Vorlesung

Sommersemester 2007

Semantikanalyse-Verfahren

Universität Zürich

Institut für Computerlinguistik

Prof. Dr. Michael Hess

1. Einleitung

1.1 Beschreibung der Vorlesung

Gehört zu den ‘‘Grundlegenden Vorlesungen’’

1. *Art:* Reines Selbststudium (mit E-Learning-Unterstützung); keine Präsenzveranstaltungen.
2. *Voraussetzungen:* Im Prinzip ECL I, ECL II sowie ‘‘Formale Grammatiken und Syntaxanalyseverfahren’’ und ‘‘Formale Grundlagen der Linguistik’’
3. *Inhalt:* Probleme der *automatischen Semantikanalyse*, und zwar anhand der Frage, wie man konkret

Inhaltlich: ± Fortsetzung von ‘‘ECL II’’

- aus deklarativen Eingabesätzen eine (intensionale) Datenbank aufbaut
- aus interrogativen Eingabesätzen Anfragen an diese Datenbank erzeugt
- diese Anfragen vom System sinnvoll beantworten lässt

Die Problemkreise im einzelnen:

1. Rolle der Semantik in der Computerlinguistik
2. Grundkonzepte der logikbasierten Fragenbeantwortung
3. Beziehung zwischen Syntax und Semantik in der natürlichen Sprache
4. Grenzen der Kompositionalität
5. Grenzen der wahrheitswertorientierten Semantik
6. Grenzen der Semantik erster Stufe
7. Grenzen der Extensionalität
8. Grenzen der naiven Ontologie

1.2 Ressourcen

1.2.1 Literatur

Literatur: Lage wenig befriedigend

Es gibt noch keine kommerziell erhältliche Gesamt-Übersicht über die konkreten Lösungsansätze für semantische Probleme in der Computerlinguistik, aber einige gute noch unveröffentlichte Texte zu ‘‘Computational Semantics’’ (auf dem WWW erhältlich), einige gute Einführungen in die Formale Semantik, eine Reihe von bewährten Werken zur Logik, sowie (etwas verstreut) einiges zu Implementationsfragen:

1. Formale Semantik allgemein (wiederholt aus ECL II)

das beste

   a. G. Chierchia, S. McConnell-Ginet, Meaning and Grammar, MIT Press, Cambridge, MA, 1990

   b. Ronnie Cann. Formal Semantics. Cambridge University Press, Cambridge, 1993

   c. L.T.F. Gamut, Logic, Language, and Meaning, Volume 1: Introduction to LogiçThe University of Chicago Press, Chicago, 1991

      L.T.F. Gamut, Logic, Language, and Meaning, Volume 2: Intensional Logic and Logic Grammar, The University Of Chicago Press, Chicago, 1991

   d. André Thayse, ed., From Standard Logic to Logic Programming, Wiley, Chichester, England, 1988

      André Thayse, ed., From Modal Logic to Deductive Databases, Wiley, Chichester, England, 1989

      André Thayse, ed., From Natural Language Processing to Logic for Expert Systems,Wiley, Chichester, England, 1991

   e. J.N. Martin, Elements of Formal Semantics, Academic, Orlando etc., 1987

2. Montague-Semantik (z.T. wiederholt aus ECL II)

hier das beste

   a. Horst Lohnstein, Formale Semantik und natürliche Sprache: Einführendes Lehrbuch, Westdeutscher Verlag Wiesbaden, 1996

   b. D.R. Dowty, R.E. Wall, S. Peters, Introduction to Montague Semantics,Reidel, Dordrecht/Boston/London, Synthese Language Library, 11, 1981

c. T.M.V. Janssen, Foundations and applications of Montague grammar; Part 1: Philosophy, framework, computer science, Centre for Mathematics and Computer Science, Amsterdam, CWI TRACT Nr. 19, 1986

T.M.V. Janssen, Foundations and applications of Montague grammar; Part 2: Applications to Natural Language, Centre for Mathematics and Computer Science, Amsterdam, CWI TRACT Nr. 28, 1986

d. G. Link, Montague-Grammatik, Fink, München, Kritische Information Nr. 71, 1979

3. Mathematische und logische Grundlagen (wiederholt aus ECL II):

zum Nachschlagen

a. B. H. Partee A. ter Meulen, R. Wall, Mathematical Methods for Linguists,Reidel, Dordrecht, Studies in Linguistics and Philosophy, 1990

b. J. Allwood, L.-G. Andersson, O. Dahl, Logic in Linguistics, Cambridge U.P, London/New York/Melbourne, Cambridge Textbooks in Linguistics, 1979

c. J.D. McCawley, Everything that Linguists have Always Wanted to Know about Logic, Blackwell, Oxford, 1981

4. Implementationsfragen

recht gut; spezifisch

a. M.A. Covington, Natural Language Processing for Prolog Programmers, Prentice Hall, Englewood Cliffs, N.J., 1994

sehr gut; mässig spezifisch

b. Daniel Jurafsky und James H. Martin, Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics and Speech Recognition,Prentice Hall,Upper Saddle River, NJ, 2000; v.a. Kapitel 15.1 - 15.3

c. Allen, J., Natural Language Understanding, Benjamin/Cummings, Menlo Park etc., 1995; v.a. Kapitel 9.1

d. Manfred Pinkal, Semantik, in: Günther Görz (Hrsg.), Einführung in die künstliche Intelligenz, Addison-Wesley, Bonn, Kap. 5.3, pp. 425-498, 1993

immer noch *sehr* aktuell

e. F.C.N. Pereira, S.M. Shieber, Prolog and Natural Language Analysis, Center for the Study of Language and Information, Menlo Park/Stanford/Palo Alto, CSLI Lecture Notes, 10, 1987; on-line erhältlich ⇒ entfernt und ⇒ lokal.

f. Robert C. Moore, Logic and Representation, Center for the Study of Language and Information, CSLI Lecture Notes 39, 1995

g. G. Gazdar, Ch. Mellish, Natural Language Processing in Prolog, Addison-Wesley, Wokingham etc., 1989

h. Annie Gal, Guy Lapalme, Patrick Saint-Dizier, Harry Somers, Prolog for Natural Languag Processing, Wiley, Chichester etc., 1991

1.2.2 Internet

On-line Dokumente am wichtigsten!

1. Unveröffentlichte *Übersichtstexte* (auf dem WWW erhältlich)

   a. ⇒Patrick Blackburn and Johan Bos, Representation and Inference for Natural Language A First Course in Computational Semantics, Vol. 1: Working with First-Order Logic und Vol. 2: Working with Discourse Representation Structures

   b. Robin Cooper, Ian Lewin and Alan Black: Prolog and Natural Language Semantics; Notes for AI3/4 Computational Semantics, 1993 ⇒hier

   c. H. Bunt & R. Muskens, Computational Semantics: An Introduction. Computing Meanings, H. Bunt & R. Muskens (eds.), Kluwer Academic Press, Dordrecht 1999, 1-31. Auch ⇒hier verfügbar.

   d. Robin Cooper, Dick Crouch, Jan van Eijck, Chris Fox, Josef van Genabith, Jan Jaspars, Hans Kamp, Manfred Pinkal, Massimo Poesio, Steve Pulman, Evaluating the State of the Art; A Framework for Computational Semantics, LRE 62-051 Deliverable D10, January 3, 1995: ⇒hier

   e. Robin Cooper, Richard Crouch, Jan van Eijck, Chris Fox, Josef van Genabith, Jan Jaspers, Hans Kamp, Manfred Pinkal, Massimo Poesio, Stephen Pulman, Espen Vestre, Describing the Approaches; A Framework for Computational Semantics, LRE 62-051 Deliverable D8, December 1994: ⇒hier

   f. Generell die FRACAS-‘‘Deliverables’’ (mit u.a. den zwei letztgenannten Texten, aber noch einigem mehr): ⇒hier

   g. Zur lexikalischen Semantik siehe auch die Übersicht ⇒hier.

2. Eine Übersicht zur Computerlinguistik und spezifisch der Semantik ist ⇒hier, mit (u.a.) dem Link auf ‘‘Projects in or Related to (Computational) Semantics’’ ⇒hier.
3. Ein Glossar ist ‘‘The Bluffer’s Guide to Computational Semantics‘‘ (1996) ⇒hier.
4. Eine spezifische WWW-Site zu ‘‘Computational Semantics’’ ist ⇒hier zu finden (noch sehr neu; wenig drin).
5. Eine sehr gute und umfassende Linkliste ist ⇒

2. NLP-Anwendungen mit starker semantischer Komponente

> ***It’s a messy field -- but an exciting one.***
>
> *Patrick Blackburn*

In ECL II wurde als bisheriges Paradebeispiel für ein ein System mit starker semantischer Komponente *Chat-80* vorgestellt.

relativ (!) einfach

Aber: Chat-80 war atypisch: (oder: sehr vereinfacht)

1. keine Dialogkomponente
2. statische Welt
3. bloss ein einziger Typ von Eingabe
4. kaum lexikalische Beziehungen
5. sehr einfache zugrundeliegende Welt

wirklich schwierig

Weit anspruchsvoller sind:

- Dialogsysteme
- textverstehende Systeme

2.1 Erstes Beispiel: Dialogsysteme

Ein grosser ‘‘Klassiker’’...

