1

Einführung

In der Informatik dreht sich alles um "Information verarbeitende Systeme". *Systeme* werden spezifiziert ("modelliert"), konstruiert ("implementiert") und analysiert ("getestet"). Der Aufbau und das Verhalten so eines Systems sind gegeben durch die Zustände, die es annehmen kann, und durch die Art, wie sich diese Zustände ändern können. Beides lässt sich mit *Regeln* abstrakt beschreiben. Ganz allgemein definieren Regeln binäre Relationen über Objekten. Die Objekte stellen dabei die möglichen Zustände eines Systems dar, und die Relationen beschreiben sein Verhalten.

Einige fundamentale Eigenschaften regelbasierter Definitionen können ganz abstrakt studiert werden, ohne die Art der Objekte oder Regeln zu kennen. Für spezifischere Eigenschaften betrachtet man dann konkrete Objekte – Wörter, Terme (Bäume) oder Graphen – und bestimmte Arten von Regeln.

Regeln sind oft *ausführbar*, und können dann dazu benutzt werden, einen Prototypen für ein System zu erzeugen. Deshalb wurden für Arten von Regeln verschiedene Computer-Sprachen und Werkzeuge entwickelt, die für *rapid prototyping* und als Grundlage für die Entwicklung von praktisch nutzbaren informationsverarbeitenden Systemen dienen können.

Im Kurs *Regelbasierte Systeme* wollen wir die grundlegenden Eigenschaften regelbasierter Systembeschreibungen studieren und *Regelbasierte Sprachen und Werkzeuge* solcher Systeme kennen lernen.

In diesem Kapitel werden wir einige Regeln (mit und ohne Variablen) über Wörtern, Termen und Graphen betrachten und einige grundlegende Eigenschaften kennen lernen.

1.1 Regeln

Regeln gibt es überall in der Informatik – und auch sonst mehr als genug. Wir betrachten in diesem Kurs nur Regeln mit einer präzisen mathematischen Definition. Die "Mütter der formalen Regeln" sind die Schlussregeln

der mathematischen Logik und die Gleichheitsdefinitionen auf Wertemengen (algebraischen Spezifikationen). So definiert

$$\frac{P \quad P \supset Q}{Q}$$

die logische Schlussregel *modus ponens* für Aussagen P und Q, und die Gleichung

$$\forall x \in Item, s \in Stack : pop(push(x, s)) = s$$

beschreibt einen allgemeinen Zusammenhang zwischen den Operationen *push* und *pop* auf Stapeln (*stacks*). Beide Regeln beschreiben zulässige Umformungen von Formeln bzw. Termen, die die Bedeutung der Objekte erhalten.

Abbildung 1.1 fasst den Inhalt des Kurses zusammen. Wir wollen Wörter,

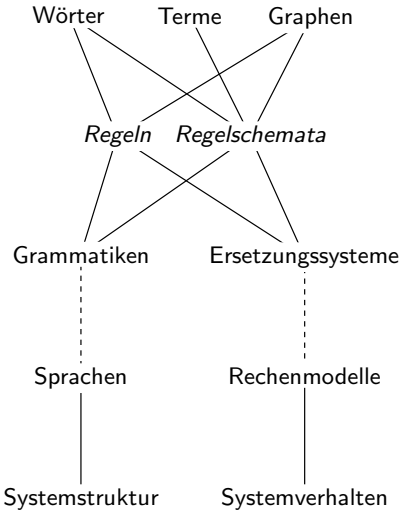

Abb. 1.1. Regelbasierte Systeme un d ihre Anwendungen

Terme und Graphen als Objekte betrachten und mit Hilfe verschiedener Regeln ersetzen. Dabei wird sich herausstellen, dass zwei Arten von Regeln unterschieden werden können: einfache Regeln und Regel-Schemata, die Variablen (bzw. Parameter) enthalten, die bei der Anwendung instantiiert werden können. Sowohl Regeln als auch Regelschemata können zu zwei verschiedenen Zwecken benutzt werden: Als Grammatik zur Definition einer *Sprache* von Objekten, und als Reduktionsssysteme, die Modelle für das Rechnen auf Objekten definieren.

