - Wert 5.

(1) Wird im ersten Schleifendurchlauf "TransportCargo" zum Transportieren des Kohlkopfs verwendet, führt dies zu einer verbotenen Situation, die im letzten Schritt der Sequenz erkannt wird. Dies führt zu einem Backtracking innerhalb der Sequenz.

(2) Wird im ersten Schleifendurchlauf die Ziege transportiert und anschließend fünfmal "FerryOver" angewendet, so wird die maximale Anzahl von Aktionen überschritten. Das Backtracking innerhalb der Sequenz nützt nun nichts mehr; es müssen auch die in vorherigen Schleifendurchläufen ausgewählten Aktionen zurückgesetzt werden.

(3) Schließlich scheitert die Schleife als ganzes, nachdem alle möglichen Aktionsfolgen erfolglos ausprobiert worden sind. Damit scheitert auch die Transaktion "CreateSolutionWith_N_Actions(5)" insgesamt. ■

2.7 Literatur

Einen ausgezeichneten Überblick über den mengentheoretischen Ansatz zur Definition von Graphregelsystemen liefert /Na 79/. Dort wird insbesondere ausführlich darauf eingegangen, welchen Einfluß die Gestalt von linken und rechten Seiten sowie der Einbettungsüberführungsregeln auf die Klasse der Graphsprachen hat. Weiterhin findet der Leser dort einen Abschnitt über die Syntaxanalyse für Graphgrammatiken sowie einen ersten Ansatz zur Programmierung mit Hilfe von Graphtests und Graphersetzungen.

Die Syntaxanalyse mit Präzedenz-Graphgrammatiken wird ausführlich und formal in /Ka 85/ beschrieben. In /Ka 82/ wird ein knapper Überblick über dieses Thema vermittelt.

Die Spezifikationssprache PROGRESS wird ausführlich in /Sc 91/ dargestellt. Als formale Grundlage für die Semantikdefinition dient dort jedoch nicht die Mengentheorie, sondern ein Logikkalkül. Die in PROGRESS verwendeten Kontrollstrukturen sind nicht nur dort, sondern auch in /SZ 91/ beschrieben.

Referenzen

/Hi 90/ M. Himsolt: *GraphEd User Manual*; Technischer Bericht, MIP - intern 1/90, Universität Passau

/Ka 82/ M. Kaul : *Parsing of graphs in linear time*, in : H. Ehrig, M. Nagl, G. Rozenberg: Graph-Grammars and Their Application to Computer Science, LNCS 153, 206-218 (1982)

/Ka 85/ M. Kaul : *Syntaxanalyse für Präzedenz-Graph-Grammatiken*, Dissertation, Universität Passau (1985)

/Na 79/ M. Nagl : *Graph-Grammatiken: Theorie, Implementierung, Anwendungen*, Vieweg Verlag (1979)

/Sc 91/ A. Schürr: *Operationales Spezifizieren mit programmierten Graphersetzungssystemen*, Deutscher Universitäts Verlag (1991)

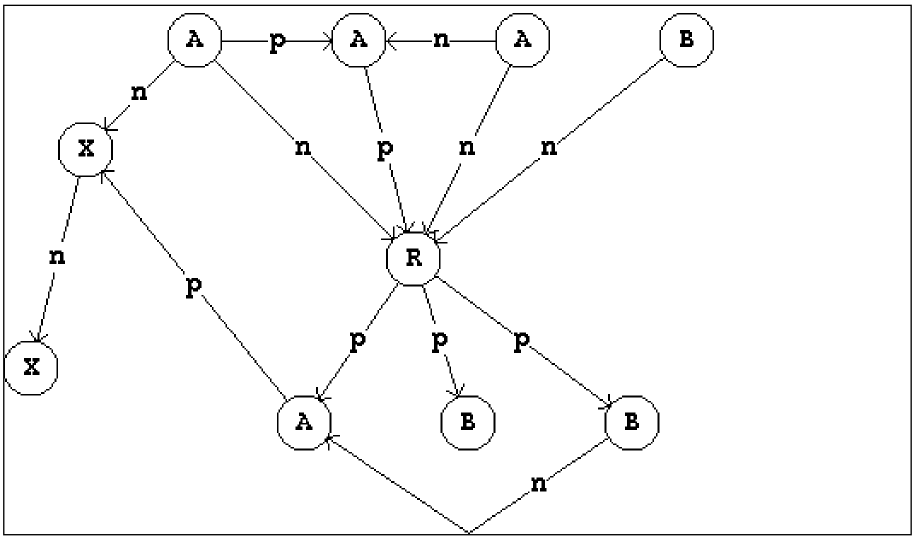
/SZ 91/ A. Schürr, A. Zündorf : *Nondeterministic Control Structures for Graph Rewriting Systems*, in : G. Schmidt, R. Berghammer : Graph-Theoretic Concepts in Computer Science, LNCS 570, 48-62 (1991)

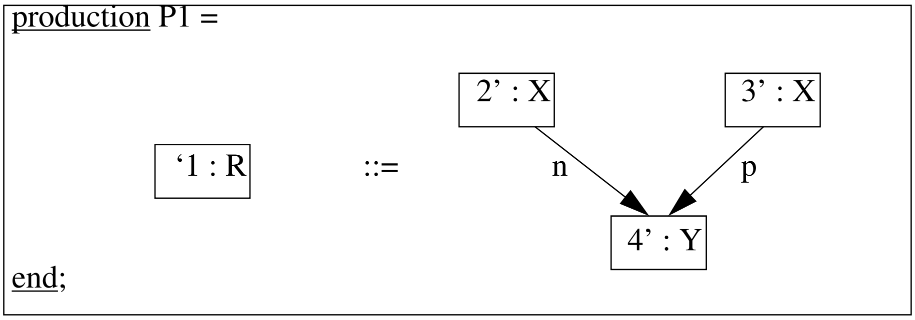
2.8 Aufgaben

Aufgabe 2.1 Anwendung einer Graphersetzungsregel

Gegeben sei folgender Graph, folgende Produktion und folgende Embeddingmenge (bzw. globale Einbettungsüberführungsregel)::

G:





EM = { (B, O, p, R, Y, O, p), (A, I, n, R, X, I, n) }

Zeichnen sie den Graphen, der durch Anwendung von P1 auf den Knoten mit der Markierung R unter Berücksichtigung der Embeddingmenge entsteht.

Aufgabe 2.2 Textuelle Graphbeschreibung

Zeichne folgenden Graphen:

G = ({ A, B, C, D, E },
{ (C, n, A), (B, n, C), (B, n, D), (B, n, E), (D, n, A), (E, n, A), (E, n, C)},
f) mit f(x) = x .

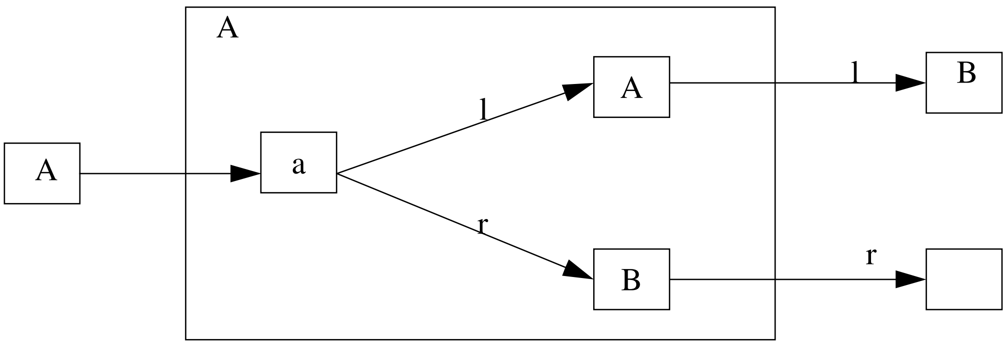
Aufgabe 2.3 Graphgrammatik für Syntaxdiagramme

(1)Erstellen Sie eine kontextfreie Graphgrammatik für Syntaxdiagramme. Orientieren Sie sich an den Beispielen zur Vorlesung (Abb. 2.3, Abb. 2.6).

- Verwenden Sie wenn möglich einbuchstabige Symbole für die Markierungen
- Die Kanten müssen nicht unbedingt markiert sein.
- Versuchen Sie mit möglichst wenig i (in) - und o (out) -Knoten auszukommen.
- Um ihre Grammatik zu testen, leiten Sie (in etwa) das Beispieldiagramm der Vorlesung (Abb. 2.5) ab.

(2)Begründen Sie an einem Beispiel, warum man nicht ganz auf in- und out-Knoten verzichten kann.

Nutzen Sie für Ihre Lösung die folgende grafische Notation aus /Hi 90/:



Das zu ersetzende Nonterminal wird durch ein großes Rechteck dargestellt. Oben links in der Ecke dieses Rechtecks ist das Label angegeben. Die rechte Regelseite bzw. der ersetzende Teilgraph ist im Innern des großen Rechtecks gezeichnet. Auch die Embedding-Regeln werden grafisch notiert. Eine Kante mit der Markierung m von einem Knoten A ausserhalb des großen Rechtecks zu einem Knoten a in dessen Innern beschreibt die Embedding-Anweisung:

(A, I, m, A, a, I, m)

Fehlende Markierungen sind Wildcards für beliebige erlaubte Markierungen.

Aufgabe 2.4 Gefädelte Binärbäume

(1) Erstellen Sie eine kontextfreie Graphgrammatik für die Ableitung gefädelter, sortierter Binärbäume (Die Knoten des Binärbaumes sind zusätzlich zu einer linearen Liste verkettet.). Für die Erzeugung der Terminalsymbole brauchen Sie nur eine Beispielsproduktion angeben.

(2) Testen Sie Ihre Graphgrammatik, indem Sie in den zunächst leeren Baum nacheinander folgende Elemente einfügen: 7, 4, 6, 9, 2.

(3) Kann man mit den bisher vorgestellten Graphgrammatiktypen eine Produktion zum Löschen eines Blattes in einem gefädeltem Binärbaum angeben? Wo liegen dabei die Schwierigkeiten?

Aufgabe 2.5 Implementierung in Modula-2

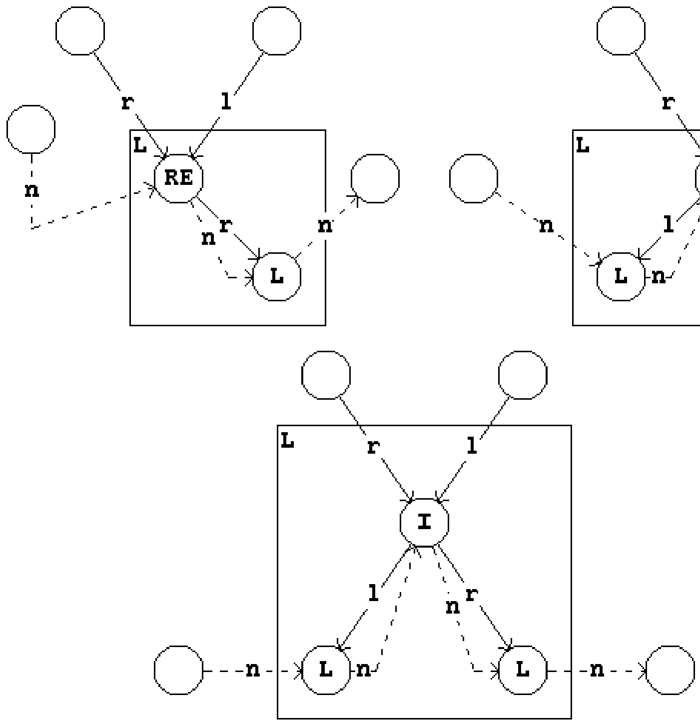
Gegeben sei folgender Ausschnitt aus Modula-2-Typdefinitionen für gefädelte Binärbäume:

```
IMPLEMENTATION MODULE ThreadedBinTree;

   FROM SYSTEM IMPORT ADDRESS;

   TYPE

      TBTElemPtr = POINTER TO TBTElemRec;

      TBTreeRec =
         RECORD
            Root    : TBTElemPtr;
            Thread  : TBTElemPtr;
         END;

      TBTElemRec =
         RECORD
            DataPtr : ADDRESS;
            Left    : TBTElemPtr;
            Right   : TBTElemPtr;
            Next    : TBTElemPtr;
         END;


   (* Anfuegen eines linken Sohns als Blatt in den Baum *)
   PROCEDURE TBTAddLeftLeaf( VAR Tree    : TBTreeRec;  (* Der Baum *)
                                 Father  : TBTElemPtr; (* Element an das
                                                          angefuegt wird *
                                 Previos : TBTElemPtr; (* Vorgaenger im
                                                          Thread (evtl. NIl
                                 New     : TBTElemPtr);(* Neues Element,
                                                          fertig konstruie
                                                          einfach reintun 
   BEGIN

       (* . . .  your turn! *)

   END TBTAddLeftLeaf;

END ThreadedBinTree.
```

Vervollständigen Sie die Prozedur TBTAddLeftLeaf.

Aufgabe 2.6 Präzendenzparser für gefädelte Binärbäume

Gegeben sie folgende Graph-Grammatik:

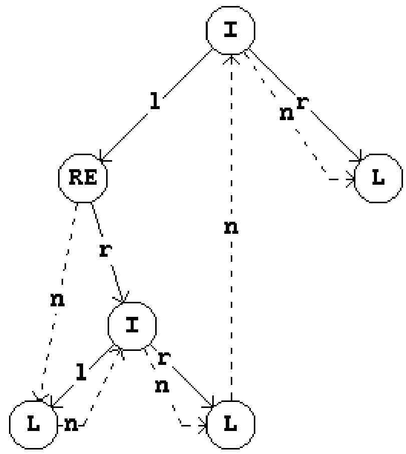


(1) Erfüllt diese Grammatik die in Def. 2.3.3 geforderte Monotonieeigenschaft? Wenn nein (womit sich die Frage fast erübrigt), ergänzen Sie die Grammatik geeignet.

(2) Erstellen Sie die in Def. 2.3.6 und Def. 2.3.8 vorgestellten Relationen In^+, Out^+, $\equiv$, $<$, $>$ und $<>$ für die in (1) erstellte Grammatik. Sind diese präzedenz-konfliktfrei?

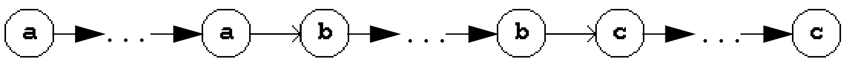
(3) Sind die Relationen aus (2) präzedenz-konfliktfrei wenn man die Fädelungskanten nicht berücksichtigt?

(4)Markieren Sie alle Henkel und reduzieren Sie folgenden Graph vollständig.



Aufgabe 2.7 $a^nb^nc^n$

Erstellen sie eine kontextfreie Graphgrammatik die Graphen folgender Struktur erzeugt:



Dabei sollen die erzeugten Graphen immer genau gleich viele a, b und c Knoten enthalten.

3. Kategorientheoretischer Ansatz

Im vorigen Kapitel haben wir unterschiedliche Arten von Graphgrammatiken und Graphersetzungssystemen für verschiedene Anwendungsgebiete kennengelernt. Ihnen allen war eines gemein: die formale Definition mit den Hilfsmitteln der Mengentheorie. In diesem Kapitel werden wir nun den formalen Apparat der Mengentheorie gegen den etwas eleganteren, dafür aber auch weniger vertrauten Formalismus der **Kategorientheorie** austauschen.

Um auch aus der Sicht des Anwenders (von Graphregelsystemen) neue Aspekte zu behandeln, werden wir uns jedoch nicht damit begnügen, alle Definitionen des vorigen Kapitels mit Hilfe eines neuen theoretischen Fundaments zu reformulieren. Vielmehr werden wir auf die Einführung kategorientheoretisch definierter Graphgrammatiken verzichten und gleich auf eine **neue Art von Graphersetzungssystemen** zusteuern, die die identische Ersetzung von Knoten (und Kanten) unterstützen (einen solchen Graphersetzungsbegriff hätten wir natürlich auch mit den formalen Hilfsmitteln des vorigen Kapitels einführen können). Da ein entsprechender Graphersetzungsbegriff und die damit einhergehende Notation von Graphersetzungsregeln stark von den Festlegungen in Kapitel 2 abweicht, werden wir ihn im Abschnitt 3.1 zunächst einmal informell einführen.

Erst danach werden wir im Abschnitt 3.2 daran gehen, alle Grundbegriffe aus Abschnitt 2.1 zu redefinieren und im Abschnitt 3.3 den neuen (sequentiellen) Graphersetzungsbegriff formal einzuführen. Zum Abschluß werden wir in Abschnitt 3.4 völlig neue Möglichkeiten des Umgangs mit Graphersetzungsregeln behandeln, die sich mit den Hilfsmitteln der Kategorientheorie leichter als mit dem mengentheoretischen Fundament formal erfassen lassen. Dazu gehören

- die Erzeugung von Graphersetzungsregeln durch Graphersetzungsregeln,
- die gleichzeitige (parallele) Anwendung mehrerer Graphersetzungsregeln auf einen Graphen und
- insbesondere die Verschmelzung mehrerer parallel anwendbarer Ersetzungsregeln zu einer semantisch äquivalenten neuen Graphersetzungsregel.

3.1 Informelle Einführung

Bevor wir im nachfolgenden Abschnitt das kategorientheoretische Fundament für einen neuen Graphersetzungsbegriff einführen werden, wollen wir diesen zunächst informell vorstellen. In Kapitel 2 hatten wir die sogenannten “Untergraphersetzungen” definiert, die sich dadurch auszeichnen, daß

- das Abbild der zu ersetzenden linken Seite ein Untergraph des Wirtsgraphen zu sein hat (alle Kanten zwischen zu ersetzenden Knoten müssen in der linken Regelseite aufgeführt sein) und
- der zu ersetzende Untergraph ganz gelöscht und durch ein Abbild der rechten Regelseite vollständig ersetzt wird.

In diesem Kapitel werden wir nun die "**Teilgraphersetzungen**" einführen, die

- die identische Ersetzung von Knoten (und Kanten) mit Übernahme des alten Kontextes unterstützen und
- nur noch bei den zu löschenden Knoten fordern, daß alle sie berührenden Kanten Bestandteil der linken Seite der Ersetzungsregel sind.

Der identisch zu ersetzende Anteil einer linken Regelseite muß also nur noch ein Teilgraph des betrachteten Wirtsgraphen sein. Da bei der identischen Ersetzung von Knoten deren Kontext automatisch erhalten bleibt, läßt sich auf diese Weise die Einbettung neu erzeugter Teile in den Wirtsgraphen direkt in der rechten Regelseite festlegen. Man kann also auf die expliziten Einbettungsüberführungsregeln aus Kapitel 2 im Prinzip verzichten[†].

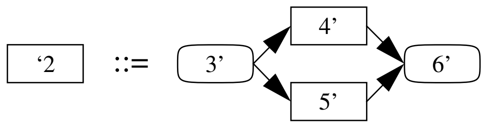
Wie man sich den Aufbau solcher Teilgraphersetzungsregeln und ihre Anwendung auf einen Graphen vorzustellen hat, soll zunächst an dem wohlvertrauten Beispiel der Syntaxdiagramme vorgeführt werden (im weiteren Verlauf dieses Kapitels werden wir uns dann einem neuen Beispiel zuwenden). Nehmen wir uns die Graphersetzungsregel "CreateBranch" aus Abb. 2.3 auf Seite 10 vor, die eine Verzweigung in einem Syntaxdiagramm erzeugt. Eine vereinfachte Version dieser Ersetzungsregel (ohne Knoten- und Kantenmarkierungen) findet man in der Abb. 3.1 a). Ihre expliziten Einbettungsüberführungsregeln müssen wir durch die identische Ersetzung von Knoten "simulieren". Einen ersten Versuch hierzu sieht man in der Abb. 3.1 b). Dort haben wir die linke Seite der Regel um zwei weitere Knoten angereichert, die identisch ersetzt werden sollen.

Auf den ersten Blick ist die neu konstruierte Ersetzungsregel leichter zu verstehen als die vorherige Version. Auf den zweiten Blick zeigt sich jedoch, daß sie in einem entscheidenden Punkt fehlerhaft ist. Sie läßt sich nämlich nur auf solche ("Sub"-) Knoten anwenden, die **genau eine einlaufende** und **genau eine auslaufende** ("next"-) Kante besitzen. Wie wir im folgenden nämlich noch sehen werden, verbietet der kategorientheoretische Ersetzungsbegriff das Löschen von Knoten, deren Kontext nicht vollständig in der linken Regelseite aufgeführt ist (und selbst wenn das Löschen erlaubt wäre, so würde die neu erzeugte Verzweigung in solchen Fällen falsch eingebettet werden).

---

† Auf die praktischen Vor- und Nachteile der verschiedenen Einbettungsüberführungsmöglichkeiten werden wir im folgenden noch zu sprechen kommen.

production CreateBranch =

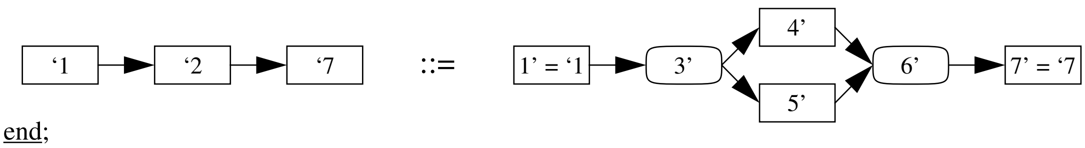


embedding copy <-∗- to 3’, -∗-> to 6’;

end;

a) Untergraphersetzung mit expliziter Einbettungsüberführungsregel

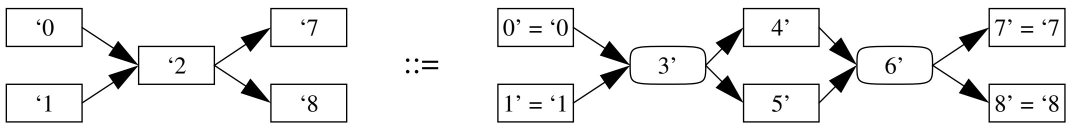
production CreateBranch =



end;

b) Teilgraphersetzung mit Einbettungsüberführung durch identische Ersetzung

production CreateBranch =



end;

c) Teilgraphersetzung mit korrekter Einbettungsüberführung

Abb. 3.1: Verschiedene Varianten der Einbettungsüberführung

In unserem Anwendungsbeispiel haben wir das Glück, daß maximal zwei ("next")-Kanten in einen Knoten einlaufen bzw. aus ihm auslaufen. Deshalb müssen wir nur zwei weitere identisch zu ersetzende Knoten einführen, um auch den Fall mit (genau) zwei ein- und auslaufenden Kanten korrekt behandeln zu können (siehe Abb. 3.1 c)). Damit die neu konstruierte Ersetzungsregel nicht nur in dem Fall genau zwei ein- bzw. auslaufender Kanten anwendbar ist, erlaubt man bei Bedarf die Zuordnung verschiedener identisch zu ersetzender Knoten (und Kanten) der linken Regelseite zu ein und demselben Knoten (Kante) im Wirtsgraphen. Auf diese Weise kann man also - mehr oder weniger umständlich und unübersichtlich - die expliziten Einbettungsüberführungsregeln des vorigen Kapitels auf die identische Ersetzung von Knoten zurückführen (falls es für die Anzahl der ein- bzw. auslaufenden Kanten eine obere Schranke gibt).

Bislang haben wir Teilgraphersetzungsregeln in der Syntax der Sprache PROGRESS notiert. So wurde die identische Ersetzung eines Knotens ‘x durch eine entsprechende Knoteninschrift " y' = ‘x " auf der rechten Regelseite zum Ausdruck gebracht. Oft wird jedoch der identisch zu ersetzende Anteil einer Graphersetzungsregel durch einen eigenen Graphen festgehalten, und seine Entsprechungen in linker und rechter Regelseite werden durch zwei (injektive) Abbildungen ausgezeichnet. Damit besteht eine Graphersetzungsregel

$$r := \; L \xleftarrow{\;k\;} K \xrightarrow{\;k'\;} R$$

aus drei Graphen und zwei Abbildungen:

(1) L ist die linke Regelseite.

(2) K[†] legt den identisch zu ersetzenden Anteil fest.

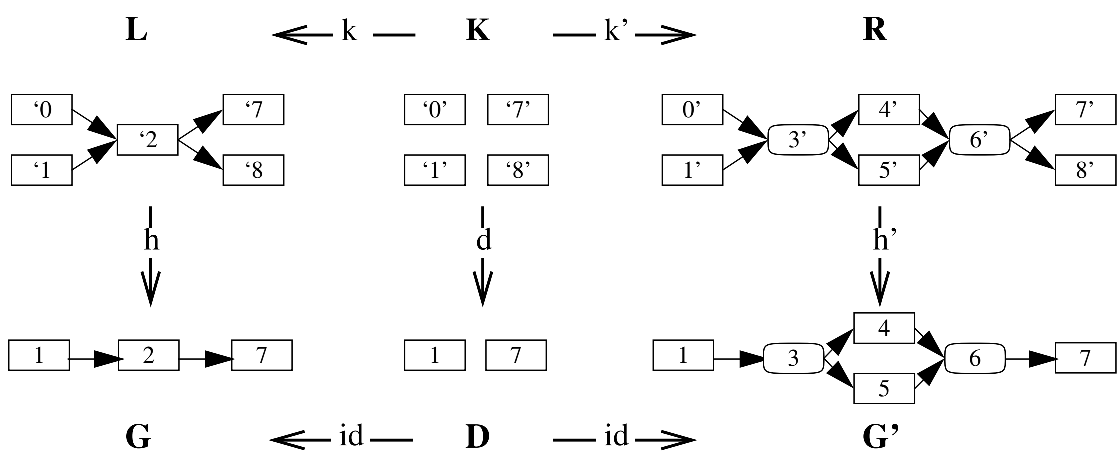
(3) R ist die rechte Regelseite.

(4) $k\colon K \to L$ zeichnet in L den Anteil aus, der identisch ersetzt und damit nicht gelöscht wird.

(5) $k'\colon K \to R$ zeichnet in R den Anteil aus, der identisch ersetzt und damit nicht neu eingefügt wird.

Ein Beispiel für diese Form der Niederschrift von Graphersetzungsregeln findet man in der Abb. 3.2. Es wurde mit Absicht so gewählt, daß k und k' **injektive Abbildungen** sind. Denn im folgenden werden wir uns auf diesen theoretisch einfacher handzuhabenden Fall beschränken (und oft sogar k und k' mit der Identität gleichsetzen). Mit der Einschränkung auf injektive Abbildungen ist eine geringe Einbuße an Ausdruckskraft verbunden; im Zuge einer Graphersetzung lassen sich so weder Kopien von Knoten erzeugen (k wäre dann nicht injektiv) noch mehrere Knoten zu einem Knoten verschmelzen (k' wäre dann nicht injektiv).

† "K" steht für Kontext.



Definition der Abbildungen:

k := { (‘0’, ‘0), … } .
k’ := { (‘0’, 0’), … } .
h := { (‘0, 1), (‘1, 1), (‘2, 2), (‘7, 7), (‘8, 7) } .
d := { (‘0’, 1), (‘1’, 1), (‘7’, 7), (‘8’, 7) } .
h’ := { (0’, 1), (1’, 1), (3’, 3), … , (7’, 7), (8’, 7) } .

Abb. 3.2: Anwendung einer Teilgraphersetzungsregel

Die Anwendung einer Teilgraphersetzungsregel auf einen Graphen G läuft damit in etwa wie folgt ab:

(1) Durch eine Abbildung h: L → G wird der zu ersetzende Teilgraph in G festgelegt.

(2) Dann ist h(k(K)) der identisch zu ersetzende Anteil in G und h(L \ k(K)) der zu löschende Anteil.

(3) Nach dem Löschen erhält man also D[†] := G \ h(L \ k(K)).

(4) Mit h’, einer Abbildung, die alle Knoten (und Kanten) in R so umbenennt, daß sie mit den Bezeichnern in D nicht (mehr) in Konflikt geraten, ist dann h’(R \ k’(K)) der neu zu erzeugende Anteil.

(5) Damit ist G’ := D ⊕ h’(R \ k’(K)) das Ergebnis der Graphersetzung.

Ein Beispiel für diese Art der Teilgraphersetzung, in dem h und h’ keine injektiven Abbildungen sind, sieht man in Abb. 3.2.

† “D” steht für Differenz.

3.2 Grundbegriffe

Im vorigen Kapitel hatten wir als Datenmodell die gerichteten, knoten- und kantenmarkierten Graphen eingeführt. Sie hatten den Nachteil, daß nur die in ihnen enthaltenen Knoten, nicht aber ihre Kanten “echte” Objekte sind. Damit war es nicht möglich,

- parallel verlaufende Kanten gleicher Markierung zwischen zwei Knoten zu ziehen,
- Kanten - ähnlich wie Knoten - zu attributieren oder
- n-stellige Relationen (mit $n \neq 2$) auf den Knoten eines Graphen als Kanten darzustellen.

Da zudem die Modellierung von Kanten als eigenständigen Objekten im Rahmen der kategorientheoretischen Ansätze eher von Vorteil ist, werden wir diese Sichtweise im folgenden pflegen. Um den Unterschied zu dem Datenmodell aus Kapitel 2 klar hervorzuheben, werden wir im folgenden die Begriffe **Hyperkante** und **Hypergraph** anstelle von Kante und Graph benutzen.

Def. 3.2.1 Markierte Hypergraphen

Ein 6-Tupel $(\mathcal{V}, \mathcal{E}, l_V, l_E, s, t)$ ist markierter Hypergraph über zwei Markierungsalphabeten $\mathcal{L}_V$ und $\mathcal{L}_E \Leftrightarrow$

(1) $\mathcal{V}$ ist eine (endliche) Menge von Knoten (-bezeichnern).

(2) $\mathcal{E}$ ist eine (endliche) Menge von Hyperkanten (-bezeichnern).

(3) l_V: $\mathcal{V} \rightarrow \mathcal{L}_V$ ist die Knotenmarkierungsfunktion, die jedem Knoten des Hypergraphen genau eine Markierung zuordnet.

(4) l_E: $\mathcal{E} \rightarrow \mathcal{L}_E$ ist die Kantenmarkierungsfunktion, die jeder Hyperkante des Hypergraphen genau eine Markierung zuordnet.

(5) $s \subseteq \mathcal{E} \times \mathcal{V}$ ist eine Relation, die jeder Hyperkante des Hypergraphen eine ggf. leere Menge von Quellknoten zuordnet.

(6) $t \subseteq \mathcal{E} \times \mathcal{V}$ ist eine Relation, die jeder Hyperkante des Hypergraphen eine ggf. leere Menge von Zielknoten zuordnet. ■

Im folgenden werden wir mit G, G’, L, … immer Hypergraphen bezeichnen. Ihre Komponenten sollen vereinbarungsgemäß mit $(\mathcal{V}, \mathcal{E}, l_V, l_E, s, t)$ bei G, $(\mathcal{V}', \mathcal{E}', l_V', l_E', s', t')$ bei G’ und $(\mathcal{V}_L, \mathcal{E}_L, l_V(L), l_E(L), s_L, t_L)$ bei L etc. benannt werden.

Def. 3.2.2 Menge von Hypergraphen

Es seien $\mathcal{L}_V$ und $\mathcal{L}_E$ zwei (disjunkte) Markierungsalphabete. Dann wird die Menge aller markierten Hypergraphen über $\mathcal{L}_V$ und $\mathcal{L}_E$ mit $\mathcal{G}_H(\mathcal{L}_V, \mathcal{L}_E)$ bezeichnet. ■

Bsp. 3.2.3 Ein “Eisenbahn”-Hypergraph

Es sei $\mathcal{L}_V$ = { Locomotive, RailJoint[†], … } und $\mathcal{L}_E$ = { On, Track[‡], … }. Dann ist der Hypergraph G aus Abb. 3.3 ein Hypergraph aus $\mathcal{G}_H(\mathcal{L}_V, \mathcal{L}_E)$ und sieht formal wie folgt aus:

(1) $\mathcal{V}$:= { 1, … , 7 }.

(2) $\mathcal{E}$:= { e1, … , e10 }.

(3) l_V := { (1, Locomotive[††]), (2, RailJoint), (3, RailJoint), … }.

(4) l_E := { (e1, On), (e2, Track), … }.

(5) s := { (e1, 1), (e2, 2), (e3, 2), … }.

(6) t := { (e1, 2), (e1, 3), (e2, 3), … }. ■

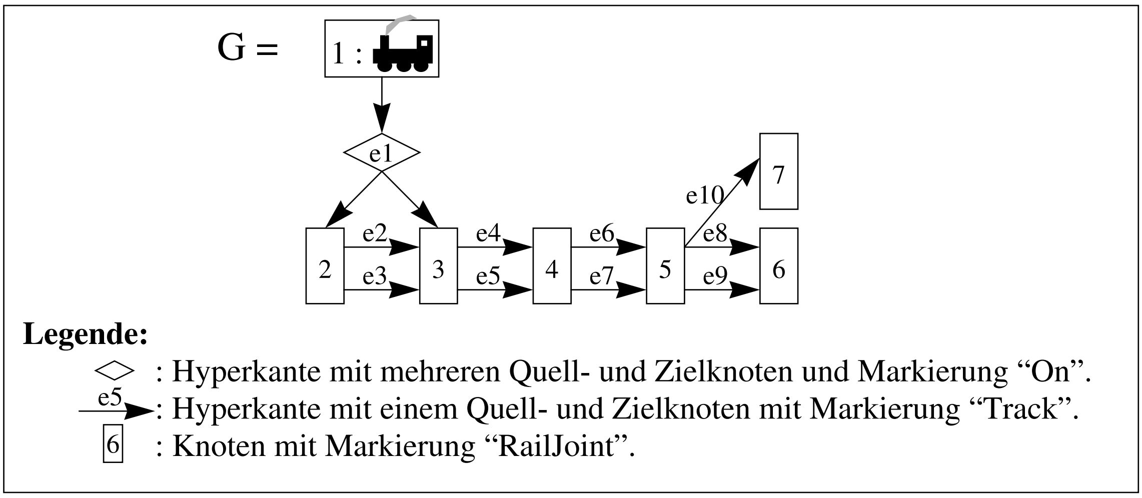


Abb. 3.3 : Ein Hypergraph

Der oben angegebene Hypergraph besteht aus 6 “RailJoint”-Knoten und 9 “Track”-Kanten, die die Schwellen und Schienen einer Eisenbahntrasse einschließlich einer auf Durchfahrt gestellten Weiche darstellen. Ein von einer Lokomotive befahrbarer Streckenabschnitt besteht aus zwei ihn begrenzenden Schwellenknoten und zwei parallel verlaufenden Schienenkanten, die die beiden Knoten miteinander verbinden. Befährt eine Lokomotive einen Streckenabschnitt, so ist sie über eine “On”-Kante mit den begrenzenden “RailJoint”-Knoten dieses Abschnitts verbunden (im weiteren Verlauf dieses Kapitels werden wir öfters zwei Lokomotiven demselben Streckenabschnitt zuordnen; es handelt sich dabei wie im wirklichen Leben um eine prinzipiell mögliche, aber unerwünschte Situation).

† RailJoint = Schwelle.
‡ Track = Schiene.
†† Die Markierung wird ausnahmsweise grafisch dargestellt.

Nach der formalen Einführung des Begriffs Hypergraph könnten wir nun die Begriffe Teilgraph und Untergraph ähnlich wie in Kapitel 2 definieren. Da wir aber im folgenden von ihnen keinen Gebrauch machen werden, wollen wir hier darauf verzichten und uns direkt den Begriffen **Homomorphismus** und **Isomorphismus** für Hypergraphen zuwenden.

Def. 3.2.4 (Homo-) Morphismus

Seien G, G' $\in \mathcal{G}_H(\mathcal{L}_V, \mathcal{L}_E)$. Ferner sei h := ($h_V$, h_E) mit h_V: $\mathcal{V} \rightarrow \mathcal{V}$' und h_E: $\mathcal{E} \rightarrow \mathcal{E}$' (im folgenden lassen wir die Indizes V und E der Komponenten von h weg, wenn sie sich aus dem Kontext bestimmen lassen). Dann wird h ein (Homo-) Morphismus oder auch Graphmorphismus von G nach G' genannt (in Zeichen: G $\tilde{\Rightarrow}$ G') $\Leftrightarrow$

(1) $\forall$ v $\in \mathcal{V}$: $l_V(v) = l_V'(h(v))$.

(2) $\forall$ e $\in \mathcal{E}$: $l_E(e) = l_E'(h(e))$.

(3) $\forall$ e $\in \mathcal{E}$, v' $\in \mathcal{V}$' : ($\exists$ v $\in \mathcal{V}$: h(v) = v' $\wedge$ (e, v) $\in$ s) $\Leftrightarrow$ (h(e), v') $\in$ s'.

(4) $\forall$ e $\in \mathcal{E}$, v' $\in \mathcal{V}$' : ($\exists$ v $\in \mathcal{V}$: h(v) = v' $\wedge$ (e, v) $\in$ t) $\Leftrightarrow$ (h(e), v') $\in$ t'. ■

Ein Morphismus h erhält also die Markierungen von Knoten und Hyperkanten. Zudem ist sichergestellt, daß das Abbild einer Hyperkante **genau** die Abbilder ihrer Quell- und Zielknoten als Quell- und Zielknoten besitzt.

Def. 3.2.5 Monomorphismus, Epimorphismus, Isomorphismus

Es sei h := (h_V, h_E) : G $\tilde{\Rightarrow}$ G'. Dann heißt h Isomorphismus (in Zeichen: h: G$\overset{\approx}{\Rightarrow}$ G') $\Leftrightarrow$

(1) h_V ist bijektiv.

(2) h_E ist bijektiv.

Die Graphen G und G' werden in diesem Falle isomorph zueinander genannt (in Zeichen: G $\cong$ G').
In analoger Weise werden Monomorphismen als injektive Homomorphismen und Epimorphismen als surjektive Homomorphismen definiert. ■

Bsp. 3.2.6 Morphismen

Es sei G der Hypergraph aus Abb. 3.3 und G_1 , G_2 die Hypergraphen aus Abb. 3.4. Dann gilt:

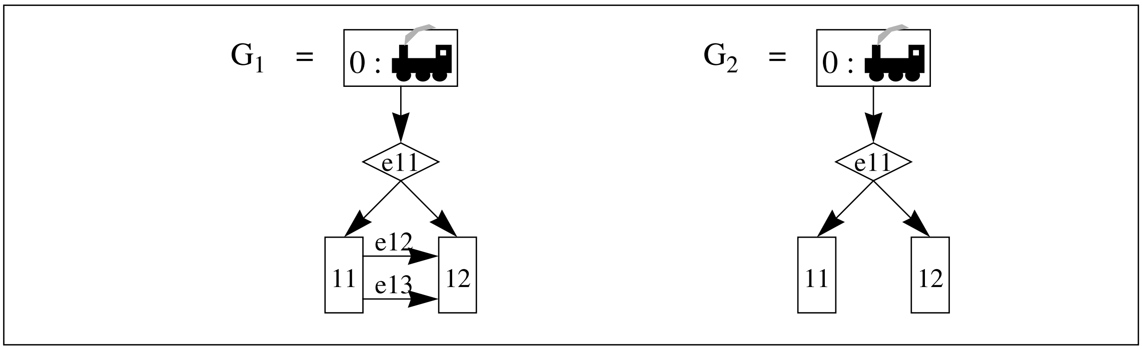


Abb. 3.4: Weitere Beispiele für Hypergraphen

(1) $h_1 : G_1 \to G := (\{ (0, 1), (11, 2), (12, 3) \}, \{ (e11, e1), (e12, e2), (e13, e5) \})$ ist kein Morphismus, da Quelle bzw. Ziel von e5 kein Abbild von 11 bzw.12 sind.

(2) $h_2 : G_1 \to G := (\{ (0, 1), (11, 2), (12, 3) \}, \{ (e11, e1), (e12, e2), (e13, e2) \})$ ist ein Morphismus, der weder injektiv noch surjektiv ist.

(3) $h_3 : G_2 \to G := (\{ (0, 1), (11, 2), (12, 2) \}, \{ (e11, e1) \})$ ist kein Morphismus, da das Abbild der Hyperkante e11 den Zielknoten 3 besitzt, der kein Abbild von 11 oder 12 ist.

(4) $h_4 : G_2 \to G := (\{ (0, 1), (11, 2), (12, 3) \}, \{ (e11, e1) \})$ ist ein Monomorphismus, aber natürlich kein Epimorphismus. ■

Mit Hilfe der eben definierten Graphmorphismen können wir nun - wie in Kapitel 2 - die Anwendungsstelle einer Graphersetzungsregel in einem Hypergraphen festhalten. Darüber hinaus werden wir mit ihrer Hilfe sogar die Vereinigung und die Differenzbildung auf Hypergraphen definieren können. Hierfür müssen wir aber zunächst die Eigenschaften von Graphmorphismen genauer untersuchen. Dazu benötigen wir den Begriff der "**Kategorie**".

Def. 3.2.7 Kategorie

Ein Tripel $(\mathcal{O}^{\dagger}, \mathcal{M}(O, O')_{O, O' \in \mathcal{O}}, \circ)$ ist eine Kategorie, wenn "$\circ$" eine assoziative Verknüpfung auf der Familie von Morphismen $\mathcal{M}(O, O')_{O, O' \in \mathcal{O}}$ mit neutralen Elementen ist, also:

(1) $\forall\, h \in \mathcal{M}(O, O')$ und $h' \in \mathcal{M}(O', O'')$: $h \circ h' \in \mathcal{M}(O, O'')$.

(2) $\forall\, h \in \mathcal{M}(O, O')$: $\exists\, id \in \mathcal{M}(O, O)$ und $id' \in \mathcal{M}(O', O')$: $id \circ h = h = h \circ id'$.

(3) $\forall\, h_1 \in \mathcal{M}(O_1, O_2), h_2 \in \mathcal{M}(O_2, O_3), h_3 \in \mathcal{M}(O_3, O_4)$:

$$h_1 \circ (h_2 \circ h_3) = (h_1 \circ h_2) \circ h_3 \;.\; ■$$

† "$\mathcal{O}$" steht für Object.

Satz 3.2.8 Die Kategorie der Graphmorphismen

Mit $\mathcal{G} := \mathcal{G}_H(\mathcal{L}_V, \mathcal{L}_E)$ einer Menge von Hypergraphen und dem unten definierten Kompositionsoperator "$\circ$" gilt:

$$(\mathcal{G}, (G \tilde{\rightarrow} G')_{G, G' \in \mathcal{G}}, \circ) \text{ ist eine Kategorie.}$$

Dabei sei $(h_{\mathcal{V}}, h_{\mathcal{E}}) \circ (h_{\mathcal{V}}', h_{\mathcal{E}}') := (h_{\mathcal{V}} \circ h_{\mathcal{V}}', h_{\mathcal{E}} \circ h_{\mathcal{E}}')$ mit $(f \circ f')(x) := f'(f(x))$.

Beweis:

Da "$\circ$" ein assoziativer Operator ist, muß nur gezeigt werden:

(1) Die identische Abbildung für Knoten und Hyperkanten ist ein Graphmorphismus für jeden Hypergraphen G: Das ergibt sich aber direkt aus der Def. 3.2.4.

(2) Die Komposition zweier Graphmorphismen h und h' ist wieder ein Graphmorphismus:

(2.1) h und h' erhalten die Markierungen von Knoten und Hyperkanten $\Rightarrow$ auch $h \circ h'$ erhält die Markierungen von Knoten und Hyperkanten.