SHRDLU: Dialog mit einem (virtuellen) Roboter über eine (virtuellen) Mikro-Welt. Die Mikro-Welt ist *extrem* eingeschränkt, aber

1. die Welt ist *dynamisch*
2. man führt einen ganzen *Dialog*
3. mit *allen Satztypen*

SHRDLU ist schon sehr alt, aber in seiner Art immer noch beispielhaft [1] Dissertation von Terry Winograd, ‘‘Understanding Natural Language’’, *1972*.

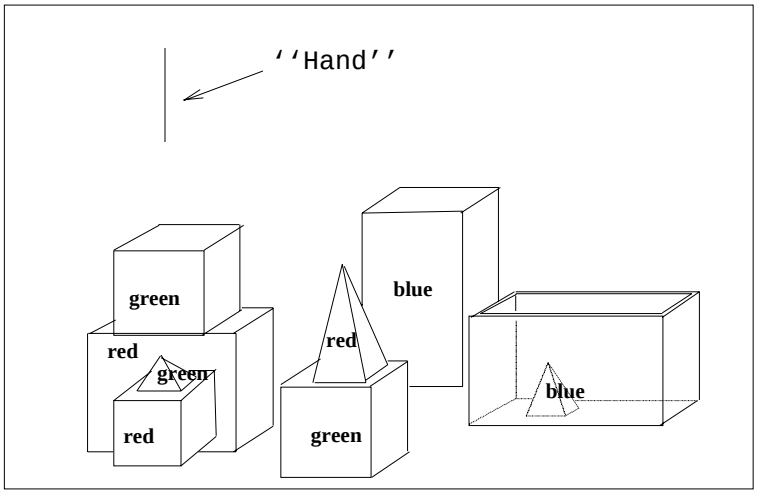
- Lobende Vorbemerkung: Winograd hat fast alle Probleme, die in der Computerlinguistik wichtig sind, erkannt und *irgendwie* gelöst.

... mit Problemen

- Relativierende Vorbemerkungen:
  1. es ist zu vermuten, dass diese Lösungen in den meisten Fällen nicht-verallgemeinerbare (ad-hoc-) Lösungen waren
  2. weil der Interpreter für den entsprechenden LISP-Dialekt verschollen ist, ist das aber kaum mehr zu verifizieren (im *Prinzip* schon - der ⇒Programmcode von SHRDLU *ist* noch vorhanden; es wurden einige mässig erfolgreiche Vesuche unternommen, das System wieder zum Leben zu erwecken; siehe ⇒ hier.)
  3. die Wirkung des Programms relativ beschränkt

Als Illustration der Probleme, die von einem Frage-Antwort-System gelöst werden müssen, taugt das System aber in jedem Fall. ***Ausgangssituation:***

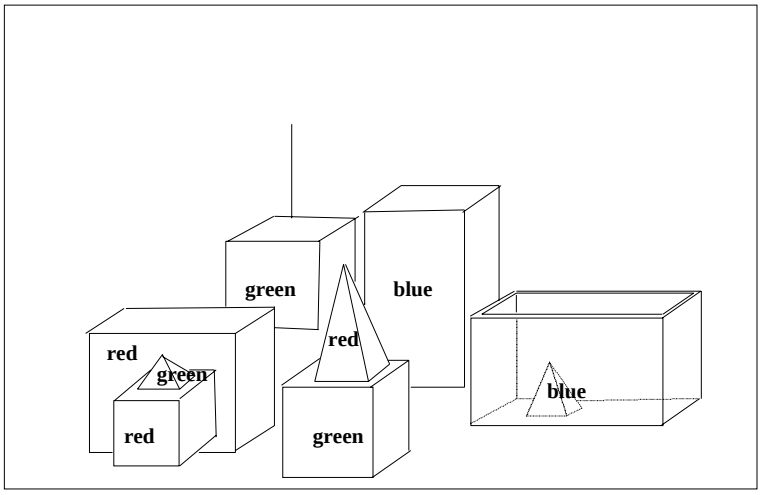
1. nach Winograd 1972; Kommentare nach Tennant 1981:142-163

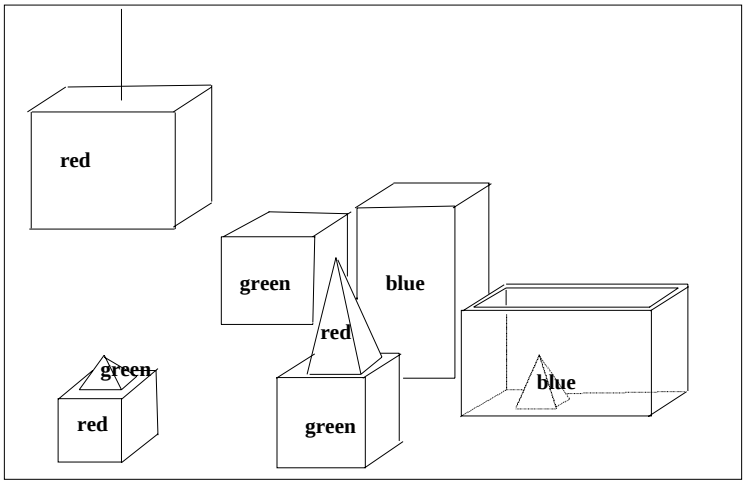


Ist das *der* Beispieldialog?!

Beispieldialog: (Fettdruck: Benutzer; Courier: SHRDLU)

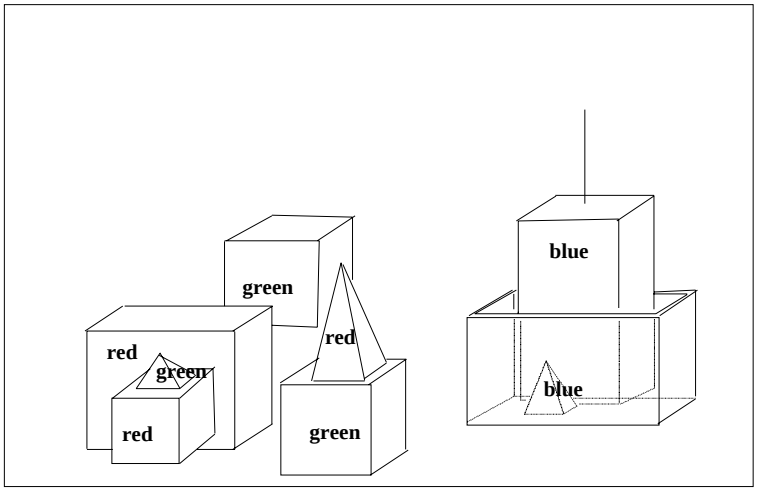
Pick up a big red block!
Ok.





Grasp the pyramid!
I don't understand which pyramid you mean.

Find a block which is taller than the one you are holding and put *it* into the box!
By ''it'' I assume you mean the block which is taller than the one I am holding.



Ok

What does the box contain?

The blue pyramid and **the** blue block.

What is *the* pyramid supported by?
The box.

How many blocks are not in the box?
Four of them.

Is at least one of them narrower than the one which I told you to pick up?
Yes, the red cube.

Is it supported?
Yes, by the table.

Can the table pick up blocks?
No.

Can a pyramid be supported by a block?
Yes.

Can a pyramid support a pyramid?
I don't know.

Stack up two pyramids!
I can't.

The blue pyramid is mine.
I understand.

I own blocks which are not red, but I don't own anything which supports a pyramid.
I understand.

Do I own the box?
No.
Do I own anything in the box?
Yes, two things: The blue block and the blue pyramid.

Will you please stack up the red blocks and either a green cube or a pyramid?
Ok.

Which cube is sitting on the table?
The large green one which supports the red pyramid.

Is there a large block behind a pyramid?
`Yes, three of them: A large red one, a large green cube and the blue one.`

Put a small one onto the green cube which supports a pyramid!
`Ok.`

SHRDLUs Welt mag als Situation sehr akademisch erscheinen. Aber es zeigt sich, dass manche der hier angetroffenen Probleme in sehr viel anwendungsnäheren Situationen ebenfalls vorkommen: In textverstehenden Systemen.

2.2 Zweites Beispiel: Textverstehende Systeme

Der Bedarf an textverstehenden Systemen ist ausgewiesen. Handbuchtexte als Beispiel. Man möchte diese z.T. sehr umfangreichen Texte von Maschinen

1. so weit ‘‘verstehen’’ lassen,
2. dass man sie nachher z.B. dazu befragen kann,
3. und zwar, wenn möglich, in natürlicher Sprache.