Im ersten Teil des Kurses untersuchen wir Grammatiken und Reduktion abstrakt, also ohne die Art der Objekte und Regeln festzulegen. Danach stu-

dieren wir Wörter, Terme, Graphen mit entsprechenden Begriffen von Regeln, Regelschemata, Grammatiken und Reduktionssystemen.

1.2 Wortgrammatiken

Die Beschreibung "formaler Sprachen" ist einer der ältesten Zweige der theoretischen Informatik. Die von Noam Chomsky in [Cho56] vorgeschlagenen Grammatiken haben dieses Feld geprägt und *kontextfreie* und *reguläre* Grammatiken werden bei der Beschreibung und Implementierung von Programmiersprachen auch praktisch eingesetzt.

Beispiel 1.1 (Eine kontextfreie Grammatik). Die Sprache $a^n b^n$ bezeichnet die Menge aller Wörter aus den Zeichen a und b, in denen einer Folge aus Zeichen a immer gleich viele Zeichen b folgen. Diese Sprache kann mit den kontextfreien Regeln

$$Z \to A \quad \text{und} A \to ab \quad \text{und} \quad A \to aAb$$

beschrieben werden. Dabei sing Z und A Nichtterminale und Z das Startsymbol der Grammatik, auf das die Regeln beliebig oft angewendet werden dürfen, bis kein Nichtterminal mehr auftritt:

$$Z \Rightarrow A \Rightarrow aAb \Rightarrow aaAbb \Rightarrow \cdots \Rightarrow a^{n-1}Ab^{n-1} \Rightarrow a^n b^n$$

Die Syntaqx von Programmiersprachen kann mit kontextfreien Grammatiken aber nicht vollständig definiert werden. Das Programm

$$\textbf{program } P \textbf{ var } x\text{: Integer; } \textbf{begin } x := y \textbf{ end}.$$

ist beispielsweise kein gültiges PASCAL-Programm, weil der Bezeichner y nicht vereinbart ist. Die Eigenschaft, dass jeder Bezeichner vereinbart werden muss, bevor er benutzt wird, ist *kontextabhängig*: Die Gültigkeit einer Zuweisung hängt davon ab, welche Vereinbarungen vor ihr stehen. Solche Eigenschaften können prinzipiell zwar mit kontextsensitiven Chomsky-Regeln beschrieben werden, deren linke Seiten mehrere Symbole enthalten dürfen. Dies hat sich aber als nicht praktiabel herausgestellt.

Für kontextabhängige Eigenschaften einer Sprache können statt dessen *Zweistufengrammatiken* verwenden. Eine Zweistufengrammatik hat kontextfreie *Regelschemata*, deren Nichtterminale keine Symbole, sondern selber Wörter über terminalen und nichtterminalen *Metasymbolen* sind. Kontextfreie Metaregeln definieren, wie aus Meta-Nichtterminalen *Meta-Wörter* abgeleitet werden können.

Die Regelschemata werden instanziiert, indem die in ihnen auftretenden Meta-Nichtterminale durch jeweils das gleiche Meta-Wort ersetzt werden. Mit der so entstehenden Menge von instantiierten Regeln, die im Allgemeinen unendlich ist, können dann die Wörter der zu beschreibenden Sprache abgeleitet werden.