(2.2) h und h' erhalten den "Kontext" einer Hyperkante $\Rightarrow$ auch $h \circ h'$ erhält den Kontext einer Hyperkante. ■

Betrachtet man die obenstehende Definition einer Kategorie, so stellt man fest, daß sie an unseren Operator "$\circ$" eigentlich nur selbstverständliche Anforderungen stellt. Damit liefert uns der Begriff der Kategorie also ein notwendiges Kriterium dafür, daß die Def. 3.2.4 für Graphmorphismen sinnvoll gewählt wurde. Im folgenden werden wir zudem sehen, daß wir hiermit eine hinreichende Basis für eine mathematisch sehr elegante Definition von Graphersetzungen und der Komposition von Graphersetzungsregeln "geschenkt" bekommen. Als Vorbereitung hierfür werden wir im Rest dieses Abschnitts die Vereinigung und Differenz von Hypergraphen sowohl konstruktiv mit den Hilfsmitteln der Mengentheorie als auch nicht-konstruktiv mit den Hilfsmitteln der Kategorientheorie einführen.

An dieser Stelle sei der Leser darauf hingewiesen, daß auch die in Abschnitt 2.1 eingeführten Homomorphismen zusammen mit den gerichteten, knoten- und kantenmarkierten Graphen eine Kategorie bilden. So könnte man auf den Gedanken kommen, auch die **Graphersetzungssysteme aus Kapitel 2 kategorientheoretisch zu definieren**. Wie wir im folgenden jedoch sehen werden, ist das **nicht möglich**, ohne die Ausdruckskraft des dort vorgestellten Ansatzes entscheidend einzuschränken.

Def. 3.2.9 Operationale Vereinigung von Hypergraphen

Seien $B, D \in \mathcal{G}_H(\mathcal{L}_V, \mathcal{L}_E)$ mit bis auf Homomorphie gemeinsamem Durchschnitt K; es gibt also $b: K \tilde{\rightarrow} B$ und $d: K \tilde{\rightarrow} D$. Dann ist G die Vereinigung von B und D bzgl.der Morphismen b und d (in Zeichen: $G := B \,{}_b\!\cup_d\, D$) $\Leftrightarrow$

(1) $G = (\mathcal{V}_G, \mathcal{E}_G, l_V(G), l_E(G), s_G, t_G)$.

(2) ~ ist die auf $\mathcal{V}_B \oplus \mathcal{V}_D$ (bzw. $\mathcal{E}_B \oplus \mathcal{E}_D$) durch b und d definierte Relation:
$x \sim y \iff \exists\, z \in \mathcal{V}_K$ (bzw. $\mathcal{E}_K$): $b(z) = x \;\wedge\; d(z) = y$.
(Die Relation setzt genau die Knoten und Kanten in Beziehung zueinander, die gemeinsame Urbilder in K besitzen.)

(3) Die Äquivalenzrelation ≈ ist die transitive, reflexive Hülle von ~.

(4) $\mathcal{V}_G := (\mathcal{V}_B \oplus \mathcal{V}_D) / \approx$, also:
$\mathcal{V}_G := \{\, [v] \mid v \in \mathcal{V}_B \oplus \mathcal{V}_D \,\}$ mit $[v] := \{\, v' \in \mathcal{V}_B \oplus \mathcal{V}_D \mid v' \approx v \,\}$.

(5) $\mathcal{E}_G := (\mathcal{E}_B \oplus \mathcal{E}_D) / \approx$, mit Äquivalenzklassenbildung analog zu der in (4).

(6) $\forall\, [v] \in \mathcal{V}_G$: $l_V(G)([v]) := \underline{\text{if}}\ v \in \mathcal{V}_B\ \underline{\text{then}}\ l_V(B)(v)\ \underline{\text{else}}\ l_V(D)(v)$.

(7) $\forall\, [e] \in \mathcal{E}_G$: $l_E(G)([e]) := \underline{\text{if}}\ e \in \mathcal{E}_B\ \underline{\text{then}}\ l_E(B)(e)\ \underline{\text{else}}\ l_E(D)(e)$.

(8) $\forall\, [e] \in \mathcal{E}_G, [v] \in \mathcal{V}_G$:
$([e], [v]) \in s_G \iff \exists\, v' \in [v]$: $\underline{\text{if}}\ e \in \mathcal{E}_B\ \underline{\text{then}}\ (e, v') \in s_B\ \underline{\text{else}}\ (e, v') \in s_D$.

(9) $\forall\, [e] \in \mathcal{E}_G, [v] \in \mathcal{V}_G$:
$([e], [v]) \in t_G \iff \exists\, v' \in [v]$: $\underline{\text{if}}\ e \in \mathcal{E}_B\ \underline{\text{then}}\ (e, v') \in t_B\ \underline{\text{else}}\ (e, v') \in t_D$. ■

Man beachte, daß für die **Wohldefiniertheit** der obigen Konstruktion wichtig ist, daß die Wahl eines Vertreters x aus einer Äquivalenzklasse [x] keinen Einfluß auf die Konstruktionsschritte (6) bis (9) hat (das läßt sich mit Def. 3.2.4 einfach nachrechnen).
Im folgenden werden wir oft den Spezialfall betrachten, daß b ein Monomorphismus und damit o.B.d.A. der identische Morphismus ist, sowie daß alle Bezeichner in B, die nicht in K verwendet werden, auch in D nicht auftreten. Dann gilt für die Konstruktion von G:

$\forall\, x \in \mathcal{V}_D$ (bzw. $\mathcal{E}_D$): [x] kann wieder x genannt werden

und $\forall\, x \in \mathcal{V}_B$ (bzw. $\mathcal{E}_B$):
$[x] := \underline{\text{if}}\ x \in b(\mathcal{V}_K) = \mathcal{V}_K$ (bzw. $\mathcal{E}_K$) $\underline{\text{then}}\ d(x)\ \underline{\text{else}}\ x$.

Bsp. 3.2.10 Vereinigung zweier Hypergraphen

Die Abb. 3.5 enthält ein Beispiel für die Vereinigung zweier Hypergraphen D und B bzgl. zweier Morphismen $b: K \xrightarrow{\sim} B$ und $d: K \xrightarrow{\sim} D$. Dabei ist d kein Monomorphismus. Im einzelnen gilt für die beteiligten Morphismen b, d, b’ und d’:

(1) Hyperkanten bzw. Knoten mit gleichen Bezeichnern werden aufeinander abgebildet.

(2) Eine Hyperkante bzw. ein Knoten mit Bezeichner x|y ist das Abbild zweier Hyperkanten bzw. Knoten x und y (oder Abbild einer Hyperkante bzw. eines Knotens x|y)†. ■

† Man beachte: “x”, “y” und “x|y” sind keine Markierungen, sondern Bezeichner von Knoten und Kanten.

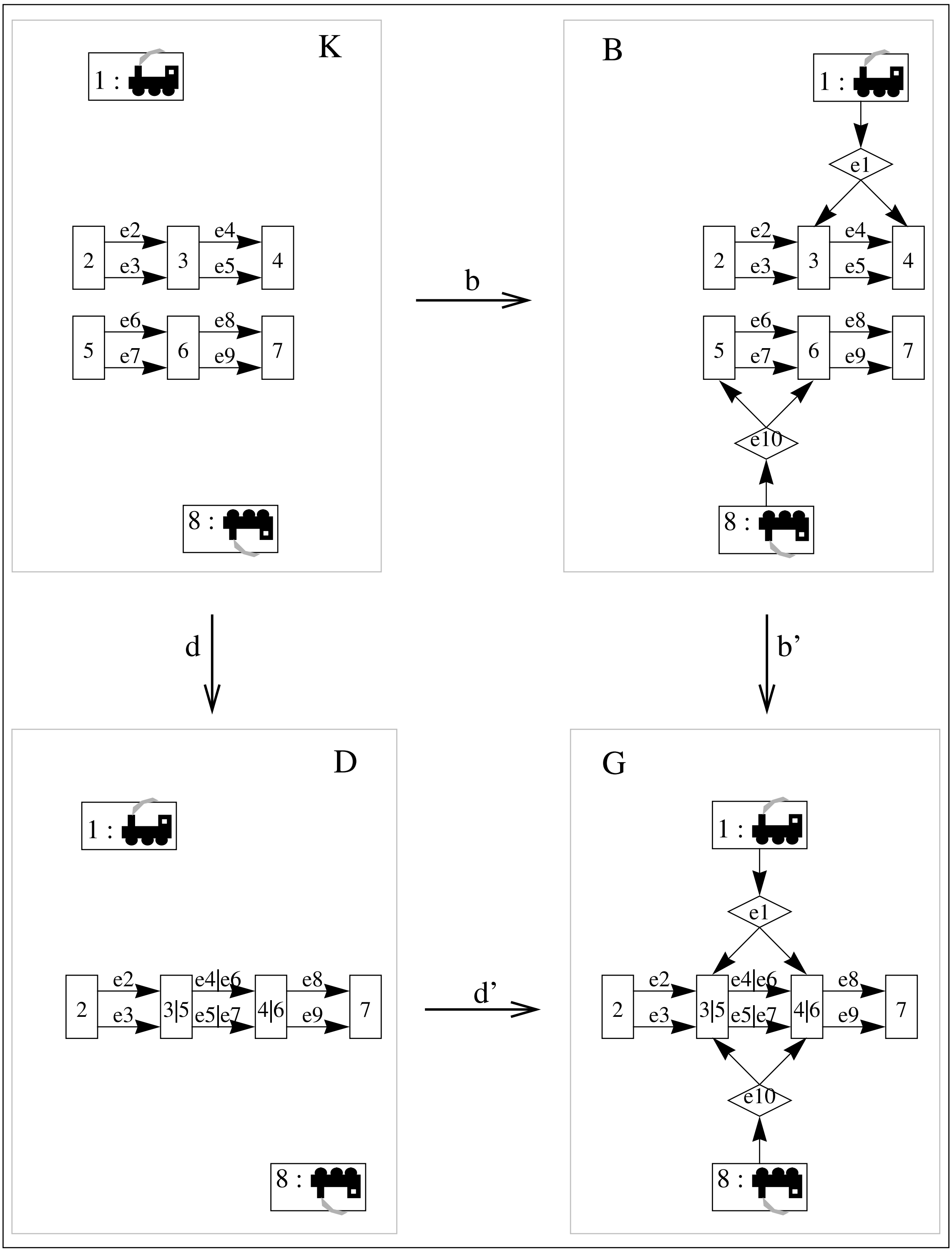


Abb. 3.5: Vereinigung zweier Hypergraphen

Nachdem wir die Vereinigung von Hypergraphen konstruktiv definiert und an einem Beispiel demonstriert haben, kommen wir nun zu ihrer kategorientheoretischen Definition. Als Vorbereitung hierfür müssen wir zunächst den Begriff des “**Pushouts**” einführen.

Def. 3.2.11 Pushout von Graphmorphismen

Sei d: K $\stackrel{\sim}{\rightarrow}$ D und b: K $\stackrel{\sim}{\rightarrow}$ B. Dann wird das in Abb. 3.6 a) gezeigte Diagramm ein Pushout-Diagramm (und G sein Pushout) genannt ⇔

(1) d’: D $\stackrel{\sim}{\rightarrow}$ G .

(2) b’: B $\stackrel{\sim}{\rightarrow}$ G .

(3) d $\circ$ d’ = b $\circ$ b’ .

(4) Für alle d”: D $\stackrel{\sim}{\rightarrow}$ H und b”: B $\stackrel{\sim}{\rightarrow}$ H
mit d $\circ$ d” = b $\circ$ b” gibt es **genau ein**
h : G $\stackrel{\sim}{\rightarrow}$ H
mit d” = d’ $\circ$ h und b” = b’ $\circ$ h . ■

$$
\begin{tikzcd}
K \arrow[r, "b"] \arrow[d, "d"'] \arrow[dr, phantom, "\mathbf{PO}"] & B \arrow[d, "b'"] \\
D \arrow[r, "d'"] & G
\end{tikzcd}
$$

a) Punkt (3) der Pushout-Definition

$$
\begin{tikzcd}
K \arrow[r, "b"] \arrow[d, "d"'] \arrow[dr, phantom, "\mathbf{PO}"] & B \arrow[d, "b'"] \arrow[ddr, "b''"] & \\
D \arrow[r, "d'"] \arrow[drr, "d''"'] & G \arrow[dr, "h"] & \\
 & & H
\end{tikzcd}
$$

b) Punkt (4) der Pushout-Definition

Abb. 3.6: Pushout-Diagramme

Ein Pushout-Diagramm ist also die Vervollständigung zweier Graphmorphismen zu einem kommutierenden Diagramm (siehe auch Abb. 3.6); dabei wird der neu entstehende Graph G so gewählt, daß er

(1) für jeden Knoten und jede Kante aus B bzw. D ein Abbild enthält,

(2) keine “überflüssigen” Knoten und Kanten besitzt und

(3) verschiedene Knoten und Kanten aus B und D nur dann miteinander identifiziert, wenn sie ein gemeinsames Urbild in K besitzen.

Auf einen Nachweis dieser Eigenschaften wollen wir hier verzichten. Ebenso werden wir die im nächsten Satz festgehaltenen weiteren Eigenschaften von Pushout-Diagrammen hier nicht beweisen:

Satz 3.2.12 Eigenschaften von Pushout-Diagrammen

(1) Der Pushout zweier Morphismen ist bis auf Isomorphie eindeutig bestimmt.

(2) Die horizontale bzw. vertikale Komposition zweier Pushout-Diagramme ergibt wiederum ein Pushout-Diagramm (siehe Abb. 3.7).

Beweis:

Kann in /RB 88/ bzw. in /Pa 69/ nachgelesen werden. ■

$$\begin{tikzcd}
K \arrow[r] \arrow[d] \arrow[rr, bend left] \arrow[dr, phantom, "\textbf{PO}"] & B \arrow[r] \arrow[d] \arrow[dr, phantom, "\textbf{PO}"] & B' \arrow[d] \\
D \arrow[r] \arrow[rr, bend right] & G \arrow[r] & G'
\end{tikzcd}$$

a) Horizontale Komposition

$$\begin{tikzcd}
K \arrow[r] \arrow[d] \arrow[dd, bend right] \arrow[dr, phantom, "\textbf{PO}"] & B \arrow[d] \arrow[dd, bend left] \\
D \arrow[r] \arrow[d] \arrow[dr, phantom, "\textbf{PO}"] & G \arrow[d] \\
D' \arrow[r] & G'
\end{tikzcd}$$

b) Vertikale Komposition

Abb. 3.7: Komposition von Pushout-Diagrammen

Satz 3.2.13 Kategorientheoretische Vereinigung von Hypergraphen

Seien B, D, K $\in \mathcal{G}_H(\mathcal{L}_V, \mathcal{L}_E)$ und b: K $\xrightarrow{\sim}$ B sowie d: K $\xrightarrow{\sim}$ D gegeben. Dann gilt für jedes G $\in \mathcal{G}_H(\mathcal{L}_V, \mathcal{L}_E)$:

G = B ${}_b\cup_d$ D (ein solches G läßt sich zu beliebigem b und d konstruieren)

$\Leftrightarrow$

Es gibt d': D $\xrightarrow{\sim}$ G und b': B $\xrightarrow{\sim}$ G, die b und d zu einem Pushout-Diagramm, wie in Abb. 3.6 a) gezeigt, vervollständigen (dadurch ist G bis auf Isomorphie eindeutig festgelegt).

Die Morphismen d' und b' sehen für $G := B \;{}_b{\cup}_d\; D$ i.a. wie folgt aus:

(1) $\forall x \in \mathcal{V}_D$ (bzw. $\mathcal{E}_D$) : $d'(x) := [x]$.

(2) $\forall x \in \mathcal{V}_B$ (bzw. $\mathcal{E}_B$) : $b'(x) := [x]$.

Beweis:

Siehe /Eh 79/ bzw. /EKL 90/. ■

Für den bereits zu Def. 3.2.9 angesprochenen Spezialfall, daß b ein Monomorphismus ist, werden wir weiterhin o.B.d.A. davon ausgehen, daß b der identische Morphismus ist und die Bezeichner in B und D nicht zueinander in Konflikt stehen:

(1) $\forall x \in \mathcal{V}_D$ (bzw. $\mathcal{E}_D$) : $d'(x) := x$.

(2) $\forall x \in \mathcal{V}_B$ (bzw. $\mathcal{E}_B$) :
$b'(x) :=$ $\underline{\text{if}}$ $x \in b(\mathcal{V}_K) = \mathcal{V}_K$ (bzw. $\mathcal{E}_K$)
$\underline{\text{then}}$ $d(x)$
$\underline{\text{else}}$ x .

Nach der Vereinigung von Hypergraphen werden wir nun die **Differenzbildung** so einführen, daß sie ein Gegenstück zur Vereinigung bildet, also daß mit geeignet gewählten Morphismen $b\colon K \overset{\sim}{\rightarrow} B$, $d\colon K \overset{\sim}{\rightarrow} D$ und $b'\colon B \overset{\sim}{\rightarrow} G$ gilt:

$$D = G \;{}_{b'}{\setminus}_b\; B \quad \Leftrightarrow \quad G = B \;{}_b{\cup}_d\; D .$$

Hierzu muß die Differenzbildung von $G \;{}_{b'}{\setminus}_b\; B$ ungefähr wie folgt definiert werden:

(1) G enthält b'(B), ein homomorphes Abbild von B.

(2) B enthält b(K), ein isomorphes Abbild von K[†].

(3) Aus G wird das Abbild von B ausschließlich des enthaltenen Abbildes von K entfernt.

Die im folgenden eingeführte Definition ist nur eine spezielle Abart der allgemeinen Differenzbildung, bei der B u.a. nicht vollständig in G "enthalten" sein müßte (via Homomorphismus). Zudem besteht der Hypergraph B ja nicht nur aus zu entfernenden Teilen, sondern enthält weitere Knoten und Hyperkanten, die festlegen, in was für einem Kontext die zu entfernenden Teile stehen[‡]. Das soll zunächst an einem Beispiel verdeutlicht werden:

Bsp. 3.2.14 Differenzbildung zweier Hypergraphen

In der Abb. 3.5 gilt: $D := G \;{}_{b'}{\setminus}_b\; B$. ■

† Die Verallgemeinerung auf ein homomorphes Abbild ist denkbar, würde aber alle nachfolgenden Definitionen und Sätze unnötig komplizieren.

‡ Das sind genau die durch K festgelegten Teile in B; "K" steht also für Kontext.

Def. 3.2.15 Operationale Differenzbildung von Hypergraphen

Seien G, B, K $\in \mathcal{G}_H(\mathcal{L}_V, \mathcal{L}_E)$ und es gebe b: K $\tilde{\to}$ B sowie b': B $\tilde{\to}$ G. Dann ist ein Hypergraph D die Differenz von B aus G bzgl. b und b' (in Zeichen: D := G $_{b'}\backslash_b$ B) $\Leftrightarrow$

(1) $D = (\mathcal{V}, \mathcal{E}, l_V, l_E, s, t)$.

(2) $\mathcal{V} := \mathcal{V}_G \setminus b'(Del\_V)$ mit $Del\_V := \mathcal{V}_B \setminus b(\mathcal{V}_K)$.

(3) $\mathcal{E} := \mathcal{E}_G \setminus b'(Del\_E)$ mit $Del\_E := \mathcal{E}_B \setminus b(\mathcal{E}_K)$.

(4) $l_V := l_V(G) \mid_{\mathcal{V}}$.

(5) $l_E := l_E(G) \mid_{\mathcal{E}}$.

(6) $s := s_G \mid_{\mathcal{E} \times \mathcal{V}}$.

(7) $t := t_G \mid_{\mathcal{E} \times \mathcal{V}}$.

(8) $\forall\, e \in \mathcal{E}_G,\, v \in b'(Del\_V)$: $(e, v) \in s_G \vee (e, v) \in t_G \Leftrightarrow e \in b'(Del\_E)$.

(9) $\forall\, v, v' \in \mathcal{V}_B :\; b'(v) = b'(v') \Rightarrow (v, v' \notin Del\_V \vee v = v')$.

(10) $\forall\, e, e' \in \mathcal{E}_B :\; b'(e) = b'(e') \Rightarrow (e, e' \notin Del\_E \vee e = e')$.

(11) b ist ein Monomorphismus. ■

Die obige Definition enthält neben dem eigentlichen Konstruktionverfahren in den Schritten (1) bis (7) vier zusätzliche Vorbedingungen für die Differenzbildung. Die erste Vorbedingung - die mit der Nr. (8), die sogenannte "**dangling edge condition**", verbietet das Löschen von Knoten, deren sie berührende Hyperkanten nicht mitgelöscht werden. Die Bedingungen (9) und (10), die sogenannten "**identification conditions**", fordern darüber hinaus, daß verschiedene Elemente in dem Hypergraphen B (Knoten oder Hyperkanten) nur dann auf dasselbe Element in G abgebildet werden, wenn letzteres nicht gelöscht wird. Alle drei Bedingungen zusammen stellen erst sicher, daß gilt:

$$D \cong G\, {}_{b'}\backslash_b\, B \quad \Rightarrow \quad G \cong B\, {}_b\cup_d\, D \quad \text{mit geeignetem } d\text{: } K \tilde{\to} D .$$

Die vierte Bedingung - die mit der Nr. (11) - stellt schließlich sicher, daß D bis auf Isomorphie der einzige Hypergraph ist (und damit die obige Konstruktion die einzig sinnvolle Art, die Differenzbildung konstruktiv zu definieren), für den gilt:

$$G \cong B\, {}_b\cup_d\, D \quad \text{mit geeignetem } d\text{: } K \tilde{\to} D.$$

Die Auswirkungen des Verletztseins dieser Bedingungen lassen sich sehr schön an einer leicht veränderten Fassung des Beispiels aus Abb. 3.5 darstellen.

Bsp. 3.2.16 Verbotene Differenzbildung

Es seien K, B, D, G $\in \mathcal{G}_H(\mathcal{L}_V, \mathcal{L}_E)$ die in Abb. 3.8 dargestellten Hypergraphen, wobei D gemäß der Schritte (1) bis (7) in Def. 3.2.15 aus K, B und G gebildet wurde. Wie man sieht,

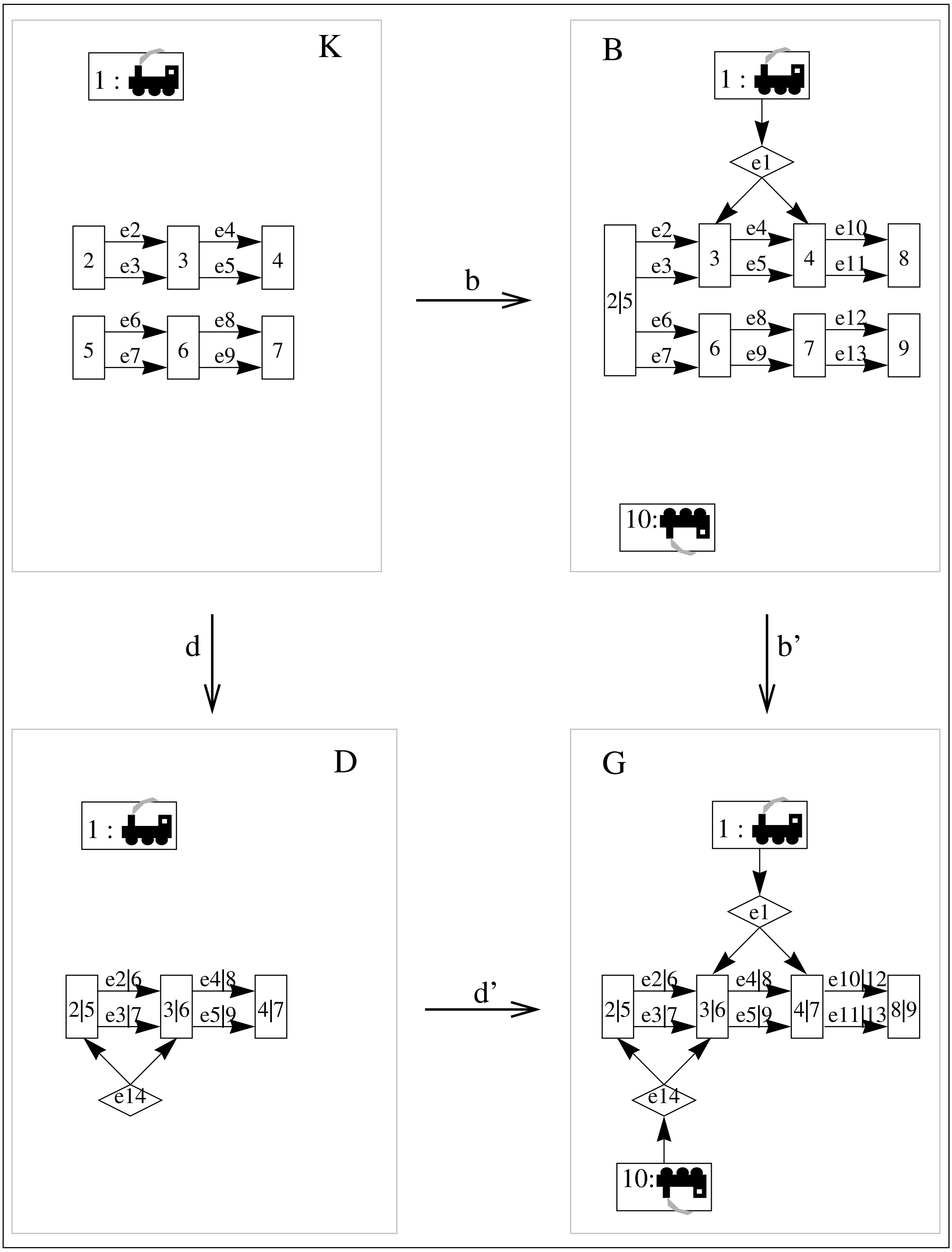


Abb. 3.8: Verbotene Differenzbildung

führt das Verletztsein der “dangling edge condition” am Knoten 10 in B und der “identification condition” an den Knoten 8 und 9 sowie den Hyperkanten e10 bis e13 in B dazu, daß mit $G' \in \mathcal{G}_H(\mathcal{L}_V, \mathcal{L}_E)$ aus Abb. 3.9 gilt:

$$G \ncong G' \cong B\ {}_b\!\cup_d D \cong B\ {}_b\!\cup_d (G\ {}_{b'}\!\backslash_b B)\ .$$

Ersetzt man in Abb. 3.8 den Hypergraphen G durch G’, so wären dann zwar die “dangling edge condition” und die “identification condition” erfüllt, aber $b: K \stackrel{\sim}{\rightarrow} B$ wäre immer noch kein Monomorphismus. Deshalb gibt es ein $H \in \mathcal{G}_H(\mathcal{L}_V, \mathcal{L}_E)$ (siehe Abb. 3.9), so daß mit geeignet gewähltem $h: K \stackrel{\sim}{\rightarrow} H$ gilt:

$$D \ncong H \quad \text{und} \quad G' \cong B\ {}_b\!\cup_h H \cong B\ {}_b\!\cup_d D\ . \blacksquare$$

Satz 3.2.17 Kategorientheoretische Differenzbildung

Seien $G, B, K \in \mathcal{G}_H(\mathcal{L}_V, \mathcal{L}_E)$ und es gebe einen Monomorphismus $b: K \stackrel{\sim}{\rightarrow} B$ sowie einen Morphismus $b': B \stackrel{\sim}{\rightarrow} G$. Dann gilt für jeden Hypergraphen $D \in \mathcal{G}_H(\mathcal{L}_V, \mathcal{L}_E)$:

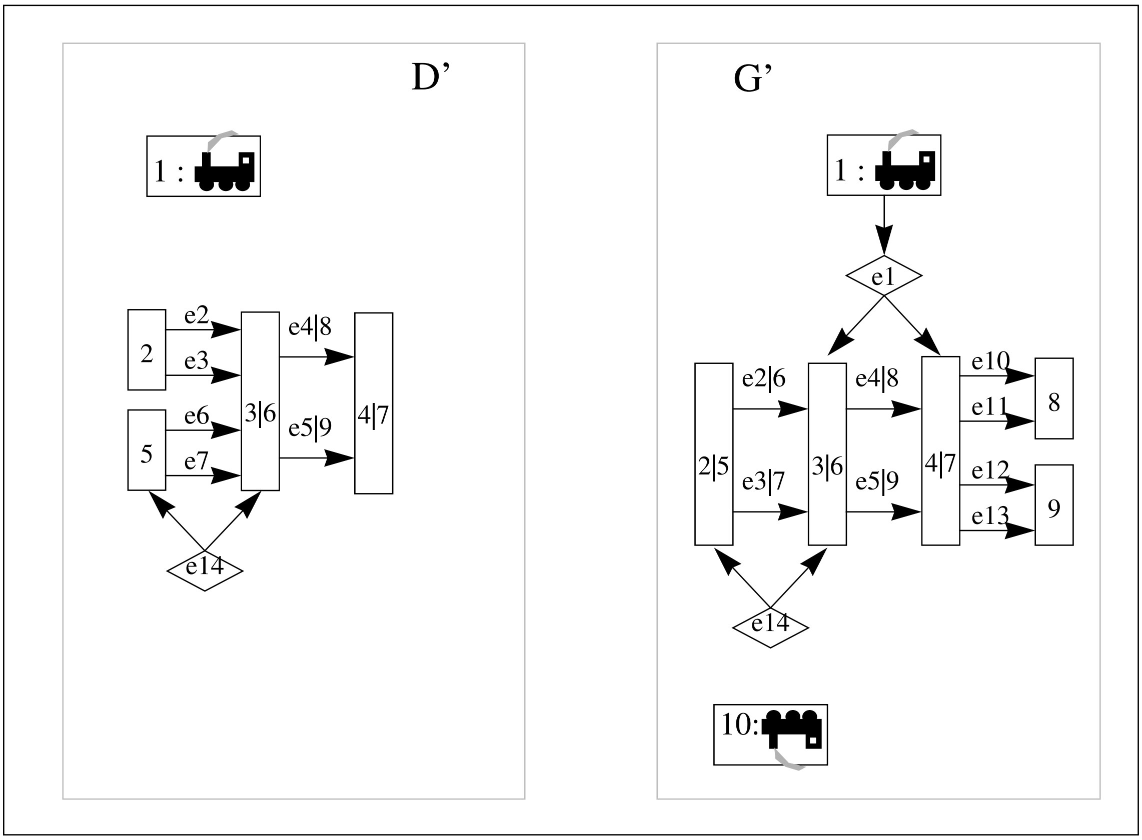


Abb. 3.9: Konsequenzen verbotener Differenzbildung

$D \cong G\ {}_{b'}\backslash_{b}\ B$
(damit sind insbesondere die Bedingungen (8) bis (11) aus Def. 3.2.15 erfüllt)

$\Leftrightarrow$

Es gibt (bis auf Isomorphie genau) ein d: $K \overset{\sim}{\rightarrow} D$ und ein d': $D \overset{\sim}{\rightarrow} G$, die b und b' zu einem Pushout-Diagramm, wie in Abb. 3.6 gezeigt, vervollständigen (D wird das Pushout-Komplement des Diagramms genannt).
Die Morphismen d und d' sehen für $D := G\ {}_{b'}\backslash_{b}\ B$ wie folgt aus:

(1) $d := b \circ b'$.

(2) $d' : D \overset{\sim}{\rightarrow} G := id(D, G)$[†].

Beweis:

Siehe /Eh 79/ bzw. /EKL 90/. ■

Korollar 3.2.18 Differenzbildung und Vereinigung

Sei gegeben b: $K \overset{\sim}{\rightarrow} B$, d: $K \overset{\sim}{\rightarrow} D$, b': $B \overset{\sim}{\rightarrow} G$ und d': $D \overset{\sim}{\rightarrow} G$ mit:

(1) b ist injektiv und damit gilt o.B.d.A.: b := id(K, B) .

(2) d' := id(D, G) .

(3) $\forall\ x \in \mathcal{V}_B$ (bzw. $\mathcal{E}_B$):
b'(x) = <u>if</u> $x \in b(\mathcal{V}_K) = \mathcal{V}_K$ (bzw. $\mathcal{E}_K$) <u>then</u> d(x) <u>else</u> x .

Dann gilt:
$G \cong B\ {}_{b}\cup_{d}\ D \Leftrightarrow$ b, d, b', d' bilden ein Pushout-Diagramm $\Leftrightarrow D \cong G\ {}_{b'}\backslash_{b}\ B$.

Beweis:

(1) Erste Äquivalenz $\Rightarrow$:
Da b = id(K, B) ist, gilt der Spezialfall aus den Anmerkungen zu Def. 3.2.9 bzw. zu Satz 3.2.13 bezüglich der Konstruktion von d' und b'. Damit bilden d' und b' gemäß Satz 3.2.13 zusammen mit b und d ein Pushout-Diagramm.

(2) Erste Äquivalenz $\Leftarrow$:
Folgt direkt aus Satz 3.2.13.

(3) Zweite Äquivalenz $\Rightarrow$:
Folgt direkt aus Satz 3.2.17.

(4) Zweite Äquivalenz $\Leftarrow$:
Mit $d = b \circ b'$ und d' = id(D, G) und Satz 3.2.17 folgt auch diese Richtung. ■

---

† “id(X,Y)” sei die identische Abbildung von X nach Y.

Die Bedingung (1) aus obigem Korollar ist eine notwendige und hinreichende Bedingung dafür, daß es bis auf Isomorphie nur **genau eine mögliche Vervollständigung** von b und b' zu einem Pushout-Diagramm gibt. Mit Hilfe der beiden anderen Bedingungen können wir sicherstellen, daß b, d, b' und d' genau die Mophismen sind (und nicht nur bis auf Isomorphie gleiche Morphismen), die im Zuge der operationalen Definition der Vereinigung und Differenzbildung benutzt werden.

3.3 Sequentielle Graphersetzungssysteme

Im vorigen Abschnitt haben wir mit der Vereinigung und der Differenzbildung die beiden Basisoperationen eingeführt, die wir für die Anwendung von (Teil-) Graphersetzungsregeln benötigen. Mit ihrer Hilfe werden wir nun einen sehr einfachen und damit theoretisch gut handhabbaren **Teilgraphersetzungsbegriff** definieren. Einfach ist er deshalb, weil wir hier - wie bereits in Abschnitt 3.1 angedeutet - auf explizite Einbettungsüberführungsregeln ganz verzichten werden. Die bereits informell vorgestellten Teilgraphersetzungsregeln mit einem sogenannten Kontext- oder Verklebungsgraph K, der den identisch zu ersetzenden Anteil in linker und rechter Regelseite durch zwei Monomorphismen festlegt[†], sieht formal wie folgt aus:

Def. 3.3.1 (Teil-) Graphersetzungsregel

Ein Tripel $r = (L, K, R) \in \mathcal{G}_H(\mathcal{L}_V, \mathcal{L}_E)^3$ wird (Teil-) Graphersetzungsregel über $\mathcal{L}_V$ und $\mathcal{L}_E$ genannt (in Zeichen: $r \in \mathcal{R}_H(\mathcal{L}_V, \mathcal{L}_E)$) $\Leftrightarrow$

(1) id: $K \xrightarrow{\sim} L$ (die identische Abbildung ist ein Morphismus von K nach L).

(2) id: $K \xrightarrow{\sim} R$ (die identische Abbildung ist ein Morphismus von K nach R). ■

Bsp. 3.3.2 Die Graphersetzungsregel "MoveTrain"

Seien L, K und R wie in Abb. 3.10 definiert. Dann ist $r = (L, K, R)$ eine Graphersetzungsregel über $\mathcal{L}_V$ und $\mathcal{L}_E$, wie sie in Bsp. 3.2.3 benutzt wurden. ■

Def. 3.3.3 (Teil-) Graphersetzungssystem

Seien $\mathcal{L}_V$ und $\mathcal{L}_E$ unsere Markierungsalphabete. Dann wird ein Quadrupel der Form $gr = (\mathcal{L}_V, \mathcal{L}_E, G_S, \mathcal{R})$ ein (Teil-) Graphersetzungssystem genannt $\Leftrightarrow$

(1) $G_S \in \mathcal{G}_H(\mathcal{L}_V, \mathcal{L}_E)$ (G_S ist ein beliebiger Startgraph).

(2) $\mathcal{R} \subseteq \mathcal{R}_H(\mathcal{L}_V, \mathcal{L}_E)$ ($\mathcal{R}$ ist eine Menge von Teilgraphersetzungsregeln). ■

† O.B.d.A. verwenden wir beide Male die identische Abbildung.

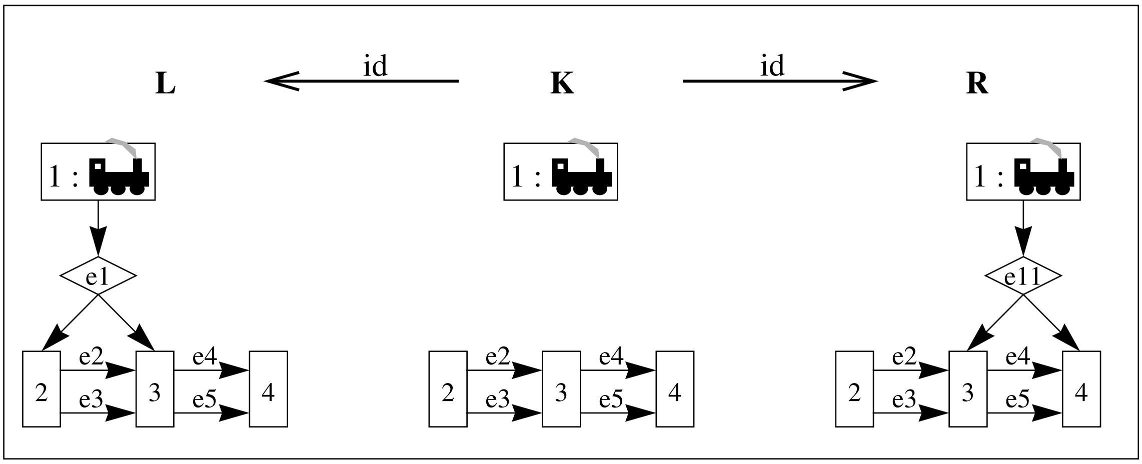


Abb. 3.10: Beispiel einer Teilgraphersetzungsregel

Nach der Definition der Bestandteile eines Graphersetzungssystems kommen wir nun zunächst zur kategorientheoretischen und danach zur operationalen Definition der Ausführung einer Graphersetzungsregel.

Def. 3.3.4 Kategorientheoretische Def. sequentieller Graphersetzungen

Es sei r = (L, K, R) $\in \mathcal{R}_H(\mathcal{L}_V, \mathcal{L}_E)$ und G, H $\in \mathcal{G}_H(\mathcal{L}_V, \mathcal{L}_E)$. Dann läßt sich H durch die Anwendung von r auf G ableiten (in Zeichen: G ~ r ~> H) $\Leftrightarrow$

Es läßt sich ein Diagramm bestehend aus zwei Pushout-Teildiagrammen konstruieren, wie es in Abb. 3.11 dargestellt wird. ■

$$\begin{tikzcd} L \arrow[d, "g"'] & K \arrow[l, "id"'] \arrow[r, "id"] \arrow[d, "d"] \arrow[dl, phantom, "\textbf{PO 1}"] \arrow[dr, phantom, "\textbf{PO 2}"] & R \arrow[d, "h"] \\ G & D \arrow[l, "id"'] \arrow[r, "id"] & H \end{tikzcd}$$

Abb. 3.11: Pushout-Definition einer Graphersetzung

Satz 3.3.5 Operationale Definition sequentieller Graphersetzungen

Seien r, G und H wie in Def. 3.3.4 vorgegeben und o.B.d.A. seien die Bezeichner von Knoten und Hyperkanten in R disjunkt zu denen in G. Dann gilt:

G ~ r ~> H

$\Leftrightarrow$

(1) Es gibt $g: L \xrightarrow{\sim} G$, das eine Anwendungsstelle von r in G festhält.

(2) $id: K \xrightarrow{\sim} L$ und $g: L \xrightarrow{\sim} G$ erfüllen zusammen die “dangling edge”- und die “identification”-Bedingung aus Def. 3.2.15 (mit B := L, b := id(K, L) und b’ := g).

(3) $D = G\ {}_g\backslash_{id}\ L$ (D ist das Pushout-Komplement in PO 1).

(4) $d = g\,|_K$.

(5) $H' = D\ {}_d\cup_{id}\ R$ (H’ ist der Pushout in PO 2).

(6) $h'(x) = \underline{\text{if}}\ x \in \mathcal{V}_K \subseteq \mathcal{V}_R$ (bzw. $\mathcal{E}_K \subseteq \mathcal{E}_R$) $\underline{\text{then}}\ d(x)\ \underline{\text{else}}\ x$.

(7) $H \cong H'$ und mit $i: H' \xrightarrow{\approx} H$ gilt: $h = h' \circ i$.

Beweis:

Die Behauptung folgt direkt aus der zweimaligen Anwendung von Korollar 3.2.18. Dabei setze man im ersten Schritt im Teildiagramm PO 1 aus Abb. 3.11

B := L, b := id(K, L), b’ := g, d’ := id(D, G)

und im zweiten Schritt im Teildiagramm PO 2 aus Abb. 3.11

B := R, b := id(K, R), b’ := h, d’ := id(D, H) . ■

Bsp. 3.3.6 Anwendung der Graphersetzungsregel “MoveTrain”

Sei MoveTrain := (L, K, R) die Graphersetzungsregel aus Bsp. 3.3.2. Dann gilt mit G und H wie in Abb. 3.12 gezeigt:

G ~MoveTrain~> H. ■

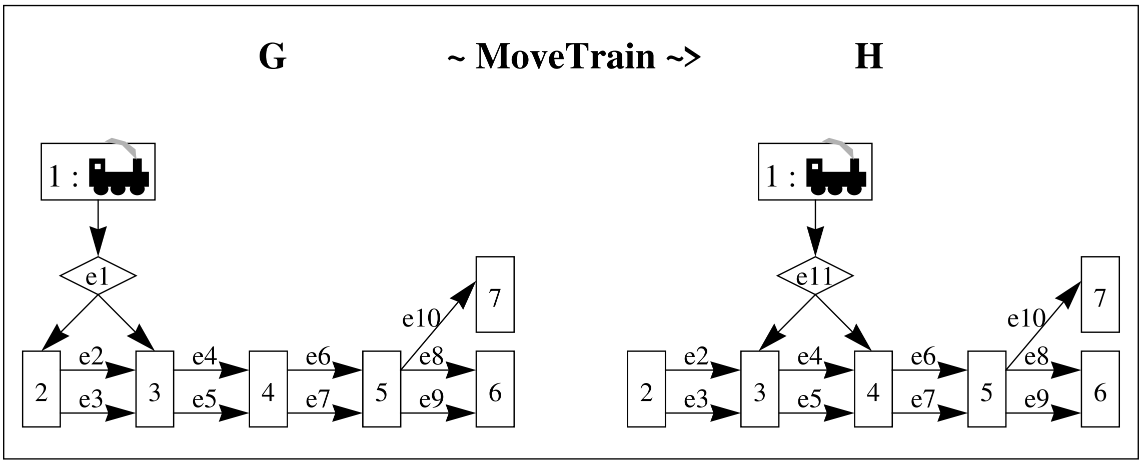


Abb. 3.12: Anwendung einer Teilgraphersetzung

Def. 3.3.7 Sequentielle Ableitbarkeit

Es sei $\mathcal{R} \subseteq \mathcal{R}_H(\mathcal{L}_V, \mathcal{L}_E)$ sowie G, H $\in \mathcal{G}_H(\mathcal{L}_V, \mathcal{L}_E)$. Dann heißt H vermittels $\mathcal{R}$ sequentiell ableitbar aus G (in Zeichen: G ~ $\mathcal{R}$~> H) $\Leftrightarrow$

$$\exists\, r \in \mathcal{R}:\ G \sim r \sim> H\,.$$

Der transitive, reflexive Abschluß von ~> wird mit $\overset{*}{\sim>}$ bezeichnet. ■

Def. 3.3.8 Sequentielle Graphsprache

Sei gr = $(\mathcal{L}_V, \mathcal{L}_E, G_S, \mathcal{R})$ ein Graphersetzungssystem. Dann ist die von gr erzeugte Sprache folgendermaßen definiert:

$$\mathcal{L}_H(gr) := \{\, G \in \mathcal{G}_H(\mathcal{L}_V, \mathcal{L}_E) \mid G_S \overset{*}{\sim>} G \,\}\,.$$ ■

Damit haben wir die einfachste Variante einer kategorientheoretischen Definition von (sequentiellen) Graphersetzungen vorgestellt. Im Vergleich zu dem mengentheoretisch fundierten Graphersetzungsbegriff im Kapitel 2 stechen einem vor allem folgende Punkte ins Auge:

(1) Für die Einbettung des Abbilds der rechten Seite einer Regel in einen Wirtsgraphen gibt es keine expliziten Regeln; sie wird vielmehr durch die identische Ersetzung von Knoten und Hyperkanten indirekt erreicht.