Handbücher: Wichtige Textsorte

Beispieltext:[2]

`All CONCEPT terminal commands are invoked by control codes. A number of control code commands, such as Return, Back Space, and Line Feed, were described above. A complete list of the 32 control codes and the commands that they invoke is given in Appendix E.`

Darüber möchte man Fragen stellen können wie

- How do I use terminal commands on the CONCEPT?

2. Benutzerhandbuch des Terminals ‘CONCEPT 108’.

- Which control codes invoke commands?
- What is the list of control codes?
- usw.

Es gibt kein ‘‘Textverstehen’’ *tout court*

Es gibt verschiedene Stufen von ‘‘Textverstehen’’ und ‘‘Befragen’’:

Istzustand: ‘‘Document retrieval’’ (fälschlicherweise oft ‘‘*Information* Retrieval’’ genannt) plus Hervorheben der Stichwörter (also ± das, was z.B. Google macht).

Repetition Dokumentenretrieval:

Zuerst: Eine *Vor*-Vorstufe des Textverstehens

1. automatisches Indexieren von *ganzen* Dokumenten
2. Lokalisieren von (ganzen!) Dokumenten nach (Kombinationen von) Stichwörtern (‘‘Kombinieren’’ kann dabei Verschiedenstes heissen: reine Boolésche Kombination, Vektorraum-basiertes oder probabilistisches Kombinieren)
3. kein Berücksichten des Inhalts.

Zum Dokumentenretrieval siehe auch Vorlesung ECL1 ⇒hier.

Indexterme ≈ Stichwörter in Dokumenten

Aus dem Dokument oben werden (z.B.) folgende Indexterme erzeugt

```
concept terminal command invoke control code
return back space line feed appendix
```

Daher: Dieses Dokument wird (mit einfachem Boole’schem Retrieval)

- *korrekterweise gefunden* bei den Anfragen

```
? What are the control codes of the CONCEPT terminal
? What are control codes invoking terminal commands
? What is a control code command?
```

Das *möchten* wir eigentlich finden...

- *nicht gefunden* (mangelnde Ausbeute [recall]) bei den Anfragen

```
? How do you run a terminal command on the CONCEPT 108?
? control codes of CONCEPT system
```

Grund:

Im *ersten* Fall: Synonyme wie

run ≈ invoke

werden nicht erkannt.

Im *zweiten* Fall: Hyperonym/Hyponym-Beziehungen wie
terminal ⊂ system

werden nicht berücksichtigt.

Achtung: Natürlich nicht kommutativ!

... das aber eher nicht

- *fälschlicherweise gefunden* (mangelnde Genauigkeit [precision]) bei den Anfragen

```
? control codes invoked by terminal commands
? concept of control code
? number of commands
? code space of the CONCEPT 108
```

Gründe:

1. syntaktische (und damit semantische) Beziehungen zwischen Wörtern ignoriert
2. Eigenname als Common Noun interpretiert
3. Stopwörter vermeintlich gefunden
4. Mehrwort-Terme nicht als solche erkannt
5. satzübergreifend gesucht (d.h. immer ganze Dokumente betrachtet)

Mögliche Verbesserungen des Dokumenten-Retrieval:

1. ‘‘Latent semantic indexing’’
2. Verwendung eines Thesaurus
3. **Oder aber:** Linguistische Ansätze: mehr von der linguistischen (v.a. syntaktischen) Information verwenden.

Aber: Der Nachteil der zu grossen Informationseinheit (= Dokument) wird damit nicht angegangen. Daher:

2.2.1 Drei Vorstufen des Textverstehens

Zwischenziele (manchmal: ‘‘Intelligent text retrieval’’)

1. Passagenretrieval
2. Antwortextraktion
3. Antwortkomposition

etwas näher am Textverstehen

Passagenretrieval

- Lokalisieren eines willkürlich grossen, fixen Text*ausschnittes*(*n* Zeilen oder *n* Zeichen),
- in welchem die gesuchte textuelle Information *enthalten* sein soll.

Heute meist verwendete **Methode** beim Passagenretrieval: Die bekannten Stichwortverfahren des IR (oft mit ein wenig ‘‘Stemming’’ verfeinert).

Beispiele: (Annahme: ‘‘Fenster’’ ist exakt eine Zeile)

- Aufgabe: ‘‘How do I use terminal commands on the CONCEPT?’’

  Anfrage: `terminal & command & concept`

  Antwort: 1. Zeile des obigen Texts (*korrekt*)
- Aufgabe: ‘‘Which control codes invoke the terminal?’’

  Anfrage: `control & code & invoke & terminal`

  Antwort: 1. Zeile des obigen Text *(falsch!)*

Probleme:

Willkürliche Fenstergrösse führt zu ...

1. Wenn stichwortbasiert: Noch weniger präzis, als das Beispiel vermuten lässt
2. Passage ist *willkürlicher* Ausschnitt, daher oft irreführend.

(Reales, anderes) **Beispiel:** Das Dokument enthält den Satz

Users do not need write permission to remove a symbolic link, provided they have write permissions in the directory.

und die Frage ist:

Do I need write permissions to remove a symbolic link?

...u.U. ganz falschen Resultaten.

Mit einem 50-Byte ‘‘Antwortfenster’’ findet man z.B.

```
" need write permission to remove a symbolic link, "
```

was aktiv irreführend ist.

***Daher:* Antwortextraktion**

ein bisschen ambitiöser, aber immer noch machbar

- Lokalisieren eines ganzen Satzes (beliebiger Länge),
- der die explizite Antwort ist oder enthält.

Zweites heisst: Finden von u.U. unzusammenhängenden) Fragmenten eines Satzes, die *zusammengehängt* einen grammatikalisch korrekten Satz ergeben.

Beispiel:

- Anfrage: ‘‘Which control codes invoke commands?’’

  Antwort: 1. Satz (korrekt)
- Anfrage: ‘‘Which control codes are invoked by terminal commands?’’

  Antwort: "keine Information" (korrekt)
- Anfrage: ‘‘What is the list of control codes?’’

  Antwort: 3. Satz (streng genommen falsch: Liste ist hier bloss *erwähnt*, nicht *verwendet*)

Hier muss man

1. extensive Syntaxanalyse
2. partielle Semantikanalyse
3. ein wenig Textanalyse

betreiben.

Antwortkomposition:

Ein weiterer Zwischenschritt, und...

1. Fragmente aus u.U. verschiedenen Sätzen lokalisieren, die
2. kombiniert (ggf. morphologisch/syntaktisch adaptiert) einen grammatikalisch korrekten Satz ergeben,

3. welcher die Antwort ist.

- Anfrage: ‘‘What are the CONCEPT control codes?’’

  Antwort: ‘‘All CONCEPT control codes are given in Appendix E’’

2.2.2 Das Endziel: Frage-Antwort-Systeme über Texten

...das Endziel.

Ein **textbasiertes ‘‘Frage-Antwort-System:**

1. findet im Text beschriebene *Fakten*,
2. u.U. nur *implizit* beschriebene,
3. und *erzeugt* daraus die Antwort.

Hier wird’s nun wirklich ambitiös

- Problem: Ausführen von im Text eingeführten Regeln erforderlich
- Problem: dazu *muss* man ganze Sätze verstehen

Beispiele:

- Anfrage: ‘‘How many terminal commands are there on the CONCEPT?’’

  Antwort: ‘‘At most 32’’
- Anfrage: ‘‘Give me the list of control codes!’’

  Antwort: <druckt Liste aus Appendix E aus>

Grundsätzliche **Anforderungen** an Frage-Antwort-System:

1. Text muss in eine geeignete Wissensrepräsentation übersetzt und in einer Wissensbasis gespeichert werden
2. Fragen müssen in eine vergleichbare Repräsentation überführt werden
3. Antworten müssen entweder
   a. in der Wissensbasis *nachgeschaut* werden

b. oder *abgeleitet* werden

3. Semantische Repräsentation als ein zentrales Problem

Für die beschriebenen Leistungen benötigt man eine semantische Repräsentation für eine u.U. dynamische sich verändernde Situation (Situationswissen), für Weltwissen und für Sprachwissen.

Aus Gründen der Machbarkeit werden wir und konzentrieren auf die noch relativ bescheidenene Anfomderungen, die ein Antwortextraktionssystem stellt.

Zusammenfassung der wichtigsten Probleme *aller* natürlichsprachlichen Systeme (z.T. wiederholt aus ECL II)

1. Syntaktische Variabilität der natürlichen Sprache

```
‘‘the commands that the control codes invoke’’ ≈
‘‘commands invoked by control codes’’ ≈
‘‘control code commands’’

‘‘a country bordering the Mediterranean and the Baltic’’ ≈
‘‘a country that borders ...’’ ≈
‘‘ ... bordering the Baltic and the Mediterranean’’
```

Vagheit ≠ Ambiguität

2. Vagheit:
   ‘‘*a number of* control code commands’’

   ‘‘... bordering African and Asian countries’’

wie immer ein (Riesen-)Problem

3. Ambiguität:
   a. Komposita (resp. ‘‘Reihungen’’)
      - **‘‘control code command’’**
        a. ((control code) command): a command *invoked by* a control code
        b. (control (code command)): a code command *to* control sth. (?)