Beispiel 1.2 (Eine Zweistufen-Grammatik). Die Wörter der Sprache $a^n b^n c^n$ bestehen aus gleich langen Sequenzen der Zeichen a, b und c. Diese Sprache kann nicht mit kontextfreien Grammatiken beschrieben werden. Die Metaregeln der Zweistufengrammatik beschreiben die Länge der Teilsequenzen.

$$N \to one \quad N \to one\ N$$

Die Regelschemata für das Ableiten der Teilsequenzen enthalten das Meta-Nichtterminal N und werden so benutzt, dass die abgeleiteten Zeichen "gezählt" werden.

$$\begin{aligned}
\langle start \rangle &\to \langle N\ a \rangle \langle N\ b \rangle \langle N\ c \rangle \\
\langle one\ a \rangle &\to a \qquad \langle one\ N\ a \rangle \to a \langle N\ a \rangle \\
\langle one\ b \rangle &\to b \qquad \langle one\ N\ b \rangle \to b \langle N\ b \rangle \\
\langle one\ c \rangle &\to c \qquad \langle one\ N\ c \rangle \to c \langle N\ c \rangle
\end{aligned}$$

Eine Ableitung mit den instantiierten Regelschemata sieht so aus:

$$\begin{aligned}
\langle start \rangle &\Rightarrow \langle one^n\ a \rangle \langle one^n\ b \rangle \langle one^n\ c \rangle \\
&\Rightarrow a \langle one^{n-1}\ a \rangle \langle one^n\ b \rangle \langle one^n\ c \rangle \\
&\Rightarrow^* a^n \langle one^n\ b \rangle \langle one^n\ c \rangle \\
&\Rightarrow^* a^n b^n c^n
\end{aligned}$$

Regeln über Wörtern können auch benutzt werden, um Ersetzungssysteme zu definieren (*string rewriting*). Das werden wir aber in diesem Kurs nicht weiter behandeln.

1.3 Termersetzung

Terme sind Ausdrücke über Funktionsnamen, Konstantennamen und Variablen, wobei für jeden Funktionsnamen eine Anzahl von Parametern festgelegt ist. Oft erlaubt man Infixschreibweise für Funktionsnamen wie "+" und benutzt dann Klammern, um die Zuordnung von Argumenten zu Operationen zu klären.

Beispiel 1.3 (Fibonacci-Zahlen). Die folgenden Termersetzungsregeln berechnen Summen und Fibonnacci-Zahlen, wobei eine natürliche Zahl n als Successor-Term der Form $S^n(0)$ dargestellt wird:

$$\begin{aligned}
x + 0 &\to x \\
x + S(y) &\to S(x + y) \\
\mathit{fib}(0) &\to 0 \\
\mathit{fib}(S(0)) &\to S(0) \\
\mathit{fib}(S(S(x))) &\to \mathit{fib}(S(x)) + \mathit{fib}(x)
\end{aligned}$$

Termersetzungsregeln sind Schemata: Sie enthalten Variablen, die Platzhalter für beliebige Terme sind. Sie werden benutzt, um Terme zu *reduzieren*,

d.h. Regeln anzuwenden, solange das möglich ist. Terme, auf die keine Regeln angewendet werden können, heißen *Normalformen*. Betrachten wir ein Beispiel:

$$\begin{aligned} \mathit{fib}(S(S(S(0)))) &\Rightarrow \mathit{fib}(S(S(0))) + \mathit{fib}(S(0)) \\ &\Rightarrow \mathit{fib}(S(0) + \mathit{fib}(0)) + \mathit{fib}(S(0)) \\ &\Rightarrow (S(0) + \mathit{fib}(0)) + \mathit{fib}(S(0)) \\ &\Rightarrow (S(0) + 0) + \mathit{fib}(S(0)) \\ &\Rightarrow S(0) + \mathit{fib}(S(0)) \\ &\Rightarrow S(0) + S(0) \\ &\Rightarrow S(S(0) + 0) \quad \Rightarrow \quad S(S(0)) \end{aligned}$$

Bei dieser Reduktion haben wir immer *links außen* eine Regel angewendet. Bei den vorliegenden Regeln liefert jede andere Reihenfolge der Regelanwendung immer die gleiche Normalform, denn das vorliegende Regelsystem ist *terminierend* und *konfluent*. Dann führt jede Ersetzungssequenz zu einer Normalform, und alle Sequenzen für einen bestimmten Term errechnen immer eine eindeutige Normalform.