(2) Mit den damit zur Verfügung stehenden Hilfsmitteln ist es nicht möglich, ganze Kantenbüschel unbekannter Kardinalität im Zuge eines Ersetzungsschrittes zu löschen, zu kopieren oder zu erzeugen.

(3) Die Anwendungsstelle einer Graphersetzungsregel ist i.a. kein isomorphes Abbild der linken Regelseite. Damit können mehrere Knoten (Hyperkanten) einer linken Seite mit einem Knoten (Hyperkante) im Wirtsgraphen identifiziert werden. Das erhöht die Ausdruckskraft des Ansatzes etwas, kann aber zu völlig kontraintuitiven Ersetzungsergebnissen führen.

(4) Die “dangling edge”-Bedingung ist - im Gegensatz zur “identification”- Bedingung - ein schwerwiegendes Handikap beim praktischen Einsatz des vorgestellten Graphersetzungsbegriffs: Knoten, deren 1-Kontext nicht vollständig in der linken Seite einer Graphersetzungsregel aufgeführt sind, können durch diese nicht gelöscht werden†.

(5) Andererseits garantieren die “dangling edge”- und die “identification”-Bedingung, daß es zu jeder Graphersetzungsregel eine inverse Graphersetzungsregel gibt, die sich ganz einfach konstruieren läßt.

---

† Das gilt nicht mehr für den in /Lö 91/ beschriebenen algebraischen Ansatz, der auf einer gänzlich anderen Pushout-Konstruktion basiert.

Satz 3.3.9 Inversenbildung

Sei r := (L, K, R) ∈ $\mathcal{R}_H(\mathcal{L}_V, \mathcal{L}_E)$. Dann ist r^{-1} := (R, K, L) ∈ $\mathcal{R}_H(\mathcal{L}_V, \mathcal{L}_E)$ und es gilt:

$$G \sim r \rightsquigarrow H \quad \Leftrightarrow \quad H \sim r^{-1} \rightsquigarrow G .$$

Beweis:

Sowohl die kategorientheoretische Definition einer Graphersetzungsregel als auch die Definition ihrer Anwendung sind bezüglich linker und rechter Seite (bzw. Argument und Ergebnis ihrer Anwendung) völlig symmetrisch. Also liefert das Vertauschen von linker und rechter Seite die inverse Graphersetzungsregel. ■

3.4 Komposition von Graphersetzungsregeln

Nach der Einführung der sequentiellen Teilgraphersetzungssysteme wenden wir uns nun einigen (hoffentlich) spannenden Problemen bezüglich der **Komposition von Graphersetzungsregeln** zu, die im Rahmen des kategorientheoretischen Ansatzes überraschend einfache Lösungen besitzen (wir werden im folgenden deshalb immer die kategorientheoretischen Definitionen und nicht ihre operationalen Alternativen benutzen):

(1) Bringt die Erzeugung von Graphersetzungsregeln durch Graphersetzungsregeln einen Zugewinn an Ausdruckskraft?

(2) Unter welchen Voraussetzungen kann man Graphersetzungsregeln gleichzeitig auf einen Hypergraphen anwenden?

(3) Erlaubt die Verschmelzung zweier Graphersetzungsregeln zu einer neuen Regel die Durchführung von Graphersetzungen, die mit der sequentiellen Anwendung der einzelnen Regeln nicht möglich sind?

(4) In welchem Zusammenhang stehen die parallele Anwendung von Graphersetzungsregeln und die Verschmelzung von Graphersetzungsregeln?

Beginnen wir mit dem einfachsten Punkt der obigen Liste, der Erzeugung von Teilgraphersetzungsregeln durch Teilgraphersetzungsregeln.

Def. 3.4.1 Erzeugung von Graphersetzungsregeln

Sei r := (L, K, R) ∈ $\mathcal{R}_H(\mathcal{L}_V, \mathcal{L}_E)$ und G, D, H ∈ $\mathcal{G}_H(\mathcal{L}_V, \mathcal{L}_E)$, so daß $G \sim r \rightsquigarrow H$ gilt und D der dabei konstruierte “Differenzgraph” ist (siehe Abb. 3.13). Dann gilt:

(1) r’ := (G, D, H) ist eine Graphersetzungsregel.

(2) Für alle G’, H’ ∈ $\mathcal{G}_H(\mathcal{L}_V, \mathcal{L}_E)$ gilt:

$$G' \sim r' \rightsquigarrow H' \quad \Rightarrow \quad G' \sim r \rightsquigarrow H' .$$

(Die Umkehrung gilt i.a. nicht, da r’ einen “weiteren” Kontext als r hat.)

$$\begin{tikzcd}
L \arrow[d] & K \arrow[l] \arrow[r] \arrow[d] & R \arrow[d] \\
G \arrow[d] & D \arrow[l] \arrow[r] \arrow[d] & H \arrow[d] \\
G' & D' \arrow[l] \arrow[r] & H'
\end{tikzcd}$$

Abb. 3.13: Erzeugung von Graphersetzungsregeln durch Graphersetzungen

Beweis:

zu (1):

Gemäß Def. 3.3.4 ist die identische Abbildung ein Morphismus von D nach G bzw. von D nach H. Damit sind alle Voraussetzungen in Def. 3.3.1 erfüllt.

zu (2):

Laut Satz 3.2.12 ist die vertikale Komposition zweier Pushout-Diagramme wiederum ein Pushout-Diagramm. Damit sind die Bedingungen von Def. 3.3.4 bzgl. der Anwendung von Teilgraphersetzungsregeln erfüllt. ■

Nimmt man also zu einer Menge von Graphersetzungsregeln $\mathcal{R}$ eine weitere Graphersetzungsregel r' hinzu, die mit Hilfe von $r \in \mathcal{R}$ erzeugt wurde, so hat das auf den Sprachschatz von $\mathcal{R}$ keinen Einfluß. Damit ist zumindest diese Art der Erzeugung von Graphersetzungsregeln durch Graphersetzungsregeln praktisch nahezu bedeutungslos.

Wenden wir uns nun einem spannenderen Thema zu, der **parallelen/gleichzeitigen Anwendung zweier Graphersetzungsregeln** r1 und r2. Hier können wir drei sukzessive allgemeiner werdende Fälle unterscheiden:

(1) Die Anwendungsstellen von r1 und r2 in einem Graphen G sind vollständig disjunkt. Dann ist es klar, daß es keine Rolle spielt, ob man zunächst r1 und dann r2 oder zunächst r2 und dann r1 oder beide Regeln gleichzeitig anwendet.

(2) Die Anwendungsstellen von r1 und r2 in einem Graphen G überschneiden sich allein in identisch zu ersetzenden Anteilen. Auch dann spielt es offensichtlich keine Rolle, ob man beide Regeln gleichzeitig oder hintereinander anwendet.

(3) Die Regeln r1 und r2 besitzen nicht nur Überschneidungen in den identisch zu ersetzenden Anteilen, sondern auch bei den zu löschenden und/oder den neu zu erzeugenden Anteilen. In diesem Fall lassen sich beide Regeln nicht mehr unabhängig voneinander ausführen, sondern müssen zuvor miteinander verschmolzen werden.

Wenden wir uns zunächst der unabhängigen Ausführung zweier konkreter Graphersetzungsregeln zu, deren Anwendungsstellen sich höchstens in den identisch zu ersetzenden Anteilen überschneiden:

Bsp. 3.4.2 Parallele Anwendung von Graphersetzungsregeln

Seien “MoveTrain1” und “MoveTrain2” die Graphersetzungsregeln aus Abb. 3.14 und “MoveTrain1 ⊕ MoveTrain2” die disjunkte Vereinigung der einzelnen Komponenten der beiden Graphersetzungsregeln zu einer neuen Graphersetzungsregel. Dann gilt mit G, G’ wie in Abb. 3.15:

(1) $\exists$ H: G ~ MoveTrain1 ~> H $\wedge$ H ~ MoveTrain2 ~> G’ .

(2) $\exists$ H’: G ~ MoveTrain2 ~> H’ $\wedge$ H’ ~ MoveTrain1 ~> G’ .

(3) G ~ MoveTrain1 ⊕ MoveTrain2 ~> G’ .

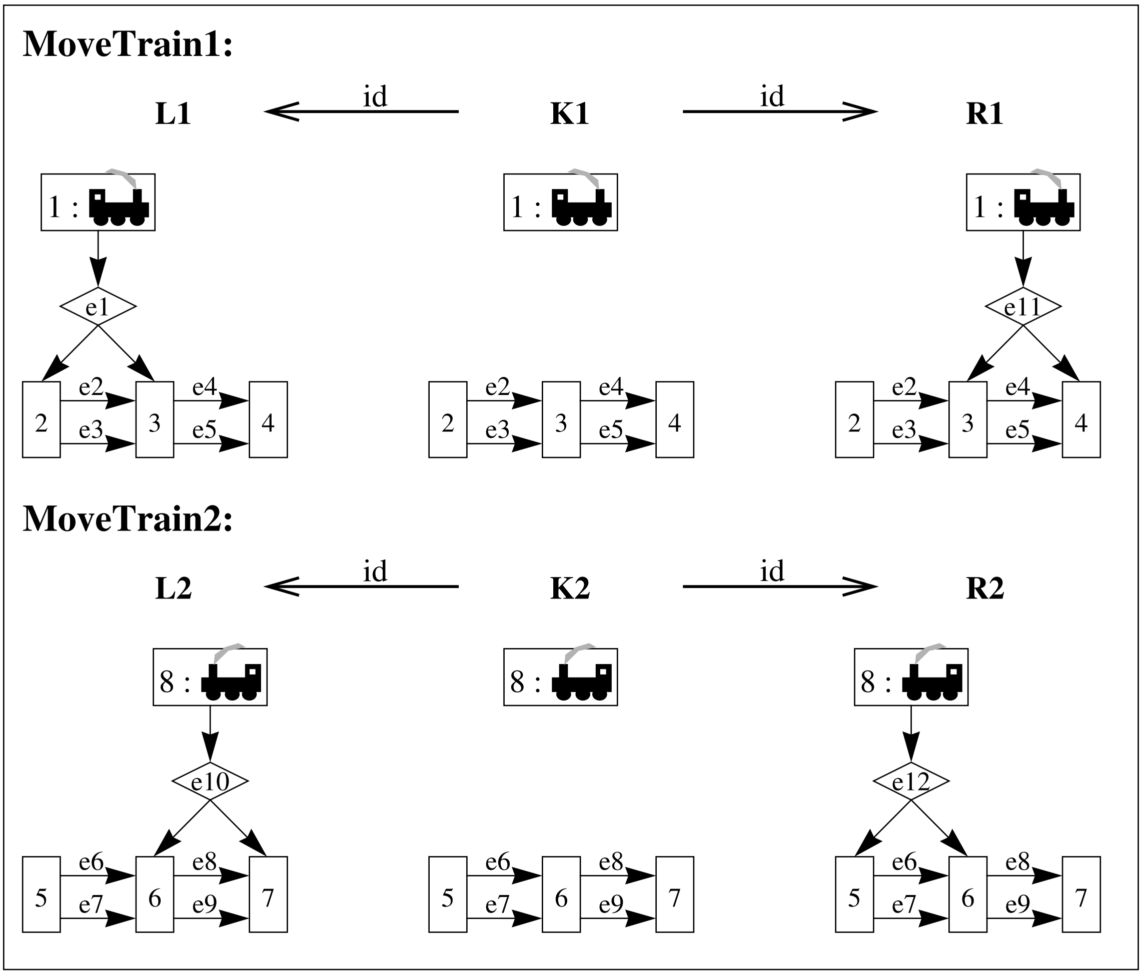


Abb. 3.14: Parallel anwendbare Ersetzungsregeln

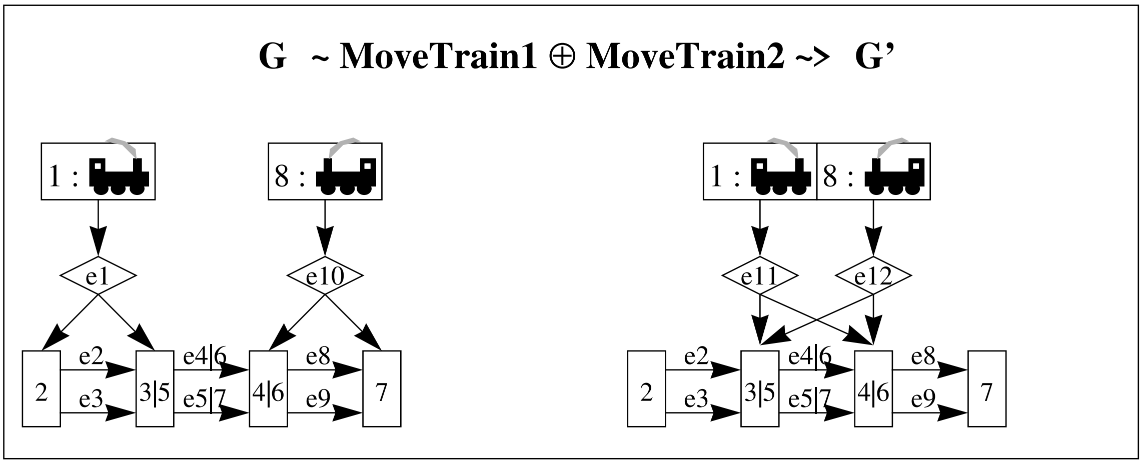


Abb. 3.15: Parallele Anwendung zweier Ersetzungsregeln

Durch die Numerierung der Knoten und Hyperkanten in Abb. 3.14 und Abb. 3.15 soll angedeutet werden, wie man sich die Zuordnung der Knoten und Hyperkanten der Ersetzungsregeln zu denen in den Graphen G und G’ vorzustellen hat. So wird etwa den Knoten 4 und 6 in der zusammengesetzten Regel “MoveTrain1 ⊕ MoveTrain2” derselbe Knoten 4|6 in G zugeordnet. ■

Das oben angegebene Anwendungsbeispiel besitzt einen entscheidenden Schönheitsfehler: Die Graphersetzungsregeln “MoveTrain1” und “MoveTrain2” verhindern nicht einmal bei ihrer sequentiellen Anwendung die **Kollision zweier Lokomotiven**. Hierfür müßte nämlich jede der beiden Regeln überprüfen, ob der folgende Streckenabschnitt tatsächlich frei ist (und bei der gleichzeitigen Anwendung beider Regeln sogar noch, ob die jeweils andere Regel versucht, denselben Streckenabschnitt zu besetzen). Da der hier vorgestellte Graphersetzungsbegriff keine (negativen) Anwendbarkeitsbedingungen kennt, bleiben uns nur die “dangling edge”- und die “identification”-Bedingung als Hilfsmittel bei der Formulierung solcher Bedingungen.

Die in Abb. 3.16 gezeigte Variante verhindert, daß die **parallele Ausführung** der beiden dort angegebenen Graphersetzungsregeln denselben Streckenabschnitt zweifach belegt. In diesem Fall würden nämlich die mit e4 und e5 bzw. e6 und e7 benannten Hyperkanten in der linken Regelseite denselben Hyperkanten in G zugeordnet werden. Das würde aber gegen die “identification”-Bedingung verstoßen. Auf diese Weise haben wir allerdings noch nicht verhindert, daß durch die Hintereinanderausführung der beiden Graphersetzungsregeln ein Zusammenstoß provoziert wird.

Letzteres kann man nur durch das zusätzliche Löschen eines bislang identisch ersetzten Knotens erreichen. Mit der in Abb. 3.17 skizzierten Veränderung unserer Grapher-

setzungsregeln machen wir uns die “dangling-edge”Bedingung zunutze. Das Löschen des mit 3 bzw. 6 bezeichneten Knotens ist nämlich nur dann möglich, wenn die linke Seite der Graphersetzungsregel “MoveTrain1*” bzw. “MoveTrain2*” alle ihn berührenden Hyperkanten im Graphen G aufführt. Wenn nun aber dem Streckenabschnitt 3-4 bzw. 5-6 bereits eine Lokomotive zugeordnet ist, so ist damit die “dangling edge”-Bedingung verletzt (leider ist sie auch dann verletzt, wenn bei 3 bzw. bei 6 eine Abzweigung beginnt).

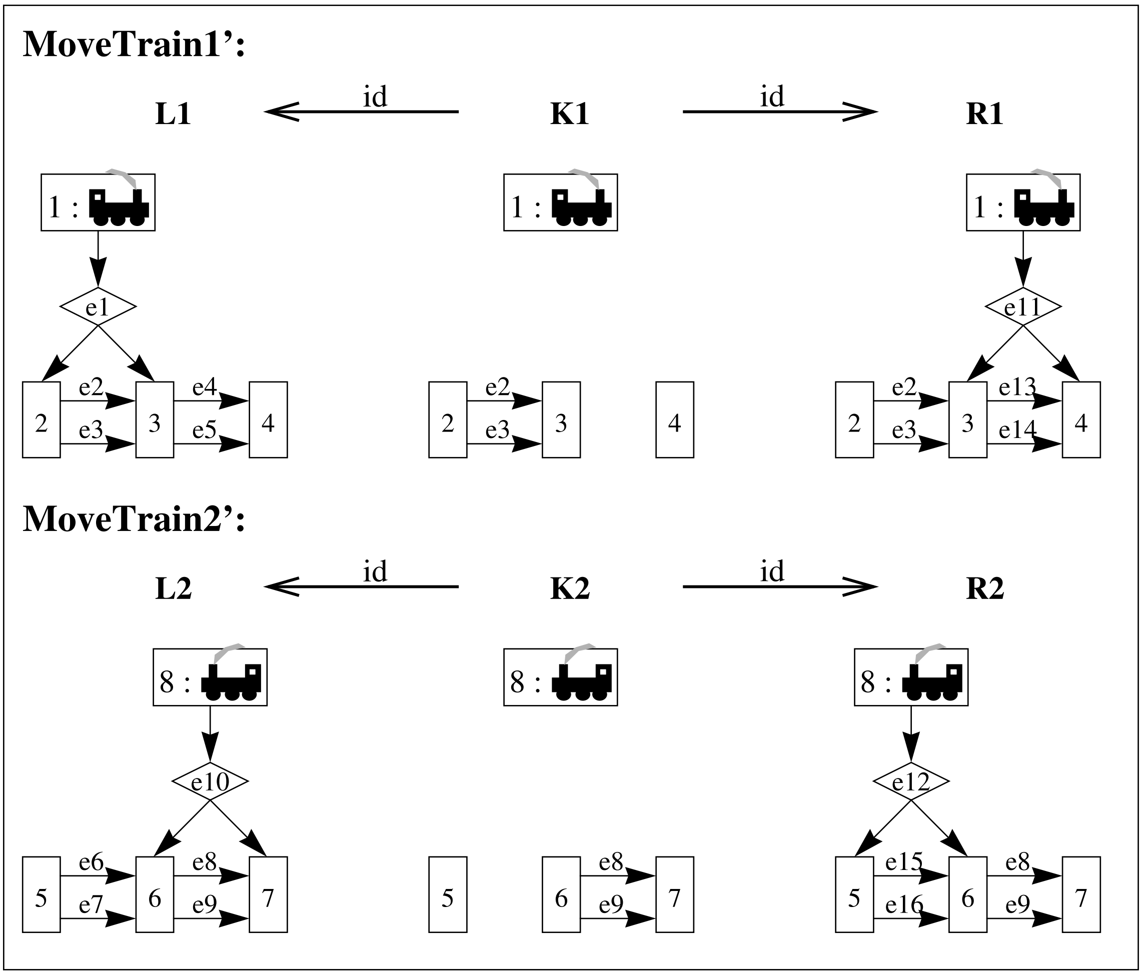


Abb. 3.16: Parallel “kollisionsfreie” Ersetzungsregeln

Mit Hilfe der beiden Graphersetzungsregeln aus Abb. 3.17 lassen sich also keine Zusammenstöße mehr herbeiführen. Will man nun aber in einem ganz bestimmten Anwendungsfall mit Hilfe dieser beiden Regeln dennoch einen Zusammenstoß provozieren, so darf man sie nicht disjunkt vereinigen bzw. getrennt voneinander ausführen, sondern muß sie “echt” miteinander verschmelzen (die disjunkte Vereinigung ist ein Spezialfall des Verschmelzens). Eine solche **Verschmelzung zweier Graphersetzungsregeln** bzw. ihrer drei

Graphkomponenten setzt voraus, daß man die zu identifzierenden Knoten und Hyperkanten einander zugeordnet hat. Dabei gehen wir der Einfachheit halber davon aus, daß jeder Knoten bzw. jede Hyperkante der einen Regel höchstens einem Knoten bzw. einer Hyperkante der anderen Regel zugeordnet wird (o.B.d.A. haben beide dann denselben Bezeichner).

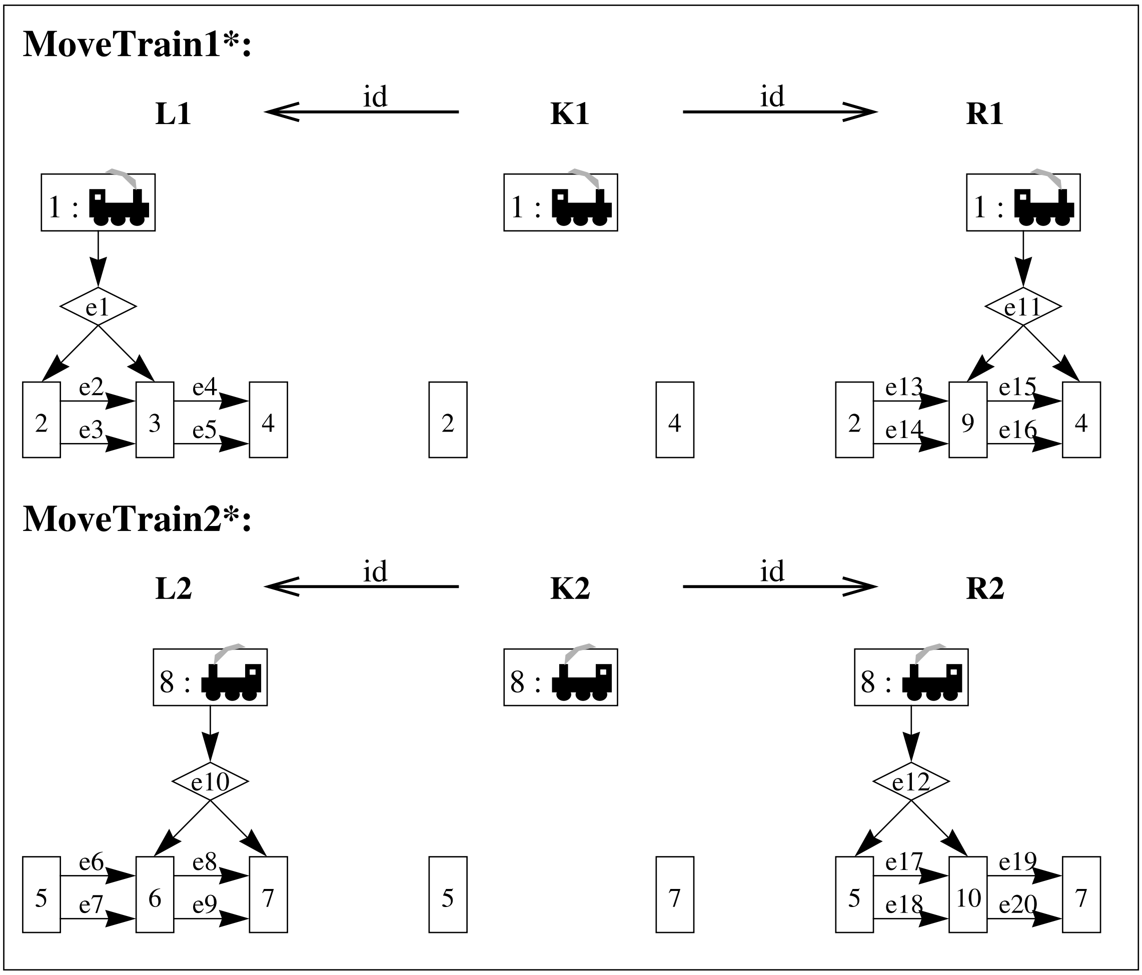


Abb. 3.17: Immer “kollisionsfreie” Ersetzungsregeln

Formal läßt sich damit das Verschmelzen zweier Teilgraphersetzungsregeln wie folgt definieren:

Def. 3.4.3 Verschmelzen von Graphersetzungsregeln

Seien r = (L, K, R), r1 = (L1, K1, R1) und r2 = (L2, K2, R2) $\in \mathcal{R}_H(\mathcal{L}_V, \mathcal{L}_E)$. Dann entsteht eine Graphersetzungsregel r' = (L', K', R') durch das Verschmelzen von r1 und r2 bzgl. r (in Zeichen: $r' := r1 +_r r2$) $\Leftrightarrow$

(1) id(L, L1), id(K, K1) und id(R, R1), die identischen Abbildungen der Komponenten von r auf die Komponenten von r1, sind (Mono-) Morphismen.

(2) id(L, L2), id(K, K2) und id(R, R2), die identischen Abbildungen der Komponenten von r auf die Komponenten von r2, sind (Mono-) Morphismen.

(3) $L' := L1 \;{}_{id}\!\cup_{id}\; L2$, $K' := K1 \;{}_{id}\!\cup_{id}\; K2$, $R' := R1 \;{}_{id}\!\cup_{id}\; R2$.

Der Spezialfall, daß r1 und r2 keine gemeinsamen Teile besitzen, also r = (∅, ∅, ∅) gilt, wird als die disjunkte Vereinigung bezeichnet und wie folgt notiert:

$$r' := r1 \oplus r2 \; . \; \blacksquare$$

Lemma 3.4.4 Wohldefiniertheit der Verschmelzung

Seien r, r1, r2 und r' wie oben definiert, es gelte also:

$$r' := r1 +_r r2 \; .$$

Dann ist $r' \in \mathcal{R}_H(\mathcal{L}_V , \mathcal{L}_E)$.

Beweis:

Mit r' = (L', K', R') muß nur gezeigt werden, daß id(K', L') und id(K', R') Morphismen sind. In Abb. 3.18 sehen wir, wie die Komponenten von r' aus denen von r1 und r2 als Pushouts abgeleitet werden. So gilt u.a

$$K' := K1 \;{}_{id(K,\,K1)}\!\cup_{id(K,\,K2)}\; K2 \; ,$$

K' ist also der Pushout von K1 und K2 bzgl. der identischen Morphismen von K nach K1 und K nach K2. Dabei können wir o.B.d.A. davon ausgehen, daß id(K1, K') und id(K2, K') die das Pushout-Diagramm vervollständigenden Morphismen sind.

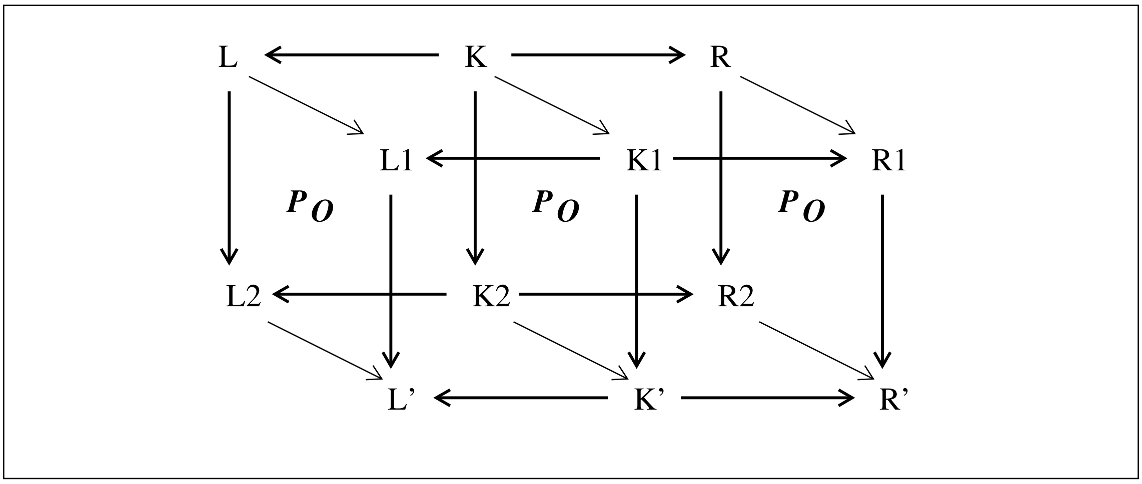


Abb. 3.18: Diagrammartige Darstellung der Verschmelzung

$$
\begin{tikzcd}
K \arrow[rr, "id"] \arrow[dd, "id"'] \arrow[ddrr, phantom, "\mathbf{PO}"] & & K1 \arrow[dr, "id"] \arrow[dd, "id"] & \\
& & & L1 \arrow[dd, "id"] \\
K2 \arrow[rr, "id"] \arrow[dr, "id"] & & K' \arrow[dr, "h"] & \\
& L2 \arrow[rr, "id"] & & L'
\end{tikzcd}
$$

Abb. 3.19: Einsatz der Pushout-Eigenschaft

Damit gibt es die Morphismen

$$h1: K1 \overset{\sim}{\rightarrow} L' := id(K1, L1) \circ id(L1, L')$$

und $$h2: K2 \overset{\sim}{\rightarrow} L' := id(K2, L2) \circ id(L2, L') \,.$$

Wendet man nun die in Def. 3.2.11 geforderten Pushout-Eigenschaften auf die in der Abb. 3.19 zusammenfassend dargestellte Situation an, so sieht man:

$$\exists\, h: K' \overset{\sim}{\rightarrow} L' : \quad id(K1, K') \circ h = h1 \quad \text{und} \quad id(K2, K') \circ h = h2.$$

Da nun h1 und h2 die identischen Abbildungen sind und K' keine Knoten und Hyperkanten enthält, die nicht bereits Bestandteil von K1 und K2 sind, gilt:

$$h = id(K', L') \,.$$

Damit ist gezeigt, daß die identische Abbildung ein Morphismus von K' nach L' ist. Genauso kann man zeigen, daß die Identität ein Morphismus von K' nach R' ist. ■ .

Nachdem wir das Verschmelzen theoretisch eingeführt haben, sollten wir nun versuchen, es auf unsere beiden Graphersetzungsregeln aus Abb. 3.17 so anzuwenden, daß eine Graphersetzungsregel "Crash" entsteht, die zwei Lokomotiven aufeinanderprallen läßt.

Bsp. 3.4.5 Verschmelzung zweier Graphersetzungsregeln

Als Vorbereitung für das Verschmelzen der beiden Regeln aus Abb. 3.17 werden wir zunächst die Numerierungen ihrer Knoten und Hyperkanten, die miteinander identifiziert werden sollen, aneinander angleichen. Dann konstruieren wir die Regel "r", die die zu identifizierenden Anteile in den neu entstandenen Regeln "MoveTrain1" und "MoveTrain2" explizit auszeichnet (man beachte, daß die beiden Regeln keine gemeinsamen Anteile besitzen, die von beiden Regeln identisch ersetzt werden; deshalb ist der identisch zu ersetzende Teil der Regel "r" leer). Das Ergebnis dieser Vorbereitungen sieht man in Abb. 3.20..

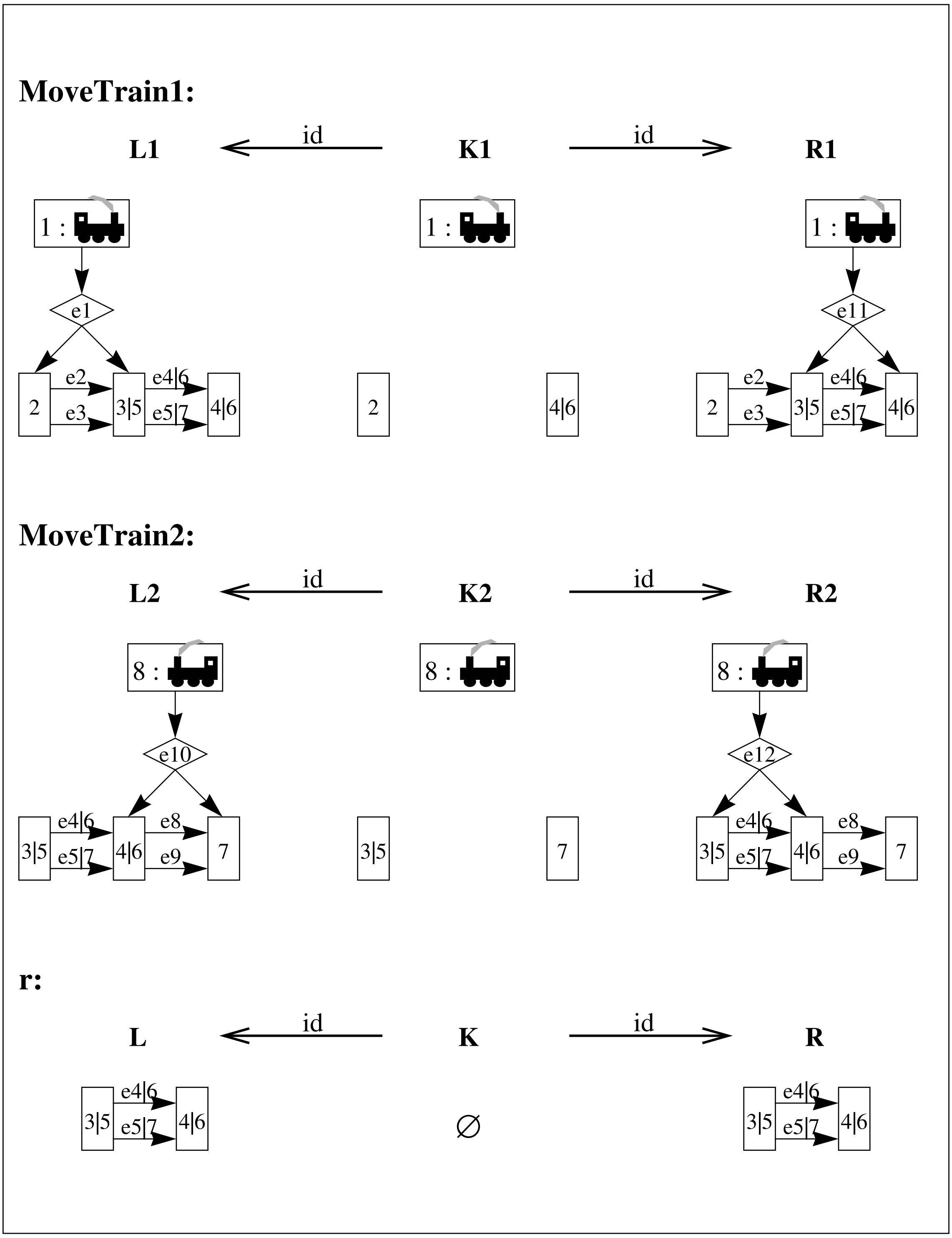


Abb. 3.20: Ausgangsregeln für das Verschmelzen

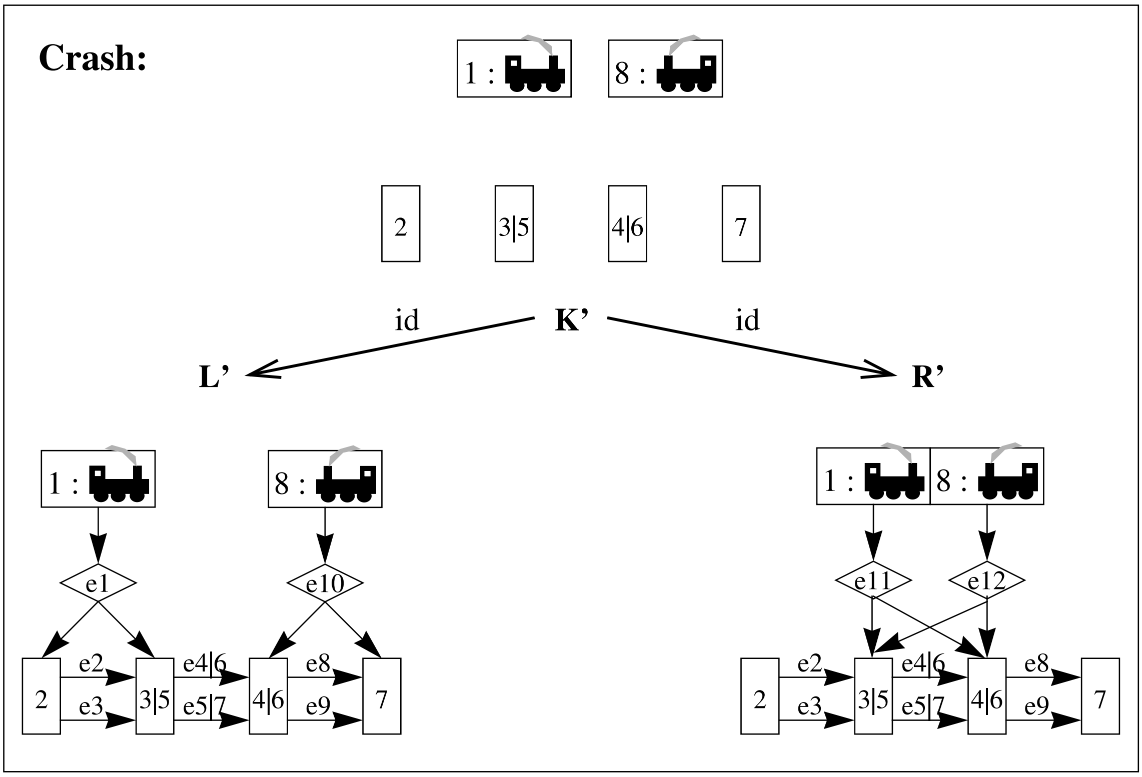


Abb. 3.21: Ausgangsregeln für das Verschmelzen

Gemäß Def. 3.4.3 können wir nun das eigentliche Verschmelzen durchführen und die Graphersetzungsregel

$$\text{Crash} := \text{MoveTrain1} +_r \text{ MoveTrain2}$$

bilden (siehe Abb. 3.21). Man beachte, daß sich die beiden Regeln "MoveTrain1" und "MoveTrain2" bzgl. der zu löschenden, identisch zu ersetzenden und einzufügenden Knoten nicht einig sind. So wird der Knoten 3|5 von "MoveTrain1" gelöscht (und danach ein Knoten mit "zufällig" derselben Nummer neu erzeugt), während der Knoten 3|5 in "MoveTrain2" identisch ersetzt wird.

Wie man an dem Ergebnis der Verschmelzung sieht,

- hat bei einem **"Identisch Ersetzen / Löschen"-Konflikt** das identische Ersetzen gegenüber dem Löschen und
- bei einem **"Identisch Ersetzen / Neu Erzeugen"-Konflikt** das identische Ersetzen gegenüber dem Neuerzeugen den Vorrang.

So gilt mit G und G' aus Abb. 3.15 tatsächlich:

$$G \sim \text{Crash} \rightsquigarrow G' \,.\ \blacksquare$$

Wie man bereits an dem obigen einfachen Beispiel sieht, kann man mit verschmolzenen Graphersetzungsregeln Graphtransformationen durchführen, die mit den Regeln, aus denen sie entstanden sind, keinesfalls möglich sind (im obigen Beispiel kann man durch Anwendung der einzelnen Regeln nie einen Zusammenstoß provozieren). So erhöht also das Verschmelzen von Graphersetzungsregeln die Anwendungsmöglichkeiten eines gegebenen Satzes von Regeln und unterstützt insbesondere

- das gleichzeitige Anwenden einer oder mehrerer Regeln an verschiedenen Stellen in einem Graphen (durch das disjunkte Vereinigen entsprechend vieler Regelkopien)
- sowie das getrennte Spezifizieren und anschließende Verschmelzen verschiedener Graphtransformationen mit gemeinsamen Anteilen.

Meist wird für dieVerschmelzung zweier Regeln r1 und r2 zu einer Regel r gefordert (anders als in unserem Beispiel), daß nur solche Knoten und Hyperkanten miteinander identifiziert werden dürfen, die von **beiden Regeln** entweder identisch ersetzt oder gelöscht oder neu erzeugt werden. In diesem Fall kann man zwei Regeln r1' und r2' konstruieren, so daß gilt:

$$\begin{aligned} G \sim r \rightsquigarrow G'' \quad & \Leftrightarrow \quad \exists\, G'\!: \; G \sim r1 \rightsquigarrow G' \;\wedge\; G' \sim r1' \rightsquigarrow G'' \\ & \Leftrightarrow \quad \exists\, G'\!: \; G \sim r2 \rightsquigarrow G' \;\wedge\; G' \sim r2' \rightsquigarrow G'' . \end{aligned}$$

Zum Abschluß sei darauf hingewiesen, daß der kategorientheoretische Ansatz nicht nur die Verschmelzung parallel anzuwendender Regeln unterstützt, sondern darüber hinaus auch die **Komposition hintereinander auszuführender Regeln** zu einer neuen Regel mit gleicher Semantik erlaubt. Das entsprechende Konstruktionsverfahren beruht jedoch u.a. auf der kategorientheoretischen Definition des Durchschnitts zweier Graphen und kann deshalb im Rahmen dieser Vorlesung - aus Zeitgründen - nicht mehr behandelt werden.

Insgesamt gesehen besitzt der kategorientheoretische Ansatz damit folgende Vor- bzw. Nachteile gegenüber dem mengentheoretischen Ansatz:

- Seine Ausdrucksfähigkeit ist - in der hier vorgestellten “klassischen” Form[†] - deutlich geringer als die des mengentheoretischen Ansatzes.
- Gerade die Einschränkungen in punkto Ausdruckskraft garantieren allerdings gewisse Eigenschaften von Graphersetzungsregeln, wie Invertierbarkeit und Komponierbarkeit, die in gewissen Anwendungsbereichen von großer Bedeutung sein können.
- Der Einsatz des mächtigen Instrumentariums der Kategorientheorie erlaubt es in vielen Fällen (dem geübten “Theoretiker”), gewisse Eigenschaften von Graphersetzungssystemen verblüffend einfach zu beweisen (so z.B. die Invertierbarkeit von Regeln).

---

† Neue, völlig anders aufgebaute kategorientheoretische Ansätze mit einer deutlich höheren Ausdruckskraft befinden sich in Entwicklung /Ba 90, Lö 91/.

Beiden Ansätzen ist eine Schwäche gemein: Sie unterstützen allein die Beschreibung graphverändernder Operationen und bieten keine Hilfsmittel für die statische Beschreibung der Eigenschaften bestimmter Klassen von Graphen und die Überprüfung, ob graphverändernde Operationen bestimmte Eigenschaften einer Klasse von Graphen erhalten.