- **‘‘computer model’’**
  a. ein Modell eines Computers
  b. ein Modell realisiert auf einem Computer

b. Anschlussmehrdeutigkeiten etc. etc.

oft unterschätzt

4. Sprechaktvielfalt
   - Aufforderung:
     Upon receipt of the CONCEPT terminal and before unpacking it, the user *should* inspect it for any damage to the external packaging material
   - implizite Aufforderung:
     Figure 1-1 *can* be folded out from the front cover for convenient reference while reading the following sections
   - Versprechen:
     Human Designed Systems *warrants* that each terminal will be free from defective materials and workmanship for ninety days from date of shipment to the original customer

Daher erforderlich:

1. Ermitteln des Sprechaktyps
2. Extrahieren des ‘‘propositionalen Kerns’’, und Übersetzen in eine
   a. (±) kanonische
   b. explizite
   c. eindeutige Sprache

Anforderungen an eine semantische Repräsentation

3. Auswertung der resultierenden Ausdrücke
4. Reaktion erzeugen

Punkt 2 ist der Schwerpunkt der semantischen Analyse.

Zwei Phasen in der computerlinguistischen Behandlung dieses Problems

1. bis ca. 1980: experimentierendes Vorgehen
   a. Vielfalt von speziellen Repräsentations-Sprachen für Textinhalte und Fragen

   b. Übersetzung von Texten und Fragen sehr ad hoc programmiert
   c. Auswertung (z.B. zur Fragenbeantwortung) aufgrund spezifischer Prozeduren

etwas beschönigend

2. ab ca. 1980 bis heute: *logikbasiertes* Vorgehen
   a. Sprachliche Eingaben werden übersetzt in ihre ‘‘Logische Form’’ (LF) im Sinne der satzsemantischen Analysen der Formalen Semantik
   b. Übersetzung erfolgt (möglichst) *systematisch* und *allgemein*
   c. Auswertung der LF erfolgt mit Hilfe automatisierbarer Inferenzverfahren

Auf das zweite, d.h. die Satzsemantik, werden wir uns im Folgenden konzentrieren.

3.1 Modelltheoretische und beweistheoretische Semantik

Aber: der Begriff ‘‘(Satz-)Semantik’’ ist nicht eindeutig:

1. im Sinne der ‘‘Formalen Semantik’’
2. im Sinne der Computerlinguistik allgemein

3.1.1 Modelltheoretische Semantik

Prototypische Problemstellungen der Formalen Semantik (cf. ECL II)

1. die *Bedeutung eines Satzes* zu verstehen heisst zu wissen, wie die reale Welt aussehen muss, damit der Satz wahr ist (modelltheoretische Auffassung)
2. *Interpretation* von Äusserungen der natürlichen Sprache: Testen des Vorliegens in der *realen* Welt, d.h.
   i. wenn Aussagesatz: Wahrheitswert ermitteln
   ii. wenn Fragesatz: unklar.
   iii. wenn Befehlssatz: ebenfalls unklar.

immer noch verbreitet in der Linguistik

3. es werden in der Regel nur isolierte Sätze betrachtet

Der Satz

1) John will now put the block in the box on the table

kann bekanntlich zweierlei bedeuten:

2) Hans wird jetzt den Klotz in *die* Schachtel auf *dem* Tisch legen

3) Hans wird jetzt den Klotz in *der* Schachtel auf *den* Tisch legen

Diese Entscheidung, welche Lesart die richtige ist, erfordert

1. Ermitteln der *Bedeutung* der zwei Nominalphrasen
   - the box on the table
   - the block in the box

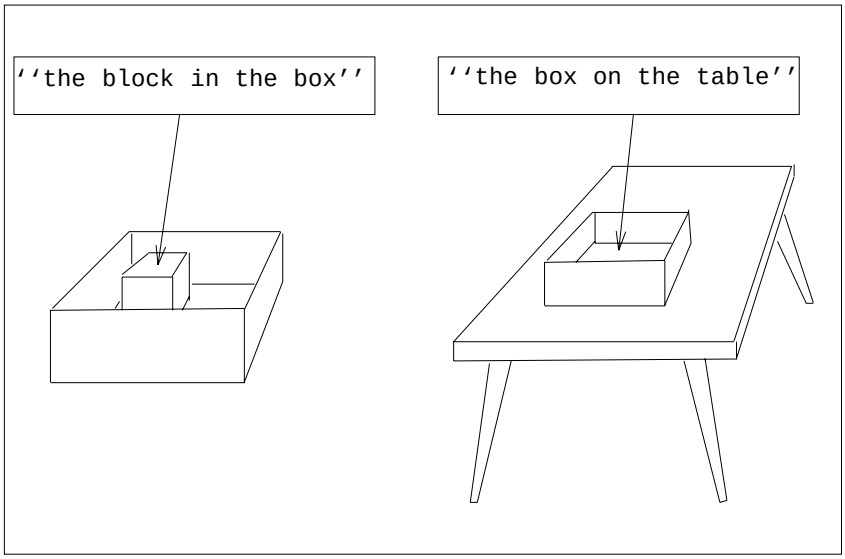
''Blick in die Welt!''

2. und deren *Interpretation:* Welcher der zwei folgenden Situationen liegt vor?
   - eine Schachtel liegt auf dem Tisch
   - ein Klotz liegt in der Schachtel

Das ist, was man bei einem *Gespräch* tun könnte, um die richtige Bedeutung zu ermitteln: In der Welt nachsehen.

''Vorliegen'' heisst also: in der *realen Welt* feststellbar sein. [3]

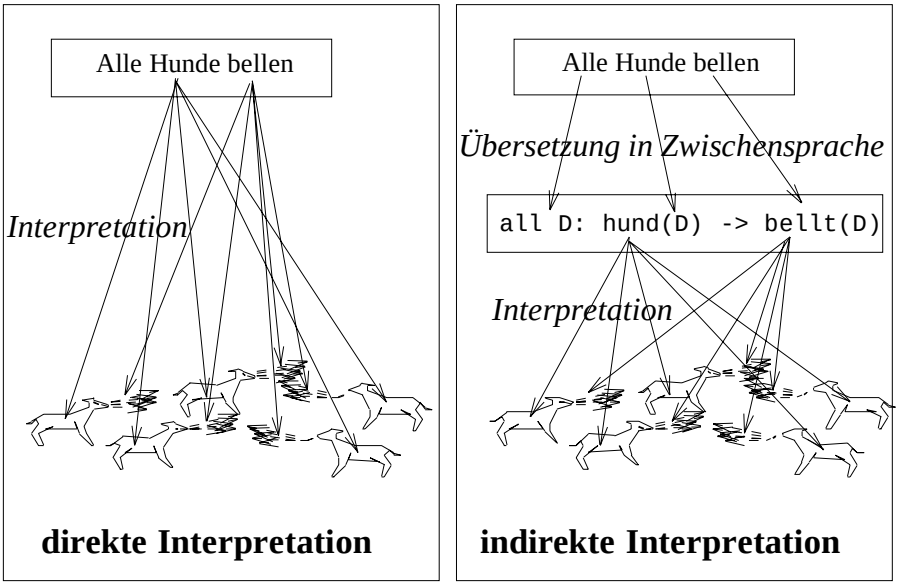
3. **Zu beachten:** Die Mathematiker haben eine etwas abgehobenere Vorstellung von Modelltheorie: Für sie bezieht man sich mit den Ausdrücken einer Sprache nicht auf Objekte der im umgangssprachlichen Sinne realen Welt, sondern auf ''mathematische Strukturen''.



Die Interpretation (also: Feststellen, ob etwas ‘‘vorliegt’’)

‘‘rekursiv’’ ist wichtig!

- muss *rekursiv* aus der Bedeutung der Konstituenten und der Art ihrer Kombination im Satz geschehen
- *könnte* im Prinzip direkt geschehen , in der Praxis aber wird sie (fast immer) via eine Zwischenrepräsentation realisiert



Wichtig: Die reale Welt, *nicht* die Zwischenrepräsentation, ist das Fundament der Semantik (in dieser Auffassung).

Gewisse Arten von Anfragen kann man so behandeln.

Man beachte auch: Datenbankabfragen à la Chat-80sind fast ganz in einem modelltheoretischen Rahmen beschreibbar.

```
continent(africa).
continent(america).
continent(antarctica).
<etc.>

ocean(arctic_ocean).
ocean(atlantic).
<etc.>

in_continent(scandinavia, europe).
in_continent(western_europe, europe).
<etc.>
```

etwas sehr vereinfacht

Die Datenbank entspricht fast 1:1 der zugrundeliegenden Realität. Vielleicht sind natürlichsprachliche Schnittstellen zu DBMSs deshalb so beliebt in der computerlinguistischen Forschung: Man kann sie im gut bekannten klassischen modelltheoretischen Rahmen erfassen.

