

UNIVERSITÄT  
KOBLENZ · LANDAU



## Das Graphenlabor

Peter Dahm, Friedbert Widmann

11/98



Fachberichte  
INFORMATIK

---

Universität Koblenz-Landau  
Institut für Informatik, Rheinau 1, D-56075 Koblenz

E-mail: [researchreports@infko.uni-koblenz.de](mailto:researchreports@infko.uni-koblenz.de),

WWW: <http://www.uni-koblenz.de/fb4/>



# Das Graphenlabor

Version 4.2

Peter Dahm  
Friedbert Widmann

Institut für Softwaretechnik  
Universität Koblenz-Landau  
Rheinau 1  
56075 Koblenz

email: {dahm|widi|ist}@uni-koblenz.de

4. August 1998

## **Zusammenfassung**

Das Graphenlabor wurde mit dem Ziel entwickelt, für die Entwicklung von Anwendungsprogrammen einen abstrakten Datentyp Graph mit einer möglichst leistungsfähigen Schnittstelle zur Verfügung zu stellen. Die vom Graphenlabor verwalteten Graphen sind dynamische, gerichtete, angeordnete, typisierte und attributierte Graphen (TGraphen).

Die zur Attributierung verwendeten Attributierungsschemata sind vom Typ des jeweiligen Knotens oder der Kante abhängig. Durch die flexible Verwaltung der Typisierung und Attributierung lassen sich auch allgemeine Anwendungen für Graphen mit erst zur Laufzeit festgelegten Graphklassen realisieren.



# Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>9</b>
<b>1.1</b>	<b>Kleine Graphenterminologie</b>	<b>10</b>
1.1.1	Gerichtete Graphen . . . . .	10
1.1.2	Ungerichtete Graphen . . . . .	11
1.1.3	Orientierung . . . . .	11
1.1.4	Weitere Termini . . . . .	12
1.1.5	Typisierung . . . . .	13
1.1.6	Attributierung . . . . .	14
1.1.7	Undo . . . . .	15
<b>1.2</b>	<b>Modellierung von Graphklassen</b>	<b>15</b>
1.2.1	Programmierung mit Graphklassen . . . . .	16
1.2.2	Anwendungen mit beliebigen Graphen . . . . .	17
<b>1.3</b>	<b>Die Wertewelt des Graphenlabors</b>	<b>18</b>
<b>2</b>	<b>Beispielanwendungen</b>	<b>22</b>
<b>2.1</b>	<b>Ein einfacher Graph</b>	<b>22</b>
2.1.1	Quelltext . . . . .	22
2.1.2	Übersetzen und Binden . . . . .	25
2.1.3	Ausführen . . . . .	26
<b>2.2</b>	<b>Ein typisierter, attributierter Graph</b>	<b>26</b>
2.2.1	Die Graphklasse <i>CityMap</i> . . . . .	26
2.2.2	Erzeugen eines Typsystems . . . . .	27
2.2.3	Erzeugen eines Graphen . . . . .	30
2.2.4	Übersetzen und Binden . . . . .	33
<b>2.3</b>	<b>Laden und Weiterverarbeiten von Graphen</b>	<b>33</b>
<b>2.4</b>	<b>Temporäre Attribute</b>	<b>35</b>
2.4.1	Beispiel: Suche im Stadtplan . . . . .	36

<b>2.5</b>	<b>Undo</b>	<b>39</b>
2.5.1	Grundkonzept . . . . .	40
2.5.2	Markierungspunkte . . . . .	41
2.5.3	Verzögerte Aktionen . . . . .	41
2.5.4	Änderung von Werten temporärer Attribute . . . . .	42
2.5.5	Logische Destruktoren bei temporären Attributen . . . . .	42
<b>2.6</b>	<b>Traversieren und Behandlung von orientierten Kanten</b>	<b>43</b>
2.6.1	Ändern von Start- oder Zielknoten von Kanten . . . . .	44
2.6.2	Reihenfolgen der Kanten ändern . . . . .	45
2.6.3	Testlauf dieser Beispiele . . . . .	46
<b>3</b>	<b>Nutzung des Graphenlabors</b>	<b>48</b>
<b>3.1</b>	<b>Verzeichnisstruktur des Graphenlabors</b>	<b>48</b>
<b>3.2</b>	<b>Übersetzen und Binden von Anwendungssoftware</b>	<b>48</b>
3.2.1	Übersetzen . . . . .	48
3.2.2	Binden . . . . .	49
3.2.3	Hilfe durch Makefiles . . . . .	49
<b>3.3</b>	<b>Nutzung von Environment-Variablen</b>	<b>50</b>
<b>3.4</b>	<b>Meldungen</b>	<b>50</b>
<b>3.5</b>	<b>Tracing</b>	<b>52</b>
3.5.1	Wie kann Tracing eingeschaltet werden ? . . . . .	52
3.5.2	Wie können Funktionen Tracing-fähig programmiert werden? . . . . .	52
<b>3.6</b>	<b>Checking</b>	<b>53</b>
<b>4</b>	<b>Referenz-Handbuch</b>	<b>55</b>
<b>4.1</b>	<b>Graphen</b>	<b>55</b>
4.1.1	Typen und Nullwerte . . . . .	55
4.1.2	Graphen anlegen und löschen: g_grfall.c . . . . .	57
4.1.3	Graphenelemente . . . . .	58
4.1.4	Graphstruktur manipulieren: g_grfman.c . . . . .	60
4.1.5	Zugriff auf Typinformationen und Attributewerte: g_grfatr.c . . . . .	63
4.1.6	Graphen temporär attributieren: g_grftmp.c . . . . .	65
4.1.7	Graphen ausgeben: g_grfppt.c . . . . .	68
4.1.8	Graphen in Dateien ablegen und laden: g_grfsto.c . . . . .	71
4.1.9	Graphen traversieren: g_grftra.c . . . . .	73
4.1.10	Knoten- oder Kantenanordnung erfragen und manipulieren: g_grford.c . . . . .	86

4.1.11	Knoten, Kanten testen: <code>g_grftst.c</code> . . . . .	88
4.1.12	Kanteninformation erfragen: <code>g_grfaux.c</code> . . . . .	90
4.1.13	Grad von Knoten erfragen: <code>g_grfdeg.c</code> . . . . .	91
4.1.14	Globale Graphdaten erfragen: <code>g_grfmsc.c</code> . . . . .	93
4.1.15	UNDO-Schnittstelle . . . . .	95
<b>4.2</b>	<b>Typsysteme</b>	<b>96</b>
4.2.1	Verwaltung der Typsysteme: <code>g_ttypsys.c</code> . . . . .	96
4.2.2	Verwaltung der Attributmenge: <code>g_ttypsys.c</code> . . . . .	98
4.2.3	Verwaltung der Typen: <code>g_ttypsys.c</code> . . . . .	100
4.2.4	Verwaltung der Attributierungsschemata . . . . .	102
4.2.5	Verwaltung der Subtyp-Relation: <code>g_ttypsys.c</code> . . . . .	103
<b>4.3</b>	<b>Wertebereiche: <code>g_domain.c</code></b>	<b>105</b>
4.3.1	Grundsätzliche Funktionalität . . . . .	105
4.3.2	Basiswertebereiche . . . . .	106
4.3.3	Listenwertebereiche . . . . .	107
4.3.4	Wertebereichssequenzen . . . . .	107
4.3.5	Tupelwertebereiche . . . . .	108
4.3.6	Record-Wertebereiche . . . . .	109
4.3.7	Liste von Bezeichnern . . . . .	111
4.3.8	Aufzählungswertebereiche . . . . .	111
<b>4.4</b>	<b>Werte: <code>g_valref.c</code></b>	<b>113</b>
4.4.1	Nullwerte . . . . .	113
4.4.2	Funktionalität der Wertereferenzen . . . . .	113
4.4.3	Graphunabhängige Werte . . . . .	114
4.4.4	Grundsätzliche Funktionalität von Werten . . . . .	115
4.4.5	Werte der Basiswertebereiche . . . . .	116
4.4.6	Listenwerte . . . . .	116
4.4.7	Tupelwerte . . . . .	118
4.4.8	Recordwerte . . . . .	119
4.4.9	Aufzählungswerte . . . . .	119
<b>4.5</b>	<b>Undo-Puffer: <code>g_undo.c</code></b>	<b>120</b>
4.5.1	Die Klasse <code>G_undoBuffer</code> . . . . .	120
<b>4.6</b>	<b>Temporäre Attribute</b>	<b>124</b>
4.6.1	Konstruktoren, Destruktoren . . . . .	124
4.6.2	Ausgabe der Attributinhalt . . . . .	124

4.6.3	Schichten von temporären Attributen . . . . .	125
4.6.4	Anbindung an den Undo-Puffer . . . . .	125
<b>4.7</b>	<b>Bezeichner: <code>g_id.c</code></b>	<b>125</b>
<b>4.8</b>	<b>Tracing : <code>g_trace.c</code></b>	<b>127</b>
<b>4.9</b>	<b>Meldungsausgabe <code>g_msg.c</code></b>	<b>128</b>
<b>4.10</b>	<b>Environment-Variablen</b>	<b>129</b>
<b>A</b>	<b>Dateiformate</b>	<b>130</b>
<b>A.1</b>	<b>Typsystemdatei</b>	<b>131</b>
A.1.1	Wertebereichsinformationen . . . . .	131
A.1.2	Attributinformationen . . . . .	132
A.1.3	Typinformationen . . . . .	132
<b>A.2</b>	<b>Graphdatei</b>	<b>133</b>
A.2.1	Graphstruktur . . . . .	134
A.2.2	Knotenbeschreibung . . . . .	134
A.2.3	Kantenbeschreibung . . . . .	134
A.2.4	Attributwerte . . . . .	135
A.2.5	Ein komplettes Beispiel . . . . .	136
<b>A.3</b>	<b>Meldungsdateien</b>	<b>137</b>
A.3.1	Meldungsköpfe . . . . .	137
A.3.2	Meldungstexte . . . . .	138
<b>B</b>	<b>Liste der Environment-Variablen</b>	<b>139</b>
<b>C</b>	<b>Unterstützte Systemumgebungen</b>	<b>140</b>
<b>C.1</b>	<b>Hinweise für spezielle Architekturen</b>	<b>141</b>
C.1.1	OS/2 mit IBM-ICC ( <code>i386-os2/ibm-icc</code> ) . . . . .	141
C.1.2	MS-Windows mit MS-VC ( <code>i386-win/ms-vc40</code> und <code>i386-win/ms-vc50</code> ) . . . . .	141
C.1.3	Intel i86pc unter Solaris2.6 mit SUN-Compiler ( <code>i386-unknown-solaris2.6/SUNWsp</code> ) . . . . .	141
C.1.4	Intel i86pc unter Solaris2.6 mit GNU-Compiler ( <code>i386-unknown-solaris2.6/GNUgcc</code> ) . . . . .	141
<b>D</b>	<b>Übersetzen und Binden</b>	<b>142</b>
<b>D.1</b>	<b>Bibliotheksvarianten</b>	<b>142</b>

---

D.1.1	Verfügbare Varianten . . . . .	143
<b>D.2</b>	<b>Präprozessor-Makros</b>	<b>143</b>
D.2.1	Auswahl der Varianten . . . . .	143
D.2.2	Anpassung an Betriebssystem und Compiler . . . . .	143
<b>D.3</b>	<b>Make-Variablen</b>	<b>145</b>
<b>D.4</b>	<b>Übersetzen des Graphenlabors</b>	<b>146</b>
<b>E</b>	<b>Fehlermeldungen</b>	<b>147</b>
<b>F</b>	<b>Übersicht über die Verzeichnisse</b>	<b>156</b>



# Kapitel 1

## Einleitung

Das Graphenlabor (kurz auch GraLab) wurde mit dem Ziel entwickelt, für die Entwicklung von Anwendungsprogrammen einen abstrakten Datentyp *Graph* mit einer möglichst leistungsfähigen Schnittstelle zur Verfügung zu stellen. Insbesondere werden folgende Anforderungen erfüllt:

- Knoten und Kanten sind eigenständige, identifizierbare Objekte.
- Mehrfachkanten, d.h. mehrere Kanten zwischen denselben Knoten, sind möglich.
- Die Kanten sind gerichtet.
- Der einem gerichteten Graphen zugrunde liegende ungerichtete Graph steht unmittelbar zur Verfügung.
- Der Graph kann dynamisch verändert werden, d.h. während der Laufzeit der Anwendung können Knoten und Kanten eingefügt und gelöscht werden.
- Die Kanten, die mit einem Knoten in Berührung stehen, sind in einer vom Programm veränderbaren Reihenfolge angeordnet.
- Alle Knoten, alle Kanten und alle mit einem Knoten inzidenten Kanten können leicht traversiert werden.
- Zur Unterscheidung von Knoten- bzw. Kantenklassen können die Knoten bzw. Kanten typisiert werden.
- Die verwendeten Typen können in einer Subtyp-Relation stehen.
- Die Knoten und Kanten können mit Werten versehen (*attribuiert*) werden. Das dabei verwendete Attributierungsschema ist vom Typ des jeweiligen Knotens oder der Kante abhängig.
- Die Attributwerte können aus global verfügbaren Wertebereichen gebildet werden. Strukturierte Wertebereiche können mittels Konstruktoren aus Basiswertebereichen gebildet werden.
- Knoten und Kanten können für Algorithmen kurzfristig mit Kontrollinformationen markiert werden. Diese temporäre Attributierung ist unabhängig von den typabhängigen Attributierungsschemata.
- Änderungen am Graphen können in einem *Undo*-Mechanismus protokolliert und wieder rückgängig gemacht werden.

Die vom Graphenlabor verwalteten Graphen sind i.allg. dynamische, gerichtete, angeordnete, typisierte und attribuierte Graphen. Sie werden in der Literatur als *TGraphen* (EBERT and FRANZKE, 1995) bezeichnet.

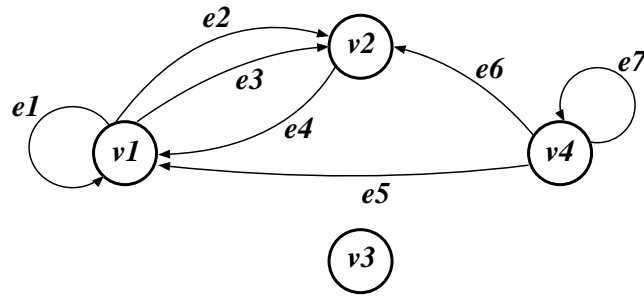


Abbildung 1.1: Ein Beispielgraph

Tabelle 1.1:  $\alpha$  und  $\omega$  des Beispielgraphs

	$e1$	$e2$	$e3$	$e4$	$e5$	$e6$	$e7$
$\alpha$	$v1$	$v1$	$v1$	$v2$	$v4$	$v4$	$v4$
$\omega$	$v1$	$v2$	$v2$	$v1$	$v1$	$v2$	$v4$

## 1.1 Kleine Graphenterminologie

Graphen sind ein leistungsfähiges Modellierungsmittel, da sie zugleich

- anschaulich sind,
- mit formalen Methoden behandelt werden können und
- effizient implementierbar sind.

Im folgenden werden grundlegende Begriffe aus der Graphentheorie erläutert. Sie werden in diesem Handbuch an vielen Stellen verwendet.

### 1.1.1 Gerichtete Graphen

Ein *gerichteter Graph* (kurz auch nur *Graph*) umfaßt eine endliche Menge von *Knoten*  $V$  und eine endliche Menge von *gerichteten Kanten* (kurz auch nur *Kanten*)  $E$ . Dabei verbindet jede *Kante*  $e \in E$  genau einen *Anfangsknoten* (oder *Startknoten*)  $v \in V$  mit genau einem *Endknoten* (oder *Zielknoten*)  $w \in V$ . Man sagt auch, Kante  $e$  führt von Knoten  $v$  nach Knoten  $w$ . Falls  $v = w$  gilt, heißt  $e$  auch *Schlinge*.

Die Beziehungen zwischen Kanten und ihren Anfangs- und Endknoten werden durch die Funktionen  $\alpha : E \rightarrow V$  und  $\omega : E \rightarrow V$  ausgedrückt.  $\alpha(e)$  ist der Anfangsknoten der Kante  $e$ ,  $\omega(e)$  der Endknoten. Zwei verschiedene Kanten dürfen durchaus denselben Anfangs- und denselben Endknoten haben und werden dann als *Mehrfachkanten* bezeichnet.

Eine Kante  $e$  heißt *inzident* zu einem Knoten  $v$ , wenn  $v$  Anfangs- oder Endknoten dieser Kante ist. Eine Kante, deren Anfangsknoten bzw. Endknoten ein Knoten  $v$  ist, heißt auch *out-Kante* bzw. *in-Kante* bezüglich Knoten  $v$ . Ein Knoten, der weder Anfangsknoten noch Endknoten irgendeiner Kante ist, heißt *isoliert*. Ein Graph kann wie in Abbildung 1.1 graphisch dargestellt werden. Die Knotenmenge  $V$  ist hier  $\{v1, v2, v3, v4\}$ , die Kantenmenge  $E$  ist  $\{e1, e2, e3, e4, e5, e6, e7\}$ . Kanten  $e2$  und  $e3$  sind Mehrfachkanten, Kanten  $e1$  und  $e7$  sind Schlingen. Knoten  $v3$  ist isoliert. Die Funktionen  $\alpha$  und  $\omega$  sind in Tabelle 1.1 dargestellt.

Das Graphenlabor repräsentiert die Menge aller Knoten und die Menge aller Kanten als (injektive) Folgen.  $Vseq$  bezeichnet die Folge aller Knoten,  $Eseq$  die Folge aller Kanten. Diese Folgen können

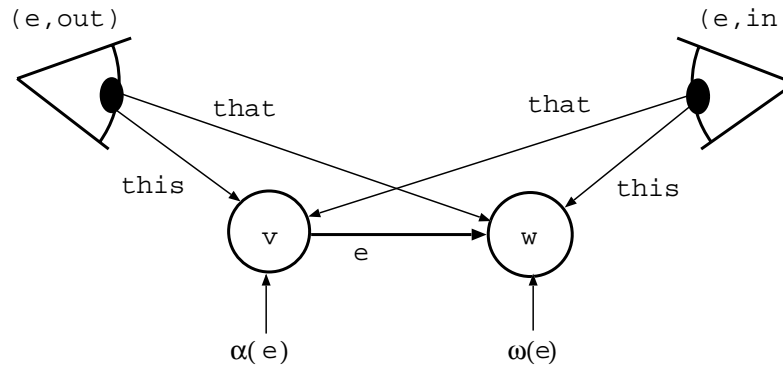


Abbildung 1.2: Orientierung von Kanten

mittels der Makros `G_forAllVertices` und `G_forAllEdges` durchlaufen werden.<sup>1</sup> Die Reihenfolge, in der das Labor die Werte zurückliefert, ist durch das Programm beeinflussbar. Weitere Details findet man bei den Beispielen in Abschnitt 2.6, S. 43, und unter der Beschreibung der Methoden in Abschnitt 4.1.9, S. 73.

Den Knoten und Kanten werden eindeutige natürliche Zahlen als Identifikationsnummern zugeordnet. Diese werden evtl. nach dem Löschen eines Knotens oder einer Kante für andere, neu erzeugte Knoten oder Kanten wiederverwendet. Zur Identifikation von Knoten und Kanten werden Variablen verwendet.

### 1.1.2 Ungerichtete Graphen

Ein *ungerichteter Graph* umfaßt eine endliche Menge von *Knoten*  $V$  und eine endliche Menge von *ungerichteten Kanten*  $E$ . Jede ungerichtete Kante  $e \in E$  verbindet entweder zwei verschiedene Knoten  $u, v \in V$  miteinander oder im Falle einer *ungerichteten Schlinge* einen Knoten  $v \in V$  mit sich selbst.

Die Beziehungen zwischen einer ungerichteten Kante und den durch sie verbundenen Knoten werden durch die Funktionen  $this, that : E \rightarrow V$  ausgedrückt.  $this(e)$  ist einer der beiden durch Kante  $e$  verbundenen Knoten,  $that(e)$  der andere Knoten.<sup>2</sup> Falls  $e$  eine Schlinge ist, gilt  $this(e) = that(e)$ . Auch in ungerichteten Graphen sind *Mehrfachkanten* erlaubt.

### 1.1.3 Orientierung

Um gerichtete und ungerichtete Graphen gleich behandeln zu können, wird im Graphenlabor das Konzept der *Orientierung* eingeführt. Eine *orientierte Kante*  $\vec{e}$  ist ein Paar  $(e, d) \in E \times Dir$  mit  $Dir = \{in, out\}$ . Dies entspricht der Tatsache, daß man eine Kante aus der Perspektive des End- oder Anfangsknotens betrachten kann. Für jede gerichtete oder ungerichtete Kante  $e$  ist  $\vec{e} = (e, out)$  die *normal* orientierte oder auch *normalisierte* Kante. Die *normale Orientierung* einer gerichteten Kante ist diejenige Orientierung, bei der man die Kante von ihrem Startknoten aus sieht, bei einer ungerichteten Kante  $e$  diejenige Orientierung, bei der man die Kante vom Knoten  $this(e)$  aus betrachtet.

Im Graphenlabor werden Kanten mit ihren normal orientierten Kanten identifiziert. Die Funktionen

<sup>1</sup> Realisiert werden diese Makros durch `first-`, `next-` Funktionspaare, die auch einzeln zur Verfügung stehen.

<sup>2</sup> Die Zuordnung ist hier willkürlich. Werden  $this(e)$  und  $that(e)$  vertauscht, ändert sich der Graph nicht. Dennoch ist die Zuordnung im weiteren als fest anzusehen.

$\alpha$ ,  $\omega$ , *this* und *that* werden auf die Menge der orientierten Kanten  $E \times Dir$  wie folgt erweitert:

$$\begin{aligned}\alpha(e, out) &:= \alpha(e) \\ \alpha(e, in) &:= \alpha(e) \\ \omega(e, out) &:= \omega(e) \\ \omega(e, in) &:= \omega(e) \\ this(e, out) &:= \alpha(e) \\ this(e, in) &:= \omega(e) \\ that(e, out) &:= \omega(e) \\ that(e, in) &:= \alpha(e)\end{aligned}$$

Man beachte, daß die Werte der Funktionen  $\alpha$  und  $\omega$  von der Orientierung der betrachteten Kante unabhängig sind und nur für gerichtete Graphen sinnvoll sind, und daß die Werte der Funktionen *this* und *that* von der Orientierung abhängen. So ist *this*( $\vec{e}$ ) immer derjenige Knoten, von dem aus man die Kante  $e$  gerade betrachtet, und *that*( $\vec{e}$ ) der Knoten am anderen Ende der Kante.

Insgesamt wird dadurch erreicht, daß zu jedem gerichteten Graph der zugrundeliegende ungerichtete Graph unmittelbar zur Verfügung steht, indem man mit orientierten Kanten arbeitet und auf den gerichteten Graph ausschließlich mit den Funktionen *this*, *that* zugreift. Soll mit dem Graphenlabor ein ungerichteter Graph verwaltet werden, kann man einen gerichteten Graphen anlegen, dessen zugrundeliegender ungerichteter Graph der gewünschte Graph ist.

Bei der Ausgabe von Graphen wird die Orientierung von Kanten durch das Vorzeichen der Identifikationsnummer ausgedrückt. Ein positives Vorzeichen bedeutet die Orientierung *out*, ein negatives Vorzeichen die Orientierung *in*.

In Abschnitt 2.6, S. 43, befinden sich Beispielprogramme, die darstellen, wie man orientierte Kanten anwenden kann. Die Methoden zur Verwaltung von orientierten Kanten werden in Abschnitt 4.1.12, S. 90, beschrieben.

### 1.1.4 Weitere Termini

Zu jedem Knoten  $v$  sind alle in- und out-Kanten als orientierte Kanten in einer Sequenz  $\Lambda(v)$  angeordnet, die im folgenden als *Folge der zum Knoten  $v$  inzidenten Kanten* bezeichnet wird. In dieser Folge erscheinen in-Kanten mit der Orientierung *in*, out-Kanten mit der Orientierung *out*, also normalisiert. Schlingen treten doppelt, nämlich in beiden Orientierungen auf. Für jede *orientierte Kante*  $\vec{e}$  aus  $\Lambda(v)$  gilt *this*( $\vec{e}$ ) =  $v$ .

Formal ergibt sich für die Funktion  $\Lambda$  die Signatur  $\Lambda : V \rightarrow \text{seq}(E \times Dir)$ . Bei gerichteten Graphen enthält  $\Lambda(v)$  die Teilfolge aller in-Kanten  $\Lambda^-(v)$  und die Teilfolge aller out-Kanten  $\Lambda^+(v)$ .<sup>3</sup> Auch diese Folgen können mit den vom Labor definierten Makros `G_forAllIncidentEdges`, `G_forAllInEdges` und `G_forAllOutEdges` durchlaufen werden. Ferner ist die Reihenfolge der Elemente der Sequenz  $\Lambda(v)$  durch Laborfunktionen beeinflussbar.

Unter dem *Grad*  $\delta(v)$  eines Knotens  $v$  versteht man die Anzahl der zum Knoten  $v$  *inzidenten Kanten*:  $\delta(v) = |\Lambda(v)|$ .<sup>4</sup> Für gerichtete Graphen definiert man analog noch den *Innengrad*  $\delta^-(v)$  und *Außengrad*  $\delta^+(v)$  eines Knoten  $v$  mittels  $\delta^-(v) = |\Lambda^-(v)|$  und  $\delta^+(v) = |\Lambda^+(v)|$ .

<sup>3</sup> Da es sich um Teilfolgen handelt, sind  $\Lambda^-(v)$  und  $\Lambda^+(v)$  durch  $\Lambda(v)$  eindeutig beschrieben.

<sup>4</sup> Bei dieser Definition werden Schlingen daher doppelt gezählt.

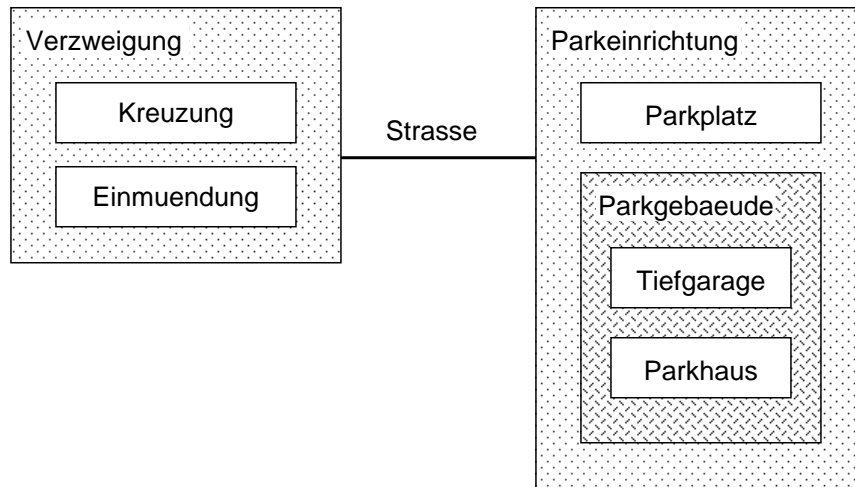


Abbildung 1.3: ER-Diagramm des Typisierungsbeispiels

### 1.1.5 Typisierung

Bei vielen Anwendungen von Graphen haben nicht alle Knoten bzw. Kanten ähnliche Bedeutungen. Vielmehr kann man Teilmengen von Knoten bzw. Kanten ähnlicher Bedeutung identifizieren.

Wenn man z.B. einen Stadtplan auf einen Graphen abbilden will, liegt es nahe, Straßenabschnitte als Kanten und Kreuzungen, Einmündungen, Parkplätze, Parkhäuser und Tiefgaragen als Knoten zu modellieren. Bei den Knoten fällt auf, daß man sie grob in zwei Mengen zerlegen kann, nämlich in die aller Verzweigungspunkte (Kreuzungen und Einmündungen) und die aller Parkeinrichtungen, die ihrerseits wieder feiner zerlegt werden können (siehe Abb. 1.3)<sup>5</sup>.

Die Einteilung von Knoten oder Kanten zu unterschiedlichen Mengen kann durch *Typisierung* realisiert werden. Die unterschiedlichen Mengen werden dabei durch *Typen* repräsentiert. Die Zugehörigkeit eines Knotens bzw. einer Kante zu einer bestimmten Menge wird dann dadurch ausgedrückt, daß dem Knoten bzw. der Kante der diese Menge repräsentierende Typ zugewiesen wird.

Falls eine Menge, die durch einen Typ A repräsentiert wird, Teilmenge der durch Typ B repräsentierten Menge ist, ist A ein *Untertyp* von B (*A is-a B*) und B ein *Obertyp* von A. So ist Parkhaus ein Untertyp von Parkgebäude. In der Abbildung wird die Untertyprelation dadurch ausgedrückt, daß der Untertyp innerhalb des Obertyps notiert ist. Die Untertyprelation ist reflexiv und transitiv, muß aber nicht notwendigerweise antisymmetrisch sein.

Weiterhin werden die Mengen der Knoten und Kanten, die einen gemeinsamen Obertyp haben, zu *Klassen* von Typen zusammengefaßt. Eine *Klasse* beinhaltet alle Kanten und Knoten des gemeinsamen Obertyps sowie aller seiner Untertypen. Dabei benutzen wir den Obertypen als Bezeichner der gesamten Klasse. Im EER-Diagramm in Abbildung 1.3 ist z.B. Parkgebäude eine *Klasse*, in der alle Knoten der *Typen* Tiefgarage, Parkhaus und Parkgebäude<sup>6</sup> enthalten sind.

<sup>5</sup> Nach diesem Modell sind keine Straßen zwischen Verzweigungen bzw. zwischen Parkeinrichtungen möglich. Deshalb ist es kaum für reale Anwendungen einsetzbar und sollte überarbeitet werden.

<sup>6</sup> Da im benutzten EER-Dialekt Parkgebäude eine totale Generalisierung darstellt, darf es keine Knoten vom Typ Parkgebäude geben.

## Realisierung

Im Graphenlabor wird Typisierung dadurch realisiert, daß Graphen ein geeignetes *Typsystem* (Instanz der Klasse `G_typeSystem`) zugeordnet wird. Dieses Typsystem beschreibt alle verwendeten Typen (`G_type`-Instanzen) und die Subtyp-Relation zwischen ihnen. Die Beschreibung eines Typs enthält eine innerhalb des Typsystems eindeutige Typbezeichnung (als String) und ein (evtl. leeres) Schema für die Attributierung der Knoten oder Kanten des Typs (siehe dazu Abschnitt 1.1.6, S. 14). Jedes Typsystem enthält einen *Nulltyp* mit der Bezeichnung "TypeNull", der automatisch Obertyp aller anderen Typen ist. Ihm ist ein leeres Attributierungsschema zugeordnet.

Jeder Knoten und jede Kante eines Graphen hat *genau einen* Typ des dem Graph zugeordneten Typsystems.<sup>7</sup> Dieser Typ ist bei der Erzeugung des Knoten mit `G_graph::createVertex()` oder der Kante mit `G_graph::createEdge()` anzugeben.

### 1.1.6 Attributierung

Für viele Anwendungen von Graphen ist es nötig, neben der reinen Graphenstruktur noch zusätzliche Daten an Knoten oder Kanten zu verwalten. Sollen Knoten oder Kanten mit Daten versehen werden, spricht man von einer *Attributierung* des Graphen. Man kann zwei Arten der Attributierung unterscheiden:

- Häufig reicht zur Abbildung eines Realitätsausschnitts auf einen Graphen die Unterscheidung von Knoten- oder Kantenarten mittels der Typisierung nicht aus. Vielmehr ist für einige (oder alle) Knoten und Kanten zusätzliche Information zu speichern. Die Art der Information hängt dabei in der Regel vom Typ des Knotens oder der Kante ab. Im Beispiel aus dem vorangehenden Abschnitt (siehe Abb. 1.3, S. 13) könnte es notwendig sein, für die Parkeinrichtungen die Namen der Einrichtung, für Parkgebäude darüberhinaus die Öffnungszeiten zu speichern.
- Für viele Algorithmen der Graphentheorie ist es notwendig, kurzfristig Steuerinformationen an Knoten oder Kanten zu schreiben, die nach Ablauf des Verfahrens wieder gelöscht werden können. Z.B. ist es für ein Suchverfahren wichtig zu wissen, welche Knoten bereits behandelt wurden und welche nicht. Attributierung dieser Art wird im folgenden als *temporäre Attributierung* bezeichnet.

## Realisierung

Bei der Definition eines Knoten- bzw. Kantentyps im Typsystem wird ein *Attributierungsschema* definiert, indem ihm einzelne *Attribute* zugeordnet werden. Anfangs ist das Attributierungsschema leer. Für jeden Knoten und jede Kante dieses Typs werden dann *Werte* erzeugt, die zu dessen Attributenschema passen. Die Werte können geändert werden und werden beim Auslagern und Einlesen eines Graphen berücksichtigt.

Zusätzlich zu den typabhängigen Attribute kann jeder Knoten und jede Kante ein *temporäres Attribut* tragen. Der Typ des Knotens bzw. der Kante spielt dabei keine Rolle.

Ein temporäres Attribut ist ein Objekt einer von `G_tempAttribute` abgeleiteten Klasse und kann somit beliebige C++-Datenstrukturen enthalten. Mit den Methoden `G_graph::setPVTemp()` und `G_graph::setPETemp()` werden die temporären Attribute an die Knoten und Kanten übergeben.

<sup>7</sup> Sollte z.B. ein Knoten eigentlich mehrere Typen haben, weil er unterschiedlichen Mengen angehört, ist es im Graphenlabor notwendig, einen gemeinsamen Untertyp zu definieren und dem Knoten diesen gemeinsamen Untertyp zuzuordnen.

Temporäre Attribute werden in *Schichten* angelegt. Dabei sind immer nur die Attribute der zuletzt erzeugten Schicht sichtbar. Nach dem Anlegen einer neuen Schicht hat in dieser Schicht kein Graphenelement ein temporäres Attribut. Erst wenn mit `setPVTemp()` oder `setPETemp()` ein temporäres Attribut eingetragen wird, ist dieses Graphenelement temporär attribuiert. Durch die Verwendung von Schichten ist es möglich, daß ein Verfahren, welches temporäre Attribute verwendet, sich eines anderen Verfahrens bedient, das ebenfalls temporäre Attribute verwendet, ohne daß sich die Attributierungen gegenseitig stören.

### 1.1.7 Undo

Bei interaktiven Programmen (z.B. bei graphischen Editoren) ist es häufig wünschenswert, Änderungen nur vorläufig zu machen und sie gegebenenfalls wieder zurücknehmen zu können. Dazu stellt das Graphenlabor einen *Undo*-Mechanismus zur Verfügung.

Man kann einen Graphen mit einem *Undo-Puffer* verbinden, in den dann bei Änderungen des Graphen alle Daten gespeichert werden, die zum Zurücksetzen der Änderungen nötig sind. Die Sicherung der Daten erfolgt automatisch durch die zur Graphenänderung verwendeten Laborfunktionen.

Sollen die Änderungen zurückgenommen werden, genügt ein `undo()`-Aufruf des verwendeten Undo-Puffers. Die gespeicherte Information wird ausgewertet und der alte Zustand des Graphen wiederhergestellt.

Durch spezielle Einträge können in diesem Puffer auch unterscheidbare Bezugspunkte gesetzt werden, zu denen man gezielt zurückgehen kann.

## 1.2 Modellierung von Graphklassen

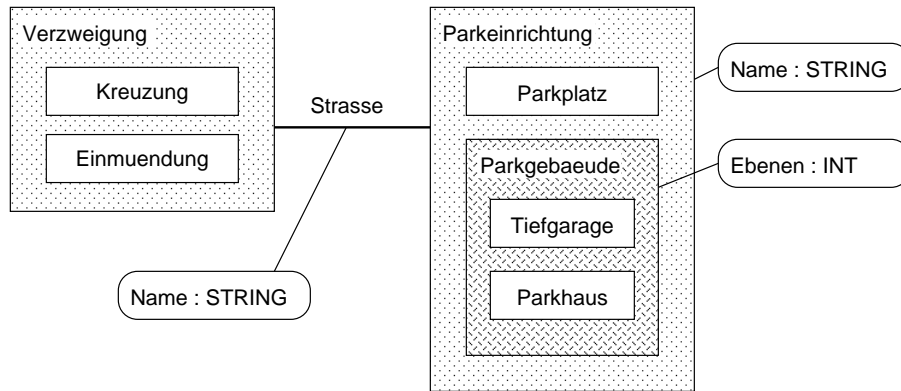
Bei der Arbeit mit Graphen wird man schnell feststellen, daß sich die Graphen in unterschiedliche Klassen einteilen lassen. Wenn man Stadtpläne in Graphen abbilden will, dann kann man z.B. Kreuzungen durch Knoten und Straßen durch Kanten modellieren. Dadurch werden sich alle Graphen dieser Stadtpläne in ihrer Struktur ähnlich sein: es wird beispielsweise Knoten vom Typ Kreuzung geben, die mit Kanten vom Typ Straße verbunden sind. Dagegen werden Graphen, die andere Sachverhalte darstellen, eine andere Struktur haben. Man kann alle *Stadtplangraphen* in eine Klasse einsortieren.

Für die Modellierung solcher Graphklassen werden folgende Modellierungsregeln empfohlen (EBERT and FRANZKE, 1995; EBERT et al., 1996):

- Jedes identifizierbare und relevante Objekt der zu modellierenden Welt wird durch einen Knoten dargestellt,
- jede Beziehung zwischen Objekten wird durch eine Kante zwischen deren Knoten dargestellt,
- ähnliche Objekte bzw. Beziehungen werden in Klassen über den Knoten bzw. Kanten zusammengefaßt,
- weitere Informationen zu den Objekten bzw. Beziehungen werden in Attributen an den Knoten bzw. Kanten abgelegt, und
- die Reihenfolge der Beziehungen wird durch die Ordnung über den Kanten realisiert.

Für die formale Spezifikation einer Graphklasse werden erweiterte Entity-Relationship-Diagramme (EER) und *GRAL*, eine *Z*-ähnliche Spezifikationssprache benutzt. EER-Diagramme eignen sich für die Beschreibung der allgemeinen Graphstruktur, da sich die Eigenschaften von TGraphen einfach abbilden lassen:

- Entity-Typen bezeichnen die Knoten-Typen,
- Relationship-Typen bezeichnen die Kanten-Typen,
- Generalisierung beschreibt die Hierarchie der Typen,

Abbildung 1.4: Graphklasse *Stadtplangraph*

- Inzidenzen zwischen Relationship-Typen und Entity-Typen bezeichnen Einschränkungen der Inzidenzstruktur der Knoten,
- Attribute an den Entity- bzw. Relationship-Typen werden für die Knoten- bzw. Kanten-Typen übernommen und

Mit der Sprache *GRAL* werden weitere Bedingungen beschrieben, die im EER-Diagramm schlecht darstellbar sind. Im Rest des Dokumentes wird nicht mehr auf *GRAL* eingegangen, da das Graphenlabor diese Bedingungen nicht überprüft.

**Beispiel:** In Abbildung 1.4 ist das EER-Diagramm für einen einfachen Stadtplan dargestellt. Für die möglichen Graphinstanzen leiten sich u.a. folgende Bedingungen ab:

1. Verzweigung, Parkeinrichtung und Parkgebaeude sind totale Generalisierungen. Diese Knotentypen stellen abstrakte Oberklassen dar und dürfen nicht instanziiert werden. Somit gibt es nur Knoten der Typen Kreuzung, Einmuendung, Parkplatz, Tiefgarage und Parkhaus.
2. Es gibt nur Kanten vom Typ Strasse.
3. Jede Straße (jede Kante) verbindet eine Verzweigung mit einer Parkeinrichtung.
4. Straßen werden durch ihren Namen näher beschrieben.
5. Parkplätze erben durch die Generalisierung das Attribut Name vom Typ Parkeinrichtung.
6. Für Tiefgaragen und Parkhäuser wird neben einem Namen die Anzahl ihrer Parkebenen gespeichert.

Dieses Modell würde z.B. verbieten, daß zwei Kreuzungen direkt mit einer Straße verbunden sind und kann so für echte Stadtpläne nur eingeschränkt benutzt werden. Dieser Mangel wird in Abschnitt 2.2, S. 26, behoben.

In den Abbildungen 1.5 und 1.6, S. 17, sind zwei mögliche Graphen dargestellt, die zur Graphklasse *Stadtplangraph* passen.

### 1.2.1 Programmierung mit Graphklassen

Der beschriebene Ansatz zur Modellierung von Graphklassen spiegelt sich auch bei der Implementierung von Applikationen mit dem Graphenlabor wider. Da mit einzelnen Programmen spezielle Probleme bearbeitet werden, werden die Graphen, auf die ein Programm zurückgreift, meistens eine ähnliche Struktur haben. Diese Programme bearbeitet Graphinstanzen einer bestimmten Graphklasse.

Wenn Graphen zwischen verschiedenen Programmen ausgetauscht werden, dann kann man die Schnittstelle der Programme durch die Graphklasse definieren. Die einzelnen Programme können unabhängig

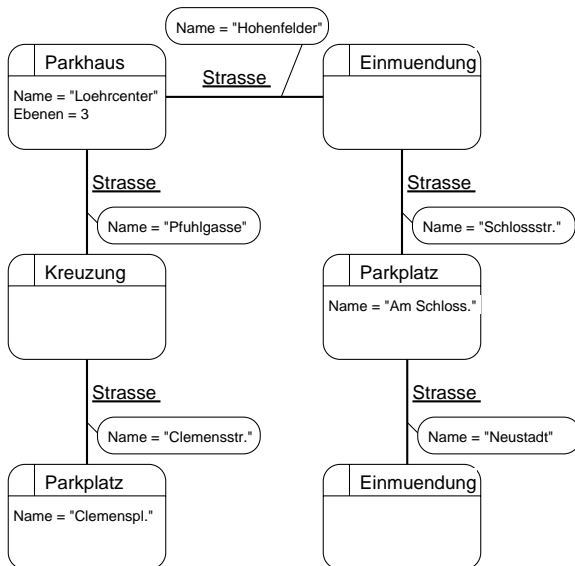


Abbildung 1.5: Koblenz Innenstadt

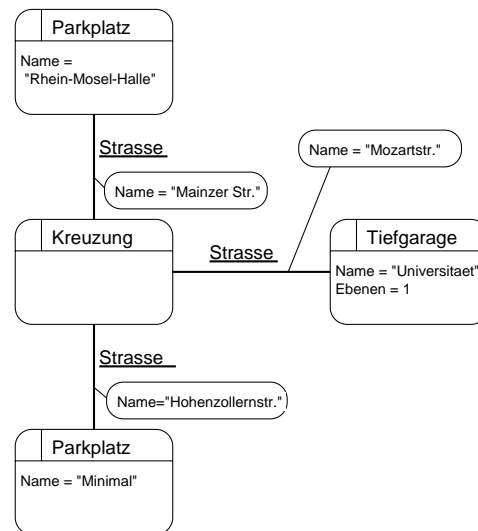


Abbildung 1.6: Koblenz Südstadt

voneinander entwickelt werden. Es muß nur die Spezifikation aus der Graphklasse eingehalten werden.

Im Graphenlabor werden die Graphklassen durch die *Typsyste* realisiert. Sie bestehen aus

- ein oder mehreren Typen, die für Knoten oder Kanten benutzt werden können,
- je einem Attributierungsschema für jeden Typ und
- einer Subtyp-Relation, in der die Typhierarchie festgelegt wird.

Damit sind viele Informationen aus den EER-Diagrammen der Graphklassen auf die Typsyste übertragbar und werden im Graphenlabor zur Konsistenzsicherung benutzt.

Für die Implementation von solchen Programmen ergibt sich folgendes Vorgehen:

1. Bei der Analyse des Problem es wird eine Graphklasse definiert, die zum Problem paßt.
2. Aus dieser Graphklasse muß ein Teilprogramm abgeleitet werden, welches das erforderliche Tysystem erzeugt.
3. Bei der Implementation der Algorithmen werden die Graphinstanzen mit dem erzeugten Tysystem verbunden. Dadurch können Knoten und Kanten mit den erforderlichen Typen angelegt werden. Dabei ist sichergestellt, daß alle Graphen der definierten Graphklasse entsprechen. Andererseits darf man bei der Manipulation von Graphen keine Graphstrukturen erzeugen, die nicht zur Graphklasse passen.

Wenn mehrere Programme dieselben Graphen benutzen, dann wird nur eine gemeinsame Graphklasse definiert. In diesem Fall kann für ein einzelnes Programm Schritt 1 und evtl. auch 2 entfallen, da sie schon für ein anderes Programm erarbeitet wurden.

## 1.2.2 Anwendungen mit beliebigen Graphen

Bei den bisher behandelten Anwendungen war zur Entwicklungszeit des Programmes die Struktur der Graphen bekannt. Das muß nicht immer so sein. Es gibt auch Anwendungen, die beliebige Graphen laden und verarbeiten. Bei diesen Anwendungen ist nur bekannt, daß die Graphen in einem bestimmten Format abgespeichert sind. Die Graphstruktur wird aus den vorhandenen Daten abgeleitet.

Das Graphenlabor kann für solche Anwendungen benutzt werden. Hierzu werden die Tysysteme in

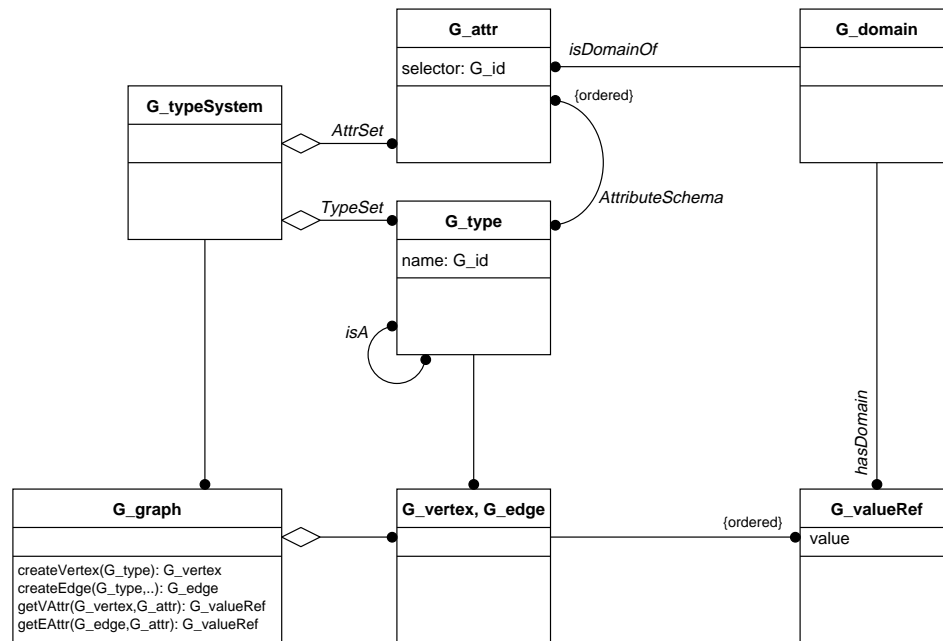


Abbildung 1.7: Zusammenhang zwischen Graphenelementen und Werten

Dateien gespeichert. Aus diesen Dateien können die Graphklassen abgeleitet und zugehörige Graphen geladen werden. Dabei werden auch die Bezeichner der Typen und Attribute rekonstruiert, sodaß man die Graphen mit ihren ursprünglichen Bedeutungen bearbeiten kann.

Man kann also mit dem Graphenlabor Programme entwickeln, die Algorithmen auf beliebigen Graphen ausführen. Die einzige Einschränkung besteht darin, daß die Graphen im Dateiformat des Graphenlabors (siehe Anhang A, S. 130) gespeichert sein müssen.

### 1.3 Die Wertewelt des Graphenlabors

Für die Attributierung von Graphen muß das Graphenlabor *Wertebereiche* und die daraus resultierenden *Werte* bereitstellen. In Abbildung 1.7 sind die Zusammenhänge zwischen den relevanten Graphenlaborklassen dargestellt.<sup>8</sup> Die *Typen* und *Attribute* werden im Typsystem (*G\_typeSystem*) definiert. Dabei muß jedem Attribut ein Wertebereich (*G\_domain*) zugeordnet werden, der den Wert dieses Attributes beschreibt. Über die Attributierungsschemata (*AttributeSchema*) werden den Typen beliebig viele Attribute zugewiesen.

Auf der anderen Seite stehen die Graphen (*G\_graph*), die bei ihrer Initialisierung mit einem Typsystem verbunden werden. Sie enthalten beliebig vielen Knoten (*G\_vertex*) und Kanten (*G\_edge*). Beim Erzeugen eines neuen Graphenelementes muß ihm ein Typ aus dem Typsystem zugewiesen werden und es werden Werte (*G\_valueRef*) angelegt, die dem Attributierungsschema des Typs entsprechen.

Die Wertebereiche (*G\_domain*) werden im *Wertebereichssystem* verwaltet, welches im Programm global verfügbar ist. Sie sind entweder *elementare Wertebereiche* (*Basiswertebereiche*), oder sie werden mit *Wertebereichskonstruktoren* erzeugt (*komplexe Wertebereiche*). Diese Zusammenhänge sind in Abbildung 1.8, S. 20, dargestellt. Dieses Diagramm stellt nicht die interne Implementation dar. Es soll nur die Zusammenhänge zwischen den Wertebereichen und ihren Operationen verdeutlichen. Die

<sup>8</sup> Die benutzte Notation ist an das Objektmodell aus RUMBAUGH et al. (1993) angelehnt.

gestrichelt gezeichneten Klassen symbolisieren Klassen, die nur intern benutzt werden. Als Schnittstelle zu Anwendungen dient nur die Klasse `G_domain`. Wenn eine Operation aufgerufen wird, die in der Unterklasse nicht definiert ist, dann erzeugt sie eine Fehlermeldung.

Als elementare Wertebereiche sind `BOOL`, `INT`, `STRING` und `DOUBLE` vordefiniert. Sie sind über Klassenvariablen der Klasse `G_domain` erreichbar. Diese Wertebereiche bilden die Basis für die konstruierten Wertebereiche. Mit der Operation `newList()` kann ein Listenwertebereich erzeugt werden, der über einem vorhandenen Wertebereich definiert ist. Die Operationen `newTuple()` und `newRecord()` erzeugen neue Tupel- bzw. Recordwertebereiche. Dabei muß eine Folge von Wertebereichen bzw. eine Zuordnung von Selektorbezeichnern zu Wertebereichen angegeben werden. Aufzählungswertebereiche werden mit `newEnum()` erzeugt und benötigen eine Liste der Aufzählungskonstanten. Diese Konstruktoroperationen sind als Klassenmethoden der Klasse `G_domain` implementiert.

Die Welt der Werte im Graphenlabor ist analog zu den Wertebereichen realisiert. In Abbildung 1.9, S. 21, sind die Zusammenhänge unter den Werten dargestellt, wobei interne Klassen wieder gestrichelt sind. Auf einen Attributwert eines Knotens oder einer Kante kann man über die Operationen `getVAttr()` und `getEAttr()` der Klasse `G_graph` (siehe Abb. 1.7, S. 18) zugreifen. Dabei erhält man eine Instanz der Klasse `G_valueRef`. Diese enthält neben einem Zeiger auf den Speicherbereich, an dem der Wert abgelegt ist, auch die Beschreibung des zugrundeliegenden Wertebereiches. Der angegebene Speicherbereich kann nur mit Kenntnis des Wertebereichs interpretiert werden.

Die Klasse `G_valueRef` ist wiederum eine abstrakte Oberklasse. Werte von elementaren Wertebereichen können mit den `updateXXX()`-Operationen gesetzt und mit den `getXXX()`-Operationen abgefragt werden. Bei konstruierten Wertebereichen werden über entsprechende Operationen einzelne Komponenten aus den komplexen Werten ausgewählt. Dabei werden wiederum Instanzen der Klasse `G_valueRef` erzeugt, die auf Komponenten in den komplexen Werten verweisen.

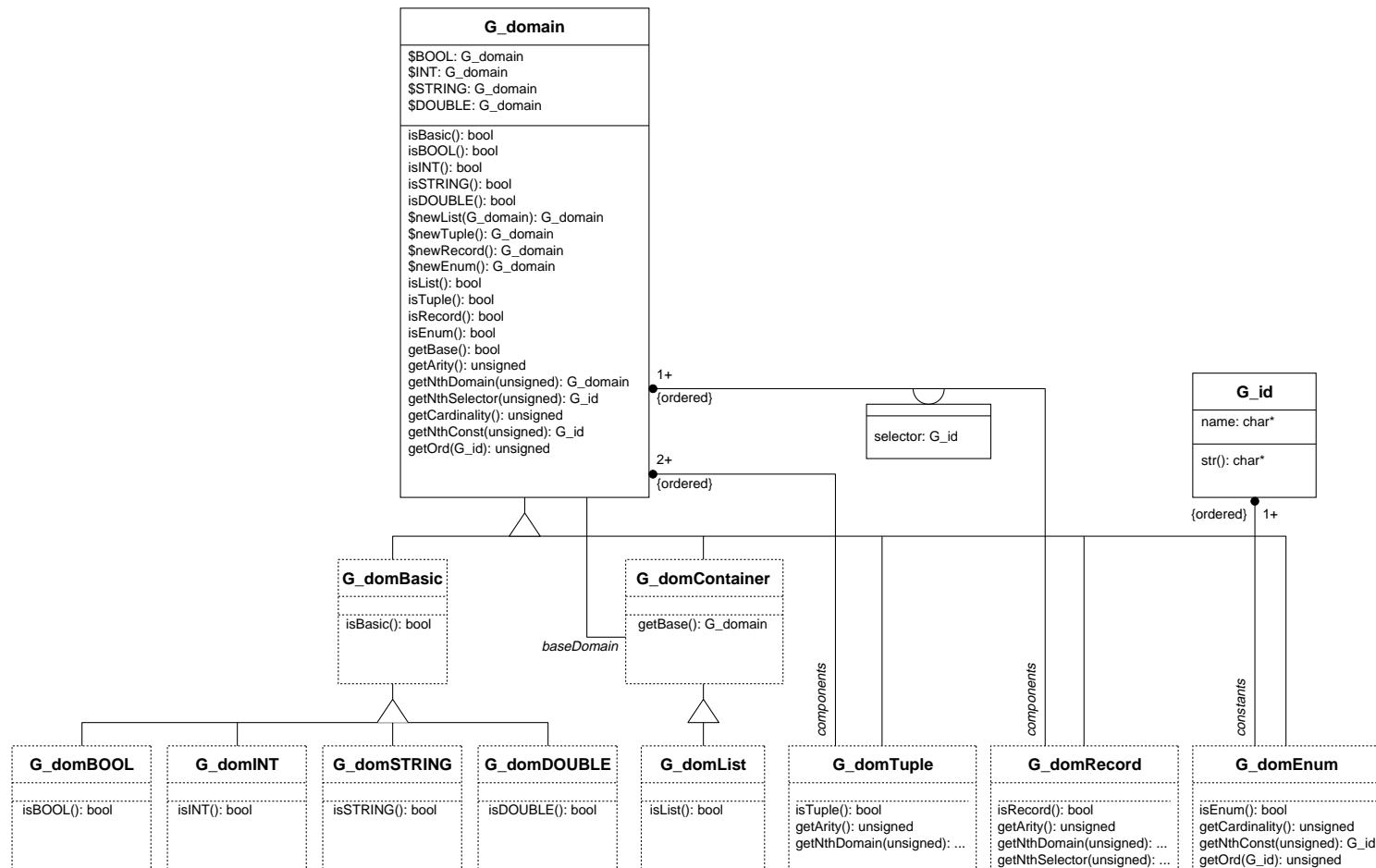


Abbildung 1.8: Objektmodell der Wertebereiche

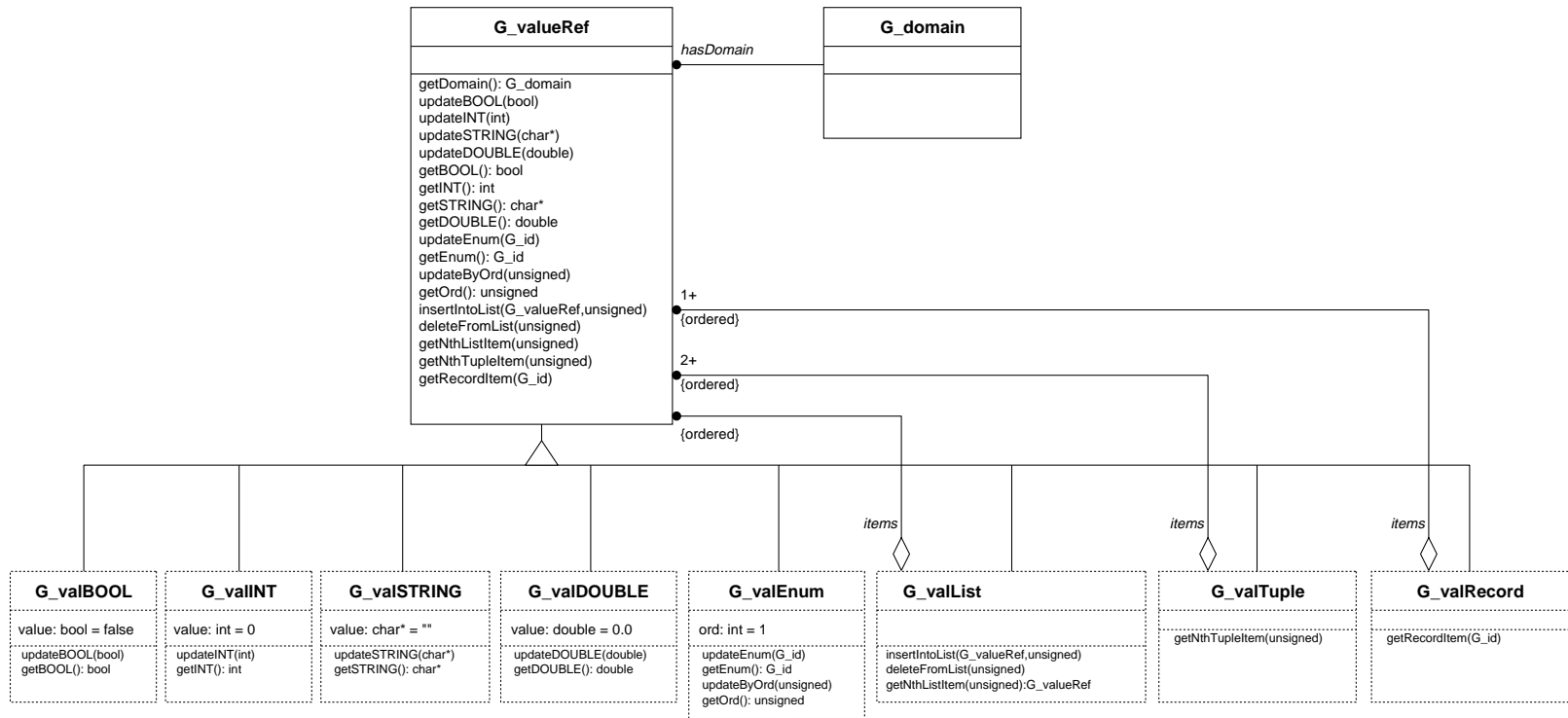


Abbildung 1.9: Objektmodell der Werte

# Kapitel 2

## Beispielanwendungen

In diesem Abschnitt soll mit verschiedenen Programmen die Anwendung des Graphenlabors dargestellt werden. Dabei wird zuerst in einem sehr einfachen Beispiel ein untypisierter Graph erzeugt und traversiert. Hierbei sind die grundlegenden Konzepte des Graphenlabors erkennbar und es werden alle Schritte erläutert, die für die Erstellung einer Applikation nötig sind.