3.5 Literatur

Zu den kategorientheoretischen Ansätzen gibt es unseres Wissens nach bislang keine umfassende Monographie. Einen recht ausführlichen, wenn auch schon etwas älteren Überblick über den "klassischen" kategorientheoretischen "**Double-Pushout**"-Ansatz findet man jedoch in /Eh 79/. Dort sind bereits alle wesentlichen Elemente des hier vorgestellten Ansatzes beschrieben. Eine Ausnahme bildet nur die Verschmelzung parallel anzuwendender Graphersetzungsregeln. Hierfür sei der Leser auf /BFH 87/ verwiesen.

Einen Überblick über neuere Enwicklungen auf dem Gebiet der algebraischen Ansätze findet man (in sehr kompakter Form) in /EKL 90/. Eine ausführliche Fassung dieses sogenannten "**Single-Pushout**"-Ansatzes stellt die Arbeit /Lö 91/ dar. Schließlich sei der Leser noch auf /Ba 90/ verwiesen. Dort wird wohl erstmalig der Versuch unternommen, eine Variante des mengentheoretischen Ansatzes mit komplexen Einbettungsüberführungsregeln auch mit den Hilfsmitteln der Kategorientheorie zu definieren. Für die kategorientheoretischen Grundbegriffe und Sätze sei der Leser auf /RB 88/ bzw. /Pa 69/ verwiesen.

/Ba 90/ Barthelmann K.: *Graphgrammatikalische Hilfsmittel zur Beschreibung verteilter Systeme*, Dissertation, IMMD-2, Universität Erlangen (1990)

/BFH 87/ Boehm P., Fondo H.-R., Habel A.: *Amalgamation of Graph Transformations: A Synchronization Mechanism*, in: Journal of Computer and System Sciences, No. 34, Academic Press (1987)

/CER 79/ Claus V., Ehrig H., Rozenberg G.: *Proc. Int. Workshop on Graph-Grammars and Their Application to Computer Science and Biology*, LNCS 73, Springer Verlag (1979)

/Eh 79/ Ehrig H.: *Introduction to the Algebraic Theory of Graph Grammars (a Survey)*, in: /CER 79/, S. 1- 69 (1979)

/EKL 90/ Ehrig H., Korff M., Löwe M.: *Tutorial Introduction to the Algebraic Approach of Graph Grammars Based on Double and Single Pushouts*, TR No. 90/21, TU Berlin (1990)

/Lö 91/ Löwe M.: *Extended Algebraic Graph Transformation*, Dissertation TU Berlin (1991)

/Pa 69/ Pareigis B.: *Kategorien und Funktoren*, Teubner-Verlag (1969)

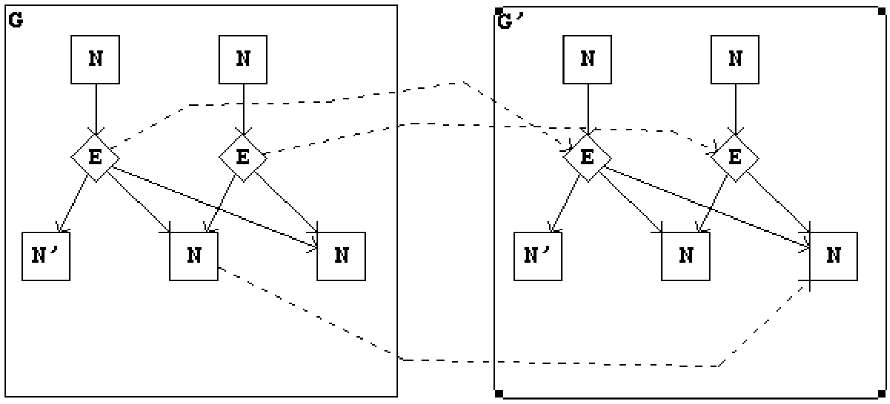
/RB 88/ Rydeheard D.E., Burstall R.M.: *Computational Category Theory*, Prentice Hall (1988)

3.6 Aufgaben

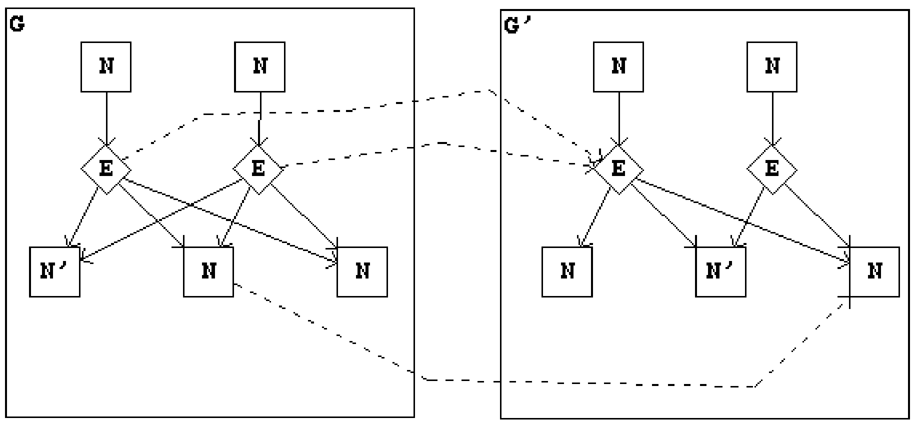
Aufgabe 3.1 Morpheus in der Graphenwelt

Vervollständigen Sie folgende Abbildungen zu *allen* möglichen Graph-Morphismen. Welche der dadurch entstehenden Abbildungen sind Isomorphismen, Epimorphismen, Monomorphismen oder gehören zu mehreren (welchen?) dieser Unterarten?

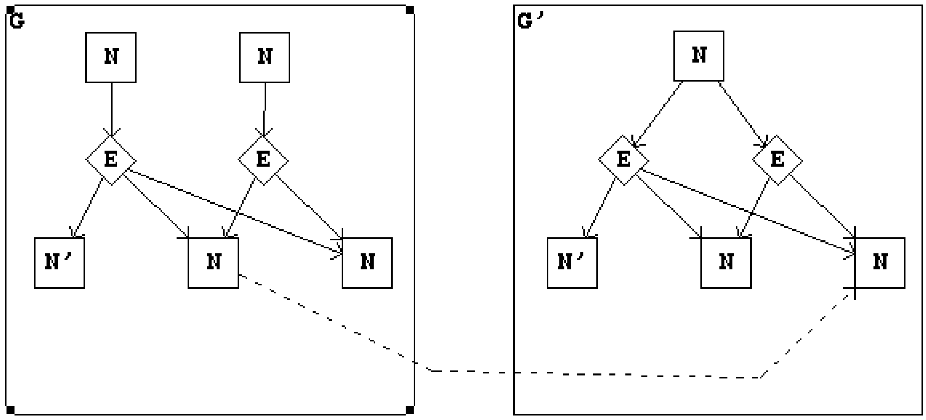
(1)



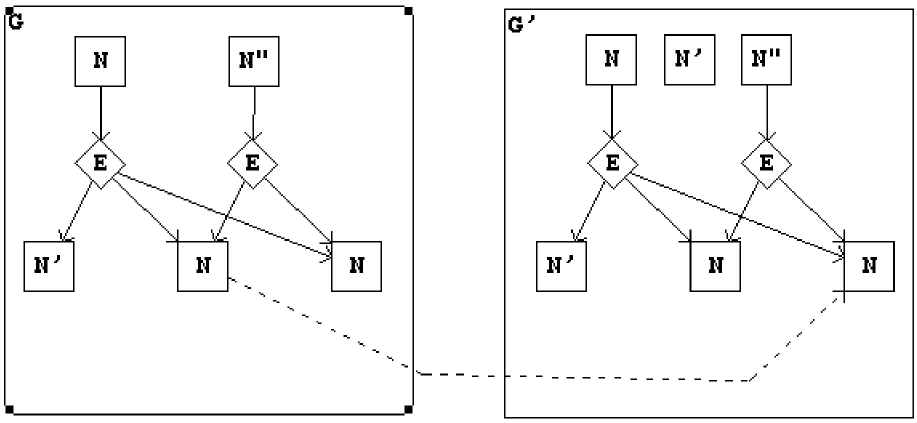
(2)



(3)



(4)

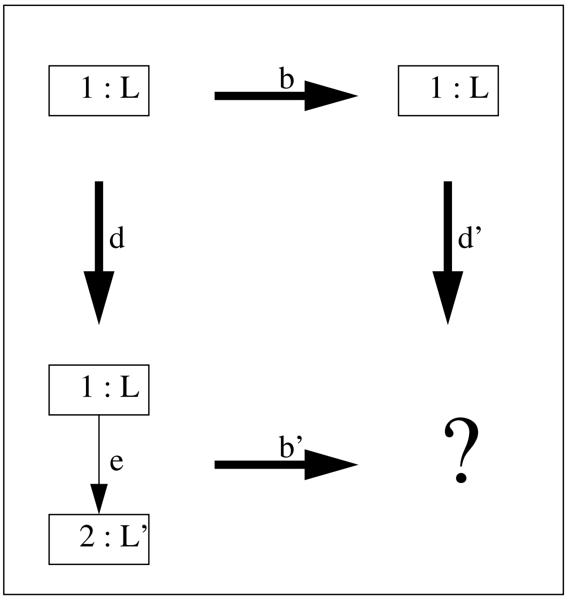


Aufgabe 3.2 Kategorientheoretische Vereinigung

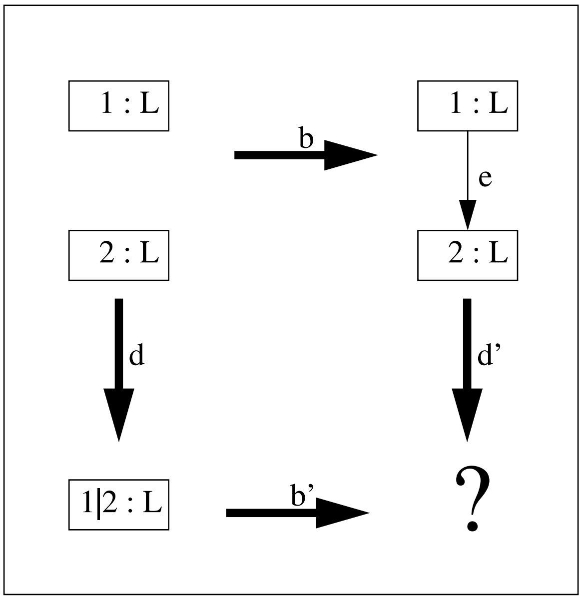
Berechnen Sie operational die folgenden Vereinigungen:

dabei seien b und d Morphismen, die einen Bezeichner x auf x bzw. auf x|y abbilden.

(1)



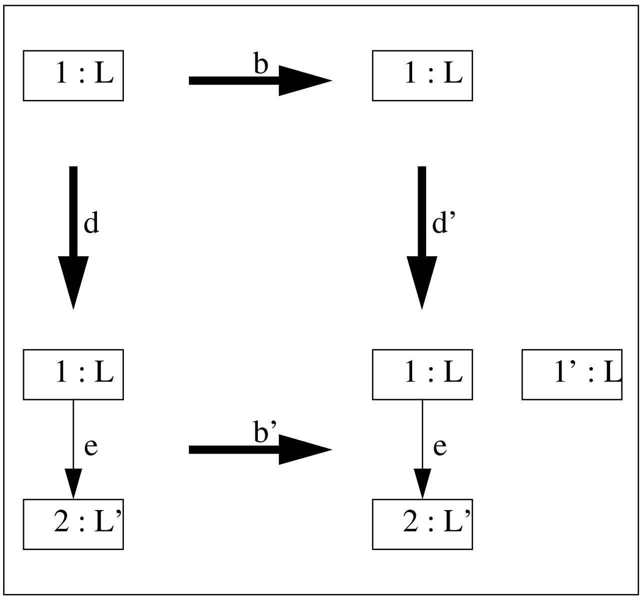
(2)



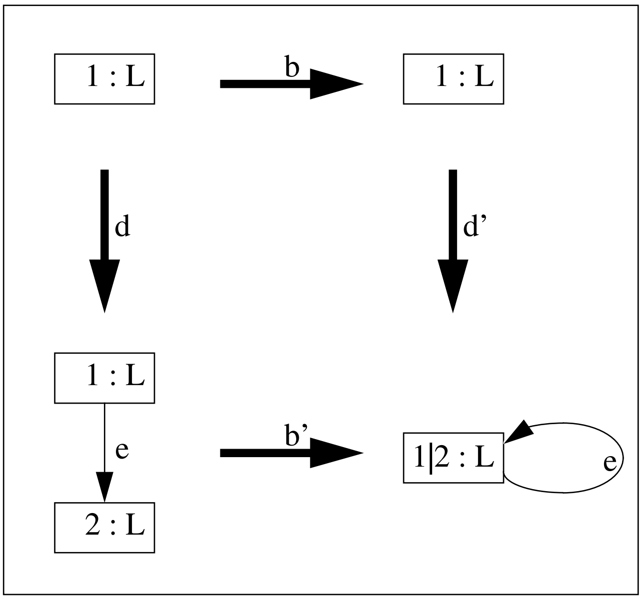
Aufgabe 3.3 Pushout-Diagramme

Zeigen Sie, daß die folgenden kommutierenden Diagramme keine Pushout-Diagramme sind:

(1)



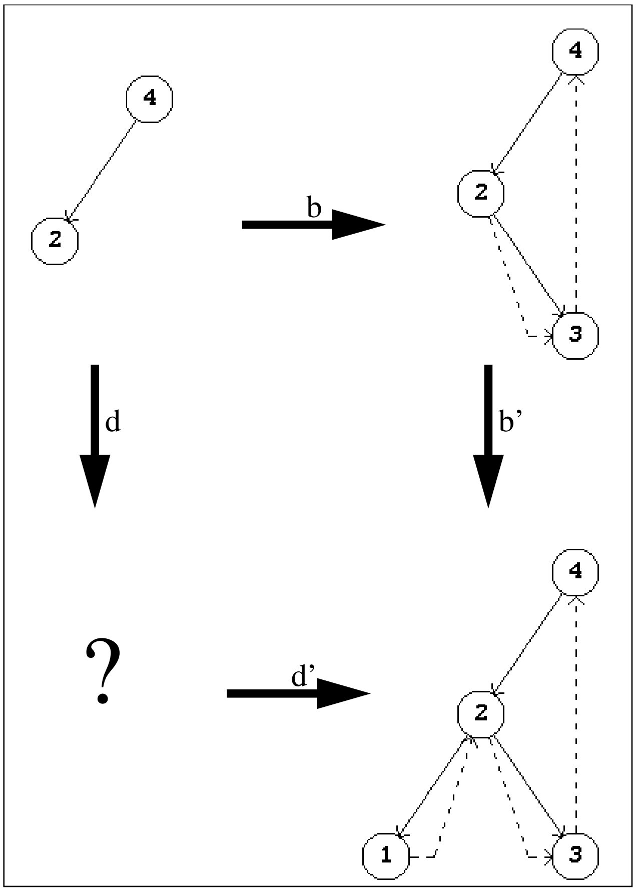
(2)



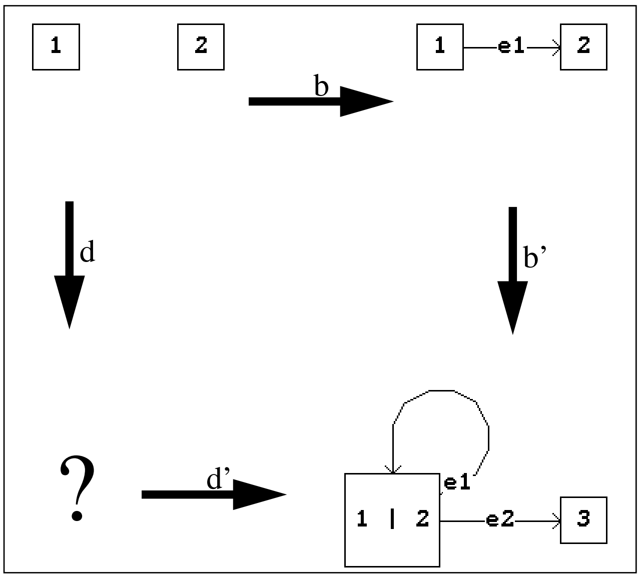
Aufgabe 3.4 Kategorientheoretische Differenz

Berechnen Sie folgende Differenzen:

(1):



(2):



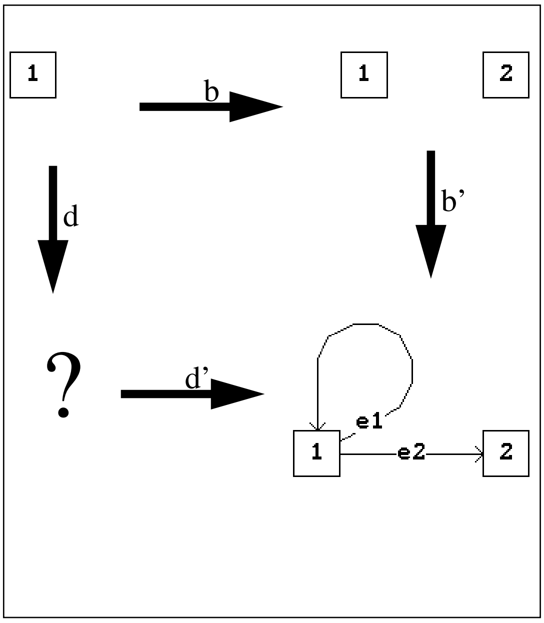
Aufgabe 3.5 Pushout-Diagramme

Können folgende Diagramme zu Pushout-Diagrammen vervollständigt werden?

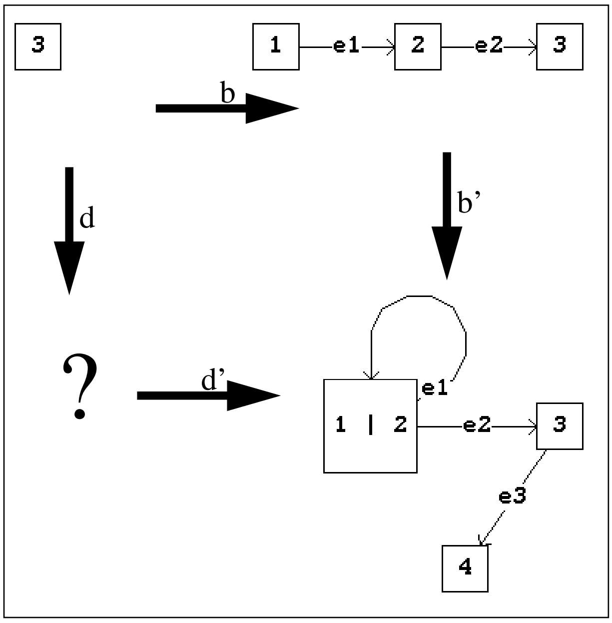
- Wenn ja: Geben Sie die Vervollständigung an. Beschreibt das so entstandene Diagramm eine Differenzbildung?
- Wenn nein: Warum nicht?

Begründen Sie Ihre Antworten anhand der Definitionen der Vorlesung.

(1)

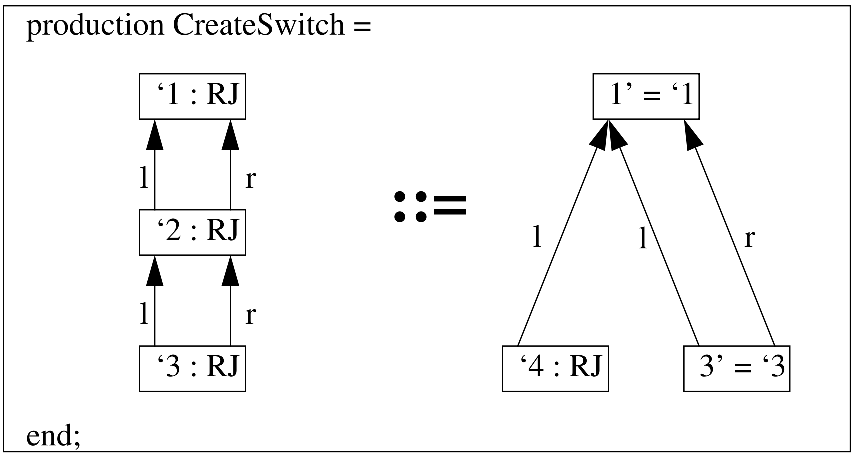


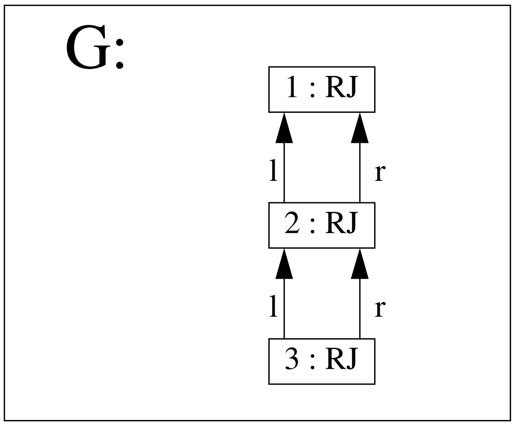
(2)



Aufgabe 3.6 PROGRESS <-> Kategorientheorie

Gegegeben sei die Produktion CreateSwitch und nachstehender Gleisabschnitt G:



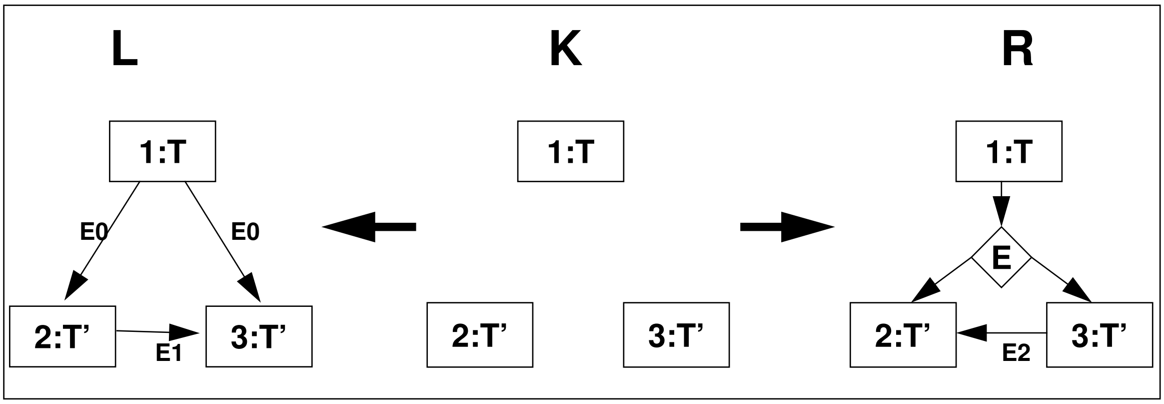


Beschreiben Sie die Anwendung von CreateSwitch auf G mit folgender Abbildung:

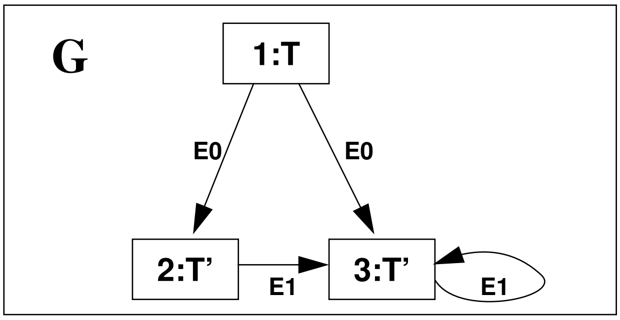
$$\begin{tikzcd} L \arrow[d] & K \arrow[l] \arrow[r] \arrow[d] & R \arrow[d] \\ G & D \arrow[l] \arrow[r] & G' \end{tikzcd}$$

Aufgabe 3.7 Regelanwendung

Gegeben sei die Ersetzungsregel

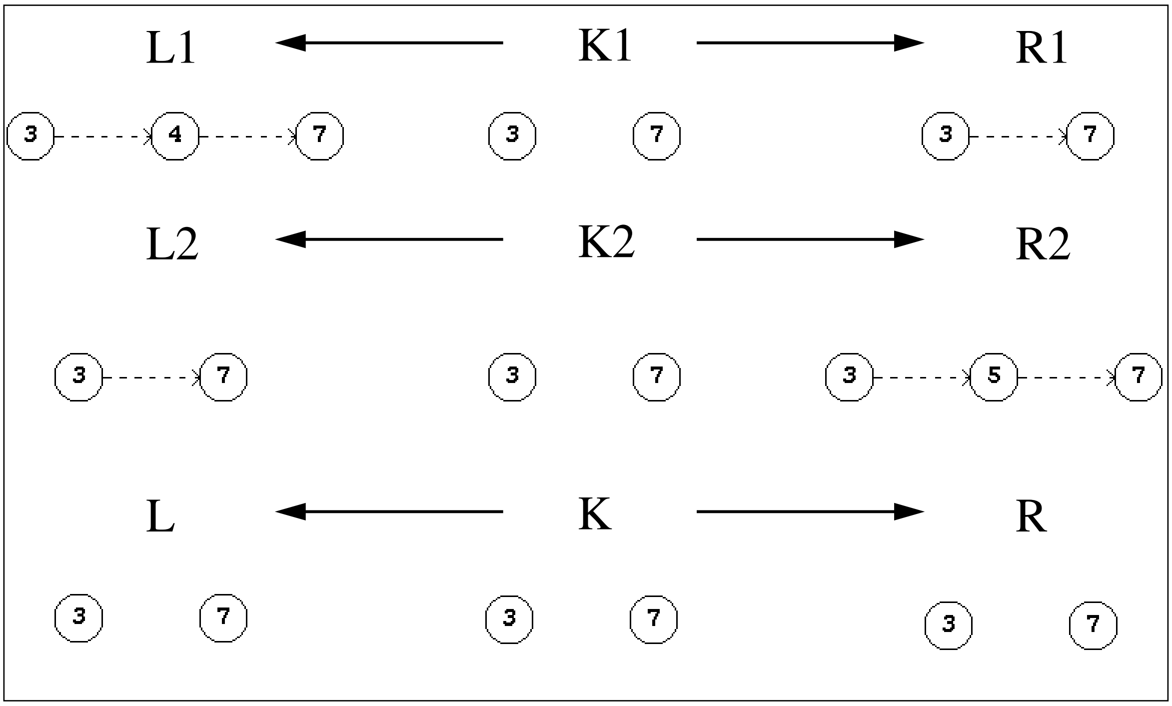


Berechnen Sie alle Graphen, die entstehen können, wenn diese Regel auf den folgenden Graphen angewendet wird. Berücksichtigen Sie dabei auch homomorphe Abbilder der linken Regelseite im Graphen!



Aufgabe 3.8 Löschen + Einfügen = Ersetzen?

Verschmelzen Sie r1 und r2 bezgl. r:

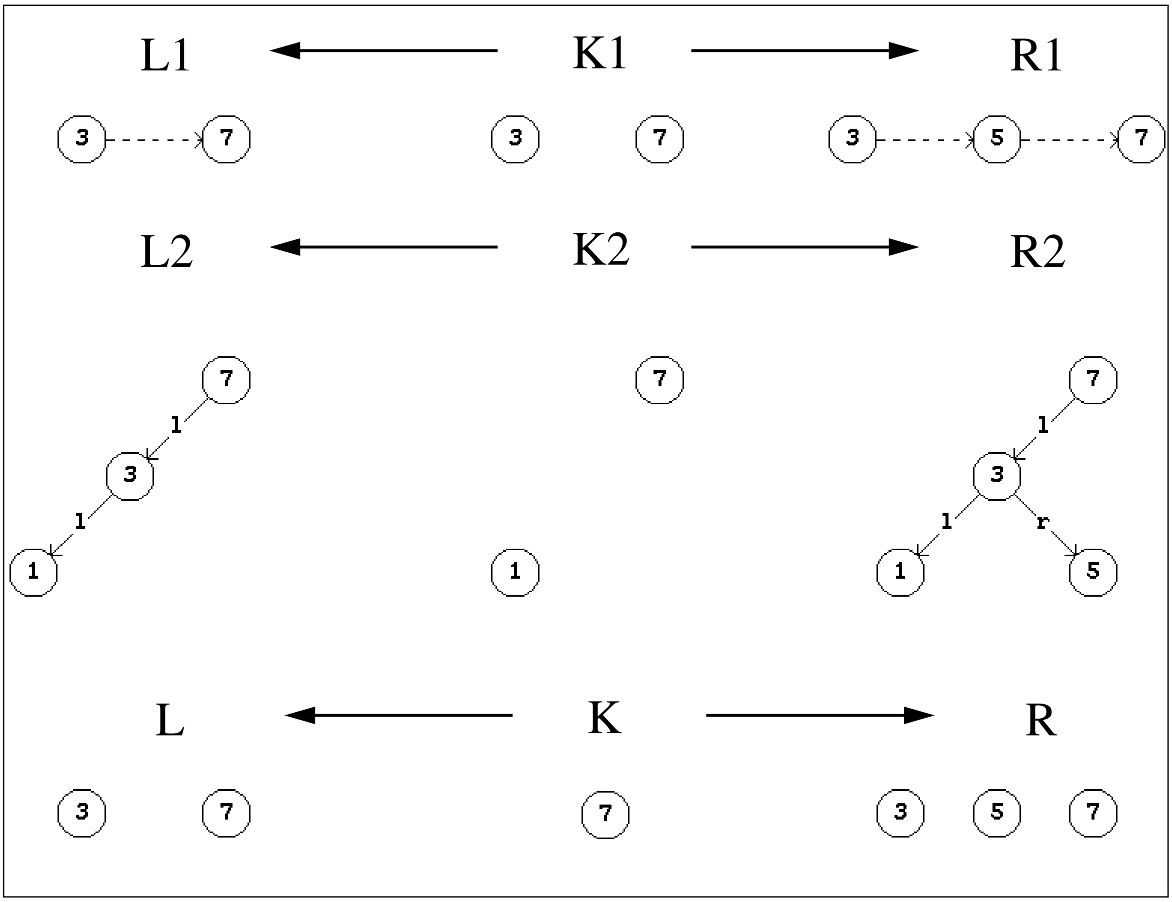


Aufgabe 3.9 Liste + Baum = gefädelter Baum?

Wir wollen eine (kategorientheoretische) Regel r konstruieren, die in einem gefädelten Baum ein neues Element einfügt, wobei gelten soll:

- Das neue Element wird rechts an ein existierendes Element angehängt.
- Das existierende Element ist linker Sohn eines anderen Knotens.
- Das existierende Element hat schon einen linken Sohn.
- Die Regel soll nicht anwendbar sein, wenn das existierende Element schon einen rechten Sohn hat.

(1)Verschmelzen Sie r1 und r2 bzgl. r:



(2)Erfüllt die in (1) entstandene Regel die oben aufgeführten Punkte?

4. Logikorientierter Ansatz

In den vorangegangenen zwei Kapiteln haben wir sowohl mengentheoretisch als auch kategorientheoretisch fundierte Graphregelsysteme kennengelernt. Bei ihrer Besprechung haben wir uns allein auf die formale Definition der grapherzeugendenen bzw. graphverändernden Regeln konzentriert. Folgende Fragestellungen wurden dabei außer acht gelassen (und waren auch im Rahmen der vorgestellten Ansätze nicht zu behandeln):

(1) Aus welchen Bestandteilen setzen sich die Graphen des Sprachschatzes eines bestimmten Graphregelsystems zusammen, und welche Integritätsbedingungen müssen beim Aufbau von Graphen aus diesen Bestandteilen beachtet werden?

(2) Wie lassen sich ableitbare Grapheigenschaften und komplexe Anwendbarkeitsbedingungen für Graphersetzungsregeln einfach formulieren und formal beschreiben?

(3) Wann ist ein Graphregelsystem in sich stimmig, d.h. unter welchen Bedingungen lassen sich mit seinen Ersetzungsregeln nur solche Graphen erzeugen, deren Aufbau nicht gegen eine zusätzliche Menge von Integritätsbedingungen verstößt?

(4) Und wie sieht eine bequeme Notationsweise aus, die den Anwender vor allen Details der zugrundeliegenden Formalismen weitgehend schützt.

Um die ersten beiden Fragen beantworten zu können, haben wir einen Graphersetzungsbegriff neu definiert, der auf dem Fundament der **Prädikatenlogik erster Stufe** (mit Gleichheit) ruht. Er unterstützt neben der identischen Ersetzung von Knoten aus Kapitel 3 und den komplexen Einbettungsüberführungsregeln aus Kapitel 2 auch die Definition von Integritätsbedingungen und Attributberechnungsregeln in Form sogenannter **Graphschemata**. Zudem erlaubt er auch die Formulierung von Anwendbarkeitsbedingungen für Graphersetzungsregeln mit Hilfe abgeleiteter Relationen. Für die Beantwortung der anderen beiden Fragen haben wir mit Hilfe des neuen, logikorientierten Formalismus eine **streng typisierte "Programmiersprache"** mit wohldefinierter Syntax und Semantik eingeführt.

Um begriffliche Verwirrungen zu vermeiden, werden wir im folgenden den zugrundeliegenden Formalismus "unseren Graphersetzungskalkül" und seine sprachliche Einkleidung "PROGRESS†" nennen. Die Sprache PROGRESS und unser logikbasierter Graphersetzungskalkül sind prinzipiell auch unabhängig voneinander einsetzbar. So haben wir ja bereits in Kapitel 2 PROGRESS als sprachliche Einkleidung für einige mengentheoretische Ansätze verwendet‡.

† "PROGRESS" steht für PROgrammierte GRaph-Ersetzungs-System-Sprache.

‡ Daraus kann man indirekt schließen, daß der hier einzuführende logikbasierte Graphersetzungskalkül alle Möglichkeiten der vorgestellten mengentheoretischen Ansätze umfaßt

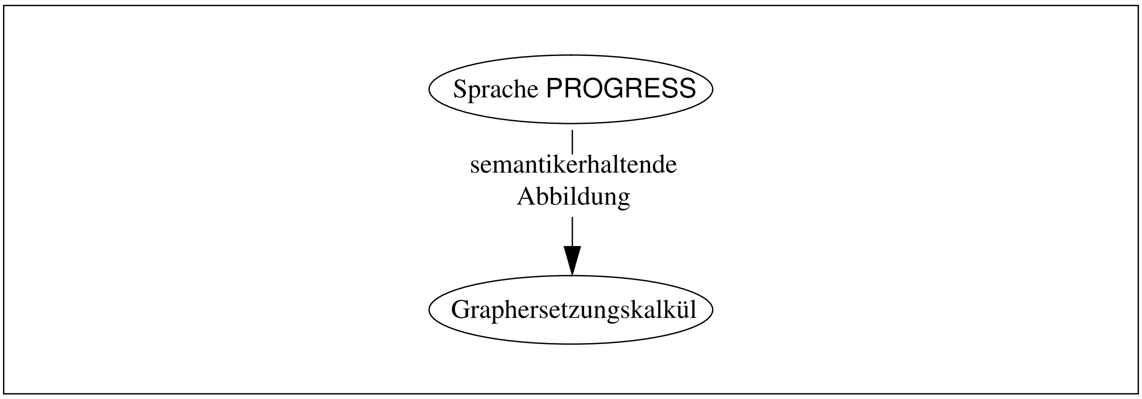


Abb. 4.1: Formale Definition der Sprache PROGRESS

Im Rahmen dieses Kapitels werden wir nun schrittweise die Sprache PROGRESS und ihren zugrundegelegten Kalkül einführen. Dabei werden wir aus Platzgründen sowohl die Sprachbeschreibung als auch die Erläuterung der semantikerhaltenden Abbildung auf unsern Kalkül (siehe Abb. 4.1) nur exemplarisch durchführen. Und auch unseren neuen Graphersetzungskalkül werden wir nur anhand von Definitionsskizzen vorstellen. Dabei werden wir in den folgenden Etappen vorgehen:

(1) In Abschnitt 4.1 erläutern wir das im folgenden laufend benutzte Beispiel und führen Graphen als Mengen prädikatenlogischer Formeln ein.

(2) In Abschnitt 4.2 befassen wir uns mit allen Bestandteilen der Sprache PROGRESS zur Beschreibung von Graphschemata (als Mengen prädikatenlogischer Formeln) und definieren den Begriff “schematreuer Graph”.

(3) In Abschnitt 4.3 kommen wir auf die Bestandteile der Sprache PROGRESS zu sprechen, die wir für die Definition komplexer Anwendbarkeitsbedingungen in Graphersetzungsregeln benötigen.

(4) In Abschnitt 4.4 führen wir schließlich Teilgraphentests und Teilgraphersetzungen (mit Hilfe des prädikatenlogischen Ableitungsbegriffs) ein, mit denen wir nur schematreue Graphen erzeugen und verändern können.

(5) Den Abschluß bildet Abschnitt 4.5 mit der Angabe einiger Literaturhinweise.

Einen ganz wesentlichen Bestandteil der Sprache PROGRESS, die sogenannten **Kontrollstrukturen** samt ihrer formalen Grundlage, den **Kontrolldiagrammen**, werden wir hier nicht behandeln. Damit haben wir uns ja bereits in Kapitel 2 bei den mengentheoretischen Ansätzen ausführlich auseinandergesetzt.

4.1 Grundbegriffe

In diesem Abschnitt führen wir die Klasse der gerichteten, attributierten, knoten- und kantenmarkierten Graphen ein, die wir im folgenden **gakk-Graphen** oder auch nur einfach **Graphen** nennen werden. Solche Graphen werden von uns dazu verwendet, konkrete Modelle abstrakter Datentypen mit PROGRESS zu spezifizieren und damit auch zu implementieren.

Für die Einführung in den Umgang mit gakk-Graphen wollen wir uns überlegen, wie sich die Sätze einer einfachen Programmiersprache als Instanzen einer gewissen Klasse von Graphen darstellen lassen. Hierfür werden wir von der textuellen Darstellung dieser Sprache abstrahieren und nur ihre abstrakte, kontextfreie Syntax zuzüglich der Bezeichnerbindungen auf sogenannte **abstrakte Syntaxgraphen** abbilden. Als Beispiel soll uns eine applikative Sprache “ExpLanguage” dienen, die das Rechnen mit ganzzahligen Ausdrücken unterstützt.

Die **konkrete Syntax** der Sprache “ExpLanguage” findet man in Abb. 4.2 (auf Klammerung und Operatorpräzedenzen wurde verzichtet, da sie beim syntaxgesteuerten Edieren mehr oder weniger bedeutungslos sind). Ihre Sätze stellen ganzzahlige Ausdrücke dar, in denen mit Hilfe geschachtelter Konstantendefinitionen der Form

“let c := <Definierender Ausdruck> in <Ausdruck> end”

Zwischenergebnisse benannt und wiederverwendet werden können.

<table>
<tr><td>(1)</td><td>EXP ::=</td><td>IntLiteral | DefExp | ApplId | PlusExp | MinusExp | MultExp | DivExp ;</td></tr>
<tr><td>(2)</td><td>IntLiteral ::=</td><td>Text ; (* Aufbau der textuellen Darstellung ganzer Zahlen. *)</td></tr>
<tr><td>(3)</td><td>DefExp ::=</td><td>“let” DeclId “:=” EXP “in” EXP “end” ;</td></tr>
<tr><td>(4)</td><td>DeclId ::=</td><td>Name ; (* Aufbau von Konstantenbezeichnern. *)</td></tr>
<tr><td>(5)</td><td>ApplId ::=</td><td>Name ;</td></tr>
<tr><td>(6)</td><td>PlusExp ::=</td><td>EXP “+” EXP ;</td></tr>
<tr><td>(7)</td><td>MinusExp ::=</td><td>EXP “-” EXP ;</td></tr>
<tr><td>(8)</td><td>MultExp ::=</td><td>EXP “*” EXP ;</td></tr>
<tr><td>(9)</td><td>DivExp ::=</td><td>EXP “/” EXP ;</td></tr>
</table>

Abb. 4.2: Konkrete Syntax der Sprache “ExpLanguage”

Die üblichen **Sichtbarkeitsregeln** für blockstrukturierte Sprachen bestimmen die Bindung angewandter Auftreten von Namen an die zugehörigen Deklarationen. Vorsicht ist allerdings bei Ausdrücken der Gestalt

“let x := ... x ... in ... end”

geboten. Hier wird das angewandte Auftreten einer Konstante “x” nicht an die Deklaration gebunden, deren Wert sie mit festlegt. Das angewandte Auftreten wird vielmehr an die nächstumfassende Deklaration desselben Namens gebunden, in der es nicht zwischen “:=” und “in”, sondern zwischen “in” und “end” plaziert ist.

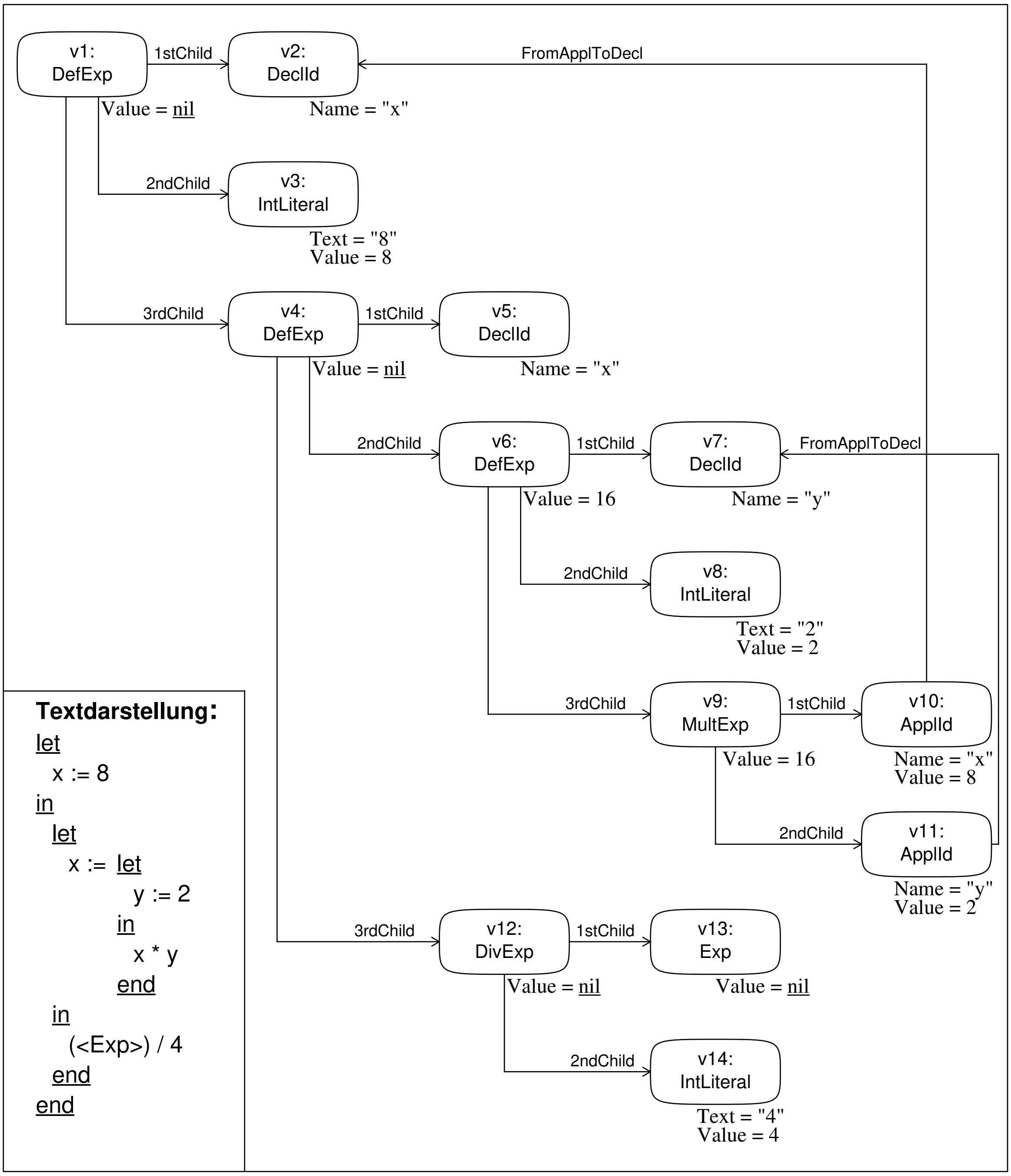


Abb. 4.3: Text- und Graphrepräsentation eines Ausdrucks

In der konkreten Syntaxdefinition der Sprache findet man im übrigen **drei verschiedene Arten von Nichtterminalsymbolen**:

(1) Das Symbol “EXP” bezeichnet den syntaktischen Bereich aller Ausdrücke, der Elemente ganz unterschiedlicher Struktur enthält.