3.1.2 Beweistheoretische Semantik

Vergleiche dagegen die Situationen, in denen man Dialogsysteme, Textverstehenssysteme à la SHRDLUetc. verwendet: Es gibt keine ‘‘reale’’ Welt!

Die reale Welt ist oft irrelevant.

- Ob die in SHRDLUs Welt vorhandenen Objekte auch in der *realen* Welt *existieren*, ist ganz egal(effektiv ist diese Frage völlig verfehlt - die Mini-Welt von SHRDLU ist explizit erfunden und irreal)
- Ob Aussagesätze *wahr* sind, wird nicht geprüft weder in SHRDLUs Welt noch in der realen Welt

Daher:

Ein sehr anderer Ansatz:

Prototypische Problemstellungen der computerlinguistischen Semantik

1. die *Bedeutung eines Satz* zu verstehen heisst zu wissen, welche Schlussfolgerungen man daraus ziehen darf (beweistheoretische Auffassung)

2. *Evaluation* über dem aus sprachlichem Material aufgebauten symbolischen Weltmodell (statt ‘‘Interpretation’’)
   a. wenn Aussagesatz: assimilieren
   b. wenn Fragesatz: über dem internen Weltmodell beantworten
   c. wenn Befehlssatz: Operation über dem internen Weltmodell ausführen

Und das entspricht sehr genau der Situation beim Verstehen eines (geschriebenen) *Texts*. Derselbe Satz (1), aber nun mit vorhergehendem Text

4) John put the block on the table. He then put the block in the box on the table

kann natürlich ‘‘*im Prinzip*’’ weiterhin auf beide Arten verstanden werden (2, 3):

2) Klotz in *die* Schachtel auf *dem* Tisch

3) Klotz in *der* Schachtel auf *den* Tisch

Aber:

- Der Leser eines Texts kann nicht ‘‘in der Welt nachschauen’’ (er ist nicht selbst in der Äusserungssituation)
- Es ist dennoch ‘‘offensichtlich’’, dass Lesart 2 richtig ist.
- Grund: Die für Lesart 3 vorausgesetzte Situation ist ausgeschlossen, d.h. ‘‘liegt’’ nicht ‘‘vor’’.
- **Aber:** ‘‘Vorliegen’’ heisst hier lediglich, in einem *symbolischen Modell* der Welt feststellbar zu sein.

```
in(block1,box1)
```
resp.
```
on(box1,table1)
```

- Das Modell wird aufgrund des gelesenen Texts aufgebaut (und dauernd nachgeführt).

Und genau diese Situation liegt auch vor beim Beantworten von Fragen über Texten:

- Aufbau eines symbolischen (‘‘mentalen’’) Modells (Assertionen)
- Schlussfolgern darüber (Fragen)
- ohne Berücksichtigung der ‘‘realen Welt’’

Siehe unten.

Also:

- Auch die Modelltheorie verwendet fast immer ‘‘symbolische Modelle der Realität’’ in Form logischer Darstellungen
- Die Semantik im Sinne der Computerlinguistik kann daher bei der Formalen Semantik viele Methoden ‘‘ausleihen’’.
- Sie muss aber umfassender als diese sein.
- **Schön:** Manche der Probleme der Formalen Semantik verschwinden in einem beweistheoretischen Rahmen ± von selbst

3.2 Anwendung auf sprachbasiertes Dokumentenretrieval

Ein typisches Laborsystem

Als Einstieg in den Einsatz linguistischer Methoden im Textverstehen: ‘‘LogDoc’’.

Ziel: Exaktes Finden von Aufsatztiteln im Bereich Computerlinguistik.

Ein Ausschnitt

```
Baseball: An Automatic Question Answerer
Computers and Thought
Logic for Natural Language Analysis Semantics
The Position of Locatives in Existential Sentences
Generalized Quantifiers and Natural Language
A Comprehensive Grammar of the English Language Semantics
Introduction to Montague Semantics
Logic in Linguistics
Introduction to Mathematical Linguistics
A Proficiency Course in English
On the Question of Definiteness in ‘An Old Man’s Book’
English Quantifier Systems
Existential Sentences in English
Referential and Quantificational Indefinites
Specifity and the Interpretation of Quantifiers
De Re Belief Generalized
There-Insertion
Getting more mileage out of existentials in English
```

Vorgehen:

1. Übersetzen der Titel (meist Nominalphrasen und Präpositionalphrasen) in eine semantische Repräsentation (konkret: Prolog)
2. Assertieren in einer logischen Datenbank
3. Übersetzen der Anfragen in dieselbe Repräsentation
4. Die zur Anfrage ‘‘passenden’’ Einträge in der Datenbank finden (‘‘einen Match finden’’)

Der Titel sei

```
A formal specification language for the
automatic design of chips by computer
```

Er wird (via Syntaxstruktur) übersetzt in die Prolog-Fakten

```
object(computer,sk-1)/300.
object(language,sk-2)/300.
object(chip,sk-3)/300.

object(design,sk-4,sk-1,sk-3)/300.
object(specification,sk-5,sk-6,sk-7)/300.

property(automatic,sk-4)/300.
property(formal,sk-5)/300.

by_with_for(sk-2,sk-5)/300.

goal(sk-2,sk-4)/300.
```

Die Syntaxstruktur des Titels (0.L.1)

Beachte:

1. Präpositionalphrase ist vielfach mehrdeutig , aber die intendierte Lesart wurde korrekt ermittelt

Alles das hier ist nicht trivial!

2. Unterscheidung zwischen relationalen und nicht-relationalen Substantiven; beachte die Verteilung der Argumente!
3. verschiedene Behandlung der Präpositionalphrasen:
   a. ‘‘*of* chips’’ ist von ‘‘design’’ subkategorisiert → *Argument*
   b. ‘‘*by* computer’’ zwar nicht subkategorisiert, dennoch → *Argument* für Agent
   c. ‘‘*for* ... design’’ ist *Adjunkt* → zusätzliche Bedingung (‘‘goal’’)

Reifikation: eine verallgemeinerbare Methode

4. Beziehungen innerhalb der Nominalkomposita werden nicht semantisch analysiert, sondern *reifiziert*
5. die Textstelle (hier: Titel-Nummer) wird als spezielles Argument repräsentiert (‘‘/300’’)

‘‘Interaktion’’, nicht ‘‘Dialog’’!

Beispiel-Interaktion:

```
Type in your query:

|: formal specifications of chips.

Query is:

object(chip,C)/D,
property(formal,S)/D,
object(specification,S,X,C)/D

Found nothing appropriate.
```

```
Type in your query:
|: the automatic design of specifications.

Query is:

property(automatic,_10839)/_4610,
object(design,_10839,_11297,_11391)/_4610,
(role(of,_10839,_12126)/_4610,
 object(specification,_12126,_12723,_12886)/_4610,[]))
 #(object(specification,_11391,_12126,_12723)/_4610,[])),

Found nothing appropriate.
```

Kein Erfolg - zu Recht. Details der Anfrage hier unwichtig.

```
Type in your query:
|: languages for the design by computer.

Query is:

goal(_9337,_9408)/_5685,
object(computer,_9806)/_5685,
object(language,_9337)/_5685,
object(design,_9408,_9806,_10841)/_5685,

The relevant document is:

300: a formal specification language for the
     automatic design of chips by computer
```

Erfolg. Die Textstelle als ‘‘Seiteneffekt’’. Kann auf Abschnitte und Kapitel etc. erweitert werden.

Unterschiede zu Chat-80: LogDoc

1. hat viel kleinere und weniger allgemeine sprachliche Abdeckung
2. verarbeitet nicht nur Fragen, sondern auch Aussagen
3. hat deutlich umfassenderen Themenbereich

Unterschiede zu SHRDLU: LogDoc

1. hat viel kleinere sprachliche Abdeckung, aber wo vorhanden, ist sie viel allgemeiner (kein Haufen von Hacks)
2. hat *viel* umfassenderen Themenbereich

Unterschiede zu Chat-80 *und* SHRDLU: LogDoc

1. kennt mehr Arten ‘‘thematischer’’ Beziehungen (z.B. Ziel, Mittel)
2. kennt mehr Typen syntaktischer Beziehungen (z.B. Nominalkomposita)
3. hat bessere Behandlung von Mehrdeutigkeiten (d.h. Pruning)
4. hat intensionale Datenbank
5. hat absolut keine Dialog- und keine Textkomponte (nur isolierte ‘‘Dokumente’’)
6. kennt (fast) keine ganzen Sätze (also keine Aussagen, sondern nur Beziehungen)

LogDoc ist sprachlich aber immer noch relativ einfach. Und es werden eben immer noch keine *Texte* ‘‘verstanden’’. Aber indem es ‘‘Dokumente’’ im Umfang eines Satzes oder darunter findet, ist es ein erster Schritt zu den folgenden Arten des (partiellen) Textverstehens.

Allerdings: Was man da *wirklich* getan hat, ist noch keineswegs klar - es ist in ± offensichtlicher Weise ein Retrieval-System zusammengehackt worden. Wie lässt sich das in einem allgemeineren Kontext formulieren?