Im zweiten Beispiel wird ein Programm nach der Vorgehensweise in Abschnitt 1.2.1, S. 16, erstellt. Dieses Programm wird in den nachfolgenden Abschnitten immer wieder aufgegriffen und erweitert, um verschiedene Konzepte des Graphenlabors vorzustellen.

Die Beispielprogramme aus diesem Kapitel befinden sich im Verzeichnis `docs/demo` unter dem Verzeichnis, in dem das Graphenlabor installiert ist (siehe Abschnitt 3.1, S. 48). Wenn man diese Programme ausprobieren oder ändern will, sollte man sie zuerst in ein eigenes Verzeichnis kopieren.

### 2.1 Ein einfacher Graph

**Problemstellung:** *Mit einem Programm soll der Baum in Abb. 2.1, S. 23, erzeugt werden. Als Ausgabe sollen alle Knoten in pre-order aufgelistet werden. Die Tiefe im Baum soll durch Punkte dargestellt werden.*

In diesem einfachen Programm sind weder die Knoten noch die Kanten typisiert, weshalb keine Graphklasse definiert wird. Dagegen muß im Graphenlabor jeder Graph mit einem Typsystem verbunden sein.

#### 2.1.1 Quelltext

Man legt mit einem beliebigen Editor eine Datei (z.B. `simple.c`) an und gibt den notwendigen Quelltext ein. Zunächst sind, neben anderen Deklarationen, die Deklarationen des Graphenlabors dem Programm mitzuteilen:

```
— demo/simple.c: —  
3 #include <iostream.h>  
4 #include <stdlib.h>  
5  
6 #include <g_graph.h>
```

Ab jetzt stehen alle Typen, globalen Variablen und Funktionsprototypen des Graphenlabors zur Verfügung. Das Programm besteht aus dem Hauptprogramm `main()`, einer Funktion `buildTree()` zum

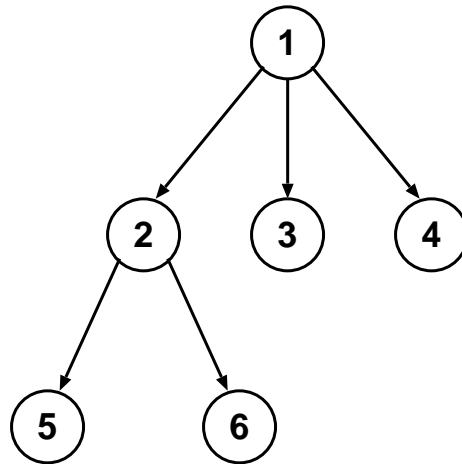


Abbildung 2.1: Einfacher Baum

Aufbau des Baumes und einer rekursiven Funktion `traverse()` zur Traversierung. Diese Funktionen werden erst weiter unten definiert. Damit sie trotzdem im Hauptprogramm benutzbar sind, muß zuerst ihre Signatur deklariert werden.

—demo/simple.c: (Forts.)—

```

7 G_vertex buildTree (G_graph &);
8 void traverse (G_graph &, G_vertex, unsigned);
9
10 int main ( int argc, char *argv[] )
11 {
12     G_typeSystem ts;
13     G_graph      g(ts);
14     G_vertex     root;
15
16     root = buildTree (g);
17     traverse (g, root, 0);
18
19     return 0;
20 }
  
```

Bei Betreten der Funktion `main` wird zunächst ein leeres Typsystem `ts` angelegt und über den Konstruktor der Klasse `G_typeSystem` initialisiert. Genauso wird ein leerer Graph `g` angelegt und mit dem Typsystem `ts` verbunden.

Alle Informationen über die Graphstruktur werden im Graph `g` abgespeichert und verwaltet. Die Knoten und Kanten *existieren* nur innerhalb des Graphen `g` und nicht als eigenständige C++-Objekte. Für den Zugriff auf einzelne Graphenelemente werden Instanzen der Klassen `G_vertex` und `G_edge` benutzt, die Referenzen auf die Graphenelemente im Graph enthalten. Will man auf Knoten oder Kanten zugreifen, dann muß man Methoden der Klasse `G_graph` aufrufen und dabei die Knoten und Kanten durch Variablen vom Typ `G_vertex` oder `G_edge` angeben. In unserem Beispiel benötigen wir eine Variable `root`, über die wir auf den Wurzelknoten (als Repräsentanten) des Baumes zugreifen können.

Nachdem alle Variablen definiert sind, wird der abgebildete Baum mit `buildTree()` aufgebaut. Diese Funktion liefert uns die Referenz des Wurzelknotens zurück. Die Funktion `traverse()` schließlich traversiert den Baum von der Wurzel `root` aus mit einer anfänglichen Rekursionstiefe von 0.

Als nächstes wird die Funktion `buildTree()` angegeben. Da jeder Knoten und jede Kante einem Typen aus dem Typsystem zugeordnet sein muß, werden sie alle dem Nulltyp zugeordnet. Hierzu wird zuerst der Nulltyp des mit dem Graph verbundenen Typsystems abgefragt und in der Variablen `tNull` gespeichert.

— demo/simple.c: (Forts.) —

```

21 G_vertex buildTree (G_graph &g)
22 {
23     G_type tNull;
24     tNull = g.getTypeSystem().G_TypeNull();
25
26     G_vertex v1, v2, v3, v4, v5, v6;
27
28     v1 = g.createVertex (tNull);
29     v2 = g.createVertex (tNull);
30     v3 = g.createVertex (tNull);
31     v4 = g.createVertex (tNull);
32     v5 = g.createVertex (tNull);
33     v6 = g.createVertex (tNull);
34
35     g.createEdge (tNull, v1, v2);
36     g.createEdge (tNull, v1, v3);
37     g.createEdge (tNull, v1, v4);
38     g.createEdge (tNull, v2, v5);
39     g.createEdge (tNull, v2, v6);
40
41     return v1;
42 }

```

Mit der Methode `G_graph::createVertex()` werden neue Knoten im Graph `g` erzeugt. Die Knotenreferenzen werden in den Variablen `v1` bis `v6` gespeichert, da sie beim Erzeugen der Kanten benötigt werden. Die Methode `G_graph::createEdge()` erzeugt neue gerichtete Kanten. Zuletzt wird die Referenz des Wurzelknotens als Rückgabewert der Funktion behandelt.

Die Funktion `traverse()` testet zuerst, ob sie sich in einem Kreis befindet.<sup>1</sup> Danach werden Punktpaare ausgegeben, die der Rekursionstiefe entsprechen.

— demo/simple.c: (Forts.) —

```

43 void traverse (G_graph &g, G_vertex v, unsigned depth)
44 {
45     if (depth >= g.vertexCount())
46     {
47         // circle detected
48         cerr << "Der Graph enthaelt einen Kreis." << endl;
49         abort ();
50     }
51     for (unsigned i = 0; i < depth; i++)
52         cout << "..";
53
54     cout << "Knoten(" << g.getVNo(v) << ")" << endl;
55
56     G_edge e;
57     G_forAllOutEdges (g, v, e)
58     {
59         traverse (g, g.omega(e), depth+1);
60     }
61 }

```

---

<sup>1</sup> Ein Kreis wird dadurch erkannt, daß die Rekursionstiefe die Knotenanzahl des Graphen überschreitet.

Im Graphenlabor wird jeder Knoten durch eine im Graph eindeutige Nummer identifiziert. Diese Nummer wird mit der Methode `G_graph::getVNo()` abgefragt und als Knoteninformation ausgedruckt. Danach werden alle Kanten in  $\Lambda^+(v)$  bearbeitet. Der Zugriff auf  $\Lambda^+(v)$  erfolgt über das Makro `G_forAllOutEdges`. Da die Endknoten  $\omega(e)$  der Kanten  $e \in \Lambda^+(v)$  die Kindknoten des gerade betrachteten Knotens  $v$  sind, wird `traverse()` mit diesen Knoten (`G_graph::omega()`) und einer höheren Rekursionstiefe aufgerufen.

### 2.1.2 Übersetzen und Binden

Das Graphenlabor gibt es in unterschiedlichen Ausprägungen, da dieselben Quelltexte

1. auf unterschiedlichen Betriebssystemen,
2. mit unterschiedlichen Compiler und
3. jeweils in verschiedenen Varianten<sup>2</sup>

benutzt werden sollen. Beim Übersetzen und Binden von Quelltexten müssen die Compiler-Parameter richtig eingestellt sein und die richtigen Bibliotheken gebunden werden. Hierfür gibt es Dateien in denen diese Parameter als voreingestellte Variablen für die Make-Programme enthalten sind. Diese werden in das eigene `makefile` geladen. Man benötigt ein kleines `makefile`, welches wiederum mit einem Texteditor erstellt wird.

```
— demo/makefile: —
3 include /home/ems/GraLab4/lib/sparc-sun-solaris2.5/GNUgcc/makecdtu.tpl
4
5 .c.o:
6     $(GRALAB_COMPILE_c) $<
7
8 simple: simple.o
9     $(GRALAB_LINK_o) -o simple simple.o -l$(GRALAB_LIB)
```

Hier wird in der ersten Zeile die oben beschriebene Datei mit den vordefinierten Variablen geladen. Der Dateiname ist aus mehreren Teilen zusammengesetzt:

1. `/home/ems/GraLab4` ist im Netzwerk der Universität in Koblenz das Verzeichnis, unter dem alle Dateien des Graphenlabors liegen,
2. `lib` ist das Verzeichnis, in dem sich die architekturabhängigen Teile des Graphenlabors befinden,
3. `sparc-sun-solaris2.5/GNUgcc` steht für das benutzte Betriebssystem und den Compiler und mit
4. `makecdtu.tpl` wird die Entwicklungsvariante angesprochen.

Eine Liste der bisher unterstützten Architekturen und Varianten findet man im Anhang C, S. 140 und D.1, S. 142.

In den Zeilen 5 und 6 wird eine Regel definiert:

Wenn eine Datei benötigt wird, die auf `.o` endet und es gibt eine neuere Datei mit demselben Basisnamen aber der Endung `.c`, dann kann die `.o`-Datei mit dem Befehl `$(GRALAB_COMPILE_c) $<` aktualisiert werden.

Dabei ist `$(GRALAB_COMPILE_c)` eine Variable, die in der Datei `makecdtu.tpl` definiert wird. Sie enthält den Aufruf des C++-Compilers mit allen nötigen Argumenten. Anstelle von `$<` wird der Name der `.c`-Datei eingesetzt.

In den Zeilen 8 und 9 wird das Programm gebunden. Dazu wird in Zeile 8 festgelegt, daß zum Erstellen der Datei `simple` (dem ausführbaren Programm) die Datei `simple.o` benötigt wird. Das Programm

<sup>2</sup> Damit kann beeinflusst werden, daß während der Entwicklung von Anwendungsprogrammen zusätzliche Plausibilitätskontrollen und Debuginformationen vorhanden sind oder das ausgelieferte Programm schnell und effizient läuft.

wird dann mit dem Befehl in Zeile 9 erzeugt. Die Variable `$(GRALAB_LINK_O)` enthält wiederum den Befehl, um eine oder mehrere Objekt-Dateien (`.o`) zu Binden. Mit `$(GRALAB_LIB)` wird die richtige Bibliotheksvariante angesprochen.

Der Quelltext des Programmes (`simple.c`) und das Makefile (`makefile`) müssen im selben Verzeichnis liegen. In diesem Verzeichnis wird das Programm `make` aufgerufen, was zu folgender Ausgabe führt. Bei einem anderen Betriebssystem oder anderem Compiler kann sie leicht abweichen.

```
docs/demo> make
g++ -Wall -fno-inline -g -I/home/ems/GraLab4/src/include \
    -I/home/ems/GraLab4/lib/sparc-sun-solaris2.5/GNUgcc \
    -DG_DEBUG -c simple.c

g++ -L/home/ems/GraLab4/lib/sparc-sun-solaris2.5/GNUgcc \
    -o simple simple.o -lgraphcdu
```

### 2.1.3 Ausführen

Sobald der Quelltext fehlerfrei übersetzt und gebunden ist, kann das Programm `simple` ausgeführt werden. Man erhält die folgende Ausgabe:

```
docs/demo> ./simple
Knoten(1)
..Knoten(2)
...Knoten(5)
...Knoten(6)
..Knoten(3)
..Knoten(4)
```

Die internen Knotennummern stimmen hierbei mit den Nummern in Abbildung 2.1, S. 23, überein. Dieses trifft nur auf dieses kleine Beispielprogramm zu. Bei großen Programmen sollte man die intern vergebenen Nummern nicht als Identifikation an der Oberfläche benutzen.

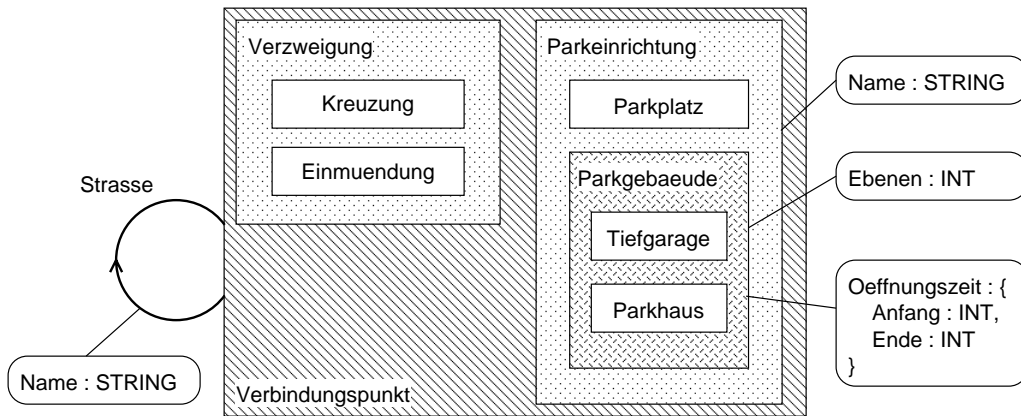
## 2.2 Ein typisierter, attributierter Graph

**Problemstellung:** *Bei dieser Anwendung soll wieder ein Stadtplan bearbeitet werden. Um die Arbeitsweise nach dem Ansatz aus Abschnitt 1.2.1, S. 16, zu verdeutlichen wird zuerst eine Graphklasse definiert.*

Im Folgenden werden alle Bezeichner in der Graphklasse und die Klartext-Bezeichnungen im Quelltext in deutsch angegeben. Für die Variablen im Programm wurden dagegen englische Namen gewählt.

### 2.2.1 Die Graphklasse *CityMap*

Da es sich wiederum um einen Stadtplan handelt, wird als Grundlage die Graphklasse aus Abbildung 1.4, S. 16, benutzt. Damit auch Straßen zwischen Verzweigungen bzw. Parkeinrichtungen möglich sind, wird die Oberklasse `Verbindungspunkt` eingeführt. Eine Kante vom Typ `Strasse` kann somit beliebig zwischen Verzweigungen und/oder Parkeinrichtungen angelegt werden. Die Klasse `Parkgebaeude` bekommt das Attribut `Oeffnungszeit`, in dem die Öffnungszeit als volle Stunden gespeichert werden. Die resultierende Graphklasse ist in Abbildung 2.2, S. 27, dargestellt.

Abbildung 2.2: Die Graphklasse *CityMap*

### 2.2.2 Erzeugen eines Typsystems

Die Quelltexte für dieses Programm werden in mehrere kleine Dateien aufgeteilt. Der Aufbau des Typsystems wird in einem separaten Programmteil erledigt. Dadurch kann dieser Teil in anderen Programmen, die auf dieselbe Graphklasse aufbauen, wiederverwendet werden. Als Schnittstelle zwischen den Programmteilen dienen das Typsystem selbst und Variablen, über die auf die Attribute und Typen zugegriffen werden kann. Solche Schnittstellen werden in C++ in Headerdateien deklariert.

#### Headerdatei `citytype.h`

In unserem Fall muß in der Headerdatei `citytype.h` deklariert werden, daß das Typsystem `cityTS` in einer C++-Datei definiert ist. Für die Record-Werte des Attributes `Oeffnungszeit` wird später ein neuer Wertebereich konstruiert. Auf diesen kann über die Variable `dTimeRange` zugegriffen werden.

```

— demo/citytype.h: —
3 extern G_typeSystem cityTS;
4 extern G_domain dTimeRange;
5
6 extern G_attr aName, aLevels, aOpeningTime;
7 extern G_type tPoint;
8 extern G_type tBranching, tCrossing, tJunction;
9 extern G_type tParking, tParkingPlace;
10 extern G_type tParkingBuilding, tParkingUnderground, tParkingHouse;
11 extern G_type tStreet;
12
13 extern void createCityTypeSystem(void);
14 extern void loadCityTypeSystem(void);

```

Für die drei Attribute, die in der Graphklasse vorkommen, werden Attributvariablen (`G_attr`) deklariert. Ebenso werden für alle Typen, die im Graph benutzt werden können, Typvariablen (`G_type`) eingeführt. Mit den letzten beiden Zeilen werden die Signaturen der Funktionen festgelegt, die von anderen Programmteilen aufgerufen werden. `createCityTypeSystem()` wird das Typsystem initialisieren und alle Typen der Graphklasse eintragen. Dieses Typsystem wird in einer Datei gespeichert, die später wieder geladen werden kann. Mit `loadCityTypeSystem()` wird ein Typsystem aus einer Datei geladen und die Variablen initialisiert.

## Erzeugen des Typsystems mit `citytype.c`

Das Typsystem wird im Programmteil `citytype.c` angelegt. Dieser Quelltext enthält alle Graphenlaboraufrufe, mit denen ein zur dargestellten Graphklasse passendes Typsystem erzeugt wird.

```
— demo/citytype.c: —
3 #include <g_graph.h>
4 #include "citytype.h"
5
```

Mit den ersten beiden Zeilen werden die Deklarationen des Graphenlabors und der Schnittstelle geladen.

```
— demo/citytype.c: (Forts.) —
6 G_typeSystem cityTS;
7 G_domain dTimeRange;
8
9 G_attr aName, aLevels, aOpeningTime;
10 G_type tPoint;
11 G_type tBranching, tCrossing, tJunction;
12 G_type tParking, tParkingPlace;
13 G_type tParkingBuilding, tParkingUnderground, tParkingHouse;
14 G_type tStreet;
```

An dieser Stelle werden die Variablen, die in der Headerdatei deklariert wurden, auch definiert. Hier wird also der Speicherplatz für diese Variablen reserviert. Dabei werden sie über die Konstruktoren der C++-Klassen initialisiert.

Oben wurde schon erwähnt, daß die Basiswertebereiche des Graphenlabors (BOOL, INT, DOUBLE, STRING) für diese Graphklasse nicht ausreichen. Es wird ein Recordwertebereich benötigt, der aus zwei Komponenten besteht. Hierfür bietet das Graphenlabor Methoden, die komplexe Wertebereiche konstruieren.

```
— demo/citytype.c: (Forts.) —
15 void ExtendDomainSystem ()
16 {
17     G_domainSequence seq;
18     seq.addDomain (G_domain::INT, "Anfang");
19     seq.addDomain (G_domain::INT, "Ende");
20     dTimeRange = G_domain::newRecord (seq);
21 }
```

Ein Record-Wertebereich besteht aus einer Menge von einzelnen Komponenten, wovon jede über einen Selektorbezeichner beschrieben wird. Diese Menge muß als Folge sequentiell aufgebaut werden und der Konstruktor-Methode `G_domain::newRecord()` übergeben werden. `newRecord()` ist eine Klassenmethode und wird deshalb mit dem Scope-Operator `::` an die Klasse `G_domain` gebunden.

Für die Attributierungsschemata der Typen müssen die Attribute zuerst im Typsystem eingerichtet werden. Diese Attribute können später den einzelnen Typen hinzugefügt werden.

```
— demo/citytype.c: (Forts.) —
22 void CreateAttributes ()
23 {
24     aName = cityTS.newAttr ("Name", G_domain::STRING);
25     aLevels = cityTS.newAttr ("Ebenen", G_domain::INT);
26     aOpeningTime = cityTS.newAttr ("Oeffnungszeit", dTimeRange);
27 }
```

Unsere Graphklasse benötigt die Attribute Name, Ebenen und Oeffnungszeit. Diese werden mit der Funktion `CreateAttributes()` erzeugt.

Nun haben wir den Rahmen so weit fertig, daß wir die einzelnen Typen generieren können. Dieses wird mit der Funktion `createCityTypeSystem()` erledigt. Dabei werden zuerst die oben beschriebenen Funktionen aufgerufen, um die benötigten Wertebereiche und Attribute anzulegen.

```
— demo/citytype.c: (Forts.) —
28 void createCityTypeSystem ()
29 {
30     ExtendDomainSystem ();
31     CreateAttributes ();
```

Für jede Klasse in der Graphklasse wird ein Typ im Typsystem angelegt. Jedem Typ wird ein eindeutiger Bezeichner zugeordnet.

```
— demo/citytype.c: (Forts.) —
32 tPoint = cityTS.newType ("Verbindungspunkt");
33 tBranching = cityTS.newType ("Verzweigung");
34 tCrossing = cityTS.newType ("Kreuzung");
35 tJunction = cityTS.newType ("Einmuendung");
36 tParking = cityTS.newType ("Parkeinrichtung");
37 tParkingBuilding = cityTS.newType ("Parkgebäude");
38 tParkingPlace = cityTS.newType ("Parkplatz");
39 tParkingUnderground = cityTS.newType ("Tiefgarage");
40 tParkingHouse = cityTS.newType ("Parkhaus");
41 tStreet = cityTS.newType ("Strasse");
```

Nun werden die Attributierungsschemata der Typen erstellt. Hierfür werden die Attribute, die schon früher erzeugt wurden, nacheinander den entsprechenden Typen hinzugefügt. Man kann hier genau nach der Graphklasse vorgehen.

```
— demo/citytype.c: (Forts.) —
42 cityTS.addAttr (tParking, aName);
43 cityTS.addAttr (tParkingBuilding, aLevels);
44 cityTS.addAttr (tParkingBuilding, aOpeningTime);
```

Die Typhierarchie wird durch eine isA-Relation im Typsystem realisiert. Diese wird durch Aufrufe der Methode `G_typeSystem::setIsA()` eingerichtet. Hierbei werden auch die Attributierungsschemata der Obertypen an die Untertypen vererbt.

```
— demo/citytype.c: (Forts.) —
45 cityTS.setIsA (tCrossing, tBranching);
46 cityTS.setIsA (tJunction, tBranching);
47 cityTS.setIsA (tBranching, tPoint);
48
49 cityTS.setIsA (tParkingUnderground, tParkingBuilding);
50 cityTS.setIsA (tParkingHouse, tParkingBuilding);
51 cityTS.setIsA (tParkingPlace, tParking);
52 cityTS.setIsA (tParkingBuilding, tParking);
53 cityTS.setIsA (tParking, tPoint);
```

Es ist weiterhin möglich die Attributierungsschemata der Typen zu ändern. Wenn bei Obertypen neue Attribute eingefügt werden, dann werden sie an alle Untertypen vererbt.

Bei Typen, die in Graphelementen benutzt werden, darf man das Attributierungsschema nicht mehr ändern. Deshalb müssen alle Typen, die in Graphen eingesetzt werden, als *benutzbar* markiert werden. Im Graphenlabor werden diese Typen *exportiert*.

```
— demo/citytype.c: (Forts.) —
54 cityTS.exportType (tCrossing);
```

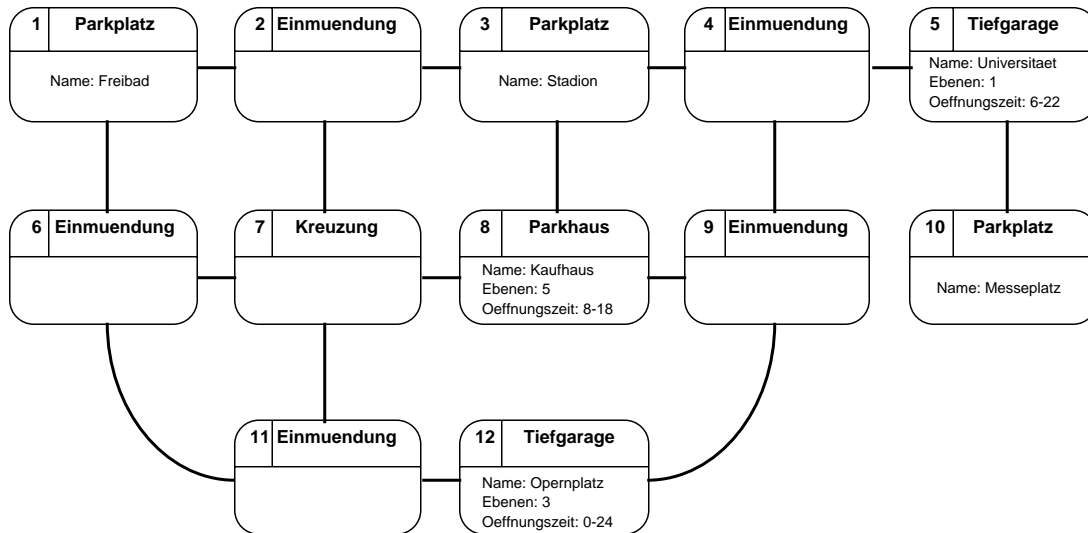


Abbildung 2.3: Stadplangraph

```

55  cityTS.exportType (tJunction);
56  cityTS.exportType (tParkingPlace);
57  cityTS.exportType (tParkingHouse);
58  cityTS.exportType (tParkingUnderground);
59  cityTS.exportType (tStreet);

```

Da eine Änderung eines Obertypen auch die Änderung der Untertypen nach sich zieht, werden bei diesen Methoden auch alle betroffenen Obertypen exportiert.

Am Ende der Funktion `createCityTypeSystem()` wird das fertige Typsystem in einer Datei gespeichert. Dieses ist nicht zwingend erforderlich. In diesem Beispielprogramm wird das Typsystem abgespeichert, damit mit der Funktion `loadCityTypeSystem()` das Laden eines Typsystems dargestellt werden kann.

— `demo/citytype.c`: (Forts.) —

```

60  cityTS.store ("cityts.t");
61  } // createCityTypeSystem()

```

Die Methode `G_typeSystem::store()` speichert das Typsystem in der Datei. Als Konvention gilt, daß Typsystemdateien mit der Erweiterung `.t` angelegt werden.

### 2.2.3 Erzeugen eines Graphen

Jetzt soll der Stadtplan in Abbildung 2.3 als Graph repräsentiert werden. Im nachfolgenden Programm wird dieser Graph aufgebaut und in einer Datei gespeichert.

— `demo/citymap.c`: —

```

3  #include <g_graph.h>
4  #include "citytype.h"
5

```

Wiederum werden zuerst die Deklarationen benötigt. Da das oben definierte Typsystem benutzt werden soll, werden auch die Deklarationen aus `citytype.h` geladen.

Das Hauptprogramm ruft zuerst die Funktion `createCityTypeSystem()` auf, damit alle benötigten Typen verfügbar sind.

```

— demo/citymap.c: (Forts.) —
6 int main (int, char *[])
7 {
8     createCityTypeSystem ();
9     G_graph g(cityTS);

```

Danach wird eine Graphinstanz angelegt und mit dem Typsystem verbunden. Das Typsystem ist über die Variable `cityTS` global verfügbar.

Für die zwölf Knoten der Abbildung werden die Variablen `v1` bis `v12` definiert.

```

— demo/citymap.c: (Forts.) —
10 G_vertex v1, v2, v3, v4, v5, v6, v7,
11         v8, v9, v10, v11, v12;
12
13 v1 = g.createVertex (tParkingPlace);
14 v2 = g.createVertex (tJunction);
15 v3 = g.createVertex (tParkingPlace);
16 v4 = g.createVertex (tJunction);
17 v5 = g.createVertex (tParkingUnderground);
18 v6 = g.createVertex (tJunction);
19 v7 = g.createVertex (tCrossing);
20 v8 = g.createVertex (tParkingHouse);
21 v9 = g.createVertex (tJunction);
22 v10 = g.createVertex (tParkingPlace);
23 v11 = g.createVertex (tJunction);
24 v12 = g.createVertex (tParkingUnderground);

```

Beim Erzeugen der Knoten muß nun der richtige Typ angegeben werden. Dabei werden auch die Attributwerte der einzelnen Knoten erzeugt und mit Nullwerten (siehe Abschn. 4.4.1, S. 113) initialisiert.

Die Kanten des Graphen, also die Straßen im Stadtplan, werden ebenfalls erzeugt.

```

— demo/citymap.c: (Forts.) —
25 g.createEdge (tStreet, v1, v2);
26 g.createEdge (tStreet, v2, v3);
27 g.createEdge (tStreet, v3, v4);
28 g.createEdge (tStreet, v4, v5);
29 g.createEdge (tStreet, v1, v6);
30 g.createEdge (tStreet, v2, v7);
31 g.createEdge (tStreet, v3, v8);
32 g.createEdge (tStreet, v4, v9);
33 g.createEdge (tStreet, v5, v10);
34 g.createEdge (tStreet, v6, v7);
35 g.createEdge (tStreet, v7, v8);
36 g.createEdge (tStreet, v8, v9);
37 g.createEdge (tStreet, v6, v11);
38 g.createEdge (tStreet, v7, v11);
39 g.createEdge (tStreet, v9, v12);
40 g.createEdge (tStreet, v11, v12);

```

Damit ist die Graphstruktur fertig. Es sind alle Knoten angelegt und über Kanten verbunden. Allerdings sind die Attributwerte aller Knoten noch mit den Nullwerten gefüllt.

Um die Attributwerte zu ändern, kann man mit den Methoden `getVAttr()` und `getEAttr()` auf die Werte der einzelnen Knoten- bzw. Kantenattribute zugreifen. Diese Methoden liefern eine Referenz auf den Wert zurück (`G_valueRef`).

Für das Beispiel werden zwei Hilfsvariablen benötigt.

```

— demo/citymap.c: (Forts.) —
41 G_valueRef val1, val2;

```

```

42
43     vall = g.getVAttr (v1, aName);
44     vall.updateSTRING ("Freibad");

```

In Zeile 44 wird eine Referenz auf den Wert des Attributes Name des Knotens `v1` geholt und in `vall` zwischengespeichert. Mit diesen Referenzen kann ähnlich umgegangen werden wie mit Zeigern in C++. `vall` enthält also keine *Kopie* des Attributwertes, sondern einen Zeiger auf die Stelle, an der der Attributwert gespeichert ist. Mit der Methode `G_valueRef::updateSTRING()` wird dem Wert ein neuer String zugewiesen.

Neben dem Zugriff auf die Attributwerte über die Attributvariablen kann man auch den Attributbezeichner benutzen. In der folgenden Zeile wird auf das Attribut Name des Knotens `v3` zugegriffen.

```

— demo/citymap.c: (Forts.) —
45     vall = g.getVAttr (v3, "Name");
46     vall.updateSTRING ("Stadion");
47
48     g.getVAttr(v10, aName).updateSTRING("Messeplatz");

```

Die vorhergehende Zeile zeigt, wie man auch ohne Hilfsvariablen auskommen kann. Der Rückgabewert der Methode `getVAttr()` wird direkt benutzt um die Methode `updateSTRING()` aufzurufen.

Wenn ein Knoten (bzw. eine Kante) mehrere Attribute hat, dann muß die Methode `getVAttr()` (bzw. `getEAttr()`) mehrfach aufgerufen werden.

```

— demo/citymap.c: (Forts.) —
49     vall = g.getVAttr (v5, aName);
50     vall.updateSTRING ("Universitaet");
51     vall = g.getVAttr (v5, aLevels);
52     vall.updateINT (1);

```

Bei strukturierten Attributwerten – in unserem Beispiel der Record des Attributes *Oeffnungszeit* – wird mit `getVAttr()` bzw. `getEAttr()` eine Referenz auf den gesamten Wert geliefert. Der Zugriff auf die einzelnen Komponenten wird über weitere Methoden realisiert.

```

— demo/citymap.c: (Forts.) —
53     vall = g.getVAttr (v5, aOpeningTime);
54     val2 = vall.getRecordItem ("Anfang");
55     val2.updateINT (6);
56     val2 = vall.getRecordItem ("Ende");
57     val2.updateINT (22);

```

Dasselbe ist noch für die Knoten `v8` und `v12` nötig.

```

— demo/citymap.c: (Forts.) —
58     g.getVAttr (v8, aName).updateSTRING ("Kaufhaus");
59     g.getVAttr (v8, aLevels).updateINT (5);
60     vall = g.getVAttr (v8, aOpeningTime);
61     vall.getRecordItem ("Anfang").updateINT (8);
62     vall.getRecordItem ("Ende").updateINT (18);
63
64     g.getVAttr (v12, aName).updateSTRING ("Opernplatz");
65     g.getVAttr (v12, aLevels).updateINT (3);
66     vall = g.getVAttr (v12, aOpeningTime);
67     vall.getRecordItem ("Anfang").updateINT (0);
68     vall.getRecordItem ("Ende").updateINT (24);

```

Zum Eintragen und Ändern der Kantenattribute muß der Attributwert mit der Methode `getEAttr()` ermittelt werden. In diesem Beispielprogramm werden jedoch keine Straßennamen eingetragen.

Zum Abschluß wird dieser Graph in einer Datei gespeichert.

```
— demo/citymap.c: (Forts.) —
69   g.store ("citymap.g");
70 } // main()
```

Hier gilt die Konvention, daß Graphdateien mit der Erweiterung `.g` enden.

## 2.2.4 Übersetzen und Binden

Zum Übersetzen und Binden wird das Makefile aus Abschnitt 2.1.2, S. 25, erweitert.

In den nächsten Zeilen wird festgelegt, daß sich das Programm `citymap` aus den beiden Teilen `citymap.o` und `citytype.o` zusammensetzt und wie es zu binden ist.

```
— demo/makefile: (Forts.) —
10
11 citymap: citymap.o citytype.o
12     $(GRALAB_LINK_o) -o citymap citymap.o citytype.o -l$(GRALAB_LIB)
13
14 citymap.o: citymap.c citytype.h
15 citytype.o: citytype.c citytype.h
```

Die letzten beiden Zeilen führen Abhängigkeiten ein. Z.B. wird festgelegt, daß die Datei `citymap.o` von den Dateien `citymap.c` und `citytype.h` abhängig ist. Wenn sich eine dieser beiden Dateien ändert, wird die Zielfeile neu erzeugt.

Um dieses Programm zu übersetzen und zu binden, muß beim Aufruf von `make` angegeben werden, welche Zielfeile erzeugt werden soll. Für dieses Beispielprogramm muß man also `make citymap` aufrufen, was beispielsweise folgende Ausgabe ergibt:

```
docs/demo> make citymap
g++ -Wall -fno-inline -g -I/home/ems/GraLab4/src/include \
-I/home/ems/GraLab4/lib/sparc-sun-solaris2.5/GNUgcc \
-DG_DEBUG -c citymap.c
g++ -Wall -fno-inline -g -I/home/ems/GraLab4/src/include \
-I/home/ems/GraLab4/lib/sparc-sun-solaris2.5/GNUgcc \
-DG_DEBUG -c citytype.c
g++ -L/home/ems/GraLab4/lib/sparc-sun-solaris2.5/GNUgcc \
-o citymap citymap.o citytype.o -lgraphcdtu
```

Das Programm kann danach mit `./citymap3` aufgerufen werden. Dabei wird keine Ausgabe am Bildschirm erzeugt, sondern nur die beiden Dateien `cityts.t` für das Typsystem und `citymap.g` für den Graphen angelegt.

## 2.3 Laden und Weiterverarbeiten von Graphen

Im vorhergehenden Abschnitt haben wir das Typsystem und den Graphen in Dateien abgespeichert. Diese Daten können in anderen Programmen wieder benutzt werden.

In der Headerdatei `citytype.h` wurde eine Signatur für die Funktion `loadCityTypeSystem()` eingeführt. Diese Funktion wird nun in der Datei `citytype.c` eingefügt. Sie wird von Programmen benutzt, die das Typsystem nicht mit der Funktion `createCityTypeSystem()` initialisieren.

```
— demo/citytype.c: (Forts.) —
```

---

<sup>3</sup> Der Bezug auf das aktuelle Verzeichnis mit `./` wird benutzt, da auf manchen Systemen im aktuellen Verzeichnis nicht nach Programmen gesucht wird.

```

62 void loadCityTypeSystem()
63 {
64     cityTS.load ("cityts.t");

```

Mit dem Aufruf von `G_typeSystem::load()` werden alle Informationen über das Typsystem aus der Datei `"cityts.t"` geladen und in `cityTS` eingetragen.

Damit ist das Typsystem geladen und benutzbar. Im Beispiel wurden aber auch globale Variablen eingeführt, über die einfach auf die Attribute und Typen des Typsystems zugegriffen werden kann. Diesen Variablen wurde bisher noch nichts zugewiesen.

— demo/citytype.c: (Forts.) —

```

65     aName = cityTS.getAttr("Name", G_domain::STRING);
66     aLevels = cityTS.getAttr("Ebenen", G_domain::INT);
67     aOpeningTime = cityTS.getFirstAttrById ("Oeffnungszeit");

```

Da es mehrere Attribute mit demselben Bezeichner aber unterschiedlichen Wertebereichen geben kann, muß jeweils der Bezeichner *und* Wertebereich angegeben werden. Das Attribut `Oeffnungszeit` hat einen konstruierten Wertebereich, der z.Zt. nicht direkt abrufbar ist. Deshalb wird dabei das erste Attribut im Typsystem benutzt, welches auf den Bezeichner paßt. Man geht davon aus, daß es kein anderes Attribut mit demselben Bezeichner aber anderem Wertebereich gibt.

Die Typvariablen werden analog behandelt.

— demo/citytype.c: (Forts.) —

```

68     tPoint = cityTS.getType ("Verbindungspunkt");
69     tBranching = cityTS.getType ("Verzweigung");
70     tCrossing = cityTS.getType ("Kreuzung");
71     tJunction = cityTS.getType ("Einmuendung");
72     tParking = cityTS.getType ("Parkeinrichtung");
73     tParkingBuilding = cityTS.getType ("Parkgebäude");
74     tParkingPlace = cityTS.getType ("Parkplatz");
75     tParkingUnderground = cityTS.getType ("Tiefgarage");
76     tParkingHouse = cityTS.getType ("Parkhaus");
77     tStreet = cityTS.getType ("Strasse");
78 }

```

Damit kann nun wieder ein Programm erstellt werden. Als kleines Beispiel werden alle Parkgebäude mit der Anzahl ihrer Parkebenen aufgelistet.

— demo/citylist.c: —

```

3 #include <iostream.h>
4 #include <g_graph.h>
5 #include "citytype.h"
6

```

Dieses ist wieder der normale Anfang des Programmtextes. Es werden die benötigten Deklarationen geladen.

— demo/citylist.c: (Forts.) —

```

7 int main (int, char *[])
8 {
9     loadCityTypeSystem ();
10    G_graph g(cityTS);
11    g.load ("citymap.g");

```

Im Hauptprogramm wird zuerst das Typsystem eingelesen. Dazu wird die vorhin definierte Funktion `loadCityTypeSystem()` benutzt. Mit diesem Typsystem wird ein Graph `g` angelegt und die Graphstruktur aus der Datei `citymap.g` geladen. Hierbei muß darauf geachtet werden, daß das Typsystem, mit dem die Graphdatei geschrieben wurde, zu dem Typsystem paßt, mit dem die Graphinstanz nun verbunden ist.

```

— demo/citylist.c: (Forts.) —
12  G_vertex v;
13  G_forAllVerticesOfClass (g, tParkingBuilding, v)
14  {
15      if (g.isAV (v, tParkingUnderground))
16          cout << "Die Tiefgarage ";
17      else
18          cout << "Das Parkhaus ";
19      cout << g.getVAttr (v, aName).getSTRING()
20          << " hat "
21          << g.getVAttr (v, aLevels).getINT()
22          << " Ebene(n)." << endl;
23  }
24  } // main()

```

Die Ausgabe des Programmes wird in einer Schleife erledigt. Dabei werden alle Parkgebäude, also Tiefgaragen oder Parkhäuser, aufgelistet. Hierfür kann man eine Schleife über die *Klasse* `Parkgebäude` benutzen. Dabei werden alle Knoten bearbeitet, die vom Typ `Parkgebäude` oder einem Untertyp davon sind. Mit der Methode `G_graph::isAV()` wird unterschieden, ob ein Knoten eine Tiefgarage oder – im anderen Fall – ein Parkhaus darstellt.

Für dieses kleine Programm wird wiederum unser Makefile erweitert:

```

— demo/makefile: (Forts.) —
16
17  citylist: citylist.o citytype.o
18      $(GRALAB_LINK_o) -o citylist citylist.o citytype.o -l$(GRALAB_LIB)
19
20  citylist.o: citylist.c citytype.h

```

Das Programm wird nun mit `make citylist` übersetzt und gebunden. Das Programm selbst wird mit `./citylist` gestartet und bringt folgende Ausgabe:

```

docs/demo> ./citylist
Die Tiefgarage Universitaet hat 1 Ebene(n).
Das Parkhaus Kaufhaus hat 5 Ebene(n).
Die Tiefgarage Opernplatz hat 3 Ebene(n).

```

## 2.4 Temporäre Attribute

Wie in der Einleitung schon geschildert (siehe Abschn. 1.1.6, S. 14), werden oftmals Kontrolldaten an Knoten oder Kanten benötigt. Z.B. benötigen viele Algorithmen Informationen darüber, welche Knoten oder Kanten schon besucht wurden. Diese Informationen sollten getrennt von den eigentlichen Knoten- und Kantenattributen verwaltet werden. Im Graphenlabor werden hierfür *temporäre Attribute* benutzt.

Ein temporäres Attribut ist dabei ein Objekt einer von `G_tempAttribute` abgeleiteten Klasse (*temporäre Attributierungs-klasse*). Eine solche Klasse kann folgende Methoden definieren:

- Eine virtuelle `print()`-Methode zur Ausgabe des Attributs bei `printVertex()` etc.
- Einen virtuellen Destruktor, falls das Attribut Zeiger auf dynamisch angeforderte Speicherbereiche enthält, die freigegeben werden sollten.

Wenn Veränderungen von Attributwerten mit dem *Undo*-Mechanismus protokolliert werden müssen, dann sind sie explizit zu programmieren (siehe Abschnitt 2.5.4, S. 42).

Temporäre Attribute werden in *Schichten* angelegt. Es gibt Schichten zur Attributierung von Knoten und Schichten zur Attributierung von Kanten. Es sind immer nur die Attribute der zuletzt erzeugten

Schicht für Knoten und die der zuletzt erzeugten Schicht für Kanten sichtbar.<sup>4</sup> Nach dem Erzeugen einer neuen Schicht trägt in dieser Schicht kein Graphenelement ein temporäres Attribut. Bei einzelnen (oder allen) Graphenelementen können nun temporäre Attribute eingetragen werden. Alle anderen Elemente tragen weiterhin kein temporäres Attribut. Eine Schicht kann auch Elemente unterschiedlicher Attributierungsklassen enthalten. Dabei muß das Verfahren wissen, welcher Klasse ein Element angehört.

### 2.4.1 Beispiel: Suche im Stadtplan

**Problemstellung:** *Es soll eine (ungerichtete) Tiefensuche in einem Graphen durchgeführt werden.*

Für die Tiefensuche nach TARJAN (1972) ist es wichtig zu wissen, ob und über welche Kante ein bestimmter Knoten schon besucht wurde. Diese Daten wollen wir als temporäre Attribute „an die Knoten schreiben“.

```
—demo/citytmp.c:—
3  #include <iostream.h> #include <g_graph.h> #include "citytype.h"
4
5  class dfsMarking : public G_tempAttribute
6  {
7  public:
8      int    number;
9      G_edge parent;
10
11     dfsMarking (unsigned n, G_edge p)
12     {
13         number = n;
14         parent = p;
15     }
16
17     virtual ostream & print (ostream &os)
18     {
19         return os << "Number = " << number
20             << ", Parent = " << parent;
21     }
22 };
```

Der Eintrag `number` gibt an, als wievielter Knoten der Knoten besucht wurde. `parent` ist die Kante, über die der Knoten im Suchbaum besucht wurde. Das `parent`-Feld enthält `G_EdgeBottom`, falls der Knoten ein Startknoten der Suche ist.<sup>5</sup> Wurde der fragliche Knoten noch nicht besucht, dann existiert kein temporäres Attribut an ihm.

Mit dem Konstruktor `dfsMarking()` werden Variablen des temporären Attributs initialisiert. Die virtuelle Methode `print()` gibt einen Klartext aus.

Weiter unten sind die Funktion `depthFirstSearch()` und `dfs()` definiert. An dieser Stelle werden ihre Signaturen deklariert. Die Funktion `indent()` wird zur Formatierung der Ausgabe benutzt.

```
—demo/citytmp.c: (Forts.)—
23 void depthFirstSearch (G_graph &g);
24 void dfs(G_graph &, int &, G_vertex, G_edge, int);
25
26 void indent(char c, int level)
27 {
```

<sup>4</sup> Die Schichten bilden zwei Stapel.

<sup>5</sup> Es handelt sich dann um den Wurzelknoten des in der Zusammenhangskomponente aufgespannten Baumes.

```

28   for (int i = 0; i < level; i++)
29       cout << c << c;
30   }

```

In der Funktion `depthFirstSearch()` wird zunächst eine neue Schicht für temporäre Knotenattribute mit der Methode `G_graph::createVTemp()` erzeugt. Dann wird in der Schleife nach einem unmarkierten Knoten gesucht. Dieser ist dann der Startknoten einer Zusammenhangskomponente, die mit der rekursiven Funktion `dfs()` abgesucht wird. Alle Knoten dieser Komponente sind danach markiert, so daß der nächste unmarkierte Knoten in einer noch nicht besuchten Komponente liegen muß. Wird kein unmarkierter Knoten mehr gefunden, dann ist die Suche abgeschlossen. Jetzt wird der temporär attributierte Graph ausgegeben und danach die Attributschicht gelöscht.

—demo/citytmp.c: (Forts.)—

```

31 void depthFirstSearch (G_graph &g)
32 {
33     g.createVTemp ();
34
35     G_vertex v;
36     int num = 0;
37     G_forAllVertices (g, v)
38     {
39         if (g.getPVTemp(v) == NULL)
40         {
41             cout << "Beginn der Komponente." << endl;
42             dfs (g, num, v, G_EdgeBottom, 0);
43             cout << "Ende der Komponente." << endl;
44         }
45     }
46     cout << g;
47
48     g.deleteVTemp ();
49 }

```

In der rekursiven Funktion `dfs()` wird der Graph `g` ab dem Knoten `v` abgesucht.

—demo/citytmp.c: (Forts.)—

```

50 void dfs(G_graph &g, int &num, G_vertex v, G_edge parent, int level)
51 {
52     dfsMarking *pM1, *pM2;
53
54     ++num;
55     pM1 = new dfsMarking (num, parent);
56     g.setPVTemp (v, pM1);
57     indent ('.', level);
58     cout << "Knoten " << v << endl;

```

Für diesen Knoten wird eine neue Markierung erzeugt. Die Variable `num` wird hochgezählt. Da sie der Funktion *by reference* übergeben wird, wird die lokale Variable in `depthFirstSearch()` geändert. Dieser Knoten wird noch auf den Bildschirm ausgegeben.

—demo/citytmp.c: (Forts.)—

```

59     G_edge e;
60     G_forAllIncidentEdges (g, v, e)
61     {
62         G_vertex w = g.thatV (e);
63         pM2 = (dfsMarking *)g.getPVTemp (w);
64         if (pM2 == NULL)
65         {
66             indent (' ', level);
67             cout << "--- Geruestkante " << e << endl;

```

```

68     dfs (g, num, w, e, level+1);
69     }
70     else if (!g.areEqualEdges (e, parent) &&
71             pM2->number < pM1->number)
72     {
73         indent (' ', level);
74         cout << "---- Sehne " << e << " nach " << w << endl;
75     }
76     }
77 }

```

Nun werden alle mit dem Knoten  $v$  inzidenten Kanten  $e$  untersucht. Wenn der Knoten  $w$  am anderen Ende der Kante  $e$  noch nicht besucht wurde, dann handelt es sich bei  $e$  um eine Gerüstkante. Es ist die Kante, über die der Knoten  $w$  im Suchbaum besucht wird. Die Suche wird rekursiv mit dem Aufruf der Funktion `dfs()` fortgesetzt. Wenn der Knoten schon früher über Gerüstkanten besucht wurde, dann ist die Kante  $e$  eine Sehne und wird als solche ausgegeben.

Zum Test des Verfahrens lesen wir den Stadtplangraphen aus dem Beispiel in Abschnitt 2.2.3, S. 30, ein und rufen das Verfahren auf.

— demo/citytmp.c: (Forts.) —

```

78 int main ()
79 {
80     loadCityTypeSystem ();
81     G_graph g(cityTS);
82
83     g.load ("citymap.g");
84
85     depthFirstSearch (g);
86
87     return 0;
88 }

```

Zum Übersetzen und Binden wird wieder das Makefile erweitert:

— demo/makefile: (Forts.) —

```

21
22 citytmp: citytmp.o citytype.o
23     $(GRALAB_LINK_o) -o citytmp citytmp.o citytype.o -l$(GRALAB_LIB)
24
25 citytmp.o: citytmp.c citytype.h

```

Mit diesem Programm erhält man folgende Ausgabe:

```

Beginn der Komponente.
Knoten v1
--- Geruestkante +e1
..Knoten v2
--- Geruestkante +e2
....Knoten v3
--- Geruestkante +e3
.....Knoten v4
--- Geruestkante +e4
.....Knoten v5
--- Geruestkante +e9
.....Knoten v10
--- Geruestkante +e8
.....Knoten v9
--- Geruestkante -e12
.....Knoten v8
--- Sehne -e7 nach v3
--- Geruestkante -e11
.....Knoten v7
--- Sehne -e6 nach v2
--- Geruestkante -e10
.....Knoten v6
--- Sehne -e5 nach v1
--- Geruestkante +e13
.....Knoten v11
--- Sehne -e14 nach v7

```

```

--- Geruestkante +e16
.....Knoten v12
--- Sehne -e15 nach v9
Ende der Komponente.
Graph: 12 of 1000 vertices, 16 of 1000 edges.
vertices:
vertex(1) [Parkplatz]
-> v2(+e1), v6(+e5),
<- .. ..
    Name: STRING: Freibad
temp attribute(1):
Number = 1, Parent = eBOT
vertex(2) [Einmuendung]
-> .., v3(+e2), v7(+e6),
<- v1(-e1), .. ..
temp attribute(1):
Number = 2, Parent = +e1
vertex(3) [Parkplatz]
-> .., v4(+e3), v8(+e7),
<- v2(-e2), .. ..
    Name: STRING: Stadion
temp attribute(1):
Number = 3, Parent = +e2

```

```

vertex(4) [Einmuendung]
-> ., v5(+e4), v9(+e8),
<- v3(-e3), ., .,
temp attribute(1):
Number = 4, Parent = +e3

vertex(5) [Tiefgarage]
-> ., v10(+e9),
<- v4(-e4), .,
  Ebenen: INT: 1
  Oeffnungszeit: RECORD(Anfang:INT,Ende:INT): (6,22)
  Name: STRING: Universitaet
temp attribute(1):
Number = 5, Parent = +e4

vertex(6) [Einmuendung]
-> ., v7(+e10), v11(+e13),
<- v1(-e5), ., .,
temp attribute(1):
Number = 10, Parent = -e10

vertex(7) [Kreuzung]
-> ., ., v8(+e11), v11(+e14),
<- v2(-e6), v6(-e10), ., .,
temp attribute(1):
Number = 9, Parent = -e11

vertex(8) [Parkhaus]
-> ., ., v9(+e12),
<- v3(-e7), v7(-e11), .,
  Ebenen: INT: 5
  Oeffnungszeit: RECORD(Anfang:INT,Ende:INT): (8,18)
  Name: STRING: Kaufhaus
temp attribute(1):
Number = 8, Parent = -e12

vertex(9) [Einmuendung]
-> ., ., v12(+e15),
<- v4(-e8), v8(-e12), .,
temp attribute(1):
Number = 7, Parent = +e8

vertex(10) [Parkplatz]
-> .,
<- v5(-e9),
  Name: STRING: Messeplatz
temp attribute(1):
Number = 6, Parent = +e9

vertex(11) [Einmuendung]
-> ., ., v12(+e16),
<- v6(-e13), v7(-e14), .,
temp attribute(1):
Number = 11, Parent = +e13

vertex(12) [Tiefgarage]
-> ., .,
<- v9(-e15), v11(-e16),
  Ebenen: INT: 3
  Oeffnungszeit: RECORD(Anfang:INT,Ende:INT): (0,24)
  Name: STRING: Opernplatz
temp attribute(1):
Number = 12, Parent = +e16

edges:
edge(1) [Strasse]: v1->v2
edge(2) [Strasse]: v2->v3
edge(3) [Strasse]: v3->v4
edge(4) [Strasse]: v4->v5
edge(5) [Strasse]: v1->v6
edge(6) [Strasse]: v2->v7
edge(7) [Strasse]: v3->v8
edge(8) [Strasse]: v4->v9
edge(9) [Strasse]: v5->v10
edge(10) [Strasse]: v6->v7
edge(11) [Strasse]: v7->v8
edge(12) [Strasse]: v8->v9
edge(13) [Strasse]: v6->v11
edge(14) [Strasse]: v7->v11
edge(15) [Strasse]: v9->v12
edge(16) [Strasse]: v11->v12

```

## 2.5 Undo

In Abschnitt 1.1.7, S. 15, wurde erwähnt, daß das Graphenlabor einen Undo-Mechanismus besitzt. Dabei werden alle Änderungen am Graphen automatisch protokolliert. Diese können später bestätigt oder rückgängig gemacht werden.

Bei der Realisierung des Undo wird davon ausgegangen, daß sich Änderungen an Graphen im wesentlichen durch Änderungen von Speicherinhalten beschreiben lassen. Somit genügt es, bei allen relevanten Zuweisungen (nicht bei Berechnung von Zwischenergebnissen in lokalen Variablen) den überschriebenen Wert und die Speicheradresse der betroffenen Variable in einem Protokollpuffer (Instanz der Klasse `G_undoBuffer`) mit den `G_undoBuffer::push()`-Methoden dieser Klasse zu sichern (vgl. Abb. 2.4, S. 40). Dies geschieht bei graphverändernden Laborfunktionen automatisch, falls der Graph mit einem Undo-Puffer verbunden worden ist.

Falls Zuweisungen (oder ein Teil davon) zurückgenommen werden sollen, durchläuft die Methode `G_undoBuffer::undo()` den Puffer rückwärts (in LIFO<sup>6</sup>-Reihenfolge) und schreibt in die geänderten Variablen wieder die alten Werte. Stehen die Änderungen (oder ein Teil davon) außer Frage und sollen daher bestätigt werden, kann mit der Methode `G_undoBuffer::commit()` die Protokollinformation (oder ein Teil davon) vorwärts (in FIFO<sup>7</sup>-Reihenfolge) wieder freigegeben werden. Man kann immer nur die letzten Änderungen zurücknehmen und immer nur die ersten, noch nicht bestätigten Änderungen dauerhaft machen.

<sup>6</sup> *last in first out*

<sup>7</sup> *first in first out*



## 2.5.2 Markierungspunkte

Um gezielt nur einen Teil der Änderungen zurückzunehmen oder zu bestätigen, kann man mit der Methode `G_undoBuffer::mark()` einen oder mehrere Markierungspunkte im Puffer setzen und sich mit dem `G_undoBuffer::commit()`- oder `G_undoBuffer::undo()`-Aufruf auf diese beziehen. Durch den zusätzlichen `unsigned`-Wert können Markierungspunkte unterscheidbar gemacht werden.