(2) Die Symbole “DefExp”, “MultExp” etc. fassen die Elemente eines syntaktischen Bereiches zusammen, die denselben syntaktischen Aufbau besitzen.

(3) Und die Symbole “IntLiteral”, “DeclId” etc. bilden die Schnittstelle zur lexikalischen Syntax der Sprache. Sie legen in unserem Falle den Aufbau der textuellen Darstellung von ganzen Zahlen und von Konstantennamen fest.

In der Abb. 4.3 sehen wir die Text- und Graphrepräsentation eines typischen Satzes der Sprache “ExpLanguage”. Da seine Eingabe mit einem syntaxgesteuerten Editor noch nicht abgeschlossen ist, enthält der Satz einen noch nicht expandierten Teilausdruck. Dieser Teilausdruck wird durch den **Platzhalter** “(<Exp>)” textuell und durch den mit Exp markierten Knoten v13 graphisch dargestellt.

Da unser Ausdruck noch nicht vollständig eingegeben ist, konnte auch sein Gesamtwert noch nicht berechnet werden. Deshalb besitzt das Value-Attribut der Knoten v1, v4, v12 und v13 keinen definierten Wert (solche undefinierten Werte werden in PROGRESS durch das Schlüsselwort nil repräsentiert). Anders ist das bei dem Teilausdruck

“let y := 2 in x * y end” .

Er hat den Wert “16”, da die Konstanten x und y an die Zahlen “8” und “2” gebunden sind.

Ausgehend von der Definition der konkreten Syntax in Abb. 4.2 wurde der systematische Entwurf des abstrakten Syntaxgraphen durch folgende **Richtlinien** gesteuert:

(1) Bilde die kontextfreie syntaktische Struktur eines “ExpLanguage”-Satzes auf eine Graphstruktur ab, die den abstrakten Syntaxbaum des Satzes darstellt. Diese Struktur bildet das Skelett des Graphen und enthält jeweils einen **markierten Knoten** für die Wurzel eines jeden Teilausdruckes. Die verschiedenen Arten von Teilausdrücken werden durch die Markierung ihrer Wurzelknoten mit den Symbolen DefExp, MultExp etc. aus der konkreten Syntaxdefinition unterschieden, die Operatoren der zugrundeliegenden, abstrakten Syntax entsprechen.

(2) Repräsentiere die Enthaltenseinsbeziehungen des abstrakten Syntaxbaumanteils durch **gerichtete Kanten**, die z.B. die Markierungen 1stChild, 2ndChild etc. besitzen, und definiere jeweils einen zusätzlichen Typ von Kanten für alle weiteren Beziehungen zwischen Knoten. In unserem Beispiel führen wir mit FromApplToDecl markierte Kanten ein, die die angewandten Auftreten von Namen an ihre Deklarationen binden.

(3) Verwende sogenannte **eigenständige (intrinisic) Knotenattribute**, um atomare Eigenschaften einzelner Knoten im Graphen abzulegen. In unserem Beispiel sind das alle Elemente der lexikalischen Syntaxbereiche “Text” und “Name”. Deshalb erhalten alle Knoten des Typs IntLiteral ein eigenständiges Text-Attribut und alle Knoten der Typen ApplId und DeclId ein eigenständiges Name-Attribut.

(4) Verwende **abgeleitete (derived) Knotenattribute** für die Codierung von Knoteneigenschaften, die sich aus anderen Attributwerten errechnen lassen. Ein solches Attribut wird abgeleitet genannt, weil sein Wert durch eine gerichtete Gleichung über anderen Attributen desselben Knotens oder benachbarter Knoten festgelegt wird. So lassen sich funktionale Attributabhängigkeiten spezifizieren, wie etwa: “Das Value-Attribut eines MultExp-Knotens muß gleich dem Produkt der Value-Attribute seiner beiden Operanden sein, die die Zielknoten der auslaufenden 1stChild- und 2ndChild-Kanten sind”.

Damit haben wir für fast alle Symbole der Syntaxdefinition aus Abb. 4.2 ein passendes Gegenstück in unserem graphischen Datenmodell gefunden. Eine Ausnahme bildet nur das Nichtterminalsymbol “EXP”, das die verschiedenen Arten von Teilausdrücken zu einem syntaktischen Bereich zusammenfaßt. Sein Gegenstück werden wir erst im folgenden Abschnitt in Form einer Knotenklassendeklaration kennenlernen.

Wenden wir uns nun der Definition von gakk-Graphen als Mengen von Formeln zu. Hierfür müssen wir zunächst einige (prädikatenlogische) Grundbegriffe einführen.

Def. 4.1.1 Graphsignatur (Def.-Skizze)

Ein 7-Tupel $\Sigma := (\mathcal{L}_F, \mathcal{L}_P, \mathcal{L}_T, \mathcal{L}_R, \mathcal{V}, \mathcal{W}, \mathcal{X})$, das die zum Aufbau einer Klasse von Graphen benötigten Symbole festlegt, wird Graphsignatur genannt. Für die einzelnen Komponenten gilt:

(1) $\mathcal{L}_F$ ist das Alphabet der Bezeichner für Attributberechnungsvorschriften.

(2) $\mathcal{L}_P$ ist das Alphabet der Bezeichner für Prädikate über Attributwerten.

(3) $\mathcal{L}_T$ ist das Alphabet der Knotenmarkierungen und Attributtypbezeichner.

(4) $\mathcal{L}_R$ ist das Alphabet der Kantenmarkierungen und Attributbezeichner.

(5) $\mathcal{V}$ ist das Alphabet der Knotenbezeichner.

(6) $\mathcal{W}$ ist das Alphabet der Bezeichner für Knotenmengen bzw. bel. Wertemengen.

(7) $\mathcal{X}$ ist das Alphabet der Subjektvariablen, über die in Formeln quantifiziert wird.

■

Bsp. 4.1.2 Graphsignatur zu “ExpLanguage”

Eine mögliche Graphsignatur Σ :=($\mathcal{L}_F$, $\mathcal{L}_P$, $\mathcal{L}_T$, $\mathcal{L}_R$, $\mathcal{V}$, $\mathcal{W}$, $\mathcal{X}$) für den Graphen aus der Abb. 4.3 besteht aus folgenden Alphabeten:

(1) $\mathcal{L}_F$:= { 0, 1, … , +, … } .

(2) $\mathcal{L}_P$:= { =, … } .

(3) $\mathcal{L}_T$:= { INTEGER, DefExp, IntLiteral, … } .

(4) $\mathcal{L}_R$:= { Value, … , FromApplToDecl, 1stChild, … } .

(5) $\mathcal{V}$:= { v1, v2, … } .

(6) $\mathcal{W}$:= { w1, w2, … } .

(7) $\mathcal{X}$:= { x1, x2, … } .

Die Alphabete $\mathcal{W}$ und $\mathcal{X}$ werden wir für die Konstruktion von “ExpLanguage”-Graphen noch nicht benötigen. Sie spielen erst bei der Definition des zugehörigen Graphschemas und der Graphersetzungsregeln eine Rolle. ■

Def. 4.1.3 Σ-Term (Def.-Skizze)

Sei Σ eine Graphsignatur. Dann wird mit TERM(Σ) die Menge aller Terme bezeichnet, die sich aus den Funktionssymbolen in $\mathcal{L}_F$ und den freien Variablen aus $\mathcal{X}$ sowie den Alphabeten $\mathcal{L}_T$, $\mathcal{L}_R$, $\mathcal{V}$ und $\mathcal{W}$ als Konstantenbezeichner aufbauen lassen. ■

Def. 4.1.4 Σ-Primformel (Def.-Skizze)

Sei Σ eine Graphsignatur. Dann wird mit PRIM(Σ) eine Menge von logischen Formeln bezeichnet, die durch die Anwendung eines Prädikatsymbols auf eine Menge von Termen entsteht, die also keine aussagenlogischen Verknüpfungsoperatoren oder prädikatenlogische Quantoren enthalten. Diese einfachen Formeln, Primformeln (atomare Formeln) genannt, dürfen neben den Prädikatssymbolen aus $\mathcal{L}_P$ folgende drei Prädikatssymbole zusätzlich enthalten:

(1) “=” für die Gleichheit zweier Σ-Terme.

(2) “*type*” mit “*type*(x, vl)” für “der Knoten x besitzt die Markierung vl” bzw. mit “*type*(τ, t)” für “der Term τ besitzt den Typ t”.

(3) “*rel*” mit “*rel*(x, el, y)” für “eine Kante mit der Markierung el verbindet den Knoten x mit dem Knoten y” bzw. mit “*rel*(x, a, τ)” für “das Attribut a besitzt am Knoten x den Wert τ”. ■

Def. 4.1.5 Σ-Formel (Def.-Skizze)

Sei Σ ein Graphsignatur. Dann wird mit FORM(Σ) die Menge aller prädikatenlogischen Formeln über den Σ-Termen TERM(Σ) bezeichnet, die sich mit Hilfe der aussagenlogischen Operatoren $\wedge$, $\vee$, … und der beiden prädikatenlogischen Quantoren $\exists$ und $\forall$ aus den Primformeln PRIM(Σ) aufbauen lassen. ■

Def. 4.1.6 Ableitungsbegriff (Def.-Skizze)

Seien Φ und Φ' zwei Mengen von prädikatenlogischen Formeln. Dann bedeutet

$$\Phi \vdash \Phi' ,$$

daß sich die Formeln von Φ' aus denen von Φ ableiten lassen†. ■

Def. 4.1.7 Σ-Graphform (Def.-Skizze)

Sei Σ eine Graphsignatur. Dann wird $F \subseteq$ FORM(Σ) Σ-Graphform genannt (in Zeichen: $F \in \mathcal{F}_L(\Sigma)$) $\Leftrightarrow$

F enthält keine Subjektvariablen (aus $\mathcal{X}$ von Σ), die nicht durch "Quantoren" gebunden sind‡. ■

Im folgenden werden wir Graphen, Graphschemata, Anwendbarkeitsbedingungen etc. als Spezialformen der allgemein gehaltenen Definition einer Σ-Graphform kennenlernen. Beginnen wir mit der Definition eines Σ-Graphen:

Def. 4.1.8 Σ-Graph (Def.-Skizze)

Sei Σ eine Graphsignatur. Dann wird eine Graphform $G \in \mathcal{F}_L(\Sigma)$ Σ-Graph genannt (in Zeichen: $G \in \mathcal{G}_L(\Sigma)$) $\Leftrightarrow$

$G \subseteq$ PRIM(Σ) und keine Formel von G enthält das Gleichheitszeichen "=". ■

Ein Σ-Graph ist also eine Menge positiver Fakten über bestimmte (Mengen von) Knoten und deren Attributwerte. Er enthält - im Gegensatz zu den im folgenden Abschnitt eingeführten Graphschemata - keine komplexen Formeln für die Festlegung von Integritätsbedingungen etc. Die in ihm "enthaltenen" Knoten und Kanten werden nicht als getrennte Mengen aufgeführt††, sondern nur implizit durch das Vorhandensein bestimmter Primformeln festgelegt. Die als PROGRESS-Datenmodell eingeführten gakk-Graphen sind ein Spezialfall der Σ-Graphen. Das sieht man sehr schön an dem folgenden Beispiel:

---

† Welcher konkrete Ableitungskalkül der Prädikatenlogik erster Stufe verwendet wird, ist für die nachfolgenden Ausführungen unerheblich.

‡ Diese Einschränkung ist ohne praktische Relevanz und vereinfacht den Umgang mit dem prädikatenlogischen Ableitungsbegriff.

††Wie etwa bei der Definition von Hypergraphen (Def. 3.2.1) in Kapitel 3.

Bsp. 4.1.9 Ein "ExpLanguage"-Graph

Sei Σ die Graphsignatur aus Bsp. 4.1.2. Dann sieht die formale Darstellung des Graphen G aus Abb. 4.3 wie folgt aus:

G := { *type*(v1, DefExp), *type*(v2, DeclId), *type*(v3, IntLiteral), … ,
rel(v1, 1stChild, v2), *rel*(v1, 2ndChild, v3), … ,
rel(v3, Value, 8), … ,
type(8, INTEGER), … , } . ■

4.2 Graphschemata und schematreue Graphen

Im vorigen Abschnitt haben wir das Datenmodell der gakk-Graphen eingeführt und die entsprechenden Graphinstanzen als Formelmengen dargestellt. In diesem Abschnitt werden wir nun daran gehen, Regeln für den Aufbau solcher Graphen in Form sogenannter **Graphschemata** aufzustellen. Hierfür werden wir zunächst den entsprechenden Teil der Sprache PROGRESS vorstellen. Anschließend werden wir dann die formale Definition des Begriffs "schematreuer Graph" kurz skizzieren.

Beginnen wir mit der Bereitstellung der **primitiven (Attribut-) Datentypen**, die man bei der Definition des Schemas einer Klasse von Graphen benötigt. Sie werden uns von einem leicht austauschbaren und/oder erweiterbaren "Standard"-Modul zur Verfügung gestellt†. In Abb. 4.4 sehen wir einen Ausschnitt des Imports aller im folgenden benötigten Typen und Funktionen. Der Import der Typen INTEGER und BOOLEAN sowie der zugehörigen Funktionen bedarf wohl keiner weiteren Erläuterungen. Anders ist das mit dem Typ Number. Seine Werte werden zur Identifikation von Knoten verwendet. Deshalb muß auch seine "Pseudofunktion" uniqueId bei jedem Aufruf einen neuen, bislang noch nie verwendeten Wert zurückliefern. Mit ihrer Hilfe werden wir nämlich jedem neu erzeugten Knoten einen eindeutigen Bezeichner zuordnen. Alles weitere hierzu erfahren wir im nachfolgenden Abschnitt 4.4.

Damit ist nur noch zu klären, warum die Bezeichner mancher Funktionen in einfache Hochkommata eingeschlossen sind. Es handelt sich hierbei um genau die Funktionen, deren Aufrufe in Präfix-, Infix- oder Postfixschreibweise‡ ohne Klammerung der Argumente notiert werden. Auf diese Weise können wir Klammerngebirge vermeiden und z.B. arithmetischen Ausdrücken (fast) die allgemein übliche Gestalt geben. Die Begrenzung dieser **Operatoren** genannten Funktionsbezeichner durch einfache Hochkommata hat im übrigen rein technische Gründe, die mit dem Bau eines Parsers für die Sprache PROGRESS zusam-

† Die Implementierung dieses Moduls geschieht in einer konventionellen Programmiersprache.

‡ Einstellige Funktionen kann man wahlweise in Präfix- und Postfix-, zweistellige Operatoren nur in Infixschreibweise verwenden.

```
from STANDARD import
    types Number, (* Wird für die Identifikation von Knoten benötigt. *)
          INTEGER, BOOLEAN;
    functions
       uniqueId:                              -> Number    (* Liefert neue Nummern. *),
       '==':  (Number, Number)                -> BOOLEAN   (* Test auf Identität. *);
    functions
       0:                            -> INTEGER,
       '+':   (INTEGER, INTEGER)     -> INTEGER,
       …
       '<':   (INTEGER, INTEGER)     -> BOOLEAN;
function '!': (Arg: INTEGER) -> INTEGER
    (* Die Fakultät als Beispiel darf nicht fehlen! *)
=
    [  Arg '<' 0 : nil   (* negatives Argument ist nicht erlaubt. *)
    |  Arg '=' 0 : 1
    |  Arg '>' 0 : Arg '*' ( (Arg '-' 1) '!' )  ]
end; (* ! *)
```

Abb. 4.4: Schnittstelle zu den primitiven Attributtypen

menhängen. Die Verwendung solcher Operatoren und damit überhaupt die Möglichkeiten zur Bildung von attributwertigen Ausdrücken in PROGRESS sieht man bei der Definition der Fakultät für den Datentyp INTEGER in Abb. 4.4. Die rekursive Fassung des Operators '!' besteht aus drei **bewachten Alternativen**, aus denen die in Schreibrichtung erste ausgewählt wird, deren **Wächter** erfüllt ist. Somit wird ein Aufruf von '!' wie folgt abgearbeitet:

(1) Zunächst wird überprüft, ob das Argument einen negativen Wert besitzt. In diesem Fall ist das Ergebnis des Aufrufs undefiniert.

(2) War die bewachende Bedingung der ersten Alternative nicht erfüllt, so wird nun überprüft, ob das Funktionsargument gleich 0 ist. In diesem Fall ist das Ergebnis gleich 1.

(3) Ist keine der ersten beiden Wächterbedingungen erfüllt, so wird schließlich die dritte Alternative der Mehrfachverzweigung mit dem rekursiven Aufruf ausgeführt.

Kommen wir nun zur **objektorientierten und rein deklarativen** Beschreibung der wesentlichen Bestandteile einer Klasse von gakk-Graphen, den attributierten, markierten Knoten und den gerichteten, markierten Kanten. "Objektorientiert" wollen wir die entsprechende Teilsprache von PROGRESS deshalb nennen, weil sie die Einteilung von Knoten mit gemeinsamen Eigenschaften in Klassen unterstützt und zudem das Konzept der Bildung von **Unterklassen mit gleichzeitiger Vererbung** von Knoteneigenschaften kennt.

```
node class AST_NODE;
    intrinsic $Id: Number := uniqueId; (* Eindeutiger Knotenbezeichner. *)
end;
node class LEAF_NODE is a AST_NODE end;
    (* Blätter im abstrakten Syntaxbaum. *)
node class PLACEHOLDER is a LEAF_NODE end;
    (* Platzhalter für noch nicht expandierte Teilbäume. *)
node class IDENT is a LEAF_NODE;
    intrinsic Name: String := nil;
end;  (* Deklarierende und angewandte Bezeichner. *)
node class LITERAL is a LEAF_NODE;
    intrinsic Text: String := nil;
end; (* Literale aller Arten, wie z.B. Dezimalkonstanten. *)
node class COMPLEX_NODE is a AST_NODE end;
    (* Alle inneren Knoten eines abstrakten Syntaxbaums. *)
node class UNARY_NODE is a COMPLEX_NODE end;
    (* Knoten mit (mindestens) einem Kind. *)
edge type 1stChild: UNARY_NODE -> AST_NODE;
node class BINARY_NODE is a UNARY_NODE end;
    (* Knoten mit (mindestens) zwei Kindern. *)
edge type 2ndChild: BINARY_NODE -> AST_NODE;

node class APPL_ID is a IDENT end;
        (* Angewandte Auftreten von Bezeichnern. *)
edge type FromApplToDecl: APPL_ID -> DECL_ID;
    (* Bindet angewandte an deklarierende Bezeichner. *)
node class DECL_ID is a IDENT end;
    (* Deklarierende Auftreten von Bezeichnern. *)

node class EXP is a AST_NODE;
    derived Value: INTEGER = nil; (* Berechneter Wert eines Ausdrucks. *)
end;
node class LEAF_EXP is a EXP, LEAF_NODE end;
node class PLACEHOLDER_EXP is a LEAF_EXP, PLACEHOLDER end;
node class APPL_ID_EXP is a LEAF_EXP, APPL_ID end;
node class LITERAL_EXP is a LEAF_EXP, LITERAL end;
```

Abb. 4.5: Schemadefinition abstrakter Syntaxgraphen für Ausdrücke

Darüber hinaus erlaubt uns diese Teilsprache die Formulierung statisch überprüfbarer und/oder dynamisch sicherzustellender **Integritätsbedingungen**. Diese Integritätsbedingungen regeln den Aufbau der Graphen einer bestimmten Klasse. So legen sie unter anderem fest, Kanten welcher Markierung zwischen Knoten welcher Markierungen verlaufen dürfen, und sie definieren funktionale Abhängigkeiten zwischen den Attributen verschiedener Knoten. In Anlehnung an den Begriff "Datenbankschema" nennen wir diese Beschreibung des internen Aufbaus einer Klasse von Graphen "die Festlegung eines **Graphschemas**".

Für die Deklaration eines Graphschemas stellt PROGRESS drei verschiedene Arten von Typdeklarationen zur Verfügung. Knoten- und Kantentypdeklarationen führen die entsprechenden Knoten- und Kantenmarkierungen der zugrunde gelegten Graphstruktur ein, während wir mit Hilfe der Knotenklassendeklarationen eine Vererbungshierarchie über den Knotenmarkierungen errichten können.

Präziser formuliert werden **Knotenklassen** in erster Linie als Bezeichner für Mengen von Knotentypen mit gemeinsamen Eigenschaften verwendet. Sie führen zudem die Konzepte der Klassifikation und der Spezialisierung in unsere Sprache ein und erlauben es, die Duplikation von Deklarationen durch (Mehrfach-) Vererbung[†] entlang einer **"is_a"-Klassenhierarchie** zu ersetzen. So erbt beispielsweise die Knotenklasse EXP alle Eigenschaften der Knotenklasse AST_NODE. Ihre Knoten sind damit nicht nur ein zulässiges Ziel für Kanten der Typen 1stChild etc., sondern besitzen darüber hinaus neben dem neu deklarierten Attribut Value auch das Attribut $Id. Zusätzlich sind Knotenklassen **Typen von Knotentypen** und damit Typen zweiter Ordnung. Sie unterstützen so den kontrollierten Gebrauch von Knotentypparametern in Teilgraphentests und Teilgraphersetzungsregeln (siehe Abb. 4.15). Beispielsweise ist die Klasse EXP aus Abb. 4.5 der Typ all der Knotentypen, deren Knoten unter anderem ein abgeleitetes Value-Attribut besitzen.

Neben der "ExpLanguage"-spezifischen Knotenklasse EXP gibt es eine ganze Reihe weiterer Knotenklassen, die kein Gegenstück in der Syntaxdefinition von Abb. 4.2 besitzen. Mit ihrer Hilfe beschreiben wir ganz allgemein den **Aufbau abstrakter Syntaxgraphen**. So definieren die Deklarationen von AST_NODE, LEAF_NODE etc. zusammen mit den Deklarationen von 1stChild, 2ndChild etc. den Aufbau eines abstrakten Syntaxbaumskeletts. In ihm gibt es zum einen Blattknoten ohne Kinder, wie Konstantenbezeichner oder Zahlliterale. Zum anderen gibt es innere Knoten mit einer festen Anzahl von Kindern. So werden alle Knoten eines Typs der Klasse BINARY_NODE in einem abstrakten Syntaxbaum genau zwei Kinder besitzen, die über auslaufende 1stChild- und 2ndChild-Kanten erreichbar sind. Eine grafische Darstellung eines Ausschnitts unserer Klassenhierarchie zeigt Abb. 4.6.

† Mehrfachvererbung heißt, daß eine Klasse mehr als eine direkte Oberklasse besitzen und deshalb ggf. in Konflikt zueinander stehende Eigenschaften von verschiedenen Oberklassen erben kann.

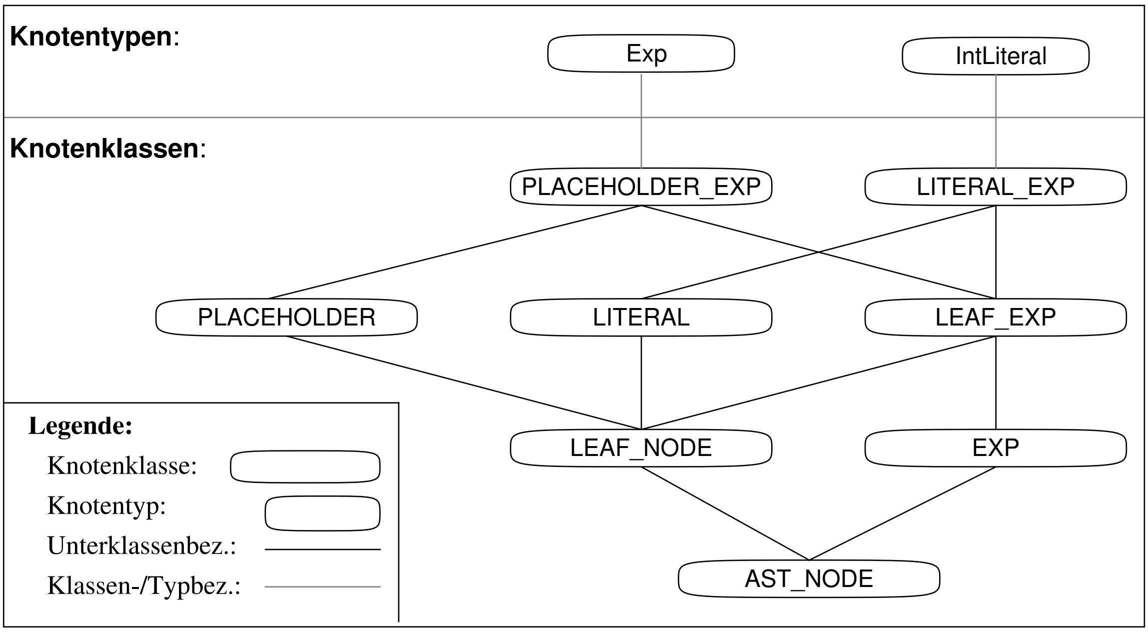


Abb. 4.6: Ausschnitt aus der Klassenhierarchie von “ExpLanguage”

Der Aufbau einer solchen Klassenhierarchie ist gewissen Regeln unterworfen. Nimmt man nämlich zu den explizit deklarierten Knotenklassen eine kleinste Knotenklasse, der keine Knotentypen angehören, und eine größte Knotenklasse, die alle Knotentypen umfaßt, hinzu, so muß diese Familie von Knotentypmengen einen **Verband** bezüglich der üblichen Enthaltenseins-Ordnung auf Mengen bilden. Diese Anforderung erzwingt eine **disziplinierte Verwendung der Mehrfachvererbung** und garantiert, daß zu je zwei beliebigen Klassen genau eine kleinste gemeinsame umfassende Klasse (**Oberklasse**) und genau eine größte gemeinsame enthaltene Klasse (**Unterklasse**) existiert. In Abb. 4.7 sieht man die grafische Darstellung einer Klassenhierarchie, die gegen diese Bedingung verstößt, da für die beiden Klassen PLACEHOLDER_EXP und LITERAL_EXP sowohl die Klasse LEAF_NODE als auch die Klasse EXP eine kleinste gemeinsame Oberklasse ist.

Wenden wir uns nun dem Thema “Attribute und Attributberechnungsregeln” und dem damit eng verbundenen Thema der “Knotentypdeklarationen” zu. Wie bereits in Abschnitt 4.1 angedeutet, gibt es zwei verschiedene Arten von Attributen, nämlich die eigenständigen und die abgeleiteten Attribute. Beginnen wir mit der Erläuterung der **eigenständige Attribute**. Ihr Wert darf nur durch explizite Zuweisungen in Produktionen verändert werden. Typische Vertreter dieser Art sind die Attribute $Id, Name und Text aus Abb. 4.5. So besagt die Deklaration von $Id, daß jeder Knoten eines Typs der Klasse AST_NODE ein entsprechendes Number-wertiges Attribut besitzt. Dieses Attribut wird bereits bei der Erzeugung

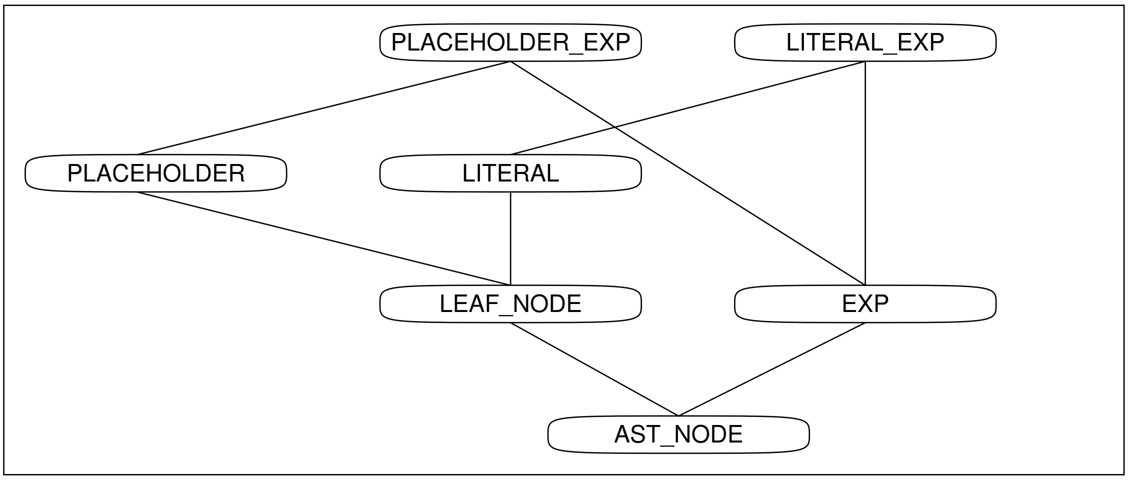


Abb. 4.7: Klassenhierarchie mit Verletzung der Verbandseigenschaft

seines Knotens durch den Aufruf der **Initialisierungsvorschrift** uniqueId mit einer eindeutigen Nummer als Wert versehen. Ähnlich sind die Deklarationen der Attribute Text und Name zu verstehen. Sie besitzen allerdings vor der erstmaligen expliziten Zuweisung einer gewünschten Zeichenfolge keinen definierten Wert.

Die Initialisierungsvorschriften für eigenständige Attribute darf man nicht mit den gerichteten Gleichungen für abgeleitete Attribute, wie etwa Value, verwechseln. Solche gerichteten Gleichungen können - ebenso wie Initialisierungsvorschriften - zusammen mit den Deklarationen ihrer Attribute eingeführt und von dort an die entsprechenden Unterklassen und die zugehörigen Knotentypen **vererbt** werden. Beide Arten von Rechenvorschriften können zudem bei jeder Unterklasse und jedem Knotentyp, der sie erbt, **redefiniert** werden.

Kommen wir zurück zu unserem **abgeleiteten Attribut** Value. Bei seiner Deklaration können wir noch keine sinnvolle Berechnungsvorschrift angeben, da jeder der zu definierenden Knotentypen der Klasse EXP seinen Wert auf eine andere Art und Weise ermitteln muß. Deshalb geben wir die entsprechenden **gerichteten Gleichungen** erst bei der Deklaration der Knotentypen in Abb. 4.8 an. Dort legen wir unter anderem fest, daß Knoten des Typs IntLiteral den Wert ihres Value-Attributes durch die Konvertierung ihres eigenen Text-Attributes erhalten. Das ist der einzige Fall, in dem wir nicht auf die Value-Attribute benachbarter Knoten Bezug nehmen müssen. Für solche Bezugnahmen benötigen wir in allen anderen Fällen sogenannte "**Pfadausdrücke**", die einen Weg im Graphen von einem Knoten zu einem anderen Knoten beschreiben. So liefert beispielsweise der Pfadausdruck 3rdOpd von einem DefExp-Knoten ausgehend den Zielknoten einer auslaufenden 3rdChild-Kante. Und der Pfadausdruck assignedExp "marschiert" von einem ApplId-Knoten ausgehend zunächst über eine auslaufende FromApplToDecl-Kante zu seiner Deklaration, von dort über eine einlau-

```
node type DeclId: DECL_ID end;
    (* Deklarierendes Auftreten eines Bezeichners für ganzzahlige Konstanten. *)
node type Exp: PLACEHOLDER_EXP end;
    (* Platzhalter für nichtexpandierte Ausdrücke. *)
node type IntLiteral: LITERAL_EXP;
    derived rule
        Value = stringToInt(Text of self);
end; (* IntLiteral *)
node type ApplId: APPL_ID_EXP;
    derived rule
        Value = Value of assignedExp;
end; (* ApplId *)
node type DefExp: TERNARY_EXP;
    derived rule
        Value = Value of 3rdOpd;
end; (* DefExpr *)
```

Abb. 4.8: Knotentypen und Attributberechnungsregeln

fende 1stChild-Kante zu dem umfassenden DefExp-Knoten und von dort dann über eine auslaufende 2ndChild-Kante zu dem Ausdruck, der den Wert des ApplId-Knotens festlegt (vgl. mit der Graphstruktur in Abb. 4.3 und der Deklaration der beiden Pfadausdrücke in Abb. 4.9). Auf diese Weise kann man also funktionale Attributabhängigkeiten und damit Datenflüsse entlang der Kanten unserer Graphstruktur deklarativ beschreiben. Mit der Definition der dabei benötigten Pfadausdrücke werden wir uns im folgenden Abschnitt 4.3 auseinandersetzen.

Wenden wir uns schließlich den **Knotentypdeklarationen** selbst zu. Wie wir bereits in den obigen Ausführungen angedeutet haben, werden sie dazu verwendet, das operationale Verhalten einer Menge von Knoten (dieses Typs) eindeutig festzulegen. Während nämlich Knotenklassendeklarationen den zulässigen Kontext eines Knotens sowie seine Attributierung definieren und ggf. Vorschläge für Berechnungsvorschriften enthalten, liefern uns erst die Knotentypdeklarationen eine eindeutige Abbildung von den Knotenmarkierungen und den Attributbezeichnern auf die zugehörigen Attributberechnungsvorschriften. Mit anderen Worten: die in Knotenklassendeklarationen angegebenen Rechenvorschriften können in den zugehörigen Unterklassen und Knotentypen redefiniert werden, während die in Knotentypdeklarationen angegebenen Rechenvorschriften unwiderruflich für alle Knoten dieses Typs gelten.

Bevor wir nun den Begriff "schematreuer Graph" formal definieren, müssen wir zunächst eine grundsätzliche Schwäche der Prädikatenlogik anhand eines Beispiels erläutern.

Bsp. 4.2.1 "Tertium non datur" (Beweis durch Widerspruch)

Sei G := { *type*(v1, ApplId) } ein einknotiger Σ-Graph. Dann wollen wir beweisen, daß sein Aufbau im Widerspruch zu der Forderung steht, daß jeder Knoten mit der Markierung ApplId Quelle einer auslaufenden FromApplToDecl-Kante ist:

$$\forall\, x: \mathit{type}(x, \mathsf{ApplId}) \rightarrow \exists\, y: \mathit{rel}(x, \mathsf{FromApplToDecl}, y)\ .$$

Hierfür müßte man bezogen auf den Knoten v1 zeigen können:

$$G \vdash \neg\ \exists\, y: \mathit{rel}(v1, \mathsf{FromApplToDecl}, y)\ .$$

Hierfür würde man jedoch eine Schlußregel der Form

$$G \nvdash\ \exists\, y: \mathit{rel}(v1, \mathsf{FromApplToDecl}, y) \ \Rightarrow\ G \vdash \neg\ \exists\, y: \mathit{rel}(v1, \mathsf{FromApplToDecl}, y)$$

benötigen. Umgangssprachlich formuliert hieße das: "Ist eine Tatsache nicht beweisbar, so gilt ihr Gegenteil als bewiesen". Klar ist, daß der uneingeschränkte Einsatz einer solchen Ableitungsregel nicht sinnvoll wäre. Beispielsweise würde dann nämlich gelten:

$$G \nvdash \{\ \phi, \neg\phi\ \} \ \Rightarrow\ G \vdash\ \{\ \phi, \neg\phi\ \}\ .$$

Zusammenfassend gesagt, benötigen wir also ein Verfahren, das aus einer Menge von Formeln, die einen Graphen darstellen, zusätzliche Formeln "erschließt", so daß die erweiterte Formelmenge

(4) zwar weiterhin **in sich widerspruchsfrei** aber auch

(5) **hinreichend vollständig** ist, um ihre Schematreue mit Hilfe des normalen Ableitungsbegriffes der Prädikatenlogik (erster Stufe) überprüfen zu können.

In unserem Fall würde uns beispielsweise ein "**Vervollständigungsoperator**" $\mathcal{C}$ genügen, der wie folgt definiert ist:

$$\mathcal{C}(G) := G \cup \{\ \neg\ \exists\, y: \mathit{rel}(v, r, y) \mid \text{es gibt kein } v': \mathit{rel}(v, r, v') \in\ G\ \}\ .$$

Mit seiner Hilfe können wir dann den geforderten Widerspruch herstellen.

$$\mathcal{C}(G) \vdash \neg\ \exists\, y: \mathit{rel}(v1, \mathsf{FromApplToDecl}, y)\ .\ \blacksquare$$

So motiviert, werden wir nun den Begriff "Vervollständigungsoperator" formal definieren und mit seiner Hilfe dann Graphschemata und schematreue Graphen einführen.

Def. 4.2.2 Σ-Graphvervollständigungsoperator

Sei Σ eine Graphsignatur. Dann ist $\mathcal{C}$: $\mathcal{G}_L(\Sigma) \rightarrow \mathrm{FORM}(\Sigma)$ ein Graphvervollständigungsoperator ⇔ es gilt für alle $G \in \mathcal{G}_L(\Sigma)$:

(1) $G \subseteq \mathcal{C}(G)$.

(2) $\mathcal{C}(G)$ ist eine widerspruchsfreie Mengen von Formeln. ■

Def. 4.2.3 Σ-Graphschema

Es sei Σ eine Graphsignatur. Dann wird ein Tupel $S := (\Phi, \mathcal{C})$ Σ-Graphschema genannt (in Zeichen: $S \in \mathcal{S}_L(\Sigma)$) ⇔

(1) $\Phi \in \mathcal{F}_L(\Sigma)$ ist eine widerspruchsfreie Menge von Formeln, die keine (Knoten/ Mengen-) Bezeichner enthalten.

(2) $\mathcal{C}$ ist ein Σ-Graphvervollständigungsoperator. ■

Bsp. 4.2.4 Ein Graphschema für "ExpLanguage"

Sei G der in Abb. 4.3 dargestellte Graph. Dann könnte ein dazu "passendes" Graphschema $S := (\Phi, \mathcal{C})$ in etwa wie folgt aussehen:

$$\begin{aligned}
\Phi := \{ & \mathit{type}(\text{MultExp}, \text{BINARY_EXP_NODE}) , \\
& \forall t: \mathit{type}(t, \text{BINARY_EXP_NODE}) \rightarrow \mathit{type}(t, \text{EXP}) \wedge \mathit{type}(t, \text{BINARY_NODE}) , \\
& \forall\, x, y: \mathit{rel}(x, \text{1stChild}, y) \rightarrow \exists\, t: \mathit{type}(x, t) \wedge \mathit{type}(t, \text{BINARY_NODE}) , \\
& \ldots , \\
& \forall\, x, v: \quad \mathit{type}(x, \text{MultExp}) \wedge \mathit{rel}(x, \text{Value}, v) \\
& \qquad\qquad \rightarrow \exists\, y1, v1, y2, v2: \; \mathit{rel}(x, \text{1stChild}, y1) \wedge \mathit{rel}(y1, \text{Value}, v1) \wedge \\
& \qquad\qquad\qquad\qquad\qquad\qquad \mathit{rel}(x, \text{2ndChild}, y2) \wedge \mathit{rel}(y2, \text{Value}, v2) \wedge \\
& \qquad\qquad\qquad\qquad\qquad\qquad v = v1 * v2 \qquad\qquad\qquad \} .
\end{aligned}$$

$\mathcal{C}(G) := G \cup \ldots$. ■

Def. 4.2.5 Schematreuer Graph

Sei $S := (\Phi, \mathcal{C}) \in \mathcal{S}_L(\Sigma)$. Dann wird ein Σ-Graph $G \in \mathcal{G}_L(\Sigma)$ schematreu bzgl. S genannt (in Zeichen: $G \in \mathcal{G}_L(S)$) ⇔

$$\mathcal{C}(G) \cup \Phi \text{ ist widerspruchsfrei.} \blacksquare$$

4.3 Abgeleitete Relationen und Teilgraphensuche

Im vorausgegangenen Abschnitt haben wir uns in der Hauptsache mit den explizit aufzuführenden Bestandteilen eines Graphen befaßt. Eine Ausnahme hiervon bildeten nur die abgeleiteten Attribute, deren Werte ja nicht explizit in einen Graphen einzutragen sind, sondern implizit durch ein System gerichteter Gleichungen festgelegt werden. Betrachtet man nun Knoten- und Kantentypdeklarationen als ein- und zweistellige Relationen über Knoten (-bezeichnern), die durch die explizite Angabe aller ihrer Instanzen in einem Graphen definiert werden, so fällt einem auf, daß uns bislang Mittel zur Definition entsprechender **abgeleiteter Relationen** fehlen.

Genau für diesen Zweck werden wir nun zwei neue Arten der funktionalen Abstraktion in PROGRESS einführen: die Deklarationen von Pfad- und Restriktionsrelationen, kurz auch Pfade und Restriktionen genannt. **Pfade** und die sie beschreibenden **Pfadausdrücke** werden für uns dabei die Rolle abgeleiteter zweistelliger Relationen auf den Knoten eines Graphen spielen. **Restriktionen** und die sie definierenden (einschränkenden) Ausdrücke unterscheiden sich allein dadurch von Pfaden und Pfadausdrücken, daß sie immer eine Teilmenge der identischen Relation darstellen und sich somit als einstellige Relationen bzw. als Mengen von Knoten in einem Graphen begreifen lassen.

Befassen wir uns zunächst mit den **Pfadausdrücken**. Sie werden in unserer Sprache für die folgende Zwecke eingesetzt:

(1) Ursprünglich wurden sie allein für die Definition komplexer Einbettungsüberführungen benötigt (siehe Kapitel 2).

(2) Als zweites und inzwischen wesentlich bedeutsameres Anwendungsgebiet kam später dann die Definition komplexer Anwendbarkeitsbedingungen für Tests und Produktionen hinzu (siehe Abschnitt 4.4).

(3) Und schließlich werden sie bei der Definition gerichteter Attributgleichungen verwendet (siehe Abb. 4.8 in Abschnitt 4.2).

```
path 1stOpd: UNARY_EXP -> EXP
        (* Liefert zu einem Operator den zugehörigen Operanden, falls es sich dabei    *)
        (* um einen Ausdruck handelt (und nicht um einen deklarierenden Bezeichner).*)
        (* 2ndOpd und 3rdOpd sind analog definiert.                                  *)
=
    -1stChild-> & instance of EXP
end; (* 1stOpd *)
path myOp: AST_NODE -> UNARY_NODE
    (* Liefert zu einem Knoten den umfassenden Operatorknoten. *)
=
    ( <-1stChild- or <-2ndChild- or <-3rdChild- ) & instance of UNARY_EXP
end; (* myOp *)
path assignedExp: APPL_ID_EXP -> EXP
    (* Liefert zu einem angewandten Auftreten die Wurzel des Ausdrucks, der ihm      *)
    (* bei seiner Deklaration zugewiesen wurde.                                       *)
=
    -FromApplToDecl-> & myOp & instance of DefExp & 2ndOpd
end; (* assignedExp *)
```

Abb. 4.9: Einfache Pfadausdrücke

Die in Abb. 4.8 fehlenden Pfaddeklarationen für die Definition von Attributberechnungsregeln und ihre Erläuterung wollen wir nun nachreichen. Beginnen wir mit der ersten Deklaration in Abb. 4.9. Der dort definierte Pfad 1stOpd setzt einen Knoten der Klasse UNARY_EXP in Bezug zu all den Knoten, die mit ihm über eine auslaufende 1stChild-Kante verbunden sind und zusätzlich zu einem Typ der Klasse EXP gehören.