Termersetzung ist eine Grundlage moderner funktionaler Sprachen. Die Funktionen werden wie Termersetzungsregeln (in einer eingeschränkten Form) definiert und wie bei der Termersetzung ausgeführt. Intern werden die Terme jedoch als Graphen dargestellt; dazu später mehr.

Formeln der Aussagenlogik und Prädikatenlogik sind eine Verallgemeinerung von Termen. Die Regeln in den "Computer-nahen" Formen der Logik werden als Klauseln (Implikationen) formuliert, die ähnlich ausgewertet werden wie Terme. Auch dazu später mehr.

Regeln auf Termen könnten auch verwendet werden, um mit Grammatiken *Baumsprachen* zu generieren. Damit werden wir uns aber nicht beschäftigen.

1.4 Graphtransformation

Graphen bestehen aus Knoten und Kanten. Darauf können sich alle einigen, die sich mit Graphen beschäftigen. Danach beginnen die Unterschiede:

- Sind Kanten gerichtet oder nicht? Sind Schleifen erlaubt? (Schleifen verbinden einen Knoten mit sich selbst.) Sind parallele Kanten erlaubt? (Parallele Kanten verbinden die gleichen Knoten, ggf. in der gleichen Richtung.)
- Sollen Knoten und/oder Kanten markiert sein? Oder getypt? Oder mit Werten attributiert?

Dies sind nur einige Fragen, über die nicht so leicht Einigkeit erzielt werden kann, und die deshalb von Fall zu Fall entschieden werden.

Transformationsregeln für Graphen beschreiben zulässige Manipulationen eines Graphen. Als Beispiel betrachten wir zunächst eine Grammatik.

Beispiel 1.4 (Flussdiagramme). In Abbildung 1.2 zeigen wir Transformationsregeln, die wohlstrukturierte Kontrollflussdiagramme erzeugen. Kleine runde Knoten sind Programmzustände, und eckige Knoten stellen Zuweisungen, Verzweigungen und Nichtterminale dar. Pfeile repräsentieren den Kontrollfluss.

In Abbildung 1.3 ist die Ableitung eines wohlstrukturierten Kontrollflussdiagramms mit diesen Regeln zu sehen. In unserem Beispiel sind die Knoten der Graphen getypt und markiert (oder handelt es sich hier um Attributwerte vom Typ *String*?); Kanten sind gerichtet, aber als Schleifen oder parallel treten sie nicht auf.

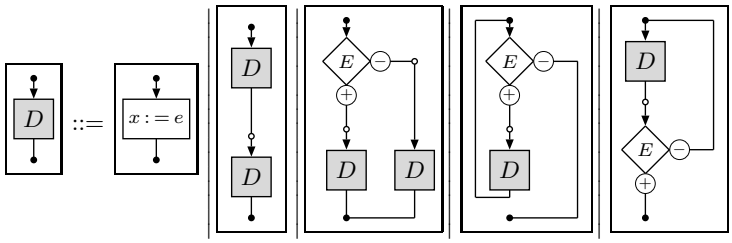

Abb. 1.2. Graph-Regeln für wohlstrukturiere Kontrollflussdiagramme

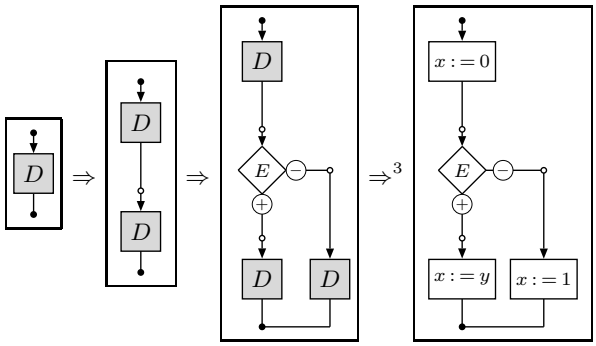

Abb. 1.3. Ableitung eines Kontrollflussdiagramms

Beispiel 1.5 (Transformation von Flussdiagrammen). In Abbildung 1.4 zeigen wir eine Transformationsregel, die ein wohlstrukturierte Kontrollflussdiagramme vereinfacht: Verzweigungen, deren Markierung TT ist, können eliminiert werden. Abbildung 1.5 zeigt eine Anwendung dieser Transformationsregel.