## 2.5.3 Verzögerte Aktionen

Leider gibt es einige Aktionen, die sich nicht als Veränderung von Speicherinhalten beschreiben lassen. Dazu gehören z.B. das Anfordern und Freigeben von dynamischem Speicherplatz, Bildschirmausgaben oder das Verändern von Dateien. Um in diesen Fällen möglichst flexibel verfahren zu können, erlaubt das Graphenlabor das Speichern von *Aktionen* im Puffer, die erst bei Aufruf von `commit()` oder `undo()` ausgeführt werden sollen. Technisch erfolgt das durch Angabe eines Zeigers auf eine Funktion, die mit einem ebenfalls anzugebenden Parameter des Typs (`void *`) zum gegebenen Zeitpunkt (bei einem `commit()`- oder `undo()`-Aufruf) aufgerufen wird.

**Beispiel 1: Freigabe von Speicherplatz** Ein Speicherbereich soll freigegeben werden. Da er nach einem `undo()`-Aufruf aber wieder zur Verfügung stehen soll<sup>8</sup>, darf man ihn nicht sofort freigeben, sondern muß den `commit()`-Aufruf abwarten:

```
G_undoBuffer uB;
...
// free(pointer);                <-- dieser Aufruf soll
                                   erst bei commit
                                   durchgeführt werden

uB.pushCommitAction(free, pointer);
```

Die gespeicherten Aktionsfunktionen müssen alle mit einem `void *`-Parameter auskommen. Sind von der Anwendung her mehrere Parameter erforderlich, müssen sie als „Parameterpaket“ im dynamischen Speicher abgelegt werden:

**Beispiel 2: Verwendung eines Parameterpakets** Die Auswirkungen der Funktion `draw(1, 2, 3)` lassen sich durch die Funktion `reverseDraw(pParams)` rückgängig machen, falls `pParams` auf ein Parameterpaket mit den Werten `(1, 2, 3)` zeigt. Da nur Funktionen als Aktionen auf den Undo-Puffer geschrieben werden können, muß der Aufruf des `delete`-Operators für das Parameterpaket `pParams` in einer Funktion stehen. Hierfür dient die Funktion `deleteParams()`.<sup>9</sup> Mit der folgenden Anweisungsfolge kann der Undo-Puffer entsprechend belegt werden:

```
G_undoBuffer uB;
...
draw(1, 2, 3);                    <-- soll bei Undo
                                   beruecksichtigt
                                   werden

pParams = new drawParams(1, 2, 3);
uB.pushAction(deleteParams, (void *)pParams);
uB.pushUndoAction(reverseDraw, (void *)pParams);
```

<sup>8</sup> Der Speicherbereich muß darüberhinaus noch an derselben Adresse liegen und denselben Inhalt haben.

<sup>9</sup> Die Funktionen `reverseDraw()` und `deleteParams()` und die Struktur des Parameterpakets müssen mit der Funktion `draw()` abgestimmt werden.

Durch den `pushAction()`-Aufruf wird die Freigabe des Parameterpakets im Puffer vermerkt, die sowohl beim `commit()`- als auch beim `undo()`-Aufruf notwendig ist. Beim `undo()` soll aber zuvor noch `reverseDraw()` aufgerufen werden, was durch den `pushUndoAction()`-Aufruf eingetragen wird. Da beim `undo()` der Puffer rückwärts abgearbeitet wird, müssen die Aktionen in umgekehrter Reihenfolge auf den Puffer geschrieben werden.<sup>10</sup>

## 2.5.4 Änderung von Werten temporärer Attribute

Änderungen von Werten *temporärer Attribute* werden nicht ohne weiteres protokolliert, da sie nicht von den durch das Labor bereitgestellten Methoden durchgeführt werden. Hier muß man durch Aufruf von `push()`-Methoden dafür sorgen, daß die Änderungen protokolliert werden.

**Beispiel 3: Attributwertänderung** Durch die folgende Sequenz wird ein Attributwert geändert:

```
G_undoBuffer uB;
G_graph g;
G_vertex v;
...
pAttr = (myAttribute *)g.getPVTemp(v);

uB.pushAssignInt(pAttr -> intValueXYZ); <-- hier werden Variable
                                         und alter Wert
                                         protokolliert

pAttr -> intValueXYZ = 17;                <-- diese Zuweisung muss
                                         protokolliert werden
```

Der Methode `pushAssignInt()` wird die zu protokollierende Variable *call-by-reference* übergeben. Damit kann im Undo-Puffer die Adresse *und* der aktuelle Wert der Variablen vermerkt werden.

## 2.5.5 Logische Destruktoren bei temporären Attributen

Wenn ein Graph mit einem Undo-Puffer verbunden ist, dann darf der Speicherplatz für temporäre Attribute nicht sofort mit dem Löschen von Graphenelementen freigegeben werden. Er kann erst dann freigegeben werden, wenn das Löschen mit `G_undoBuffer::commit()` bestätigt wird. Deshalb ruft das Graphenlabor bei `G_graph::deleteVertex()` und `G_graph::deleteEdge()` die Destruktoren der betroffenen Attributobjekte nicht sofort auf, falls der Graph mit einem Undo-Puffer verbunden ist. Der Destruktoraufruf erfolgt erst beim bestätigenden `commit()`-Aufruf.

Damit die Attributobjekte aber erfahren, daß sie logisch eigentlich schon gelöscht sind, ruft das Labor unmittelbar bei `deleteVertex()` und `deleteEdge()` sozusagen als „logischen Destruktor“ die virtuelle Methode `G_tempAttribute::logicalDelete()` auf, die bei Bedarf entsprechend überladen (durch Redefinition in der Attributklasse) werden kann. Wenn danach das Löschen durch einen Aufruf der Methode `G_undoBuffer::undo()` zurückgenommen wird, wird die virtuelle Methode `G_tempAttribute::logicalUndelete()` aufgerufen, um dem Objekt mitzuteilen, daß es doch nicht gelöscht wurde. Auch diese Methode kann bei Bedarf überladen werden. Bei der Definition von Attributklassen sollte der „logische“ Anteil der beim Löschen des Objekts erforderlichen Aktionen in die Methode `logicalDelete()` verlagert werden. Im eigentlichen Destruktor sollte nur noch die Freigabe der zugeordneten Speicherbereiche stehen.

Falls der Graph nicht mit einem Undo-Puffer verbunden ist, wird beim Löschen von Attributen immer zuerst der logische, dann der normale Destruktor aufgerufen.

<sup>10</sup> Bei Aktionen für `commit()` muß die normale Reihenfolge verwendet werden.

**Beispiel 4: Logische Destruktoren** Die Attributklasse `graphicalAttribute` beschreibe Attribute, die in irgendeiner Form auf dem Bildschirm dargestellt werden. Dazu gebe es eine Methode `show()`, die die Attributdarstellung erzeugt, und eine Methode `hide()`, mit der die Darstellung gelöscht wird. Falls nun Knoten oder Kanten mit Attributen dieser Klasse gelöscht werden, sollte die grafische Darstellung verschwinden. Dies gilt auch dann, wenn die Löschung wieder zurückgenommen werden soll. Daher muß ein logischer Destruktor definiert werden, der zum richtigen Zeitpunkt `hide()` aufruft. Analog muß beim Zurücksetzen des Löschens das Attribut wieder grafisch dargestellt werden.

```
void graphicalAttribute::logicalDelete()
{
    hide();
}

void graphicalAttribute::logicalUndelete()
{
    show();
}
```

## 2.6 Traversieren und Behandlung von orientierten Kanten

Für die Beispiele in diesem Abschnitt wird von dem Graph in Abbildung 2.5, S. 44, ausgegangen. Im Beispielprogramm muß zuerst ein solcher Graph aufgebaut werden. Dazu wird ähnlich wie in Abschnitt 2.1, S. 22, vorgegangen.

```
— demo/orient.c: —
3 #include <iostream.h>
4 #include <g_graph.h>
5
6 G_typeSystem ts;
7 G_graph g(ts);
8 G_vertex v1, v2, v3, v4, v5, v6;
9 G_edge e1, e2, e3, e4, e5;
```

Da der Graph und die Knoten- bzw. Kantenreferenzen in mehreren Funktionen benötigt werden, sind sie hier global definiert.

```
— demo/orient.c: (Forts.) —
10 void buildTree ()
11 {
12     G_type tNull;
13     tNull = ts.G_TypeNull();
14
15     g.reInitialize ();
16
17     v1 = g.createVertex (tNull);
18     v2 = g.createVertex (tNull);
19     v3 = g.createVertex (tNull);
20     v4 = g.createVertex (tNull);
21     v5 = g.createVertex (tNull);
22     v6 = g.createVertex (tNull);
23
24     e1 = g.createEdge (tNull, v1, v2);
25     e2 = g.createEdge (tNull, v1, v3);
26     e3 = g.createEdge (tNull, v4, v1);
27     e4 = g.createEdge (tNull, v2, v5);
28     e5 = g.createEdge (tNull, v2, v6);
29 }
```

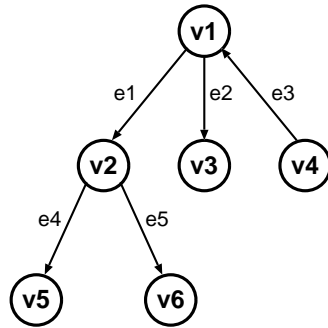


Abbildung 2.5: Der Graph am Anfang

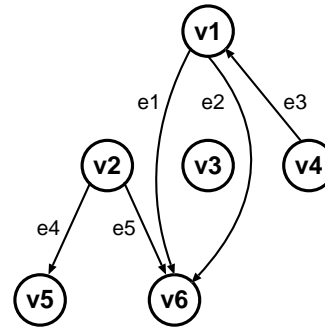


Abbildung 2.6: Nach exChangeOmega ( )

Da bei jedem Aufruf von `buildTree()` dieselbe Graphstruktur erzeugt werden soll, wird der Graphinhalt zuerst mit `G_graph::reInitialize()` gelöscht und initialisiert. Danach wird die Graphstruktur in Abbildung 2.5, S. 44, aufgebaut. Für dieses Beispielprogramm gilt dann wieder, daß die internen Identifikationsnummern mit den Nummern in der Abbildung übereinstimmen.

### 2.6.1 Ändern von Start- oder Zielknoten von Kanten

**Problemstellung:** *Alle Kanten, die aus dem Knoten v1 herausführen, sollen so umgelegt werden, daß sie danach zum Knoten v6 führen.*

Da bei dieser Operation die Folge  $\Lambda(v1)$  nicht verändert wird, kann das Makro `G_forAllOutEdges` benutzt werden. Das Umlegen zu einem anderen Knoten wird dadurch realisiert, daß die Endknoten  $\omega(e)$  der Kanten  $e \in \Lambda^+(v1)$  geändert wird. Man erhält den Graphen in Abbildung 2.6.

— demo/orient.c: (Forts.) —

```

30 void exChangeOmega()
31 {
32     G_edge e;
33     G_forAllOutEdges (g, v1, e)
34     {
35         g.changeOmega (e, v6);
36     }
37 }

```

**Problemstellung:** *In dem Graph in Abb. 2.6 soll der Knoten v1 gelöscht werden. Davor sollen alle inzidenten Kanten zum Knoten v4 umgelegt werden.*

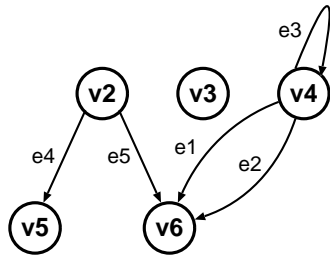
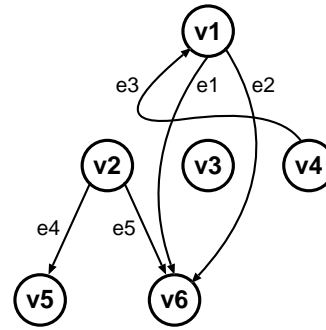
Das Makro `G_forAllIncidentEdges` kann hier nicht benutzt werden, weil durch das Umlegen der Kanten werden sie aus der Inzidenzfolge der Knotens v1 gelöscht und somit wird  $\Lambda(v1)$  verändert. Deshalb wird mit `G_graph::first()` die erste zum Knoten v1 inzidente Kante ermittelt und zum Knoten v4 umgelegt. Dieses wird fortgesetzt, bis es keine Kanten mehr gibt, die zum Knoten v1 inzident sind.

— demo/orient.c: (Forts.) —

```

38 void exChangeThis()
39 {
40     G_edge e;
41     e = g.first (v1);
42     while (e != G_EdgeBottom)
43     {
44         g.changeThis (e, v4);

```

Abbildung 2.7: Nach `exChangeThis()`Abbildung 2.8: Nach `exPutBefore()`

```

45     e = g.first (v1);
46   }
47   g.deleteVertex (v1);
48 }

```

Für das *Umlegen* der Kante wird die Methode `G_graph::changeThis()` benutzt. Da die Kante  $e$  vom Knoten  $v1$  aus betrachtet wird, wird das Ende, welches bisher am Knoten  $v1$  war, zum Knoten  $v4$  umgelegt. Das folgende Programmstück würde dasselbe bewirken.

```

if (g.isNormal (e))
    g.changeAlpha (e, v4);
else
    g.changeOmega (e, v4);

```

Das Ergebnis der Funktion `exChangeThis()` ist in Abbildung 2.7 dargestellt.

## 2.6.2 Reihenfolgen der Kanten ändern

Für dieses Beispiel wird der Graph neu aufgebaut und die Funktion `exChangeOmega()` darauf ausgeführt. Dadurch erhält man wieder den Graphen in Abbildung 2.6, S. 44. Die Reihenfolge der Kanten  $e1$ ,  $e2$  und  $e3$  am Knoten  $v1$  in dieser Abbildung entspricht auch seiner Inzidenzfolge.

$$\Lambda(v1) = \langle (e1, out), (e2, out), (e3, in) \rangle$$

—demo/orient.c: (Forts.)—

```

49 void exPutBefore()
50 {
51     buildTree ();
52     exChangeOmega ();

```

**Problemstellung:** Für eine Kante  $e$ , die vom Knoten  $v4$  zum Knoten  $v1$  führt, soll die Inzidenzfolge am Knoten  $v1$  so umgestellt werden, daß sie unmittelbar vor der ersten aus dem Knoten  $v1$  ausgehenden Kante  $f$  steht.

—demo/orient.c: (Forts.)—

```

53     G_edge e, f;
54     e = g.edgeFromTo (v4, v1);
55     if (e == G_EdgeBottom)
56         return;
57     f = g.firstOut (v1);
58     if (f == G_EdgeBottom)
59         return;
60     g.putBefore (g.reverse (e), f);
61 } // exPutBefore()

```

Wenn es keine Kante  $e$  von  $v_4$  nach  $v_1$  oder keine aus dem Knoten  $v_1$  ausgehende Kante  $f$  gibt, dann kann die Funktion abgebrochen werden. Ansonsten müssen die Kanten  $e$  und  $f$  in der Inzidenzliste des Knotens  $v_1$  umsortiert werden. Dazu müssen die *orientierten Kanten* so der Methode `G_graph::putBefore()` übergeben werden, daß sie vom Knoten  $v_1$  aus betrachtet werden. Da `G_graph::edgeFromTo()` die Kante  $e$  in *normalisierter Orientierung* liefert, also vom Anfangsknoten  $v_4$  aus gesehen, muß die Orientierung dieser Kante umgedreht werden.

Das Ergebnis der Funktion `exPutBefore()` ist in Abbildung 2.8, S. 45, dargestellt.

### 2.6.3 Testlauf dieser Beispiele

Für einen Test und eine Protokollausgabe wird aus diesen Funktionen ein Programm erstellt.

— demo/orient.c: (Forts.) —

```
62 int main ()
63 {
64     buildTree ();
65     cout << "Der Graph am Anfang" << endl << g << endl;
66     exChangeOmega ();
67     cout << "nach exChangeOmega()" << endl << g << endl;
68     exChangeThis ();
69     cout << "nach exChangeThis()" << endl << g << endl;
70     exPutBefore ();
71     cout << "nach exPutBefore()" << endl << g << endl;
72     return 0;
73 }
```

Das Makefile wird wieder um zwei Zeilen erweitert.

— demo/makefile: (Forts.) —

```
26 orient: orient.o
27     $(GRALAB_LINK_o) -o orient orient.o -l$(GRALAB_LIB)
```

Dieses Programm erzeugt dann folgende Ausgabe:

```
docs/demo> make orient
g++ -Wall -fno-inline -g -I/home/ems/GraLab4/src/include \
-I/home/ems/GraLab4/lib/sparc-sun-solaris2.5/GNUgcc \
-DG_DEBUG -c orient.c
g++ -L/home/ems/GraLab4/lib/sparc-sun-solaris2.5/GNUgcc \
-o orient orient.o -lgraphcdu
docs/demo> ./orient
Der Graph am Anfang
Graph: 6 of 1000 vertices, 5 of 1000 edges.
vertices:
vertex(1) [TypeNull]
-> v2(+e1), v3(+e2), ..
<- .., .., v4(-e3),

vertex(2) [TypeNull]
-> .., v5(+e4), v6(+e5),
<- v1(-e1), .., ..

vertex(3) [TypeNull]
-> ..
<- v1(-e2),

vertex(4) [TypeNull]
-> v1(+e3),
<- ..

vertex(5) [TypeNull]
-> ..
<- v2(-e4),

vertex(6) [TypeNull]
-> ..
<- v2(-e5),

edges:
edge(1) [TypeNull]: v1->v2
edge(2) [TypeNull]: v1->v3
edge(3) [TypeNull]: v4->v1
edge(4) [TypeNull]: v2->v5
edge(5) [TypeNull]: v2->v6

nach exChangeOmega()
Graph: 6 of 1000 vertices, 5 of 1000 edges.
vertices:
vertex(1) [TypeNull]
-> v6(+e1), v6(+e2), ..
<- .., .., v4(-e3),

vertex(2) [TypeNull]
-> v5(+e4), v6(+e5),
<- .., ..

vertex(3) [TypeNull]
->
<-

vertex(4) [TypeNull]
-> v1(+e3),
<- ..

vertex(5) [TypeNull]
-> ..
<- v2(-e4),

vertex(6) [TypeNull]
-> .., .., ..
<- v2(-e5), v1(-e1), v1(-e2),

edges:
edge(1) [TypeNull]: v1->v6
```

```
edge(2) [TypeNull]: v1->v6
```

```
edge(3) [TypeNull]: v4->v1
```

```
edge(4) [TypeNull]: v2->v5
```

```
edge(5) [TypeNull]: v2->v6
```

```
nach exChangeThis()
```

```
Graph: 5 of 1000 vertices, 5 of 1000 edges.
```

```
vertices:
```

```
vertex(2) [TypeNull]
```

```
-> v5(+e4), v6(+e5),
```

```
<- ., .,
```

```
vertex(3) [TypeNull]
```

```
->
```

```
<-
```

```
vertex(4) [TypeNull]
```

```
-> v4(+e3), v6(+e1), v6(+e2), .,
```

```
<- ., ., ., v4(-e3),
```

```
vertex(5) [TypeNull]
```

```
-> .,
```

```
<- v2(-e4),
```

```
vertex(6) [TypeNull]
```

```
-> ., ., .,
```

```
<- v2(-e5), v4(-e1), v4(-e2),
```

```
edges:
```

```
edge(1) [TypeNull]: v4->v6
```

```
edge(2) [TypeNull]: v4->v6
```

```
edge(3) [TypeNull]: v4->v4
```

```
edge(4) [TypeNull]: v2->v5
```

```
edge(5) [TypeNull]: v2->v6
```

```
nach exPutBefore()
```

```
Graph: 6 of 1000 vertices, 5 of 1000 edges.
```

```
vertices:
```

```
vertex(1) [TypeNull]
```

```
-> ., v6(+e1), v6(+e2),
```

```
<- v4(-e3), ., .,
```

```
vertex(2) [TypeNull]
```

```
-> v5(+e4), v6(+e5),
```

```
<- ., .,
```

```
vertex(3) [TypeNull]
```

```
->
```

```
<-
```

```
vertex(4) [TypeNull]
```

```
-> v1(+e3),
```

```
<- .,
```

```
vertex(5) [TypeNull]
```

```
-> .,
```

```
<- v2(-e4),
```

```
vertex(6) [TypeNull]
```

```
-> ., ., .,
```

```
<- v2(-e5), v1(-e1), v1(-e2),
```

```
edges:
```

```
edge(1) [TypeNull]: v1->v6
```

```
edge(2) [TypeNull]: v1->v6
```

```
edge(3) [TypeNull]: v4->v1
```

```
edge(4) [TypeNull]: v2->v5
```

```
edge(5) [TypeNull]: v2->v6
```

# Kapitel 3

## Nutzung des Graphenlabors

### 3.1 Verzeichnisstruktur des Graphenlabors

Alle Dateien des Graphenlabors liegen unter einem zentralen Verzeichnis. Dieses *Wurzelverzeichnis*<sup>1</sup> wird in diesem Dokument mit `$(GRALAB_HOME)` abgekürzt. Darunter gibt es mehrere Unterverzeichnisse mit den einzelnen Dateien. Die Bezeichnung *arch/comp* steht für eine bestimmte Systemarchitektur und einen Compiler (siehe Anhang C, S. 140).

**src** Alle Quelltexte zum Graphenlabor.  
**src/include** Die Headerdateien zum Übersetzen von Anwendungen.  
**src/include/inline** Methoden, die je nach gewählter Variante entweder als separate Funktionen oder als *inline* expandierter Code implementiert sind.  
**lib/arch/comp** Alle Bibliotheken und andere Dateien, die vom jeweiligen System und Compiler abhängen (z.B. Header-Dateien mit systemabhängigen Defines).  
**msg** Alle Meldungsdateien für die Laufzeitmeldungen des Graphenlabors.  
**example** Ein einfaches Beispielprogramm.  
**docs** Die Quelltexte dieser Dokumentation.  
**docs/demo** Einige Beispielprogramme aus dieser Dokumentation.  
**build/arch/comp** Verzeichnisse, in denen die Bibliotheken für die unterschiedlichen Systeme übersetzt werden.

Eine ausführliche Liste der Verzeichnisstruktur befindet sich in Anhang F, S. 156.

Für Installationen auf anderen Maschinen sollte diese Verzeichnisstruktur beibehalten bleiben.

### 3.2 Übersetzen und Binden von Anwendungssoftware

#### 3.2.1 Übersetzen

Zum Übersetzen von Quelltextdateien zu GraLab-Anwendungen müssen diese die Datei `g_graph.h` im Verzeichnis `$(GRALAB_INCDIR)` mit dem `#include`-Befehl einfügen. Am einfachsten geschieht dies durch den Befehl `#include <g_graph.h>` im Quelltext und einem Makefile, welches den Aufruf des Compilers mit den Optionen `-I$(GRALAB_INCDIR) -I$(GRALAB_LIBDIR)` enthält.

---

<sup>1</sup> An der Universität in Koblenz `/home/ems/GraLab4`.

### 3.2.2 Binden

Zum Binden einer GraLab-Anwendung zu einem lauffähigen Programm muß eine der Bibliotheken `libgraphxxx.a` aus dem Verzeichnis `$(GRALAB_LIBDIR)` verwendet werden. Die *Varianten* (siehe Anhang D.1, S. 142) sind:

**libgraphcdtu.a** Das ist die Bibliothek für noch in der Entwicklung befindliche Programme. Sie erlaubt uneingeschränkt Checking und Tracing. Sie enthält auch die Debugging-Information für die Graphenlaborfunktionen.

**libgraphpu.a** Das ist die Bibliothek zum Ermitteln der Laufzeiteigenschaften von Programmen. Sie enthält Profiling-Informationen für die Graphenlaborfunktionen, damit man die Programme mit `gprof` analysieren kann. Sie vollzieht kein Checking der den Laborfunktionen übergebenen Parameter und erlaubt weder Tracing noch Debugging der Laborfunktionen. Damit Anwendungen mit dieser Bibliothek gebunden werden können, müssen sie mit den Optionen `-DNO_CHK` `-DNO_TRC` übersetzt werden.

**libgraphu.a** Das ist die Bibliothek für ausgetestete Anwendungen. Sie vollzieht kein Checking der den Laborfunktionen übergebenen Parameter und erlaubt weder Tracing noch Debugging der Laborfunktionen. Damit Anwendungen mit dieser Bibliothek gebunden werden können, müssen sie mit den Optionen `-DNO_CHK` `-DNO_TRC` übersetzt werden.

### 3.2.3 Hilfe durch Makefiles

Die fehlerfreie Funktion des Graphenlabors kann nur dann garantiert werden, wenn die Parameter beim Compilieren der Quelltexte und beim Binden des Programmes richtig angegeben sind. Damit sich hier keine Fehler einschleichen, wird empfohlen, für eigene Anwendungsprogramme Makefiles zu erstellen. In diese Makefiles können Dateien (*Makefile-Templates*) eingeladen werden, die Variablen für den richtigen Compiler- und Linkeraufruf definieren. Sie sind an das jeweilige Betriebssystem, den Compiler und die Bibliotheksvariante angepaßt. Im Anhang D.3, S. 145, sind diese Variablen beschrieben.

Im nachfolgenden Beispiel soll das Programm `prog`, bestehend aus den Teilen `a.c`, `b.c` und `c.c` erzeugt werden.

```

— use/makefile: —
3 GRALAB_HOME = /home/ems/GraLab4
4 GRALAB_ARCH = sparc-sun-solaris2.5/GNUgcc
5 GRALAB_VARIANT = cdtu
6
7 include $(GRALAB_HOME)/lib/$(GRALAB_ARCH)/make$(GRALAB_VARIANT).tpl
8
9 .c.o:
10     $(GRALAB_COMPILE_c) $<
11
12 OBJFILES = a.o b.o c.o
13
14 prog : $(OBJFILES)
15     $(GRALAB_LINK_o) -o prog $(OBJFILES) -l$(GRALAB_LIB)

```

Mit diesem Makefile wird das Programm `prog` durch das Kommando

```
make prog
```

übersetzt und gebunden. Wenn das Programm fertig entwickelt ist, kann mit demselben Makefile die optimierte Variante des Programmes erzeugt werden. Hierzu müssen zuerst alle Objekt-Dateien gelöscht werden. Danach wird das Programm mit der `u`-Variante erzeugt:

```
rm *.o
make prog GRALAB_VARIANT=u
```

Selbst dann, wenn man eine Kopie der Graphenlabor-Dateien in einem anderen Verzeichnis verwendet, kann dieses Makefile benutzt werden. Hierzu muß die Verzeichnisstruktur aus Abschnitt 3.1, S. 48, beibehalten sein. Wenn das Graphenlabor unter dem Verzeichnis `/myGraLab` installiert ist, sieht der `make`-Aufruf wie folgt aus:

```
make prog GRALAB_HOME=/myGraLab
```

### 3.3 Nutzung von Environment-Variablen

Das Verhalten des Graphenlabors wird durch einige Environment-Variablen beeinflusst. Z.B. existiert eine Variable `G_NMAX`, die angibt, wieviele Knoten ein Graph zunächst enthalten kann. Falls diese Variablen nicht definiert sind, wird mit Defaultwerten gearbeitet. Die Defaultwerte sind als Konstanten gleichen Namens in `g_config.h` definiert.

Welche Variablen durch diesen Mechanismus behandelt werden und welche Bedeutung sie haben, kann man Anhang B, S. 139, entnehmen.

### 3.4 Meldungen

Die Funktionen des Graphenlabors erzeugen Warn- bzw. Fehlermeldungen auf dem Standardfehlergerät `stderr`, sofern sie eine unerwartete oder eine fehlerhafte Situation erkennen. Dies ist z.B. der Fall, wenn eine Datei nicht geöffnet werden oder der Inhalt einer Datei nicht interpretiert werden kann. Sofern *Checking* (siehe Abschnitt 3.6, S. 53) eingeschaltet ist, erzeugen Graphenlaborfunktionen auch bei offensichtlich unsinnigen Parametern Meldungen.

Die Meldungen werden in drei Klassen eingeteilt:

**Warnungen** werden bei unerwarteten Situationen ausgegeben, die jedoch nicht notwendigerweise einen Fehler darstellen.

**Fehlermeldungen** werden in Fehlersituationen ausgegeben, in denen das System evtl. noch sinnvoll die Abarbeitung fortsetzen kann. Diese Fehlersituationen werden gezählt und das Programm solange fortgesetzt, bis eine maximale Fehlerzahl erreicht ist. Dann wird das Programm (mit `exit(1)`) abgebrochen. Die maximale Fehlerzahl kann durch die Umgebungsvariable `G_MAXERROR` (siehe Anhang B, S. 139) eingestellt werden.

**fatale Fehlermeldungen** treten auf, wenn die Fehlersituation so schwerwiegend ist, daß die Fortsetzung des Programms wahrscheinlich sinnlos ist. In diesem Fall wird das Programm unverzüglich (mit `exit(2)`) abgebrochen.

Jede Meldung besteht aus folgenden Teilen:

- die Art (Warnung, Fehler oder fataler Fehler) des Fehlers,
- die Funktion, die die Situation erkannt hat,
- die Beschreibung der Situation (dazu können evtl. aktuelle Parameter der Funktion zählen, welche die Situation erkannt hat),
- das Verhalten des Programms in dieser Situation und
- eine Empfehlung, wie diese Situation vermieden werden kann.

Eine Meldung wird durch eine *Gruppenbezeichnung*<sup>2</sup> und eine Meldungsnummer identifiziert. Da-

<sup>2</sup> Unter einer Gruppe ist hier (nur) eine logisch zusammenhängende Menge von Funktionen und Methoden zu verstehen.

durch wird es ermöglicht, dieselben Nummern in unterschiedlichen Gruppen für unterschiedliche Meldungen zu verwenden.

Die nachfolgende Fehlermeldung wurde mit dem folgendem Programm erzeugt:

```
#include <g_graph.h>
int main (int, char *[])
{
    G_typeSystem ts;
    G_graph g(ts);
    G_type tNull = ts.G_TypeNull();
    G_vertex v, w;
    v = g.createVertex (tNull);
    w = g.createVertex (tNull);
    g.deleteVertex (w);
    g.createEdge (tNull, v, w);
}
```

Es werden zuerst zwei Knoten *v* und *w* angelegt, von denen einer wieder gelöscht wird. An diesem Punkt enthält die Variable *w* einen Hinweis auf einen Knoten im Graph *g*, der nicht mehr existiert. Der Versuch, eine Kante zwischen *v* und *w* zu ziehen muß fehlschlagen.

```
***** ERROR grf007 in G_graph::createEdge(TypeNull,v1,v2)
***** Condition found : 'v2' is an unused vertex in graph 'bffff500'
***** Action taken : function aborted
***** Reaction proposed: call function with an existing vertex
```

Die erste Zeile gibt die Meldungsart (hier: ERROR, also eine normale *Fehlermeldung*), die den Fehler feststellende Funktion (`G_graph::createEdge()`) sowie Gruppenbezeichnung (*grf*) und Meldungsnummer (007) an. Die zweite Zeile beschreibt das Verhalten des Systems, nämlich den Abbruch der Funktion. Die dritte Zeile gibt eine Empfehlung, wie dieser Fehler zu vermeiden ist: man sollte die Funktion mit Knoten aufrufen, die im Graph existieren.

Die Texte aller Meldungen sind nicht im Quelltext des Graphenlabors fest vorgegeben. Die für alle Meldungen gleichen Texte

```
***** MESSAGE
***** ERROR
***** FATAL ERROR
***** Condition found :
***** Action taken : und
***** Reaction proposed :
```

sowie das Format (gemäß `printf()`), in dem Meldungsdatei, Meldungsnummer und Funktion ausgegeben werden („ %s%03d in %s“), werden der Datei *msgems* in einem der in der Environmentvariable *G\_MSGPATH* angegebenen Verzeichnisse entnommen. Mit der Methode `addDirectory()` (der Klasse *G\_msg*) können weitere Verzeichnisse angegeben werden. Die für die Meldung spezifischen Texte (diese enthalten je nach Meldung ebenfalls Formatangaben gemäß `printf()` für die Ausgabe von Parametern etc.) befinden sich in einer der Dateien *msggroup* (im Beispiel *msggrf*). Jede Gruppe hat somit eine eigene Meldungsdatei. Dort werden die Texte anhand der Meldungsnummer (im Beispiel 7) gesucht. Zum Format dieser Dateien siehe Anhang A.3, S. 137.

Sollen die Meldungen in einer anderen Sprache als Englisch erfolgen, sind die Texte in diesen Dateien entsprechend zu ändern oder ein Verzeichnis mit entsprechenden geänderten Dateien in der Environment-Variable *G\_MSGPATH* anzugeben.

Das Graphenlabor beinhaltet Funktionen, mit denen eigene Programme Meldungen in analoger Weise erzeugen können. Zu Details siehe Abschnitt 4.9, S. 128.

## 3.5 Tracing

Das Graphenlabor enthält einige Funktionen und Makros, die eine Laufzeitverfolgung (im weiteren nur *Tracing* genannt) einer GraLab-Anwendung während ihrer Entwicklung ermöglichen. Einige Funktionen geben dann zur Laufzeit in eine Datei aus, in welcher Reihenfolge sie mit welchen Parametern in welcher Schachtelungstiefe aufgerufen werden und welchen Wert sie zurückgeben. Dabei ist zu beachten, daß jeweils für alle Funktionen, die in derselben Quelltextdatei *xxx.c* definiert sind, nur zusammen Tracing ein- bzw. ausgeschaltet werden kann.

### 3.5.1 Wie kann Tracing eingeschaltet werden ?

Zunächst muß die Anwendung, für die Tracing genutzt werden soll, mit einer Bibliotheksvariante gebunden werden, die Tracing unterstützt (z.B. *libgraphcdtu.a*). Dann kann mit der Methode `G_trace::set()` für einzelne Quelltextdateien Tracing ein- bzw. ausgeschaltet werden. Die Methode erwartet einen Dateinamen<sup>3</sup> und einen logischen `bool`-Wert. Ist der `bool`-Wert gleich `true`, wird Tracing für die angegebenen Quelltextdateien ein-, andernfalls ausgeschaltet. Mit einer gleichnamigen Methode mit nur einem logischen `bool`-Parameter kann Tracing auch ganz aus- bzw. eingeschaltet werden.

Die Tracing-Ausgaben können mit der Methode `G_trace::setFile()` auch in eine Datei umgeleitet werden.

### 3.5.2 Wie können Funktionen Tracing-fähig programmiert werden?

Das Graphenlabor stellt (über *g\_graph.h*) die Makros `G_trc`, `G_trcEnter`, `G_trcLeave` und `G_TrcDeclaration` zur Verfügung. Das Makro `G_TrcDeclaration` definiert lokal in der `.c`-Datei ein Steuerobjekt, in dem zur Laufzeit festgehalten wird, ob Tracing für diese Quelltext-Datei ein- oder ausgeschaltet ist.

Die übrigen Makros werden mit einem Stück C-Code verwendet: `G_trcEnter(codeFragment)`, `G_trcLeave(codeFragment)` bzw. `G_trc(codeFragment)`.

Die Makros testen zur Laufzeit, ob momentan Tracing für die aktuelle Funktion (bzw. die Quelltextdatei, in der diese definiert wird) eingeschaltet ist. Ist dies der Fall, werden gemäß der Schachtelungstiefe von Funktionsaufrufen Einrückungszeichen in die Tracing-Datei ausgegeben und dann die in *codeFragment* angegeben C++-Anweisungen ausgeführt. Dies werden in der Regel Ausgabeanweisungen in den Tracing-Stream sein, der über die Zeigervariable `G_trace::out` vom Typ `ostream*` erreicht wird. Zusätzlich verwalten `G_trcEnter` und `G_trcLeave` noch einen internen Zähler für die Schachtelungstiefe.

Dazu ein Beispiel:

Gegeben sei folgende Funktion `exampleFunc`:

```
G_TrcDeclaration;
...
int exampleFunc (int x, char *txt)
{
    int result;
    FILE *fp;

    G_trcEnter( *G_Trace.out << "exampleFunc("
```

<sup>3</sup> Es muß der komplette Dateipfad oder entsprechende Auslassungszeichen (wildcards) angegeben werden.

```

                                << x << ", " << txt
                                << ')'; )

...
name="xyz.dat";
fp=fopen(name, flags);
if (fp==NULL)
{
    G_trc( *G_Trace.out << "cannot open file \"
                                << name << '\"'; )
}
...
result = ... ;
G_trcLeave( *G_Trace.out << result; )
}

```

Wenn Tracing für diese Funktion eingeschaltet ist und sie in der Schachtelungstiefe 3 mit den Parametern `x=4`, `txt="abcdef"` aufgerufen wird, die Datei `xyz.dat` nicht öffnen kann und den Wert `result=12` berechnet, werden in die Tracing-Datei folgende Ausgaben geschrieben:

```

| | | exampleFunc(4, abcdef)
| | | | cannot open file "xyz.dat"
| | | ->12<-

```

Um bei voll ausgetesteten und für fehlerfrei befundenen Anwendungen volle Ausführungsgeschwindigkeit zu erreichen, ist es nicht notwendig, alle Tracing-Makros aus den Quelltexten zu entfernen. Es genügt, alle Quelltexte mit der zusätzlichen Definition `NO_TRC` zu übersetzen (siehe Abschnitt 3.2, S. 48). Dann löscht der Präprozessor alle Tracing-Makros, so daß zur Laufzeit in Bezug auf Tracing nichts mehr geschieht.

## 3.6 Checking

Um die Entwicklung von Anwendungen mit dem Graphenlabor zu erleichtern, prüft nahezu jede Laborfunktion ihre Aufrufparameter hinsichtlich ihrer Plausibilität und erzeugt ggf. Fehlermeldungen. Damit dieses *Checking* durchgeführt werden kann, muß die Anwendung mit einer Bibliotheksvariante gebunden werden, die es unterstützt (z.B. `libgraphcdtu.a`). Checking ist dann eingeschaltet, kann aber zur Laufzeit über die globale, logische Variable `G_Chk` vom Typ `int` mit dem Wert `false` aus- und mit dem Wert `true` eingeschaltet werden.

In eigenen Programmteilen kann derselbe Checking-Mechanismus verwendet werden. Dazu müssen die Prüfanweisungen mit dem Makro `G_chk` eingeklammert werden:

```

/* "normale" Anweisungen
...
*/

G_chk(
    /* checking Anweisungen */
    ...
)

/* weitere normale Anweisungen */
...
*/

```

Das Makro `G_chk` ist in `g_graph.h` definiert und sorgt dafür, daß die geklammerten Anweisungen nur dann ausgeführt werden, wenn die Variable `G_Chk` mit einem Wert  $\neq 0$  belegt ist. Die geklammerten Anweisungen werden mittels bedingter Compilierung gelöscht, falls der Quelltext mit der Compileroption `-DNO_CHK` übersetzt wird, so daß dann in Bezug auf Checking zur Laufzeit nichts mehr geschieht.

# Kapitel 4

## Referenz-Handbuch

Zur Beschreibung der Klassen des Graphenlabors werden zunächst die benötigten Typen vorgestellt. Abbildung 4.1, S. 56, gibt einen Überblick über die Zusammenhänge zwischen den Klassen. Dabei sind Konstrukte, die nur intern im Graphenlabor zur Speicherung der Daten benutzt werden, wiederum als gestrichelte Klassen dargestellt. Die abstrakten Klassen (kursiver Klassenname) werden nur für die einfachere Darstellung des Klassendiagramms benutzt. Für die Klassen `G_domain` und `G_valueRef` gibt es in den Abbildungen 1.8, S. 20, und 1.9, S. 21, detaillierte Klassendiagramme.

Es folgen, nach Gruppen geordnet, Beschreibungen der zur Verfügung gestellten Methoden. Die Einteilung nach *Gruppen* entspricht der Aufteilung der Methoden auf die Quelltexte, in denen sie definiert sind. Da der Name des zugeordneten Quelltextes für das Ein-/Ausschalten des Tracing (siehe Abschn. 3.5, S. 52) wichtig ist, findet sich dieser in der Überschrift der entsprechenden Gruppe wieder.

### 4.1 Graphen

Die Klasse `G_graph` realisiert die eigentliche Datenstruktur Graph. Instanzen dieser Klasse sind Graphen. Intern werden alle Knoten- und Kanteninformationen in Tabellen gespeichert. Die Längen dieser Tabellen geben an, wieviele Knoten bzw. Kanten der Graph gerade enthalten kann. Reichen die Tabellen durch Erzeugen eines Knotens bzw. einer Kante nicht mehr aus, werden sie verdoppelt.

#### 4.1.1 Typen und Nullwerte

Die Klasse `G_graph` arbeitet mit folgenden Datentypen:

**G\_vertex:** (siehe Abschnitt 4.1.3, S. 58)

Alle Knoten werden in Instanzen der Klasse `G_graph` gespeichert. Die Knotenvariablen vom Typ `G_vertex` werden benutzt, um einen einzelnen Knoten in einem Graphen zu adressieren.

Der undefinierte Knoten ist `G_VerxBottom` und wird von einigen Methoden als Rückgabewert im Fehlerfall verwendet. Die Identifikationsnummer eines Knotens erhält man durch Anwendung der Methode `G_graph::getVNo()` auf diesen Knoten. Umgekehrt erhält man den Knoten mit einer bestimmten Identifikationsnummer mit der Methode `G_graph::getV()`.

**G\_edge** (siehe Abschnitt 4.1.3, S. 59)

Alle Kanten werden in Instanzen der Klasse `G_graph` gespeichert. Die Kantenvariablen vom Typ `G_edge` werden benutzt, um eine einzelne Kante in einem Graphen zu adressieren. Für den

Abbildung 4.1: Überblick über die Klassen des Graphenlabors

Zugriff auf gerichtete Graphen wird auch die Orientierung der Kante<sup>1</sup> in der Kantenvariablen gespeichert.

Die undefinierte Kante ist `G_EdgeBottom` und wird von einigen Methoden als Rückgabewert im Fehlerfall verwendet. Die Identifikationsnummer einer Kante mit Orientierungsvorzeichen erhält man durch Anwendung der Methode `G_graph::getENo()` auf diese Kante. Umgekehrt erhält man die Kante mit einer bestimmten Identifikationsnummer mit der Methode `G_graph::getE()`.

Ferner nimmt `G_graph` Bezug auf folgende Klassen, die in eigenen Abschnitten beschrieben werden:

**G\_typeSystem, G\_type, G\_attr:** (siehe Abschnitt 4.2, S. 96)

Jedem Graphen ist ein Typsystem zugeordnet, in dem alle verwendeten Typen und Attribute beschrieben ist.

**G\_valueRef:** (siehe Abschnitt 4.4, S. 113)

Die Attributwerte der Graphenelemente können abgefragt und verändert werden.

**G\_tempAttribute:** (siehe Abschnitt 4.6, S. 124)

Knoten oder Kanten eines temporär attributierten Graphen können ein temporäres Attribut tragen.

**G\_undoBuffer:** (siehe Abschnitt 4.5, S. 120)

Einem Graphen kann ein Undopuffer zur automatischen Protokollierung von Änderungen zugeordnet sein.

#### 4.1.2 Graphen anlegen und löschen: `g_grfall.c`

Graphen werden mit dem Konstruktor `G_graph` angelegt und mit dem Destruktor `~G_graph` gelöscht. Außerdem kann man mit der Methode `reInitialize()` eine bestehende Graphinstanz neu initialisieren.

```
G_graph::G_graph ( const G_typeSystem &typeSystem )
G_graph::G_graph ( const G_typeSystem &typeSystem, unsigned vM )
G_graph::G_graph ( const G_typeSystem &typeSystem, unsigned vM,
                    unsigned eM )
```

Initialisiert eine Graphinstanz. Der Graph ist leer, d.h. er enthält keine Knoten und keine Kanten. Er wird mit dem Typsystem `typeSystem` verbunden. Die Parameter `vM` und `eM` geben die gewünschten (Start-)Größen der internen Tabellen an.

##### Parameter:

**unsigned vM** : Initiale Knotenanzahl; wenn nicht angegeben, wird der Wert der Umgebungsvariablen `G_NMAX` oder ein vordefinierter Wert gesetzt (siehe Anhang B, S. 139).

**unsigned eM** : Initiale Kantenanzahl; wenn nicht angegeben, wird der Wert der Umgebungsvariablen `G_MMAX` oder ein vordefinierter Wert gesetzt (siehe Anhang B, S. 139).

**Fehlerfälle:** Es wird eine der folgenden Meldungen ausgegeben.

**grf022** Die Tabellen für die Knoteninformationen wurden verkleinert.

**grf023** Die Tabellen für die Kanteninformationen wurden verkleinert.

---

<sup>1</sup> siehe Abschnitt 1.1.3, S. 11,

**G\_graph::~~G\_graph ( )**

Löscht die Graphinstanz mit allen Knoten und Kanten. Bei Graphen mit temporären Attributen werden die Destruktoren aller temporären Attribute aufgerufen. Anschließend wird der für die internen Tabellen reservierte Speicherplatz wieder freigegeben.

```
void G_graph::reInitialize ( )
```

```
void G_graph::reInitialize ( unsigned vM )
```

```
void G_graph::reInitialize ( unsigned vM, unsigned eM )
```

Löscht die bestehende Graphinstanz und initialisiert sie neu. Der Graph ist weiterhin mit dem ursprünglichen Typsystem verbunden, enthält danach aber keine Knoten und Kanten. Die Parameter *vM* und *eM* geben die gewünschten (Start-)Größen der internen Tabellen an.

**Parameter:**

**unsigned vM** : Initiale Knotenanzahl; wenn nicht angegeben, wird der Wert der Umgebungsvariablen *G\_NMAX* oder ein vordefinierter Wert gesetzt (siehe Anhang B, S. 139).

**unsigned eM** : Initiale Kantenanzahl; wenn nicht angegeben, wird der Wert der Umgebungsvariablen *G\_MMAX* oder ein vordefinierter Wert gesetzt (siehe Anhang B, S. 139).

**Fehlerfälle:** Es wird eine der folgenden Meldungen ausgegeben.

**grf022** Die Tabellen für die Knoteninformationen wurden verkleinert.

**grf023** Die Tabellen für die Kanteninformationen wurden verkleinert.

**4.1.3 Graphenelemente**

Die Daten der Knoten und Kanten eines Graphen werden in einer Instanz der Klasse *G\_graph* gespeichert. Für den Zugriff auf diese Informationen müssen Instanzen der Klassen *G\_vertex* und *G\_edge* benutzt werden.

**Knoten: g\_vertex.c****G\_vertex::G\_vertex ( )**

Erzeugt eine neue Knotenvariable und initialisiert sie mit *G\_VertexBottom*.

**Aufwand:**  $O(1)$

Da Instanzen dieser Klasse immer über einen Konstruktor initialisiert werden, kann diese Klasse nicht in *union*-Datentypen benutzt werden. Um trotzdem Knotenvariablen in *union*-Datentypen, wie sie z.B. bei Parsergeneratoren eingesetzt werden, benutzen zu können, gibt es eine Klasse *\_G\_vertex*, deren Instanzen nicht initialisiert werden. Variablen der Typen *G\_vertex* und *\_G\_vertex* können beliebig an Stellen, an denen ein *G\_vertex* erwartet wird, benutzt werden.

**Konstanten:** Es gibt eine Konstante vom Typ *G\_vertex:G\_VertexBottom*. Sie wird z.B. als Fehlerrückmeldung bei Methoden eingesetzt.

```
bool G_vertex::operator== ( const G_vertex &v ) const
bool G_vertex::operator!= ( const G_vertex &v ) const
```

Testet ob diese Instanz auf den selben (nicht auf den selben) Knoten wie die Variable `v` verweist. Dabei werden nur die Inhalte der Variablen überprüft. Es ist nicht garantiert, daß die Knoten in einem Graphen existieren.

**Aufwand:**  $O(1)$ , inline

```
unsigned G_vertex::hashVal ( ) const
```

Liefert einen Hash-Wert der Knotenvariablen. Dieser kann benutzt werden, um Knoten in Hash-Tabellen einzufügen (z.B bei Symboltabellen).

**Bemerkung:** Der Wert wird mit der Funktion `multiplicativehash()` (KNUTH, 1973, S. 508) berechnet.

**Aufwand:**  $O(1)$

```
ostream& G_vertex::print ( ostream &os ) const
ostream& operator<< ( ostream &os, const G_vertex &v )
```

Gibt die Kantenvariable `v` formatiert auf den Stream `os` aus. Es wird ein `v` mit nachfolgender Identifikationsnummer erzeugt. Wenn `v==G_VertexBottom` ist, dann wird "vBOT" ausgegeben.

**Aufwand:**  $O(1)$

```
istream& G_vertex::read ( istream &is )
istream& operator>> ( istream &is, G_vertex &v )
```

Liest eine formatierte Knotenvariable von dem Stream `is`.

**Aufwand:**  $O(1)$

## Kanten: `g_edge.c`

```
G_edge::G_edge ( )
```

Erzeugt eine neue Kantenvariable und initialisiert sie mit `G_EdgeBottom`.

**Aufwand:**  $O(1)$

Da Instanzen dieser Klasse immer über einen Konstruktor initialisiert werden, kann diese Klasse nicht in `union`-Datentypen benutzt werden. Um trotzdem Kantenvariablen in `union`-Datentypen, wie sie z.B. bei Parsergeneratoren eingesetzt werden, benutzen zu können, gibt es eine Klasse `_G_edge`, deren Instanzen nicht initialisiert werden. Variablen der Typen `G_edge` und `_G_edge` können beliebig an Stellen, an denen ein `G_edge` erwartet wird, benutzt werden.

**Konstanten:** Es gibt eine Konstante vom Typ `G_edge: G_EdgeBottom`. Sie wird z.B. als Fehlerrückmeldung bei Methoden eingesetzt.

```
bool G_edge::operator== ( const G_edge &e ) const
bool G_edge::operator!= ( const G_edge &e ) const
```

Testet ob diese Instanz auf die selbe (nicht auf die selbe) Kante wie die Variable *e* verweist und die selbe (oder nicht die selbe) Orientierung haben. Dabei werden nur die Inhalte der Variablen überprüft. Es ist nicht garantiert, daß die Kanten in einem Graphen existieren.

**Aufwand:**  $O(1)$ , inline

```
unsigned G_edge::hashVal ( ) const
```

Liefert einen Hash-Wert der Kantenvariablen. Dieser kann benutzt werden, um Kanten in Hash-Tabellen einzufügen (z.B bei Symboltabellen). Dabei geht die Orientierung der Kante in die Berechnung des Hashwertes mit ein. I.A. gilt

$$e.hashVal() \neq g.reverse(e).hashVal()$$

**Bemerkung:** Der Wert wird mit der Funktion `multiplicativehash()` (KNUTH, 1973, S.508) berechnet.

**Aufwand:**  $O(1)$

```
ostream& G_edge::print ( ostream &os ) const
ostream& operator<< ( ostream &os, const G_edge &e )
```

Gibt die Kantenvariable *e* formatiert auf den Stream *os* aus. Es wird zuerst ein Vorzeichen + oder - ausgegeben, welches die Orientierung der Kante angibt. Danach kommt *e* und die Identifikationsnummer. Wenn `e==G_EdgeBottom` ist, dann wird "eBOT" ausgegeben.

**Aufwand:**  $O(1)$

```
istream& G_edge::read ( istream &is )
istream& operator>> ( istream &is, G_edge &e )
```

Liest eine formatierte Kantenvariable von dem Stream *is*.

**Aufwand:**  $O(1)$

```
ostream& G_edge::printNormal ( ostream &os ) const
```

Gibt die Kantenvariable *e* formatiert auf den Stream *os* aus. Dabei geht die Information über die Orientierung der Kante verloren. Es wird ein *e* und die Identifikationsnummer ausgegeben. Wenn `e==G_EdgeBottom` ist, dann wird "eBOT" ausgegeben.

**Aufwand:**  $O(1)$

#### 4.1.4 Graphstruktur manipulieren: `g_grfman.c`

Mit folgenden Methoden können Knoten oder Kanten hinzugefügt und wieder gelöscht werden. Außerdem können Start- und Endknoten von Kanten geändert werden. Zu Identifikationsnummern können die zugehörigen Knoten oder Kanten gefunden werden und umgekehrt.

G\_vertex **G\_graph::createVertex** ( G\_type aType )

G\_vertex **G\_graph::createVertex** ( G\_type aType, unsigned vNo )

Erzeugt im Graph einen neuen Knoten vom Typ aType und gibt diesen zurück. Der Knoten ist isoliert, das heißt mit keiner Kante inzident, und trägt keine temporären Attribute. In die Folge aller Knoten des Graphen wird der neue Knoten als letzter eingetragen. Wenn vNo fehlt, gibt das Labor dem Knoten willkürlich eine freie Identifikationsnummer. Reichen die internen Tabellen nicht für einen weiteren Knoten aus, werden sie automatisch vergrößert.

**Fehlerfälle:** Es wird G\_VertexBottom zurückgegeben und eine der folgenden Meldungen ausgegeben.

**grf001** Die Nummer vNo liegt außerhalb des Bereiches, der durch die internen Tabellen abgedeckt ist.

**grf002** Die Nummer vNo ist schon für einen anderen Knoten vergeben.

**grf005** Die Vergrößerung der internen Tabellen ist wegen Speichermangel nicht möglich.

**grf038** Durch einen Programmfehler sind inkonsistente Informationen über die internen Tabellen entstanden.

**grf040** Die Speicherstruktur für die Attributwerte konnte wegen Speichermangel nicht angelegt werden.

**grf042** Der Typ aType wurde noch nicht exportiert und kann deshalb noch nicht benutzt werden.

**Aufwand:** Bei willkürlicher Vergabe der Identifikationsnummer:  $O(1)$

Bei Angabe einer Identifikationsnummer mit vNo:  $O(vMax)$ , da in in der Liste der freien Knoten die Nummer vNo gesucht wird.

Der Aufwand für die Vergrößerung der internen Tabellen ist nicht berücksichtigt.

G\_edge **G\_graph::createEdge** ( G\_type aType, G\_vertex alpha, G\_vertex  
omega )

G\_edge **G\_graph::createEdge** ( G\_type aType, int eNo, G\_vertex alpha,  
G\_vertex omega )

Erzeugt eine gerichtete Kante vom Typ aType von Knoten alpha nach Knoten omega und gibt diese in normaler Orientierung zurück. Die Kante trägt keine temporären Attribute. In die Folge aller Kanten des Graphen wird die neue Kante als letzte eingetragen. Auch in die Folgen  $\Lambda(\alpha)$  und  $\Lambda(\omega)$  wird die neue Kante als letzte eingetragen.<sup>2</sup> Wenn eNo fehlt, gibt das Labor der Kante willkürlich eine freie Identifikationsnummer. Reichen die internen Tabellen nicht für eine weitere Kante aus, werden sie automatisch vergrößert.

**Fehlerfälle:** Es wird G\_EdgeBottom zurückgegeben und eine der folgenden Meldungen ausgegeben.

**grf003** Die Nummer eNo liegt außerhalb des Bereiches, der durch die internen Tabellen abgedeckt ist.

**grf004** Die Nummer eNo ist schon für eine andere Kante vergeben.

**grf005** Die Vergrößerung der internen Tabellen ist wegen Speichermangel nicht möglich.

**grf007** Eine der Knotenvariablen alpha oder omega beinhaltet keinen Knoten des Graphen.

**grf039** Durch einen Programmfehler sind inkonsistente Informationen über die internen Tabellen entstanden.

**grf041** Die Speicherstruktur für die Attributwerte konnte wegen Speichermangel nicht an-

<sup>2</sup> Handelt es sich um eine Schlinge, wird die Kante erst als out-Kante und dann als in-Kante an die Folge angehängt.

gelegt werden.

**grf043** Der Typ `aType` wurde noch nicht exportiert und kann deshalb noch nicht benutzt werden.

**Aufwand:** Bei willkürlicher Vergabe der Identifikationsnummer:  $O(1)$

Bei Angabe einer Identifikationsnummer mit `eNo`:  $O(eMax)$ , da in in der Liste der freien Kanten die Nummer `eNo` gesucht wird.

Der Aufwand für die Vergrößerung der internen Tabellen ist nicht berücksichtigt.

```
void G_graph::deleteVertex ( G_vertex v )
```

Löscht den Knoten `v` aus dem Graph. Alle mit `v` inzidenten Kanten werden durch Aufrufe von `deleteEdge()` ebenfalls gelöscht. Falls vorhanden werden die Destruktoren der temporären Attribute aller Schichten aufgerufen.

**Fehlerfälle:** Es wird folgende Meldung ausgegeben.

**grf007** Der Knoten `v` existiert nicht im Graphen.

**Aufwand:**  $O(|\Lambda(v)|) + O(\#Schichten\ temporärer\ Knotenattribute)$

```
void G_graph::deleteEdge ( G_edge e )
```

Löscht die Kante `e`. Falls vorhanden werden die Destruktoren der temporären Attribute aller Schichten aufgerufen.

**Fehlerfälle:** Es wird folgende Meldung ausgegeben.

**grf008** Die Kante `e` existiert nicht im Graphen.

**Aufwand:**  $O(|\Lambda(\alpha(e))|) + O(|\Lambda(\omega(e))|) + O(\#Schichten\ temporärer\ Kantenattribute)$

```
bool G_graph::changeAlpha ( G_edge e, G_vertex v )
```

Legt für die (bereits vorhandene) Kante `e` den neuen Startknoten `v` fest. Dazu wird `e` aus  $\Lambda(\alpha(e))$  entfernt und als out-Kante an  $\Lambda(v)$  angehängt. Stimmt `v` mit dem alten Startknoten von `e` überein, wird die Kante in  $\Lambda(v)$  an das Ende verschoben.

**Rückgabewerte:** `true`, wenn die Operation erfolgreich durchgeführt wurde.

**Fehlerfälle:** Es wird `false` zurückgegeben und eine der folgenden Meldungen ausgegeben.

**grf007** Der Knoten `v` existiert nicht im Graphen.

**grf008** Die Kante `e` existiert nicht im Graphen.

**Aufwand:**  $O(|\Lambda(\alpha(e))|)$ .

```
bool G_graph::changeOmega ( G_edge e, G_vertex v )
```

Legt für die (bereits vorhandene) Kante `e` den neuen Zielknoten `v` fest. Dazu wird `e` aus  $\Lambda(\omega(e))$  entfernt und als in-Kante an  $\Lambda(v)$  angehängt. Stimmt `v` mit dem alten Zielknoten von `e` überein, wird die Kante in  $\Lambda(v)$  an das Ende verschoben.