3.3 Anwendung auf Fragenbeantwortung über Texten

Die Grundideen der logikbasierten Fragenbeantwortung über ganzen Texten mit richtigen Sätzen sei im folgenden illustriert anhand eines erfundenen Texts und mit von Hand gemachten Übersetzungen. .

Beispieltext:

Alle Computer bestehen aus einer Zentraleinheit, einer Tastatur und einem Monitor. *Die Tastatur* dient der Eingabe von Daten, *der Monitor* ihrer Ausgabe, und *die Zentraleinheit* ihrer Verarbeitung. Die Zentraleinheit des Computers der Marke ‘Stride’ verwendet einen Chip des Typs MC-68010.

Darüber möchte man Fragen stellen können wie

- Gibt es einen Computer, der eine Zentraleinheit vom Typ MC68010 verwendet?
- Welchen Typ von Chip verwendet die Zentraleinheit des ‘Stride’?
- Wozu dient die Zentraleinheit eines ‘Stride’?
- usw.

Wie soll man das anstellen?

3.3.1 Die Interpretation von Aussagen als Axiome

(Viele) Aussagesätze können als Aussagen der Logik erster Stufe dargestellt werden. Das ist Thema fast der gesamten formalen Semantik, aber statt der Ermittlung des Wahrheitswerts von Aussagesätzen wollen wir hier (wie oben dargestellt) etwas ‘‘assimilieren’’. Was?

Repräsentation 1. Satz: (‘‘Alle Computer bestehen aus einer Zentraleinheit, einer Tastatur und einem Monitor’’) in Prädikatenkalkül erster Stufe:

```
∀ C: computer(C) →
       (∃ Z: zentraleinheit(Z) ∧ ist_teil(Z,C)
     ∧ ∃ T: tastatur(T) ∧ ist_teil(T,C)
     ∧ ∃ M: monitor(M) ∧ ist_teil(M,C))
```

Vorläufig ohne gute Begründung (± weil wir es so gewohnt sind), wollen wir das aber in Prolog darstellen:

```
zentraleinheit(sk1(C))  :- computer(C).
ist_teil(sk1(C),C)      :- computer(C).

tastatur(sk2(C))        :- computer(C).
ist_teil(sk2(C),C)      :- computer(C).

monitor(sk3(C))         :- computer(C).
ist_teil(sk3(C),C)      :- computer(C).
```

3.3.2 Die Interpretation von Texten als Axiomensysteme

Wenn alle Sätze eines Texts voneinander unabhängig *wären*, dann wäre möglich

```
Satz1 =>  ∃ X: ...
Satz2 =>     ∧ ∀ X: ...
Satz3 =>        ∧ ∃ X: ...
Satz4 =>           ∧ ∃ X: ...
...
```

Dann könnte man die Sätze in einer beliebigen Reihenfolge lesen, ohne dass sich das Verständnis des Texts ändern dürfte (die logische Konjunktion ist kommutativ). Das 1ist natürlich *nicht* der Fall! Z.B. hier nicht:

Beispiel: 2. Satz muss gleichsam *expandiert* werden zu

Die Tastatur *eines jeden Computers, zu dem sie gehört,* dient der Eingabe von Daten

und dann übersetzt werden als:

```
∀ C: computer(C) →
        ∃ T: tastatur(T) ∧ ist_teil(T,C)
        ∧ ∀ D: daten(D)
        → ∀ E: eingabe(E,D,C) → dient(T,E)

< analog für Rest des Satzes >
```

3. Satz:

```
∃ C: computer(C) ∧ marke(C,stride) ∧
        ∃ Z: zentraleinheit(Z)  ∧ ist_teil(Z,C) ∧
        ∃ CH: chip(CH) ∧ typ(CH,mc_68010)
        ∧ verwendet(Z,CH)
```

Jetzt kann man alle Ausdrücke durch Konjunktionen zu einem Axiomensystem verknüpfen, und die Wissensbasis ist fertig.

Sieht in Prolog dann so aus:

```
zentraleinheit(sk1(C))  :- computer(C).
ist_teil(sk1(C),C)      :- computer(C).
tastatur(sk2(C))        :- computer(C).
ist_teil(sk2(C),C)      :- computer(C).
monitor(sk3(C))         :- computer(C).
ist_teil(sk3(C),C)      :- computer(C).

tastatur(sk4(C))        :- computer(C).
ist_teil(sk4(C),C)      :- computer(C).
dient(sk4(C),E)         :- computer(C),
                           daten(D), eingabe(E,D,C).

monitor(sk5(C))         :- computer(C).
ist_teil(sk5(C),C)      :- computer(C).
dient(sk5(C),A)         :- computer(C),
                           daten(D), ausgabe(A,D,C).

zentraleinheit(sk6(C))  :- computer(C).
ist_teil(sk6(C),C)      :- computer(C).
dient(sk6(C),V)         :- computer(C),
                           daten(D), verarbeitung(V,D).

computer(sk7).
marke(sk7,stride).
zentraleinheit(sk8).
ist_teil(sk8,sk7).
chip(sk9).
typ(sk9,mc_68010).
verwendet(sk8,sk9).
```

Achtung:

1. Das ist ein *sehr* provisorisches Resultat. Präsuppositionen, referentielle Ausdrücke (u.v.a.m.) sind ignoriert worden.
2. Auf welchem *Weg* man diese Zusammenhänge zwischen den Sätzen ermittelt hat, wird in dieser Vorlesung ausdrücklich *nicht* erörtert. Siehe dazu die Vorlesung ⇒‘‘Diskursanalysemethoden ’’ Im folgenden wird einfach immer angenommen, dass man den Zusammenhang zwischen den Sätzen eines Texts ‘‘irgendwie’’ gefunden hat.

3.3.3 Die Interpretation von Fragen als Theoreme

Was fehlt noch für die Anwendung auf ein Frage-Antwort-System?

Behandlung von Fragesätzen! Umrissmässig in ECL2 erwähnt.

Nochmals:

- Die formale Semantik kennt *an sich* keine Fragesätze

Wie soll man die Fragen dann maschinell beantworten?

Antwort: Logik ist nicht nur Sprache, sondern auch Kalkül.

Schlussfolgerungsschemata erlauben formale Ableitungen, z.B. *Modus ponens:*

```
p → q          (Axiom 1)
p              (Axiom 2)
_____________
q              (zu beweisende Aussage = Theorem)
```

Beispiel:

```
∀ T: terminal_command(T)
        → ∃ C: control_code(C) ∧ invoke(C,T)

terminal_command('Line Feed')
_____________________________________________
∃ C: control_code(C) ∧ invoke(C,'Line Feed')
```

Diese Aussage kann man formal aus den zwei Axiomen ableiten.

Was macht man also mit Fragesätzen?

- Man behandelt Fragen als zu beweisende Aussagen, also als *Theoreme*
- Den Beweis lässt man von einem *automatischen Theorembeweiser* durchführen.
- Ein gelungener Beweis heisst ‘‘ja’’, ein fehlgeschlagener ‘‘nein’’

- Extraktion der während des Beweises aufgebauten Variablenbindungen erlaubt auch Anwendung auf W-Fragen. Siehe unten.

Wenn man Prolog als Sprache und den Prolog-Interpreter als automatischen Theorembeweiser verwendet, wird das alles sehr verständlich und direkt anwendbar:

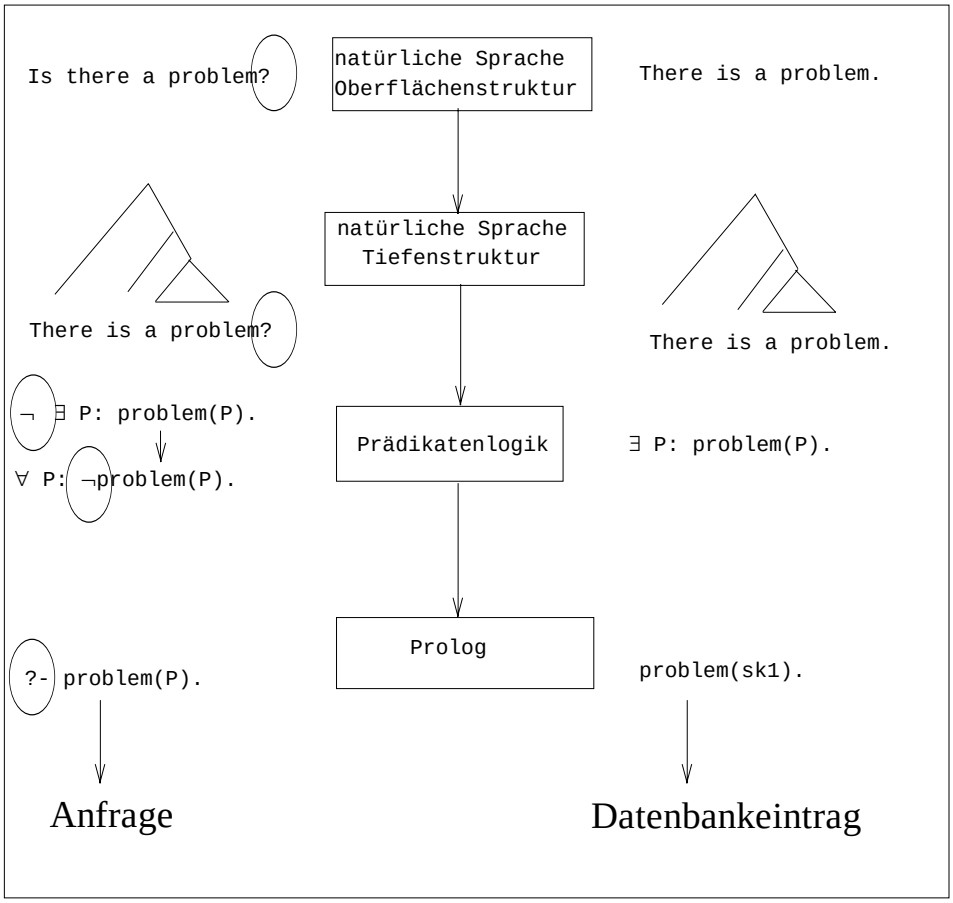
1. Axiome werden dem Axiomensystem *direkt* zugefügt (in Prolog-Terminologie: Klauseln werden dem Programm zugefügt)