Ähnlich einfach ist die Deklaration des Pfades myOp. Sein Aufruf liefert zu einem Knoten der Klasse AST_NODE den Quellknoten einer (beliebigen) einlaufenden Kante, die vom Typ 1stChild, 2ndChild oder 3rdChild ist. Dieser Kantentraversierung ist eine Typeinschränkung nachgeschaltet, die zum Ausdruck bringt, daß in unseren “ExpLanguage”-Syntaxgraphen die Quellknoten solcher Kanten immer zur Klasse EXP und damit zu ihrer Unterklasse UNARY_EXP gehören. Die Klasse UNARY_EXP ist nämlich die größte gemeinsame Unterklasse der Klassen EXP und UNARY_NODE. Und die Klasse UNARY_NODE ist ihrerseits genau die Klasse all der Knoten, aus denen irgendwelche Kanten der drei oben aufgeführten Typen auslaufen dürfen.

Die Zeichenfolgen -T-> und <-T- stehen also für das Entlanglaufen einer Kante des Typs T von der Quelle zum Ziel und umgekehrt, der Operator & für das Hintereinanderschalten mehrerer Traversierungen, der Operator <u>or</u> für die Zusammenfassung mehrerer gleichwertiger Traversierungsalternativen und der Ausdruck <u>instance of</u> C für die Einschränkung auf Knoten eines Typs der Klasse C.

Damit kommen wir zu assignedExp, der letzten Deklaration in Abb. 4.9. Sie zeichnet alle Knotenpaare (aid, exp) eines Graphen aus, für die gilt:

(1) Der Knoten aid gehört zur Klasse APPL_ID_EXP.

(2) Es gibt einen Knoten did, der über eine einlaufende FromApplToDecl-Kante mit dem Knoten aid verbunden ist.

(3) Es gibt einen Knoten def des Typs DefExp, der gemäß der Pfaddeklaration myOp von dem Knoten did aus erreichbar ist.

(4) Für den Knoten exp muß schließlich gelten, daß er durch die in 2ndOpd beschriebene Graphtraversierung von dem Knoten def aus erreichbar ist.

In dem Graphen aus Abb. 4.3 sind das genau die Knotenpaare (v10, v3) und (v11, v8) .

An dieser Stelle ist ein Einschub zu den **Typverträglichkeitsregeln** der Sprache PROGRESS angebracht. Sie spielen nämlich gerade beim Umgang mit Pfadausdrücken eine sehr wichtige Rolle. Durch eine ziemlich rigide Festlegung des Begriffes “korrekt typisiert” haben wir versucht, den Anwender von PROGRESS vor einer Vielzahl möglicher

Flüchtigkeitsfehler zu schützen und ihn darüber hinaus zur Niederschrift **verständlicher** Pfaddeklarationen zu zwingen. So werden die Ausdrücke

Value of -FromApplToDecl->

und

Value of (-FromApplToDecl-> & 2ndOpd)

beide zu Recht als fehlerhaft betrachtet, da die Zielknoten von FromApplToDecl-Kanten weder ein Value-Attribut besitzen noch Quelle von 2ndChild -Kanten sein können. Darüber hinaus wird aber auch der Ausdruck

Value of (-FromApplToDecl-> & myOp & 2ndOpd)

als fehlerhaft abgelehnt. Aus den Deklarationen des Graphschemas läßt sich nämlich nicht entnehmen, daß in diesem Fall die Quellknoten von 1stChild-Kanten (im Pfad myOp) immer auch Quellknoten von 2ndChild-Kanten (im Pfad 2ndOpd) sind. Deshalb wird vom Anwender erwartet, daß er bei der Formulierung des Ausdrucks durch ein zusätzliches

instance of DefExp

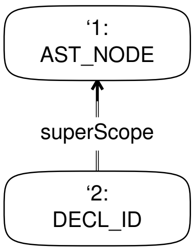
einen anderen Leser der Spezifikation darauf hinweist, Knoten welchen Typs er zwischen den Aufrufen von myOp und 2ndOpd erwartet.[†]

Nach dieser ausführlichen Erläuterung einfach aufgebauter Pfadausdrücke können wir uns nun einem etwas anspruchsvollerem Beispiel zuwenden. Es handelt sich hierbei um die **Definition der Bindungsregeln** für Konstantenbezeichner in “ExpLanguage”-Syntaxgraphen in der Abb. 4.10. Auch dort haben wir wieder versucht, die wirklich sprachspezifischen Anteile der Spezifikation von den Teilen abzutrennen, die für eine größere Klasse blockstruktierter Sprachen Gültigkeit besitzen. So haben wir die Beschreibung der Bindungsregeln auf vier Pfaddeklarationen verteilt, von denen allein die namens superScope “ExpLanguage”-spezifische Details enthält. Sie liefert nämlich zu einem beliebigen Knoten des abstrakten Syntaxbaumskeletts von “ExpLanguage”-Graphen die Wurzel des kleinsten umfassenden Gültigkeitsbereichs.

Der Rumpf der Pfaddeklaration superScope besteht aus zwei ineinandergeschachtelten **Mehrfachverzweigungen**, die wir in ähnlicher Form bereits bei den attributwertigen Ausdrücken in Abb. 4.4 kennengelernt haben. Dort waren allerdings die einzelnen Zweige der Mehrfachverzweigung bewacht. In dem jetzigen Beispiel können wir auf solche Wächter verzichten, da die bewachende Bedingung “mein bewachter Ausdruck liefert mindestens einen definierten Wert” nicht explizit angegeben werden muß. Mit Hilfe der beiden Mehrfachverzweigungen können wir nun die folgenden Fälle unterscheiden:

† Eine formale Definition der Typverträglichkeitsregeln findet man in /Sch 91/.

```
path myDecl: APPL_ID -> DECL_ID
    (* Sucht zu einem angewandten Auftreten seine Deklaration im Syntaxbaum.    *)
=
    visibleDecl(Name of self)
end; (* myDecl *)

path visibleDecl(OfName: String): AST_NODE -> DECL_ID
    (* Liefert zu einem Bezeichnernamen, die dazu passende und an der aktuellen *)
    (* Stelle im Baum sichtbare Deklaration.                                    *)
=
        superScope
 & { not with declInScope(OfName) : superScope }
 & declInScope(OfName)
end; (* visibleDecl *)

path declInScope(OfName: String): AST_NODE -> DECL_ID
    (* Sucht nach einer Deklaration des angegebenen Namens im vorgegebenen     *)
    (* Gültigkeitsbereich.                                                      *)
=
    ‘1 => ‘2 in
```




```
    condition Name of ‘2 ‘=*’ OfName;
end; (* declInScope *)

path superScope: AST_NODE -> AST_NODE
    (* Liefert den kleinsten umfassenden Gültigkeitsbereich zu einem Knoten.   *)
    (* Bei den zweiten Operanden von DefExp-Knoten muß man aufpassen. Ihr      *)
    (* kleinster umfassender Gültigkeitsbereich ist   n i c h t   ihr DefExp-Knoten.   *)
=
    [ <-2ndChild- & superScope          (* Vom zweiten Kind aus kommend muß    *)
                                        (* man in jedem Fall noch höher!       *)
    |   ( <-1stChild- or <-3rdChild- ) (* Vom ersten oder dritten Kind kommend *)
     &  [ instance of DefExp            (* ist man bei einem DefExp-Knoten fertig. *)
        | superScope ] ]
end; (* superScope *)
```

Abb. 4.10: Beispiele für komplexe Pfade

(1) Der aktuelle Knoten ist das zweite Kind eines Operators. Dann ist dieser Operator in keinem Fall die Wurzel des kleinsten umfassenden Gültigkeitsbereichs, selbst wenn es sich um einen Knoten des Typs DefExp handelt. Deshalb steigen wir in diesem Fall durch einen rekursiven Aufruf von superScope im abstrakten Syntaxbaum weiter auf.

(2) Der aktuelle Knoten ist das erste oder dritte Kind eines Operators. Dann müssen wir wiederum mehrere Fälle unterscheiden:

   (2.1) Der umfassende Operator ist ein Knoten des Typs DefExp. Dann haben wir die Wurzel des gesuchten Gültigkeitsbereichs gefunden.

   (2.2) Ist der umfassende Operator kein Knoten des Typs DefExp und scheitert deshalb die Ausführung des ersten Zweiges der Mehrfachverzweigung, so steigen wir in ihrem zweiten Zweig durch einen rekursiven Aufruf von superScope weiter nach oben.

(3) Ist schließlich keiner der beiden Zweige der inneren und damit auch der äußeren Mehrfachverzweigung ausführbar, weil der aktuelle Knoten die Wurzel eines abstrakten Syntaxbaumskeletts ist, so scheitert die Abarbeitung des gesamten Pfadausdrucks.

Damit kommen wir zu der Pfaddeklaration declInScope, die im wesentlichen den Pfad superScope invertiert (zu jedem Pfadausdruck kann man einen inversen Pfadausdruck konstruieren). Sie wurde zur Abwechslung mal als **Teilgraphensuche** und damit grafisch notiert. Bei ihrem Aufruf wird der mit ‘1 bezeichnete Knoten vorgegeben, der die Wurzel eines Gültigkeitsbereichs sein sollte. Zurückgeliefert wird im Erfolgsfall ein mit ‘2 bezeichneter DECL_ID-Knoten, der innerhalb des angegebenen Gültigkeitsbereichs liegt - die Anwendung des als Doppelpfeil notierten Pfades superScope auf den Knoten ‘2 muß den Knoten ‘1 liefern - und dessen Name-Attribut den als Parameter übergebenen Wert OfName trägt.

So gerüstet können wir uns nun daran wagen, den Pfad myDecl zu erläutern. Er legt fest, zwischen welchen angewandten und deklarierenden Auftreten von Bezeichnern FromApplToDecl-Kanten zulässig sind. In seinem Rumpf wird zunächst das Name-Attribut eines vorgegebenen angewandten Auftretens gelesen. Danach wird der Hilfspfad visibleDecl mit dem gelesenen Attributwert als Argument aufgerufen.

In der Pfaddeklaration visibleDecl bestimmen wir dann als erstes den kleinsten umfassenden Gültigkeitsbereich des vorgegebenen Knotens. Anschließend wird der durch geschweifte Klammern eingeschlossene und bewachte Pfad superScope so lange **iteriert**, wie die ihn **bewachende Bedingung**

<u>not</u> <u>with</u> declInScope(OfName)

gültig ist. Mit anderen Worten: wir steigen so lange von Gültigkeitsbereich zu Gültigkeitsbereich nach oben, bis wir einen gefunden haben, der mindestens eine Deklaration mit dem Namen OfName enthält. Haben wir schließlich einen solchen Gültigkeitsbereich erreicht, so liefern wir den Bezeichner einer seiner Deklarationen mit dem gesuchten Namen zurück (in unserem Anwendungsbeispiel gibt es höchsten eine solche Deklaration je Gültigkeitsbereich). Gibt es jedoch keine gültige Deklaration dieses Namens, so wird der Aufruf von superScope bei der Wurzel des äußersten Gültigkeitsbereichs scheitern und damit die Abarbeitung von myDecl erfolglos abgebrochen.

Bevor wir uns nun zu der formalen Definition einer abgeleiteten Relation und insbesondere der dabei verwendbaren Teilgraphensuche zuwenden, wollen wir wenigstens noch ein Beispiel für die Definition einer **abgeleiteten Knotenmenge** vorstellen. Es handelt sich hierbei um die **Restriktion** erroneousExp aus Abb. 4.11. Sie bestimmt die Menge all der Teilbäume in unserem "ExpLanguage"-Syntaxgraphen, die "fehlerhaft" sind. Das sind die Knoten des Typs DivExp, deren zweite Operanden den Wert 0 haben, und die Knoten des Typs ApplId, die an keine Deklaration gebunden sind.

Die formale Definition textuell notierter Pfadausdrücke und Restriktionen ist relativ einfach und wird deshalb hier nur anhand eines Beispiels demonstriert:

Bsp. 4.3.1 Semantik von "erroneousExp"

Mit erroneousExp - wie in Abb. 4.11 definiert - gilt bzgl. der Typisierung dieser Restriktion:

$$\forall x\colon \text{erroneousExp}(x) \rightarrow \exists\, t\colon \mathit{type}(x, t) \wedge \mathit{type}(t, \text{EXP})\ .$$

Und die Ausführungssemantik der Restriktion kann man in dem folgenden prädikatenlogischen Ausdruck festhalten:

$$\begin{aligned}\forall x\colon \text{erroneousExp}(x) \leftrightarrow\ & (\mathit{type}(x, \text{DivExp}) \wedge \exists\, y\colon \text{2ndOp}(x, y) \wedge \mathit{rel}(y, \text{Value}, 0)) \\ \vee\ & (\mathit{type}(x, \text{ApplId}) \wedge \neg \exists\, y\colon \mathit{rel}(x, \text{FromApplToDecl}, y))\ .\ \blacksquare\end{aligned}$$

```
restriction erroneousExp: EXP
   (* Bestimmt alle fehlerhaften Ausdrücke im Graphen. *)
   =
      ( instance of DivExp & valid( Value of 2ndOpd '=' 0 ) )
   or
      ( instance of ApplId & not with -FromApplToDecl-> )
end; (* erroneousExp *)
```

Abb. 4.11: Beispiel einer einfachen Restriktion

Kommen wir nun aber zu der **formalen Definition der Teilgraphensuche**, die wir z.B. in Abb. 4.10 für die Deklaration eines Pfades eingesetzt haben. Hierfür sind einige Vorbereitungen notwendig. Vor allem müssen wir zunächst einmal festlegen, wie wir die Knotenbezeichner des spezifizierten Teilgraphmusters den tatsächlichen Knotenbezeichnern in einem Wirtsgraphen zuordnen, in dem nach unserem Muster gesucht wird. Hierfür können wir leider nicht wie in Kapitel 2 oder Kapitel 3 einfache Funktionen verwenden. Für die Behandlung von Einbettungsüberführungsregeln im nachfolgenden Abschnitt benötigen wir nämlich Mengenbezeichner in den angegebenen Teilgraphmustern (linken Regelseiten), denen wir eine Menge von Knotenbezeichnern eines Wirtsgraphen zuordnen können. Mit anderen Worten, anstelle einer Funktion benötigen wir eine Relation:

Def. 4.3.2 Umbenennungsrelation (Def.-Skizze)

Seien F, F' $\in \mathcal{F}_L(\Sigma)$ und $\mathcal{V}_F$, $\mathcal{W}_F$ bzw. $\mathcal{V}_{F'}$, $\mathcal{W}_{F'}$ seien die Mengen von Knoten-/Mengenbezeichner, die in F bzw. F' Verwendung finden. Dann wird eine Relation

$$u \subseteq (\mathcal{V}_F \times \mathcal{V}_{F'}) \cup (\mathcal{W}_F \times \mathcal{W}_{F'}) \cup (\mathcal{W}_F \times \mathcal{V}_{F'})$$

eine Umbenennungsrelation von F nach F' genannt $\Leftrightarrow$

(1) Für alle $v \in \mathcal{V}_F$: $|u(v)| = 1$ (mit $u(x) := \{ y \mid (x, y) \in u \}$) ;
jeder Knotenbezeichner wird also auf genau einen Knotenbezeichner abgebildet.

(2) Für alle $w \in \mathcal{W}_F$: $u(w) \subseteq \mathcal{W}_{F'}$ und $|u(w)| = 1$ oder $u(w) \subseteq \mathcal{V}_{F'}$;
jeder Mengenbezeichner wird also entweder auf genau einen Mengenbezeichner oder auf eine Menge von Knotenbezeichnern abgebildet.

Mit u* wird die Fortsetzung† von u auf Σ-Terme bzw. Σ-Formeln bezeichnet und es gelte mit $\Phi \in \mathcal{F}_L(\Sigma)$ folgende Schreibabkürzung:

$$u^*(\Phi) := \{ \phi' \in \mathrm{FORM}(\Sigma) \mid \phi \in \Phi \text{ und } (\phi, \phi') \in u^* \} . \blacksquare$$

Bsp. 4.3.3 Umbenennungsrelation

Seien F, F' $\in \mathcal{F}_L(\Sigma)$ mit $\mathcal{V}_F := \{ v1 \}$, $\mathcal{V}_{F'} := \{ v11, v12, v21, v22 \}$, $\mathcal{W}_F := \{ w1, w2 \}$ und $\mathcal{W}_{F'} := \{\}$. Dann gilt mit u := { (v1, v11), (w2, v21), (w2, v22) } als Umbenennungsrelation von F nach F':

u*({ *type*(v1, t) }) = { *type*(v11, t) } .
u*({ *rel*(v1, r, w1) }) = { } .
u*({ *type*(w2, t) }) = { *type*(v21, t), *type*(v22, t) } . ■

† u* setzt zwei Terme oder Formeln zueinander in Beziehung, wenn der eine durch konsistente Ersetzung von Knoten- und Mengenbezeichnern aus dem anderen hervorgeht, wobei ein Bezeichner immer nur solche Bezeichner ersetzen darf, zu denen er bzgl. u in Beziehung steht.

Def. 4.3.4 Σ-(Graph-)Morphismus

Seien F, F' ∈ $\mathcal{F}_L(\Sigma)$. Dann definiert eine Σ-Umbenennungsrelation u von F nach F' einen Σ-Morphismus von F nach F' (in Zeichen: u: F $\tilde{\rightarrow}$ F') ⇔

$$F' \models u^*(F) \ .$$

Sind F, F' ∈ $\mathcal{G}_L(\Sigma) \subseteq \mathcal{F}_L(\Sigma)$ zwei Σ-Graphen, dann wird die Relation u Σ-Graphmorphismus genannt. ■

Satz 4.3.5 Die Kategorie der Σ-Graph(form)en

Sei "∘" die übliche Komposition zweistelliger Relationen†. Dann bildet $\mathcal{F}_L(\Sigma)$ bzw. $\mathcal{G}_L(\Sigma)$ zusammen mit der Menge der oben definierten Σ-(Graph)-Morphismen und "∘" eine Kategorie. ■

Beweis:

Siehe einschlägige Literatur /Sch 91/. ■

Def. 4.3.6 Teilgraph

Seien G, G' ∈ $\mathcal{G}_L(\Sigma)$ zwei Graphen. Dann wird G Teilgraph von G' bzgl. einer Umbenennungsrelation u genannt (in Zeichen: G $\subseteq_u$ G') ⇔

u: G $\tilde{\rightarrow}$ G'. ■

Satz 4.3.7 Äquivalenz der Enthaltenseinsbeziehung und der Teilgrapheigenschaft

Seien G, G' ∈ $\mathcal{G}_L(\Sigma)$. Dann gilt:

G ist Teilgraph von G' bzgl. einer Umbenennungsrelation u

⇔ die Formelmenge u*(G) ist in G' enthalten.

Beweis:

$$\begin{array}{ll} & G \subseteq_u G' \\ \Leftrightarrow_{\text{wg. Def. 4.3.6}} & u: G \tilde{\rightarrow} G' \\ \Leftrightarrow_{\text{wg. Def. 4.3.4}} & G' \models u^*(G) \\ \multicolumn{2}{l}{\Leftrightarrow_{u^*(G) \text{ und } G' \text{ sind Mengen von Primformeln ohne "="}}} \\ & u^*(G) \subseteq G' \text{ (als Formelmengen).} \ ■ \end{array}$$

Def. 4.3.8 Teilgraph unter Nebenbedingungen

Sei S := (Φ, $\mathcal{C}$) ein Σ-Graphschema sowie G, G' ∈ $\mathcal{G}_L(S)$ und F ∈ $\mathcal{F}_L(\Sigma)$ eine Menge von Nebenbedingungen, deren Formeln ausschließlich Knoten-/Mengenbezeichner von G enthalten. Dann wird G Teilgraph von G' bzgl. einer Umbenennungsrelation u und unter den Nebenbedingungen F genannt (in Zeichen: G $\subseteq_{u,F}$ G') ⇔

† (u ∘ u')(x, z) ⇔ es gibt ein y: u(x, y) und u'(y, z)

$$\begin{tikzcd}
F \cup G \arrow[d, "u"'] & G \arrow[l, "id"'] \arrow[d, "u"] \\
\mathcal{C}(G) \cup \Phi & G' \arrow[l, "id"']
\end{tikzcd}$$

Abb. 4.12: Teilgraphensuche mit Nebenbedingungen

(1) $G \subseteq_u G'$, also $G' \vdash u^*(G)$.

(2) $u: F \cup G \xrightarrow{\sim} \Phi \cup \mathcal{C}(G')$, also $\Phi \cup \mathcal{C}(G') \vdash u^*(F \cup G)$.

Diese beiden Forderungen sind mit

$id: G \xrightarrow{\sim} F \cup G$, da $id^*(G) = G \subseteq F \cup G$ und damit $F \cup G \vdash id^*(G)$ gilt
$id: G' \xrightarrow{\sim} \Phi \cup \mathcal{C}(G')$, da $id^*(G') = G' \subseteq \mathcal{C}(G')$ und damit $\Phi \cup \mathcal{C}(G') \vdash id^*(G')$ gilt

äquivalent zu der Existenz des Diagramms in Abb. 4.12. ■

Bsp. 4.3.9 Teilgraphensuche

Mit dem in Abb. 4.13 dargestellten Ausschnitt einer PROGRESS-Spezifikation bzw. dem folgenden Graphschema $S := (\Phi, \mathcal{C})$ mit

$\Phi := \{ \forall y$: unused(y) $\leftrightarrow \neg \exists x$: *rel*(x, FromApplToDecl, y) , … } und
$\mathcal{C}(G) := G \cup \{ \neg \exists x$: *rel*(x, r, v') | es gibt kein v: *rel*(v, r, v') $\in G \} \cup$ …

sowie einem S-schematreuen Graphen

G := { *type*('1, DefExp), type('2, DeclId), *rel*('1, 1stChild, '2) }

und der Nebenbedingung

F := { unused('2) }

ist G ein Teilgraph des Graphen G' aus Abb. 4.3 mit

u := { ('1, v4), ('2, v5) }.

Mit

G' := { *type*(v4, DefExp), *type*(v5, DeclId), *rel*(v4, 1stChild, v5), … }

und der obigen Definition von $\mathcal{C}$ gilt nicht nur

$\mathcal{C}(G') \cup \Phi \vdash u^*(G)$

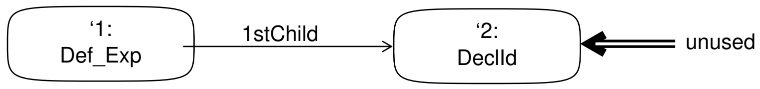
sondern auch

$\neg \exists x$: *rel*(x, FromApplToDecl, v5) $\in \mathcal{C}(G')$.

$\Rightarrow$

$\mathcal{C}(G') \cup \Phi \vdash$ { unused (v5) } . ■

```
restriction unused: DECL_ID =
   not with <-FromApplToDecl-
end; (* unused *)
test UnusedDeclInGraph =
```




```
end; (* UnusedDeclInGraph *)
```

Abb. 4.13: Spezifikation eines komplexen Teilgraphentests

4.4 Schematreue Graphersetzungen

Im Vorangegangenen haben wir alle die Teile von PROGRESS kennengelernt, die die statische Beschreibung attributierter Graphstrukturen und der daraus ableitbaren Eigenschaften unterstützen. Hierzu gehörte unter anderem auch eine grafische Teilsprache für die Definition komplexer Pfadausdrücke, die in diesem Abschnitt den Ausgangspunkt für die Definition der **kleinsten Zugriffsoperationen auf Graphen** bildet. Diese Zugriffsoperationen werden in zwei verschiedene Arten unterteilt:

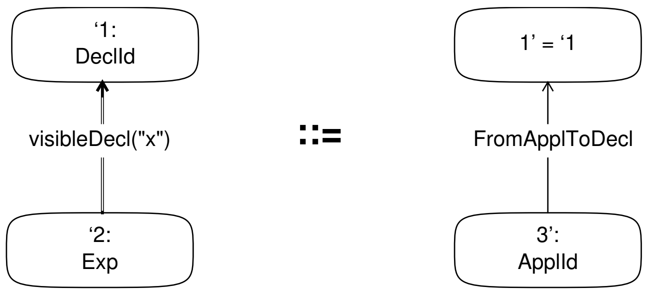
(1) Da gibt es zum einen die rein lesenden Zugriffsoperationen auf attributierte Graphen, **Tests** genannt. Sie suchen nach einem bestimmten Teilgraphen in einem Graphen und verändern im Zuge ihrer Abarbeitung allenfalls über formale Ausgabeparameter die Werte programmlokaler Variablen (in umfassenden Kontrollstrukturen).

(2) Zum anderen gibt es die **Produktionen**† als unteilbare, graphverändernde Operationen, die nach einem Teilgraphen mit bestimmten Eigenschaften in einem Graphen suchen und ihn durch einen neuen Teilgraphen ersetzen. Ebenso wie Tests können auch Produktionen über Ausgabeparameter die Werte temporär existierender, programmlokaler Variablen verändern.

Wie wir im folgenden sehen werden, erlauben uns Tests und Produktionen eine weitgehend **deklarative und grafische Definition** einfacher Zugriffsoperationen auf eine Datenstruktur. Sie stellen damit das Bindeglied zwischen einer rein deklarativen Beschreibung statischer Graphstrukturen und einer mehr oder weniger prozeduralen Beschreibung dynamischer, graphverändernder Abläufe dar, wie wir sie bereits in Kapitel 2 mit den Transaktionen von PROGRESS kennengelernt haben.

† Wir verwenden hier - nicht ganz korrekt - den Begriff “Produktion”, um für die sprachliche Einkleidung einer Teilgraphersetzungsregel des zugrunde gelegten Graphersetzungskalküls einen anderen Begriff als Ersetzungsregel zur Verfügung zu haben.

```
production ExpandToXIdent
    (* Ersetzt einen beliebigen Platzhalter durch den Bezeichner "x". *)
=
```




```
    embedding
        redirect <-1stChild-, <-2ndChild-, <-3rdChild- from '2 to 3';
    transfer Name of 3' := "x";
end; (* ExpandToXIdent *)
```

Abb. 4.14: Auf den Graphen aus Abb. 4.3 anwendbare Produktion

Beide Arten von Operationen sind im übrigen aus der Sicht des Anwenders “**unteilbar**” bzw. “**atomar**”, weil ihre Anwendung

- entweder ohne graphveränderende und variablenverändernde Wirkung scheitert
- oder aber einen schematreuen Graphen direkt in einen anderen schematreuen Graphen überführt.

Für eine kurze Beschreibung der **Wirkungsweise von Produktionen** (und damit auch von Tests) betrachten wir die Anwendung einer Produktion auf den Graphen aus Abb. 4.3. Sie soll einen nicht expandierten Teilausdruck durch ein angewandtes Auftreten des Bezeichners “x” ersetzen und gleichzeitig an die zugehörige Deklaration binden. In der Abb. 4.14 sehen wir die entsprechende Deklaration. Ihre Ausführung erfolgt in fünf Schritten:

(1) **Teilgraphensuche**: Finde ein Paar von Knoten im Wirtsgraphen aus Abb. 4.3, das die Anforderungen aus der linken Seite der Produktion erfüllt. Es wird also nach einem beliebigen Knoten des Typs Exp gesucht, der sich an einer Stelle im abstrakten Syntaxbaumgerüst des Graphen befindet, an dem eine Konstante namens “x” sichtbar ist. Diese Forderung wird in dem Pfadausdruck visibleDecl("x") festgehalten (vgl. mit Abb. 4.10). Er verlangt, daß von dem mit '2 bezeichneten Knoten aus über einen bestimmten Kantenzug ein Knoten des Typs DeclId erreichbar ist, dessen Name-Attribut den Wert "x" trägt. In unserem Beispiel ist (v5, v13) das einzige Knotenpaar, das diese Bedingungen erfüllt.

(2) **Teilgraphersetzung**: Lösche zunächst alle Knoten des gefundenen Teilgraphen, die nicht identisch ersetzt werden sollen, samt aller ein- und auslaufenden Kanten. Füge dann in den Wirtsgraphen Abbilder aller neuen Knoten und Kanten der rechten Seite der Produktion ein. In unserem Beispiel wird also der mit ‘2 bezeichnete Knoten v13 einschließlich der in ihn einlaufenden 1stChild-Kante gelöscht und anschließend der mit 3’ bezeichnete Knoten vom Typ ApplId neu erzeugt. Dieser Knoten wird zudem über eine auslaufende FromApplToDecl -Kante mit dem identisch ersetzten Knoten v5 verbunden, dessen Kontext und Attribute ansonsten unverändert bleiben.

(3) **Einbettungsüberführung**: Modifiziere die Verbindungen des neuen Teilgraphen mit dem verbleibenden Rest des Wirtsgraphen, die sich aus der identischen Ersetzung von Knoten ergeben. Für diesen Zweck kann man - in Abhängigkeit vom Kontext des ersetzten Teilgraphen - einbettende Kanten löschen und ummarkieren bzw. weitere Kanten hinzufügen. So können wir für jede Kante mit der Markierung ...Child, die vor der Ersetzung in den Knoten v13 einlief, eine neue Kante zu dem mit ‘3 bezeichneten neuen Knoten ziehen, wobei Markierung und Quellknoten der alten Kante übernommen werden. In unserem Fall handelt es sich dabei um genau eine 1stChild-Kante.

(4) **Attributtransfer**: Weise den eigenständigen Attributen der Knoten des neu eingesetzten Teilgraphen geeignete Werte zu. In unserem Falle müssen wir das Attribut Name des neu erzeugten Knotens mit dem Wert "x" initialisieren.

(5) **Attributneuauswertung**: Berechne alle abgeleiteten Attribute im Graphen neu, deren Werte - bezogen auf die zugehörigen gerichteten Gleichungen - nun inkonsistent sind. In unserem Fall sind das die Value-Attribute des neu erzeugten Knotens sowie der alten Knoten v12, v4 und v1, die nunmehr einen definierten Wert besitzen.

In den obigen Ausführungen wurde durchgehend der Begriff “**Teilgraph**” (anstelle von “**Untergraph**”) verwendet. Damit sollte darauf hingewiesen werden, daß auch solche Teile eines Wirtsgraphen ersetzt werden können, deren Struktur in der linken Seite der entsprechenden Ersetzungsregel nicht vollständig angegeben ist (vgl. mit der Definition von Teilgraph und Untergraph in Abschnitt 2.1 auf Seite 6). So würde die in Abb. 4.14 aufgeführte Produktion auch solche Knotenpaare ersetzen, die durch zusätzliche Kanten direkt miteinander verbunden sind (was allerdings in diesem Beispiel nie der Fall sein kann). Solche zusätzlichen Kanten würden gemeinsam mit dem Knoten ‘2 aus dem Graphen entfernt werden.

Damit haben wir die Ausführung der Produktion ExpandToXIdent beschrieben. Abschließend sei der Leser zum wiederholten Male auf den potentiell **nichtdeterministischen Charakter** der Teilgraphensuche hingewiesen. Würde nämlich z.B. ein Graph mehrere

Platzhalterknoten enthalten, so müßten wir zusätzliche Vorkehrungen treffen, um ggf. die Anwendung der Produktion auf einen bestimmten Platzhalter im Graphen zu erzwingen.

Eine in der Praxis unerwünschte Vereinfachung der Spezifikation stellt das gleichzeitige Eintragen und Binden der Konstanten “x” dar. Auf diese Weise verhindern wir das Eintragen von Konstantenbezeichnern, zu denen es keine Deklarationen im Graphen gibt. Will man jedoch die Existenz von ungebundenen angewandten Auftreten in Ausdrücken und damit in den zugehörigen Syntaxgraphen zulassen, so muß man die Produktion ExpandToXIdent in zwei Teile zerschlagen. Durch geeignete **Kontrollstrukturen** müßten wir dann dafür sorgen,

- daß zunächst nur die Konstante “x” anstelle des entsprechenden Platzhalters in den Graphen eingetragen wird,
- daß dann versucht wird, sie an die kleinste umfassende Deklaration desselben Namens zu binden,
- und daß schließlich, falls keine passende Deklaration existiert, je nach Wunsch entweder die gesamte Graphtransformation ohne graphverändernde Wirkung abgebrochen wird oder aber der neue Teilausdruck als fehlerhaft markiert wird.

Als letztes muß man an der Produktion ExpandToXIdent noch kritisieren, daß sie sich nur zur Erzeugung eines angewandten Auftretens des Bezeichners “x” eignet. Im Rest dieses Abschnitts werden wir nun einen Ausschnitt einer PROGRESS-Spezifikation vorstellen, die allgemein verwendbare Tests und Produktionen für das syntaxgesteuerte Edieren von “ExpLanguage”-Graphen zur Verfügung stellt und die eben dargestellten Schwächen vermeidet.

In Abb. 4.15 sehen wir erste Beispiele für sehr einfach aufgebaute Tests und Produktionen unserer Spezifikation, die aufgrund des Einsatzes **formaler Parameter** keine “ExpLanguage”-spezifischen Bestandteile enthalten:

(1) Der Test ASTIsOfType überprüft, ob es in einem Graphen einen Knoten mit der Bezeichnernummer Current und dem Typ CurrentType gibt (das Präfix “AST” steht im übrigen für “**A**bstract **S**yntax **T**ree”). Dabei ist Current ein attributwertiger Eingabeparameter und CurrentType ein knotentypwertiger Eingabeparameter.

(2) Die Produktion ASTCreateRoot erzeugt einen Platzhalterknoten für die Wurzel eines neuen abstrakten Syntaxbaumskeletts im Graphen. Sie liefert zudem in dem attributwertigen Ausgabeparameter NewCurrent den initialen Wert des $Id-Attributs des neu erzeugten Wurzelknotens zurück (siehe Initialisierungsvorschrift in Abb. 4.6).

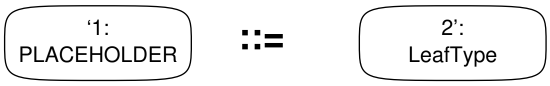
```
test ASTIsOfType  < CurrentType: AST_NODE >
                  ( Current: Number )
   (* Hier wird überprüft, ob der Knoten vom richtigen Typ ist. *)
=
     ‘1:
     CurrentType

   condition
      $Id of ‘1 ‘==’ Current;
end; (* ASTIsOfType *)

production ASTCreateRoot  < RootType: PLACEHOLDER >
                          ( out NewCurrent: Number )
   (* Erzeugt die Wurzel eines neuen Syntaxbaumes im Graphen. *)
=
     ε ::=     1’:
               RootType

   return
      NewCurrent := $Id of 1’;
end; (* ASTCreateRoot *)

production ASTChangeIdent ( Current: Number; NewName: String )
   (* Weist einem Bezeichner einen neuen Wert für sein Namensattribut zu. *)
=
     ‘1:       ::=     1’ = ‘1
     IDENT

   condition
      $Id of ‘1 ‘==’ Current;
   transfer
      Name of 1’ := NewName;
end; (* ASTChangeIdent *)
```

Abb. 4.15: Parametrisierte Tests und Produktionen

(3) Und die Produktion ASTChangeIdent weist dem Name-Attribut eines bestimmten Knotens einen neuen Wert zu. Auch dieser Knoten wird über sein $Id-Attribut im Graphen identifiziert.

Damit kommen wir bereits zur Abb. 4.16 mit einem weiteren Beispiel einer Produktion für den Aufbau abstrakter Syntaxbaumskelette in einem Graphen. Sie ersetzt einen angegebenen Platzhalterknoten durch einen Blattknoten. Der Typ dieses Knotens wird wiederum

```
production ASTExpandToLeafNode    < LeafType: LEAF_NODE >
                                  ( Current: Number; out NewCurrent: Number )
    (* Ersetzt einen Platzhalter durch einen Knoten mit einem Kind. *)
=
```




```
    condition
        $Id of ‘1 ‘==’ Current;
    embedding
        redirect <-1stChild-, <-2ndChild-, <-3rdChild- from ‘1 to 2’;
    return
        NewCurrent := $Id of 2’;
end; (* ASTExpandToLeafNode *)
```

Abb. 4.16: Produktion mit Einbettungsüberführungsregel

als Parameter (zur Laufzeit) übergeben. Trotzdem garantieren auch hier die **Typverträglichkeitsregeln** von PROGRESS, daß das Ergebnis der Anwendung von ASTExpandToLeafNode immer ein schematreuer Graph ist. Ein Knoten eines Typs der Klasse LEAF_NODE darf ja Ziel all der Kanten sein, die in der Einbettungsüberführungsregel der Produktion angegeben sind. Verboten wäre hingegen die folgende Einbettungsüberführungsregel:

redirect -1stChild-> from ‘1 to 2’.

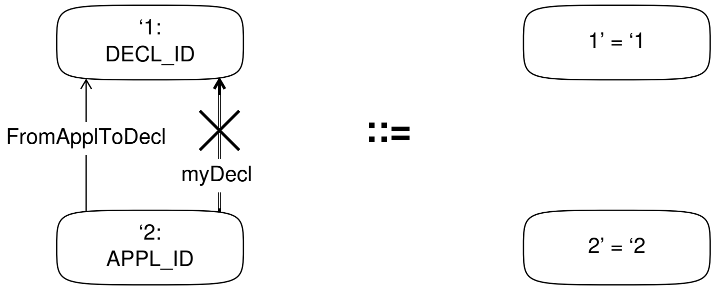
Denn zum einen kann aus einem Knoten der Klasse PLACEHOLDER nie eine Kante des Typs 1stChild auslaufen, und zum anderen darf aus einem LEAF_NODE-Knoten eine solche Kante auch nicht auslaufen. Somit wäre der Versuch, alle auslaufenden 1stChild-Kanten am Knoten ‘1 auf den Knoten 2’ zu übertragen, ziemlich sinnlos.

Doch nun zurück von den Typverträglichkeitsregeln und den Einbettungsüberführungsregeln zu der nächsten Gruppe von Produktionen, die wir hier betrachten wollen. Sie findet man in Abb. 4.17. Mit ihrer Hilfe wollen wir den Einsatz **komplexer Anwendbarkeitsbedingungen** vorführen. Die beiden ersten Produktionen benötigen wir für die Aktualisierung von Bezeichnerbindungen in zwei Schritten:

Die Produktion TruncateBindingOfWrongBoundAppl sucht nach jeweils einem falsch gebundenen angewandten Auftreten und löscht dessen Bindungskante. Das betrifft genau die Knotenpaare, die zwar durch eine FromApplToDecl-Kante aber nicht durch den Pfadausdruck myDecl miteinander verbunden sind.

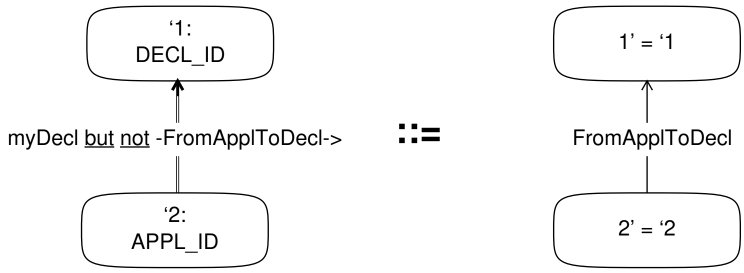
Die Produktion BindUnboundAppl sucht nach jeweils einem ungebundenen angewandten Auftreten, zu dem eine passende Deklaration existiert, und bindet es. Das betrifft

production TruncateBindingOfWrongBoundAppl
(Hebt die Bindung eines falsch gebundenen angewandten Auftretens auf. *)*
=



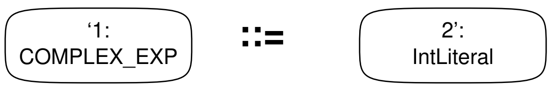
end; *(* TruncateBindingOfWrongBoundAppl *)*

production BindUnboundAppl
(Bindet ein ungebundenes angewandtes Auftreten mit passender Deklarationr*)*
=



end; *(* BindUnboundAppl *)*

production SimplifyExpWithDefinedValue
(Achtung: Das Löschen der Kinder von ‘1 fehlt. *)*
=



condition
def(Value of ‘1);*(* Überflüssige Abfrage, da bei undefiniertem Wert*)*
(der folgende Attributtransfer sowieso scheitert.*)*
embedding
redirect <-1stChild-, <-2ndChild-, <-3rdChild- from ‘1 to 2’;
transfer
Text of 2’ := mkIntToString(Value of ‘1);
end; *(* SimplifyExpWithDefinedValue *)*

Abb. 4.17: Produktionen mit komplexen Anwendbarkeitsbedingungen

genau die Knotenpaare, die zwar durch keine FromApplToDecl-Kante aber dafür durch den Pfadausdruck myDecl miteinander verbunden sind.

Die letzte Deklaration in Abb. 4.17 ist das erste Beispiel einer “ExpLanguage”-spezifischen Produktion. Sie sucht nach einem beliebigen Teilausdruck mit definiertem Wert und ersetzt ihn durch eine entsprechende Konstante, ohne jedoch die Kinder (und Enkelkinder und ...) des mit ‘1 bezeichneten Knoten zu löschen (das müßte man ggf. durch die wiederholte Anwendung weiterer Produktionen erledigen). Die **Attributierungsbedingung**

def(Value of ‘1)

bedarf wohl einer kurzen Erklärung. In ihr fordern wir allein aus Gründen der Lesbarkeit, daß der mit ‘1 bezeichnete Knoten ein definiertes Value-Attribut besitzt. Überflüssig ist diese Forderung deshalb, weil bei einem undefinierten Value-Attribut spätestens die Zuweisung an das Text-Attribut des neuen Knotens 2’ scheitern würde. Damit würde aber auch die Anwendung der Produktion selbst auf den ausgewählten Knoten ohne graphverändernde Wirkung scheitern und stattdessen mit einer hoffentlich besser geeigneten Anwendungsstelle ein erneuter Versuch gestartet werden (falls es noch eine weitere mögliche Anwendungsstelle gibt).

Kommen wir nun zur formalen Definition von Graphersetzungsregeln, die angewendet auf einen schematreuen Graphen nur wiederum einen schematreuen Graphen erzeugen können†.

Def. 4.4.1 Schematreue (Teil-) Graphersetzungsregel

Sei $S \in \mathcal{S}_L(\Sigma)$ ein Graphschema. Dann wird ein Quadrupel $r := (AL, L, R, AR)$ mit AL, $AR \in \mathcal{F}_L(\Sigma)$ sowie mit $L, R \in \mathcal{G}_L(\Sigma)$ eine schematreue (Teil-) Graphersetzungsregel genannt (in Zeichen: $r \in \mathcal{R}_L(\Sigma)$) $\Leftrightarrow$

(1) Die Menge der Anwendbarkeitsbedingungen (Vorbedingungen) AL enthält nur Knoten- und Mengenbezeichner aus der linken Regelseite L.

(2) Die Menge der Anwendbarkeitsbedingungen (Nachbedingungen) AR enthält nur Knoten- und Mengenbezeichner aus der rechten Regelseite R. ■

† Das läßt sich für die hier vorgestellte Fassung der Sprache PROGRESS allein durch statische Überprüfungen auf den Ersetzungsregeln selbst sicherstellen; i.a. müssen jedoch Laufzeitüberprüfungen durchgeführt werden, um die Schematreue eines erzeugten Graphen zu gewährleisten.

Bsp. 4.4.2 Die Teilgraphersetzungsregel “ExpandToXIdent”

Betrachten wir die formale Definition der Produktion ExpandToXIdent aus Abb. 4.14, die alle wesentlichen Bestandteile von PROGRESS-Produktionen enthält, aber keine formalen Ein- bzw. Ausgabeparameter besitzt†.

Um auch ein Beispiel für eine Nachbedingung zu haben, wollen wir zusätzlich fordern, daß der Teilausdruck, an den "x" gebunden wird, und damit das angewandte Auftreten selbst, nach der Ersetzung einen definierten Wert besitzt‡. Damit ergibt sich folgende Definition der Ersetzungsregel r := (AL, L, R, AR) mit:

(1) AL := { visibleDecl("x", ‘1, ‘2) } .