**Rückgabewerte:** `true`, wenn die Operation erfolgreich durchgeführt wurde.

**Fehlerfälle:** Es wird `false` zurückgegeben und eine der folgenden Meldungen ausgegeben.

**grf007** Der Knoten `v` existiert nicht im Graphen.

**grf008** Die Kante  $e$  existiert nicht im Graphen.

**Aufwand:**  $O(|\Lambda(\omega(e))|)$ .

`bool G_graph::changeThis ( G_edge e, G_vertex v )`

Legt für die orientierte Kante  $e$  den neuen *this*-Knoten  $v$  fest.<sup>3</sup>

**Rückgabewerte:** `true`, wenn die Operation erfolgreich durchgeführt wurde.

**Fehlerfälle:** Es wird `false` zurückgegeben und eine der folgenden Meldungen ausgegeben.

**grf007** Der Knoten  $v$  existiert nicht im Graphen.

**grf008** Die Kante  $e$  existiert nicht im Graphen.

**Aufwand:**  $O(|\Lambda(this(e))|)$ .

`bool G_graph::changeThat ( G_edge e, G_vertex v )`

Legt für die orientierte Kante  $e$  den neuen *that*-Knoten  $v$  fest.<sup>4</sup>

**Rückgabewerte:** `true`, wenn die Operation erfolgreich durchgeführt wurde.

**Fehlerfälle:** Es wird `false` zurückgegeben und eine der folgenden Meldungen ausgegeben.

**grf007** Der Knoten  $v$  existiert nicht im Graphen.

**grf008** Die Kante  $e$  existiert nicht im Graphen.

**Aufwand:**  $O(|\Lambda(that(e))|)$ .

#### 4.1.5 Zugriff auf Typinformationen und Attributewerte: `g_grfatr.c`

`const G_typeSystem & G_graph::getTypeSystem ( ) const`

Liefert das mit dem Graph verbundene Typsystem.

**Rückgabewerte:** Eine Referenz auf das Typsystem. Dieses Typsystem darf nicht verändert werden.

**Aufwand:**  $O(1)$ , inline

`G_type G_graph::getVType ( G_vertex v ) const`

Liefert den Typ des Knotens  $v$ .

**Fehlerfälle:** Es wird `G_TypeBottom` zurückgegeben und folgende Meldung ausgegeben.

**grf007** Der Knoten  $v$  existiert nicht im Graph.

**Aufwand:**  $O(1)$ , inline

`G_type G_graph::getEType ( G_edge e ) const`

Liefert den Typ der Kante  $e$ .

**Fehlerfälle:** Es wird `G_TypeBottom` zurückgegeben und folgende Meldung ausgegeben.

<sup>3</sup> Ist  $e$  normal orientiert, wird  $\alpha(e)$ , ansonsten  $\omega(e)$  geändert.

<sup>4</sup> Ist  $e$  normal orientiert, wird  $\omega(e)$ , ansonsten  $\alpha(e)$  geändert.

**grf008** Die Kante  $e$  existiert nicht im Graph.

**Aufwand:**  $O(1)$ , inline

```
bool G_graph::isAV ( G_vertex v, G_type aType ) const
```

Testet, ob der Knoten  $v$  der Klasse  $aType$  angehört.

**Rückgabewerte:** `true` wenn  $type(v)$  isA\*  $aType$ , `false` sonst.

**Fehlerfälle:** Es wird `false` zurückgegeben und eine der folgenden Meldungen ausgegeben.

**grf007** Der Knoten  $v$  existiert nicht im Graph.

**grf016** Der Typ  $aType$  ist nicht in dem Typsystem eingetragen, welches mit dem Graph verbunden ist.

**Aufwand:**  $O(1)$ , inline

```
bool G_graph::isAE ( G_edge e, G_type aType ) const
```

Testet, ob die Kante  $e$  der Klasse  $aType$  angehört.

**Rückgabewerte:** `true` wenn  $type(e)$  isA\*  $aType$ , `false` sonst.

**Fehlerfälle:** Es wird `false` zurückgegeben und eine der folgenden Meldungen ausgegeben.

**grf008** Die Kante  $e$  existiert nicht im Graph.

**grf016** Der Typ  $aType$  ist nicht in dem Typsystem eingetragen, welches mit dem Graph verbunden ist.

**Aufwand:**  $O(1)$ , inline

```
bool G_graph::changeVType ( G_vertex v, G_type newType )
```

Tragt  $newType$  als neuen Typ des Knotens  $v$  ein. Dabei wird auch das *Attributierungsschema* von  $newType$  übernommen. Attributwerte, bei denen Attributbezeichner *und* Wertebereich übereinstimmen, bleiben erhalten. Alle anderen Werte werden gelöscht bzw. initialisiert.

**Fehlerfälle:** Es wird `false` zurückgegeben und eine der folgenden Meldungen ausgegeben.

**grf007** Der Knoten  $v$  existiert nicht im Graph.

**grf044** Der Typ  $newType$  ist nicht exportiert.

**grf045** Für die neuen Attributwerte konnte kein Speicher alloziert werden.

**typ002** Der Typ  $newType$  existiert nicht im Typsystem.

**Aufwand:**  $O(|\text{AttributSchema}(type(v))| * |\text{AttributSchema}(newType)|)$

```
bool G_graph::changeEType ( G_edge e, G_type newType )
```

Tragt  $newType$  als neuen Typ der Kante  $e$  ein. Dabei wird auch das *Attributierungsschema* von  $newType$  übernommen. Attributwerte, bei denen Attributbezeichner *und* Wertebereich übereinstimmen, bleiben erhalten. Alle anderen Werte werden gelöscht bzw. initialisiert.

**Fehlerfälle:** Es wird `false` zurückgegeben und eine der folgenden Meldungen ausgegeben.

**grf008** Die Kante  $e$  existiert nicht im Graph.

**grf044** Der Typ  $newType$  ist nicht exportiert.

**grf045** Für die neuen Attributwerte konnte kein Speicher alloziert werden.

**typ002** Der Typ `newType` existiert nicht im Typsystem.

**Aufwand:**  $O(|\text{AttributSchema}(\text{type}(e))| * |\text{AttributSchema}(\text{newType})|)$

```
G_valueRef G_graph::getVAttr ( G_vertex v, G_attr aAttr ) const
G_valueRef G_graph::getVAttr ( G_vertex v, const char *aAttrId )
                                const
```

Liefert eine Referenz auf den Wert des *Attributes* `aAttr` bzw. `aAttrId` des Knotens `v`.

**Rückgabewerte:** Eine Wertereferenz, über die der Attributwert manipuliert und ausgelesen werden kann.

**Fehlerfälle:** Es wird `G_ValueBottom` zurückgegeben und eine der folgenden Meldungen ausgegeben.

**grf007** Die Knoten `v` existiert nicht im Graph.

**typ002** Das Attribut `aAttr` ist nicht im Typsystem eingetragen.

**typ003** Das Attributschema zum Typ des Knoten `v` enthält kein Attribut `aAttr`.

**typ004** Der Typ des Knoten `v` existiert nicht im Typsystem.

**Aufwand:**  $O(1)$ , beim Aufruf mit `aAttr`

$O(|\text{AttributeSchema}(\text{type}(v))|)$ , beim Aufruf mit `aAttrId`, da das Attribut im Attributschema gesucht wird.

```
G_valueRef G_graph::getEAttr ( G_edge e, G_attr aAttr ) const
G_valueRef G_graph::getEAttr ( G_edge e, const char *aAttrId ) const
```

Liefert eine Referenz auf den Wert des *Attributes* `aAttr` bzw. `aAttrId` der Kante `e`.

**Rückgabewerte:** Eine Wertereferenz, über die der Attributwert manipuliert und ausgelesen werden kann.

**Fehlerfälle:** Es wird `G_ValueBottom` zurückgegeben und eine der folgenden Meldungen ausgegeben.

**grf007** Die Kante `e` existiert nicht im Graph.

**typ002** Das Attribut `aAttr` ist nicht im Typsystem eingetragen.

**typ003** Das Attributschema zum Typ der Kante `e` enthält kein Attribut `aAttr`.

**typ004** Der Typ der Kante `e` existiert nicht im Typsystem.

**Aufwand:**  $O(1)$ , beim Aufruf mit `aAttr`

$O(|\text{AttributSchema}(\text{type}(e))|)$ , beim Aufruf mit `aAttrId`, da das Attribut im Attributschema gesucht wird.

#### 4.1.6 Graphen temporär attributieren: `g_grftmp.c`

Für Algorithmen können die Graphen *temporär attributiert* werden. Bei mehreren, rekursiv benutzten Algorithmen kann man mehrere voneinander unabhängige *Schichten* von temporären Attributen anlegen. In jeder Schicht können einzelne Knoten bzw. Kanten mit temporären Attributen versehen werden.

```
int G_graph::createVTemp ( )
```

Erzeugt eine neue, leere Schicht temporärer Attribute für Knoten. Evtl. vorhandene Schichten werden verdeckt. Alle Knoten des Graphen tragen anschließend kein temporäres Attribut in der aktuellen Schicht.

**Rückgabewerte:** Die Anzahl der vorhandenen Attributierungsschichten für Knoten.

**Fehlerfälle:** Es wird 0 zurückgegeben und folgende Meldung ausgegeben.

**grf024** Die Schicht konnte wegen Speichermangel nicht angelegt werden.

**Aufwand:**  $O(1)$  und Aufwand der Freispeicherverwaltung.

```
bool G_graph::setPVTemp ( G_vertex v, G_tempAttribute *pTemp )
```

Setzt für Knoten  $v$  das temporäre Attribut  $pTemp$  in der aktuellen Schicht und liefert `true`. Wenn der Knoten in der aktuellen Schicht bereits ein temporäres Attribut hatte, wird das alte Attribut per Destruktor gelöscht.

Wird der Methode anstelle eines Attributs der `NULL`-Zeiger übergeben, wird nur das alte Attribut gelöscht. Der Knoten ist anschließend in der aktuellen Schicht nicht attribuiert.

**Parameter:**

**G\_tempAttribute \*pTemp :** Das temporäre Attribut muß auf dem dynamischen Speicher alloziert worden sein, da das Labor z.B. beim Löschen des Knotens den Destruktor aufrufen wird.

**Bemerkung:** Das Graphenlabor ist für die Deallokation verantwortlich — das Anwendungsprogramm darf nicht deallozieren.

**Fehlerfälle:** Es wird `false` zurückgegeben und eine der folgenden Meldungen ausgegeben.

**grf007** Der Knoten  $v$  existiert nicht.

**grf029** Im Graph existiert keine Schicht temporärer Knotenattribute.

**Aufwand:**  $O(1)$

```
G_tempAttribute * G_graph::getPVTemp ( G_vertex v ) const
```

Liefert einen Zeiger auf das temporäre Attribut des Knoten  $v$  in der aktuellen Schicht.

**Rückgabewerte:** `NULL`, wenn für den Knoten in der aktuellen Schicht kein Attribut eingetragen ist.

Ansonsten ein Zeiger auf eine Instanz der Oberklasse `G_tempAttribute`. Wenn eine eigene Attributklasse definiert wurde, dann muß mit einem `cast` der Zeiger angepaßt werden:

```
myAttr *pAttr = (myAttr *)g->getPVAttr (v);
```

**Bemerkung:** Das Attributobjekt darf nicht vom Anwendungsprogramm dealloziert werden.

**Fehlerfälle:** Es wird `NULL` zurückgegeben und eine der folgenden Meldungen ausgegeben.

**grf007** Der Knoten  $v$  existiert nicht im Graph.

**grf031** Im Graph existiert keine Schicht temporärer Knotenattribute.

**Aufwand:**  $O(1)$

```
void G_graph::deleteVTemp ( )
```

Löscht die aktuelle Schicht temporärer Attribute und alle in ihr befindlichen Attribute. Die darunter liegende Schicht wird wieder sichtbar.

**Fehlerfälle:** Es wird folgende Meldung ausgegeben.

**grf013** Im Graph existiert keine Schicht temporärer Knotenattribute.

**Aufwand:** Der Aufwand für das Löschen eines einzelnen Attributes ist konstant und wird für jedes Knotenattribut in der aktuellen Schicht benötigt.

```
int G_graph::getVTempLevel ( ) const
```

Liefert die Anzahl der Attributierungsschichten für Knoten. Falls keine Schicht vorhanden ist, liefert die Methode den Wert 0.

**Aufwand:**  $O(1)$ , inline

```
int G_graph::createETemp ( )
```

Erzeugt eine neue, leere Schicht temporärer Attribute für Kanten. Evtl. vorhandene Schichten werden verdeckt. Alle Kanten des Graphen tragen anschließend kein temporäres Attribut in der aktuellen Schicht.

**Rückgabewerte:** Die Anzahl der Attributierungsschichten für Kanten.

**Fehlerfälle:** Es wird 0 zurückgegeben und folgende Meldung ausgegeben.

**grf024** Die Schicht konnte wegen Speichermangel nicht angelegt werden.

**Aufwand:**  $O(1)$  und Aufwand der Freispeicherverwaltung.

```
bool G_graph::setPETemp ( G_edge e, G_tempAttribute *pTemp )
```

Setzt für Kante  $e$  das temporäre Attribut  $pTemp$  in der aktuellen Schicht und liefert `true`. Wenn die Kante in der aktuellen Schicht bereits ein temporäres Attribut hatte, wird das alte Attribut per Destruktor gelöscht.

Wird der Methode anstelle eines Attributs der `NULL`-Zeiger übergeben, wird nur das alte Attribut gelöscht. Die Kante ist anschließend in der aktuellen Schicht nicht attribuiert.

**Parameter:**

**G\_tempAttribute \*pTemp :** Das temporäre Attribut muß auf dem dynamischen Speicher alloziert worden sein, da das Labor z.B. beim Löschen der Kante den Destruktor aufrufen wird.

**Bemerkung:** Das Graphenlabor ist für die Deallokation verantwortlich — das Anwendungsprogramm darf nicht deallozieren.

**Fehlerfälle:** Es wird `false` zurückgegeben und eine der folgenden Meldungen ausgegeben.

**grf008** Die Kante  $e$  existiert nicht.

**grf029** Im Graph existiert keine Schicht temporärer Kantenattribute.

**Aufwand:**  $O(1)$

```
G_tempAttribute * G_graph::getPETemp ( G_edge e ) const
```

Liefert einen Zeiger auf das temporäre Attribut der Kante  $e$  in der aktuellen Schicht.

**Rückgabewerte:** NULL, wenn für die Kante in der aktuellen Schicht kein Attribut eingetragen ist.

Ansonsten ein Zeiger auf eine Instanz der Oberklasse `G_tempAttribute`. Wenn eine eigene Attributklasse definiert wurde, dann muß mit einem `cast` der Zeiger angepaßt werden:

```
myAttr *pAttr = (myAttr *)g->getPEAttr (e);
```

**Bemerkung:** Das Attributobjekt darf nicht vom Anwendungsprogramm dealloziert werden.

**Fehlerfälle:** Es wird NULL zurückgegeben und eine der folgenden Meldungen ausgegeben.

**grf008** Der Kante `e` existiert nicht im Graph.

**grf031** Im Graph existiert keine Schicht temporärer Kantenattribute.

**Aufwand:**  $O(1)$

```
void G_graph::deleteETemp ( )
```

Löscht die aktuelle Schicht temporärer Attribute und alle in ihr befindlichen Attribute. Die darunter liegende Schicht wird wieder sichtbar.

**Fehlerfälle:** Es wird folgende Meldung ausgegeben.

**grf013** Im Graph existiert keine Schicht temporärer Kantenattribute.

**Aufwand:** Der Aufwand für das Löschen eines einzelnen Attributes ist konstant und wird für jedes Kantenattribut in der aktuellen Schicht benötigt.

```
int G_graph::getETempLevel ( ) const
```

Liefert die Anzahl der Attributierungsschichten für Kanten. Falls keine Schicht vorhanden ist, liefert die Methode den Wert 0.

**Aufwand:**  $O(1)$ , inline

#### 4.1.7 Graphen ausgeben: `g_grfpert.c`

##### Ausgabe als Klartext

Die folgenden Funktionen dienen der übersichtlichen Ausgabe von Knoten, Kanten oder ganzen Graphen.

```
ostream & G_graph::printVertex ( ostream &oS, G_vertex v ) const
```

Schreibt alle verfügbaren Informationen zu Knoten `v` in den Ausgabestrom `oS` und liefert diesen zurück. Im einzelnen werden ausgegeben:

- die Identifikationsnummer
- die Identifikationsnummern der über out-Kanten erreichbaren Knoten sowie die der zugehörigen out-Kanten
- die Identifikationsnummern der über in-Kanten erreichbaren Knoten sowie die der zugehörigen in-Kanten
- der Knotentyp (als Identifikationsstring)
- die Attributwerte
- bei temporär attributierten Graphen die temporären Attribute aller Schichten

**Fehlerfälle:** Es wird folgende Meldung ausgegeben.

**grf007** Der Knoten  $v$  existiert nicht im Graph.

**Aufwand:**  $O(|\Lambda(v)|) + O(|\text{AttributSchema}(\text{type}(v))|)$   
 $+O(\#\text{Schichten temporärer Attribute})$

```
ostream & G_graph::printEdge ( ostream &oS, G_edge e ) const
```

Schreibt alle verfügbaren Informationen zu Kante  $e$  in den Ausgabestrom  $oS$  und liefert diesen zurück. Im einzelnen werden ausgegeben:

- die Identifikationsnummer
- die Identifikationsnummern von Start- und Endknoten
- der Kantentyp (als Identifikationsstring),
- die Attributwerte
- bei temporär attributierten Graphen die temporären Attribute aller Schichten

**Fehlerfälle:** Es wird folgende Meldung ausgegeben.

**grf008** Die Kante  $e$  existiert nicht im Graph.

**Aufwand:**  $O(|\text{AttributSchema}(\text{type}(e))|)$   
 $+O(\#\text{Schichten temporärer Attribute})$

```
ostream & G_graph::print ( ostream &oS ) const
```

Schreibt alle verfügbaren Informationen zum Graphen in den Ausgabestrom  $oS$  und liefert diesen zurück. Im einzelnen werden ausgegeben:

- aktuelle Knoten- und Kantenanzahl
- Größe der internen Tabellen (d.h. die maximale Anzahl von Knoten oder Kanten, die der Graph ohne Vergrößerung der Tabellen aufnehmen kann)
- Informationen zu allen Knoten (Aufrufe von `printVertex()` in der Reihenfolge  $Vseq$ )
- Informationen zu allen Kanten (Aufrufe von `printEdge()` in der Reihenfolge  $Eseq$ )

**Fehlerfälle:**

Fehler können nur auftreten, wenn die interne Datenstruktur beschädigt ist.

**Aufwand:** Es wird  $|Vseq|$  mal `printVertex()` und  $|Eseq|$  mal `printEdge()` aufgerufen.

```
ostream & operator<< ( ostream &oS, const G_graph &g )
```

Der Operator `<<` ist so überladen, daß mit ihm `print()` aufgerufen werden kann. Deshalb führt `oS<<g` zum Aufruf `g.print(oS)`.

### Ausgabe im HTML-Format

In den Ausgaben im HTML-Format werden Sprungmarken eingebaut, die mit normalen WWW-Browsern verfolgt werden können.

```
ostream & G_graph::htmlPrintVertex ( ostream &oS, G_vertex v ) const
```

Schreibt alle verfügbaren Informationen zu Knoten  $v$  in den Ausgabestrom  $oS$  und liefert

diesen zurück. Im einzelnen werden ausgegeben:

- die Identifikationsnummer
- die Identifikationsnummern der über out-Kanten erreichbaren Knoten sowie die der zugehörigen out-Kanten.
- die Identifikationsnummern der über in-Kanten erreichbaren Knoten sowie die der zugehörigen in-Kanten.
- der Knotentyp (als Identifikationsstring)
- die Attributwerte
- bei temporär attributierten Graphen die temporären Attribute aller Schichten

**Fehlerfälle:** Es wird folgende Meldung ausgegeben.

**grf007** Der Knoten  $v$  existiert nicht.

**Aufwand:**  $O(|\Lambda(v)|) + O(|\text{AttributSchema}(\text{type}(v))|) + O(\#\text{Schichten temporärer Attribute})$

```
ostream & G_graph::htmlPrintEdge ( ostream &oS, G_edge e ) const
```

Schreibt alle verfügbaren Informationen zu Kante  $e$  in den Ausgabestrom  $oS$  und liefert diesen zurück. Im einzelnen werden ausgegeben:

- die Identifikationsnummer
- die Identifikationsnummern von Start- und Endknoten
- der Kantentyp (als Identifikationsstring),
- die Attributwerte
- bei temporär attributierten Graphen die temporären Attribute aller Schichten

**Fehlerfälle:** Es wird folgende Meldung ausgegeben.

**grf008** Die Kante  $e$  existiert nicht.

**Aufwand:**  $O(|\text{AttributSchema}(\text{type}(e))|) + O(\#\text{Schichten temporärer Attribute})$

```
ostream & G_graph::htmlPrint ( ostream &oS ) const
void G_graph::htmlPrint ( const char *fileName ) const
```

Schreibt alle verfügbaren Informationen zum Graphen in den Ausgabestrom  $oS$  und liefert diesen zurück. Im einzelnen werden ausgegeben:

- aktuelle Knoten- und Kantenanzahl
- Größe der internen Tabellen (d.h. die maximale Anzahl von Knoten oder Kanten, die der Graph ohne Vergrößerung der Tabellen aufnehmen kann)
- Informationen zu allen Knoten (Aufrufe von `htmlPrintVertex()` in der Reihenfolge  $Vseq$ )
- Informationen zu allen Kanten (Aufrufe von `htmlPrintEdge()` in der Reihenfolge  $Eseq$ )

**Parameter:**

**const char \*fileName :** Eine neue Datei mit dem Namen `fileName.html` wird erzeugt und die Ausgabe darin abgespeichert.

**Aufwand:** Es wird  $|Vseq|$  mal `htmlPrintVertex()` und  $|Eseq|$  mal `htmlPrintEdge()` aufgerufen.

### 4.1.8 Graphen in Dateien ablegen und laden: `g_grfsto.c`

Mit den folgenden Methoden können Graphen in Dateien gespeichert und wieder geladen werden.

```
bool G_graph::store ( const char *fileName ) const
bool G_graph::store ( ostream &os ) const
```

Schreibt alle Informationen zum Graphen, mit Ausnahme der temporären Attributierung, in die Datei mit dem Namen `fileName` bzw. in den Stream `os`. Es werden gespeichert:

- das komplette mit dem Graph verbundene Typsystem
- die Folge aller Knoten  $Vseq$  samt inzidenter Kanten  $\Lambda(v), v \in V$
- die Folge aller Kanten  $Eseq$
- die Typisierung und Attributierung aller Knoten und aller Kanten

**Parameter:**

**const char \*fileName :** Falls bereits eine Datei dieses Namens existiert, wird sie überschrieben.

**ostream &os :** Der Stream `os` muß beschreibbar und im Binary-Mode (`G_IOS_BIN`) sein.

**Fehlerfälle:** Es wird `false` zurückgegeben und eine der folgenden Meldungen ausgegeben.

**grf015** Die Datei `fileName` konnte nicht angelegt werden.

**grf015** Der Stream `os` ist nicht beschreibbar.

**grf033** Es trat ein Fehler beim Speichern auf.

```
bool G_graph::load ( const char *fileName )
bool G_graph::load ( istream &is )
bool G_graph::load ( const char *fileName, const G_typeSystem &ts )
bool G_graph::load ( istream &is, const G_typeSystem &ts )
```

Liest einen mit `store()` geschriebenen Graphen wieder ein. Dazu wird die Graphinstanz zuerst initialisiert und danach mit den Informationen aus `fileName` (bzw. `is`) gefüllt.

**Parameter:**

**istream &is :** Der Stream `is` muß zum Lesen und im Binary-Mode (`G_IOS_BIN`) angelegt sein.

**const G\_typeSystem &ts :** Wenn angegeben, dann muß der Graph in `fileName` (bzw. `is`) zum Typsystem `ts` passen. Die Graphinstanz wird mit dem Typsystem `ts` verbunden.

**Bemerkung:** Vor dem Laden des Graphen wird überprüft, ob der Graph in `fileName` (bzw. `is`) zu dem Typsystem paßt, mit dem die Graphinstanz verbunden ist (bzw. zu `ts`). Dieses trifft zu, wenn das Typsystem in `fileName` (bzw. `is`) eine *Untermenge* des Typsystems der Graphinstanz ist. Es müssen alle Typen, Attribute und die *isA*-Relation genauso definiert sein. Wenn im neuen Typsystem Typen um Attribute erweitert wurden, dann werden diese Attributwerte beim Laden mit Nullwerten initialisiert.

**Fehlerfälle:** Es wird `false` zurückgegeben und eine der folgenden Meldungen ausgegeben.

**grf021** Beim Lesen der Typinformationen trat ein Fehler auf.

**grf025** Wegen Speichermangel konnten die internen Tabellen nicht in der erforderlichen Größe erzeugt werden.

**grf030** Vom Stream `is` kann nicht gelesen werden.

**grf030** Die Datei `fileName` konnte nicht geöffnet werden.

**grf035** Beim Lesen der Graphstruktur trat ein Syntaxfehler auf.

**grf040,grf041** Wegen Speichermangel konnten die Attributwerte nicht angelegt werden.

**grf050,grf051** Ein in der Graphdatei benutzter Typ existiert nicht in dem Typsystem, welches mit dem Graph verbunden ist.

```
bool G_graph::loadIncreaseTypeSystem ( const char *fileName,
                                       G_typeSystem &ts )
bool G_graph::loadIncreaseTypeSystem ( istream &is, G_typeSystem &ts
                                       )
```

Liest einen mit `store()` geschriebenen Graphen wieder ein. Dabei wird das Typsystem `ts` um die Typen erweitert, die in `fileName` bzw. `is` definiert sind. Die Graphinstanz wird danach mit dem Typsystem `ts` verbunden.

#### Parameter:

**istream &is** : Der Stream `is` muß zum Lesen und im Binary-Mode (`G_IOS_BIN`) angelegt sein.

**Bemerkung:** Vor dem Laden des Graphen wird das Typsystem `ts` um die Informationen aus der Datei erweitert. Die Graphinstanz ist danach mit dem Typsystem `ts` verbunden. Beim Erweitern des Typsystemes sind alle Operationen durchführbar, die auch über normale Laborfunktionen am Typsystem `ts` möglich sind. Z.B. können neue Typen oder Attribute definiert werden. Dagegen können Typen nur dann um Attribute erweitert werden, wenn sie nicht exportiert sind.

**Fehlerfälle:** Es wird `false` zurückgegeben und eine der folgenden Meldungen ausgegeben.

**grf021** Beim Lesen der Typinformationen trat ein Fehler auf.

**grf025** Wegen Speichermangel konnten die internen Tabellen nicht in der erforderlichen Größe erzeugt werden.

**grf030** Vom Stream `is` kann nicht gelesen werden.

**grf030** Die Datei `fileName` konnte nicht geöffnet werden.

**grf035** Beim Lesen der Graphstruktur trat ein Syntaxfehler auf.

**grf040,grf041** Wegen Speichermangel konnten die Attributwerte nicht angelegt werden.

**grf050,grf051** Ein in der Graphdatei benutzter Typ existiert nicht in dem Typsystem, welches mit dem Graph verbunden ist.

#### Intern benutzte Methoden zum Speichern und Laden

Die nachfolgenden Methoden werden intern zum Speichern und Laden der Graphstruktur benutzt. Das Typsystem wird nicht gespeichert. Deshalb muß in dem Programm sichergestellt werden, daß Graphstruktur und Typsystem zusammenpassen. Hiermit kann man z.B. viele kleine Graphen mit dem selben Typsystem in eine gemeinsame Datei speichern.

```
ofstream oFile("filename.garchive",
               ios::out|ios::trunc|G_IOS_BIN);
// typesystem == g1.getTypeSystem() == g2.getTypeSystem()
typesystem.storeOn (oFile);
g1.storeStructure (oFile);
g2.storeStructure (oFile);
oFile.close ();
```

Beim Laden von Graphen können die Typen und Attribute in der Datei andere Indexnummern haben als die Instanz des Typsystems im Arbeitsspeicher. Zum Übersetzen dieser Nummern dient die Datenstruktur `G_trans`. In sie wird in `restoreFrom()` bzw. `checkFrom()` die Übersetzungsfunktionen eingetragen. Bei `loadStructure()` werden diese Übersetzungsfunktionen benutzt.

```

ifstream iFile("filename.garchive", ios:in|G_IOS_BIN);
G_trans trans;
G_typeSystem typesystem;
typesystem.restoreFrom (iFile, trans);
G_graph g1(typesystem);
G_graph g2(typesystem);
// typesystem == g1.getTypeSystem() == g2.getTypeSystem()
g1.loadStructure (iFile, trans)
g2.loadStructure (iFile, trans);
iFile.close ();

```

```
bool G_graph::storeStructure ( ostream& os ) const
```

Schreibt die Graphstruktur auf den Stream `os`. Die Methode verhält sich wie `store()`, nur daß das Typsystem *nicht* gespeichert wird.

**Bemerkung:** Graphdateien, die mit dieser Methode erstellt wurden, können *nicht* mit den Methoden `load()` oder `loadIncreaseTypeSystem()` geladen werden.

Bei falscher Benutzung dieser Methode können inkonsistente Graph-Dateien entstehen.

```
bool G_graph::loadStructure ( istream& is, G_trans& tr )
```

Lädt die Graphstruktur aus dem Stream `is`. Die darin vorkommenden Attribut- und Typnummern werden mit `tr` übersetzt. Die Methode verhält sich wie `load()`, nur daß das Typsystem *nicht* überprüft wird.

**Bemerkung:** Graphdateien, die mit der Methode `store()` erstellt wurden, können *nicht* mit dieser Methode geladen werden.

Bei falscher Benutzung dieser Methode können inkonsistente Graph-Instanzen entstehen.

#### 4.1.9 Graphen traversieren: `g_grftra.c`

Für das Traversieren sind verschiedene Makros verfügbar:

**G\_forAllVertices** über alle Knoten des Graphen

**G\_forAllEdges** über alle Kanten des Graphen

**G\_forAllIncidentEdges** über alle zu einem Knoten inzidenten Kanten

**G\_forAllInEdges** über alle in einen Knoten einlaufenden Kanten

**G\_forAllOutEdges** über alle aus einem Knoten ausgehenden Kanten

Wenn diese Makros in dieser Form angewandt werden, dann werden typunabhängig alle Graphenelemente traversiert. Für das typabhängige Traversieren gibt es diese Makros mit den Suffixen `OfClass` und `OfType`.

Die Makros sind mit Iterationsmethoden realisiert, die ebenfalls benutzbar sind. Die Zusammenhänge zwischen den Makros und den Methoden sind in Tabelle 4.1, S. 74, dargestellt.

Sequenz	Startmethode	Iterationsmethode
$Vseq$	G_vertex firstVertex()	G_vertex nextVertex(G_vertex curV)
$Eseq$	G_edge firstEdge()	G_edge nextEdge(G_edge curE)
$\Lambda(v)$	G_edge first(G_vertex v)	G_edge next(G_edge curE)
$\Lambda^+(v)$	G_edge firstOut(G_vertex v)	G_edge nextOut(G_edge curE)
$\Lambda^-(v)$	G_edge firstIn(G_vertex v)	G_edge nextIn(G_edge curE)

Tabelle 4.1: Iterationsmethoden

### Typunabhängiges Traversieren

```
G_forAllVertices ( G_graph g, G_vertex v )
  { statements }
```

Führt für jeden Knoten des Graphen  $g$  die Anweisungen `statements` aus. Der aktuelle Knoten steht dabei jeweils in der Variablen  $v$ . Die Reihenfolge entspricht der Folge aller Knoten des Graphen  $Vseq$ .

**Beispiel:** Durch folgendes Beispiel wird `process()` für alle Knoten des Graphen aufgerufen:

```
G_forAllVertices (g, v)
{
  process (g, v);
}
```

**Bemerkung:** Im Gegensatz zu Methoden muß bei Makros der Graph explizit angegeben werden.

Innerhalb der Schleife darf die Folge aller Knoten  $Vseq$  nicht durch Erzeugen, Löschen oder Umordnen von Knoten verändert werden. Sie wird mit den Methoden `firstVertex()` und `nextVertex()` realisiert.

**Aufwand:** Jeder Knoten wird einmal bearbeitet:  $O(|Vseq|)$

```
G_forAllEdges ( G_graph g, G_edge e )
  { statements }
```

Führt für jede Kante des Graphen  $g$  die Anweisungen `statements` aus. Die aktuelle Kante steht dabei jeweils in der Variablen  $e$ . Die Reihenfolge entspricht der Folge aller Kanten des Graphen  $Eseq$ .

**Bemerkung:** Im Gegensatz zu Methoden muß bei Makros der Graph explizit angegeben werden.

Innerhalb der Schleife darf die Folge aller Kanten  $Eseq$  nicht durch Erzeugen, Löschen (auch indirekt durch Löschen eines Knotens) oder Umordnen von Kanten verändert werden. Die Schleife wird mit den Methoden `firstEdge()` und `nextEdge()` realisiert.

**Aufwand:** Jede Kante wird einmal bearbeitet:  $O(|Eseq|)$

```
G_forAllIncidentEdges ( G_graph g, G_vertex v, G_edge e )
  { statements }
```

Führt für jede mit dem Knoten  $v$  inzidente Kante des Graphen  $g$  die Anweisungen `statements` aus. Die aktuelle Kante steht dabei jeweils in der Variablen  $e$ . Die Reihenfolge, in der die Kanten bearbeitet werden, entspricht  $\Lambda(v)$ . Schlingen werden zweimal, nämlich für

beide Orientierungen bearbeitet.

**Bemerkung:** Im Gegensatz zu Methoden muß bei Makros der Graph explizit angegeben werden.

In der Schleife darf die Folge  $\Lambda(v)$  der zu  $v$  inzidenten Kanten nicht durch Hinzufügen, Löschen (auch indirekt durch Löschen eines Knotens) oder Umordnen von Kanten verändert werden. Die Schleife wird mit den Methoden `first()` und `next()` realisiert.

**Aufwand:** Jede zu  $v$  inzidente Kante wird einmal bearbeitet:  $O(|\Lambda(v)|)$

```
G_forAllInEdges ( G_graph g, G_vertex v, G_edge e )
  { statements }
```

Führt für jede in-Kante des Knotens  $v$  die Anweisungen `statements` aus. Die aktuelle Kante steht dabei jeweils in der Variablen `e`. Die Reihenfolge, in der die Kanten bearbeitet werden, entspricht  $\Lambda^-(v)$ .

**Bemerkung:** Im Gegensatz zu Methoden muß bei Makros der Graph explizit angegeben werden.

In der Schleife darf die Folge  $\Lambda(v)$  der zu  $v$  inzidenten Kanten nicht durch Hinzufügen, Löschen (auch indirekt durch Löschen eines Knotens) oder Umordnen von Kanten verändert werden. Die Schleife wird mit den Methoden `firstIn()` und `nextIn()` realisiert.

**Aufwand:** Es wird über alle inzidenten Kanten gesucht:  $O(|\Lambda(v)|)$

```
G_forAllOutEdges ( G_graph g, G_vertex v, G_edge e )
  { statements }
```

Führt für jede out-Kante des Knotens  $v$  die Anweisungen `statements` aus. Die aktuelle Kante steht dabei jeweils in der Variablen `e`. Die Reihenfolge, in der die Kanten bearbeitet werden, entspricht  $\Lambda^+(v)$ .

**Bemerkung:** Im Gegensatz zu Methoden muß bei Makros der Graph explizit angegeben werden.

In der Schleife darf die Folge  $\Lambda(v)$  der zu  $v$  inzidenten Kanten nicht durch Hinzufügen, Löschen (auch indirekt durch Löschen eines Knotens) oder Umordnen von Kanten verändert werden. Die Schleife wird mit den Methoden `firstOut()` und `nextOut()` realisiert.

**Aufwand:** Es wird über alle inzidenten Kanten gesucht:  $O(|\Lambda(v)|)$

```
G_vertex G_graph::firstVertex ( ) const
```

Liefert den ersten Knoten aus der Knotensequenz des Graphen.

**Rückgabewerte:** Wenn kein Knoten vorhanden ist, wird `G_VertexBottom` geliefert.

**Aufwand:**  $O(1)$ , inline

```
G_vertex G_graph::nextVertex ( G_vertex v ) const
```

Liefert den Knoten, der in der Knotensequenz des Graphen unmittelbar hinter  $v$  steht.

**Rückgabewerte:** Wenn es hinter  $v$  keine weiteren Knoten mehr gibt, wird `G_VertexBottom` zurückgegeben.

**Fehlerfälle:** Es wird `G_VertexBottom` zurückgegeben und folgende Meldung ausgegeben.

**grf007** Der Knoten  $v$  existiert nicht.

**Aufwand:**  $O(1)$ , inline

`G_edge G_graph::firstEdge ( ) const`

Liefert die erste Kante aus der Kantensequenz des Graphen.

**Rückgabewerte:** Wenn keine Kante vorhanden ist, wird `G_EdgeBottom` geliefert.

**Aufwand:**  $O(1)$ , inline

`G_edge G_graph::nextEdge ( G_edge e ) const`

Liefert die Kante, die in der Kantensequenz des Graphen unmittelbar hinter `e` steht.

**Rückgabewerte:** Wenn es hinter `e` keine weiteren Kanten mehr gibt, wird `G_EdgeBottom` zurückgegeben.

**Fehlerfälle:** Es wird `G_EdgeBottom` zurückgegeben und folgende Meldung ausgegeben.  
**grf008** Die Kante `e` existiert nicht.

**Aufwand:**  $O(1)$ , inline

`G_edge G_graph::first ( G_vertex v ) const`

Liefert die erste Kante aus der Kantensequenz des Knotens `v`.

**Rückgabewerte:** Wenn keine Kante vorhanden ist, wird `G_EdgeBottom` geliefert.

**Fehlerfälle:** Es wird `G_EdgeBottom` zurückgegeben und folgende Meldung ausgegeben.  
**grf007** Der Knoten `v` existiert nicht.

**Aufwand:**  $O(1)$ , inline

`G_edge G_graph::next ( G_edge e ) const`

Liefert die Kante, die in der Kantensequenz des Knotens `thisV(e)` unmittelbar hinter `e` steht.

**Rückgabewerte:** Wenn es hinter `e` keine weiteren Kanten mehr gibt, wird `G_EdgeBottom` zurückgegeben.

**Fehlerfälle:** Es wird `G_EdgeBottom` zurückgegeben und folgende Meldung ausgegeben.  
**grf008** Die Kante `e` existiert nicht.

**Aufwand:**  $O(1)$ , inline

`G_edge G_graph::firstIn ( G_vertex v ) const`

Liefert die erste in den Knoten `v` einlaufende Kante.

**Rückgabewerte:** Wenn keine Kante vorhanden ist, wird `G_EdgeBottom` geliefert.

**Fehlerfälle:** Es wird `G_EdgeBottom` zurückgegeben und folgende Meldung ausgegeben.  
**grf007** Der Knoten `v` existiert nicht.

**Aufwand:** Da in der Liste der inzidenten Kanten die erste eingehende Kante gesucht wird:  
 $O(|\Lambda(v)|)$

`G_edge G_graph::nextIn ( G_edge e ) const`

Liefert die nächste in den Knoten  $\omega(e)$  einlaufende Kante.

**Rückgabewerte:** Wenn es hinter  $e$  keine weiteren Kanten mehr gibt, wird `G_EdgeBottom` zurückgegeben.

**Fehlerfälle:** Es wird `G_EdgeBottom` zurückgegeben und folgende Meldung ausgegeben.  
**grf008** Die Kante  $e$  existiert nicht.

**Aufwand:** Da in der Liste der inzidenten Kanten die nächste eingehende Kante gesucht wird:  
 $O(|\Lambda(\omega(e))|)$

`G_edge G_graph::firstOut ( G_vertex v ) const`

Liefert die erste vom Knoten  $v$  ausgehende Kante.

**Rückgabewerte:** Wenn keine Kante vorhanden ist, wird `G_EdgeBottom` geliefert.

**Fehlerfälle:** Es wird `G_EdgeBottom` zurückgegeben und folgende Meldung ausgegeben.  
**grf007** Der Knoten  $v$  existiert nicht.

**Aufwand:** Da in der Liste der inzidenten Kanten die erste ausgehende Kante gesucht wird:  
 $O(\Lambda(v))$

`G_edge G_graph::nextOut ( G_edge e ) const`

Liefert die nächste vom Knoten  $\alpha(e)$  ausgehende Kante.

**Rückgabewerte:** Wenn es hinter  $e$  keine weiteren Kanten mehr gibt, wird `G_EdgeBottom` zurückgegeben.

**Fehlerfälle:** Es wird `G_EdgeBottom` zurückgegeben und folgende Meldung ausgegeben.  
**grf008** Die Kante  $e$  existiert nicht.

**Aufwand:** Da in der Liste der inzidenten Kanten die nächste ausgehende Kante gesucht wird:  
 $O(|\Lambda(\alpha(e))|)$

### Traversieren über Klassen

Beim Traversieren über eine Typklasse  $t$  werden alle Graphenelemente besucht, die vom Typen  $t$  oder einer seiner Untertypen sind.

`G_forAllVerticesOfClass ( G_graph g, G_type t, G_vertex v )`  
 { statements }

Führt für jeden Knoten des Graphen  $g$ , dessen Typ  $t$  oder ein Untertyp von  $t$  ist, die Anweisungen `statements` aus. Der aktuelle Knoten steht dabei jeweils in der Variablen  $v$ . Die Reihenfolge entspricht der Folge aller Knoten des Graphen  $V_{seq}$ .

**Beispiel:** Durch folgendes Beispiel wird `process()` für alle Knoten des Graphen aufgerufen:

```
G_forAllVerticesOfClass (g, t, v)
{
  process (g, v);
}
```

**Bemerkung:** Im Gegensatz zu Methoden muß bei Makros der Graph explizit angegeben werden.

Innerhalb der Schleife darf die Folge aller Knoten  $Vseq$  nicht durch Erzeugen, Löschen oder Umordnen von Knoten verändert werden. Sie wird mit `firstVertexOfClass()` und `nextVertexOfClass()` realisiert.

**Aufwand:** Jeder Knoten wird einmal bearbeitet:  $O(|Vseq|)$

```
G_forAllEdgesOfClass ( G_graph g, G_type t, G_edge e )
  { statements }
```

Führt für jede Kante des Graphen  $g$ , deren Typ  $t$  oder ein Untertyp von  $t$  ist, die Anweisungen `statements` aus. Die aktuelle Kante steht dabei jeweils in der Variablen  $e$ . Die Reihenfolge entspricht der Folge aller Kanten des Graphen  $Eseq$ .

**Bemerkung:** Im Gegensatz zu Methoden muß bei Makros der Graph explizit angegeben werden.

Innerhalb der Schleife darf die Folge aller Kanten  $Eseq$  nicht durch Erzeugen, Löschen (auch indirekt durch Löschen eines Knotens) oder Umordnen von Kanten verändert werden. Die Schleife wird mit den Methoden `firstEdgeOfClass()` und `nextEdgeOfClass()` realisiert.

**Aufwand:** Jede Kante wird einmal bearbeitet:  $O(|Eseq|)$

```
G_forAllIncidentEdgesOfClass ( G_graph g, G_type t, G_vertex v,
                               G_edge e )
  { statements }
```

Führt für jede mit dem Knoten  $v$  inzidente Kante des Graphen  $g$ , deren Typ  $t$  oder ein Untertyp von  $t$  ist, die Anweisungen `statements` aus. Die aktuelle Kante steht dabei jeweils in der Variablen  $e$ . Die Reihenfolge, in der die Kanten bearbeitet werden, entspricht  $\Lambda(v)$ . Schlingen werden zweimal, nämlich für beide Orientierungen bearbeitet.

**Bemerkung:** Im Gegensatz zu Methoden muß bei Makros der Graph explizit angegeben werden.

In der Schleife darf die Folge  $\Lambda(v)$  der zu  $v$  inzidenten Kanten nicht durch Hinzufügen, Löschen (auch indirekt durch Löschen eines Knotens) oder Umordnen von Kanten verändert werden. Die Schleife wird mit den Methoden `firstOfClass()` und `nextOfClass()` realisiert.

**Aufwand:** Jede zu  $v$  inzidente Kante wird einmal bearbeitet:  $O(|\Lambda(v)|)$

```
G_forAllInEdgesOfClass ( G_graph g, G_type t, G_vertex v, G_edge e )
  { statements }
```

Führt für jede in-Kante des Knotens  $v$ , deren Typ  $t$  oder ein Untertyp von  $t$  ist, die Anweisungen `statements` aus. Die aktuelle Kante steht dabei jeweils in der Variablen  $e$ . Die Reihenfolge, in der die Kanten bearbeitet werden, entspricht  $\Lambda^-(v)$ .

**Bemerkung:** Im Gegensatz zu Methoden muß bei Makros der Graph explizit angegeben werden.

In der Schleife darf die Folge  $\Lambda(v)$  der zu  $v$  inzidenten Kanten nicht durch Hinzufügen, Löschen (auch indirekt durch Löschen eines Knotens) oder Umordnen von Kanten verändert werden. Sie wird mit den Methoden `firstInOfClass()` und `nextInOfClass()` realisiert.

**Aufwand:** Es wird über alle inzidenten Kanten gesucht:  $O(|\Lambda(v)|)$

```
G_forAllOutEdgesOfClass ( G_graph g, G_type t, G_vertex v, G_edge e )
  { statements }
```

Führt für jede out-Kante des Knotens  $v$ , deren Typ  $t$  oder ein Untertyp von  $t$  ist, die Anweisungen `statements` aus. Die aktuelle Kante steht dabei jeweils in der Variablen  $e$ . Die Reihenfolge, in der die Kanten bearbeitet werden, entspricht  $\Lambda^+(v)$ .

**Bemerkung:** Im Gegensatz zu Methoden muß bei Makros der Graph explizit angegeben werden.

In der Schleife darf die Folge  $\Lambda(v)$  der zu  $v$  inzidenten Kanten nicht durch Hinzufügen, Löschen (auch indirekt durch Löschen eines Knotens) oder Umordnen von Kanten verändert werden. Sie wird mit den Methoden `firstOutOfClass()` und `nextOutOfClass()` realisiert.

**Aufwand:** Es wird über alle inzidenten Kanten gesucht:  $O(|\Lambda(v)|)$

```
G_vertex G_graph::firstVertexOfClass ( G_type t ) const
```

Liefert den ersten Knoten aus der Knotensequenz des Graphen, dessen Typ  $t$  oder ein Untertyp von  $t$  ist.

**Rückgabewerte:** Wenn kein passender Knoten vorhanden ist, wird `G_VertexBottom` geliefert.

**Fehlerfälle:** Es wird `G_VertexBottom` zurückgegeben und eine der folgenden Meldungen ausgegeben.

**typ004** Der Typ  $t$  ist nicht im Typsystem eingetragen.

**grf042** Der Typ  $t$  ist nicht exportiert.

**Aufwand:** Da in der Liste aller Knoten der erste Knoten der Klasse  $t$  gesucht wird:  $O(|Vseq|)$

```
G_vertex G_graph::nextVertexOfClass ( G_type t, G_vertex v ) const
```

Liefert den ersten Knoten des Typs  $t$  oder ein Untertyp von  $t$ , der in der Knotensequenz des Graphen hinter  $v$  steht.

**Rückgabewerte:** Wenn es nach  $v$  keinen passenden Knoten mehr gibt, wird `G_VertexBottom` zurückgegeben.

**Fehlerfälle:** Es wird `G_VertexBottom` zurückgegeben und eine der folgenden Meldungen ausgegeben.

**typ004** Der Typ  $t$  ist nicht im Typsystem eingetragen.

**grf007** Der Knoten  $v$  existiert nicht.

**grf042** Der Typ  $t$  ist nicht exportiert.

**Aufwand:** In der Liste aller Knoten wird der nächste Knoten der Klasse  $t$  gesucht:  $O(|Vseq|)$

```
G_edge G_graph::firstEdgeOfClass ( G_type t ) const
```

Liefert die erste Kante aus der Kantensequenz des Graphen, deren Typ  $t$  oder einem Untertypen von  $t$  entspricht.

**Rückgabewerte:** Wenn keine passende Kante vorhanden ist, wird `G_EdgeBottom` geliefert.

**Fehlerfälle:** Es wird `G_EdgeBottom` zurückgegeben und eine der folgenden Meldungen ausgegeben.

**typ004** Der Typ  $t$  ist nicht im Typsystem eingetragen.

**grf042** Der Typ  $t$  ist nicht exportiert.

**Aufwand:** Da in der Liste aller Kanten die erste Kante der Klasse  $t$  gesucht wird:  $O(|E_{seq}|)$

`G_edge G_graph::nextEdgeOfClass ( G_type t, G_edge e ) const`

Liefert die erste Kante, die in der Kantensequenz des Graphen hinter  $e$  steht und vom Typ  $t$  oder einem Untertypen von  $t$  ist.

**Rückgabewerte:** Wenn es nach  $e$  keine weitere passende Kante gibt, wird `G_EdgeBottom` zurückgegeben.

**Fehlerfälle:** Es wird `G_VertexBottom` zurückgegeben und eine der folgenden Meldungen ausgegeben.

**typ004** Der Typ  $t$  ist nicht im Typsystem eingetragen.

**grf008** Die Kante  $e$  existiert nicht.

**grf042** Der Typ  $t$  ist nicht exportiert.

**Aufwand:** Da in der Liste aller Kanten die nächste Kante der Klasse  $t$  gesucht wird:  $O(|E_{seq}|)$

`G_edge G_graph::firstOfClass ( G_type t, G_vertex v ) const`

Liefert die erste Kante aus der Kantensequenz des Knotens  $v$ , die vom Typ  $t$  oder einem Untertypen von  $t$  ist.

**Rückgabewerte:** Wenn keine passende Kante vorhanden ist, wird `G_EdgeBottom` geliefert.

**Fehlerfälle:** Es wird `G_EdgeBottom` zurückgegeben und eine der folgenden Meldungen ausgegeben.

**typ004** Der Typ  $t$  ist nicht im Typsystem eingetragen.

**grf007** Der Knoten  $v$  existiert nicht.

**grf042** Der Typ  $t$  ist nicht exportiert.

**Aufwand:** Da die Kantensequenz des Knotens  $v$  nach einer passenden Kante durchsucht wird:  $O(|\Lambda(v)|)$

`G_edge G_graph::nextOfClass ( G_type t, G_edge e ) const`

Liefert die erste Kante, die in der Kantensequenz des Knotens  $thisV(e)$  hinter  $e$  steht und vom Typ  $t$  oder einem Untertypen von  $t$  ist.

**Rückgabewerte:** Wenn es hinter  $e$  keine passende Kante mehr gibt, wird `G_EdgeBottom` zurückgegeben.

**Fehlerfälle:** Es wird `G_EdgeBottom` zurückgegeben und eine der folgenden Meldungen ausgegeben.

**typ004** Der Typ  $t$  ist nicht im Typsystem eingetragen.

**grf008** Die Kante  $e$  existiert nicht.

**grf042** Der Typ  $t$  ist nicht exportiert.

**Aufwand:** Da die Kantensequenz des Knotens  $v = this(e)$  nach einer passenden Kante durchsucht wird:  $O(|\Lambda(this(e))|)$

G\_edge **G\_graph::firstInOfClass** ( G\_type t, G\_vertex v ) const

Liefert die erste in den Knoten v einlaufende Kante, die vom Typen t oder einem Untertypen von t ist.

**Rückgabewerte:** Wenn keine passende Kante vorhanden ist, wird G\_EdgeBottom geliefert.

**Fehlerfälle:** Es wird G\_EdgeBottom zurückgegeben und eine der folgenden Meldungen ausgegeben.

**typ004** Der Typ t ist nicht im Typsystem eingetragen.

**grf007** Der Knoten v existiert nicht.

**grf042** Der Typ t ist nicht exportiert.

**Aufwand:** In der Liste der inzidenten Kanten wird eine passende Kante gesucht:  $O(|\Lambda(v)|)$

G\_edge **G\_graph::nextInOfClass** ( G\_type t, G\_edge e ) const

Liefert die nächste in den Knoten  $\omega(e)$  einlaufende Kante, die vom Typen t oder einem Untertypen von t ist.

**Rückgabewerte:** Wenn es hinter e keine passende Kante mehr gibt, wird G\_EdgeBottom zurückgegeben.

**Fehlerfälle:** Es wird G\_EdgeBottom zurückgegeben und eine der folgenden Meldungen ausgegeben.

**typ004** Der Typ t ist nicht im Typsystem eingetragen.

**grf008** Die Kante e existiert nicht.

**grf042** Der Typ t ist nicht exportiert.

**Aufwand:** Da in der Liste der inzidenten Kanten die nächste passende Kante gesucht wird:  $O(|\Lambda(\omega(e))|)$

G\_edge **G\_graph::firstOutOfClass** ( G\_type t, G\_vertex v ) const

Liefert die erste vom Knoten v ausgehende Kante, die vom Typen t oder einem Untertypen von t ist.

**Rückgabewerte:** Wenn keine passende Kante vorhanden ist, wird G\_EdgeBottom geliefert.

**Fehlerfälle:** Es wird G\_EdgeBottom zurückgegeben und eine der folgenden Meldungen ausgegeben.

**typ004** Der Typ t ist nicht im Typsystem eingetragen.

**grf007** Der Knoten v existiert nicht.

**grf042** Der Typ t ist nicht exportiert.

**Aufwand:** Da in der Liste der inzidenten Kanten die erste passende Kante gesucht wird:  $O(\Lambda(v))$

G\_edge **G\_graph::nextOutOfClass** ( G\_type t, G\_edge e ) const

Liefert die nächste vom Knoten  $\alpha(e)$  ausgehende Kante, die vom Typen t oder einem Untertypen von t ist.

**Rückgabewerte:** Wenn es hinter e keine passende Kante mehr gibt, wird G\_EdgeBottom zurückgegeben.

**Fehlerfälle:** Es wird G\_EdgeBottom zurückgegeben und eine der folgenden Meldungen aus-

gegeben.

**typ004** Der Typ  $t$  ist nicht im Typsystem eingetragen.

**grf008** Die Kante  $e$  existiert nicht.

**grf042** Der Typ  $t$  ist nicht exportiert.

**Aufwand:** Da in der Liste der inzidenten Kanten die nächste passende Kante gesucht wird:  
 $O(|\Lambda(\alpha(e))|)$

### Traversieren über Typen

Beim Traversieren über einen Typ  $t$  werden alle Graphenelemente besucht, die genau vom Typen  $t$  sind.

```
G_forAllVerticesOfType ( G_graph g, G_type t, G_vertex v )
  { statements }
```

Führt für jeden Knoten des Graphen  $g$ , dessen Typ  $t$  ist, die Anweisungen `statements` aus. Der aktuelle Knoten steht dabei jeweils in der Variablen  $v$ . Die Reihenfolge entspricht der Folge aller Knoten des Graphen  $Vseq$ .

**Beispiel:** Durch folgendes Beispiel wird `process()` für alle Knoten des Graphen aufgerufen:

```
G_forAllVerticesOfType (g, t, v)
{
  process (g, v);
}
```

**Bemerkung:** Im Gegensatz zu Methoden muß bei Makros der Graph explizit angegeben werden.

Innerhalb der Schleife darf die Folge aller Knoten  $Vseq$  nicht durch Erzeugen, Löschen oder Umordnen von Knoten verändert werden. Sie wird `firstVertexOfType()` und `nextVertexOfType()` realisiert.

**Aufwand:** Jeder Knoten wird einmal bearbeitet:  $O(|Vseq|)$

```
G_forAllEdgesOfType ( G_graph g, G_type t, G_edge e )
  { statements }
```

Führt für jede Kante des Graphen  $g$ , deren Typ  $t$  ist, die Anweisungen `statements` aus. Die aktuelle Kante steht dabei jeweils in der Variablen  $e$ . Die Reihenfolge entspricht der Folge aller Kanten des Graphen  $Eseq$ .

**Bemerkung:** Im Gegensatz zu Methoden muß bei Makros der Graph explizit angegeben werden.

Innerhalb der Schleife darf die Folge aller Kanten  $Eseq$  nicht durch Erzeugen, Löschen (auch indirekt durch Löschen eines Knotens) oder Umordnen von Kanten verändert werden. Die Schleife wird mit den Methoden `firstEdgeOfType()` und `nextEdgeOfType()` realisiert.

**Aufwand:** Jede Kante wird einmal bearbeitet:  $O(|Eseq|)$

```
G_forAllIncidentEdgesOfType ( G_graph g, G_type t, G_vertex v, G_edge
                               e )
  { statements }
```

Führt für jede mit dem Knoten  $v$  inzidente Kante des Graphen  $g$ , deren Typ  $t$  ist, die Anweisungen `statements` aus. Die aktuelle Kante steht dabei jeweils in der Variablen  $e$ . Die Reihenfolge, in der die Kanten bearbeitet werden, entspricht  $\Lambda(v)$ . Schlingen werden zweimal, nämlich für beide Orientierungen bearbeitet.

**Bemerkung:** Im Gegensatz zu Methoden muß bei Makros der Graph explizit angegeben werden.

In der Schleife darf die Folge  $\Lambda(v)$  der zu  $v$  inzidenten Kanten nicht durch Hinzufügen, Löschen (auch indirekt durch Löschen eines Knotens) oder Umordnen von Kanten verändert werden. Die Schleife wird mit den Methoden `firstOfType()` und `nextOfType()` realisiert.