2. Theoreme werden bewiesen (in Prolog-Terminologie: Anfragen werden gestellt; technisch korrekt: dem Axiomensystem als *negierte* Axiome zugefügt)

3. der Theorembeweiser sucht automatisch einen Beweis (in Prolog-Terminologie: der Prolog-Interpreter sucht eine Antwort; technisch korrekt: ein Widerspruch im Axiomensystem wird gesucht)

Problem: Man muss vom Prädikatenkalkül irgendwie nach Prolog kommen. Siehe unten.

Also:



Exakte Form der logischen Repräsentation von J/N-Fragen:

- hängt vom Beweisverfahren ab
- ist aber immer *fast* dieselbe wie für Aussagesätze
- **Beispiel** Refutationsbeweis (Standard-Verfahren für Prolog):

```
Assertion:    ∃ X: (hund(X) ∧ schläft(X))
Theorem:    ¬ ∃ X: (hund(X) ∧ schläft(X))
```

Mit anderen Worten:

1. Syntaxanalyse sollte zuerst Unterschiede Fragesätze/Aussagesätze/(Befehlssätze) *konzentrieren*
2. dann ist Übersetzung von Fragesätzen und Aussagesätzen im Kern *gleich*

```
Some department employs a tutor .

decl(s(np(det(some)
          n(department))
       vp(verb(employ)
          np(det(a)
             n(tutor)))))


Does some department employ a tutor ?

interr(s(np(det(some)
            n(department))
         vp(verb(employ)
            np(det(a)
               n(tutor)))))
```

Dann nur noch

```
syntax_to_logic(decl(Syntax),Logic)
        :-      syntax_to_logic(Syntax,Logic).
syntax_to_logic(interr(Syntax),~(Logic))
        :-      syntax_to_logic(Syntax,Logic).
```

Beachte: ~ ist Negationssymbol!

1. Anwendung: J/N-Fragen:

Frage: ‘‘Gibt es einen Computer, der eine Zentraleinheit vom Typ MC68010 verwendet?’’

Antwort: ‘‘ja’’

Man übersetzt die Frage ebenfalls in Logik:

```
¬ ∃ C: computer(C) ∧ ∃ Z: zentraleinheit(Z)
↑!     ∧ verwendet(C,Z) ∧ typ(Z,mc_68010).
```

resp. in Prolog

```
:- computer(C), zentraleinheit(Z), verwendet(C,Z),
   typ(Z,mc_68010).
```

Lässt sich über dem aus dem Text abgeleiteten Axiomensystem beweisen.

- Das Gelingen/Misslingen des Beweises ist die Antwort auf eine J/N-Frage

2. Anwendung: W-Fragen:
Frage: ‘‘Welchen Typ von Chip verwendet die Zentraleinheit des ‘Stride’?’’

Antwort: ‘‘MC 68010’’

Die Frage wird ‘‘paraphrasiert’’ als
‘‘Gibt es etwas, das die Zentraleinheit des ‘Stride’ als Typ von Chip verwendet?’’
oder vielleicht eher als
‘‘Die Zentraleinheit des ‘Stride’ verwendet was als Typ von Chip?’’

```
¬ ∃ X:   ∃ C: marke(C,stride) ∧
        ∃ Z: zentraleinheit(Z) ∧
        ist_teil(Z,C) ∧
        ∃ CH: chip(CH) ∧
        verwendet(Z,CH)
        ∧ typ(CH,X)
```

resp.

```
?-  marke(C,stride), zentraleinheit(Z), chip(CH),
    ist_teil(Z,C), verwendet(Z,CH), typ(CH,X).
```

Beachte:

- die Variable `X` hat am Ende des Beweises deren Wert ‘mc_68010’ angenommen
- während eines erfolgreichen Beweises aufgebaute Variablenbindungen sind die Antwort auf eine W-Frage

Mit etwas mehr Aufwand kann man ganz analog vorgehen für
Frage: ‘‘Wozu dient die Zentraleinheit eines ‘Stride’?’’

Antwort: ‘‘Zum Verarbeiten von Daten’’

Für intelligentere Antworten: Verwendung von *allgemeinem Weltwissen:*

Frage: ‘‘Welchen Typ von Chip verwendet der ‘Stride’?’’

Antwort: ‘‘MC 68010’’

erforderliches Wissen:

```
Wenn gilt:      ‘X enthält Y’ und ‘Y verwendet Z’
dann gilt auch: ‘X verwendet Z’
```

d.h.: Die Relation ‘‘verwendet’’ ist vererbbar in der Teil-Ganzes-Hierarchie; ist nicht bei allen Eigenschaften und Relationen so!

Ist natürlich ebenfalls[4] als logische Regel darstellbar:

$$\forall X: \forall Y: \forall Z: \quad ist\_teil(X,Y) \wedge verwendet(Y,Z) \rightarrow verwendet(X,Z)$$

Aber Achtung: Stimmt diese Regel *immer*? Derartige Regeln sind u.U. sehr bereichsabhängig. Eine Aussage wird nicht dadurch allgemein wahr, dass sie die Form eines Axioms annimmt!

Vorteile des Verfahrens:

1. es gibt eine grosse Anzahl gut verstandener Schlussfolgerungsmethoden für die Logik
2. für einige davon gibt es effiziente allgemeine automatische Theorembeweiser
3. als mögliche *Formen einer Antwort* ergeben sich automatisch:

4. Weltwissen und Textwissen werden hier also gleich behandelt. Ist das immer sinnvoll? Sind Texte immer zuverlässig? Muss man Texte daher dennoch immer auf ihren Wahrheitswert hin testen? Was tun, wenn Text und Wissensbasis inkonsistent?

a. Ja/Nein-Frage: *Gelingen* oder *Misslingen* des Beweises
b. W-Frage: zusätzlich die beim Beweis aufgebauten *Variablenbindungen*
c. u.U. *zusätzlich:* Ableiten einer sprachlich ausformulierten Antwort (aus Variablenbindungen und ggf. Elementen der Frage)

4. allgemeines *Weltwissen* lässt sich ebenfalls in Logik ausdrücken und daher leicht im Frage-Antwort-System integrieren.

Die Vorteile dieses Ansatzes sind also erheblich. Allerdings ist auch der Aufwand erheblich:

1. Man braucht eine Form der Logik, welche genug *ausdrucksstark* ist, um möglichst alle uns interessierenden Tatbestände abzubilden.
2. Man braucht eine Form der Logik, welche für das *effiziente* automatische Theorembeweisen geeignet ist.
3. Man muss den natürlichsprachlichen Input automatisch in diese Logik *übersetzen*.

Im folgenden:

1. Vergleich der Eignung verschiedener Logiken für die maschinelle Verarbeitung
2. Übersetzung von Syntaxstrukturen in Logische Formen
3. einige Problemfälle

3.4 Prädikatenlogik, Klausellogik, Horn-Klausel-Logik

3.4.1 Klausellogik, eine Paraphrase der Prädikatenlogik

3.4.1.1 Syntaktische Variabilität der Prädikatenlogik

Zur Logik: Siehe ⇒hier in Vorlesung der ECL2.

Problem: Die normale Notation der Logik lässt sehr viele Variationene zu (‘‘Logik’’ i.S.v. Prädikatenlogik): Man kann in vielen Fällen ein und dasselbe auf verschiedene Art ausdrücken. So sind das alles äquivalente Ausdrucksweisen:

```
∀ H: (hund(H) → ∀ P: (postbote(P) → beisst(H,P)))
∀ H: ¬ (hund(H) ∨ ∀ P: (postbote(P) → beisst(H,P)))
∀ H: ¬ (hund(H) ∨ ∀ P: ¬ (postbote(P) ∨ beisst(H,P)))
∀ H: (hund(H) → ∀ P: ¬ (postbote(P) ∨ beisst(H,P)))
¬ ∃ ¬ (hund(H) → ∀ P: (postbote(P) → beisst(H,P)))
¬ ∃ ¬ (¬ hund(H) ∨ ∀ P: (postbote(P) → beisst(H,P)))
etc. etc.
```

Unendlich viele Darstellungen sind möglich.