(2) L := { *type*(‘1, DeclId), *type*(‘2, Exp),
rel(w1, 1stChild, ‘2), *rel*(w2, 2ndChild, ‘2), *rel*(w3, 3rdChild, ‘2) } .

(3) R := { *type*(‘1, DeclId), *type*(3’, ApplId), *rel*(3’, FromApplToDecl, ‘1),
rel(w1, 1stChild, 3’), *rel*(w2, 2ndChild, 3’), *rel*(w3, 3rdChild, 3’),
rel(3’, Name, "x") } .

(4) AR := { ∃ v: *rel*(3’, Value, v) } .

Dabei seien w1, w2 und w3 Mengenbezeichner. Sie werden im Zuge der Teilgraphensuche einer maximalen und ggf. leeren Menge von Knoten des Wirtsgraphen zugeordnet. Auf diese Weise lassen sich alle in den mit ‘2 bezeichneten Knoten vor der Ersetzung einlaufenden “Baum”-Kanten auf den neu zu erzeugenden umd mit 3’ bezeichneten Knoten umlenken. ■

Damit haben wir angedeutet, wie man sich die Übersetzung von PROGRESS-Tests und Produktionen in die Teilgraphersetzungsregeln unseres logikbasierten Kalküls vorzustellen hat. Allerdings haben wir hierbei sowohl das Problem der **Parametrisierung** von Ersetzungsregeln als auch das Problem des **Löschens von Knoten** mit unbekanntem Kontext oder unbekannten Attributen ausgeklammert. Ersteres erfordert die Erweiterung der formalen Definition von Teilgraphersetzungsregeln um Mengen von Ein- und Ausgabeparametern und die Berücksichtigung dieser Mengen bei der Definition des entsprechenden Ableitungsbegriffes††.

Das von den kategorientheoretischen Ansätzen aus Kapitel 3 bereits bekannte “**dangling edge**”-Problem läßt sich jedoch ganz einfach durch den Einsatz von Mengenbe-

† Auf die formale Behandlung parametrisierter Ersetzungsregeln wollen wir hier aus Aufwandsgründen verzichten.

‡ Solche Nachbedingungen können fast immer in entsprechende Vorbedingungen umformuliert werden. Deshalb unterstützt PROGRESS die Definition von Nachbedingungen nicht, ist also etwas weniger allgemein als die hier vorgestellte formale Definition von Ersetzungsregeln.

††Näheres hierzu in /Sch 91/.

zeichnern aus der Welt schaffen. So genügt es, die linke Seite L einer Ersetzungsregel für jeden zu löschenden Knoten v und jede potentiell aus ihm auslaufende Kante eines Typs r (mit einlaufenden Kanten und Attributen verfährt man analog) um eine Formel der Gestalt

$$rel(\mathrm{v}, \mathrm{r}, \mathrm{w})$$

mit einem Mengenbezeichner w zu erweitern. Damit werden im Zuge einer Regelanwendung nicht nur der Knoten v sondern auch alle ihn verlassenden r-Kanten gelöscht.

Def. 4.4.3 Schematreue Teilgraphersetzung

Sei $S := (\Phi, \mathcal{C}) \in \mathcal{S}_L(\Sigma)$ u. $G, G' \in \mathcal{G}_L(S)$. Dann gilt mit $r := (AL, L, R, AR) \in \mathcal{R}_L(S)$: der Graph G' läßt sich vermittels r aus G ableiten (in Zeichen: $G \sim r \rightsquigarrow G'$) $\Leftrightarrow$

(1) Es gibt einen Morphismus $u: L \xrightarrow{\sim} G$ mit: $L \subseteq_{u,AL} G$
(der durch u ausgewählte Teilgraph in G erfüllt die Vorbedingungen AL).

(2) Es gibt keinen Morphismus $\hat{u} : L \xrightarrow{\sim} G$ mit: $L \subseteq_{\hat{u}, AL} G$ und $u \subset \hat{u}$
(u ist also maximal bzgl. der Zuordnung von Knotenbezeichner in G zu Mengenbezeichner in L).

(3) Es gibt einen Morphismus $w: R \xrightarrow{\sim} G'$ mit: $R \subseteq_{u, AR} G'$
(der durch w ausgewählte Teilgraph in G' erfüllt die Nachbedingungen AR).

(4) Der Morphismus w bildet jeden Knotenbezeichner (Mengenbezeichner) aus R, der nicht in K bzw. L auftritt, auf einen Knotenbezeichner (Mengenbezeichner) in G' ab, der nicht in H bzw. in G auftritt.

(5) Mit $K := L \cap R$ gilt:
$v := \{ (x, y) \in u \mid x \text{ ist Bezeichner in } K \} = \{ (x, y) \in w \mid x \text{ ist Bezeichner in } K \}$
(auf dem identisch zu ersetzenden Anteil K stimmen u und w überein).

(6) Es gibt $H \in \mathcal{G}_L(\Sigma)$: $G \setminus (u^*(L) \setminus v^*(K)) = H = G' \setminus (w^*(R) \setminus v^*(K))$
(H, der gemeinsame Anteil von G und G', muß nicht schematreu sein). ■

Die obige Definition einer schematreuen Teilgraphersetzung beinhaltet bereits ein konstruktives Verfahren für die Anwendung einer Regel r auf einen Graphen G. Mit den Bezeichnern wie oben gilt nämlich:

(1) Durch einen Morphismus u wird ein zu L und AL passender, maximaler Teilgraph in G ausgesucht.

(2) Dann wird aus dem Durchschnitt von L und R der identisch zu ersetzende Anteil K der Ersetzungsregel gebildet.

$$\begin{tikzcd}
AL \cup L \arrow[d, "u"'] \arrow[dr, phantom, "(1)"] & L \arrow[l, "id"'] \arrow[d, "u"'] \arrow[dr, phantom, "(2)"] & K \arrow[l, "id"'] \arrow[r, "id"] \arrow[d, "v"] \arrow[dr, phantom, "(3)"] & R \arrow[r, "id"] \arrow[d, "w"] \arrow[dr, phantom, "(4)"] & R \cup AR \arrow[d, "w"] \\
\mathcal{C}(G) \cup \Phi & G \arrow[l, "id"'] & H \arrow[l, "id"'] \arrow[r, "id"] & G' \arrow[r, "id"] & \mathcal{C}(G') \cup \Phi
\end{tikzcd}$$

Abb. 4.18: Teilgraphersetzungen als kommutierende Diagramme

(3) Im nächsten Schritt wird aus G das Abbild von L ausschließlich aller identisch zu ersetzenden Anteile entfernt, also $H := G \setminus (u^*(L) \setminus v^*(K))$ gebildet; v ist hierbei die Einschränkung von u auf die Bezeichner in K.

(4) Anschließend wird v so zu einer Relation w erweitert, daß jeder neue Bezeichner in R auf genau einen neuen Bezeichner abgebildet wird, der nicht in H bzw. in G auftritt.

(5) Damit kann man dann $G' := H \cup (w^*(R) \setminus v^*(K))$ konstruieren.

(6) Abschließend muß man überprüfen, ob das konstruierte G' schematreu ist und die Nachbedingungen aus AR erfüllt.

Neben diesem expliziten Konstruktionverfahren gibt es noch eine weitere Möglichkeit, die Anwendung einer schematreuen Graphersetzungsregel mit Vor- und Nachbedingungen formal darzustellen:

Satz 4.4.4 Diagrammartige Darstellung der Teilgraphersetzung

Mit den Bezeichnungen wie in Def. 4.4.3 und G ~ r ~> G' läßt sich das Diagramm aus Abb. 4.18 mit den kommutierenden Teildiagrammen (1) bis (4) konstruieren.

Beweis:

Auf die Existenz der Teildiagramme (1) und (4) wurde bereits in der Def. 4.3.8 hingewiesen. Für die Existenz der Teildiagramme (2) und (3) genügt es zu zeigen, daß das in Schritt (5) aus Def. 4.4.3 konstruierte v ein Morphismus von K nach H ist (dann gilt per definitionem: $id \circ u = v \circ id = id \circ w$). Für den Beweis dieser Tatsache starten wir mit Def. 4.4.3, Bed. (1):

$$u: L \overset{\sim}{\rightarrow} G$$

$\Leftrightarrow$ Def. 4.3.6, Satz 4.3.7 u. $K \subseteq L$

$$u^*(K) \subseteq u^*(L) \subseteq G$$

$\Rightarrow$ Bed. (5) aus Def. 4.4.3

$v^*(K) \subseteq G$

$\Rightarrow_{\text{einfache Umformung}}$

$v^*(K) \setminus (u^*(L) \setminus v^*(K)) \subseteq G \setminus (u^*(L) \setminus v^*(K))$

$\Leftrightarrow_{\text{einfache Umformung}}$

$v^*(K) \subseteq G \setminus (u^*(L) \setminus v^*(K))$

$\Leftrightarrow_{\text{Bed. (6) aus Def. 4.4.3}}$

$v^*(K) \subseteq H$

$\Leftrightarrow_{\text{Def. 4.3.6 u. Satz 4.3.7}}$

$v: K \xrightarrow{\sim} H$. ■

Diese Darstellung einer Teilgraphersetzung als eine Aneinanderreihung kommutierender Diagramme erinnert sehr an die kategorientheoretische Definition aus Kapitel 3. Leider sind die zentralen Eigenschaften des kategorientheoretischen Graphersetzungsbegriffes hier nicht erfüllt. So besitzen die obigen Teildiagramme i.a. nicht die Pushout-Eigenschaft und auch die Invertierbarkeit der Ersetzungsregeln (vgl. mit Satz Satz 3.3.9 aus Kapitel 3) ist in aller Regel nicht gegeben. Beides steht im ursächlichen Zusammenhang mit der Einführung von Mengenbezeichnern und der Verwendung von Relationen anstelle von Funktionen als Morphismen. Somit haben wir einen **Zugewinn an Ausdrucksstärke** (in punkto Löschen von Knoten mit unbekanntem Kontext und Einbettungsüberführungsregeln) mit dem **Verzicht auf theoretische Eigenschaften** erkauft.

Def. 4.4.5 Schematreue sequentielle Ableitbarkeit

Es sei $\mathcal{R} \subseteq \mathcal{R}_L(S)$ sowie G, G' $\in \mathcal{G}_L(S)$. Dann heißt G' vermittels $\mathcal{R}$ sequentiell ableitbar aus G (in Zeichen: G ~ $\mathcal{R}$~> G') $\Leftrightarrow$

$$\exists\, r \in \mathcal{R}:\ G \sim r \sim> G'.$$

Der transitive, reflexive Abschluß von ~> wird mit $\overset{*}{\sim>}$ bezeichnet. ■

Def. 4.4.6 Schematreues (Teil-) Graphersetzungssystem

Sei Σ eine Graphsignatur und S $\in \mathcal{S}_L(\Sigma)$ ein Σ-Graphschema. Dann wird ein Tripel der Form gr = (S, G_S , $\mathcal{R}$) ein schematreues (Teil-) Graphersetzungssystem genannt $\Leftrightarrow$

(1) $G_S \in \mathcal{G}_L(S)$ (G_S ist ein beliebiger schematreuer Startgraph).

(2) $\mathcal{R} \subseteq \mathcal{R}_L(S)$ ($\mathcal{R}$ ist eine Menge von Teilgraphersetzungsregeln). ■

Def. 4.4.7 Schematreue sequentielle Graphsprache

Sei gr = (S , G_S , $\mathcal{R}$) ein schematreues Graphersetzungssystem. Dann ist die von gr erzeugte Sprache folgendermaßen definiert:

$$\mathcal{L}_L(gr) := \{\, G \in \mathcal{G}_L(S) \mid G_S \overset{*}{\sim>} G \,\}\ .$$ ■

Ausgehend von dem damit eingeführten Begriff “schematreues Graphersetzungssystem” könnten wir nun - wie in Kapitel 2 - sogenannte “**programmierte schematreue Graphersetzungssysteme**” einführen. Da wir hierfür jedoch nur die Definitionen aus Abschnitt 2.6 leicht verändert wiederholen müßten, wollen wir darauf verzichten. Stattdessen beschließen wir diesen Abschnitt lieber mit einem kleinen Beispiel für die Deklaration programmierter Graphersetzungen mit Hilfe der Transaktionen von PROGRESS (auch sie haben wir bereits im Abschnitt 2.6 kennengelernt).

Den Anfang unseres Beispiels in Abb. 4.19 bildet die Transaktion ExpandToXIdent, der als formaler Eingabeparameter ein Verweis auf den zu ersetzende Platzhalter übergeben wird und die als Ausgabeparameter einen Verweis auf den dafür eingesetzten Bezeichner liefert. Sie überprüft zunächst, ob der übergebene Verweis tatsächlich auf einen Exp-Platzhalter zeigt. Dann ersetzt sie ihn durch einen Knoten des Typs ApplId, weist dessen Name-Attribut den Wert "x" zu und stößt dabei das Aktualisieren der Bezeichnerbindungen an.

Die für das **Aktualisieren von Bezeichnerbindungen** zuständige und in BSAChangeIdent[†] eingesetzte Transaktion UpdateBindings wurde so abgefaßt, daß sie prinzipiell nach einer beliebigen Abfolge syntaxbaumverändernder Graphersetzungen anwendbar ist. Zudem hatten bei ihrer Definition die beiden Anforderungen “Lesbarkeit” und “Abstraktheit” gegenüber der effizienten Ausführbarkeit unbedingten Vorrang. Ihr Rumpf besteht aus einer Schleife, deren Inneres so oft ausgeführt wird, wie das möglich ist. Das bedeutet, daß zunächst die Produktion TruncateBindingOfWrongBoundAppl so lange aufgerufen wird, bis der vorgegebene Graph keine weitere Anwendungsstelle mehr bereitstellt und damit keine inkonsistenten Bindungen mehr enthält. Erst danach wird die Produktion BindUnboundAppl so oft wie möglich aufgerufen und damit nach und nach jede zulässige, aber bislang fehlende Bezeichnerbindung in den Graphen neu eingetragen.

Damit ist die Vorstellung der Sprache PROGRESS und des dazugehörigen logikbasierten Graphersetzungskalküls beendet. Zusammenfassend können wir PROGRESS als eine **streng und statisch typisierte Programmiersprache** bezeichnen, die deklarative und prozedurale Sprachelemente besitzt für

- einen **datenflußorientierten Stil** der Programmierung durch den Einsatz gerichteter Attributgleichungen,
- einen (eingeschränkt) **relationalen Stil** der Programmierung durch die Verwendung textuell notierter Pfadausdrücke, grafisch notierter Teilgraphentests und nichtdeterministischer Kontrollstrukturen,

† “BSA” steht für “Block-Strukturierter Abstrakter Syntaxgraph”.

```
transaction ExpandToXIdent(InCurrent: Number; out OutCurrent: Number)
    (* Definition der früheren Produktion als Aneinanderreihung allgemein ver-     *)
    (* wendbarer Transaktionen.                                                     *)
=
    use Current: Number := InCurrent in
        ASTIsOfType<Exp>(Current)
    &   ExpandToLeafNode<ApplId>(Current, out Current)
    &   ExpChangeIdent(Current, "x")
    &   OutCurrent := Current
    end
end; (* ExpandToXIdent *)

transaction ExpChangeIdent ( Current: Number; NewName: String )
    (* Zunächst wird überprüft, ob der angegebene Knoten vom richtigen Typ ist.    *)
    (* Erst dann wird die geplante Graphveränderung durchgeführt.                   *)
=
    ( ASTIsOfType<DeclId>(Current)  or  ASTIsOfType<ApplId>(Current) )
 &  BSAChangeIdent(Current, NewName)
end; (* ExpChangeIdent *)


transaction BSAChangeIdent ( Current: Number; NewName: String )
    (* Ändert den Namen eines Bezeichners und aktualisiert danach alle ungültig    *)
    (* gewordenen Bezeichnerbindungen.                                              *)
=
    ASTChangeIdent(Current, NewName)
 &  UpdateBindings
end; (* BSAChangeIdent *)
transaction UpdateBindings
    (* Diese Transaktion bringt nach einer beliebigen Veränderung des abstrakten   *)
    (* Syntaxbaums alle Bindungen wieder in Ordnung.                                *)
=
    loop
        TruncateBindingOfWrongBoundAppl
    else
        BindUnboundAppl
    end    (* Wird ausgeführt, solange noch falsch gebundene oder ungebundene      *)
           (* angewandte Auftreten mit passender Deklaration existieren.            *)
end; (* UpdateBindings *)
```

Abb. 4.19: Definition einiger Transaktionen

- einen (eingeschränkt) **objektorientierten Stil** der Programmierung durch den Einsatz von Mehrfachvererbung und dynamischer Bindung von Attributbezeichnern an Berechnungsregeln,
- einen **regelorientierten Stil** der Programmierung durch die Deklaration von Produktionen und die Verwendung nichtdeterministischer Kontrollstrukturen
- und einen **imperativen Stil** der Programmierung durch die Verwendung deterministischer Kontrollstrukturen innerhalb von Transaktionen.

Alles das haben wir versucht, anhand eines **konkreten Beispiels** aus dem geplanten Hauptanwendungsbereich der Sprache PROGRESS vorzuführen. So wurde die logische Internstruktur einfacher Ausdrücke als abstrakter Syntaxgraph modelliert und eine repräsentative Auswahl von Zugriffsoperationen spezifiziert, die man für das syntaxgesteuerte Edieren, Analysieren und Vereinfachen solcher Ausdrücke benötigt.

Für den dabei zugrundegelegten **logikbasierten Graphersetzungsbegriff** gilt, daß er von der Methodik her - soweit möglich - den kategorientheoretischen Ansätzen ähnelt ohne dabei in punkto Ausdruckskraft Abstriche gegenüber den mengentheoretischen Ansätzen in Kauf zu nehmen. Durch die Hinzunahme sogenannter Graphschemata wurde darüber hinaus die Basis für die Definition ableitbarer Grapheigenschaften und “korrekter” (schematreuer) Teilgraphersetzungsregeln geschaffen. Bislang nicht untersucht wurde, ob es - aufgrund der bestehenden Ähnlichkeiten - möglich ist, die Vorgehensweise des kategorientheoretischen Ansatzes bei der Definition paralleler Graphersetzungen auf den logikbasierten Ansatz zu übertragen.

4.5 Literatur

Der hier vorgestellte logikorientierte Graphersetzungsbegriff und seine sprachliche Einkleidung ist erst im Rahmen der Dissertation /Schü 91/ entstanden. Deshalb gibt es zu diesem Kapitel kaum weiterführende Literatur. Eine Ausnahme bilden allenfalls die Dissertation /We 91/, die eine umfangreichere Anwendung der Sprache PROGRESS enthält, und die Dissertation /Le 88/, in der der direkte Vorläufer dieser Sprache definiert wird. Für die hier vorausgesetzten prädikatenlogischen Grundbegriffe sei dem Leser /Schö 89/ empfohlen.

/Le 88/ Lewerentz C.: *Interaktives Entwerfen großer Programmsysteme*, IFB 194, Springer Verlag (1988)

/Schö 89/ Schöning U.: *Logik für Informatiker*, 2. Auflage, BI-Wissenschaftsverlag (1989)

/Schü 91/ Schürr A.: *Operationales Spezifizieren mit programmierten Graphersetzungssystemen*, Deutscher Universitätsverlag (1991)

/We 91/ Westfechtel B.: *Revisions- und Konsistenzkontrolle in einer integrierten Softwareentwicklungsumgebung*, IFB 280, Springer Verlag

4.6 Aufgaben

Aufgabe 4.1 Gefädelte Bäume in PROGRESS

(1) Erstellen Sie eine PROGRESS-Spezifikation für die gefädelten Binärbäume aus Aufgabe 2.4, die folgende Punkte erfüllt:

- Es soll eine Create-Produktion geben, die den Wurzelknoten für den leeren Baum erzeugt.
- Der einzutragende neue Wert soll als Parameter übergeben werden.
- Die Produktionen suchen sich selbst genau die richtige Einfügestelle. Sie sollen nur dann anwendbar sein, wenn die richtige Situation vorliegt.

(2) Erweitern Sie die Spezifikation aus (1) um eine Produktion, die den Graphen unverändert läßt, aber nur dann anwendbar ist, wenn der übergebene Wert im aktuellen Baum bereits vorhanden ist.

Versuchen Sie möglichst viele Embedding-Anweisungen einzusparen und durch identisch ersetzte Knoten zu ersetzen.

Aufgabe 4.2 Gefädelte Bäume in PROGRESS; Die Zweite

Schreiben Sie mit Hilfe der Produktionen aus Aufgabe 4.1 eine Transaktion, die ganz allgemein einen übergebenen Wert in einen existierenden Baum einträgt, aber nur, wenn der Wert in dem Baum noch nicht enthalten ist. D.h., die Transaktion muß erst testen, ob der Wert im Baum schon vorhanden ist. Falls nicht, muß links angefügt werden oder rechts angefügt werden oder das erste Element eingefügt werden.

Aufgabe 4.3 Gefädelte Bäume in PROGRESS; Die Dritte

Erweitern Sie die Spezifikation für gefädelte Binärbäume aus Aufgabe 4.1. Alle Baumknoten sollen ein Attribut ’Hight’ besitzen, das die Höhe dieses Knotens im Baum angibt (Blätter haben die Höhe 1).

Aufgabe 4.4 Logigkalkül

Gegeben sei folgendes Graphschema in PROGRESS-Notation und durch die Formelmenge Phi:

```
node_class PERSON end;
node_class MAN is_a PERSON end;
node_class WOMAN is_a PERSON end;
node_type Rogue : MAN end;
node_type Gentleman : MAN end;
node_type Lady : WOMAN end;
```

```
Phi = {
∀ t: type( t, MAN) <–> (   t = Rogue
                          ∨ t = Gentleman),
∀ t: type( t, WOMAN) <–> t = Lady,
∀ t: (type( t, MAN) ∨ type( t, WOMAN))
              –> type( t, PERSON)
}
```

(1) Erweitern Sie obige Spezifikation und das Schema Phi um einen Kantentyp Husband, der von Knoten der Klasse MAN nach WOMAN läuft. Stellen Sie in Phi zusätzlich sicher, daß eine solche Kante immer nur genau einen MAN mit genau einer WOMAN verbindet.

(2) Gegeben sei folgender Graph G:

Erweitern Sie Phi um Formeln für G.

(3) Beweisen Sie, daß der Graph G aus (2) nicht schematreu ist. Definieren Sie dazu einen geeigneten Vervollständigungsoperator C, der die negativen Formeln aufnimmt, die für diesen Beweis notwendig sind.

Aufgabe 4.5 Rebalancierung

Erweitern Sie die Spezifikation für gefädelte Binärbäume um eine Attributdefinition, Produktionen und eine Transaktion zur Rebalancierung von TBTree's. Die Transaktion soll die Produktionen zur Rebalancierung so lange anwenden, bis im ganzen Baum eine Balance kleiner |1| erreicht ist.

Aufgabe 4.6 Kalkül

Gegeben sei folgendes Graphschema in PROGRESS-Notation und durch die Formelmenge Phi:

<u>node_class</u> PERSON <u>end</u>;
<u>node_class</u> MAN <u>is_a</u> PERSON <u>end</u>;
<u>node_class</u> WOMAN <u>is_a</u> PERSON <u>end</u>;
<u>node_type</u> Rogue : MAN <u>end</u>;
<u>node_type</u> Gentleman : MAN <u>end</u>;
<u>node_type</u> Lady : WOMAN <u>end</u>;
<u>edge_type</u> Husband : WOMAN -> MAN <u>end</u>;
<u>production</u> Marry =

[‘1 : WOMAN] -~~Husband~~->
[‘2 : MAN] <-~~Husband~~-

::=

[‘1 : WOMAN] -Husband-> [‘2 : MAN]

<u>end</u>;

Phi = {
$\forall$ t: type(t, MAN) $\Leftrightarrow$ (t = Rogue
$\vee$ t = Gentleman),
$\forall$ t: type(t, WOMAN) $\Leftrightarrow$ t = Lady,
(type(t, MAN) $\vee$ type(t, WOMAN))
$\Rightarrow$ type(t, PERSON),
$\forall$ v, v' : rel(v, Husband, v') $\Rightarrow$
$\exists$ t, t' : type(v, t) $\wedge$
type(v', t') $\wedge$
type(t, WOMAN) $\wedge$
type(t', MAN)
}

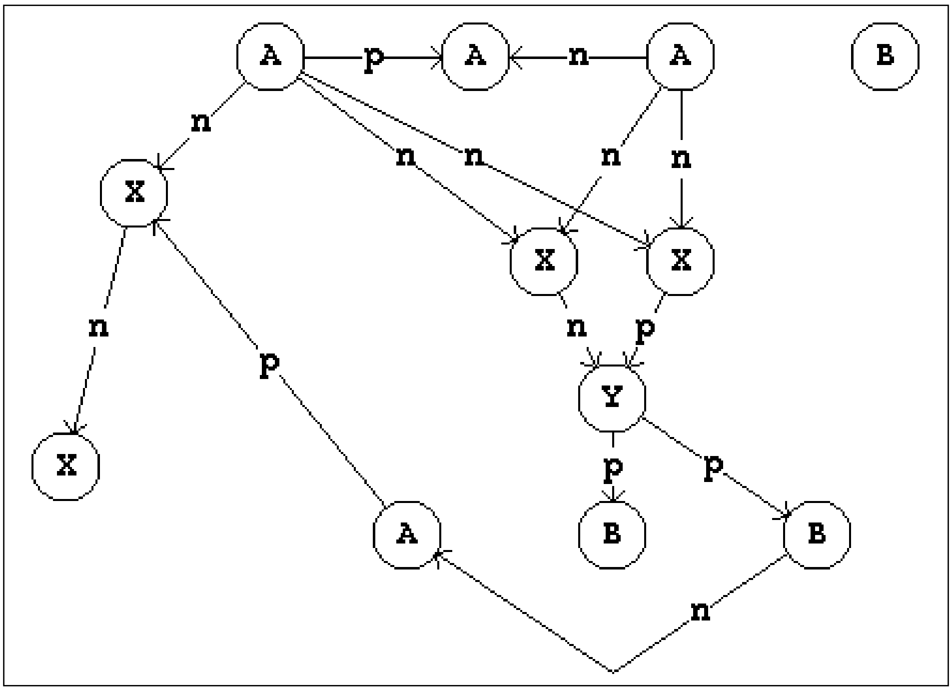
Geben Sie die Formelmengen AL, L, R und AR für die Produktion Marry an.

Anhang A. Musterlösungen

Lösungen zu Kapitel 2.

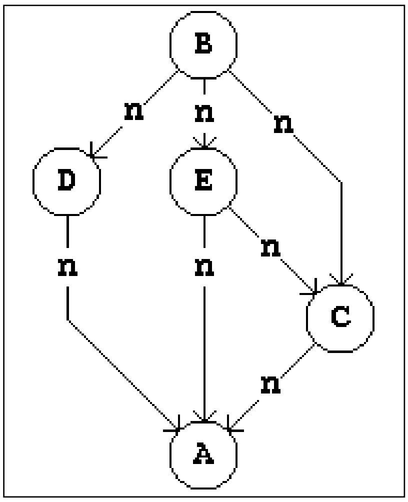
Aufgabe 2.1: Anwendung einer Graphersetzungsregel

Je nach Layout entsteht in etwa folgender Graph:



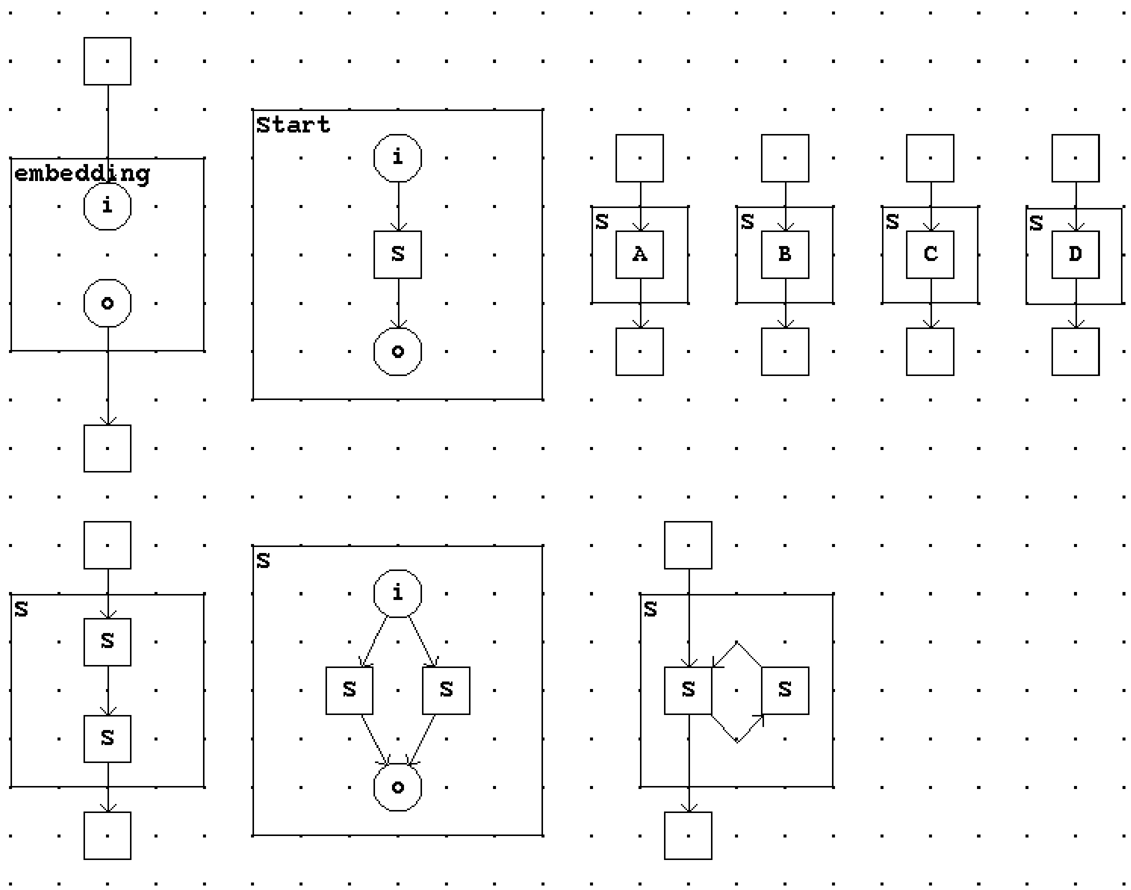
Aufgabe 2.2 Textuelle Graphbeschreibung

Je nach Layout entsteht in etwa folgender Graph



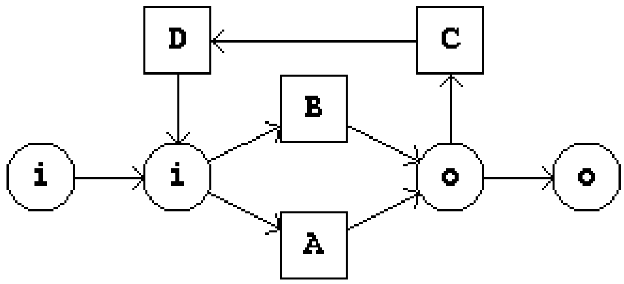
Aufgabe 2.3 Graphgrammatik für Syntaxdiagramme

(1) Folgende Graphgrammatik-Produktionen führen zu wohlgeformten Syntaxdiagrammen:

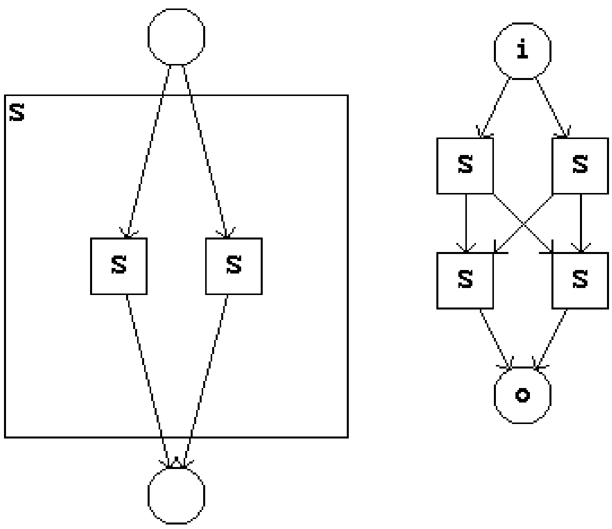


Das Rechteck mit dem Label embedding stellt zusätzliche globale Embedding-Vorschriften dar. In unserer Graphgrammatik sollen grundsätzlich einlaufende Kanten immer (wenn möglich) auf einen Knoten mit dem Label i, und auslaufende Kanten auf einen Knoten mit dem Label o umgelenkt werden.

Damit kann folgender Graph abgeleitet werden: (Darstellung gedreht)



(2) Folgen zwei Branch-Subdiagramme aufeinander so sind die i (in) - und o (out) - Knoten notwendig:

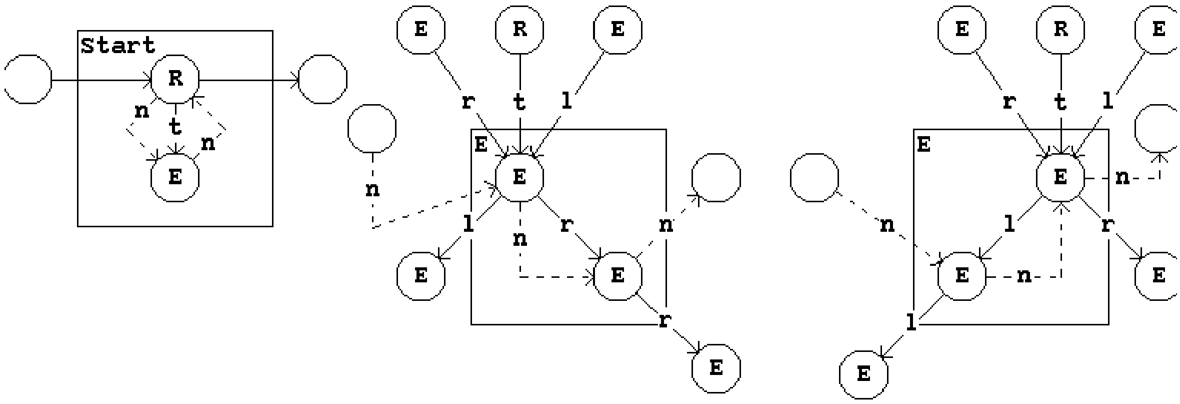


Im obigen Fall erzeugt die Produktion für Verzweigungen, wenn man sie auf zwei Knoten einer Sequenz anwendet, kein wohlgeformtes Syntaxdiagramm.

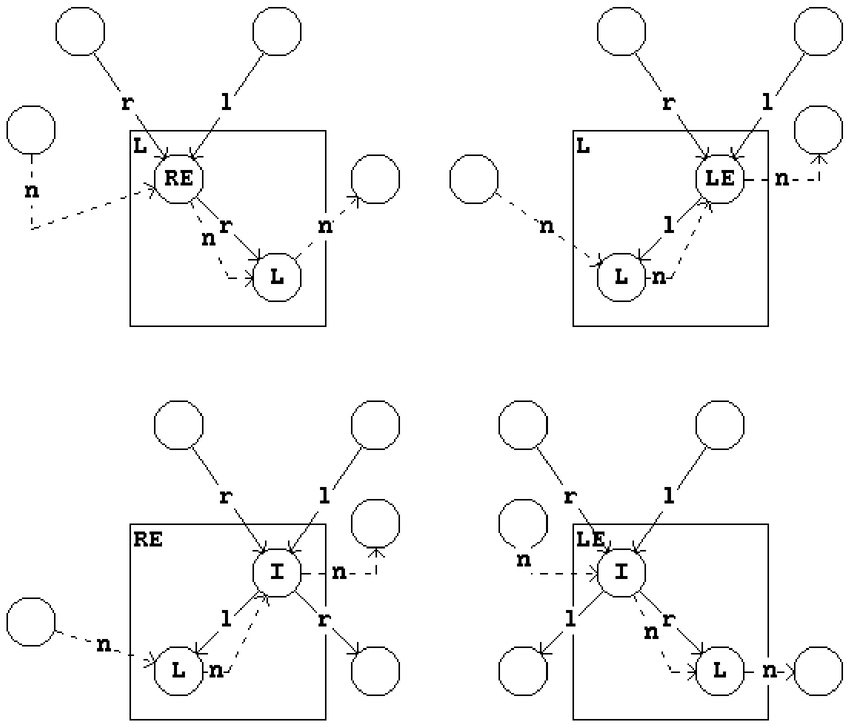
Aufgabe 2.4 Gefädelte Binärbäume

(1) Grammatik zur Erzeugung gefädelter Binärbäume

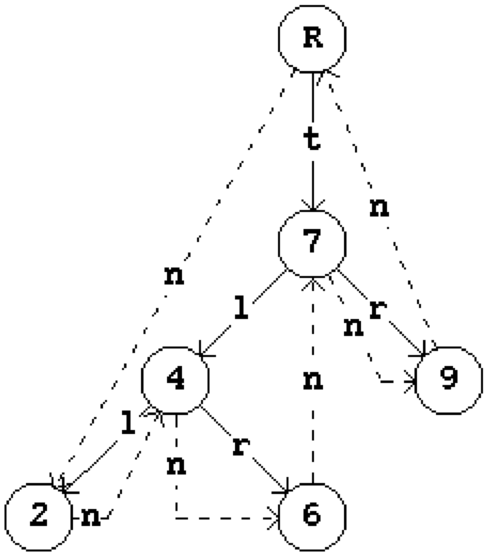
- Erste Variante: Einfügen auch mitten im Baum möglich.:



- Zweite Variante: Einfügen nur ‘unten’ möglich.



(2) Bei mir ist folgender Baum rausgekommen



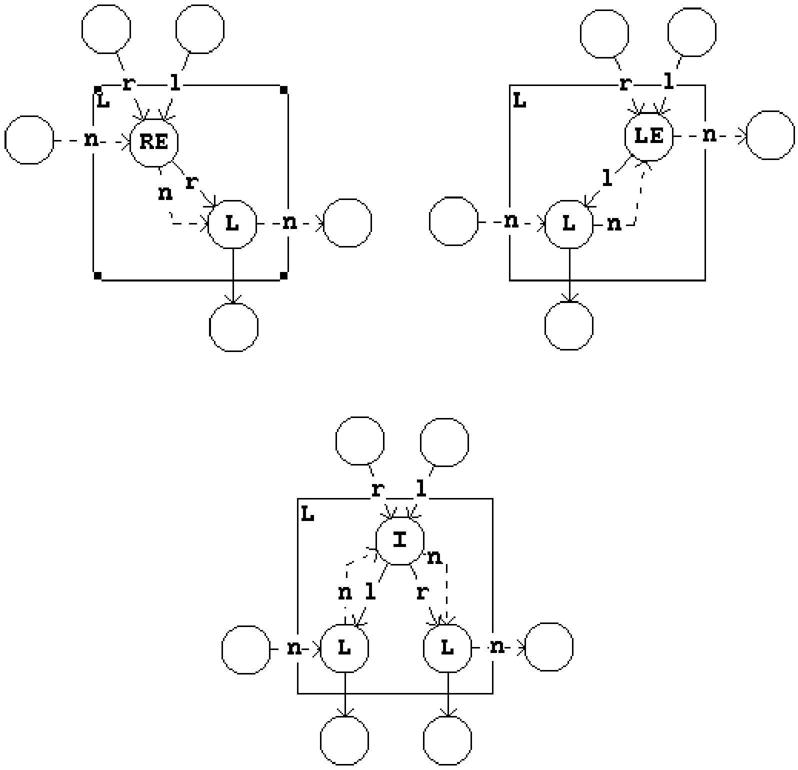
(3) Ich habe keine praktikable Lösung für das Löschen von Blättern eines gefädelten Binarbaums mit einer kontextfreien Graphgrammatik mit 1-Kontext Embedding gefunden. Das Problem besteht darin die Fädelung zu reparieren. Man müßte zwischen zwei Kontextknoten (dem Vorgänger und dem Nachfolger des zu löschenden Knotens) eine Kante ziehen können, was unser Embedding ja nicht zuläßt.

Aufgabe 2.5 Modula-2 Prozedur für TBTAddLeftLeaf

```
IMPLEMENTATION MODULE DThreadedBinTree;
   FROM SYSTEM IMPORT ADDRESS;
   TYPE
      TBTElemPtr = POINTER TO TBTElemRec;

      TBTreeRec =
         RECORD
            Root        : TBTElemPtr;
            StartThread : TBTElemPtr;
            EndThread   : TBTElemPtr;
         END;

      TBTElemRec =
         RECORD
            DataPtr  : ADDRESS;
            Left     : TBTElemPtr;
            Right    : TBTElemPtr;
            Previous : TBTElemPtr;
            Next     : TBTElemPtr;
         END;

   (* Anfuegen eines linken Sohns als Blatt an den Baum *)
   PROCEDURE TBTAddLeftLeaf( VAR Tree    : TBTreeRec;  (* Der Baum *)
                              Father  : TBTElemPtr; (* Element an das
                                                       angefuegt wird *)
                              New     : TBTElemPtr);(* Neues Element,
                                                      fertig konstruiert,
                                                       einfach reintun *)
   BEGIN
     (* -- Der neue linke Sohn hat keine Nachfolger: *)
     New^.Right := NIL;
     New^.Left  := NIL;
     (* -- Das neue Element einhaengen: *)
     Father^.Left := New;

     (* Nun die Einbettung in den Kontext. Dazu muss ein Sonderfall abgefangen
        werden: Der Wert des neuen Knotens ist der bisher kleinste.
        (Der Knoten wird also "ganz links unten" angefuegt und ist damit
       das erste Element in der doppelt verketteten Liste.)               *)
     New.Previous := Father^.Previous;
     IF (Father^.Previous = NIL) THEN
         Tree^.StartThread := New;
     ELSE
         Father^.Previous^.Next := New;
     END;

     (* -- Die beiden Knoten der "rechten Seite" verketten: *)
     New^.Next := Father;
     Father^.Previous := New;
   END TBTAddLeftLeaf;

END DThreadedBinTree.
```

Aufgabe 2.6 Präzendenzparser für gefädelte Binärbäume

(1) Um die Montonieeingenschaft für Präzedenzgraphgrammatiken zu erfüllen, mußten



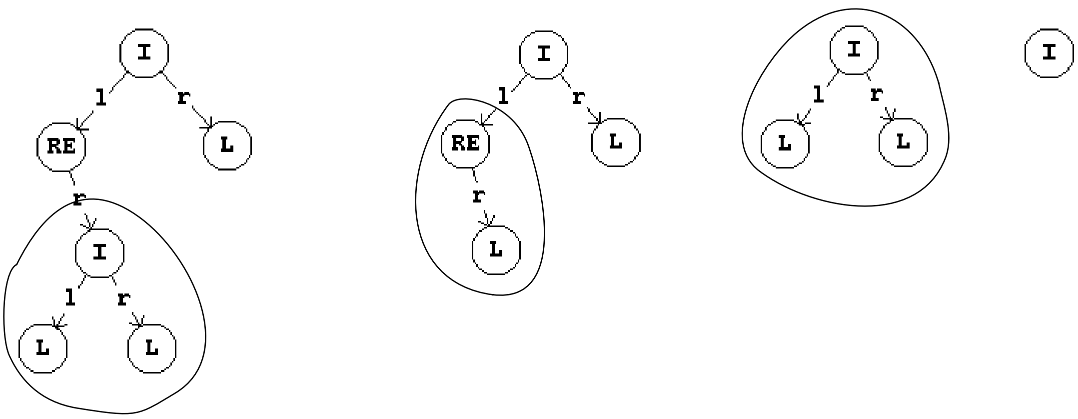
auch für auslaufende Baumkanten Embeddingvorschriften angegeben werden, obwohl durch die Konstruktion der Grammatik auch so sichergestellt ist, daß keine auslaufende Kante existiert, die im Zuge der Ersetzung verloren gehen könnten.