**Aufwand:** Jede zu  $v$  inzidente Kante wird einmal bearbeitet:  $O(|\Lambda(v)|)$

```
G_forAllInEdgesOfType ( G_graph g, G_type t, G_vertex v, G_edge e )
  { statements }
```

Führt für jede in-Kante des Knotens  $v$ , deren Typ  $t$  ist, die Anweisungen `statements` aus. Die aktuelle Kante steht dabei jeweils in der Variablen  $e$ . Die Reihenfolge, in der die Kanten bearbeitet werden, entspricht  $\Lambda^-(v)$ .

**Bemerkung:** Im Gegensatz zu Methoden muß bei Makros der Graph explizit angegeben werden.

In der Schleife darf die Folge  $\Lambda(v)$  der zu  $v$  inzidenten Kanten nicht durch Hinzufügen, Löschen (auch indirekt durch Löschen eines Knotens) oder Umordnen von Kanten verändert werden. Sie wird mit den Methoden `firstInOfType()` und `nextInOfType()` realisiert.

**Aufwand:** Es wird über alle inzidenten Kanten gesucht:  $O(|\Lambda(v)|)$

```
G_forAllOutEdgesOfType ( G_graph g, G_type t, G_vertex v, G_edge e )
  { statements }
```

Führt für jede out-Kante des Knotens  $v$ , deren Typ  $t$  ist, die Anweisungen `statements` aus. Die aktuelle Kante steht dabei jeweils in der Variablen  $e$ . Die Reihenfolge, in der die Kanten bearbeitet werden, entspricht  $\Lambda^+(v)$ .

**Bemerkung:** Im Gegensatz zu Methoden muß bei Makros der Graph explizit angegeben werden.

In der Schleife darf die Folge  $\Lambda(v)$  der zu  $v$  inzidenten Kanten nicht durch Hinzufügen, Löschen (auch indirekt durch Löschen eines Knotens) oder Umordnen von Kanten verändert werden. Sie wird mit den Methoden `firstOutOfType()` und `nextOutOfType()` realisiert.

**Aufwand:** Es wird über alle inzidenten Kanten gesucht:  $O(|\Lambda(v)|)$

```
G_vertex G_graph::firstVertexOfType ( G_type t ) const
```

Liefert den ersten Knoten aus der Knotensequenz des Graphen, dessen Typ  $t$  ist.

**Rückgabewerte:** Wenn kein passender Knoten vorhanden ist, wird `G_VertexBottom` geliefert.

**Fehlerfälle:** Es wird `G_VertexBottom` zurückgegeben und eine der folgenden Meldungen ausgegeben.

**typ004** Der Typ  $t$  ist nicht im Typsystem eingetragen.

**grf042** Der Typ  $t$  ist nicht exportiert.

**Aufwand:** Da in der Liste aller Knoten der erste Knoten der Klasse  $t$  gesucht wird:  $O(|V_{seq}|)$

`G_vertex G_graph::nextVertexOfType ( G_type t, G_vertex v ) const`

Liefert den ersten Knoten des Typs  $t$ , der in der Knotensequenz des Graphen hinter  $v$  steht.

**Rückgabewerte:** Wenn es nach  $v$  keinen passenden Knoten mehr gibt, wird `G_VertexBottom` zurückgegeben.

**Fehlerfälle:** Es wird `G_VertexBottom` zurückgegeben und eine der folgenden Meldungen ausgegeben.

**typ004** Der Typ  $t$  ist nicht im Typsystem eingetragen.

**grf007** Der Knoten  $v$  existiert nicht.

**grf042** Der Typ  $t$  ist nicht exportiert.

**Aufwand:** In der Liste aller Knoten wird der nächste Knoten der Klasse  $t$  gesucht:  $O(|V_{seq}|)$

`G_edge G_graph::firstEdgeOfType ( G_type t ) const`

Liefert die erste Kante aus der Kantensequenz des Graphen, deren Typ  $t$  ist.

**Rückgabewerte:** Wenn keine passende Kante vorhanden ist, wird `G_EdgeBottom` geliefert.

**Fehlerfälle:** Es wird `G_EdgeBottom` zurückgegeben und eine der folgenden Meldungen ausgegeben.

**typ004** Der Typ  $t$  ist nicht im Typsystem eingetragen.

**grf042** Der Typ  $t$  ist nicht exportiert.

**Aufwand:** Da in der Liste aller Kanten die erste Kante der Klasse  $t$  gesucht wird:  $O(|E_{seq}|)$

`G_edge G_graph::nextEdgeOfType ( G_type t, G_edge e ) const`

Liefert die erste Kante, die in der Kantensequenz des Graphen hinter  $e$  steht und vom Typ  $t$  ist.

**Rückgabewerte:** Wenn es hinter  $e$  keine passende Kante mehr gibt, wird `G_EdgeBottom` zurückgegeben.

**Fehlerfälle:** Es wird `G_EdgeBottom` zurückgegeben und eine der folgenden Meldungen ausgegeben.

**typ004** Der Typ  $t$  ist nicht im Typsystem eingetragen.

**grf008** Die Kante  $e$  existiert nicht.

**grf042** Der Typ  $t$  ist nicht exportiert.

**Aufwand:** Da in der Liste aller Kanten die nächste Kante der Klasse  $t$  gesucht wird:  $O(|E_{seq}|)$

`G_edge G_graph::firstOfType ( G_type t, G_vertex v ) const`

Liefert die erste Kante aus der Kantensequenz des Knotens  $v$ , die vom Typ  $t$  ist.

**Rückgabewerte:** Wenn keine passende Kante vorhanden ist, wird `G_EdgeBottom` geliefert.

**Fehlerfälle:** Es wird `G_EdgeBottom` zurückgegeben und eine der folgenden Meldungen ausgegeben.

**typ004** Der Typ  $t$  ist nicht im Typsystem eingetragen.

**grf007** Der Knoten  $v$  existiert nicht.

**grf042** Der Typ  $t$  ist nicht exportiert.

**Aufwand:** Da die Kantensequenz des Knotens  $v$  nach einer passenden Kante durchsucht wird:  
 $O(|\Lambda(v)|)$

`G_edge G_graph::nextOfType ( G_type t, G_edge e ) const`

Liefert die erste Kante, die in der Kantensequenz des Knotens  $thisV(e)$  hinter  $e$  steht und vom Typ  $t$  ist.

**Rückgabewerte:** Wenn es hinter  $e$  keine passende Kante mehr gibt, wird `G_EdgeBottom` zurückgegeben.

**Fehlerfälle:** Es wird `G_EdgeBottom` zurückgegeben und eine der folgenden Meldungen ausgegeben.

**typ004** Der Typ  $t$  ist nicht im Typsystem eingetragen.

**grf008** Die Kante  $e$  existiert nicht.

**grf042** Der Typ  $t$  ist nicht exportiert.

**Aufwand:** Da die Kantensequenz des Knotens  $v = this(e)$  nach einer passenden Kante durchsucht wird:  $O(|\Lambda(this(e))|)$

`G_edge G_graph::firstInOfType ( G_type t, G_vertex v ) const`

Liefert die erste Kante in den Knoten  $v$  einlaufende Kante, die vom Typen  $t$  ist.

**Rückgabewerte:** Wenn keine passende Kante vorhanden ist, wird `G_EdgeBottom` geliefert.

**Fehlerfälle:** Es wird `G_EdgeBottom` zurückgegeben und eine der folgenden Meldungen ausgegeben.

**typ004** Der Typ  $t$  ist nicht im Typsystem eingetragen.

**grf007** Der Knoten  $v$  existiert nicht.

**grf042** Der Typ  $t$  ist nicht exportiert.

**Aufwand:** In der Liste der inzidenten Kanten wird nach der ersten passenden Kante gesucht:  
 $O(|\Lambda(v)|)$

`G_edge G_graph::nextInOfType ( G_type t, G_edge e ) const`

Liefert die nächste in den Knoten  $\omega(e)$  einlaufende Kante, die vom Typen  $t$  ist.

**Rückgabewerte:** Wenn es hinter  $e$  keine passende Kante mehr gibt, wird `G_EdgeBottom` zurückgegeben.

**Fehlerfälle:** Es wird `G_EdgeBottom` zurückgegeben und eine der folgenden Meldungen ausgegeben.

**typ004** Der Typ  $t$  ist nicht im Typsystem eingetragen.

**grf008** Die Kante  $e$  existiert nicht.

**grf042** Der Typ  $t$  ist nicht exportiert.

**Aufwand:** Da in der Liste der inzidenten Kanten die nächste passende Kante gesucht wird:  
 $O(|\Lambda(\omega(e))|)$

`G_edge G_graph::firstOutOfType ( G_type t, G_vertex v ) const`

Liefert die erste Kante vom Knoten  $v$  ausgehende Kante, die vom Typen  $t$  ist.

**Rückgabewerte:** Wenn keine passende Kante vorhanden ist, wird `G_EdgeBottom` geliefert.

**Fehlerfälle:** Es wird `G_EdgeBottom` zurückgegeben und eine der folgenden Meldungen ausgegeben.

**typ004** Der Typ  $t$  ist nicht im Typsystem eingetragen.

**grf007** Der Knoten  $v$  existiert nicht.

**grf042** Der Typ  $t$  ist nicht exportiert.

**Aufwand:** Da in der Liste der inzidenten Kanten die erste passende Kante gesucht wird:  $O(\Lambda(v))$

`G_edge G_graph::nextOutOfType ( G_type t, G_edge e ) const`

Liefert die nächste vom Knoten  $\alpha(e)$  ausgehende Kante, die vom Typen  $t$  ist.

**Rückgabewerte:** Wenn es nach  $e$  keine weitere passende Kante gibt, wird `G_EdgeBottom` zurückgegeben.

**Fehlerfälle:** Es wird `G_EdgeBottom` zurückgegeben und eine der folgenden Meldungen ausgegeben.

**typ004** Der Typ  $t$  ist nicht im Typsystem eingetragen.

**grf008** Die Kante  $e$  existiert nicht.

**grf042** Der Typ  $t$  ist nicht exportiert.

**Aufwand:** Da in der Liste der inzidenten Kanten die nächste passende Kante gesucht wird:  $O(|\Lambda(\alpha(e))|)$

#### 4.1.10 Knoten- oder Kantenanordnung erfragen und manipulieren: `g_grford.c`

Mit dem Graphenlabor werden *TGraphen* (siehe EBERT and FRANZKE, 1995) verarbeitet, die u.a. *geordnet* sind. In dieser Ordnung ist festgelegt,

- daß alle Knoten eine feste Reihenfolge (*Vseq*) haben,
- daß alle Kanten eine feste Reihenfolge (*Eseq*) haben und
- daß die Kanten, die zu einem Knoten  $v$  inzident sind, ebenfalls eine Reihenfolge ( $\Lambda(v)$ ) haben.

Mit den Traversierungsmethoden in Abschnitt 4.1.9, S. 73, können die Graphenelemente nach diesen Ordnungen traversiert werden. Mit folgenden Funktionen kann die Reihenfolge der Folgen *Vseq*, *Eseq* oder  $\Lambda(v)$ ,  $v \in V$  erfragt und geändert werden.

##### Globale Knotensequenz *Vseq*

Die Methoden `isVertexBefore()`, `putVertexBefore()` und `putVertexAfter()` beziehen sich auf die globale Sequenz aller Knoten (*Vseq*).

`bool G_graph::isVertexBefore ( G_vertex v, G_vertex w ) const`

Testet, ob der Knoten  $v$  in *Vseq* vor Knoten  $w$  vorkommt.

**Fehlerfälle:** Es wird `false` zurückgegeben und folgende Meldung ausgegeben.

**grf007** Einer der Knoten  $v$  und  $w$  existiert nicht.

**Aufwand:**  $O(|Vseq|)$

```
void G_graph::putVertexBefore ( G_vertex v, G_vertex w )
```

Verändert die Position des Knotens  $v$  in  $Vseq$  so, daß nach dem Aufruf der Knoten  $v$  unmittelbar vor Knoten  $w$  steht.

**Fehlerfälle:** Es wird eine der folgenden Meldungen ausgegeben.

**grf007** Einer der Knoten  $v$  und  $w$  existiert nicht.

**grf010** Mit  $v$  und  $w$  wird auf denselben Knoten referiert.

**Aufwand:**  $O(1)$

```
void G_graph::putVertexAfter ( G_vertex v, G_vertex w )
```

Verändert die Position des Knotens  $v$  in  $Vseq$  so, daß nach dem Aufruf der Knoten  $v$  unmittelbar hinter Knoten  $w$  steht.

**Fehlerfälle:** Es wird eine der folgenden Meldungen ausgegeben.

**grf007** Einer der Knoten  $v$  und  $w$  existiert nicht.

**grf010** Mit  $v$  und  $w$  wird auf denselben Knoten referiert.

**Aufwand:**  $O(1)$

### Globale Kantensequenz $Eseq$

Die Methoden `isEdgeBefore()`, `putEdgeBefore()` und `putEdgeAfter()` beziehen sich auf die globale Sequenz aller Kanten ( $Eseq$ ).

```
bool G_graph::isEdgeBefore ( G_edge e, G_edge f ) const
```

Testet, ob die Kante  $e$  in  $Eseq$  vor der Kante  $f$  vorkommt.

**Fehlerfälle:** Es wird `false` zurückgegeben und folgende Meldung ausgegeben.

**grf008** Eine der Kanten  $e$  und  $f$  existiert nicht.

**Aufwand:**  $O(|Eseq|)$

```
void G_graph::putEdgeBefore ( G_edge e, G_edge f )
```

Verändert die Position der Kante  $e$  in  $Eseq$  so, daß nach dem Aufruf die Kante  $e$  unmittelbar vor Kante  $f$  steht.

**Fehlerfälle:** Es wird eine der folgenden Meldungen ausgegeben.

**grf008** Eine der Kanten  $e$  und  $f$  existiert nicht.

**grf010** Mit  $e$  und  $f$  wird auf dieselbe Kante referiert.

**Aufwand:**  $O(1)$

```
void G_graph::putEdgeAfter ( G_edge e, G_edge f )
```

Verändert die Position der Kante  $e$  in  $Eseq$  so, daß nach dem Aufruf die Kante  $e$  unmittelbar hinter Kante  $f$  steht.

**Fehlerfälle:** Es wird eine der folgenden Meldungen ausgegeben.

**grf008** Eine der Kanten  $e$  und  $f$  existiert nicht.

**grf010** Mit  $e$  und  $f$  wird auf dieselbe Kante referiert.

**Aufwand:**  $O(1)$

**Inzidenzliste**  $\Lambda(v)$

Die Methoden `isBefore()`, `putBefore()` und `putAfter()` beziehen sich auf die lokale Sequenz aller Kanten, die zu einem Knoten inzident sind ( $\Lambda(v)$ ).

```
bool G_graph::isBefore ( G_edge e, G_edge f ) const
```

Testet, ob die Kante  $e$  in  $\Lambda(\text{this}(e))$  vor der Kante  $f$  vorkommt.

**Fehlerfälle:** Es wird `false` zurückgegeben und eine der folgenden Meldungen ausgegeben.

**grf008** Eine der Kanten  $e$  und  $f$  existiert nicht.

**grf009** Es kann nicht getestet werden, da  $\text{this}(e) \neq \text{this}(f)$ .

**Aufwand:**  $O(|\Lambda(\text{this}(e))|)$

```
void G_graph::putBefore ( G_edge e, G_edge f )
```

Verändert die Position der Kante  $e$  in  $\Lambda(\text{this}(e))$  so, daß nach dem Aufruf die Kante  $e$  unmittelbar vor Kante  $f$  steht.

**Fehlerfälle:** Es wird eine der folgenden Meldungen ausgegeben.

**grf008** Eine der Kanten  $e$  und  $f$  existiert nicht.

**grf009**  $\text{this}(e) \neq \text{this}(f)$

**grf010**  $e$  und  $f$  sind dieselbe orientierte Kante.

**Aufwand:**  $O(|\Lambda(\text{this}(e))|)$

```
void G_graph::putAfter ( G_edge e, G_edge f )
```

Verändert die Position der Kante  $e$  in  $\Lambda(\text{this}(e))$  so, daß nach dem Aufruf die Kante  $e$  unmittelbar hinter Kante  $f$  steht.

**Fehlerfälle:** Es wird eine der folgenden Meldungen ausgegeben.

**grf008** Eine der Kanten  $e$  und  $f$  existiert nicht.

**grf009**  $\text{this}(e) \neq \text{this}(f)$

**grf010**  $e$  und  $f$  sind dieselbe orientierte Kante.

**Aufwand:**  $O(|\Lambda(\text{this}(e))|)$

#### 4.1.11 Knoten, Kanten testen: `g_grftst.c`

```
bool G_graph::isVertex ( G_vertex v ) const
```

Testet, ob  $v$  auf einen vorhandenen Knoten referiert.

**Aufwand:**  $O(1)$ , inline

```
bool G_graph::isEdge ( G_edge e ) const
```

Testet, ob  $e$  auf eine vorhandene Kante referiert.

**Aufwand:**  $O(1)$ , inline

`bool G_graph::isValidV ( G_vertex v ) const`

Liefert `true`, wenn die Identifikationsnummer in  $v$  innerhalb des Bereiches liegt, den die internen Tabellen abdecken.

**Aufwand:**  $O(1)$ , inline

`bool G_graph::isValidE ( G_edge e ) const`

Liefert `true`, wenn die Identifikationsnummer in  $e$  innerhalb des Bereiches liegt, den die internen Tabellen abdecken.

**Aufwand:**  $O(1)$ , inline

`bool G_graph::isVertexBottom ( G_vertex v ) const`

Liefert `true`, wenn  $v$  gleich `G_VertexBottom` ist.

**Aufwand:**  $O(1)$ , inline

`bool G_graph::isEdgeBottom ( G_edge e ) const`

Liefert `true`, wenn  $e$  gleich `G_EdgeBottom` ist.

**Aufwand:**  $O(1)$ , inline

`bool G_graph::isNormal ( G_edge e ) const`

Liefert `true`, wenn die orientierte Kante  $e$  in normaler Form vorliegt. (D.h. Sicht vom Kantenanfang zur Kantenspitze)

**Aufwand:**  $O(1)$ , inline

`bool G_graph::isReverse ( G_edge e ) const`

Liefert `true`, wenn die orientierte Kante  $e$  **nicht** in normaler Form vorliegt. (D.h. Sicht von der Kantenspitze zum Kantenanfang)

**Aufwand:**  $O(1)$ , inline

`bool G_graph::areEqualVertices ( G_vertex v1, G_vertex v2 ) const`

Überprüft, ob die beiden Knotenvariablen  $v1$  und  $v2$  auf denselben Knoten im Graph referieren.

**Fehlerfälle:** Es wird `false` zurückgegeben und folgende Meldung ausgegeben.

**grf007** Einer der Knoten  $v1$  und  $v2$  existiert nicht.

**Aufwand:**  $O(1)$ , inline

`bool G_graph::areEqualEdges ( G_edge e1, G_edge e2 ) const`

Überprüft, ob die Kantenvariablen `e1` und `e2` auf dieselbe Kante im Graph referieren. Dabei wird die Orientierung der Kanten **nicht** unterschieden.

**Fehlerfälle:** Es wird `false` zurückgegeben und eine der folgenden Meldungen ausgegeben.  
**grf008** Eine der Kanten `e1` und `e2` existiert nicht.

**Aufwand:**  $O(1)$ , inline

#### 4.1.12 Kanteninformation erfragen: `g_grfaux.c`

Die folgenden Funktionen geben Strukturinformationen zu Kanten. Eine kurze Einführung zu *orientierten Kanten* findet sich in Abschnitt 1.1.3, S. 11.

`G_vertex G_graph::alpha ( G_edge e ) const`

Liefert den Anfangsknoten der gerichteten Kante `e`.

**Fehlerfälle:** Es wird `G_VertexBottom` zurückgegeben und folgende Meldung ausgegeben.  
**grf008** Die Kante `e` existiert nicht.

**Aufwand:**  $O(1)$ , inline

`G_vertex G_graph::omega ( G_edge e ) const`

Liefert den Endknoten der gerichteten Kante `e`.

**Fehlerfälle:** Es wird `G_VertexBottom` zurückgegeben und folgende Meldung ausgegeben.  
**grf008** Die Kante `e` existiert nicht.

**Aufwand:**  $O(1)$ , inline

`G_vertex G_graph::thisV ( G_edge e ) const`

Liefert den Knoten, von dem aus man die orientierte Kante `e` betrachtet.

**Fehlerfälle:** Es wird `G_VertexBottom` zurückgegeben und folgende Meldung ausgegeben.  
**grf008** Die Kante `e` existiert nicht.

**Aufwand:**  $O(1)$ , inline

`G_vertex G_graph::thatV ( G_edge e ) const`

Liefert den Knoten am anderen Ende der orientierten Kante `e`.

**Fehlerfälle:** Es wird `G_VertexBottom` zurückgegeben und folgende Meldung ausgegeben.  
**grf008** Die Kante `e` existiert nicht.

**Aufwand:**  $O(1)$ , inline

`G_edge G_graph::normal ( G_edge e ) const`

Liefert die normalisierte Darstellung der orientierten Kante `e`. D.h. in der Perspektive vom Anfangsknoten zum Endknoten.

**Fehlerfälle:** Es wird `G_EdgeBottom` zurückgegeben und folgende Meldung ausgegeben.

**grf008** Die Kante `e` existiert nicht.

**Aufwand:**  $O(1)$ , inline

```
G_edge G_graph::reverse ( G_edge e ) const
```

Dreht die Orientierung der orientierten Kante `e` um.

**Fehlerfälle:** Es wird `G_EdgeBottom` zurückgegeben und folgende Meldung ausgegeben.

**grf008** Die Kante `e` existiert nicht.

**Aufwand:**  $O(1)$ , inline

#### 4.1.13 Grad von Knoten erfragen: `g_grfdeg.c`

Mit folgenden Funktionen kann der Grad von Knoten abgefragt werden.

```
unsigned G_graph::degree ( G_vertex v ) const
```

Liefert die Anzahl der mit dem Knoten `v` inzidenten Kanten. Schlingen werden dabei doppelt gezählt.

**Fehlerfälle:** Es wird `0` zurückgegeben und folgende Meldung ausgegeben.

**grf007** Der Knoten `v` existiert nicht.

**Aufwand:**  $O(|\Lambda(v)|)$

```
unsigned G_graph::inDegree ( G_vertex v ) const
```

Liefert die Anzahl der in-Kanten an Knoten `v`.

**Fehlerfälle:** Es wird `0` zurückgegeben und folgende Meldung ausgegeben.

**grf007** Der Knoten `v` existiert nicht.

**Aufwand:**  $O(|\Lambda(v)|)$

```
unsigned G_graph::outDegree ( G_vertex v ) const
```

Liefert die Anzahl der out-Kanten an Knoten `v`.

**Fehlerfälle:** Es wird `0` zurückgegeben und folgende Meldung ausgegeben.

**grf007** Der Knoten `v` existiert nicht.

**Aufwand:**  $O(|\Lambda(v)|)$

```
unsigned G_graph::degreeOfClass ( G_vertex v, G_type t ) const
```

Liefert die Anzahl der mit dem Knoten `v` inzidenten Kanten, die der Klasse `t` angehören. Schlingen werden dabei doppelt gezählt.

**Fehlerfälle:** Es wird `0` zurückgegeben und eine der folgenden Meldungen ausgegeben.

**typ004** Der Typ `t` existiert nicht im Typsystem.

**grf007** Der Knoten `v` existiert nicht.

**Aufwand:**  $O(|\Lambda(v)|)$

`unsigned G_graph::inDegreeOfClass ( G_vertex v, G_type t ) const`

Liefert die Anzahl der in-Kanten an Knoten  $v$  die der Klasse  $t$  angehören.

**Fehlerfälle:** Es wird 0 zurückgegeben und eine der folgenden Meldungen ausgegeben.

**typ004** Der Typ  $t$  existiert nicht im Typsystem.

**grf007** Der Knoten  $v$  existiert nicht.

**Aufwand:**  $O(|\Lambda(v)|)$

`unsigned G_graph::outDegreeOfClass ( G_vertex v, G_type t ) const`

Liefert die Anzahl der out-Kanten an Knoten  $v$  die der Klasse  $t$  angehören.

**Fehlerfälle:** Es wird 0 zurückgegeben und eine der folgenden Meldungen ausgegeben.

**typ004** Der Typ  $t$  existiert nicht im Typsystem.

**grf007** Der Knoten  $v$  existiert nicht.

**Aufwand:**  $O(|\Lambda(v)|)$

`unsigned G_graph::degreeOfType ( G_vertex v, G_type t ) const`

Liefert die Anzahl der mit dem Knoten  $v$  inzidenten Kanten, die vom Typ  $t$  sind. Schlingen werden dabei doppelt gezählt.

**Fehlerfälle:** Es wird 0 zurückgegeben und eine der folgenden Meldungen ausgegeben.

**typ004** Der Typ  $t$  existiert nicht im Typsystem.

**grf007** Der Knoten  $v$  existiert nicht.

**Aufwand:**  $O(|\Lambda(v)|)$

`unsigned G_graph::inDegreeOfType ( G_vertex v, G_type t ) const`

Liefert die Anzahl der in-Kanten an Knoten  $v$  die vom Typ  $t$  sind.

**Fehlerfälle:** Es wird 0 zurückgegeben und eine der folgenden Meldungen ausgegeben.

**typ004** Der Typ  $t$  existiert nicht im Typsystem.

**grf007** Der Knoten  $v$  existiert nicht.

**Aufwand:**  $O(|\Lambda(v)|)$

`unsigned G_graph::outDegreeOfType ( G_vertex v, G_type t ) const`

Liefert die Anzahl der out-Kanten an Knoten  $v$  die vom Typ  $t$  sind.

**Fehlerfälle:** Es wird 0 zurückgegeben und eine der folgenden Meldungen ausgegeben.

**typ004** Der Typ  $t$  existiert nicht im Typsystem.

**grf007** Der Knoten  $v$  existiert nicht.

**Aufwand:**  $O(|\Lambda(v)|)$

#### 4.1.14 Globale Graphdaten erfragen: `g_grfmsc.c`

Allgemeine Informationen über Graphen können mit den folgenden Methoden abgefragt werden.

`unsigned G_graph::vertexCount ( ) const`

Liefert die Anzahl der Knoten im Graph.

**Aufwand:**  $O(1)$ , inline

`unsigned G_graph::edgeCount ( ) const`

Liefert die Anzahl der Kanten im Graph.

**Aufwand:**  $O(1)$ , inline

`unsigned G_graph::vertexMax ( ) const`

Liefert die Größe der internen Knotentabellen.

**Aufwand:**  $O(1)$ , inline

`unsigned G_graph::edgeMax ( ) const`

Liefert die Größe der internen Kantentabellen.

**Aufwand:**  $O(1)$ , inline

`G_edge G_graph::edgeFromTo ( G_vertex alpha, G_vertex omega ) const`

Liefert die erste, normalisierte Kante in der Folge der zu alpha inzidenten Kanten, die zum Knoten omega führt.

**Rückgabewerte:** Existiert keine passende Kante, wird `G_EdgeBottom` geliefert.

**Fehlerfälle:** Es wird `G_EdgeBottom` zurückgegeben und folgende Meldung ausgegeben.  
**grf007** Einer der Knoten alpha und omega existiert nicht.

**Aufwand:**  $O(|\Lambda(\alpha)|)$

`G_edge G_graph::edgeBetween ( G_vertex v, G_vertex w ) const`

Liefert die erste der zu v inzidenten Kanten zwischen den Knoten v und w. Die Kante e ist so orientiert, daß  $this(e) = v$  gilt.

**Rückgabewerte:** Existiert keine passende Kante, wird `G_EdgeBottom` geliefert.

**Fehlerfälle:** Es wird `G_EdgeBottom` zurückgegeben und folgende Meldung ausgegeben.  
**grf007** Einer der Knoten alpha und omega existiert nicht.

**Aufwand:**  $O(|\Lambda(v)|)$

`G_edge G_graph::edgeFromToOfType ( G_type type, G_vertex alpha, G_vertex omega ) const`

Liefert die erste, normalisierte Kante in der Folge der zu `alpha` inzidenten Kanten, die zum Knoten `omega` führt und vom Typ `type` ist.

**Rückgabewerte:** Existiert keine passende Kante, wird `G_EdgeBottom` geliefert.

**Fehlerfälle:** Es wird `G_EdgeBottom` zurückgegeben und eine der folgenden Meldungen ausgegeben.

**typ004** Der Typ `type` ist nicht im Typsystem eingetragen.

**grf007** Einer der Knoten `alpha` und `omega` existiert nicht.

**Aufwand:**  $O(|\Lambda(\alpha)|)$

`G_edge G_graph::edgeBetweenOfType ( G_type type, G_vertex v, G_vertex w ) const`

Liefert die erste der zu `v` inzidenten Kanten, zwischen den Knoten `v` und `w` liegt und vom Typ `type` ist. Die Kante `e` ist so orientiert, daß  $this(e) = v$  gilt.

**Rückgabewerte:** Existiert keine passende Kante, wird `G_EdgeBottom` geliefert.

**Fehlerfälle:** Es wird `G_EdgeBottom` zurückgegeben und eine der folgenden Meldungen ausgegeben.

**typ004** Der Typ `type` ist nicht im Typsystem eingetragen.

**grf007** Einer der Knoten `alpha` und `omega` existiert nicht.

**Aufwand:**  $O(|\Lambda(v)|)$

`G_edge G_graph::edgeFromToOfClass ( G_type type, G_vertex alpha, G_vertex omega ) const`

Liefert die erste, normalisierte Kante in der Folge der zu `alpha` inzidenten Kanten, die zum Knoten `omega` führt und vom Typ `type` oder einer seiner Untertypen ist.

**Rückgabewerte:** Existiert keine passende Kante, wird `G_EdgeBottom` geliefert.

**Fehlerfälle:** Es wird `G_EdgeBottom` zurückgegeben und eine der folgenden Meldungen ausgegeben.

**typ004** Der Typ `type` ist nicht im Typsystem eingetragen.

**grf007** Einer der Knoten `alpha` und `omega` existiert nicht.

**Aufwand:**  $O(|\Lambda(\alpha)|)$

`G_edge G_graph::edgeBetweenOfClass ( G_type type, G_vertex v, G_vertex w ) const`

Liefert die erste der zu `v` inzidenten Kanten, zwischen den Knoten `v` und `w` liegt und vom Typ `type` oder einer seiner Untertypen ist. Die Kante `e` ist so orientiert, daß  $this(e) = v$  gilt.

**Rückgabewerte:** Existiert keine passende Kante, wird `G_EdgeBottom` geliefert.

**Fehlerfälle:** Es wird `G_EdgeBottom` zurückgegeben und eine der folgenden Meldungen ausgegeben.

**typ004** Der Typ `type` ist nicht im Typsystem eingetragen.

**grf007** Einer der Knoten `alpha` und `omega` existiert nicht.

**Aufwand:**  $O(|\Lambda(v)|)$

`G_vertex G_graph::getV ( unsigned vNo ) const`

Erzeugt eine Knotenvariable mit der Identifikationsnummer `vNo`. Dabei wird nicht überprüft, ob ein solcher Knoten im Graph existiert.

**Aufwand:**  $O(1)$ , inline

`G_edge G_graph::getE ( int eNo ) const`

Erzeugt eine Kantenvariable mit der Identifikationsnummer `eNo`. Die Kante ist out-orientiert, wenn `eNo > 0` und in-orientiert, wenn `eNo < 0`. Dabei wird nicht überprüft, ob eine solche Kante im Graph existiert.

**Aufwand:**  $O(1)$ , inline

`unsigned G_graph::getVNo ( G_vertex v ) const`

Liefert die Identifikationsnummer des Knotens `v`.

**Aufwand:**  $O(1)$ , inline

`int G_graph::getENo ( G_edge e ) const`

Liefert die Identifikationsnummer der Kante `e`.

**Aufwand:**  $O(1)$ , inline

#### 4.1.15 UNDO-Schnittstelle

Oft will man Änderungen an Graphen protokollieren und später eventuell rückgängig machen. Hierzu wird im Graphenlabor mit der Klasse `G_undoBuffer` (siehe Abschnitt 4.5, S. 120) ein Undo-Puffer realisiert.

`void G_graph::setPUndoBuffer ( G_undoBuffer *pUndoBuf )`

Verbindet den Graphen mit einem Undo-Puffer. Durch die Übergabe eines `NULL`-Zeigers kann die automatische Protokollierung abgeschaltet werden.

Falls der Graph bereits mit einem Undo-Puffer verbunden war, werden alle im alten Puffer protokollierten Änderungen mit `G_undoBuffer::commit()` bestätigt.

**Aufwand:**  $O(1)$ , inline, wenn der Graph mit keinem Puffer verbunden war. Ansonsten muß der alte Puffer abgearbeitet werden.

`G_undoBuffer * G_graph::getPUndoBuffer ( ) const`

Liefert einen Zeiger auf den Undo-Puffer, mit dem der Graph verbunden ist.

**Rückgabewerte:** `NULL`, falls der Graph mit keinem Undo-Puffer verbunden ist.

**Aufwand:**  $O(1)$ , inline

## 4.2 Typsysteme

Die Klasse `G_typeSystem` verwaltet die von Graphen verwendeten Typen. Jedem Graph ist genau eine Instanz dieser Klasse zugeordnet. Ein *Typsystem* beinhaltet

1. eine Menge von Typen (repräsentiert durch `G_type`-Instanzen, siehe Abschnitt 4.2.3, S. 100),
2. eine Menge von Attributen (repräsentiert durch `G_attr`-Instanzen, siehe Abschnitt 4.2.2, S. 98),
3. eine Menge von Attributschemata, die jedem Typen eine Menge von Attributen zuordnen (siehe Abschnitt 4.2.4, S. 102),
4. und die Subtyp-Relation unter diesen Typen (siehe Abschnitt 4.2.5, S. 103).

Typsysteme werden unabhängig von Graphen in Dateien gespeichert. Jedes Typsystem enthält einen Nulltyp (`G_TypeNull()`) mit der Bezeichnung "TypeNull", dem ein leeres Attributschema zugeordnet ist. Dieser Typ ist (automatisch) Obertyp aller anderen Typen.

### 4.2.1 Verwaltung der Typsysteme: `g_ttypsys.c`

```
G_typeSystem::G_typeSystem ( )
```

```
G_typeSystem::G_typeSystem ( unsigned numType )
```

```
G_typeSystem::G_typeSystem ( unsigned numType, unsigned numAttr )
```

Legt ein neues Typsystem an. Es enthält nur den Nulltyp (Abrufbar mit `G_TypeNull()`). Anfangs wird Speicherplatz für `numType` Typen und `numAttr` Attribute reserviert. Falls dieser Speicherplatz später nicht ausreicht, wird er automatisch vergrößert.

**Fehlerfälle:** Es wird folgende Meldung ausgegeben.

**mem301** Der für das Typsystem nötige Arbeitsspeicher konnte nicht reserviert werden.

```
void G_typeSystem::reInitialize ( )
```

Löscht alle im Typsystem eingetragenen Attribute und Typen. Danach enthält das Typsystem nur den Nulltyp.

**Aufwand:** Da einzelne Typen lokale Daten haben können, müssen alle Typen überprüft werden:  
 $O(\#Typen)$

```
G_typeSystem::~~G_typeSystem ( )
```

Löscht ein Typsystem mit allen darin eingetragenen Typen und Attributen.

**Aufwand:** Da einzelne Typen lokale Daten haben können, müssen alle Typen überprüft werden:  
 $O(\#Typen)$

### Speichern und Laden von Typsystemen: `g_ttypsto.c`

```
bool G_typeSystem::store ( const char *fileName ) const
```

```
bool G_typeSystem::store ( ostream & os ) const
```

Speichert ein Typsystem in einer Datei mit dem Namen `fileName` oder auf dem Stream `os`.

**Parameter:**

**const char \*fileName** : Es wird eine neue Datei mit dem Namen `fileName` erzeugt.

**ostream & os** : Der Stream `os` muß zum Schreiben und im Binary-Mode angelegt sein (siehe Präprozessormakro `G_IOS_BIN`).

**Fehlerfälle:** Es wird `false` zurückgegeben und eine der folgenden Meldungen ausgegeben.

**typ015** Die Datei `fileName` konnte nicht angelegt werden.

**typ015** Der Stream `os` ist nicht beschreibbar.

**typ016** Es trat ein Fehler beim Schreiben auf.

**dom001** Durch einen Programmfehler enthält ein Attribut einen Verweis auf einen ungültigen Wertebereich.

```
bool G::typeSystem::load ( const char *fileName )
```

```
bool G::typeSystem::load ( ostream & is )
```

Löscht das Typsystem und initialisiert es mit den Attributen und Typen aus der Datei `fileName` (dem Stream `is`). Dabei wird das Wertebereichssystem um die in den Attributen verwendeten Wertebereiche erweitert.

**Parameter:**

**istream & is** : Der Stream `is` muß zum Lesen und im Binary-Mode angelegt sein (siehe Präprozessormakro `G_IOS_BIN`).

**Fehlerfälle:** Es wird `false` zurückgegeben und eine der folgenden Meldungen ausgegeben.

**typ017** Die Datei `fileName` kann nicht geöffnet werden.

**typ017** Vom Stream `is` kann nicht gelesen werden.

**typ018** Es trat ein Fehler beim Lesen auf.

**typ030** Beim Lesen der Daten wurde ein Syntaxfehler festgestellt.

```
bool G::typeSystem::increase ( const char *fileName )
```

```
bool G::typeSystem::increase ( ostream & is )
```

Erweitert das Typsystem um die Attribute und Typen, die in der Datei `fileName` (dem Stream `is`) definiert sind. Dabei wird das Wertebereichssystem um die in den Attributen verwendeten Wertebereiche erweitert.

**Parameter:**

**istream & is** : Der Stream `is` muß zum Lesen und im Binary-Mode angelegt sein (siehe Präprozessormakro `G_IOS_BIN`).

**Fehlerfälle:** Es wird `false` zurückgegeben und eine der folgenden Meldungen ausgegeben.

**typ017** Die Datei `fileName` kann nicht geöffnet werden.

**typ017** Vom Stream `is` kann nicht gelesen werden.

**typ018** Es trat ein Fehler beim Lesen auf.

**typ030** Beim Lesen der Daten wurde ein Syntaxfehler festgestellt.

### Interne Methoden zum Speichern und Laden: `g_typsto.c`

Diese Methoden werden normalerweise intern von anderen Labormethoden benutzt. Bei besonderen Anwendungen können sie auch direkt aufgerufen werden (siehe auch Abschnitt 4.1.8, S. 72). Bei falscher Benutzung dieser Methoden entstehen inkonsistente Zustände im Labor.

```
bool G_typeSystem::storeOn ( ostream& os ) const
```

Schreibt die im Typsystem benutzten Wertebereiche und das Typsystem in den Stream `os`.

```
bool G_typeSystem::restoreFrom ( istream& is, G_trans& tr )
```

Erweitert das Typsystem um die Attribute und Typen, die im Stream `is` beschrieben sind. Dabei wird in `tr` eine Funktion erzeugt, welche die Attribut- und Typnummern im Stream auf die internen Nummern im Typsystem abbildet.

**Rückgabewerte:** `true`, falls das Typsystem ohne Fehler erweitert werden konnte.

```
bool G_typeSystem::checkFrom ( istream& is, G_trans& tr )
```

Überprüft, ob im Stream `is` Attribute oder Typen enthalten sind, welche nicht zum Typsystem passen. Dabei wird in `tr` eine Funktion erzeugt, welche die Attribut- und Typnummern im Stream auf die internen Nummern im Typsystem abbildet.

**Rückgabewerte:** `true`, falls keine Konsistenzprobleme auftraten.

### 4.2.2 Verwaltung der Attributmenge: `g_typsys.c`

Ein Attribut ist ein Tupel aus Bezeichner und Wertebereich. Bevor ein solches Attribut einem Typen zugeordnet werden kann, muß es im Typsystem definiert werden. Es können mehrere Attribute mit dem selben Bezeichner aber unterschiedlichen Wertebereichen definiert sein. Erst bei der Zuordnung zu Typen mit `addAttr()` wird die Eindeutigkeit der Bezeichner im Attributierungsschema verlangt.

```
G_attr G_typeSystem::newAttr ( const char *aAttrId, G_domain aDom )
```

Trägt das Attribut ( $aAttrId \mapsto aDom$ ) in die Attributmenge des Typsystems ein.

**Rückgabewerte:** Eine Variable vom Typ `G_attr`, über die das Attribut im Typsystem adressiert werden kann.

Wenn es in der Attributmenge schon ein Tupel ( $aAttrId \mapsto aDom$ ) gibt – wenn das Attribut schon früher einmal eingetragen wurde – wird kein neues Attribut eingetragen, sondern ein Verweis auf das alte Attribut geliefert und eine Meldung (`typ011`) ausgegeben.

**Fehlerfälle:** Es wird `G_AttrBottom` zurückgegeben und folgende Meldung ausgegeben.  
**typ012** Der Wertebereich `aDom` ist nicht definiert.

**Aufwand:** Es muß überprüft werden, ob dieses Attribut schon in der Attributmenge enthalten ist:  $O(\#Attribute)$

```
bool G_typeSystem::knowsAttr ( const char *aAttrId, G_domain aDom )
                             const
```

Überprüft, ob das Attribut ( $aAttrId \mapsto aDom$ ) in der Attributmenge eingetragen ist.

**Aufwand:**  $O(\#Attribute)$ , da die ganze Attributmenge durchsucht werden muß.

```
G_attr G_typeSystem::getAttr ( const char *aAttrId, G_domain aDom )
                               const
```

Sucht in der Attributmenge nach einem Attribut ( $aAttrId \mapsto aDom$ ).

**Rückgabewerte:** Die entsprechende Attributvariable, bzw. `G_AttrBottom`, wenn das Attribut nicht in der Attributmenge existiert.

**Aufwand:**  $O(\#Attribute)$ , da die ganze Attributmenge durchsucht werden muß.

Informationen zu den eingetragenen Attributen können mit den nächsten beiden Methoden abgefragt werden.

```
G_domain G_typeSystem::getDomain ( G_attr aAttr ) const
```

Liefert den Wertebereich des Attributes `aAttr`.

**Fehlerfälle:** Es wird `G_DomainBottom` zurückgegeben und folgende Meldung ausgegeben.  
**typ002** Das Attribut `aAttr` ist nicht im Typsystem eingetragen.

**Aufwand:**  $O(1)$

```
G_id G_typeSystem::getAttrId ( G_attr aAttr ) const
```

Liefert den Bezeichner des Attributes `aAttr`.

**Fehlerfälle:** Es wird `G_IdBottom` zurückgegeben und folgende Meldung ausgegeben.  
**typ002** Das Attribut `aAttr` ist nicht im Typsystem eingetragen.

**Aufwand:**  $O(1)$

Man kann Schleifen über alle in der Attributmenge eingetragenen Attribute (mit bestimmtem Bezeichner) aufbauen. Diese Schleifen sehen wie folgt aus:

```
G_typeSystem ts;
....
G_attr a;
for (a = ts.getFirstAttr(); ts.OK (a); a = ts.getNextAttr (a))
    handleAttribute (ts, a);
```

```
G_attr G_typeSystem::getFirstAttr ( ) const
```

Liefert das erste in die Attributmenge eingetragene Attribut.

**Rückgabewerte:** `G_AttrBottom`, wenn keine Attribute definiert sind.

**Aufwand:**  $O(1)$

```
G_attr G_typeSystem::getNextAttr ( G_attr aAttr ) const
```

Liefert das nächste in der Attributmenge eingetragene Attribut.

**Rückgabewerte:** `G_AttrBottom`, wenn es keine weiteren Attribute gibt.

**Aufwand:**  $O(1)$

`G_attr G_typeSystem::getFirstAttrById ( const char *aAttrId ) const`  
Liefert das erste in die Attributmenge eingetragene Attribut mit dem Bezeichner `aAttrId`.

**Rückgabewerte:** `G_AttrBottom`, wenn es kein Attribut mit dem Bezeichner `aAttrId` gibt.

**Aufwand:**  $O(\#Attribute)$

`G_attr G_typeSystem::getNextAttrById ( G_attr aAttr ) const`

Liefert das nächste in der Attributmenge eingetragene Attribut mit demselben Bezeichner wie `aAttr`.

**Rückgabewerte:** `G_AttrBottom`, wenn es kein weiteres Attribut gibt.

**Aufwand:**  $O(\#Attribute)$

`bool G_typeSystem::OK ( G_attr aAttr ) const`

Testet, ob es sich bei `aAttr` um ein Attribut handelt, welches in der Attributmenge eingetragen ist.

**Aufwand:**  $O(1)$

Ein indizierter Zugriff ist mit `getNthAttr()` möglich.

`unsigned G_typeSystem::getAttrCount ( ) const`

Liefert die Anzahl der in der Attributmenge eingetragenen Attribute.

**Rückgabewerte:** Die Anzahl der Attribute — 0, wenn die Attributmenge leer ist.

**Aufwand:**  $O(1)$

`G_attr G_typeSystem::getNthAttr ( unsigned n ) const`

Liefert das `n`'te Attribut der Attributmenge.

**Parameter:**

**unsigned n :** Es wird ab 1 gezählt.

**Fehlerfälle:** Es wird `G_AttrBottom` zurückgegeben und folgende Meldung ausgegeben.

**typ013** Der Index `n` liegt außerhalb der erlaubten Grenzen.

**Aufwand:**  $O(1)$

### 4.2.3 Verwaltung der Typen: `g_tsys.c`

Typen werden durch einen im Typsystem eindeutigen Typbezeichner identifiziert. Dazu muß ein neuer Typ zuerst im Typsystem registriert werden.

`G_type G_typeSystem::newType ( const char *aTypeId )`

Trägt einen neuen Typen mit dem Bezeichner `aTypeId` im Typsystem ein.

**Rückgabewerte:** Eine Variable vom Typ `G_type`. Mit dieser Variable kann später dieser Typ im Typsystem adressiert werden.

Wenn es schon einen Typen mit dem Bezeichner `aTypeId` gibt, dann wird eine Referenz auf den Typen geliefert und eine Meldung (`typ010`) ausgegeben.

**Fehlerfälle:** Es wird `G_TypeBottom` zurückgegeben und folgende Meldung ausgegeben.

**typ009** Es konnte kein Arbeitsspeicher für die internen Tabellen angelegt werden.

**Aufwand:**  $O(\#Typen)$ , da überprüft werden muß, ob es den Typen schon gibt.

```
bool G_typeSystem::knowsType ( const char *aTypeId ) const
```

Testet, ob in der Typmenge ein Typ mit dem Bezeichner `aTypeId` enthalten ist.

**Aufwand:**  $O(\#Typen)$

```
G_type G_typeSystem::getType ( const char *aTypeId ) const
```

Liefert die Typvariable des Typen mit dem Bezeichner `aTypeId`.

**Rückgabewerte:** Eine Variable, über die der Typ im Typsystem adressiert werden kann.

**Aufwand:**  $O(\#Typen)$ , da über alle Typen gesucht wird.

```
G_id G_typeSystem::getTypeId ( G_type aType ) const
```

Liefert den Typbezeichner zu einer Typvariablen.

**Fehlerfälle:** Es wird `G_IdNull` zurückgegeben und folgende Meldung ausgegeben.

**typ004** Der Typ `aType` ist nicht im Typsystem eingetragen.

**Aufwand:**  $O(1)$

```
unsigned G_typeSystem::getTypeCount ( ) const
```

Ermittelt die Anzahl, der in der Typmenge eingetragenen Typen.

**Aufwand:**  $O(1)$

```
G_type G_typeSystem::getNthType ( unsigned n ) const
```

Liefert eine Typvariable des `n`'ten eingetragenen Typs.

**Parameter:**

**unsigned n :** Es wird ab 1 gezählt.

**Fehlerfälle:** Es wird `G_TypeBottom` zurückgegeben und folgende Meldung ausgegeben.

**typ013** Der Index `n` liegt außerhalb der erlaubten Grenzen.

**Aufwand:**  $O(1)$

```
bool G_typeSystem::OK ( G_type aType ) const
```

Testet, ob die Variable `aType` eine Typvariable des Typsystems ist. Diese Methode kann als Laufbedingung in Schleifen benutzt werden.

**Aufwand:**  $O(1)$

Damit ein Typ für das Erzeugen von Graphenelementen benutzt werden kann, muß sichergestellt sein, daß sich danach seine Struktur, insbesondere sein Attributschema, nicht mehr ändert. Da er auch alle Attributschemata der Obertypen erbt, dürfen sich auch ihre Strukturen nicht mehr ändern. Hierfür muß ein Typ *exportiert* werden.

```
void G_typeSystem::exportType ( G_type aType )
```

Markiert den Typ `aType` und alle eingetragene Obertypen von `aType` als *exportierte Typen*. D.h. ihre Struktur kann nicht mehr geändert werden und man kann den Typ für die Erzeugung von Graphenelementen benutzen.

**Fehlerfälle:** Es wird eine der folgenden Meldungen ausgegeben.

**typ004** Der Typ `aType` ist nicht im Typsystem eingetragen.

**typ005** Der Typ `aType` wurde schon früher exportiert.

**Aufwand:** Es werden weitere Informationen über die Attributschemata der Typen berechnet, die bei der Verwendung im Graph benötigt werden. Da alle Obertypen, die noch nicht exportiert sind, jetzt exportiert werden, werden diese Informationen für all jene Typen berechnet.

```
bool G_typeSystem::isExported ( G_type aType ) const
```

Testet, ob der Typ `aType` schon exportiert wurde.

**Fehlerfälle:** Es wird `false` zurückgegeben und folgende Meldung ausgegeben.

**typ004** Der Typ `aType` ist nicht im Typsystem eingetragen.

**Aufwand:**  $O(1)$

#### 4.2.4 Verwaltung der Attributierungsschemata

Jedem Typ ist ein *Attributierungsschema* zugeordnet. In dieses Schema können beliebig viele Attribute mit unterschiedlichen Attributbezeichnern eingetragen werden.

```
bool G_typeSystem::addAttr ( G_type aType, G_attr aAttr )
```

Erweitert die Attributschemata der Klasse `aType` um das Attribut `aAttr`.

**Fehlerfälle:** Es wird `false` zurückgegeben und eine der folgenden Meldungen ausgegeben.

**typ002** Das Attribut `aAttr` ist nicht im Typsystem eingetragen.

**typ004** Der Typ `aType` ist nicht im Typsystem eingetragen.

**typ005** Der Typ `aType` ist schon exportiert.

**typ007** In der Klasse `aType` gibt es schon ein Attribut mit demselben Bezeichner aber anderem Wertebereich.

**typ008** Gibt den Typen aus, bei dem ein Konflikt zwischen den eingetragenen Attributen und dem neuen Attribut auftritt.

**Aufwand:** Es muß zuerst für alle Typen der Klasse `aType`, d.h. alle Untertypen von `aType`, überprüft werden, ob sich ihre Attribute mit dem Attribut `aAttr` vertragen. Danach muß das Attribut `aAttr` bei allen Typen der Klasse `aType` eingetragen werden.

```
bool G_typeSystem::hasAttr ( G_type aType, G_attr aAttr ) const
```

Überprüft, ob das Attribut `aAttr` im Attributschema des Typen `aType` eingetragen ist. Da im Attributschema auch alle geerbten Typen enthalten sind, werden auch Attribute akzeptiert, die von einem Obertypen stammen.

**Fehlerfälle:** Es wird `false` zurückgegeben und eine der folgenden Meldungen ausgegeben.

**typ002** Das Attribut `aAttr` ist nicht im Typsystem eingetragen.

**typ004** Der Typ `aType` ist nicht im Typsystem eingetragen.

**Aufwand:**  $O(|\text{AttributSchema}(aType)|)$ , da das ganze Attributschema des Typen `aType` durchsucht wird.

```
unsigned G_typeSystem::getAttrCount ( G_type aType ) const
```

Ermittelt die Anzahl der Attribute im Attributschema des Typen `aType`.

**Fehlerfälle:** Es wird 0 zurückgegeben und folgende Meldung ausgegeben.

**typ004** Der Typ `aType` ist nicht im Typsystem eingetragen.

**Aufwand:**  $O(1)$

```
G_attr G_typeSystem::getNthAttr ( G_type aType, unsigned n ) const
```

Liefert das `n`'te Attribut des Typen `aType`.

**Parameter:**

**unsigned n :** Es wird ab 1 gezählt.

**Fehlerfälle:** Es wird `G_AttrBottom` zurückgegeben und eine der folgenden Meldungen ausgegeben.

**typ004** Der Typ `aType` ist nicht im Typsystem eingetragen.

**typ005** Der Wert `n` liegt außerhalb der erlaubten Grenzen.

**Aufwand:**  $O(1)$

```
G_attr G_typeSystem::getAttr ( G_type aType, const char *aAttrId )
                               const
```

Sucht in dem Attributschema des Typen `aType` nach einem Attribut mit dem Bezeichner `aAttrId`

**Rückgabewerte:** Eine Attributvariable, über die das Attribut im Typsystem adressiert wird oder `G_AttrBottom`, wenn es kein Attribut mit dem Bezeichner `aAttrId` gibt.

**Fehlerfälle:** Es wird `G_AttrBottom` zurückgegeben und folgende Meldung ausgegeben.

**typ004** Der Typ `aType` ist nicht im Typsystem eingetragen.

**Aufwand:**  $O(|\text{AttributSchema}(aType)|)$ , da das Attributschema des Typen `aType` durchsucht wird.

#### 4.2.5 Verwaltung der Subtyp-Relation: `g_typsyst.c`

```
bool G_typeSystem::setIsA ( G_type subType, G_type superType )
```

Trägt im Typsystem ein, daß `subType` ein Untertyp des Typen `superType` ist.

**Fehlerfälle:** Es wird `false` zurückgegeben und eine der folgenden Meldungen ausgegeben.

**typ004** Der Untertyp `subType` oder Der Obertyp `superType` ist nicht im Typsystem eingetragen.

**typ005** Der Untertyp `subType` ist schon exportiert.

**typ007** Im Attributschema des Obertyps `superType` gibt es ein Attribut, welches sich nicht mit einem Attribut eines Typs der Klasse `subType` verträgt.

**typ008** Gibt den Typen aus, bei dem ein Konflikt zwischen den Attributen auftritt.

**Bemerkung:** Die Methode bildet selbständig den transitiven Abschluß der Relation. Nach den Aufrufen von `setIsA(typA, typB)` und `setIsA(typB, typC)` steht also auch `typA` und `typC` in der Subtyp-Relation.

**Aufwand:** Es muß zuerst für alle Attribute des OberTyps `superType` überprüft werden, ob sie sich mit den Attributen aller Typen in der Klasse `subType` vertragen. Danach müssen alle Attribute des OberTyps `superType` bei allen Untertypen von `subType` eingetragen werden und die transitive Untertyp-Relation erweitert werden.

```
bool G_typeSystem::isA ( G_type subType, G_type superType ) const
```

Testet, ob der Typ `subType` ein Untertyp von `superType` ist oder ob `subType` derselbe Typ wie `superType` ist.

**Fehlerfälle:** Es wird `false` zurückgegeben und folgende Meldung ausgegeben.

**typ004** Der Untertyp `subType` oder Der Obertyp `superType` ist nicht im Typsystem eingetragen.

**Aufwand:**  $O(1)$

### Aufzählen der Ober- oder Untertypen

```
G_type G_typeSystem::getFirstSubType ( G_type aType ) const
```

```
G_type G_typeSystem::getNextSubType ( G_type aType, G_type lastType  
                                     ) const
```

Traversieren alle (echten) Untertypen des Typen `aType`. Der Typ `aType` selbst wird nicht geliefert.

**Fehlerfälle:** Es wird `G_TypeBottom` zurückgegeben und folgende Meldung ausgegeben.

**typ004** Einer der Typen `aType` und `lastType` ist nicht im Typsystem eingetragen.

**Aufwand:**  $O(\#Typen)$

```
G_type G_typeSystem::getFirstSuperType ( G_type aType ) const
```

```
G_type G_typeSystem::getNextSuperType ( G_type aType, G_type  
                                       lastType ) const
```

Traversieren alle (echten) Obertypen des Typen `aType`. Der Typ `aType` selbst wird nicht geliefert.

**Fehlerfälle:** Es wird `G_TypeBottom` zurückgegeben und folgende Meldung ausgegeben.

**typ004** Einer der Typen `aType` und `lastType` ist nicht im Typsystem eingetragen.

**Aufwand:**  $O(\#\text{Typen})$

### 4.3 Wertebereiche: `g_domain.c`

Das ganze Wertebereichssystem wird global für das ganze Programm verwaltet. Es ist nicht möglich, lokal in Typsystemen oder Graphen eigene Wertebereiche zu definieren.

#### 4.3.1 Grundsätzliche Funktionalität

Ein Wertebereich wird durch eine Instanz der Klasse `G_domain` repräsentiert.

**Konstanten** Es gibt folgende Konstanten vom Typ `G_domain`:

**`G_DomainBottom`:** Bezeichnet eine Variable vom Typ `G_domain`, die einen ungültigen Wert enthält. Sie wird von einigen Methoden als Rückgabewert bei einem Fehler benutzt.

**`G_domain::G_domain ( )`**

Erzeugt eine neue Variable, die einen Wertebereich aufnehmen kann.

**Aufwand:**  $O(1)$ , inline.

**`G_domain::G_domain ( const G_domain &aDom )`**

Erzeugt eine Kopie von `aDom`.

**Aufwand:**  $O(1)$ , inline.

**`bool G_domain::operator== ( const G_domain &aDom ) const`**

Testet, ob zwei Wertebereichsvariablen auf denselben Wertebereich referieren.

**Aufwand:**  $O(1)$ , inline.

**`bool G_domain::operator!= ( const G_domain &aDom ) const`**

Testet, ob zwei Wertebereichsvariablen auf verschiedene Wertebereiche referieren.

**Aufwand:**  $O(1)$ , inline.

**`G_domain & G_domain::operator= ( const G_domain &aDom )`**

Weist der Instanz den Wertebereich `aDom` zu.