Die Regel enthält zwei Variablen D_1 und D_2, die Platzhalter für beliebige Kontrollflussdiagramme sind. Sie ist also ein Schema.

Transformation von Graphen ist längst nicht so bekannt wie Wortgrammatiken und Termersetzung. Dies liegt sicher auch daran, dass sie schwerer zu implementieren sind, weil sie visuell sind. Auch ist der Algorithmus für die Regelanwendung im allgemeinen aufwändig.

Trotzdem gewinnt dieses Feld an Bedeutung, nicht zuletzt, weil auch viele Computersysteme mitlerweile Diagramme als Benutzerschnittstellen haben und Diagrammsprachen wie UML in der Softwaretechnik immer wichtiger werden.

Einige Anwendungen der Graphtransformation sind:

- Jede Datenstruktur lässt sich abstrakt als Graph verstehen. Das ist nicht nur nützlich zur Programm-Visualisierung, sondern kann auch dazu dienen, die Struktur komplexer Daten, wie blattverbundener Bäume, zyklisch verkettete Listen oder gitterartige Strukturen, mit Graphgrammatiken zu beschreiben, und zu überprüfen, ob Operationen auf den Daten diese Struktur bewahren. Diese sogenannte *shape analysis* ist ein Forschungsfeld des Übersetzerbaus.
- *Model-driven Architecture* ist ein *buzzword* in der Softwaretechnik. Da Modelle graphartige Diagramme sind, können auch hier Grammatiken benutzt werden, um die wohlgeformten Modelle zu beschreiben. Und Transforma-

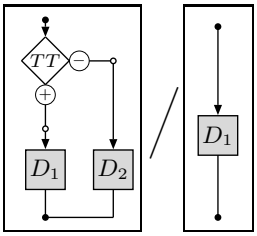

Abb. 1.4. Transformationsregel für Kontrollflussdiagramme

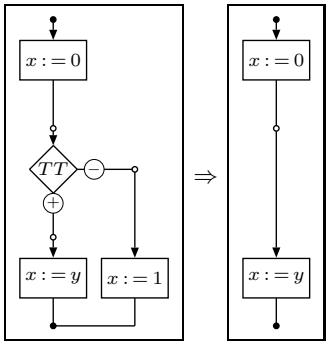

Abb. 1.5. Transformation eines Kontrollflussdiagramms

tionsregeln können dazu benutzt werden, zulässige Modellveränderungen zu beschreiben und zu implementieren.

Literaturhinweise

Es gibt viele Lehrbücher zu abstrakten und konkreten regelbasierten Systemen.

- Das Lehrbuch *Term rewriting and all that* von Franz Baader und Tobias Nipkow [BN98].
- Ein Übersichtsartikel zu Termersetzung von Jan Wilem Klop [Klo92].
- Das Handbuch [tMBKdV03] der TERESE-Gruppe zu Termersetzungssystemen.
- Bände I *Foundations* und II *Applications, Languages, and Tools* des Graphtransformations-Handbuchs [Roz97, EEKR99].
- Das Buch über *Algebraischen Graphtransformation* [EEPT06].

Der Inhalt dieser Bücher kann natürlich nicht einmal annähernd im Kurs behandelt werden; sie eignen sich aber zum Nachschlagen und zur Vertiefung. Andererseits werden wir im Kurs auch Konzepte besprechen, die in diesen Büchern nicht zu finden sind. Dazu gibt es in jedem Kapitel noch weitere Literaturangaben.

(Dies ist Fassung 1.1 von 26. Oktober 2010.)