(2) Die Relationen sind nicht konfliktfrei. Die Konflikte sind **fett** hervorgehoben:

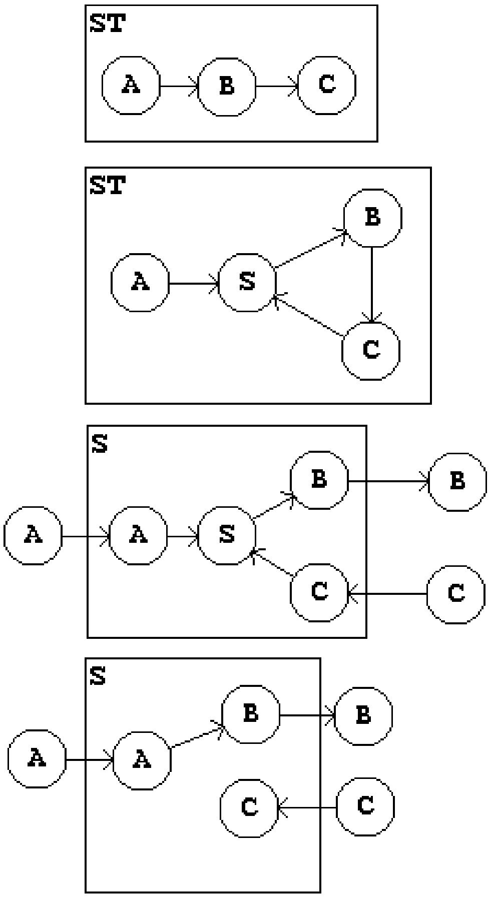
$\equiv$	IN^+	$<$	OUT^+	$>$	$<>$
RE r L	L r RE	RE r RE	L n L	**L n LE**	L n RE
LE l L	L l RE	RE r LE	L n LE	**L n I**	L n L
RE n L	L n RE	RE r I		LE n LE	LE n RE
L n LE	L r LE	RE n RE		LE n I	LE n L
I l L	L l LE	**RE n L**			
I r L	L n L	LE l RE	$(OUT^+)^{-1}$		
L n I	L r I	LE l LE			
I n L	L l I	LE l I	L n L		
		I l RE	LE n L		
		I l LE			
	$IN = IN^+$	I l I			
		I r RE			
		I r LE			
		I r I			
		I n RE			
		I n L			

(3) Ja! Die Konflikte entstehen nur durch die Fädelungskanten.

(4)



Aufgabe 2.7 $a^n b^n c^n$

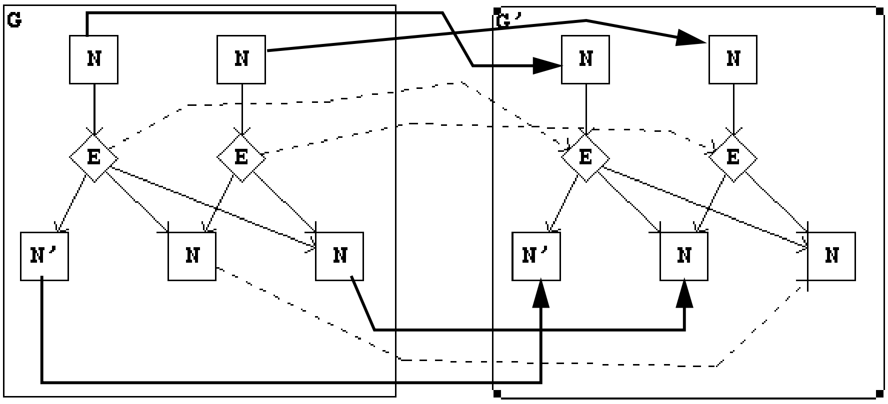


Die durch diese Produktionen erzeugten Graphen sind eine zusammenhängende Liste gleich vieler A, B und C Knoten.

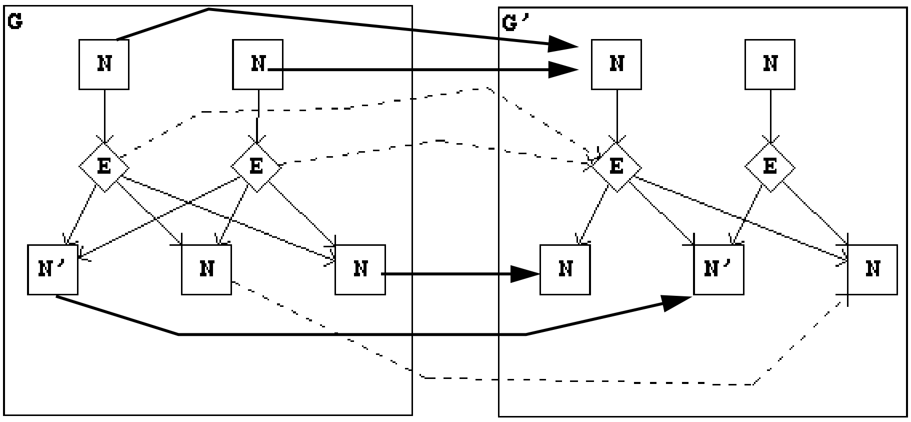
Lösungen zu Kapitel 3.

Aufgabe 3.1 Morpheus in der Graphenwelt

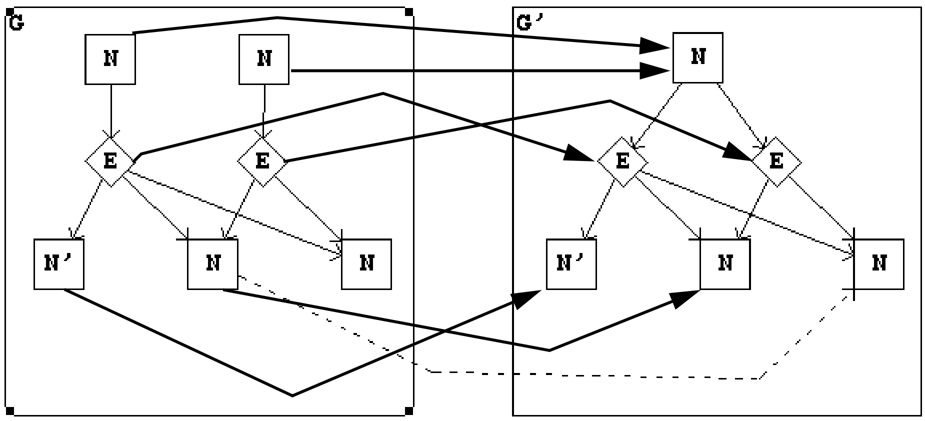
(1) Isomorphismus:



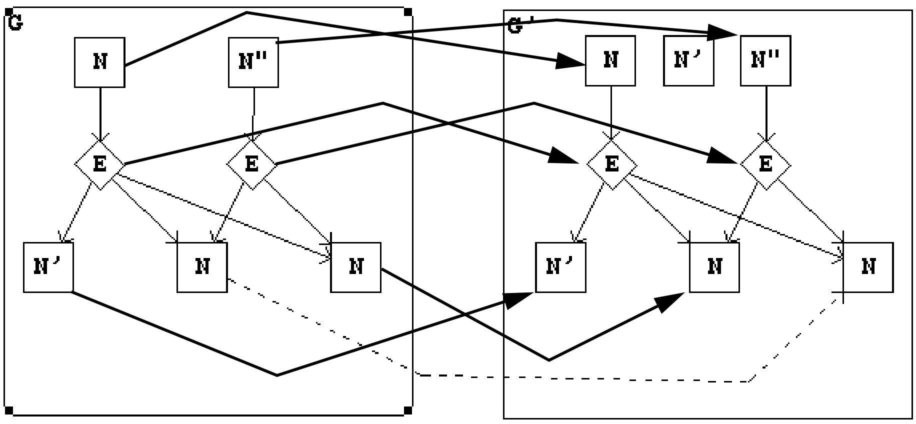
(2) Homomorphismus:



(3) Epimorphismus:

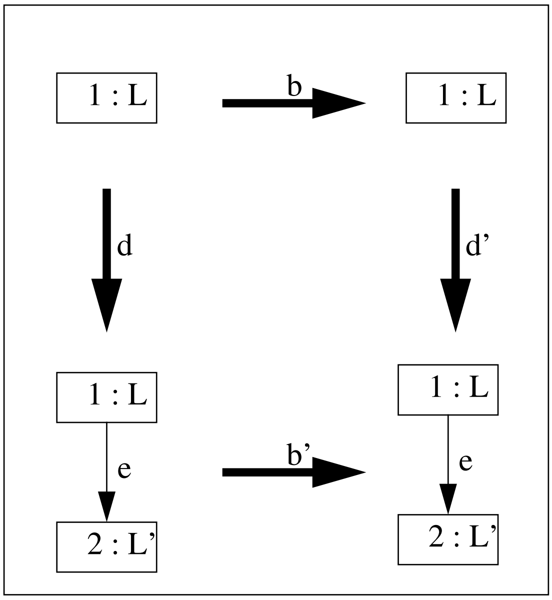


(4) Monomorphismus:

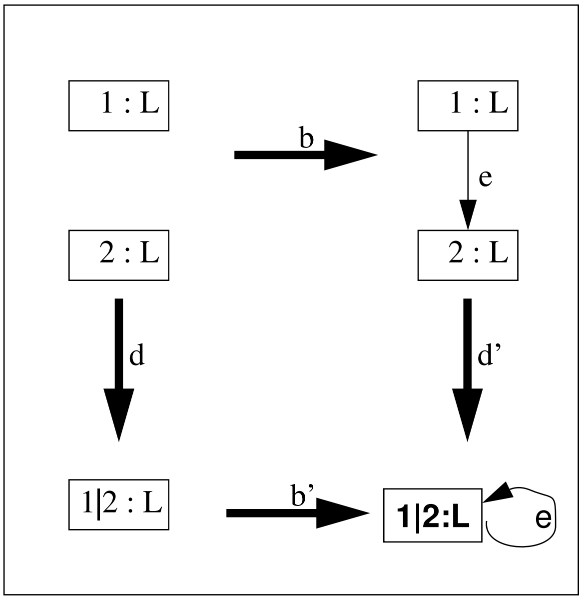


Aufgabe 3.2 Kategorientheoretische Vereinigung

(1):

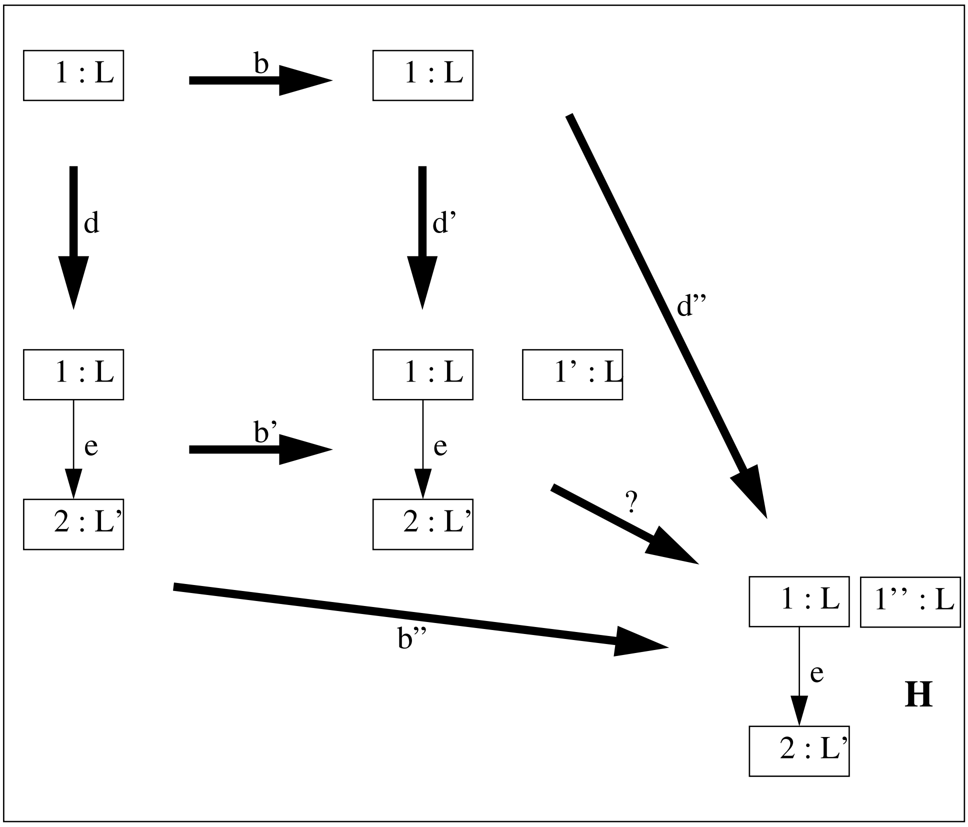


(2):



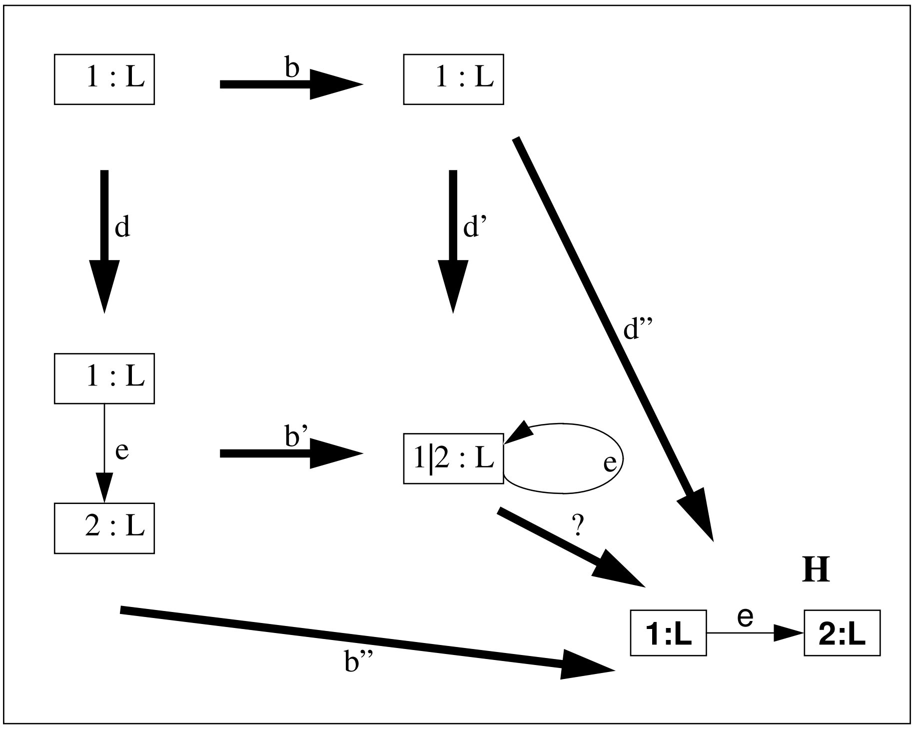
Aufgabe 3.3 Pushout-Diagramme

(1):



Der Knoten 1' kann sowohl auf den Knoten 1 als auch den Knoten 1'' in H abgebildet werden, d.h. die Eindeutigkeit ist verletzt!

(2):



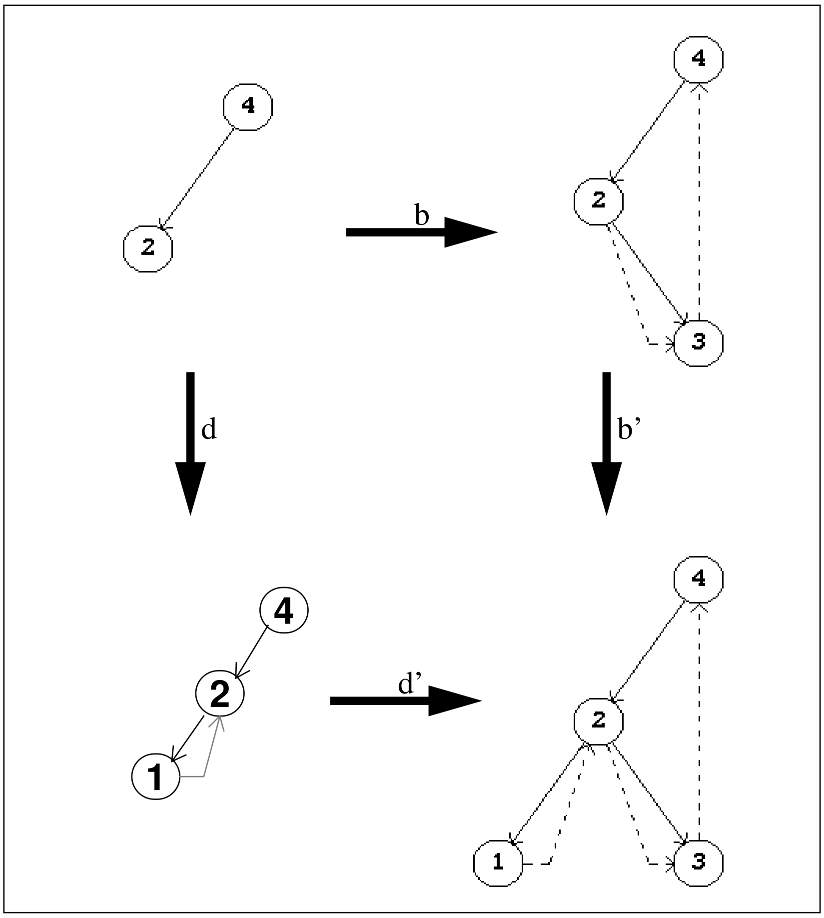
Es gibt keine Möglichkeit einen Homomorphismus h zu definieren mit

$$d'' = d' \circ h$$

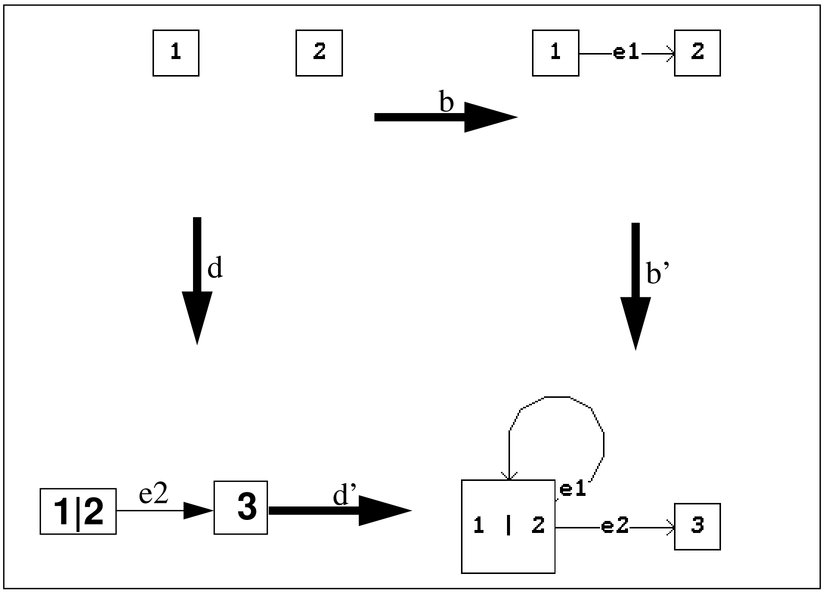
denn in H existiert kein Knoten mit einer Schlinge, auf den der Knoten 1|2 abgebildet werden könnte.

Aufgabe 3.4 Kategorientheoretische Differenz

(1):



(2):



Aufgabe 3.5 Pushout-Diagramme

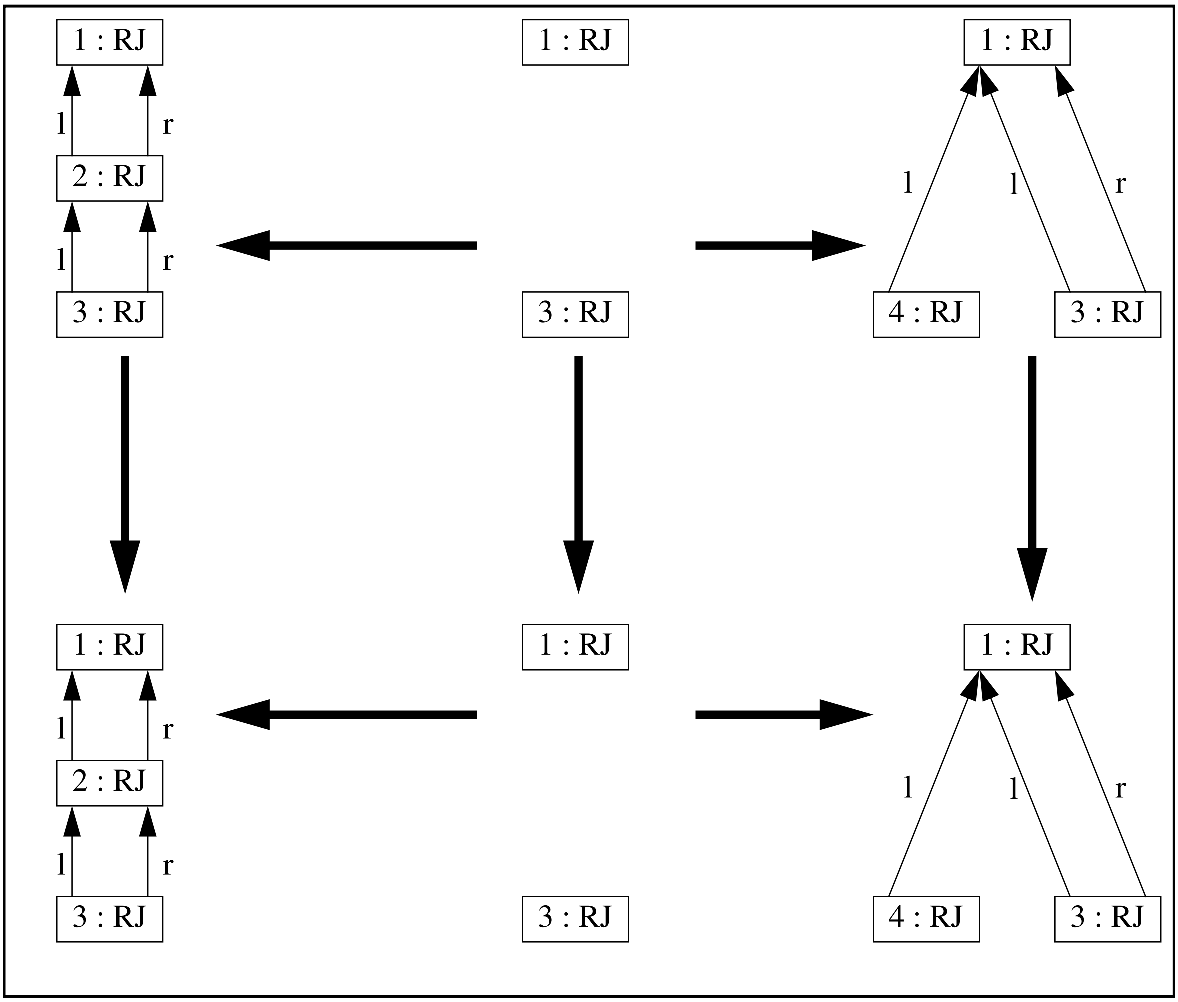
(1): Nein, die “dangling edge condition” ist verletzt.

Der Knoten 2 wird gelöscht ohne die anhängende Kante e2 mitzulöschen.

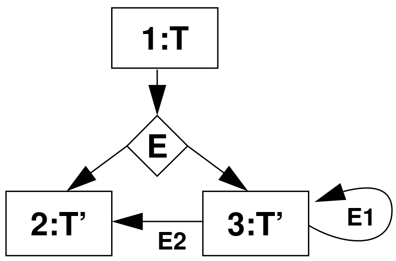
(2): Nein, die “identification condition” ist verletzt.

Es wird versucht die Knoten 1 und 2 zu löschen, die aber in G verschmolzen sind.

Aufgabe 3.6 PROGRESS <-> Kategorientheorie

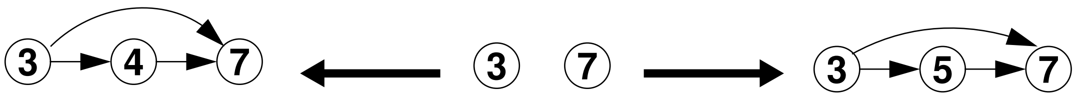


Aufgabe 3.7 Regelanwendung



Also, eigentlich war diese Aufgabe so gedacht, daß die Produktion zwei Anwendungsstellen im Graphen haben sollte. Die naheliegende, wobei die rechte Regelseite entsprechend den vorgegebenen Knotennummern auf den Graphen abgebildet wird, erzeugt den oben dargestellten Ergebnisgraphen. Zusätzlich sollte die Produktion auf den Teilgraphen, der durch die Knoten 1und 3 aufgespannt wird, anwendbar sein (deswegen die Schlinge); die Knoten 2 und 3 in L sollten also auf den Knoten 3 in G abgebildet werden. Eine solche Anwendung ist aber nicht möglich, da dabei die "Identification Condition" verletzt würde. Die beiden zu löschenden E0-Kanten in L würden auf dieselbe E0-Kante in G abgebildet werden (auch Kanten sind ja "Objekte", die der identification-Bedingung unterworfen sind).

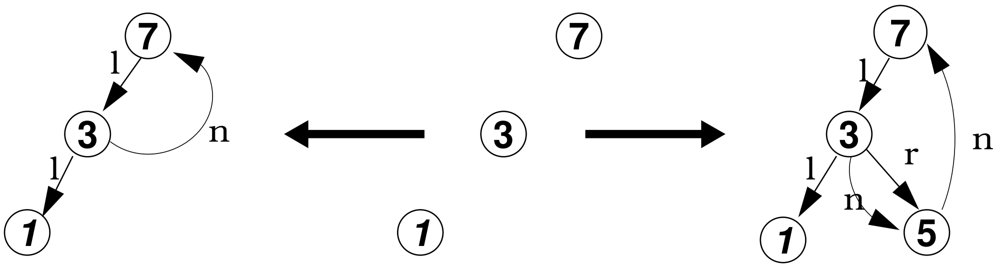
Aufgabe 3.8 Löschen + Einfügen = Ersetzen?



Es ist leider nicht die beabsichtigte Regel für das Ersetzen von Knoten in Listen entstanden, da die Kante von 3 nach 7 nicht zur üblichen Listenstruktur gehört.

Aufgabe 3.9 Liste + Baum = gefädelter Baum?

(1):



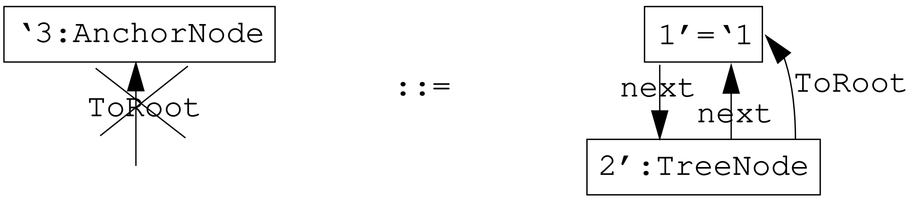
(2): Ja! Aber: Die obige Regel leistet tatsächlich das gewünschte. Die Produktion soll nur anwendbar sein, wenn der Knoten 3 kein rechtes Kind besitzt. Dies wurde in der ursprünglichen Produktion r2 durch die “Dangling Edge Condition” sichergestellt, da dort der Knoten 3 im Zuge der Ersetzung gelöscht und wieder neu erzeugt wird. Dies ist in der verschmolzenen Regel nicht mehr der Fall. Hier ist lediglich durch die Fädelungskante von 3 nach 7 sichergestellt, daß die Produktion nur in korrekten Situationen angewendet wird.

Lösungen zu Kapitel 4.

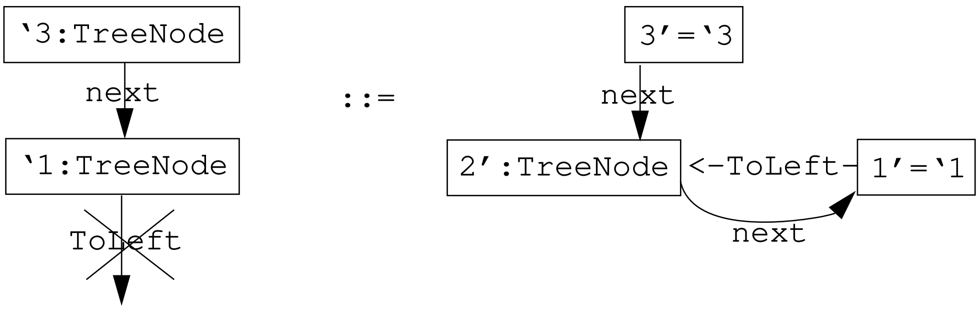
Aufgabe 4.1 Gefädelte Bäume in PROGRESS

```
spec Aufgabe4_1
from Standard import
  types BOOLEAN;
  functions
    true           : -> BOOLEAN,
    false          : -> BOOLEAN,
    'or'           : (BOOLEAN, BOOLEAN)
                     -> BOOLEAN,
    'and'          : (BOOLEAN, BOOLEAN)
                     -> BOOLEAN;
  types INTEGER;
  functions
    0               : -> INTEGER,
    '='             : (INTEGER, INTEGER)-> BOOLEAN,
    '<'             : (INTEGER, INTEGER)-> BOOLEAN,
    '<='            : (INTEGER, INTEGER)-> BOOLEAN,
    '>'             : (INTEGER, INTEGER)-> BOOLEAN,
    '=>'            : (INTEGER, INTEGER)-> BOOLEAN;

section StandardScheme

  section Nodes

    node_class Node
      intrinsic  Value : INTEGER := 0;
    end;

    node_type Anchor : Node end;

    node_type TreeNode : Node end;

  end;

  section Edges

    edge_type ToRoot : Node -> Node;

    edge_type ToLeft : Node -> Node;
```

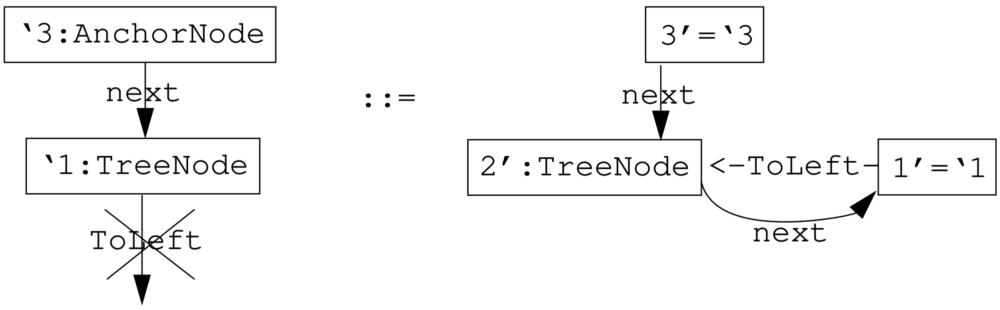
```
    edge_type ToRight : Node -> Node;

    edge_type Next : Node -> Node;

  end;

end;

section Productions

  production Create
  =
            ::=  `3:AnchorNode
  end;

  production InsertInEmptyTree (NewValue : INTEGER)
  =
```




```
    transfer Value of 2' := NewValue ;
  end;

  production InsertLeftSon (NewValue : INTEGER)
  =
```



```
  condition
     (NewValue '<=' (Value of '1)) 'and'
     ((Value of '3) '<' NewValue);
  transfer Value of 2' := NewValue ;
end;


production InsertRightSon (NewValue : INTEGER)
=
(* analog zu InsertLeftSon *)
end;


production InsertMinimum (NewValue : INTEGER)
=
```




```
  condition NewValue '<=' (Value of '1);
  transfer Value of 2' := NewValue ;
end;

production InsertMaximum (NewValue : INTEGER)
=
(* analog zu InsertMinimum *)
end;

production TestProd (NewValue : INTEGER)
=
```


```
  condition (Value of '1) '=' NewValue;
end;
```

```
  test AlreadyInTree (NewValue : INTEGER)
  =
    `1:TreeNode

    condition (Value of `1) `=' NewValue;
  end;

end;
end.
```

Aufgabe 4.2 Gefädelte Bäume in PROGRESS; Die Zweite

```
transaction Insert (NewValue : INTEGER)
=
  choose
    when not AlreadyInTree( NewValue) then
        InsertInEmptyTree( NewValue)
      or InsertMinimum( NewValue)
      or InsertMaximum( NewValue)
      or InsertLeftSon( NewValue)
      or InsertRightSon( NewValue)
  end
end;
```

Da AlreadyInTree ein Test ist, ist es möglich diesen in einem not-Statement zu negieren. Damit wird also sichergestellt, daß das neue Element noch nicht im Baum vorhanden ist. Danach werden in dem or-Statement die verschiedenen Einfügeoperationen ausprobiert. Da in jeder Situation höchstens eine der Produktionen anwendbar ist, wird automatisch die jeweils richtige vom System ausgewählt.

Aufgabe 4.3 Gefädelte Bäume in PROGRESS; Die Dritte

```
node_class TreeNode
  ...
  derived Height : INTEGER = [ max(Height of -l->, Height of -r->) '+' 1 |
                               Height of -l-> '+' 1 | Height of -r-> '+' 1 | 1 ]
end;
```

In obiger Attributberechnungsregel wird als erstes versucht, das HIGHT-Attribut als das Maximum der Höhen der beiden Kinder + 1 zu bestimmen. Scheitert dies, etwa weil eine der benötigten auslaufenden Kanten fehlt, so wird versucht, ob wenigstens ein Kind (rechts oder links) vorhanden ist. In diesen Fällen ist die Höhe die Höhe des einzelnen Kinds +1. Sonst befindet man sich auf einem Blatt und dessen Höhe wird mit 1 definiert.

Aufgabe 4.4 Logigkalkül

(1): edge_type Husband: MAN -> WOMAN;

Phi' = Phi $\cup$ { $\forall$ v_1, v_2: rel(v_1,Husband, v_2) -> $\exists$ t_1, t_2: type(v_1, t_1) $\wedge$ type(v_2, t_2)
$\wedge$ type(t_1, WOMAN)
$\wedge$ type(t_2, MAN),
$\forall$ v_1, v_2: ($\exists$ v: rel(v, Husband, v_2) $\wedge$ rel(v, Husband, v_2) -> v_1=v_2),
$\forall$ v_3, v_4: ($\exists$ v': rel(v_3, Husband, v') $\wedge$ rel(v_4, Husband, v') -> v_3=v_4)
}

(2):

Phi'' = Phi' $\cup$ { type(Bill, Rogue), type(Ben, Gentleman), rel(Bill, Husband, Ben) }

(3):

C(Phi'') = Phi'' $\cup$ { $\neg$ type(v, t) | type(v, t) $\notin$ Phi'' für v $\in$ {Bill, ...}, t $\in$ {Lady, ...}}

=> $\neg$ type(Bill, Lady) $\in$ C(Phi'')

aber: rel(Bill, Husband, Ben) $\in$ C(Phi'')

=> in C(Phi'') läßt sich folgern $\exists$ t: type(Bill, t) $\wedge$ type(t, WOMAN)

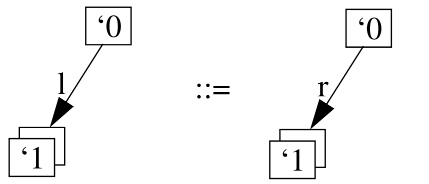
=> in C(Phi'') läßt sich folgern type(Bill, Lady) => Widerspruch!

Aufgabe 4.5 Rebalancierung

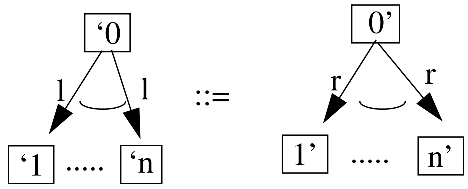
```
node_class  TreeNode
  ...
  derived  Balance : INTEGER = [Height of -l-> | 0] '-' [Height of -r-> | 0]
  derived  Height : INTEGER = [ max(Height of -l->, Height of -r->) '+' 1 |
                                 Height of -l-> '+' 1 | Height of -r-> '+' 1 | 1 ]
end;
```

Um die Anzahl der benötigten Produktionen klein zu halten, fassen wur jeweils mehrere verschiedene Ersetzungssituationen in einer Regel zusammen. Hierzu verwenden wir sogenannte Mengenknoten, die für eine (ggf. auch leere) Menge von Knoten stehen.

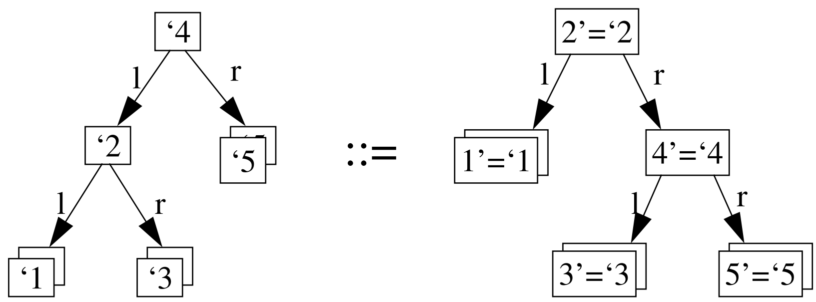
Beispiel:



ist äquivalent zu einer unbeschränkten Anzahl von Regeln der Form:



production Rebalance1



condition

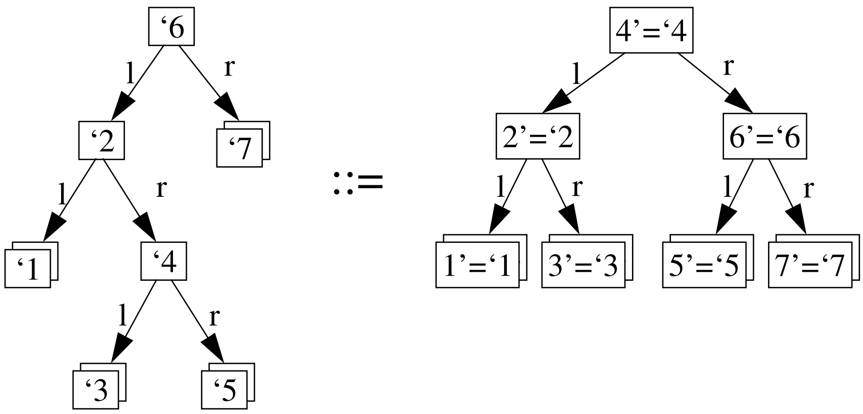
(Balance of ‘1) ‘>’ 1; (Balance of ‘2) ‘>’ 0;

embedding

redirect <-l-, <-r-, <-t- from ‘1 to 2’

end;

production Rebalance2



condition

(Balance of ‘1) ‘>’ 1; (Balance of ‘2) ‘<=’ 0;

embedding

redirect <-l-, <-r-, <-t- from ‘1 to 4’

end;

Bemerkung:

Die Produktionen für die symmetrischen Fälle sind analog aufgebaut.

Die Fädelungskanten bleiben automatisch erhalten!

Aufgabe 4.6 Kalkül

AL = { type(t1, WOMAN), type(t2, MAN),
¬ ∃ v: rel(‘1, Husband, v),
¬ ∃ v’: rel(v’, Husband, ‘2) }

L = { type(‘1, t1), type(‘2, t2) }

R = { type(‘1, t1), type(‘2, t2),
rel(‘1, Husband, ‘2) }

AR = { }

Vorwort

Das vorliegende Skript ist die Ausarbeitung einer zweistündigen Vorlesung gleichen Namens, die im WS 91/92 an der RWTH Aachen gehalten wurde. Das Ziel dieser Vorlesung bestand darin, Studenten im Hauptstudium einen Eindruck davon zu vermitteln,

- was Graphersetzungssysteme bzw. Graphgrammatiken sind,
- wofür man sie in der Praxis verwenden kann
- und welche theoretischen Konzepte ihnen zugrunde liegen.

Diesen Anspruch einzulösen, hat sich als nicht ganz einfach erwiesen. Denn das Gebiet der Graphgrammatiken ist bereits über 20 Jahre alt und hat inzwischen eine Fülle von Arbeiten hervorgebracht. Dennoch existiert bislang noch kein einheitlicher Rahmen, in den sich alle wichtigen Arbeiten einordnen lassen. So gibt es verschiedene Familien von Graphgrammatikansätzen, die auf derart unterschiedlichen Fundamenten wie der Mengentheorie oder der Kategorientheorie beruhen und sich bislang ziemlich unabhängig voneinander fortentwickelt haben. Auch findet man neben verschiedenen Tagungsbänden nur eine Monographie (aus dem Jahre 1979 von unserem gemeinsamen Doktorvater Prof. M. Nagl), die einen umfassenderen Überblick über die gesamte Disziplin gibt.

Deshalb haben wir - die Autoren dieses Skripts - erst gar nicht versucht, für alle vorgestellten Ansätze ein gemeinsames formales Rahmenwerk zu entwickeln. Vielmehr haben wir den mengentheoretisch definierten, den kategorientheoretisch fundierten und den - von uns selbst entwickelten - logikbasierten Spielarten von Graphersetzungssystemen jeweils ein eigenes Kapitel gewidmet und uns bei ihrem Vergleich auf das Kriterium der praktischen Einsetzbarkeit und Ausdruckskraft beschränkt.

Da wir bei unseren Studenten und den Lesern dieses Skripts keine theoretischen Vorkenntnisse etwa im Bereich der Kategorientheorie voraussetzen wollten, mußten wir uns oft auf grundlegende Definitionen und deren Motivation beschränken sowie auf die Vorstellung neuester Forschungsergebnisse, weiterführender Theoreme und vor allem auf den Beweis bestimmter Sätze verzichten. Deshalb möchten wir uns an dieser Stelle bei all den Mitstreitern der Graphgrammatikgemeinde entschuldigen, deren Arbeiten wir überhaupt nicht oder nur stark vereinfacht dargestellt haben.

Aufgrund der lange Zeit stark theoretischen Ausrichtung der Graphgrammatikdisziplin waren auch wir oft in der Versuchung, den Bezug zur Praxis zu vernachlässigen. Dieser Gefahr sind wir - hoffentlich - durch die Einbeziehung vieler Beispiele und Übungsaufgaben entgangen (die Lösungen der Aufgaben befinden sich im Anhang des Skripts). Auch haben wir uns bei der Vorstellung jeder Familie von Graphersetzungssystemen auf einige beson-

ders interessante Punkte aus der Sicht des Anwenders konzentriert. Bei den mengentheoretischen Ansätzen in Kapitel 2 waren dies das Parsen von Graphstrukturen und die gemischt prozedural/regelbasierte Programmierung, bei den kategorientheoretischen Ansätzen in Kapitel 3 die Komposition und parallele Anwendung von Regeln und bei unserem eigenen logikbasierten Ansatz in Kapitel 4 die Definition und Überprüfung abgeleiteter Grapheigenschaften.

So hoffen wir, daß es uns gelungen ist, das Gebiet der Graphersetzungssysteme bzw. Graphgrammatiken für einen größeren Studenten- und Leserkreis verständlich darzustellen und damit bei Ihnen das Interesse an diesem noch lebhaft in Entwicklung befindlichen Gebiet der Informatik zu erwecken.

Abschließend bedanken wir uns ganz besonders bei A. Zündorf für die Entwicklung immer neuer Übungsaufgaben, bei Th. Liese für seinen Beitrag beim Edieren dieser Aufgaben und ihrer Musterlösungen sowie bei J. Schwartz und P. Heimann für das Korrekturlesen diverser Teile dieses Skripts.

Aachen, den 1. Mai 1992, *Andy Schürr und Bernhard Westfechtel*