**Aufwand:**  $O(1)$ , inline.

**`bool G_domain::OK ( ) const`**

Testet, ob mit einer Wertebereichsvariablen ein gültiger Wertebereich referiert wird.

**Aufwand:**  $O(1)$ , inline.

**Intern benutzte Methoden** Eher für die Organisation der Laufzeitumgebung sind die folgenden Methoden gedacht. Sie können aber auch von Applikationen benutzt werden.

```
size_t G_domain::getValueSize ( ) const
```

Liefert die Speichergröße, die ein Wert dieses Wertebereiches beansprucht.

**Rückgabewerte:** Die Speichergröße des Wertes in Bytes.

**Fehlerfälle:** Es wird 0 zurückgegeben und folgende Meldung ausgegeben.

**dom001** Die Variable verweist auf keinen Wertebereich.

**Aufwand:**  $O(1)$ , inline.

```
int G_domain::getAlignment ( ) const
```

Liefert den Alignmentwert, an dem alle Werte dieses Wertebereich ausgerichtet sein müssen.

**Fehlerfälle:** Es wird 1 zurückgegeben und folgende Meldung ausgegeben.

**dom001** Die Instanz verweist auf keinen Wertebereich.

**Aufwand:**  $O(1)$ , inline.

**Stream-Ausgaben** sind beim Testen von Programmen nützlich.

```
ostream & G_domain::printDomain ( ostream &os ) const
ostream & operator<< ( ostream &os, const G_domain &aDom )
```

Gibt die Wertebereichsinformationen in einer lesbaren Form auf os aus.

**Fehlerfälle:** Es wird folgende Meldung ausgegeben.

**dom001** Die Instanz (bzw. aDom) verweist auf keinen Wertebereich.

**Aufwand:** Bei strukturierten Wertebereichen müssen alle Komponenten rekursiv durchlaufen werden.

### 4.3.2 Basiswertebereiche

In der Bibliothek stehen von Anfang an drei Basiswertebereiche zur Verfügung. Diese sind mit den folgenden Variablen abrufbar.

```
static G_domain G_domain::BOOL
static G_domain G_domain::INT
static G_domain G_domain::DOUBLE
static G_domain G_domain::STRING
```

Instanzen können auf diese Basiswertebereiche hin überprüft werden.

```
bool G_domain::isBasic ( ) const
bool G_domain::isBOOL ( ) const
bool G_domain::isINT ( ) const
bool G_domain::isDOUBLE ( ) const
bool G_domain::isSTRING ( ) const
```

Testen, ob es sich um einen Basiswertebereich handelt.

**Fehlerfälle:** Es wird 0 zurückgegeben und folgende Meldung ausgegeben.

**dom001** Die Instanz verweist auf keinen Wertebereich.

**Aufwand:**  $O(1)$ , inline

### 4.3.3 Listenwertebereiche

In eine Variable des Wertebereiches *LISTE* kann eine Sequenz von Elementen aus einem gegebenen Wertebereich eingetragen werden.

```
static G_domain G_domain::newList ( G_domain aDom )
```

Erzeugt einen neuen Listenwertebereich mit aDom als Grundwertebereich.

**Rückgabewerte:** Wenn es schon einen Listenwertebereich über aDom gibt, dann wird ein Verweis auf diese Liste zurückgegeben.

**Fehlerfälle:** Es wird `G_DomainBottom` zurückgegeben und folgende Meldung ausgegeben.

**dom002** Der Wertebereich aDom existiert nicht.

**Aufwand:**  $O(1)$

```
bool G_domain::isList ( ) const
```

Testet, ob es ein Listen-Wertebereich ist.

**Fehlerfälle:** Es wird `false` zurückgegeben und folgende Meldung ausgegeben.

**dom001** Die Instanz verweist auf keinen Wertebereich.

**Aufwand:**  $O(1)$

```
G_domain G_domain::getBase ( ) const
```

Liefert den Grundwertebereich, auf dem die Liste definiert ist.

**Fehlerfälle:** Es wird `G_DomainBottom` zurückgegeben und folgende Meldung ausgegeben.

**dom101** Der Wertebereich stellt keine Liste dar.

**Aufwand:**  $O(1)$

```
size_t G_domain::getSlotSize ( ) const
```

Liefert die Größe, die ein Element in dieser Liste beansprucht.

**Fehlerfälle:** Es wird 0 zurückgegeben und folgende Meldung ausgegeben.

**dom101** Der Wertebereich stellt keine Liste dar.

**Aufwand:**  $O(1)$

### 4.3.4 Wertebereichssequenzen

Für die Tupel- und Record-Wertebereiche muß beim Erzeugen eines neuen Wertebereiches eine Sequenz (bei Tupeln) bzw. eine Menge von Bezeichner-Wertebereichspaaren (bei Records) angegeben

werden. Um diese Sequenzen zu erstellen, gibt es die Klasse `G_domainSequence`.

**`G_domainSequence::G_domainSequence ( )`**

Erzeugt eine neuen Instanz, in die Wertebereiche eingetragen werden können.

**Aufwand:**  $O(1)$

`bool G_domainSequence::addDomain ( G_domain aDom )`

`bool G_domainSequence::addDomain ( G_domain aDom, const char *aId )`

Erweitert die Sequenz um den Wertebereich `aDom` bzw. das Bezeichner-Wertebereich-Paar (`aId`  $\mapsto$  `aDom`).

**Fehlerfälle:** Es wird `false` zurückgegeben und eine der folgenden Meldungen ausgegeben.

**dom003** Es konnte kein Arbeitsspeicher für die internen Tabellen angefordert werden.

**dom201** Der Selektorbezeichner `aId` ist schon in der Sequenz eingetragen.

**Aufwand:**  $O(1)$ , bzw.  $O(|\text{Sequenz}|)$ , wenn die interne Datenstruktur vergrößert wird.

`G_domain G_domainSequence::getNthDomain ( int idx ) const`

Liefert den Wertebereich der `idx`'ten Komponente. Die Reihenfolge wird durch die Aufrufe von `addDomain( )` festgelegt.

**Parameter:**

**int idx :** Es wird ab 1 gezählt.

**Fehlerfälle:** Es wird `G_DomainBottom` zurückgegeben und folgende Meldung ausgegeben.

**dom204** Der Index `idx` liegt außerhalb der erlaubten Grenzen.

**Aufwand:**  $O(1)$

`G_id G_domainSequence::getNthId ( int idx ) const`

Liefert den Bezeichner der `idx`'ten Komponente. Die Reihenfolge wird durch die Aufrufe von `addDomain( )` festgelegt.

**Rückgabewerte:** Bei einer Sequenz ohne Bezeichner, wie sie für Tupel-Wertebereiche benötigt wird, wird `G_IdBottom` zurückgegeben.

**Fehlerfälle:** Es wird `G_DomainBottom` zurückgegeben und folgende Meldung ausgegeben.

**dom204** Der Index `idx` liegt außerhalb der erlaubten Grenzen.

**Aufwand:**  $O(1)$

### 4.3.5 Tupelwertebereiche

Zum Erzeugen von neuen Tupelwertebereichen muß zuerst eine Sequenz von Wertebereichen mit der Klasse `G_domainSequence` erzeugt werden.

`static G_domain G_domain::newTuple ( const G_domainSequence &domSeq  
)`

Erzeugt einen Tupelwertebereich, der über den Komponenten aus `domSeq` definiert ist.

**Rückgabewerte:** Wenn schon ein Tupelwertebereich über dieselbe Sequenz von Wertebereichen existiert, wird ein Verweis auf diesen zurückgegeben.

**Fehlerfälle:** Es wird `G_DomainBottom` zurückgegeben und folgende Meldung ausgegeben.  
**dom210** In `domSeq` sind keine Wertebereiche eingetragen.

**Aufwand:**  $O(|\text{domSeq}|)$

```
bool G_domain::isTuple ( ) const
```

Testet, ob es ein Tupel-Wertebereich ist.

**Rückgabewerte:** `true`, wenn die Instanz ein Tupelwertebereich aus beliebigen Komponenten ist.

**Fehlerfälle:** Es wird `false` zurückgegeben und folgende Meldung ausgegeben.  
**dom001** Die Instanz enthält keinen Wertebereich.

**Aufwand:**  $O(1)$

```
unsigned G_domain::getArity ( ) const
```

Liefert die Stelligkeit eines Tupels.

**Fehlerfälle:** Es wird `0` zurückgegeben und folgende Meldung ausgegeben.  
**dom202** Die Instanz enthält keinen Tupelwertebereich.

**Aufwand:**  $O(1)$

```
G_domain G_domain::getNthDomain ( unsigned idx ) const
```

Liefert den `idx`'ten Wertebereich.

**Parameter:**

**unsigned idx :** Es wird ab 1 gezählt.

**Fehlerfälle:** Es wird `G_DomainBottom` zurückgegeben und eine der folgenden Meldungen ausgegeben.

**dom202** Die Instanz enthält keinen Tupelwertebereich.

**dom204** Der Index `idx` liegt außerhalb des gültigen Bereiches.

**Aufwand:**  $O(1)$

#### 4.3.6 Record-Wertebereiche

Zum Erzeugen von neuen Record-Wertebereichen muß zuerst eine Sequenz von Bezeichner-Wertebereich-Paaren mit der Klasse `G_domainSequence` erzeugt werden.

```
static G_domain G_domain::newRecord ( const G_domainSequence &domSeq  
                                     )
```

Erzeugt einen Record-Wertebereich, der über den Komponenten aus `domSeq` definiert ist.

**Rückgabewerte:** Wenn es schon einen Record-Wertebereich gibt, der über dieselbe Menge von Bezeichner-Wertebereich-Paaren definiert ist, dann wird ein Verweis auf diesen zurückgegeben.

**Fehlerfälle:** Es wird `G_DomainBottom` zurückgegeben und eine der folgenden Meldungen ausgegeben.

**dom210** In `domSeq` sind keine Wertebereiche eingetragen.

**dom211** In `domSeq` fehlt ein Selektorbezeichner.

**Aufwand:**  $O(|\text{domSeq}|^2)$

```
bool G_domain::isRecord ( ) const
```

Testet, ob es ein Record-Wertebereich ist.

**Rückgabewerte:** `true`, wenn die Instanz ein Record aus beliebigen Komponenten ist.

**Fehlerfälle:** Es wird `false` zurückgegeben und folgende Meldung ausgegeben.

**dom001** Die Instanz enthält keinen Wertebereich.

**Aufwand:**  $O(1)$

```
unsigned G_domain::getAriety ( ) const
```

Liefert die Stelligkeit eines Records.

**Fehlerfälle:** Es wird 0 zurückgegeben und folgende Meldung ausgegeben.

**dom202** Die Instanz enthält keinen Record-Wertebereich.

**Aufwand:**  $O(1)$

```
G_domain G_domain::getNthDomain ( unsigned idx ) const
```

Liefert den `idx`'ten Wertebereich.

**Parameter:**

**unsigned idx :** Es wird ab 1 gezählt.

**Fehlerfälle:** Es wird `G_DomainBottom` zurückgegeben und eine der folgenden Meldungen ausgegeben.

**dom202** Die Instanz enthält keinen Record-Wertebereich.

**dom204** Der Index `idx` liegt außerhalb des gültigen Bereiches.

**Aufwand:**  $O(1)$

```
G_id G_domain::getNthSelector ( unsigned idx ) const
```

Liefert den Bezeichner der `idx`'ten Komponente.

**Parameter:**

**unsigned idx :** Es wird ab 1 gezählt.

**Fehlerfälle:** Es wird `G_IdBottom` zurückgegeben und eine der folgenden Meldungen ausgegeben.

**dom202** Die Instanz enthält keinen Record-Wertebereich.

**dom204** Der Index `idx` liegt außerhalb des gültigen Bereiches.

**Aufwand:**  $O(1)$

### 4.3.7 Liste von Bezeichnern

Beim Erzeugen eines neuen Aufzählungswertebereichs muß eine Liste von Bezeichnern angegeben werden. Mit der Klasse `G_idSequence` kann eine Folge von Bezeichnern erstellt werden.

**G\_idSequence::G\_idSequence ( )**

Erzeugt eine neue, leere Bezeichnerliste.

**Aufwand:**  $O(1)$

```
bool G_idSequence::add ( G_id aId )
bool G_idSequence::add ( const char *aId )
```

Fügt den Bezeichner `aId` an das Ende der Bezeichnerliste an.

**Fehlerfälle:** Es wird `false` zurückgegeben und eine der folgenden Meldungen ausgegeben.

**dom003** Es konnte kein Arbeitsspeicher für die internen Tabellen angefordert werden.

**dom301** Der Bezeichner `aId` ist schon in der Liste eingetragen.

**Aufwand:**  $O(1)$ , bzw.  $O(|\text{Sequenz}|)$  wenn die interne Datenstruktur vergrößert wird.

```
G_id G_idSequence::getNthId ( unsigned idx ) const
```

Liefert den `idx`'ten Bezeichner in der Liste. Die Reihenfolge wird durch die Aufrufe von `add()` festgelegt.

**Parameter:**

**int idx :** Es wird ab 1 gezählt.

**Fehlerfälle:** Es wird `G_idBottom` zurückgegeben und folgende Meldung ausgegeben.

**dom304** Der Index `idx` liegt außerhalb des gültigen Bereiches.

**Aufwand:**  $O(1)$

### 4.3.8 Aufzählungswertebereiche

Zum Erzeugen eines neuen Aufzählungswertebereichs muß zuerst eine Liste von Konstanten mit der Klasse `G_idSequence` erzeugt werden. Die Konstanten bilden in der Reihenfolge, wie sie in der Liste definiert sind, eine feste Ordnung. Dabei bekommt die zuerst eingetragene Konstante die *Ordnungsnummer* 1, die nachfolgenden werden aufsteigend numeriert. Ein neuer Wert eines Aufzählungswertebereiches wird mit der ersten Konstante initialisiert.

**Beispiel:** Im folgenden Beispiel wird ein Wertebereich für die vier Jahreszeiten angelegt. Sie sind so geordnet, daß „*Frühling*“ der erste und „*Winter*“ der letzte konstante Wert ist.

```
G_idSequence seasonConsts;
seasonConsts.add ("Fruehling");
seasonConsts.add ("Sommer");
```

```

seasonConsts.add ("Herbst");
seasonConsts.add ("Winter");
G_domain seasonDom = G_domain::newEnum (seasonConsts);

```

```

static G_domain G_domain::newEnum ( const G_idSequence &constSeq )

```

Erzeugt einen neuen Aufzählungswertebereich, der über den Konstanten in `constSeq` definiert ist.

**Rückgabewerte:** Wenn schon ein Aufzählungswertebereich über dieselbe Sequenz von Konstanten existiert, wird ein Verweis auf diesen zurückgegeben.

**Fehlerfälle:** Es wird `G_DomainBottom` zurückgegeben und folgende Meldung ausgegeben.  
**dom310** In `constSeq` sind keine Konstanten eingetragen.

**Aufwand:**  $O(|\text{constSeq}|)$

```

bool G_domain::isEnum ( ) const

```

Testet ob es ein Aufzählungswertebereich ist.

**Rückgabewerte:** `true`, wenn der Wertebereich ein Aufzählungswertebereich aus beliebigen Konstanten ist.

**Fehlerfälle:** Es wird `false` zurückgegeben und folgende Meldung ausgegeben.  
**dom001** Die Instanz enthält keinen Wertebereich.

**Aufwand:**  $O(1)$

```

unsigned G_domain::getCardinality ( ) const

```

Liefert die Anzahl der Konstanten in einem Aufzählungswertebereich.

**Fehlerfälle:** Es wird `0` zurückgegeben und folgende Meldung ausgegeben.  
**dom302** Die Instanz enthält keinen Aufzählungswertebereich.

**Aufwand:**  $O(1)$

```

G_id G_domain::getNthConst ( unsigned idx ) const

```

Liefert die `idx`'te Konstante des Aufzählungswertebereiches.

**Parameter:**

**unsigned idx :** Es wird ab 1 gezählt.

**Fehlerfälle:** Es wird `G_IdBottom` zurückgegeben und eine der folgenden Meldungen ausgegeben.

**dom302** Die Instanz enthält keinen Aufzählungswertebereich.

**dom304** Der Index `idx` liegt außerhalb des gültigen Bereiches.

**Aufwand:**  $O(1)$

```

unsigned G_domain::getOrd ( G_id aId ) const

```

```

unsigned G_domain::getOrd ( const char *aId ) const

```

Liefert die Ordnungsnummer der Konstante `aId`.

Wertebereich	Nullwert
BOOL	false
INT	0
STRING	" "      leerer String
DOUBLE	0.0
List	<>      leere Liste
Tuple	je Komponente ihren Nullwert
Record	je Komponente ihren Nullwert
Enum	die erste definierte Aufzählungskonstante

Tabelle 4.2: Nullwerte

**Fehlerfälle:** Es wird 0 zurückgegeben und eine der folgenden Meldungen ausgegeben.

**dom302** Die Instanz enthält keinen Aufzählungswertebereich.

**dom311** Der Aufzählungswertebereich enthält keine Konstante aId.

**Aufwand:**  $O(|\text{constSeq}|)$

## 4.4 Werte: g\_valref.c

*Werte* sind einzelne Elemente aus Wertebereichen. Wenn man sich die Wertebereiche als *Klassen* vorstellt, dann sind *Werte* *Instanzen* von Wertebereichen.

Für die Behandlung von Werten wird die Klasse `G_valueRef` bereitgestellt. Instanzen dieser Klasse können Referenzen zu beliebigen Werten sein und enthalten neben einem Zeiger auf den Datenbereich die Informationen über den Wertebereich, dem die Daten angehören. Mit diesen Referenzen kann ähnlich wie mit *C*-Zeigern umgegangen werden.

### 4.4.1 Nullwerte

Für jeden Wert muß ausreichend Arbeitsspeicher reserviert und verwaltet werden. Damit kein Wert einen undefinierten Zustand einnimmt, werden sie bei der Allokation direkt initialisiert. Als *Nullwerte* werden die Konstanten aus Tabelle 4.2 eingetragen.

### 4.4.2 Funktionalität der Wertereferenzen

```
G_valueRef::G_valueRef ( )
```

Legt eine neue, uninitialisierte Referenz an.

**Aufwand:**  $O(1)$

```
G_valueRef & G_valueRef::operator= ( const G_valueRef &aValue )
```

Weist der Instanz die Referenz `aValue` zu. Es wird kein neuer Wert angelegt. Beide Referenzen verweisen dann auf dieselbe Datenstruktur.

**Aufwand:**  $O(1)$

**G\_valueRef::~G\_valueRef ( )**

Löscht nur die Wertreferenz — der reservierte Speicherbereich bleibt erhalten.

**Aufwand:**  $O(1)$

### 4.4.3 Graphunabhängige Werte

Es ist möglich, neben den graphabhängigen Werten, die als Attribute von Graphenelementen auftreten, auch graphunabhängige Werte zu benutzen. Diese werden benötigt, um Elemente zu füllen, die zu Mengen oder Listen hinzugefügt werden.

**bool G\_valueRef::createValue ( G\_domain aDom )**

Erzeugt einen neuen Wert aus dem Wertebereich aDom, initialisiert diesen mit einem geeigneten Nullwert (siehe Absch. 4.4.1, S. 113) und aktualisiert die Referenz.

**Fehlerfälle:**

Wenn der Wert nicht angelegt werden kann (Arbeitsspeicher reicht nicht), wird `false` zurückgegeben.

**Aufwand:** Bei strukturierten Wertebereichen wird die Initialisierung rekursiv für alle Komponenten durchgeführt.

**G\_valueRef::G\_valueRef ( G\_domain aDom )**

Legt eine neue Referenz und die Speicherstruktur für einen Wert aus aDom an. Dieser Wert wird mit einem geeigneten Nullwert initialisiert (siehe Absch. 4.4.1, S. 113).

**Fehlerfälle:**

Wenn der Wert nicht angelegt werden kann (Arbeitsspeicher reicht nicht), wird eine uninitialisierte Referenz erzeugt.

**Bemerkung:** Dieses ist eine Abkürzung für den Default-Konstruktor mit nachfolgendem Aufruf von `createValue()`.

**Aufwand:** Bei strukturierten Wertebereichen wird die Initialisierung rekursiv für alle Komponenten durchgeführt.

**bool G\_valueRef::deleteValue ( )**

Gibt den Arbeitsspeicher, der für den Wert reserviert wurde, wieder frei.

**Fehlerfälle:** Es wird `false` zurückgegeben und eine der folgenden Meldungen ausgegeben.

**val021** Die Instanz ist ein Attributwert eines Graphenelementes. Diese können nur insgesamt mit `G_graph::deleteVertex()` oder `G_graph::deleteEdge()` gelöscht werden.

**val022** Der Wert bezieht sich auf einen unbekanntem Wertebereich.

**val023** Der Wert ist eine Komponente eines strukturierten Wertes. Es kann nur der gesamte strukturierte Wert gelöscht werden.

**Aufwand:** Bei strukturierten Werten müssen die Komponenten rekursiv gelöscht werden.

#### 4.4.4 Grundsätzliche Funktionalität von Werten

Werte können kopiert und zugewiesen werden.

`G_valueRef G_valueRef::copyValue ( ) const`

Liefert eine Wertreferenz, die auf eine Kopie des referierten Wertes verweist.

**Rückgabewerte:** Eine neue, graphunabhängige Wertreferenz, initialisiert mit einem neuen Speicherbereich für den Wert.

**Aufwand:** Bei strukturierten Werten müssen die Werte rekursiv kopiert werden.

`bool G_valueRef::assignValue ( const G_valueRef &aValue )`

Kopiert den Wert in `aValue` in den Speicherbereich auf den die Instanz verweist. Hierfür müssen beide Referenzen mit Werten aus demselben Wertebereich initialisiert sein. Der alte Wert der Instanz wird überschrieben.

**Fehlerfälle:** Es wird `false` zurückgegeben und folgende Meldung ausgegeben.

**typ001** Die Referenz `aValue` bezieht sich auf einen anderen Wertebereich.

**Aufwand:** Bei strukturierten Werten müssen die Komponenten rekursiv abgearbeitet werden.

`G_domain G_valueRef::getDomain ( ) const`

Liefert den Wertebereich, dem der Wert angehört.

**Aufwand:**  $O(1)$ , inline

Wertebereiche und Werte können auch ausgedruckt werden.

`ostream & G_valueRef::printDomain ( ostream &os ) const`

Gibt die Wertebereichsinformation auf `os` aus.

**Aufwand:** Bei strukturierten Datentypen müssen alle Komponenten rekursiv ausgegeben werden.

`ostream & G_valueRef::printValue ( ostream &os ) const`

Gibt den Wert auf `os` aus.

**Aufwand:** Bei strukturierten Datentypen müssen alle Komponenten rekursiv ausgegeben werden — bei Mengen und Listen alle Elemente.

`ostream & operator« ( ostream &os, const G_valueRef &aValue )`

Gibt die Wertebereichsinformation und den Wert der Referenz `aValue` auf den Stream `os` aus.

**Aufwand:** Setzt sich aus den Aufrufen von `printDomain()` und `printValue()` zusammen.

Weiterhin können beliebige Werte miteinander verglichen werden.

```
bool G_valueRef::isEqualValue ( const G_valueRef &aValue ) const
```

Testet, ob zwei Werte aus demselben Wertebereich sind und den gleichen Wert beinhalten.

**Aufwand:** Da die Werte strukturiert sein können, muß der Vergleich rekursiv durch alle Komponenten gehen.

#### 4.4.5 Werte der Basiswertebereiche

Basiswerte können gesetzt und ausgelesen werden.

```
void G_valueRef::updateBOOL ( bool aBool )
void G_valueRef::updateINT ( int aInt )
void G_valueRef::updateSTRING ( const char *aString )
void G_valueRef::updateDOUBLE ( double aDouble )
```

Setzt den Wert auf aBool, aInt, aString oder aDouble.

**Fehlerfälle:** Es wird eine der folgenden Meldungen ausgegeben.  
**val011** Der Wert wurde für einen anderen Wertebereich initialisiert.  
**val002** Es konnte kein Speicher für den Wertinhalt angelegt werden.

**Aufwand:**  $O(1)$

```
bool G_valueRef::getBOOL ( ) const
int G_valueRef::getINT ( ) const
const char * G_valueRef::getString ( ) const
double G_valueRef::getDOUBLE ( ) const
```

Liefert den Inhalt von Basiswerten.

**Rückgabewerte:** Bei Strings wird ein Zeiger in die reservierte Datenstruktur geliefert. Nachdem der String mit updateSTRING( ) geändert wurde, darf ein mit getString( ) ermittelter Zeiger nicht mehr benutzt werden.

**Fehlerfälle:** Es wird 0/NULL zurückgegeben und folgende Meldung ausgegeben.  
**val043** Der Wert wurde für einen anderen Wertebereich initialisiert.

**Aufwand:**  $O(1)$

#### 4.4.6 Listenwerte

Listen bestehen aus mehreren Elementen desselben Wertebereiches. Diese Elemente haben eindeutige Positionen in der Liste.

```
G_valueRef G_valueRef::insertIntoList ( const G_valueRef &aItem,
                                         unsigned pos )
```

Erzeugt an der Position pos der Liste Platz für ein neues Element und kopiert den Wert von aItem an diese Position.

**Parameter:**

**unsigned pos :** Es wird ab 1 gezählt. Falls die Position hinter der aktuellen Länge der

Liste liegt, wird das Element angefügt.

**Rückgabewerte:** Eine Wertereferenz, die auf den eingefügten Wert in der Liste referiert.

**Fehlerfälle:** Es wird `G_ValueBottom` zurückgegeben und eine der folgenden Meldungen ausgegeben.

**val031** Die Position `pos` ist ungültig.

**val032** Die Methode wurde auf einen Wert angewandt, der keine Liste darstellt.

**val033** Die Liste ist nicht über dem Wertebereich von `aItem` definiert.

**Aufwand:**  $O(1)$  und der Aufwand für das Kopieren des Wertes.

```
void G_valueRef::deleteFromList ( unsigned pos )
```

Löscht das `pos`'te Element aus der Liste und gibt den dafür reservierten Speicher frei.

**Parameter:**

**unsigned pos :** Es wird ab 1 gezählt.

**Fehlerfälle:** Es wird eine der folgenden Meldungen ausgegeben.

**val041** Die Position `pos` ist ungültig.

**val042** Die Methode wurde auf einen Wert angewandt, der keine Liste darstellt.

**Aufwand:**  $O(1)$  und der Aufwand für das rekursive Löschen strukturierter Werte.

```
bool G_valueRef::isMember ( const G_valueRef &aItem ) const
```

Testet, ob in der Liste ein Element mit demselben Wert wie `aItem` enthalten ist.

**Fehlerfälle:** Es wird `false` zurückgegeben und eine der folgenden Meldungen ausgegeben.

**val042** Die Methode wurde auf einen Wert angewandt, der keine Liste darstellt.

**val043** Die Liste ist nicht über dem Wertebereich von `aItem` definiert.

**Aufwand:** Für jedes Listenelement wird die Methode `isEqualValue()` aufgerufen.

```
int G_valueRef::getCardinality ( ) const
```

Liefert die Länge der Liste — also die Anzahl der Elemente in der Liste.

**Fehlerfälle:** Es wird `-1` zurückgegeben und folgende Meldung ausgegeben.

**val043** Der Wert wurde für einen anderen Wertebereich initialisiert.

**Aufwand:**  $O(1)$

```
G_valueRef G_valueRef::getNthListItem ( unsigned pos ) const
```

Liefert eine Referenz auf das `pos`'te Elemente der Liste.

**Parameter:**

**unsigned pos :** Es wird ab 1 gezählt.

**Fehlerfälle:** Es wird `G_ValueBottom` zurückgegeben und eine der folgenden Meldungen ausgegeben.

**val041** Die Position `pos` ist ungültig.

**val042** Die Methode wurde auf einen Wert angewandt, der keine Liste darstellt.

**Aufwand:**  $O(1)$

Das Traversieren aller Elemente einer Liste kann mit den folgenden drei Methoden wie folgt erledigt werden:

```
for (G_valueRef i = list.first (); list.OK (i); i = list.next (i))
    handleListItem (i);
```

G\_valueRef **G\_valueRef::first** ( ) const

Liefert das erste Element einer Liste.

**Rückgabewerte:** G\_ValueBottom, wenn die Liste leer ist.

**Fehlerfälle:** Es wird G\_ValueBottom zurückgegeben und folgende Meldung ausgegeben.

**val042** Die Methode wurde auf einen Wert angewandt, der keine Liste darstellt.

**Aufwand:**  $O(1)$

G\_valueRef **G\_valueRef::next** ( G\_valueRef &aValue ) const

Liefert das Element, welches in einer Liste hinter aValue liegt.

**Fehlerfälle:** Es wird G\_ValueBottom zurückgegeben und eine der folgenden Meldungen ausgegeben.

**val042** Die Methode wurde auf einen Wert angewandt, der keine Liste darstellt.

**val043** Die Liste ist nicht über dem Wertebereich von aValue definiert.

**val044** Das Element aValue stammt nicht aus der Liste.

**Aufwand:**  $O(1)$

bool **G\_valueRef::OK** ( const G\_valueRef &aValue ) const

Überprüft, ob aValue ein Element dieser Liste ist.

**Fehlerfälle:** Es wird false zurückgegeben und eine der folgenden Meldungen ausgegeben.

**val003** aValue enthält einen ungültigen Wert.

**val004** aValue enthält einen ungültigen Wert.

**val042** Die Methode wurde auf einen Wert angewandt, der keine Liste darstellt.

**val043** Die Liste ist nicht über dem Wertebereich von aValue definiert.

**val044** Das Element aValue stammt nicht aus der Liste.

**Aufwand:**  $O(1)$

#### 4.4.7 Tupelwerte

Ein Tupel wird über einer Sequenz von mehreren Komponenten gebildet. Auf die einzelnen Komponenten kann mit der flg. Methode zugegriffen werden.

G\_valueRef **G\_valueRef::getNthTupleItem** ( unsigned pos ) const

Liefert eine Referenz auf die pos'te Komponente des Tupels.

**Parameter:**

**unsigned pos** : Es wird ab 1 gezählt.

**Fehlerfälle:** Es wird `G_ValueBottom` zurückgegeben und eine der folgenden Meldungen ausgegeben.

**val051** Es gibt keine `pos`'te Komponente.

**val052** Die Methode wurde auf einen Wert angewandt, der keinen Tupel enthält.

**Aufwand:**  $O(1)$

#### 4.4.8 Recordwerte

Ein Record besteht aus mehreren Komponenten, die über eindeutige Selektorbezeichner angesprochen werden. Auf die einzelnen Komponenten kann mit der flg. Methode zugegriffen werden.

```
G_valueRef G_valueRef::getRecordItem ( const char *sel )
```

```
G_valueRef G_valueRef::getRecordItem ( const G_id &sel )
```

Liefert eine Referenz auf die Komponente mit dem Bezeichner `sel`.

**Fehlerfälle:** Es wird `G_ValueBottom` zurückgegeben und eine der folgenden Meldungen ausgegeben.

**val061** Die Methode wurde auf einen Wert angewandt, der keinen Record enthält.

**val062** Der Record enthält keine Komponente mit dem Bezeichner `sel`.

**Aufwand:**  $O(\#Record\text{-Komponenten})$

#### 4.4.9 Aufzählungswerte

Ein Wert eines Aufzählungswertebereiches ist eine der Konstanten, die im Wertebereich registriert sind. Diese Konstanten haben im Aufzählungswertebereich eine feste Ordnung. Die Ordnungsnummer kann abgefragt werden und z.B. mit anderen Werten verglichen werden.

```
G_id G_valueRef::getEnum ( ) const
```

Liefert die Aufzählungskonstante des Wertes.

**Fehlerfälle:** Es wird `G_IdBottom` zurückgegeben und folgende Meldung ausgegeben.

**val043** Die Methode wurde auf einen Wert angewandt, der keinen Aufzählungstyp enthält.

**Aufwand:**  $O(1)$

```
void G_valueRef::updateEnum ( G_id aId )
```

```
void G_valueRef::updateEnum ( const char *aId )
```

Setzt den Wert auf die Aufzählungskonstante `aId`

**Fehlerfälle:** Es wird eine der folgenden Meldungen ausgegeben.

**val011** Die Methode wurde auf einen Wert angewandt, der keinen Aufzählungstyp enthält.

**dom311** Die Konstante `aId` ist nicht im Aufzählungswertebereich registriert.

**Aufwand:**  $O(|\text{constSeq}|)$

```
unsigned G_valueRef::getOrd ( ) const
```

Liefert die Ordnungsnummer der Aufzählungskonstante im Wertebereich.

**Rückgabewerte:** Die Ordnungsnummern werden ab 1 gezählt.

**Fehlerfälle:** Es wird 0 zurückgegeben und folgende Meldung ausgegeben.

**val043** Die Methode wurde auf einen Wert angewandt, der keinen Aufzählungstyp enthält.

**Aufwand:**  $O(1)$

```
void G_valueRef::updateByOrd ( unsigned aOrd )
```

Setzt den Wert auf die Aufzählungskonstante mit der Ordnungsnummer aOrd.

**Fehlerfälle:** Es wird eine der folgenden Meldungen ausgegeben.

**val011** Die Methode wurde auf einen Wert angewandt, der keinen Aufzählungstyp enthält.

**dom304** Die Ordnungsnummer aOrd liegt außerhalb des gültigen Bereiches.

**Aufwand:**  $O(1)$

## 4.5 Undo-Puffer: g\_undo.c

Zur Verwendung des Undo darf die Graphenlabor-Anwendung **nicht** mit der Option „-DNO\_UNDO“ übersetzt und muß mit einer „undofähigen“ Graphenlaborbibliothek gebunden werden. Derartige Bibliotheksvarianten erkennt man an einem „u“ im Dateinamen (z.B. „libgraphu.a“).

### 4.5.1 Die Klasse G\_undoBuffer

Die Klasse `G_undoBuffer` realisiert die Verwaltung der Undo-Information.

#### Erzeugen und Löschen eines Undo-Puffers

Ein Undo-Puffer wird durch den Konstruktor `G_undoBuffer::G_undoBuffer` erzeugt und mit dem Destruktor `G_undoBuffer::~G_undoBuffer` wieder gelöscht.

```
G_undoBuffer::G_undoBuffer ( )
```

```
G_undoBuffer::G_undoBuffer ( unsigned size )
```

Erzeugt einen Undo-Puffer, der `size` Protokolleinträge<sup>5</sup> verwalten kann.

**Fehlerfälle:** Es wird folgende Meldung ausgegeben.

**undo003** Der Speicherplatz reicht nicht für einen Puffer der angegebenen Kapazität `size`. Es wurde ein Puffer mit geringerer Kapazität angelegt.

**Bemerkung:** Falls sich bei der Benutzung des Puffers herausstellt, daß seine Kapazität nicht zur Aufnahme der Undo-Information ausreicht, wird der Puffer automatisch vergrößert.

**Aufwand:** Es wird die Freispeicherverwaltung des Systems benutzt.

---

<sup>5</sup> Zur Protokollierung einer graphverändernden Aktion werden unterschiedlich viele Einträge benötigt, z.B. für `G_graph::createVertex()` ca. 10 Einträge.

**G\_undoBuffer::~G\_undoBuffer ( )**

Gibt den vom Undo-Puffer belegten Speicherplatz wieder frei. Dazu werden alle protokollierten Änderungen mittels `commit()` bestätigt.

**Markierungen setzen**

Markierungen dienen zum Festsetzen von Bezugspunkten in einem Undo-Puffer für gezieltes Bestätigen und Rücksetzen von Änderungen. Mehrere Markierungen können durch einen `unsigned`-Wert unterschieden werden.

```
void G_undoBuffer::mark ( unsigned markerValue )
```

Trägt eine Markierung mit dem Wert `markerValue` in den Undo-Puffer ein.

**Fehlerfälle:** Es wird folgende Meldung ausgegeben.

**undo001** Der Puffer konnte nicht vergrößert werden. Alle eingetragenen Änderungen wurden bestätigt.

**Aufwand:**  $O(1)$ , wenn die interne Speicherstruktur nicht vergrößert werden muß.

```
int G_undoBuffer::isMarked ( unsigned markerValue )
```

Sucht im Undo-Puffer nach einer Markierung mit dem Wert `markerValue`. Falls eine solche Markierung gefunden wurde, liefert die Methode `true`, andernfalls `false`.

**Aufwand:**  $O(|\text{undoBuffer}|)$

**Änderungen bestätigen**

Bei der Bestätigung von Änderungen wird der Undo-Puffer in FIFO-Reihenfolge geleert. Dabei werden alle mit `pushCommitAction()` und `pushAction()` gespeicherten Aktionen ausgeführt.

```
void G_undoBuffer::commit ( )
```

```
void G_undoBuffer::commit ( unsigned markerValue )
```

Bestätigt alle im Puffer protokollierten Änderungen bis zur ersten Markierung mit dem Wert `markerValue`. Existiert keine solche Markierung, wird der komplette Pufferinhalt abgearbeitet. Die Markierung selbst wird entfernt. Falls wegen zu vieler Eintragungen ein Überlauf vermerkt worden ist, wird dieser Vermerk gelöscht, und der Puffer ist wieder benutzbar.

**Aufwand:**  $O(|\text{undoBuffer}|)$  und der Aufwand der eingetragenen Funktionen für `action()` und `commitAction()`.

**Änderungen zurücksetzen**

Beim Zurücksetzen von Änderungen wird der Undo-Puffer in LIFO-Reihenfolge bearbeitet. Dabei werden alle protokollierten Zuweisungen zurückgenommen sowie alle mit `pushUndoAction()` und `pushAction()` gespeicherten Aktionen ausgeführt.

```
void G_undoBuffer::undo ( )
void G_undoBuffer::undo ( unsigned markerValue )
```

Setzt alle im Puffer protokollierten Änderungen seit der letzten Markierung mit dem Wert `markerValue` zurück. Existiert keine solche Markierung, wird der komplette Pufferinhalt abgearbeitet. Die Markierung selbst wird entfernt.

**Fehlerfälle:** Es wird folgende Meldung ausgegeben.

**undo002** Seit dem letzten `commit ( )`-Aufruf trat Pufferüberlauf auf. Es wurden keine Änderung zurückgesetzt.

**Aufwand:**  $O(|\text{undoBuffer}|)$  und der Aufwand der eingetragenen Funktionen für `action ( )` und `undoAction ( )`.

### Zuweisungen protokollieren

Mit den folgenden Methoden können Zuweisungen im Puffer protokolliert werden, die bei einem `undo ( )`-Aufruf zurückgesetzt werden sollen.

```
void G_undoBuffer::pushAssignInt ( int &intVar )
```

Protokolliert eine Zuweisung an die `int`-Variable `intVar`. Diese Methode muß vor der zu protokollierenden Zuweisung aufgerufen werden.

**Beispiel:** Die Zuweisung an die Variable `varX` soll im Undo-Puffer `uB` protokolliert werden:

```
uB.pushAssignInt(varX);
varX = <expr>;
```

**Fehlerfälle:** Es wird folgende Meldung ausgegeben.

**undo001** Der Puffer ist voll und konnte die Zuweisung nicht aufnehmen. Alle eingetragenen Änderungen werden bestätigt und der Überlauf im Puffer vermerkt, so daß ein folgender `undo ( )`-Aufruf einen weiteren Fehler erzeugt. Bis zum nächsten `commit ( )`-Aufruf werden keine Einträge mehr aufgenommen.

**Aufwand:**  $O(1)$ , wenn der Puffer nicht vergrößert wird.

```
void G_undoBuffer::pushAssignUInt ( unsigned &uIntVar )
```

Protokolliert eine Zuweisung an die `unsigned`-Variable `uIntVar`. Diese Methode muß vor der zu protokollierenden Zuweisung aufgerufen werden.

**Fehlerfälle:**

Wie bei `pushAssignInt ( )`.

**Aufwand:**  $O(1)$ , wenn der Puffer nicht vergrößert wird.

```
void G_undoBuffer::pushAssignPointer ( void * &pVar )
```

Protokolliert eine Zuweisung an die Zeiger-Variable `pVar`. Diese Methode muß vor der zu protokollierenden Zuweisung aufgerufen werden.

**Fehlerfälle:**

Wie bei `pushAssignInt ( )`.

**Aufwand:**  $O(1)$ , wenn der Puffer nicht vergrößert wird.

### Eintragen von Aktionen

Mit den folgenden Methoden können Aktionen in den Undo-Puffer eingetragen werden, die erst bei einem Aufruf von `undo()`, `commit()` oder bei beiden ausgeführt werden sollen.

```
void G_undoBuffer::pushUndoAction ( void (*undoAction)(void *),  
                                     void *parameter )
```

Vermerkt im Undo-Puffer, daß bei einem `undo()`-Aufruf die Funktion `undoAction` mit dem Parameter `parameter` aufgerufen werden soll.

**Fehlerfälle:**

Wie bei `pushAssignInt()`. Falls es zum Überlauf des Puffers kam, wird die übergebene Aktion nicht ausgeführt.

**Aufwand:**  $O(1)$ , wenn der Puffer nicht vergrößert wird.

```
void G_undoBuffer::pushCommitAction ( void (*commitAction)(void *),  
                                       void *parameter )
```

Vermerkt im Undo-Puffer, daß bei einem `commit()`-Aufruf die Funktion `commitAction` mit dem Parameter `parameter` aufgerufen werden soll.

**Fehlerfälle:**

Wie bei `pushAssignInt()`. Falls es zum Überlauf des Puffers kam, wird die übergebene Aktion sofort ausgeführt.

**Aufwand:**  $O(1)$ , wenn der Puffer nicht vergrößert wird.

```
void G_undoBuffer::pushAction ( void (*action)(void *), void  
                                *parameter )
```

Vermerkt im Undo-Puffer, daß sowohl bei einem `commit()`- als auch bei einem `undo()`-Aufruf die Funktion `action` mit dem Parameter `parameter` aufgerufen werden soll.

**Fehlerfälle:**

Wie bei `pushAssignInt()`. Falls es zum Überlauf des Puffers kam, wird die übergebene Aktion sofort ausgeführt.

**Aufwand:**  $O(1)$ , wenn der Puffer nicht vergrößert wird.

### Pufferstatus prüfen

Mit den folgenden Methoden kann geprüft werden, ob ein Überlauf eingetreten ist oder für wieviele Eintragungen der Puffer noch Platz bietet.

```
bool G_undoBuffer::isOk ( )
```

Liefert `true`, falls kein Überlauf eingetreten ist, andernfalls `false`.

```
unsigned G_undoBuffer::isAvailable ( )
```

Liefert die Anzahl der noch freien Einträge. Nach einem Überlauf liefert die Methode immer den Wert 0.

## 4.6 Temporäre Attribute

Die Klasse `G_tempAttribute` ist eine Basisklasse aller Klassen von temporären Attributen. Zum Markieren von Graphenelementen müssen Attributsklassen von dieser Basisklasse abgeleitet werden. Darin können beliebige Werte definiert werden. Das Graphenlabor hat hierdurch keinen Zugriff auf die Werte von temporären Attributen. Sollen Veränderungen an diesen Werten im Undo-Puffer protokolliert werden, muß dies explizit programmiert werden.

Die temporären Attribute werden in separaten Schichten für Knoten und Kanten verwaltet. Jeder Schicht wird vom Graphenlabor eine eindeutige Identifikationsnummer zugewiesen.

Die Schnittstelle zum Graphenlabor wird über einige Methoden realisiert, die in der abstrakten Basis-klasse definiert sind und von den abgeleiteten Klassen überschrieben werden.

### 4.6.1 Konstruktoren, Destruktoren

```
G_tempAttribute::G_tempAttribute ( ... )
```

Initialisiert das temporäre Attribut. Abgeleitete Klassen können hier ihre Werte initialisieren.

```
virtual G_tempAttribute::~~G_tempAttribute ( )
```

Löst das temporäre Attribut. Speicher, der von der Freispeicherverwaltung angefordert wurde, muß hier wieder freigegeben werden.

**Bemerkung:** Einige Compiler verlangen, daß Klassen, die virtuelle Methoden enthalten, auch einen virtuellen Destruktor haben. In diesem Fall wird ein Destruktor mit leerem Rumpf definiert.

### 4.6.2 Ausgabe der Attributinhalte

Wenn die Knoten- oder Kanteninformationen in *lesbarer* Form ausgegeben werden (dieses geschieht z.B. bei `G_graph::printVertex()` und `G_graph::printEdge()`), dann sollten auch die Informationen der temporären Attribute dargestellt werden.

```
virtual ostream & G_tempAttribute::print ( ostream &os )
```

Schreibt den Inhalt des Attributes in den Ausgabestrom `os` und liefert diesen zurück. In welcher Form die Ausgabe genau erfolgt, ist für jede Klasse unterschiedlich, so daß diese Methode in jeder abgeleiteten Klasse überschrieben werden sollte.

### 4.6.3 Schichten von temporären Attributen

Man kann mit den Methoden `G_graph::createVTemp()` und `G_graph::createETemp()` mehrere Schichten temporärer Attribute angelegen.

```
int G_tempAttribute::getLayerId ( ) const
```

Liefert die Identifikationsnummer der Schicht, der das temporäre Attribut angehört.

```
G_tempAttribute * G_tempAttribute::getNextInStack ( ) const
```

Liefert das nächste temporäre Attribut des Graphenelementes aus dem Stapel der Schichten.

**Bemerkung:** Wenn zu einem Graphenelement in einer Schicht kein temporäres Attribut eingetragen wurde, dann wird diese Schicht übersprungen. Deshalb sollte mit `getLayerId()` überprüft werden, ob das gefundene Attribut aus der gewollten Schicht ist.

### 4.6.4 Anbindung an den Undo-Puffer

Wenn ein Graph mit einem Undo-Puffer verbunden ist, dann dürfen beim Löschen von Graphenelementen die temporären Attribute nicht sofort gelöscht werden. Sie müssen zuerst nur als *gelöscht* markiert werden.

```
virtual void G_tempAttribute::logicalDelete ( )
```

Wird vom Graphenlabor immer dann aufgerufen, wenn das zugehörige Attributobjekt logisch gelöscht wird (z.B. bei `deleteVertex()`). Der eigentliche Destruktoraufruf erfolgt erst beim Bestätigen der Änderungen (`commit()`). Falls ein Attributobjekt zu einem Zeitpunkt gelöscht wird, zu dem keine Änderungen protokolliert werden (wenn der Graph mit keinem Undo-Puffer verbunden ist), wird sowohl der logische als auch der richtige Destruktor vom Labor aufgerufen.

```
virtual void G_tempAttribute::logicalUndelete ( )
```

Wird dann aufgerufen, wenn das zugehörige Attributobjekt zuvor logisch gelöscht wurde, und dieses Löschen durch einen `undo()`-Aufruf zurückgenommen werden soll.

## 4.7 Bezeichner: g\_id.c

Bei Record-Wertebereichen und Attributen werden viele Bezeichner auftauchen. Dabei werden immer nur neue Bezeichner eingeführt, nie alte gelöscht (außer beim Löschen eines Typsystems). Die Texte der Bezeichner werden in einem global geführten Speicherbereich gehalten. Ein einzelner Bezeichner wird durch einen Index in diese Tabelle dargestellt.

Aus Effizienzgründen können Bezeichner doppelt in der Tabelle enthalten sein. D.h. wenn ein neuer Bezeichner definiert wird, wird nicht überprüft, ob es den Bezeichner schon gibt. Der neue Bezeichner wird immer an das Ende der Tabelle angefügt und eine Referenz darauf erstellt.

## Schnittstelle

Einzelne Bezeichner sind Instanzen der Klasse `G_id`. Über diese Variablen werden die Namen der Bezeichner in die Stringtabelle eingetragen.

## Konstanten

**`G_id G_IdBottom`:** Diese Konstante stellt einen *ungültigen* Bezeichner dar und wird benutzt, um eine fehlerhafte Rückgabe bei Methoden auszudrücken.

**`G_id G_IdNull`:** Diese Konstante beinhaltet einen *eindeutigen* Null-String. Sie wird von Methoden benutzt, die mit einem Fehler enden, aber einen druckbaren Bezeichner liefern müssen.

## Konstruktoren, Destruktoren

**`G_id::G_id ( )`**

Erzeugt eine neue Instanz eines Bezeichners. Die Referenz verweist auf `G_IdBottom`.

**Aufwand:**  $O(1)$ , inline

**`G_id::G_id ( const char *aName )`**

Trägt den Namen `aName` in die Stringtabelle ein und erzeugt einen neuen Bezeichner, der darauf verweist.

**Fehlerfälle:** Es wird folgende Meldung ausgegeben.

**mem301** Der Name konnte wegen Speichermangel nicht in die Stringtabelle eingetragen werden.

**Aufwand:**  $O(|aName|)$

**`G_id::G_id ( const G_id &aId )`**

Erzeugt einen zweiten Bezeichner, der auf denselben Namen verweist.

**Aufwand:**  $O(1)$ , inline

**`G_id & G_id::operator= ( const G_id &aId )`**

Weist der Instanz den Bezeichner `aId` zu. Danach verweisen beide Bezeichner auf den selben String.

**Aufwand:**  $O(1)$ , inline.

## Sonstige Methoden

**`const char * G_id::str ( ) const`**

Wandelt einen Bezeichner in einen String (`const char *`) um.

**Rückgabewerte:** Der zugehörige Zeiger auf einen Namen oder NULL, falls der Bezeichner ungültig ist. Dieser Zeiger kann ungültig werden, wenn durch das Eintragen neuer Namen in die Stringtabelle der Speicherblock verschoben wurde. Deshalb sollte dieser Zeiger nach Funktionsaufrufen wieder neu ermittelt werden.

**Aufwand:**  $O(1)$ , inline.

```
bool G_id::OK ( ) const
```

Testet ob der Bezeichner gültig ist.

**Aufwand:**  $O(1)$ , inline.

## Vergleichsoperationen

```
bool G_id::operator== ( const G_id &aId ) const
```

Vergleicht, ob zwei Bezeichner auf denselben Namen verweisen — es wird der Text verglichen.

**Aufwand:**  $O(|name(aId)|)$

```
bool G_id::operator== ( const char *aName ) const
```

Vergleicht einen Bezeichner mit dem String aName.

**Aufwand:**  $O(|aName|)$

## 4.8 Tracing : g\_trace.c

Die Klasse verwaltet Steuerinformation zur Laufzeitverfolgung. Für jeden Quelltext (also in jeder *name.c*-Datei), in dem zu verfolgende Funktionen oder Methoden stehen, sollte mit dem Makro `G_TrceDeclaration` ein Steuerobjekt (unmittelbar nach den `include`-Anweisungen) definiert werden. Dieses Objekt wird durch seinen Konstruktor in eine globale Tabelle mit dem Namen der Quelltextdatei eingetragen, ist ansonsten aber nur in dem Quelltext unter dem Bezeichner `G_TrceObj` sichtbar (da als `static` deklariert). Die durch die Makros `G_trc`, `G_trcEnter` und `G_trcLeave` geklammerten Anweisungen werden nur dann ausgeführt, wenn für den Quelltext Tracing eingeschaltet ist. Tracing kann für Quelltexte mit der Methode `set()` ein- bzw. ausgeschaltet werden. Tracing-Ausgaben erfolgen immer in den Ausgabestrom, auf den die Variable `G_trace::out` (Typ: `ostream*`) zeigt. Sie zeigt zunächst auf `cout`, die Ausgabe kann aber mit der Methode `G_trace::setFile()` in eine Datei umgeleitet werden.

```
G_trace::G_trace ( const char *name )
```

Erzeugt ein Kontrollobjekt und trägt es unter dem Namen *name* in die globale Tabelle der Kontrollobjekte ein.

**Bemerkung:** Das Anwendungsprogramm sollte diesen Konstruktor nicht direkt aufrufen, sondern das Makro `G_TrceDeclaration` verwenden.

```
static int G_trace::set ( const char *pattern, int flag )
```

Schaltet für alle Quelltextdateien, deren Namen dem Suchmuster `pattern` genügen, Tracing gemäß `flag` aus (`flag=false`) oder ein (`flag=true`). Das Muster darf die Wildcards „\*“ (null oder mehrere beliebige Zeichen) und „?“ (genau ein beliebiges Zeichen) enthalten. Die Methode liefert die Anzahl der durch das Muster getroffenen Dateien zurück.

**Bemerkung:** Einige Compiler liefern beim Makro `__NAME__` den Dateinamen mit kompletter Pfadangabe, andere nur den Dateinamen zurück. Man sollte deshalb den Pfad im `pattern` als Wildcard angeben (Bsp.: `*/g_trace.c`).

```
static void G_trace::set ( int flag )
```

Schaltet Tracing global gemäß `flag` aus oder ein, ohne die Auswahl an Quelltextdateien zu verändern.

```
static void G_trace::setFile ( const char *fileName )
```

Öffnet eine Datei unter dem Namen `fileName` für Schreibzugriffe und lenkt alle folgenden Tracingausgaben in diese um. Falls zuvor bereits Tracing umgeleitet war, wird die alte Datei geschlossen.

## 4.9 Meldungsausgabe `g_msg.c`

Die Klasse `G_msg` dient der Steuerung der Ausgabe von Warn- und Fehlermeldungen. Die auszugebenden Meldungen werden aus einer Datei gelesen, die sich einem der von Environmentvariable `G_MSGPATH` angegebenen Verzeichnisse befindet. Mit der Methode `addDirectory()` können weitere Verzeichnisse angegeben werden.

Für die unterschiedlichen Arten der Meldungen gibt es drei global verfügbare Instanzen der Klasse `G_msg`.

**G\_warning** Die Meldungen werden formatiert und mit dem Kopftext für Warnmeldungen ausgegeben.

**G\_error** Diese Meldungen werden mit dem Kopftext für Fehlermeldungen ausgegeben. Gleichzeitig werden diese Meldungen gezählt. Wenn der Wert aus der Umgebungsvariablen `G_MAXERROR` überschritten wird, wird das Programm mit dem Aufruf der Systemfunktion `exit(1)` abgebrochen.

**G\_fatal** Diese Meldungen werden mit dem Kopftext für fatale Fehler ausgegeben. Danach wird das Programm sofort mit einem Funktionsaufruf `exit(2)` abgebrochen.

```
void G_msg::msg ( int nr, const char *group, const char *fun, ... )
```

Gibt die Meldung mit der Nummer `nr` zur Gruppe `group` evtl. mit weiteren Parametern in die Standardfehlerausgabe (`stderr`) als Meldung aus. Dabei wird `fun` als diejenige Funktion angegeben, die die Fehlersituation erkannt hat. `fun` darf Umwandlungsspezifizierer enthalten, für die weitere Parameter übergeben werden müssen. Das Labor lädt dazu den Text dieser Meldung aus einer Datei mit dem Namen `msggroup` aus einem der in Variable `G_MSGPATH` angegebenen Verzeichnisse. Enthält dieser Text Umwandlungsspezifizierer gemäß der Standardfunktion `printf()` (z.B. `%d`, `%s` etc.), müssen der Methode

entsprechend weitere Parameter übergeben werden.

**Bemerkung:** Zum Format der Meldungstextdateien siehe Anhang A.3, S. 137.

**Fehlerfälle:**

Falls die Fehlermeldungsdatei nicht geöffnet werden kann, wird anstelle des Fehlermeldungstextes die Nachricht „Can't open message file *fileName*“ ausgegeben. Falls die Meldungsdatei zwar geöffnet werden kann, sie aber keine Meldung mit der geforderten Nummer enthält, wird die Nachricht „Can't find message number *nr*“ ausgegeben.

```
static void G_msg::setHeaders ( const char *fileName )
```

Liest die Einleitungstexte der Meldungen aus der Datei *fileName* ein. Zum Format dieser Datei siehe Anhang A.3.1, S. 137.

```
static void G_msg::addDirectory ( const char *directory )
```

Fügt das Verzeichnis *directory* zu den Verzeichnissen hinzu, in denen nach Meldungsdateien gesucht werden soll.

## 4.10 Environment-Variablen

Mit folgenden Funktionen können die Werte von Environment-Variablen gelesen werden:

```
int G_getEnvVar ( const char *varId, int defaultValue )
```

Liefert den Wert der Environment-Variable *varId* als *int*-Wert. Existiert diese Variable nicht oder enthält sie keinen *int*-Wert, liefert die Funktion *defaultValue*.

```
unsigned G_getEnvVar ( const char *varId, unsigned defaultValue )
```

Liefert den Wert der Environment-Variable *varId* als *unsigned*-Wert. Existiert diese Variable nicht oder enthält sie keinen *unsigned*-Wert, liefert die Funktion *defaultValue*.

```
void G_getEnvVar ( const char *varId, char *dest, const char  
                  *defaultValue, int maxLength )
```

Liest den Wert der Environment-Variable *varId* als String nach *dest*. Es werden maximal *maxLength-1* Zeichen gelesen und das Resultat mit dem Stringende-Zeichen abgeschlossen. Existiert die Variable nicht, wird *defaultValue* verwendet.

# Anhang A

## Dateiformate

Die Dateien, in denen die Typsysteminformationen bzw. die Graphstruktur abgespeichert werden, sind i.a. so aufgebaut, daß während des Lesens der Datei sofort die internen Datenstrukturen aufgebaut werden können. Deshalb sind z.B. bei den Typinformationen zuerst alle Typen und danach die Subtyprelation gespeichert.

Die Dateien sind als reine Textdateien abgespeichert. Dadurch bekommt man die Möglichkeit, mit relativ einfachen Mitteln die Dateien zu bearbeiten. Leider gibt es auch hierbei Unterschiede zwischen den Betriebssystemen. Deshalb wurde festgelegt, daß der Zeilenumbruch im Unix-Stil vorliegt. Am Ende der Zeile steht ein einfaches *NewLine* ( $10_{10} = A_{16}$ ) — also kein *CarriageReturn-NewLine* wie unter MS-DOS und auch kein *NewLine-CarriageReturn* wie unter MAC-OS.

Den meisten Daten wird eine eindeutige Identifikationsnummer zugewiesen, über die sie später benutzt werden. Diese Nummern werden vom Graphenlabor erzeugt und sind von der jeweiligen Maschinenarchitektur, dem Betriebssystem und den geladenen Typsystemen abhängig. Beim Laden von Graphenlabordateien müssen die Identifikationsnummern an den aktuellen Zustand angepaßt werden. Dieses ist auch bei fest vorgegebenen Daten (wie z.B. der Wertebereich *String*, etc.) erforderlich. In den entsprechenden Funktionen des Graphenlabors ist diese Anpassung enthalten, sodaß Graphenlabordateien beliebig zwischen unterschiedlichen Computer ausgetauscht werden können. Wenn andere Programme Graphenlabor-Dateien lesen oder schreiben, dann müssen die Identifikationsnummern entsprechend behandelt werden.

**Notation der Grammatik** Für die Beschreibung der Dateien wurde eine EBNF-ähnliche Syntax verwendet.

1. Grammatikvariablen sind mit  $\langle Var \rangle$  notiert und werden an einer anderen Stelle im Dokument mit  $\langle Var \rangle ::=$  definiert.
2. Schlüsselwörter und andere Textteile, die genauso in der Datei stehen, sind mit 'Text' notiert.
3. Alternativen sind mit dem senkrechten Strich notiert:  $AltB \mid AltA$ .
4. Ausdrücke können mit Klammern gebunden werden:  $(Ausdruck)$
5. Optionale Teile stehen in eckigen Klammern:  $[Optional]$
6. Teile, die beliebig oft wiederholt werden können (0 oder mehr) stehen in geschweiften Klammern:  $\{Iteration\}$
7. Werte, die aus der Datei gelesen und interpretiert werden, sind als  $\langle Wert \rangle$  notiert. Spezielle Werte sind:

$\langle SP \rangle$	Ein einzelnes Leerzeichen
$\langle NL \rangle$	Ein einzelner Zeilenumbruch (" $\backslash n$ ")

- ⟨Number⟩ Ein natürlichzahliger Wert inklusive 0
- ⟨String⟩ Da in einem String jedes beliebige Zeichen stehen kann, wird eine Pascal-ähnliche Notation benutzt. Zuerst ist die Länge des Strings (als Integer  $\geq 0$ ) gespeichert. Danach folgt als Trennzeichen ein einzelnes Leerzeichen und direkt darauf die angegebene Anzahl an einzelnen Zeichen.

Beispiele:

	String in C-Notation	String in GraLab4-Datei
	"Friedbert"	9_␣Friedbert
	"Friedbert Widmann"	17_␣Friedbert_␣Widmann
	" "	0_␣

## A.1 Typsystemdatei

Da Attribute, und somit die Typen und das Typsystem, auf die Wertebereiche zurückgreifen, müssen zu jedem Typsystem auch die Informationen über die benutzten Wertebereiche abgespeichert werden. Deshalb besteht eine Typsystemdatei aus den benutzten Wertebereichen ( ⟨*UsedDomains*⟩ ), allen im Typsystem definierten Attributen und allen Typen. Die Attributschemata der einzelnen Typen und die Subtyprelation zwischen den Typen wird mit den Typinformationen ( ⟨*TypeDefs*⟩ ) gespeichert.

$$\langle \textit{TypeSysFile} \rangle ::= \langle \textit{UsedDomains} \rangle \langle \underline{\textit{NL}} \rangle \\ \langle \textit{AttrDefs} \rangle \langle \underline{\textit{NL}} \rangle \\ \langle \textit>TypeDefs} \rangle \langle \underline{\textit{NL}} \rangle$$

### A.1.1 Wertebereichsinformationen

Die Wertebereichsinformationen werden durch die Anzahl der aufgeführten Wertebereiche und ein Schlüsselwort eingeleitet. Danach wird jeder Wertebereich in einer separaten Zeile beschrieben. Dabei wird jedem Wertebereich eine Nummer zugeordnet ( ⟨*DomIdx*⟩ ), über die er später adressiert wird.

Jeder Wertebereich wird durch einen Buchstaben eingeleitet. Als weitere Informationen folgen bei den Listen und Multimengen noch die Angabe über den Basiswertebereich, bei Tupeln die Liste der Tupelkomponenten und bei Records die Liste der Selektor-Wertebereich-Paare.

In der Datei tauchen auch die Basiswertebereiche auf. Damit lassen sich die Nummern bestimmen, über die sie später adressiert werden.

$$\langle \textit{UsedDomains} \rangle ::= \langle \underline{\textit{Number}} \rangle \langle \underline{\textit{SP}} \rangle \textit{DOMAINS:} \langle \underline{\textit{NL}} \rangle \\ \{ \langle \textit{DomDef} \rangle \} \\ \langle \textit{DomDef} \rangle ::= \langle \textit{DomIdx} \rangle \langle \underline{\textit{SP}} \rangle \langle \textit{DomDescr} \rangle \langle \underline{\textit{NL}} \rangle \\ \langle \textit{DomIdx} \rangle ::= \langle \underline{\textit{Number}} \rangle \\ \langle \textit{DomDescr} \rangle ::= \textit{'B'}$$

- |  $\textit{'I'}$  – Integer
- |  $\textit{'S'}$  – String
- |  $\textit{'D'}$  – Double
- |  $\textit{'L'}$  ( ' ⟨*DomIdx*⟩ ' ) – List
- |  $\textit{'T'}$  ( ' ⟨*DomIdx*⟩ { ' , ' ⟨*DomIdx*⟩ } ' ) – Tuple
- |  $\textit{'R'}$  ( ' ⟨*RecComp*⟩ { ' , ' ⟨*RecComp*⟩ } ' ) – Record
- |  $\textit{'E'}$  ( ' ⟨*String*⟩ { ' , ' ⟨*String*⟩ } ' ) – Enumeration

$$\langle \textit{RecComp} \rangle ::= \langle \underline{\textit{String}} \rangle \textit{' : ' } \langle \textit{DomIdx} \rangle$$

**Beispiel:** In der folgenden Beispieldatei sind fünf Wertebereiche beschrieben. Dieses sind die Basiswertebereiche *Integer*, *String* und *Double* und die konstruierten Wertebereiche

- 36 : Record aus zwei Komponenten, beide vom Typ *Integer*, mit den Selektorbezeichnungen Anfang und Ende  
 76 : Liste über *Double*.

---

Typsystem: Teil 1

---

```

5 DOMAINS :
0 I
12 S
24 D
36 R(6 Anfang:0,4 Ende:0)
76 L(24)

```

---

### A.1.2 Attributinformationen

Die Attribute werden ebenfalls mit ihrer Anzahl und einem Schlüsselwort eingeleitet. Danach folgen genau <Number> Zeilen mit einzelnen Attributen. Jedem Attribut wird wiederum eine Identifikationsnummer zugewiesen, über die es später adressiert wird. Dann besteht jedes Attribut aus einem Bezeichner und dem Wertebereich, über dem es definiert ist.

$$\begin{aligned} \langle \text{AttrDefs} \rangle & ::= \langle \text{Number} \rangle \langle \text{SP} \rangle \text{'ATTRIBUTEDEFINITIONS:'} \langle \text{NL} \rangle \\ & \quad \{ \langle \text{AttrDef} \rangle \} \\ \langle \text{AttrDef} \rangle & ::= \langle \text{AttrIdx} \rangle \langle \text{SP} \rangle \langle \text{String} \rangle \langle \text{SP} \rangle \langle \text{DomIdx} \rangle \langle \text{NL} \rangle \\ \langle \text{AttrIdx} \rangle & ::= \langle \text{Number} \rangle \end{aligned}$$

**Beispiel:** Mit den Wertebereichen aus dem letzten Beispiel und den folgenden Attributdefinitionen erhält man die Attribute:

- 0: Name : *String*  
 8: Werte : *List of Double*  
 16: Bereich : *Record(Anfang : Integer, Ende : Integer)*

---

Typsystem: Teil 2

---

```

3 ATTRIBUTEDEFINITIONS :
0 4 Name 12
8 5 Werte 76
16 7 Bereich 36

```

---

### A.1.3 Typinformationen

Die Typinformationen beginnen wiederum mit der Anzahl der definierten Typen und einem Schlüsselwort. Danach folgen die einzelnen Typbeschreibungen, die Subtyprelation und die Exportinformationen. Hierbei erscheint auch der Nulltyp des Typsystems.

$$\begin{aligned} \langle \text{TypeDefs} \rangle & ::= \langle \text{Number} \rangle \langle \text{SP} \rangle \text{'TYPES:'} \langle \text{NL} \rangle \\ & \quad \{ \langle \text{TypeDesc} \rangle \} \\ & \quad \{ \langle \text{TypeIsA} \rangle \} \\ & \quad \{ \langle \text{TypeExport} \rangle \} \end{aligned}$$

Jedem Typ wird eine Identifikationsnummer zugewiesen, über die er später adressiert wird. An dieser Stelle wird der Typbezeichner und sein Attributschema (als Aufzählung der Attribute) definiert. Wenn dem Typ keine Attribute zugeordnet sind, dann entfallen auch die Klammern um die Attributliste. Attribute, die ein Typ an seine Untertypen vererbt hat, erscheinen in den Attributlisten aller Untertypen.

$$\begin{aligned} \langle \text{TypeDesc} \rangle & ::= \langle \text{TypeIdx} \rangle \langle \text{SP} \rangle \langle \text{String} \rangle \langle \text{SP} \rangle \\ & \quad [ \text{'('} \langle \text{AttrIdx} \rangle \{ \text{' , ' } \langle \text{AttrIdx} \rangle \} \text{' )' } \\ & \quad \text{' . ' } \langle \text{NL} \rangle \\ \langle \text{TypeIdx} \rangle & ::= \langle \text{Number} \rangle \end{aligned}$$

Die Subtyprelation besteht jeweils aus einer Typnummer, dem Schlüsselwort `isa` und einer Liste der Obertypen. Da die Relation reflexiv ist, hat jeder Typ mindestens sich selbst als Obertypen.

$$\begin{aligned} \langle \text{TypeIsA} \rangle & ::= \langle \text{TypeIdx} \rangle \langle \text{SP} \rangle \\ & \quad \text{' isa ' } \langle \text{SP} \rangle \langle \text{TypeIdx} \rangle \{ \text{' , ' } \langle \text{TypeIdx} \rangle \} \text{' . ' } \langle \text{NL} \rangle \end{aligned}$$

Abschließend wird für jeden Typ beschrieben, ob er schon exportiert wurde oder ob er nach dem Laden des Typsystems noch verändert werden kann.

$$\begin{aligned} \langle \text{TypeExport} \rangle & ::= \langle \text{TypeIdx} \rangle \langle \text{SP} \rangle \\ & \quad \text{' isExported ' } \langle \text{SP} \rangle ( \text{' 0 ' } | \text{' 1 ' } ) \langle \text{NL} \rangle \end{aligned}$$

**Beispiel:** Im folgenden Beispiel werden vier Typen definiert.

---

Typsystem: Teil 3

---

```
5 TYPES:
0 8 TypeNull .
24 6 Person (0).
48 8 Lotterie (0,8).
72 10 hatGesetzt (16).
96 11 Angestellte (0).
0 isa 0.
24 isa 0,24.
48 isa 0,48.
72 isa 0,72.
96 isa 0,24,96.
0 isExported 1
24 isExported 1
48 isExported 1
72 isExported 1
96 isExported 0
```

---

## A.2 Graphdatei

Da das Typsystem, mit dem eine Graphinstanz beim Speichern einer Datei verbunden war, nicht unbedingt jenes ist, mit dem eine andere Graphinstanz diese Datei wieder liest, muß sichergestellt werden, daß diese Typsysteme zueinander passen (siehe hierzu Abschnitt 4.1.8, S. 71). Damit dieser Test vor

dem Laden der Graphstruktur möglich ist, wird in jeder Graphdatei auch die komplette Typsysteminformation (siehe Anhang A.1, S. 131) abgespeichert.

$$\langle \text{GraphFile} \rangle ::= \langle \text{TypeSysFile} \rangle \\ \langle \text{GraphStructure} \rangle$$

### A.2.1 Graphstruktur

Die Graphstruktur wird mit dem Schlüsselwort `GRAPH:` eingeleitet. Dabei wird die Größe der internen Tabellen beim Speichern der Datei angegeben (`NMax`: Tabellengröße für Knoten, `MMax`: für Kanten). Mit `NCnt` und `MCnt` wird die Anzahl der existierenden Knoten und Kanten angegeben. Nach einer Leerzeile folgen die Beschreibungen der `NCnt` Knoten und der `MCnt` Kanten.

Die Reihenfolge der Knoten und Kanten entspricht der globalen Kanten- und Knotensequenz (und muß beibehalten werden).

$$\langle \text{GraphStructure} \rangle ::= \text{'GRAPH: (' } \langle \text{NMax} \rangle \langle \text{SP} \rangle \langle \text{MMax} \rangle \langle \text{SP} \rangle \\ \langle \text{NCnt} \rangle \langle \text{SP} \rangle \langle \text{MCnt} \rangle \text{' )' } \langle \text{NL} \rangle \\ \langle \text{NL} \rangle \\ \{ \langle \text{VertexDesc} \rangle \} \langle \text{NL} \rangle \\ \{ \langle \text{EdgeDesc} \rangle \} \langle \text{NL} \rangle \\ \langle \text{NMax} \rangle ::= \langle \text{Number} \rangle \\ \langle \text{MMax} \rangle ::= \langle \text{Number} \rangle \\ \langle \text{NCnt} \rangle ::= \langle \text{Number} \rangle \\ \langle \text{MCnt} \rangle ::= \langle \text{Number} \rangle$$

### A.2.2 Knotenbeschreibung

Jedem Knoten wird eine Identifikationsnummer zugewiesen. Über diese Nummer wird er auch in den internen Tabellen adressiert. Beschrieben wird ein Knoten durch seinen Typ, die Liste der inzidenten Kanten und seine Attributwerte.

$$\langle \text{VertexDesc} \rangle ::= \langle \text{VIdx} \rangle \langle \text{SP} \rangle \langle \text{TypeIdx} \rangle \langle \text{SP} \rangle \langle \text{IncidentEdges} \rangle \langle \text{NL} \rangle \\ \langle \text{AttrValues} \rangle \langle \text{NL} \rangle \\ \langle \text{VIdx} \rangle ::= \langle \text{Number} \rangle$$

In der Inzidenzliste sind alle Kanten aufgeführt, die mit dem Knoten verbunden sind. Dabei haben ausgehende Kanten kein Vorzeichen, eingehende Kanten das Vorzeichen `-`. Die Reihenfolge der Kanten gibt die Inzidenzsequenz des geordneten Graphen wieder (und darf nicht geändert werden). Isolierte Knoten mit einer leeren Inzidenzliste `( )` gespeichert.<sup>1</sup>

$$\langle \text{IncidentEdges} \rangle ::= \text{' (' } [ \langle \text{DirectedEdge} \rangle \{ \text{' , ' } \langle \text{DirectedEdge} \rangle \} ] \text{' )' } \\ \langle \text{DirectedEdge} \rangle ::= \langle \text{EIdx} \rangle \\ | \text{' - ' } \langle \text{EIdx} \rangle$$

### A.2.3 Kantenbeschreibung

Jeder Kante wird eine Identifikationsnummer zugewiesen. Über diese Nummer wird sie auch in den internen Tabellen adressiert. Beschrieben wird eine Kante durch ihren Typ und ihre Attributwerte.

<sup>1</sup> Durch einen Fehler in einer früheren Graphenlaborversion existieren aber auch Graphdateien, bei denen isolierte Knoten mit `(0)` gespeichert sind.

Die Anfangs- und Endknoten der Kanten sind hier nicht aufgeführt, da sie schon durch die Inzidenzlisten der Knoten bestimmt sind.

$$\begin{aligned} \langle \text{EdgeDesc} \rangle & ::= \langle \text{EIdx} \rangle \langle \text{SP} \rangle \langle \text{TypeIdx} \rangle \langle \text{NL} \rangle \\ & \quad \langle \text{AttrValues} \rangle \langle \text{NL} \rangle \\ \langle \text{EIdx} \rangle & ::= \langle \text{Number} \rangle \end{aligned}$$

#### A.2.4 Attributwerte

Da sich das Attributierungsschema eines Typs zwischen dem Speichern und Laden geändert haben kann, können die Attribute nicht anhand der aktuellen Typdaten geladen werden. Deshalb wird zu jedem Graphenelement die Anzahl der Attributwerte und danach genau  $\langle \text{Number} \rangle$  Zeilen mit einzelnen Attributwerten gespeichert.

$$\langle \text{AttrValues} \rangle ::= \langle \text{Number} \rangle \langle \text{NL} \rangle \{ \langle \text{AttrValue} \rangle \}$$

Ein einzelner Attributwert wird durch die Identifikationsnummer des Attributes bestimmt. Darüber ist auch das Format festgelegt, mit dem der Wert gespeichert ist.

$$\langle \text{AttrValue} \rangle ::= \langle \text{AttrIdx} \rangle \text{ ' : ' } \langle \text{SP} \rangle \langle \text{Value} \rangle \langle \text{NL} \rangle$$

Die Basiswerte *Integer* und *Double* werden im Standardformat von C/C++ gespeichert. Für *Strings* wird das auf Seite 131 beschriebene Format benutzt. Bei Werten vom Typ *List* wird zuerst die Anzahl der Elemente und dann eine Liste der Werte gespeichert, deren Format dem Wertebereich entspricht, über dem die Liste oder Multimenge definiert ist. Bei *Tuple*- und *Record*-Wertebereichen steht die Anzahl und der Wertebereich der einzelnen Komponenten durch die Attributdefinition fest.

$\langle \text{Value} \rangle ::=$	<code>'false'   'true'</code>	– Boolean
	$\langle \text{Integer} \rangle$	– Integer
	$\langle \text{Double} \rangle$	– Double
	$\langle \text{String} \rangle$	– String
	$\langle \text{Number} \rangle \langle \text{SP} \rangle$	
	<code>'&lt; ' [ <math>\langle \text{Value} \rangle</math> { ' , ' <math>\langle \text{Value} \rangle</math> } ] '&gt;'</code>	– List
	<code>'( ' [ <math>\langle \text{Value} \rangle</math> { ' , ' <math>\langle \text{Value} \rangle</math> } ] ')'</code>	– Tuple, Record
	$\langle \text{Integer} \rangle$	– Enumeration

### A.2.5 Ein komplettes Beispiel

Wenn mit dem Typsystem (Graphschema) in Abbildung A.1 der Graph aus Abbildung A.2 erzeugt und abgespeichert wird, dann erhält man die in Abbildung A.3 dargestellte Datei.

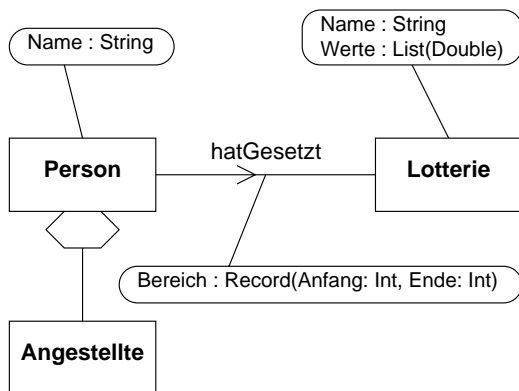


Abbildung A.1: Graphschema

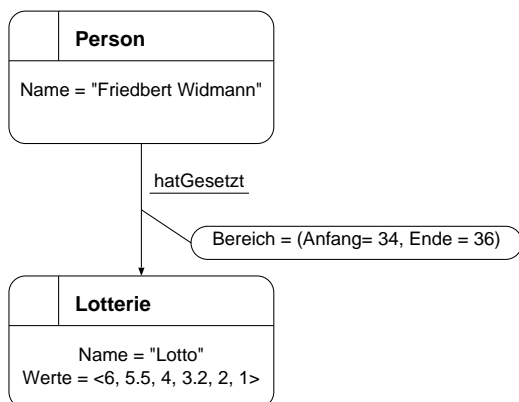


Abbildung A.2: Graphinstanz

```

5 DOMAINS:
0 I
12 S
24 D
36 R(6 Anfang:0,4 Ende:0)
76 L(24)
  
```

```

3 ATTRIBUTEDEFINITIONS:
0 4 Name 12
8 5 Werte 76
16 7 Bereich 36
  
```

```

5 TYPES:
0 8 TypeNull .
24 6 Person (0).
48 8 Lotterie (0,8).
72 10 hatGesetzt (16).
96 11 Angestellte (0).
0 isa 0.
24 isa 0,24.
48 isa 0,48.
72 isa 0,72.
96 isa 0,24,96.
0 isExported 1
24 isExported 1
48 isExported 1
72 isExported 1
96 isExported 0
  
```

```

GRAPH:(1000 1000 2 1)
  
```

```

3 24 (2)
1
0: 17 Friedbert Widmann
2 48 (-2)
2
0: 5 Lotto
8: 6 <6,5.5,4,3.2,2,1>

2 72
1
16: (34,36)
  
```

Abbildung A.3: Die Graphdatei bsp.g

## A.3 Meldungsdateien

Meldungstexte zerfallen in zwei Teile, nämlich die einleitenden Texte, die bei allen Meldungen gleich sind (Meldungsköpfe), und die für jede Meldung individuellen Texte.

### A.3.1 Meldungsköpfe

In einer Datei des folgenden Formats sind die Texte angegeben, mit denen Fehlermeldungen eingeleitet werden und die einzelnen Teile einer Meldung voneinander abgegrenzt werden. Diese Datei wird mit der Methode `G_msg::setHeaders()` gelesen.

```

<msgHeads> ::= <MsgStr> <NL>
              <ErrStr> <NL>
              <FErStr> <NL>
              <InfStr> <NL>
              <ConFndStr> <NL>
              <ActTakStr> <NL>
              <ReaProStr> <NL>

<MsgStr>    ::= <STR>
<ErrStr>    ::= <STR>
<FErStr>    ::= <STR>
<InfStr>    ::= <STR>
<ConFndStr> ::= <STR>
<ActTakStr> ::= <STR>
<ReaProStr> ::= <STR>
<STR>       ::= { <PCHAR> }
<PCHAR>     ::= any printable character but "\n"

```

Alle Strings dürfen maximal 100 Zeichen lang sein. Sie haben folgende Bedeutung:

- <**MsgStr**> Anfang einer Warnmeldung
- <**ErrStr**> Anfang einer Fehlermeldung
- <**FErStr**> Anfang der Meldung eines fatalen Fehlers
- <**InfStr**> Formatstring gemäß `printf`, der Art der Ausgabe von Gruppenname, Meldungsnummer und Funktionsname. Dieser String muß also "%s", "%d" und "%s" in dieser Reihenfolge enthalten.
- <**ConFnd**> Anfang der Beschreibung der Fehlerbedingung
- <**ActTak**> Anfang der Beschreibung des Verhaltens des Graphenlabors (bzw. Anwendungsprogramms bei eigenen Fehlermeldungen)
- <**ReaPro**> Anfang des Vorschlags zur Fehlerbeseitigung

### Beispiel

Das Graphenlabor verwendet standardmäßig die Texte aus der Datei `msgems`:

```

***** MESSAGE
***** ERROR
***** FATAL ERROR

```

```

    %s%03d in %s
    ***** Condition found   :
    ***** Action taken     :
    ***** Reaction proposed:

```

### A.3.2 Meldungstexte

Für jede *Gruppe* von Programmtexten existiert eine Meldungsdatei mit dem Namen `msg<group>`. Diese Dateien werden bei jeder Meldung durch die Methode `G_msg::msg()` gelesen.

```

⟨messages⟩ ::= { ⟨message⟩ }
⟨message⟩  ::= ⟨msgHead⟩ [ ⟨conFnd⟩ ] [ ⟨actTak⟩ ] [ ⟨reaPro⟩ ]  ⟨NL⟩
⟨msgHead⟩  ::= `MSG' ⟨DIGIT⟩ ⟨DIGIT⟩ ⟨DIGIT⟩ ⟨NL⟩
⟨conFnd⟩   ::= `C' { ' ⟨STRING⟩ ' }' ⟨NL⟩
⟨actTak⟩   ::= `A' { ' ⟨STRING⟩ ' }' ⟨NL⟩
⟨reaPro⟩   ::= `R' { ' ⟨STRING⟩ ' }' ⟨NL⟩
⟨STRING⟩   ::= { ⟨CHAR⟩ }
⟨CHAR⟩     ::= any character but " } "

```

Zur Ausgabe von zusätzlichen Parametern müssen die Strings in `⟨conFnd⟩`, `⟨actTak⟩` und `⟨reaPro⟩` zusammen die erforderlichen Umwandlungsspezifizierer („%d“ etc.) enthalten, deren Anzahl und Reihenfolge vom Programm für jede Meldung fest vorgegeben ist.

Für die Fehlermeldung im Beispiel aus Abschnitt 3.4, S. 50, können folgende Zeilen in der Datei `msggrf` stehen:

```

MSG007
C {`v%u' is an unused vertex in graph `x'}
A {function aborted}
R {call function with an existing vertex}

```

Innerhalb der Methode `createEdge()` wird die Ausgabe mit dem Aufruf von `G_msg::msg()` erzeugt:

```

G_error.msg(7, "grf", "G_graph::createEdge(%s,v%u,v%u)",
            getTypeSystem().getTypeId(t).str(), /* 4 */
            getVNo(v),                          /* 5 */
            getVNo(w),                          /* 6 */
            getVNo(w),                          /* 7 */
            (void*)this                          /* 8 */
            );

```

Die ersten drei Parameter sind für `msg()` zwingend. Sie bezeichnen die Meldungsnummer, die Gruppe und die Signatur der Funktion, in der die Meldung ausgegeben wird. Mit den folgenden Parametern werden zuerst die Umwandlungsspezifizierer im dritten Parameter (`fun`) von `msg()` gefüllt. In diesem Beispiel also drei Parameter. Die restlichen Parameter füllen die Umwandlungsspezifizierer in den Meldungstexten aus der Meldungsdatei. Als Ergebnis bekommt man folgende Ausgabe.

```

***** ERROR grf007 in G_graph::createEdge(TypeNull,v1,v2)
***** Condition found   : `v2' is an unused vertex in graph `bffff500'
***** Action taken     : function aborted
***** Reaction proposed: call function with an existing vertex

```

## Anhang B

# Liste der Environment-Variablen

Folgende Environment-Variablen beeinflussen das Verhalten des Graphenlabors. Falls sie nicht gesetzt sind, werden Defaultwerte aus `src/include/g_config.h` verwendet.

- G\_NMAX** Der `int`-Wert gibt die Anzahl der Knoten an, die ein Graph unmittelbar nach der Erzeugung aufnehmen kann. Der Wert kann durch die explizite Angabe dieser Größe im Konstruktoraufruf `G_graph::G_graph()` ignoriert werden. Werden mehr Knoten erzeugt, wird für den jeweiligen Graphen die Anzahl verdoppelt.
- G\_MMAX** Der `int`-Wert gibt die Anzahl der Kanten an, die ein Graph unmittelbar nach der Erzeugung aufnehmen kann. Der Wert kann durch die explizite Angabe dieser Größe im Konstruktoraufruf `G_graph::G_graph()` ignoriert werden. Werden mehr Kanten erzeugt, wird für den jeweiligen Graphen die Anzahl verdoppelt.
- G\_MAXERROR** Der `int`-Wert gibt an, nach wievielen Fehlern das Graphenlabor das Programm abbricht. Wenn dieser Wert auf 0 gesetzt wird, wird das Programm nie abgebrochen.
- G\_MSGPATH** Der String gibt an, in welchen Verzeichnissen nach Meldungsdateien gesucht wird. Mit der Methode `G_msg::addDirectory()` können zusätzliche Verzeichnisse angegeben werden.

# Anhang C

## Unterstützte Systemumgebungen

Zur Zeit werden die in Tabelle C.1 aufgeführten Architekturen unterstützt. Unter diesen Architekturen wurden Testprogramme erzeugt und erfolgreich abgearbeitet. Die dabei benutzten Header-Dateien und Bibliotheken befinden sich in den Verzeichnissen<sup>1</sup>

`/home/ems/GraLab4/lib/⟨ArchitekturVerzeichnis⟩.`

Im selben Verzeichnis sind weitere Informationsdateien, die auf Besonderheiten der Architekturen eingehen.<sup>2</sup>

In diesen Verzeichnissen befinden sich auch Archive mit fertigen Bibliotheken. Diese enthalten alle Dateien, die für die Benutzung des Graphenlabors nötig sind und sind für alle Bibliotheksvarianten (siehe Anhang D.1, S. 142) vorhanden.

Für die Graphenlaborbibliothek muß ein Verzeichnis angelegt werden, unter dem alle Dateien installiert werden. Dieses Verzeichnis ist das **GraLab**-Wurzelverzeichnis und wird über die Make-Variable `$(GRALAB_HOME)` angesprochen. Wenn die Archivdatei in diesem Verzeichnis entpackt wird, werden die Dateien in die richtigen Unterverzeichnisse verteilt.

Im Verzeichnis `$(GRALAB_HOME)/example` befindet sich ein kleines Beispielprogramm. Dafür muß im `makefile` nur die Architektur und die Bibliotheksvariante angepaßt werden. Dann kann mit

<sup>1</sup> Die Verzeichnisangabe bezieht sich auf die Installation an der Universität in Koblenz. Für andere Installationen muß `/home/ems/GraLab4` entsprechend ersetzt werden.

<sup>2</sup> Auf dem ftp-Server: `ftp://ftphost.uni-koblenz.de/outgoing/GraLab/GraLab4/lib/`

Prozessor	Betriebssystem	Compiler	Verzeichnisname
SPARC	Solaris2.5	Sun WorkShop	<code>sparc-sun-solaris2.5/SUNWspro</code>
SPARC	Solaris2.5	GNU g++	<code>sparc-sun-solaris2.5/GNUgcc</code>
i386	Solaris2.6	Sun WorkShop	<code>i386-unknown-solaris2.6/SUNWspro</code>
i386	Solaris2.6	GNU g++	<code>i386-unknown-solaris2.6/GNUgcc</code>
SPARC	SunOS 4.1.3	GNU g++	<code>sparc-sun-sunos4.1.3/GNUgcc</code>
i386	OS/2	IBM icc	<code>i386-os2/ibmicc</code>
i486	Linux	GNU g++	<code>i486-unknown-linux/GNUgcc</code>
i386	MS Windows <sup>†</sup>	Visual C++ 4.0	<code>i386-win/ms-vc40</code>
i386	MS Windows <sup>†</sup>	Visual C++ 5.0	<code>i386-win/ms-vc50</code>

<sup>†</sup> Nur die 32-Bit-Versionen — getestet auf Microsoft WindowsNT 4.0

Tabelle C.1: Unterstützte Architekturen

```
make GRALAB_HOME=(lokales GRALAB-Verzeichnis)
```

das Beispiel übersetzt werden.

## C.1 Hinweise für spezielle Architekturen

Bei der Entwicklung des Graphenlabors wurde der Schwerpunkt auf unixähnliche Systeme gelegt. Bei Systemen, die zu weit von den Unix-Standards abweichen, gelten besondere Regeln. Diese sind normalerweise in Textdateien in den Verzeichnissen beschrieben.

### C.1.1 OS/2 mit IBM-ICC (i386-os2/ibm-icc)

Unter diesem System haben die Objektdateien die Erweiterung `.obj` und die ausführbaren Programme `.exe`. Außerdem muß dem Linker der komplette Pfad der Bibliotheksdatei übergeben werden. Deshalb sehen die entsprechenden Zeilen im `makefile` wie folgt aus:

```
prog.exe : prog.obj
    $(GRALAB_LINK_o) /Feprog.exe prog.obj \
    $(GRALAB_LIBDIR)\$(GRALAB_LIBFILE)
```

### C.1.2 MS-Windows mit MS-VC (i386-win/ms-vc40 und i386-win/ms-vc50)

Hierbei wird meistens mit der Entwicklungsumgebung von Microsoft gearbeitet. Deshalb werden die Abhängigkeiten zwischen den einzelnen Dateien als Projekte und nicht in `makefiles` definiert. Die Einbindung des Graphenlabors muß hier *von Hand* erfolgen und in die Projektdefinition der Entwicklungsumgebung eingetragen werden. Eine Anleitung hierzu findet man in der Datei `msdev.txt` im Architekturverzeichnis.

Alternativ hierzu kann auch mit normalen Makefiles gearbeitet werden.

### C.1.3 Intel i86pc unter Solaris2.6 mit SUN-Compiler (i386-unknown-solaris2.6/SUNWsprow)

Im Optimierer des Compilers haben wir einen Bug gefunden, wodurch an manchen Stellen falscher Code erzeugt wird. Deshalb wird die Release-Variante nur mit `-O3` optimiert.

### C.1.4 Intel i86pc unter Solaris2.6 mit GNU-Compiler (i386-unknown-solaris2.6/GNUgcc)

Im Netzwerk der Universität in Koblenz wird dieser Compiler nicht vollständig unterstützt. Deshalb kann diese Konstellation nur eingeschränkt zum Debuggen benutzt werden und unterstützt kein Profiling.

# Anhang D

## Übersetzen und Binden

### D.1 Bibliotheksvarianten

Vom Graphenlabor existieren unterschiedliche Varianten, die mit optionaler Funktionalität ausgestattet sind. Diese Varianten werden durch einen Buchstabenschlüssel identifiziert.

**Check (c):** Beim Aufruf von Graphenlabor-Funktionen werden die Parameter überprüft. Wenn dabei festgestellt wird, daß die übergebenen Parameter nicht zum Graphen passen (z.B. Knotenvariable referiert auf einen Knoten, der inzwischen gelöscht ist), dann wird eine Meldung ausgegeben und eine entsprechende Reaktion ausgeführt.

Wenn diese Option **nicht** gewählt wird, dann muß das Makro `NO_CHK` definiert werden. Dieses wird von den Makefile-Templates in die Make-Variable `$(GRALAB_CPPFLAGS)` eingetragen.

**Debug (d):** Während der Entwicklung von Programmen kann man zur Fehlersuche einen Debugger einsetzen. Dazu muß der Compiler beim Übersetzen der Quelltexte zusätzliche Debug-Informationen erzeugen.

Mit dieser Option werden Debug-Informationen erzeugt. Außerdem werden Codeteile aktiviert, die Bibliotheksoperationen protokollieren. Hierfür wird das Makro `G_DEBUG` gesetzt. Die nötigen Programmparameter werden von den Makefile-Templates in die Make-Variablen `$(GRALAB_CFLAGS)` und `$(GRALAB_CPPFLAGS)` eingetragen.

**Profile (p):** Während dem Ablauf eines Programmes werden Daten über das Zeitverhalten gesammelt und protokolliert. Diese können später mit einem *Profiler* ausgewertet werden.

In dieser Bibliotheksvariante sind Steuerinformationen für die Profiler enthalten. Die nötigen Programmparameter werden von den entsprechenden Makefile-Templates in die Make-Variablen `$(GRALAB_CFLAGS)` und `$(GRALAB_LDFLAGS)` eingetragen.

**Trace (t):** Beim Tracing werden die Funktionsaufrufe von auswählbaren Teilen des Graphenlabors protokolliert.

Wenn diese Option **nicht** gewählt wird, dann muß das Makro `NO_TRC` gesetzt werden. Dieses wird von den Makefile-Templates in die Make-Variable `$(GRALAB_CPPFLAGS)` eingetragen.

**Undo (u):** Die Änderungen an Graphen und anderen Datenstrukturen können in einem Undo-Puffer protokolliert und zurückgenommen werden.

Wenn diese Option **nicht** gewählt wird, dann muß das Makro `NO_UNDO` gesetzt werden. Dieses wird von den Makefile-Templates in die Make-Variable `$(GRALAB_CPPFLAGS)` eingetragen.

Diese Optionen können miteinander verknüpft werden. Allerdings sind nicht alle Kombinationen sinnvoll. Z.B. macht es keinen Sinn, die Laufzeit eines Programmes zu analysieren, welches Tracing-Ausgaben erzeugt. Deshalb gibt es keine Variante mit p- und t-Funktion.

Bei Kombinationen müssen die Buchstaben in alphabetischer Reihenfolge zusammengefügt werden und bilden dann einen Teil der Dateinamen für die *Makefile-Templates* (siehe Abschnitt 3.2.3, S. 49).  
(make<opt>.tpl)

### D.1.1 Verfügbare Varianten

Zur Zeit sind folgende Varianten verfügbar:

**cdtu:** Eine Entwicklungsvariante mit Informationen für die Debugger und Check- und Tracecode.

**dpu:** Eine Analysevariante mit Informationen für Debugger und Profiler.<sup>1</sup>

**pu:** Eine optimierte Analysevariante mit Informationen für Profiler.

**u:** Eine Releasevariante die optimierten Code enthält.

## D.2 Präprozessor-Makros

Beim Übersetzen des Graphenlabors muß es an einigen Stellen auf die Maschine bzw. die Variante angepaßt werden. Diese Abhängigkeiten werden mit Präprozessor-Makros realisiert. Dabei kann man grundsätzlich zwischen Abhängigkeiten von der gewünschten Bibliotheksvariante und den Abhängigkeiten von der Maschine, dem Betriebssystem und des Compilers unterscheiden.

### D.2.1 Auswahl der Varianten

Die folgenden Makros werden beim Aufruf des Compilers angegeben. Sie werden in den Makefile-Templates in die Variable  $\$(GRALAB_CPPFLAGS)$  eingetragen.

**NO\_CHK** Das Graphenlabor führt keine Plausibilitätskontrolle der Aufrufparameter durch (siehe Abschnitt 3.6, S. 53).

**NO\_TRC** Das Graphenlabor erzeugt kein Protokoll der Funktionsaufrufe (siehe Abschnitt 3.5, S. 52).

**NO\_UNDO** Es wird eine Variante erzeugt, die keinen Undo-Puffer (siehe Abschnitt 1.1.7, S. 15) unterstützt. Dadurch sind einige Funktionen des Graphenlabors schneller.

**G\_DEBUG** Es werden Programmteile eingebaut, die das Verhalten von manchen Bibliotheksfunktionen protokollieren.

### D.2.2 Anpassung an Betriebssystem und Compiler

Die Quelltexte des Graphenlabors hängen an einigen Stellen vom benutzten Betriebssystem und dem Compiler ab. Hierfür werden in den Headerdateien `g_sysdep.h` und `g_align.h` Präprozessor-Makros definiert und in den Quelltexten für bedingte Compilierung benutzt. Da diese Einstellungen für die verschiedenen Architekturen unterschiedlich sind, befinden sich die Headerdateien in den Verzeichnissen mit architekturabhängigen Dateien ( $\$(GRALAB_HOME)/lib/\$(GRALAB_ARCH)$ ).

Normalerweise werden diese Dateien beim Compilieren der Bibliothek automatisch erzeugt. Hierzu werden die Programme `g_sysdep` und `g_align` benutzt. Man sollte diese Dateien nur dann ändern, wenn diese Programme nicht erfolgreich waren.

---

<sup>1</sup> Diese Variante sollte nicht mehr benutzt werden, da die Debug-Informationen die Laufzeiteigenschaften beeinflussen können.

In `g_sysdep.h` sind allgemeine Abhängigkeiten vom System beschrieben.

**G\_UINTSIZE** Gibt an, wieviele Bit eine Variable vom Typ `unsigned int` hat. Diese Größe wird für die Bitstrings der `isA`-Relation im Typsystem benutzt.

**G\_KNOWS\_BOOL** Wird definiert, wenn der Compiler den `bool`-Typ schon eingebaut hat. Da der C++-Typ `bool` erst im ANSI-Standard vom Herbst '97 definiert ist, wird er bei älteren Compiler von einer Klasse `bool` emuliert.

**G\_KNOWS\_NEWVECTOR** Manche Compiler benutzen unterschiedliche `new`-Operatoren für einzelne Instanzen und Arrays von Instanzen. Bei Compiler, die das unterstützen muß dieses Makro definiert werden.

**G\_INIT\_REF\_BUG** Manche Compiler initialisieren globale Referenzen nicht statisch zur Compilezeit sondern dynamisch zur Laufzeit des Programmes. Dieses Makro wird definiert um diesen Fehler zu umgehen.

**G\_KNOWS\_EXPLICIT**

**G\_EXPLICIT** Bestimmt, ob der Compiler das Schlüsselwort `explicit` kennt. Wenn dem so ist, dann wird das Makro `G_KNOWS_EXPLICIT` auf 1 gesetzt und `G_EXPLICIT` expandiert zu `explicit`. Ansonsten ist `G_KNOWS_EXPLICIT` undefiniert und `G_EXPLICIT` expandiert zum leeren String.

**G\_HAS\_UNISTD\_H** Wird definiert, wenn die Headerdatei `unistd.h` vorhanden ist.

**G\_HAS\_IO\_H** Wird definiert, wenn die Headerdatei `io.h` vorhanden ist.

**G\_HAS\_STRSTREA\_H** Wird definiert, wenn die Headerdatei `strstrea.h` vorhanden ist. Diese Datei gibt es auf manchen Systemen, deren Dateinamen nur acht Zeichen lang sein darf.

**G\_HAS\_STRSTREAM\_H** Wird definiert, wenn die Headerdatei `strstream.h` vorhanden ist.

**G\_HAS\_STROTUL** Wird definiert, wenn die C-Funktion `strtoul()` in der Laufzeitbibliothek des Compilers enthalten ist. Ansonsten wird die Umwandlung mit der Funktion `sscanf()` durchgeführt.

**G\_FREE\_RETURNS\_INT** Wird definiert, wenn die Funktion `free()` einen `int`-Wert zurückliefert. Normalerweise hat `free()` keinen Rückgabewert (`void`).

**G\_FREE\_HAS\_CHARARG** Wird definiert, wenn die Funktion `free()` einen `char*`- anstatt einem `void*`-Zeiger erwartet.

**G\_PAGESIZE** Gibt an, wie groß eine Seite der virtuellen Speicherverwaltung des Betriebssystems ist. Wenn diese Größe nicht automatisch ermittelt werden kann, wird eine Seitengröße von `4 kByte` angenommen.

**G\_HAS\_SC\_PAGESIZE** Wird definiert, wenn die Seitengröße mit `sysconf(_SC_PAGESIZE)` festgestellt wird.

**G\_HAS\_GETPAGESIZE** Wird definiert, wenn die Seitengröße mit `getpagesize()` ermittelt wird.

**G\_IOS\_BIN** Enthält den Wert, der benutzt werden muß, um einen Stream im Binär-Modus zu öffnen. Dieses Makro kann beim Öffnen von Dateien benutzt werden:

```
istream is("input.g", ios::in|G_IOS_BIN);
```

**G\_HAS\_IOS\_BINARY** Wird definiert, wenn der Binär-Modus mit `ios::binary` aktiviert wird.

**G\_HAS\_IOS\_BIN** Wird definiert, wenn der Binär-Modus mit `ios::bin` aktiviert wird.

**G\_ALIGN\_CHAR, G\_ALIGN\_INT, ...** Die Attributwerte werden maschinennah in Speicherblöcke abgespeichert. Hierfür müssen die Adressen bekannt sein, an denen das System die Daten verwalten kann. In der Datei `g_align.h` sind diese Adressen für alle im Graphenlabor verwendeten Datentypen enthalten.

## D.3 Make-Variablen

In den Makefile-Template-Dateien werden folgende Variablen gesetzt:

Tabelle D.1: Make-Variablen

Variable	Bedeutung	Wert bei GNU g++ unter Solaris 2.5
GRALAB_ARCH GRALAB_LIBTYPE	Rechnerarchitektur Bibliotheksvariante (siehe Abs. D.1)	sparc-sun-solaris2.5/GNUgcc z.B cdtu
Verzeichnisse des Graphenlabor		
GRALAB_HOME GRALAB_SRCDIR GRALAB_INCDIR GRALAB_LIBDIR	Wurzelverzeichnis des Graphenlabors Verzeichnis der Quelltexte Verzeichnis der Header-Dateien Verzeichnis der architekturabhängigen Dateien	/home/ems/GraLab4 \$(GRALAB_HOME)/src \$(GRALAB_SRCDIR)/include \$(GRALAB_HOME)/lib/\$(GRALAB_ARCH)
Programme		
GRALAB_CC GRALAB_AR GRALAB_CP GRALAB_RM GRALAB_CAT GRALAB_RANLIB GRALAB_INSTDIR GRALAB_INSTLIB	Compiler Bibliotheksmanager Programm zum Kopieren Programm zum Löschen Auflisten von Dateien Archivverzeichnis Erzeugt ein Verzeichnis Installiert Dateien im Zielverzeichnis	g++ -v2.7.1 ar cp rm -f cat ranlib install -m 777 -d install -m 644
Aufrufoptionen für Programme		
GRALAB_CPPFLAGS GRALAB_CFLAGS GRALAB_LDFLAGS GRALAB_LDLIBS GRALAB_ARFLAGS	C-Preprozessor C++-Compiler Linker Bibliotheken zum Linken Bibliotheksmanager	-I\$(GRALAB_INCDIR) -I\$(GRALAB_LIBDIR) -Wall -fno-inline -g -L\$(GRALAB_LIBDIR) rv
Dateinamen-Suffixe und -Prefixe		
GRALAB_C GRALAB_O GRALAB_E GRALAB_A GRALAB_DEP GRALAB_APRE	C++-Dateien compilierte Objekt-Dateien ausführbare Programme Funktionsbibliotheken Dateiabhängigkeiten Prefix von Bibliotheken	.c .o .a .dep lib
Systemabhängige Trennzeichen		
GRALAB_PATHSEP GRALAB_DIRSEP	für Verzeichnislisten zwischen Verzeichnissen in Dateinamen	: /
Bezeichnung der Bibliothek		
GRALAB_LIBBASE GRALAB_LIB	Basisname der GraLab-Bibliothek Name der gewählten Bibliothek	graph Bsp.: graph\$(GRALAB_LIBTYPE) = graphcdtu

Tabelle D.1: ... Make-Variablen

Variable	Bedeutung	Wert bei GNU g++ unter Solaris 2.5
GRALAB_LIBFILE	kompletter Dateiname der GraLab-Bibliothek	<code>\$(GRALAB_APRE) \$(GRALAB_LIB)\$(GRALAB_A) = libgraphcdtu.a</code>
Befehle für Make-Regeln		
GRALAB_COMPILE_C	Compiler-Aufruf	<code>\$(GRALAB_CC) \$(GRALAB_CFLAGS) \$(GRALAB_CPPFLAGS) -c</code>
GRALAB_LINK_O	Linker-Aufruf	<code>\$(GRALAB_CC) \$(GRALAB_LDFLAGS)</code>
GRALAB_LINK_C	Compilieren und Linken	<code>\$(GRALAB_LINK_O) \$(GRALAB_CFLAGS) \$(GRALAB_CPPFLAGS)</code>
GRALAB_DEPEND_C	Erzeugen der Dateihängigkeiten	<code>\$(GRALAB_CC) \$(GRALAB_CFLAGS) \$(GRALAB_CPPFLAGS) -M</code>

## D.4 Übersetzen des Graphenlabors

Die Graphenlabor-Bibliothek kann auch auf uni-fremden Rechnern installiert werden. Neben den fertig compilierten Bibliotheken (siehe Anhang C, S. 140) sind auch die reinen Quelltexte verfügbar. Die Archive befinden sich im GraLab-Wurzelverzeichnis `/home/ems/GraLab42` und sind als `.tar.gz` für die Unix-Welt und als `.zip` für die OS/2-Welt vorhanden.

Das Archiv wird in ein beliebiges Verzeichnis ausgepackt. Dieses ist dann das Wurzelverzeichnis und wird in der Make-Variablen `$(GRALAB_HOME)` angegeben. Danach befinden sich die Quelltexte im Unterverzeichnis `src`. Die Maketemplate-Dateien für die unterstützten Architekturen sind im Unterverzeichnis `lib`. Im Unterverzeichnis `build` befinden sich Makefile-Strukturen, mit denen die Bibliotheken compiliert werden.

Zum Erzeugen einer Bibliothek wechselt man in das entsprechende Verzeichnis unter `build`. Mit

```
make GRALAB_HOME=(lokales Verzeichnis) GRALAB_LIBTYPE=cdtu all
```

wird eine Bibliothek erzeugt und im Zielverzeichnis installiert. An der Stelle von `cdtu` muß die gewünschte Bibliotheksvariante stehen.

Bevor eine neue Bibliotheksvariante erzeugt wird, muß zuerst mit

```
make GRALAB_HOME=(lokales Verzeichnis) veryclean
```

das Verzeichnis aufgeräumt werden.

Dabei muß die Umgebungsvariable `PATH` so eingestellt sein, daß Programme im aktuellen Verzeichnis gefunden werden, wenn sie ohne Verzeichnisangabe aufgerufen werden.

<sup>2</sup> Auf dem ftp-Server: `ftp://ftphost.uni-koblenz.de/outgoing/GraLab/GraLab4`

# Anhang E

## Fehlermeldungen

In diesem Kapitel sind alle Fehlermeldungen aufgeführt. In der Tabelle E.1 sind die Meldungsnummern, wie sie hier und im Kapitel 4, S. 55, angegeben sind, den Dateien zugeordnet, in denen die Meldungstexte definiert werden. So ist der Text zur Meldung **grf007** in der Datei `msg/msggrf` unter der Nummer `MSG007` zu finden. Die Meldungszeilen werden mit einem Kennbuchstaben eingeleitet. Dieser gibt an, ob in der Zeile die `Condition`, die `Action` oder die `Reaction` beschrieben ist. Weitere Einzelheiten zu den Meldungen befinden sich im Abschnitt 3.4, S. 50, und im Anhang A.3, S. 137.

Bei Änderungen an den Meldungstexten muß beachtet werden, daß die Reihenfolge und die Typen der Parameter in den Meldungen vom Graphenlabor fest vorgegeben ist. Deshalb dürfen die Formatspezifizierer nicht geändert oder in der Reihenfolge umpositioniert werden. Sie haben folgende Bedeutungen:

- %s** Zeichenkette, z.B. Typbezeichner  
Knoten und Kanten werden meistens als Zeichenkette spezifiziert, da ihre Bezeichnung aus  $v$  bzw.  $\pm e$  und der Identifikationsnummer besteht.
- %u** Positive ganze Zahl (incl. 0), z.B. interne Identifikationsnummer für Knoten oder Kanten, Indices in bzw. Größe von Tabellen
- %u** Ganze Zahl, z.B. interne Identifikationsnummer für Kanten, Indices in Listen
- %x** Zeiger auf eine Datenstruktur, z.B. die Adresse einer Graph- oder Typsysteminstanz

Nummer	Meldungsdatei	Gruppe der Meldung
domXXX	msg/msgdom	Wertebereichssystem
grfXXX	msg/msggrf	Graphstruktur
memXXX	msg/msgmem	Freispeicherverwaltung
typXXX	msg/msgtyp	Typsystem
undoXXX	msg/msgundo	Undo-Puffer
valXXX	msg/msgval	Verwaltung der (Attribut-)Werte

Tabelle E.1: Zuordnung der Meldungsnummern zu Meldungsdateien

**dom001**

C: wrong domain identifier: %d  
A: nothing done  
siehe `getAlignment()` S. 106, `getValueSize()` S. 106, `isBasic()` S. 107, `isEnum()` S. 112, `isList()` S. 107, `isRecord()` S. 110, `isTuple()` S. 109, `printDomain()` S. 106, `store()` S. 97

**dom002**

C: can't use unregistered domain  
R: call method with a usable domain  
A: return `G_DomainBottom`  
siehe `newList()` S. 107

**dom003**

C: can't allocate memory  
siehe `addDomain()` S. 108, `add()` S. 111

**dom101**

C: method not allowed for domain '%s'  
R: use BAG or LIST domain  
siehe `getBase()` S. 107, `getSlotSize()` S. 107

**dom201**

C: selector '%s' is already used for domain '%s'  
R: use another selector identifier  
A: selector not added  
siehe `addDomain()` S. 108

**dom202**

C: method not allowed for domain '%s'  
R: use TUPLE or RECORD domain  
siehe `getArity()` S. 109, `110()` S., `getNthDomain()` S. 109, `110()` S., `getNthSelector()` S. 110

**dom204**

C: index is out of range  
R: use an index in %d..%d  
siehe `getNthDomain()` S. 109, `110()` S., `getNthSelector()` S. 110, `getNthDomain()` S. 108, `getNthId()` S. 108

**dom210**

C: can't create domain without components  
R: add any domains to the `G_domainSequence`  
A: return `G_DomainBottom`  
siehe `newRecord()` S. 110, `newTuple()` S. 109

**dom211**

C: component %d (%s) has no selector  
R: all domains must have a selector identifier  
A: return `G_DomainBottom`  
siehe `newRecord()` S. 110

**dom301**

C: constant '%s' is already registered  
R: use another constant identifier  
A: constant not added  
siehe `add()` S. 111

**dom302**

C: method not allowed for domain '%s'  
R: use ENUM domain  
siehe `getCardinality()` S. 112, `getNthConst()` S. 112, `getOrd()` S. 113

**dom304**

C: index is out of range

R: use an index in %d..%d  
 siehe getNthConst() S. 112, getNthId() S. 111, updateByOrd() S. 120

**dom310**

C: can't create domain without constants  
 R: add any identifiers to the G\_idSequence  
 A: return G\_DomainBottom  
 siehe newEnum() S. 112

**dom311**

C: constant not registered  
 A: return 0  
 siehe getOrd() S. 113, updateEnum() S. 119

**grf001**

C: '%u' is not a valid vertex number in graph '%x'  
 A: no vertex created, G\_VertexBottom returned  
 R: use a valid number (1-'%u' in this graph)  
 siehe createVertex() S. 61

**grf002**

C: '%s' already exists  
 A: no vertex created, G\_VertexBottom returned  
 R: use unique numbers  
 siehe createVertex() S. 61

**grf003**

C: %u is not a valid edge number in graph %x  
 A: no vertex created, G\_EdgeBottom returned  
 R: use a valid number (1-%u in this graph)  
 siehe createEdge() S. 61

**grf004**

C: e%u already exists  
 A: no edge created, G\_EdgeBottom returned  
 R: use unique numbers  
 siehe createEdge() S. 61

**grf005**

C: out of memory  
 A: graph is not increased  
 siehe createEdge() S. 61, createVertex() S. 61

**grf007**

C: '%s' is an unused vertex in graph '%x'  
 A: function aborted  
 R: call function with an existing vertex  
 siehe areEqualVertices() S. 89, changeAlpha() S. 62, changeOmega() S. 62,  
 changeThat() S. 63, changeThis() S. 63, changeVType() S. 64, createEdge() S. 61,  
 degree() S. 91, degreeOfClass() S. 91, degreeOfType() S. 92, deleteVertex() S. 62,  
 edgeBetween() S. 93, edgeBetweenOfClass() S. 94, edgeBetweenOfType() S. 94,  
 edgeFromTo() S. 93, edgeFromToOfClass() S. 94, edgeFromToOfType() S. 94, first() S. 76,  
 firstIn() S. 76, firstInOfClass() S. 81, firstInOfType() S. 85, firstOfClass() S. 80,  
 firstOfType() S. 85, firstOut() S. 77, firstOutOfClass() S. 81, firstOutOfType() S. 86,  
 getEAttr() S. 65, getPVTemp() S. 66, getVAttr() S. 65, getVType() S. 63,  
 htmlPrintVertex() S. 70, inDegree() S. 91, inDegreeOfClass() S. 92,  
 inDegreeOfType() S. 92, isAV() S. 64, isVertexBefore() S. 86, nextVertex() S. 75,  
 nextVertexOfClass() S. 79, nextVertexOfType() S. 84, outDegree() S. 91,  
 outDegreeOfClass() S. 92, outDegreeOfType() S. 92, printVertex() S. 69,  
 putVertexAfter() S. 87, putVertexBefore() S. 87, setPVTemp() S. 66

**grf008**

C: '%s' is an unused edge in graph '%x'  
 A: function aborted  
 R: call function with an existing edge  
 siehe alpha() S. 90, areEqualEdges() S. 90, changeAlpha() S. 62, changeEType() S. 64,  
 changeOmega() S. 62, changeThat() S. 63, changeThis() S. 63, deleteEdge() S. 62,  
 getEType() S. 63, getPETemp() S. 68, htmlPrintEdge() S. 70, isAE() S. 64, isBefore() S. 88,  
 isEdgeBefore() S. 87, next() S. 76, nextEdge() S. 76, nextEdgeOfClass() S. 80,  
 nextEdgeOfType() S. 84, nextIn() S. 77, nextInOfClass() S. 81, nextInOfType() S. 85,  
 nextOfClass() S. 80, nextOfType() S. 85, nextOut() S. 77, nextOutOfClass() S. 82,  
 nextOutOfType() S. 86, normal() S. 91, omega() S. 90, printEdge() S. 69, putAfter() S. 88,  
 putBefore() S. 88, putEdgeAfter() S. 87, putEdgeBefore() S. 87, reverse() S. 91,  
 setPETemp() S. 67, thatV() S. 90, thisV() S. 90

**grf009**

C: this(%s) is not equal to this(%s) in graph %x  
 A: function aborted  
 R: call function with edges incident with the same vertex  
 siehe isBefore() S. 88, putAfter() S. 88, putBefore() S. 88

**grf010**

C: put-operation called with equal arguments  
 A: function aborted  
 R: call function with distinct arguments  
 siehe putAfter() S. 88, putBefore() S. 88, putEdgeAfter() S. 87, putEdgeBefore() S. 87,  
 putVertexAfter() S. 87, putVertexBefore() S. 87

**grf013**

C: no temporary attributes to delete in graph %x  
 A: no attributes deleted  
 R: do not call this function if no attributes were created  
 siehe deleteETemp() S. 68, deleteVTemp() S. 67

**grf015**

C: unable to open file '%s' for writing  
 A: no graph written  
 R: verify if you chose a possible filename  
 siehe store() S. 71

**grf016**

C: not a type in type system '%x' (associated with graph '%x')  
 A: function aborted  
 R: use types existing in associated type system  
 siehe isAE() S. 64, isAV() S. 64

**grf021**

C: something is wrong in graph file '%s',  
 see previous message  
 A: empty graph created  
 R: check your graph file  
 siehe load() S. 71, loadIncreaseTypeSystem() S. 72

**grf022**

C: out of memory  
 A: graph can contain %u vertices only  
 siehe G\_graph() S. 57, reInitialize() S. 58

**grf023**

C: out of memory  
 A: graph can contain %u edges only  
 siehe G\_graph() S. 57, reInitialize() S. 58

**grf024**

C: out of memory

---

A: no temporary attribute layer created  
siehe createETemp() S. 67, createVTemp() S. 66

**grf025**

C: out of memory  
A: no graph created, NULL pointer returned  
siehe load() S. 71, loadIncreaseTypeSystem() S. 72

**grf029**

C: no temporary attribute layer created  
A: temporary attribute not set  
R: create temporary attribute layer  
siehe setPETemp() S. 67, setPVTemp() S. 66

**grf030**

C: unable to open file '%s' for reading  
A: no graph created, NULL pointer returned  
R: check your filename  
siehe load() S. 71, loadIncreaseTypeSystem() S. 72

**grf031**

C: no temporary attribute layer created  
A: NULL pointer returned  
R: create temporary attribute layer  
siehe getPETemp() S. 68, getPVTemp() S. 66

**grf033**

C: writing file failed  
A: file is unusable  
R: check whether you exceeded your disk quota  
siehe store() S. 71

**grf035**

C: format error in graph file:  
'%s' expected  
R: check your graph file  
siehe load() S. 72, loadIncreaseTypeSystem() S. 72

**grf038**

C: something is wrong in your software, %u of %u vertieces  
A: program aborted  
R: contact your software developer  
siehe createVertex() S. 61

**grf039**

C: something is wrong in your software, %u of %u edges  
A: program aborted  
R: contact your software developer  
siehe createEdge() S. 61

**grf040**

C: out of memory for vertex attributes  
A: no vertex created, G\_VertexBottom returned  
siehe createVertex() S. 61, load() S. 72, loadIncreaseTypeSystem() S. 72

**grf041**

C: out of memory for edge attributes  
A: no edge created, G\_EdgeBottom returned  
siehe createEdge() S. 61, load() S. 72, loadIncreaseTypeSystem() S. 72

**grf042**

C: type '%s' can't be used  
A: no vertex created, G\_VertexBottom returned

R: use an exported type  
 siehe createVertex() S. 61, firstEdgeOfClass() S. 80, firstEdgeOfType() S. 84,  
 firstInOfClass() S. 81, firstInOfType() S. 85, firstOfClass() S. 80, firstOfType() S. 85,  
 firstOutOfClass() S. 81, firstOutOfType() S. 86, firstVertexOfClass() S. 79,  
 firstVertexOfType() S. 83, nextEdgeOfClass() S. 80, nextEdgeOfType() S. 84,  
 nextInOfClass() S. 81, nextInOfType() S. 85, nextOfClass() S. 80, nextOfType() S. 85,  
 nextOutOfClass() S. 82, nextOutOfType() S. 86, nextVertexOfClass() S. 79,  
 nextVertexOfType() S. 84

**grf043**

C: type '%s' can't be used  
 A: no edge created, G\_EdgeBottom returned  
 R: use an exported type  
 siehe createEdge() S. 62

**grf044**

C: type '%s' can't be used  
 A: type not changed, function aborted  
 R: use an exported type  
 siehe changeEType() S. 64, changeVType() S. 64

**grf045**

C: out of memory for attributes  
 A: type not changed, function aborted  
 siehe changeEType() S. 64, changeVType() S. 64

**grf050**

C: type %d is not usable:  
 see previous messages  
 A: create vertex %d as 'TypeNull'  
 siehe load() S. 72, loadIncreaseTypeSystem() S. 72

**grf051**

C: type %d is not usable:  
 see previous messages  
 A: create edge %d as 'TypeNull'  
 siehe load() S. 72, loadIncreaseTypeSystem() S. 72

**mem301**

C: out of memory  
 A: program aborted  
 R: exit some of your programs  
 siehe G\_id() S. 126, G\_typeSystem() S. 96

**typ001**

C: domain mismatch: '%s' expected, '%s' used  
 A: no value assigned  
 R: use corresponding values  
 siehe assignValue() S. 115

**typ002**

C: type-system '%x' has no attribute '%s'  
 A: function aborted  
 R: use a registered attribute  
 siehe changeEType() S. 64, changeVType() S. 64, getEAttr() S. 65, getVAttr() S. 65,  
 addAttr() S. 102, getAttrId() S. 99, getDomain() S. 99, hasAttr() S. 103

**typ003**

C: type '%s' has no attribute '%s' in type-system '%x'  
 A: function aborted  
 R: use another attribute  
 siehe getEAttr() S. 65, getVAttr() S. 65

**typ004**

C: type-system '%x' has no type '%s'  
 A: function aborted  
 R: use a registered type  
 siehe degreeOfClass() S. 91, degreeOfType() S. 92, edgeBetweenOfClass() S. 94,  
 edgeBetweenOfType() S. 94, edgeFromToOfClass() S. 94, edgeFromToOfType() S. 94,  
 firstEdgeOfClass() S. 80, firstEdgeOfType() S. 84, firstInOfClass() S. 81,  
 firstInOfType() S. 85, firstOfClass() S. 80, firstOfType() S. 84, firstOutOfClass() S. 81,  
 firstOutOfType() S. 86, firstVertexOfClass() S. 79, firstVertexOfType() S. 83,  
 getEAttr() S. 65, getVAttr() S. 65, inDegreeOfClass() S. 92, inDegreeOfType() S. 92,  
 nextEdgeOfClass() S. 80, nextEdgeOfType() S. 84, nextInOfClass() S. 81,  
 nextInOfType() S. 85, nextOfClass() S. 80, nextOfType() S. 85, nextOutOfClass() S. 82,  
 nextOutOfType() S. 86, nextVertexOfClass() S. 79, nextVertexOfType() S. 84,  
 outDegreeOfClass() S. 92, outDegreeOfType() S. 92, addAttr() S. 102, exportType() S. 102,  
 getAttr() S. 103, getAttrCount() S. 103, getFirstSubType() S. 104,  
 getFirstSuperType() S. 104, getNthAttr() S. 103, getTypeId() S. 101, hasAttr() S. 103,  
 isA() S. 104, isExported() S. 102, setIsA() S. 104

**typ005**

C: type '%s' is already exported in type-system '%x'  
 A: function aborted  
 R: use a usable type  
 siehe addAttr() S. 102, exportType() S. 102, getNthAttr() S. 103, setIsA() S. 104

**typ007**

C: type-class '%s' has an attribute '%s'  
 in type-system '%x'  
 A: function aborted  
 R: use another attribute identifier  
 siehe addAttr() S. 102, setIsA() S. 104

**typ008**

C: sub-type '%s' has an attribute '%s:%s'  
 A: see below message  
 siehe addAttr() S. 102, setIsA() S. 104

**typ009**

C: out of memory  
 A: type-system '%x' is not increased  
 siehe newType() S. 101

**typ012**

C: unknown domain  
 A: function aborted  
 R: use a domain from the domain-system  
 siehe newAttr() S. 98

**typ013**

C: index is out of range  
 R: use an index in %d..%d  
 siehe getNthAttr() S. 100, getNthType() S. 101

**typ015**

C: unable to open file '%s' for writing  
 A: no type-system written  
 R: verify if you chose a possible filename  
 siehe store() S. 97

**typ016**

C: writing file failed  
 A: file is unusable  
 R: check whether you exceeded your disk quota  
 siehe store() S. 97

**typ017**

C: unable to open file '%s' for reading  
A: type-system not changed  
R: check your filename  
siehe increase() S. 97, load() S. 97

**typ018**

C: something is wrong in type-system file '%s',  
see previous message  
A: type-system not changed  
R: check your file  
siehe increase() S. 97, load() S. 97

**typ030**

C: format error in type-system file:  
'%s' expected  
A: type-system unusable  
R: check your file  
siehe increase() S. 97, load() S. 97

**undo001**

C: overflow in undoBuffer  
A: undo disabled until next call of undoBuffer::commit  
R: increase size of undoBuffer  
siehe mark() S. 121, pushAssignInt() S. 122

**undo002**

C: undo not possible since overflow occurred previously  
A: undo not executed  
R: increase size of undoBuffer  
siehe undo() S. 122

**undo003**

C: out of memory  
A: undoBuffer can contain %u items only  
siehe G\_undoBuffer() S. 120

**val002**

C: out of memory  
siehe updateBOOL() S. 116

**val003**

C: corrupted domain  
siehe OK() S. 118

**val004**

C: corrupted data  
siehe OK() S. 118

**val011**

C: domain violation in assignment %s:=%s  
A: leave value unchanged  
R: check your program  
siehe updateBOOL() S. 116, updateByOrd() S. 120, updateEnum() S. 119

**val021**

C: can't destroy value: value is a graph attribute  
A: 'false' returned  
siehe deleteValue() S. 114

**val022**

C: can't destroy value: unspecified domain  
A: 'false' returned  
siehe deleteValue() S. 114

**val023**

C: can't destroy value: value is a part of a structured data  
A: 'false' returned  
siehe deleteValue() S. 114

**val031**

C: can't manipulate list at position %d  
A: G\_ValueBottom returned  
siehe insertIntoList() S. 117

**val032**

C: domain violation: %s is no list  
A: G\_ValueBottom returned  
siehe insertIntoList() S. 117

**val033**

C: domain violation: %s expected instead of %s  
A: G\_ValueBottom returned  
siehe insertIntoList() S. 117

**val041**

C: can't manipulate list at position %d  
R: use range %d..%d  
siehe deleteFromList() S. 117, getNthListItem() S. 117

**val042**

C: domain violation: %s is no list  
siehe deleteFromList() S. 117, first() S. 118, getNthListItem() S. 117, isMember() S. 117,  
next() S. 118, OK() S. 118

**val043**

C: domain violation: %s expected instead of %s  
siehe getBOOL() S. 116, getCardinality() S. 117, getEnum() S. 119, getOrd() S. 120,  
isMember() S. 117, next() S. 118, OK() S. 118

**val044**

C: values are assigned to different graphs  
siehe next() S. 118, OK() S. 118

**val051**

C: can't manipulate tuple at position %d  
R: use range %d..%d  
siehe getNthTupleItem() S. 119

**val052**

C: domain violation: %s is no tuple  
siehe getNthTupleItem() S. 119

**val061**

C: domain violation: %s is no record  
siehe getRecordItem() S. 119

**val062**

C: domain violation: '%s' has no component named '%s'  
siehe getRecordItem() S. 119

# Anhang F

## Übersicht über die Verzeichnisse

<code>/home/ems/GraLab4/</code>	Wurzelverzeichnis
<code>./src/</code>	Quelltextdateien
<code>makelist.lib</code>	Liste der benutzten Dateien
<code>g_align.c, g_sysdep.c</code>	Systemkonfiguration
<code>g_domain.c, g_domsto.c</code>	Wertebereichssystem
<code>g_tsys.c, g_tysto.c</code>	Typsystem
<code>g_grfall.c, g_grfatr.c, ...</code>	Graph
<code>...</code>	weitere Quelldateien
<code>./include/</code>	Header-Dateien
<code>g_graph.h, g_grfcls.h, ...</code>	
<code>./inline/</code>	Header-Dateien mit Inline-Funktionen
<code>g_domain.h, g_grfatr.h, ...</code>	
<code>./lib/</code>	architekturabhängige Dateien
<code>./sparc-sun-solaris2.5/</code>	Systemumgebung
<code>./GNUgcc/</code>	Compiler
<code>readme</code>	Beschreibung der Optionen
<code>makecdtu.tpl, ..., makeu.tpl</code>	Makefile-Templates
<code>g_align.h, g_sysdep.h</code>	System-Headerdateien
<code>libgraphcdtu.a, ..., libgraphu.a</code>	Bibliotheken zum Binden
<code>gralab4-cdtu.tar.gz,</code>	Archive
<code>..., gralab4-u.tar.gz</code>	
<code>./SUNWspro/</code>	weitere Compiler
<code>..., (analog)</code>	
<code>./sparc-sun-sunos4.1.3/ ...</code>	weitere Systemumgebungen
<code>./i386-unknown-solaris2.6/ ...</code>	
<code>./i486-unknown-linux/ ...</code>	
<code>./i386-os2/ ...</code>	
<code>./i386-win/ ...</code>	
<code>./msg/</code>	Laufzeitmeldungen
<code>msgdom, msgems, msggrf, ...</code>	
<code>./example/</code>	Beispielprogramm
<code>makefile, parkhaus.c</code>	
<code>./docs/</code>	Quellen der Dokumentation
<code>libgraph.ps</code>	Handbuch, PostScript
<code>./libgraph/</code>	Handbuch, HTML-Version
<code>./build/</code>	Arbeitsverzeichnis zum Compilieren
<code>./sparc-sun-solaris2.5/</code>	
<code>./GNUgcc/</code>	
<code>makefile</code>	
<code>./SUNWspro/ ...</code>	
<code>./sparc-sun-sunos4.1.3/ ...</code>	
<code>./i386-unknown-solaris2.6/ ...</code>	
<code>./i486-unknown-linux/ ...</code>	
<code>./i386-os2/ ...</code>	
<code>./i386-win/ ...</code>	

# Literaturverzeichnis

- EBERT, J. and A. FRANZKE (1995). *A Declarative Approach to Graph Based Modeling*. In MAYR, E., G. SCHMIDT, and G. TINHOFER, eds.: *Graphtheoretic Concepts in Computer Science*, no. 903 in LNCS, pp. 38–50, Berlin. Springer.
- EBERT, J., A. WINTER, P. DAHM, A. FRANZKE, and R. SÜTTENBACH (1996). *Graph Based Modeling and Implementation with EER/GRAL*. In THALHEIM, B., ed.: *15th International Conference on Conceptual Modeling (ER'96), Proceedings*, no. 1157 in LNCS, pp. 163–178, Berlin. Springer.
- KNUTH, DONALD E. (1973). *Sorting and Searching*, vol. 3 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA, USA.
- RUMBAUGH, JAMES, M. BLAHA, W. PREMERLANI, F. EDDY und W. LORENSEN (1993). *Objekt-orientiertes Modellieren und Entwerfen*. Hanser, München.
- TARJAN, ROBERT (1972). *Depth-first search and linear graph algorithms*. *SIAM Journal on Computing*, 1(2):146–160.

# Index

Unterstrichene Seitennummern verweisen auf Seiten, die wichtige Informationen über die Einträge enthalten, z.B. die genaue Definition oder eine detaillierte Beschreibung. Normal gesetzte Seitennummern beziehen sich auf andere Erwähnungen im Text.

Die meisten Methoden erscheinen doppelt im Index. Unter *Klassen* sind sie nach Klassen sortiert. Unter *Methoden* erscheinen sie nach Methodennamen sortiert, mit den Klassen, in denen sie definiert sind.

- $\alpha$ , [10](#)
- Anfangsknoten, [10](#)
- Attribut, [14](#), [18](#), [26](#), [27](#), [29](#), [64](#), [65](#)
  - temporäres, [14](#), [14](#), [35](#), [42](#), [65](#)
- Attributierung, [14](#)
- Attributierungsschema, [14](#), [28](#), [64](#), [102](#)
- Außengrad, [12](#)
  
- Basiswertebereich, [18](#)
  
- Checking, [50](#), [53](#)
  
- $\delta$ , *siehe* Grad
  - $\delta^+$ , *siehe* Außengrad
  - $\delta^-$ , *siehe* Innengrad
  
- Endknoten, [10](#)
- exportierter Typ, [102](#), [102](#)
  
- gerichtete Kante, [10](#)
- gerichteter Graph, [10](#)
- Grad, [12](#)
- Graph, [9](#), [10](#)
  - gerichteter, [10](#)
  - ungerichteter, [11](#)
- Gruppe, [50](#), [51](#), [55](#), [128](#), [137](#), [138](#)
  
- in-Kante, [10](#)
- Innengrad, [12](#)
- inzident, [10](#)
- inzidente Kante, [12](#)
- is-a, [13](#)
- isoliert, [10](#)
  
- Kante, [10](#), [10](#)
  - gerichtete, [10](#)
  - in-, [10](#)
  - inzidente, [10](#), [12](#)
  - Mehrfach-, [10](#)
  - normalisierte, [46](#)
  - orientierte, [11](#), [12](#), [46](#), [90](#)
  - out-, [10](#)
  - ungerichtete, [11](#)
  
- Klasse, [13](#), [13](#), [35](#)
- Klassen
  - \_G\_edge, [59](#)
  - \_G\_vertex, [58](#)
  - G\_domain, [18](#), [19](#), [105–110](#), [112](#)
    - G\_domain, [105](#)
    - getAlignment, [106](#)
    - getArity, [109](#), [110](#)
    - getBase, [107](#)
    - getCardinality, [112](#)
    - getNthConst, [112](#)
    - getNthDomain, [109](#), [110](#)
    - getNthSelector, [110](#)
    - getOrd, [112](#)
    - getSlotSize, [107](#)
    - getValueSize, [106](#)
    - isBasic, [106](#)
    - isBOOL, [106](#)
    - isDOUBLE, [106](#)
    - isEnum, [112](#)
    - isINT, [106](#)
    - isList, [107](#)
    - isRecord, [110](#)
    - isSTRING, [106](#)
    - isTuple, [109](#)
    - newEnum, [112](#)
    - newList, [107](#)
    - newRecord, [109](#)
    - newTuple, [108](#)
    - OK, [105](#)
    - operator=, [105](#)
    - operator==, [105](#)
    - printDomain, [106](#)
  - G\_domainSequence, [108](#), [109](#)
    - addDomain, [108](#)
    - G\_domainSequence, [108](#)
    - getNthDomain, [108](#)
    - getNthId, [108](#)
  - G\_edge, [18](#), [23](#), [55](#), [58–60](#)
    - G\_edge, [59](#)
    - hashVal, [60](#)

- operator==, 59
- print, 60
- printNormal, 60
- read, 60
- G\_getEnvVar, 129
- G\_graph, 18, 19, 57, 58, 60–73, 75–77, 79–81, 83–95
  - ~G\_graph, 58
- alpha, 90
- areEqualEdges, 90
- areEqualVertices, 89
- changeAlpha, 62
- changeEType, 64
- changeOmega, 62
- changeThat, 63
- changeThis, 63
- changeVType, 64
- createEdge, 61
- createETemp, 67
- createVertex, 60, 61
- createVTemp, 65
- degree, 91
- degreeOfClass, 91
- degreeOfType, 92
- deleteEdge, 62
- deleteETemp, 68
- deleteVertex, 62
- deleteVTemp, 66
- edgeBetween, 93
- edgeBetweenOfClass, 94
- edgeBetweenOfType, 94
- edgeCount, 93
- edgeFromTo, 93
- edgeFromToOfClass, 94
- edgeFromToOfType, 93
- edgeMax, 93
- first, 76
- firstEdge, 76
- firstEdgeOfClass, 79
- firstEdgeOfType, 84
- firstIn, 76
- firstInOfClass, 81
- firstInOfType, 85
- firstOfClass, 80
- firstOfType, 84
- firstOut, 77
- firstOutOfClass, 81
- firstOutOfType, 85
- firstVertex, 75
- firstVertexOfClass, 79
- firstVertexOfType, 83
- G\_graph, 57
- getE, 95
- getEAttr, 65
- getENo, 95
- getETempLevel, 68
- getEType, 63
- getPETemp, 67
- getPUndoBuffer, 95
- getPVTemp, 66
- getTypeSystem, 63
- getV, 95
- getVAttr, 65
- getVNo, 95
- getVTempLevel, 67
- getVType, 63
- htmlPrint, 70
- htmlPrintEdge, 70
- htmlPrintVertex, 69
- inDegree, 91
- inDegreeOfClass, 92
- inDegreeOfType, 92
- isAE, 64
- isAV, 64
- isBefore, 88
- isEdge, 88
- isEdgeBefore, 87
- isEdgeBottom, 89
- isNormal, 89
- isReverse, 89
- isValidE, 89
- isValidV, 89
- isVertex, 88
- isVertexBefore, 86
- isVertexBottom, 89
- load, 71
- loadIncreaseTypeSystem, 72
- loadStructure, 73
- next, 76
- nextEdge, 76
- nextEdgeOfClass, 80
- nextEdgeOfType, 84
- nextIn, 77
- nextInOfClass, 81
- nextInOfType, 85
- nextOfClass, 80
- nextOfType, 85
- nextOut, 77
- nextOutOfClass, 81
- nextOutOfType, 86
- nextVertex, 75
- nextVertexOfClass, 79
- nextVertexOfType, 84
- normal, 90
- omega, 90
- outDegree, 91
- outDegreeOfClass, 92
- outDegreeOfType, 92
- print, 69
- printEdge, 69
- printVertex, 68
- putAfter, 88

- putBefore, 88
- putEdgeAfter, 87
- putEdgeBefore, 87
- putVertexAfter, 87
- putVertexBefore, 87
- reInitialize, 58
- reverse, 91
- setPETemp, 67
- setPUndoBuffer, 95
- setPVTemp, 66
- store, 71
- storeStructure, 73
- thatV, 90
- thisV, 90
- vertexCount, 93
- vertexMax, 93
- G\_id, 126, 127
  - G\_id, 126
  - OK, 127
  - operator=, 126
  - operator==, 127
  - str, 126
- G\_idSequence, 111
  - add, 111
  - G\_idSequence, 111
  - getNthId, 111
- G\_msg, 51, 128, 129
  - addDirectory, 129
  - msg, 128
  - setHeaders, 129
- G\_tempAttribute, 14, 35, 124, 125
  - ~G\_tempAttribute, 124
  - G\_tempAttribute, 124
  - getLayerId, 125
  - getNextInStack, 125
  - logicalDelete, 125
  - logicalUndelete, 125
  - print, 124
- G\_trace, 127, 128
  - G\_trace, 127
  - set, 128
  - setFile, 128
- G\_trans, 73
- G\_type, 14
- G\_typeSystem, 14, 18, 96–104
  - ~G\_typeSystem, 96
  - addAttr, 102
  - checkFrom, 98
  - exportType, 102
  - G\_typeSystem, 96
  - getAttr, 99, 103
  - getAttrCount, 100, 103
  - getAttrId, 99
  - getDomain, 99
  - getFirstAttr, 99
  - getFirstAttrById, 100
  - getFirstSubType, 104
  - getFirstSuperType, 104
  - getNextAttr, 99
  - getNextAttrById, 100
  - getNextSubType, 104
  - getNextSuperType, 104
  - getNthAttr, 100, 103
  - getNthType, 101
  - getType, 101
  - getTypeCount, 101
  - getTypeId, 101
  - hasAttr, 103
  - increase, 97
  - isA, 104
  - isExported, 102
  - knowsAttr, 98
  - knowsType, 101
  - load, 97
  - newAttr, 98
  - newType, 100
  - OK, 100, 101
  - reInitialize, 96
  - restoreFrom, 98
  - setIsA, 103
  - store, 96
  - storeOn, 98
- G\_undoBuffer, 39, 40, 95, 120–123
  - ~G\_undoBuffer, 120
  - commit, 121
  - G\_undoBuffer, 120
  - isAvailable, 123
  - isMarked, 121
  - isOk, 123
  - mark, 121
  - pushAction, 123
  - pushAssignInt, 122
  - pushAssignPointer, 122
  - pushAssignUInt, 122
  - pushCommitAction, 123
  - pushUndoAction, 123
  - undo, 121, 122
- G\_valueRef, 18, 19, 113–120
  - ~G\_valueRef, 113
  - assignValue, 115
  - copyValue, 115
  - createValue, 114
  - deleteFromList, 117
  - deleteValue, 114
  - first, 118
  - G\_valueRef, 113, 114
  - getBOOL, 116
  - getCardinality, 117
  - getDomain, 115
  - getDOUBLE, 116
  - getEnum, 119
  - getINT, 116

- getNthListItem, 117
- getNthTupleItem, 118
- getOrd, 119
- getRecordItem, 119
- getSTRING, 116
- insertIntoList, 116
- isEqualValue, 116
- isMember, 117
- next, 118
- OK, 118
- operator=, 113
- printDomain, 115
- printValue, 115
- updateBOOL, 116
- updateByOrd, 120
- updateDOUBLE, 116
- updateEnum, 119
- updateINT, 116
- updateSTRING, 116
- G\_vertex, 18, 23, 55, 58, 59
  - G\_vertex, 58
  - hashVal, 59
  - operator==, 58
  - print, 59
  - read, 59
- operator«, 59, 60, 69, 106, 115
- operator», 59, 60
- v, 127
- Knoten, [10](#), [11](#)
  - Anfangs-, [10](#)
  - End-, [10](#)
  - isolierter, [10](#)
  - Start-, [10](#)
  - Ziel-, [10](#)
- Konstanten
  - G\_DomainBottom, 105
  - G\_EdgeBottom, 36, 57, 59, 60
  - G\_IdBottom, 126
  - G\_IdNull, 126
  - G\_VertexBottom, 55, 58, 59
- $\Lambda$ , [12](#)
- Make-Variablen
  - GRALAB\_A, 145
  - GRALAB\_APRE, 145
  - GRALAB\_AR, 145
  - GRALAB\_ARCH, 145
  - GRALAB\_ARFLAGS, 145
  - GRALAB\_C, 145
  - GRALAB\_CAT, 145
  - GRALAB\_CC, 145
  - GRALAB\_CFLAGS, 142, 145
  - GRALAB\_COMPILE\_c, 25, 146
  - GRALAB\_CP, 145
  - GRALAB\_CPPFLAGS, 142, 143, 145
  - GRALAB\_DEP, 145
  - GRALAB\_DEPEND\_c, 146
  - GRALAB\_DIRSEP, 145
  - GRALAB\_E, 145
  - GRALAB\_HOME, 48, 140, 145, 146
  - GRALAB\_INCDIR, 48, 145
  - GRALAB\_INSTDIR, 145
  - GRALAB\_INSTLIB, 145
  - GRALAB\_LDFLAGS, 142, 145
  - GRALAB\_LDLIBS, 145
  - GRALAB\_LIB, 26, 145
  - GRALAB\_LIBBASE, 145
  - GRALAB\_LIBDIR, 49, 145
  - GRALAB\_LIBFILE, 146
  - GRALAB\_LIBTYPE, 145
  - GRALAB\_LINK\_c, 146
  - GRALAB\_LINK\_o, 26, 146
  - GRALAB\_O, 145
  - GRALAB\_PATHSEP, 145
  - GRALAB\_RANLIB, 145
  - GRALAB\_RM, 145
  - GRALAB\_SRCDIR, 145
- Makefile-Template, [49](#), 143
- Makro
  - G\_ALIGN\_xxx, 144
  - G\_chk, 53
  - G\_DEBUG, 142, 143
  - G\_EXPLICIT, 144
  - G\_forAllEdges, 11, 74
  - G\_forAllEdgesOfClass, 78
  - G\_forAllEdgesOfType, 82
  - G\_forAllIncidentEdges, 12, 44, 74
  - G\_forAllIncidentEdgesOfClass, 78
  - G\_forAllIncidentEdgesOfType, 82
  - G\_forAllInEdges, 12, 75
  - G\_forAllInEdgesOfClass, 78
  - G\_forAllInEdgesOfType, 83
  - G\_forAllOutEdges, 12, 25, 44, 75
  - G\_forAllOutEdgesOfClass, 79
  - G\_forAllOutEdgesOfType, 83
  - G\_forAllVertices, 11, 74
  - G\_forAllVerticesOfClass, 77
  - G\_forAllVerticesOfType, 82
  - G\_FREE\_HAS\_CHARARG, 144
  - G\_FREE\_RETURNS\_INT, 144
  - G\_HAS\_GETPAGESIZE, 144
  - G\_HAS\_IO\_H, 144
  - G\_HAS\_IOS\_BIN, 144
  - G\_HAS\_IOS\_BINARY, 144
  - G\_HAS\_SC\_PAGESIZE, 144
  - G\_HAS\_STRSTREA\_H, 144
  - G\_HAS\_STRSTREAM\_H, 144
  - G\_HAS\_STRTOUL, 144
  - G\_HAS\_UNISTD\_H, 144
  - G\_INIT\_REF\_BUG, 144
  - G\_IOS\_BIN, 71, 72, 97, 144
  - G\_KNOWS\_BOOL, 144

- G.KNOWS\_EXPLICIT, 144
- G.KNOWS\_NEWVECTOR, 144
- G.MSGPATH, 51
- G.PAGESIZE, 144
- G.trc, 52, 127
- G.TrcDeclaration, 52, 127
- G.trcEnter, 52, 127
- G.trcLeave, 52, 127
- G.UINTSIZE, 144
- NO\_CHK, 142, 143
- NO\_TRC, 142, 143
- NO\_UNDO, 142, 143
- Mehrfachkante, [10](#), 11
- Methoden
  - ~G\_graph
    - G\_graph, 58
  - ~G\_tempAttribute
    - G\_tempAttribute, 124
  - ~G\_typeSystem
    - G\_typeSystem, 96
  - ~G\_undoBuffer
    - G\_undoBuffer, 120
  - ~G\_valueRef
    - G\_valueRef, 113
- 110, 148
- add
  - G\_idSequence, 111, 148
- addAttr
  - G\_typeSystem, 98, 102, 152, 153
- addDirectory
  - G\_msg, 51, 128, 129
- addDomain
  - G\_domainSequence, 108, 148
- alpha
  - G\_graph, 90, 150
- areEqualEdges
  - G\_graph, 90, 150
- areEqualVertices
  - G\_graph, 89, 149
- assignValue
  - G\_valueRef, 115, 152
- changeAlpha
  - G\_graph, 62, 149, 150
- changeEType
  - G\_graph, 64, 150, 152
- changeOmega
  - G\_graph, 62, 149, 150
- changeThat
  - G\_graph, 63, 149, 150
- changeThis
  - G\_graph, 45, 63, 149, 150
- changeVType
  - G\_graph, 64, 149, 152
- checkFrom
  - G\_typeSystem, 73, 98
- commit
  - G\_undoBuffer, 39–42, 95, 121–123
  - G\_undoStack, 125
- copyValue
  - G\_valueRef, 115
- createEdge
  - G\_graph, 14, 24, 61, 138, 149, 151, 152
- createETemp
  - G\_graph, 67, 125, 151
- createValue
  - G\_valueRef, 114
- createVertex
  - G\_graph, 14, 24, 60, 61, 149, 151, 152
- createVTemp
  - G\_graph, 37, 65, 125, 151
- degree
  - G\_graph, 91, 149
- degreeOfClass
  - G\_graph, 91, 149, 153
- degreeOfType
  - G\_graph, 92, 149, 153
- deleteEdge
  - G\_graph, 42, 62, 114, 150
- deleteETemp
  - G\_graph, 68, 150
- deleteFromList
  - G\_valueRef, 117, 155
- deleteValue
  - G\_valueRef, 114, 154, 155
- deleteVertex
  - G\_graph, 42, 62, 114, 125, 149
- deleteVTemp
  - G\_graph, 66, 150
- edgeBetween
  - G\_graph, 93, 149
- edgeBetweenOfClass
  - G\_graph, 94, 149, 153
- edgeBetweenOfType
  - G\_graph, 94, 149, 153
- edgeCount
  - G\_graph, 93
- edgeFromTo
  - G\_graph, 46, 93, 149
- edgeFromToOfClass
  - G\_graph, 94, 149, 153
- edgeFromToOfType
  - G\_graph, 93, 149, 153
- edgeMax
  - G\_graph, 93
- exportType
  - G\_typeSystem, 102, 153
- first
  - G\_graph, 44, 75, 76, 149
  - G\_valueRef, 118, 155
- firstEdge
  - G\_graph, 74, 76
- firstEdgeOfClass

- G\_graph, 78, 79, 152, 153
- firstEdgeOfType
  - G\_graph, 82, 84, 152, 153
- firstIn
  - G\_graph, 75, 76, 149
- firstInOfClass
  - G\_graph, 78, 81, 149, 152, 153
- firstInOfType
  - G\_graph, 83, 85, 149, 152, 153
- firstOfClass
  - G\_graph, 78, 80, 149, 152, 153
- firstOfType
  - G\_graph, 83, 84, 149, 152, 153
- firstOut
  - G\_graph, 75, 77, 149
- firstOutOfClass
  - G\_graph, 79, 81, 149, 152, 153
- firstOutOfType
  - G\_graph, 83, 85, 149, 152, 153
- firstVertex
  - G\_graph, 74, 75
- firstVertexOfClass
  - G\_graph, 78, 79, 152, 153
- firstVertexOfType
  - G\_graph, 82, 83, 152, 153
- G\_domain
  - G\_domain, 105
- G\_domainSequence
  - G\_domainSequence, 108
- G\_edge
  - G\_edge, 59
- G\_getEnvVar, 129
- G\_graph
  - G\_graph, 57, 150
- G\_id
  - G\_id, 126, 152
- G\_idSequence
  - G\_idSequence, 111
- G\_tempAttribute
  - G\_tempAttribute, 124
- G\_trace
  - G\_trace, 127
- G\_TypeNull
  - G\_typeSystem, 96
- G\_typeSystem
  - G\_typeSystem, 96, 152
- G\_undoBuffer
  - G\_undoBuffer, 120, 154
- G\_valueRef
  - G\_valueRef, 113, 114
- G\_vertex
  - G\_vertex, 58
- getAlignment
  - G\_domain, 106, 148
- getArity
  - G\_domain, 109, 110, 148
- getAttr
  - G\_typeSystem, 99, 103, 153
- getAttrCount
  - G\_typeSystem, 100, 103, 153
- getAttrId
  - G\_typeSystem, 99, 152
- getBase
  - G\_domain, 107, 148
- getBOOL
  - G\_valueRef, 116, 155
- getCardinality
  - G\_domain, 112, 148
  - G\_valueRef, 117, 155
- getDomain
  - G\_typeSystem, 99, 152
  - G\_valueRef, 115
- getDOUBLE
  - G\_valueRef, 116
- getE
  - G\_graph, 57, 95
- getEAttr
  - G\_graph, 19, 31, 32, 65, 149, 152, 153
- getENo
  - G\_graph, 57, 95
- getEnum
  - G\_valueRef, 119, 155
- getETempLevel
  - G\_graph, 68
- getEType
  - G\_graph, 63, 150
- getFirstAttr
  - G\_typeSystem, 99
- getFirstAttrById
  - G\_typeSystem, 100
- getFirstSubType
  - G\_typeSystem, 104, 153
- getFirstSuperType
  - G\_typeSystem, 104, 153
- getINT
  - G\_valueRef, 116
- getLayerId
  - G\_tempAttribute, 125
- getNextAttr
  - G\_typeSystem, 99
- getNextAttrById
  - G\_typeSystem, 100
- getNextInStack
  - G\_tempAttribute, 125
- getNextSubType
  - G\_typeSystem, 104
- getNextSuperType
  - G\_typeSystem, 104
- getNthAttr
  - G\_typeSystem, 100, 103, 153
- getNthConst
  - G\_domain, 112, 148, 149

- getNthDomain
  - G\_domain, 109, 110, 148
  - G\_domainSequence, 108, 148
- getNthId
  - G\_domainSequence, 108, 148
  - G\_idSequence, 111, 149
- getNthListItem
  - G\_valueRef, 117, 155
- getNthSelector
  - G\_domain, 110, 148
- getNthTupleItem
  - G\_valueRef, 118, 155
- getNthType
  - G\_typeSystem, 101, 153
- getOrd
  - G\_domain, 112, 148, 149
  - G\_valueRef, 119, 155
- getPETemp
  - G\_graph, 67, 150, 151
- getPUndoBuffer
  - G\_graph, 95
- getPVTemp
  - G\_graph, 66, 149, 151
- getRecordItem
  - G\_valueRef, 119, 155
- getSlotSize
  - G\_domain, 107, 148
- getSTRING
  - G\_valueRef, 116
- getType
  - G\_typeSystem, 101
- getTypeCount
  - G\_typeSystem, 101
- getTypeId
  - G\_typeSystem, 101, 153
- getTypeSystem
  - G\_graph, 63
- getV
  - G\_graph, 55, 95
- getValueSize
  - G\_domain, 106, 148
- getVAttr
  - G\_graph, 19, 31, 65, 149, 152, 153
- getVNo
  - G\_graph, 25, 55, 95
- getVTempLevel
  - G\_graph, 67
- getVType
  - G\_graph, 63, 149
- hasAttr
  - G\_typeSystem, 103, 152, 153
- hashVal
  - G\_edge, 60
  - G\_vertex, 59
- htmlPrint
  - G\_graph, 70
- htmlPrintEdge
  - G\_graph, 70, 150
- htmlPrintVertex
  - G\_graph, 69, 70, 149
- increase
  - G\_typeSystem, 97, 154
- inDegree
  - G\_graph, 91, 149
- inDegreeOfClass
  - G\_graph, 92, 149, 153
- inDegreeOfType
  - G\_graph, 92, 149, 153
- insertIntoList
  - G\_valueRef, 116, 155
- isA
  - G\_typeSystem, 104, 153
- isAE
  - G\_graph, 64, 150
- isAV
  - G\_graph, 35, 64, 149, 150
- isAvailable
  - G\_undoBuffer, 123
- isBasic
  - G\_domain, 106, 148
- isBefore
  - G\_graph, 88, 150
- isBOOL
  - G\_domain, 106
- isDOUBLE
  - G\_domain, 106
- isEdge
  - G\_graph, 88
- isEdgeBefore
  - G\_graph, 87, 150
- isEdgeBottom
  - G\_graph, 89
- isEnum
  - G\_domain, 112, 148
- isEqualValue
  - G\_valueRef, 116, 117
- isExported
  - G\_typeSystem, 102, 153
- isINT
  - G\_domain, 106
- isList
  - G\_domain, 107, 148
- isMarked
  - G\_undoBuffer, 121
- isMember
  - G\_valueRef, 117, 155
- isNormal
  - G\_graph, 89
- isOk
  - G\_undoBuffer, 123
- isRecord
  - G\_domain, 110, 148

- isReverse
  - G\_graph, 89
- isSTRING
  - G\_domain, 106
- isTuple
  - G\_domain, 109, 148
- isValidE
  - G\_graph, 89
- isValidV
  - G\_graph, 89
- isVertex
  - G\_graph, 88
- isVertexBefore
  - G\_graph, 86, 149
- isVertexBottom
  - G\_graph, 89
- knowsAttr
  - G\_typeSystem, 98
- knowsType
  - G\_typeSystem, 101
- load
  - G\_graph, 71, 73, 150–152
  - G\_typeSystem, 34, 97, 154
- loadIncreaseTypeSystem
  - G\_graph, 72, 73, 150–152
- loadStructure
  - G\_graph, 73
- logicalDelete
  - G\_tempAttribute, 42, 125
- logicalUndelete
  - G\_tempAttribute, 42, 125
- mark
  - G\_undoBuffer, 41, 121, 154
- msg
  - G\_msg, 128, 138
- multiplicativehash, 59, 60
- newAttr
  - G\_typeSystem, 98, 153
- newEnum
  - G\_domain, 19, 112, 149
- newList
  - G\_domain, 19, 107, 148
- newRecord
  - G\_domain, 19, 28, 109, 148
- newTuple
  - G\_domain, 19, 108, 148
- newType
  - G\_typeSystem, 100, 153
- next
  - G\_graph, 75, 76, 150
  - G\_valueRef, 118, 155
- nextEdge
  - G\_graph, 74, 76, 150
- nextEdgeOfClass
  - G\_graph, 78, 80, 150, 152, 153
- nextEdgeOfType
  - G\_graph, 82, 84, 150, 152, 153
- nextIn
  - G\_graph, 75, 77, 150
- nextInOfClass
  - G\_graph, 78, 81, 150, 152, 153
- nextInOfType
  - G\_graph, 83, 85, 150, 152, 153
- nextOfClass
  - G\_graph, 78, 80, 150, 152, 153
- nextOfType
  - G\_graph, 83, 85, 150, 152, 153
- nextOut
  - G\_graph, 75, 77, 150
- nextOutOfClass
  - G\_graph, 79, 81, 150, 152, 153
- nextOutOfType
  - G\_graph, 83, 86, 150, 152, 153
- nextVertex
  - G\_graph, 74, 75, 149
- nextVertexOfClass
  - G\_graph, 78, 79, 149, 152, 153
- nextVertexOfType
  - G\_graph, 82, 84, 149, 152, 153
- normal
  - G\_graph, 90, 150
- OK
  - G\_domain, 105
  - G\_id, 127
  - G\_typeSystem, 100, 101
  - G\_valueRef, 118, 154, 155
- omega
  - G\_graph, 25, 90, 150
- operator«, 59, 60, 69, 106, 115
- operator=
  - G\_domain, 105
  - G\_id, 126
  - G\_valueRef, 113
- operator==
  - G\_domain, 105
  - G\_edge, 59
  - G\_id, 127
  - G\_vertex, 58
- operator», 59, 60
- outDegree
  - G\_graph, 91, 149
- outDegreeOfClass
  - G\_graph, 92, 149, 153
- outDegreeOfType
  - G\_graph, 92, 149, 153
- print
  - G\_edge, 60
  - G\_graph, 69
  - G\_tempAttribute, 35, 124
  - G\_vertex, 59
- printDomain
  - G\_domain, 106, 148

- G\_valueRef, 115
  - printEdge
    - G\_graph, 69, 124, 150
  - printNormal
    - G\_edge, 60
  - printValue
    - G\_valueRef, 115
  - printVertex
    - G\_graph, 35, 68, 69, 124, 149
  - push
    - G\_undoBuffer, 39
  - pushAction
    - G\_undoBuffer, 121, 123
  - pushAssignInt
    - G\_undoBuffer, 42, 122, 123, 154
  - pushAssignPointer
    - G\_undoBuffer, 122
  - pushAssignUInt
    - G\_undoBuffer, 122
  - pushCommitAction
    - G\_undoBuffer, 121, 123
  - pushUndoAction
    - G\_undoBuffer, 121, 123
  - putAfter
    - G\_graph, 88, 150
  - putBefore
    - G\_graph, 46, 88, 150
  - putEdgeAfter
    - G\_graph, 87, 150
  - putEdgeBefore
    - G\_graph, 87, 150
  - putVertexAfter
    - G\_graph, 86, 87, 149, 150
  - putVertexBefore
    - G\_graph, 86, 87, 149, 150
  - read
    - G\_edge, 60
    - G\_vertex, 59
  - reInitialize
    - G\_graph, 44, 58, 150
    - G\_typeSystem, 96
  - reInitialize()
    - G\_graph, 57
  - restoreFrom
    - G\_typeSystem, 73, 98
  - reverse
    - G\_graph, 91, 150
  - set
    - G\_trace, 52, 127, 128
  - setFile
    - G\_trace, 52, 127, 128
  - setHeaders
    - G\_msg, 129, 137
  - setIsA
    - G\_typeSystem, 29, 103, 153
  - setPETemp
    - G\_graph, 14, 15, 67, 150, 151
  - setPUndoBuffer
    - G\_graph, 40, 95
  - setPVTemp
    - G\_graph, 14, 15, 66, 149, 151
  - store
    - G\_graph, 71–73, 150, 151
    - G\_typeSystem, 30, 96, 148, 153
  - storeOn
    - G\_typeSystem, 98
  - storeStructure
    - G\_graph, 73
  - str
    - G\_id, 126
  - thatV
    - G\_graph, 90, 150
  - thisV
    - G\_graph, 90, 150
  - undo
    - G\_undoBuffer, 39–42, 121–123, 154
    - G\_undoStack, 125
  - updateBOOL
    - G\_valueRef, 116, 154
  - updateByOrd
    - G\_valueRef, 120, 149, 154
  - updateDOUBLE
    - G\_valueRef, 116
  - updateEnum
    - G\_valueRef, 119, 149, 154
  - updateINT
    - G\_valueRef, 116
  - updateSTRING
    - G\_valueRef, 32, 116
  - vertexCount
    - G\_graph, 93
  - vertexMax
    - G\_graph, 93
- Nulltyp, [14](#), [24](#)  
Nullwert, [31](#), [113](#)
- Obertyp, [13](#)  
 $\omega$ , [10](#)  
Ordnungsnummer, [111](#)  
orientierte Kante, [11](#), [12](#), [90](#)  
Orientierung, [11](#)  
normale, [11](#)  
out-Kante, [10](#)
- Profiler, [142](#)
- Schicht, [15](#), [35](#), [65](#)  
Schlinge, [10](#)  
ungerichtete, [11](#)  
Startknoten, [10](#)
- temporäres Attribut, [14](#), [14](#), [35](#), [42](#), [65](#)

TGraph, 9, 86  
that, [11](#), 12, 63  
this, [11](#), 12, 63  
Tiefensuche, 36  
Tracing, [52](#)  
Typ, 13, [13](#), 18, 63  
    exportieren, 102, [102](#)  
    Klasse, [13](#), 35  
    Ober-, [13](#)  
    Unter-, [13](#)  
Typisierung, [13](#)  
Typsystem, [14](#), 17, 63, [96](#)

Umgebungsvariablen  
    G\_MAXERROR, 128  
    G\_MMAX, 57, 58  
    G\_MSGPATH, 128  
    G\_NMAX, 57, 58

Undo, [15](#), 35  
Undo-Puffer, 15  
ungerichtete Kante, [11](#)  
ungerichtete Schlinge, [11](#)  
ungerichteter Graph, [11](#)  
Untertyp, [13](#)

Variablen  
    BOOL  
        G\_domain, 106  
    DOUBLE  
        G\_domain, 106  
    G\_Chk, 53  
    G\_error, 128  
    G\_fatal, 128  
    G\_warning, 128  
    INT  
        G\_domain, 106  
    out  
        G\_trace, 52, 127  
    STRING  
        G\_domain, 106

Variante, 49

Wert, 14, [18](#)  
Wertebereich, [18](#)  
    Basis-, [18](#)  
    elementarer, [18](#)  
    komplexer, [18](#)  
    konstruierter, [18](#)  
Wertebereichssystem, [18](#)  
Wurzelverzeichnis, [48](#)

Zielknoten, 10

Available Research Reports (since 1995):

1998

- 11/98 *Peter Dahm, Friedbert Widmann.* Das Graphenlabor.
- 10/98 *Jörg Jooss, Thomas Marx.* Workflow Modeling according to WfMC.
- 9/98 *Dieter Zöbel.* Schedulability criteria for age constraint processes in hard real-time systems.
- 8/98 *Wenjin Lu, Ulrich Furbach.* Disjunctive logic program = Horn Program + Control program.
- 7/98 *Andreas Schmid.* Solution for the counting to infinity problem of distance vector routing.
- 6/98 *Ulrich Furbach, Michael Kühn, Frieder Stolzenburg.* Model-Guided Proof Debugging.
- 5/98 *Peter Baumgartner, Dorothea Schäfer.* Model Elimination with Simplification and its Application to Software Verification.
- 4/98 *Bernt Kullbach, Andreas Winter, Peter Dahm, Jürgen Ebert.* Program Comprehension in Multi-Language Systems.
- 3/98 *Jürgen Dix, Jorge Lobo.* Logic Programming and Nonmonotonic Reasoning.
- 2/98 *Hans-Michael Hanisch, Kurt Lautenbach, Carlo Simon, Jan Thieme.* Zeitstempelnetze in technischen Anwendungen.
- 1/98 *Manfred Kamp.* Managing a Multi-File, Multi-Language Software Repository for Program Comprehension Tools — A Generic Approach.

1997

- 32/97 *Peter Baumgartner.* Hyper Tableaux — The Next Generation.
- 31/97 *Jens Woch.* A component-based and abstractivistic Agent Architecture for the modelling of MAS in the Social Sciences.
- 30/97 *Marcel Bresink.* A Software Test-Bed for Global Illumination Research.
- 29/97 *Marcel Bresink.* Deutschsprachige Terminologie des Radiosity-Verfahrens.
- 28/97 *Jürgen Ebert, Bernt Kullbach, Andreas Panse.* The Extract-Transform-Rewrite Cycle - A Step towards MetaCARE.
- 27/97 *Jose Arrazola, Jürgen Dix, Mauricio Osorio.* Confluent Rewriting Systems for Logic Programming Semantics.
- 26/97 *Lutz Priese.* A Note on Nondeterministic Reversible Computations.
- 25/97 *Stephan Philippi.* System modelling using Object-Oriented Pr/T-Nets.
- 24/97 *Lutz Priese, Yurii Rogojine, Maurice Margenstern.* Finite H-Systems with 3 Test Tubes are not Predictable.
- 23/97 *Peter Baumgartner (Hrsg.).* Jahrestreffen der GI-Fachgruppe 1.2.1 'Deduktionssysteme' — Kurzfassungen der Vorträge.
- 22/97 *Jens M. Felderhoff, Thomas Marx.* Erkennung semantischer Integritätsbedingungen in Datenbankanwendungen.
- 21/97 *Angelika Franzke.* Specifying Object Oriented Systems using GDMO, ZEST and SDL'92.
- 20/97 *Angelika Franzke.* Recommendations for an Improvement of GDMO.
- 19/97 *Jürgen Dix, Luís Moniz Pereira, Teodor Przymusiński.* Logic Programming and Knowledge Representation (LPKR '97) (Proceedings of the ILPS '97 Postconference Workshop).
- 18/97 *Lutz Priese, Harro Wimmel.* A Uniform Approach to True-Concurrency and Interleaving Semantics for Petri Nets.
- 17/97 *Ulrich Furbach (Ed.).* IJCAI-97 Workshop on Model Based Automated Reasoning.
- 16/97 *Jürgen Dix, Frieder Stolzenburg.* A Framework to Incorporate Non-Monotonic Reasoning into Constraint Logic Programming.
- 15/97 *Carlo Simon, Hanno Ridder, Thomas Marx.* The Petri Net Tools Neptun and Poseidon.
- 14/97 *Juha-Pekka Tolvanen, Andreas Winter (Eds.).* CAISE'97 — 4th Doctoral Consortium on Advanced Information Systems Engineering, Barcelona, June 16-17, 1997, Proceedings.
- 13/97 *Jürgen Ebert, Roger Süttenbach.* An OMT Metamodel.
- 12/97 *Stefan Brass, Jürgen Dix, Teodor Przymusiński.* Super Logic Programs.
- 11/97 *Jürgen Dix, Mauricio Osorio.* Towards Well-Behaved Semantics Suitable for Aggregation.

- 10/97** *Chandrabose Aravindan, Peter Baumgartner.* A Rational and Efficient Algorithm for View Deletion in Databases.
- 9/97** *Wolfgang Albrecht, Dieter Zöbel.* Integrating Fixed Priority and Static Scheduling to Maintain External Consistency.
- 8/97** *Jürgen Ebert, Alexander Fronk.* Operational Semantics of Visual Notations.
- 7/97** *Thomas Marx.* APRIL - Visualisierung der Anforderungen.
- 6/97** *Jürgen Ebert, Manfred Kamp, Andreas Winter.* A Generic System to Support Multi-Level Understanding of Heterogeneous Software.
- 5/97** *Roger Süttenbach, Jürgen Ebert.* A Booch Metamodel.
- 4/97** *Jürgen Dix, Luis Pereira, Teodor Przymusiński.* Prolegomena to Logic Programming for Non-Monotonic Reasoning.
- 3/97** *Angelika Franzke.* GRAL 2.0: A Reference Manual.
- 2/97** *Ulrich Furbach.* A View to Automated Reasoning in Artificial Intelligence.
- 1/97** *Chandrabose Aravindan, Jürgen Dix, Ilkka Niemelä.* DisLoP: A Research Project on Disjunctive Logic Programming.
- 1996**
- 28/96** *Wolfgang Albrecht.* Echtzeitplanung für Alters- oder Reaktionszeitanforderungen.
- 27/96** *Kurt Lautenbach.* Action Logical Correctness Proving.
- 26/96** *Frieder Stolzenburg, Stephan Höhne, Ulrich Koch, Martin Volk.* Constraint Logic Programming for Computational Linguistics.
- 25/96** *Kurt Lautenbach, Hanno Ridder.* Die Lineare Algebra der Verklemmungsvermeidung — Ein Petri-Netz-Ansatz.
- 24/96** *Peter Baumgartner, Ulrich Furbach.* Refinements for Restart Model Elimination.
- 23/96** *Peter Baumgartner, Peter Fröhlich, Ulrich Furbach, Wolfgang Nejdl.* Tableaux for Diagnosis Applications.
- 22/96** *Jürgen Ebert, Roger Süttenbach, Ingar Uhe.* Meta-CASE in Practice: a Case for KOGGE.
- 21/96** *Harro Wimmel, Lutz Priese.* Algebraic Characterization of Petri Net Pomset Semantics.
- 20/96** *Wenjin Lu.* Minimal Model Generation Based on E-Hyper Tableaux.
- 19/96** *Frieder Stolzenburg.* A Flexible System for Constraint Disjunctive Logic Programming.
- 18/96** *Ilkka Niemelä (Ed.).* Proceedings of the ECAI'96 Workshop on Integrating Nonmonotonicity into Automated Reasoning Systems.
- 17/96** *Jürgen Dix, Luis Moniz Pereira, Teodor Przymusiński.* Non-monotonic Extensions of Logic Programming: Theory, Implementation and Applications (Proceedings of the JICSLP '96 Postconference Workshop W1).
- 16/96** *Chandrabose Aravindan.* DisLoP: A Disjunctive Logic Programming System Based on PROTEIN Theorem Prover.
- 15/96** *Jürgen Dix, Gerhard Brewka.* Knowledge Representation with Logic Programs.
- 14/96** *Harro Wimmel, Lutz Priese.* An Application of Compositional Petri Net Semantics.
- 13/96** *Peter Baumgartner, Ulrich Furbach.* Calculi for Disjunctive Logic Programming.
- 12/96** *Klaus Zitzmann.* Physically Based Volume Rendering of Gaseous Objects.
- 11/96** *J. Ebert, A. Winter, P. Dahm, A. Franzke, R. Süttenbach.* Graph Based Modeling and Implementation with EER/GRAL.
- 10/96** *Angelika Franzke.* Querying Graph Structures with G<sup>2</sup>QL.
- 9/96** *Chandrabose Aravindan.* An abductive framework for negation in disjunctive logic programming.
- 8/96** *Peter Baumgartner, Ulrich Furbach, Ilkka Niemelä.* Hyper Tableaux.
- 7/96** *Ilkka Niemelä, Patrik Simons.* Efficient Implementation of the Well-founded and Stable Model Semantics.
- 6/96** *Ilkka Niemelä.* Implementing Circumscription Using a Tableau Method.
- 5/96** *Ilkka Niemelä.* A Tableau Calculus for Minimal Model Reasoning.
- 4/96** *Stefan Brass, Jürgen Dix, Teodor C. Przymusiński.* Characterizations and Implementation of Static Semantics of Disjunctive Programs.
- 3/96** *Jürgen Ebert, Manfred Kamp, Andreas Winter.* Generic Support for Understanding Heterogeneous Software.

**2/96** *Stefan Brass, Jürgen Dix, Ilkka Niemelä, Teodor. C. Przymusiński.* A Comparison of STATIC Semantics with D-WFS.

**1/96** *J. Ebert (Hrsg.).* Alternative Konzepte für Sprachen und Rechner, Bad Honnef 1995.

### 1995

**21/95** *J. Dix and U. Furbach.* Logisches Programmieren mit Negation und Disjunktion.

**20/95** *L. Priese, H. Wimmel.* On Some Compositional Petri Net Semantics.

**19/95** *J. Ebert, G. Engels.* Specification of Object Life Cycle Definitions.

**18/95** *J. Dix, D. Gottlob, V. Marek.* Reducing Disjunctive to Non-Disjunctive Semantics by Shift-Operations.

**17/95** *P. Baumgartner, J. Dix, U. Furbach, D. Schäfer, F. Stolzenburg.* Deduktion und Logisches Programmieren.

**16/95** *Doris Nolte, Lutz Priese.* Abstract Fairness and Semantics.

**15/95** *Volker Rehrmann (Hrsg.).* 1. Workshop Farbbildverarbeitung.

**14/95** *Frieder Stolzenburg, Bernd Thomas.* Analysing Rule Sets for the Calculation of Banking Fees by a Theorem Prover with Constraints.

**13/95** *Frieder Stolzenburg.* Membership-Constraints and Complexity in Logic Programming with Sets.

**12/95** *Stefan Brass, Jürgen Dix.* D-WFS: A Confluent Calculus and an Equivalent Characterization..

**11/95** *Thomas Marx.* NetCASE — A Petri Net based Method for Database Application Design and Generation.

**10/95** *Kurt Lautenbach, Hanno Ridder.* A Completion of the S-invariance Technique by means of Fixed Point Algorithms.

**9/95** *Christian Fahrner, Thomas Marx, Stephan Philippi.* Integration of Integrity Constraints into Object-Oriented Database Schema according to ODMG-93.

**8/95** *Christoph Steigner, Andreas Weihrauch.* Modelling Timeouts in Protocol Design..

**7/95** *Jürgen Ebert, Gottfried Vossen.* I-Serializability: Generalized Correctness for Transaction-Based Environments.

**6/95** *P. Baumgartner, S. Brünig.* A Disjunctive Positive Refinement of Model Elimination and its Application to Subsumption Deletion.

**5/95** *P. Baumgartner, J. Schumann.* Implementing Restart Model Elimination and Theory Model Elimination on top of SETHEO.

**4/95** *Lutz Priese, Jens Klieber, Raimund Lakmann, Volker Rehrmann, Rainer Schian.* Echtzeit-Verkehrszeichenerkennung mit dem Color Structure Code — Ein Projektbericht.

**3/95** *Lutz Priese.* A Class of Fully Abstract Semantics for Petri-Nets.

**2/95** *P. Baumgartner, R. Hähnle, J. Posegga (Hrsg.).* 4th Workshop on Theorem Proving with Analytic Tableaux and Related Methods — Poster Session and Short Papers.

**1/95** *P. Baumgartner, U. Furbach, F. Stolzenburg.* Model Elimination, Logic Programming and Computing Answers.