Das ist eine Belastung für den Theorembeweiser: Er muss bei jedem Schritt einen Ausdruck immer auf alle denkbaren notationellen Varianten hin prüfen.

- **Aber:** Das ist nicht ‘‘die’’ Darstellung für Logik.
- **Ziel:** eine (möglichst) kanonische Notation.

Eine deutlich kanonischere Notation ist die der *Klausel-Logik*. Jede Aussage, die sich im Prädikatenkalkül erster Stufe ausdrücken lässt, lässt sich in eine Aussage in Klausel-Logik umformen.

3.4.1.2 Umformung von Prädikatenlogik in Klausel-Logik

ELIMINATION DER IMPLIKATION:

```
P → Q  ==> ¬ P ∨ Q
```

Beachte: ==> ist keine Äquivalenz!

Negation soll nur noch auf atomare Formeln angewendet werden. Konversion durch (u.a.) de Morgans Gesetze

```
¬(X ∧ Y) ==> ¬ X ∨ ¬ Y
¬(X ∨ Y) ==> ¬ X ∧ ¬ Y
```

STANDARDISIEREN DER VARIABLEN:

Derselbe Variablenname nur noch im Skopus eines einzigen Quantors verwendet

```
∀ X: q(X) → ∃ X: p(X) ∧ r(X)
==>
∀ X: q(X) → ∃ Y: p(Y) ∧ r(Y)
```

ELIMINATION DES EXISTENZQUANTORS:

existentiell quantifizierte Variablen werden durch Skolemfunktionen ersetzt

```
∃ X: dog(X) ∧ red(X)
==>
dog(sk1) ∧ red(sk1)
```

KONVERSION IN PRENEX-FORM:

Alle universellen Quantoren werden an den Anfang gezogen; Skopus aller Quantoren ist ganze Formel

```
∀ M:((¬ man(M) ∨ ∀ W:(¬woman(W) ∨
¬loves(M,W)))) ∨ happy(M)
==>
∀ M ∀ W (¬man(M) ∨ ¬woman(W) ∨
¬loves(M,W)) ∨ happy(M)
```

ELIMINATION DER ALLQUANTOREN:

```
∀ M ∀ W (¬man(M) ∨ ¬woman(W) ∨ ¬loves(M,W))
∨ happy(M)
==>
(¬man(M) ∨ ¬woman(W) ∨ ¬loves(M,W)) ∨ happy(M)
```

KONJUNKTIVE NORMALFORM:

Überführen in eine Konjunktion von Disjunktionen

```
X ∨ (Y ∧ Z)  ==> (X ∨ Y) ∧ (X ∨ Z)
```

ELIMINATION DER KONJUNKTIONEN:

Erzeugen einer Menge von Klauseln, von denen jede aus einer Disjunktion von Literalen besteht

```
(P ∨ ¬Q) ∧ (R ∨ ¬S ∨ ¬T)
==>
{ P, ¬Q }
{ R, ¬S, ¬T }
```

VARIABLEN AUSEINANDERSTANDARDISIEREN:

Jeder Variablenname nur noch in einer einzigen Klausel

```
{ p(X), ¬q(X) }
{ p(X), ¬s(X), ¬t(Y) }
==>
{ p(X), ¬q(X) }
{ p(Z), ¬s(Z), ¬t(Y) }
```

NB: Die ‘‘{’’ ‘‘}’’ werden im folgenden unterdrückt

Zu beachten: Die Beziehung Prädikatenlogik:Klausellogik ist *keine Äquivalenzbeziehung:* Die Skolemisierte Version impliziert die existentiell quantifizierte. aber

Äquivalenz: wechselseitige Implikation

nicht umgekehrt:

```
(1)            ∃  X p(X)
```

```
Nach Skolemisierung:
```

```
(2)            p(sk1)
```

wo ‘‘sk1’’ eine Skolemkonstante ist. (2) impliziert (1), aber nicht umgekehrt.

Es kann aber bewiesen werden, dass eine Formel in erweiterter Klauselform genau dann unerfüllbar ist, wenn die entsprechende Formel in prädikatenlogischer Notation unerfüllbar ist[5]

Hintergrundinformation (0.L.2)

3.4.1.3 Einige notationelle Vereinfachungen

Die Klausel-Notation, wie wir sie bisher verwendet haben, ist nicht eben eingängig. Oft werden in der Klausel-Logik daher noch einige notationelle Vereinfachungen eingeführt, die zwar nicht von grundsätzlicher Bedeutung sind, aber das Leben des Benützers wesentlich einfacher machen. Einige davon haben wir schon bisher verwendet, da sie mit dem Prädikatenkalkül erster Stufe kompatibel sind. Genaugenommen sind diese notationellen Konventionen nur in *Prolog* einigermassen allgemein üblich. In Darstellungen über die Klausel-Logik im allgemeinen und die Horn-Klausel-Logik im besonderen findet man verschiedene andere Notationen. Sie sind aber nur in trivialer Weise anders als die hier vorgestellte Art der Darstellung. Die Verwendung der Prolog-Konventionen auch für die Klausel-Logik im

5. Loveland 1978:38 ff.

schrittweise näher ...

allgemeinen erleichtert die Kommunikation aber beträchtlich.

Die erste dieser Konventionen ist die, Klauseln immer dann wieder als Implikationen zu schreiben, wenn die Klausel die folgende generelle Form hat

```
¬P, ¬Q, R, S  ==>  P , Q → R , S
```

Das heisst, dass das Komma im Antezedens für die Konjunktion steht, im Konsequens hingegen für die Disjunktion (oft wird für die Disjunktion auch ein ";" verwendet).

Sodann wird die Bedeutung des Implikationssymbols erweitert:

... an die Darstellung ...

Implikationssymbol ohne Antezedens: (d.h. nur ein positives Literal)

```
→ P  ==>  true → P    ==> P
```

(wenn etwas wahr ist, ohne dass eine bestimmte Bedingung erfüllt sein muss, dann ist es einfach wahr). Ein Implikationssymbol ohne Konsequens: (nur ein negatives Literal)

```
P →  ==>  ¬P
```

... von Prolog

Die drei Änderungen ergeben die sog. *Kowalski-Klauselform*.

Weitere (schon früher verwendete) Konventionen bestehen darin,

1. alle Variablennamen beginnen mit Grossbuchstaben
2. statt des Implikationspfeils Verwendung des Zeichens ‘‘:-’’
3. am Ende einer jeden Klausel ein Punkt

Wenn man alle diese Notationskonventionen anwendet, ergeben sich folgende Darstellungen für die verschiedenen Typen von Aussagen. Partikularaussagen wie zum Beispiel

```
hund(barry)
braun(barry)
```

bleiben sich fast ganz gleich:

```
hund(barry).
braun(barry).
```

Hier kam nur der Punkt dazu (aber man bedenke, dass man dafür in den Büchern

manchmal auch so etwas antreffen wird):

```
H(b)
```

Allaussagen werden in dieser Form der Klausel-Logik als *Regeln* (Schlussregeln) ausgedrückt. Dabei sind natürlich alle Variablen implizit universell quantifiziert. Was im Prädikatenkalkül erster Stufe folgendermassen dargestellt würde

Das ist bekannt

```
∀ T: (säuger(T)  →  tier(T))
für ‘‘alle Säuger sind Tiere’’

==>

tier(T)  :-  säuger(T).
```

Existenzaussagen sind in dieser Form der Klausellogik (wie in allen Arten von Klausellogik) nicht mehr unbedingt intutiver als die Darstellung der Prädikatenlogik. Im normalen Prädikatenkalkül würde man den Satz ‘‘In Indien gibt es Löwen’’ formulieren als

das hoffentlich auch

```
∃ X: (löwe(X) ∧ in(X,indien))
für ‘‘es gibt Löwen in Indien’’

==>

löwe(sk1).
in(sk1,indien).
```

wobei ‘‘indien’’ natürlich ein echter Eigenname ist.

‘‘echte’’ und ‘‘unechte’’ Eigennamen

In der modifizierten Klauseldarstellung wird das zum zweiten Ausdruck. Wie schon früher erwähnt, könnte man statt ‘‘sk1’’ auch ‘‘löwe1’’ oder ‘‘01001’’ wählen; die exakte Form der künstlichen Namen ist *im Prinzip* nicht wichtig.

All dies erhöht lediglich die Leserlichkeit und ist inhaltlich irrelevant. Aber durch diese Art der Klausel-Notation werden logische Axiomen-Systeme in einem gewissen Sinn auch anders ‘‘zerschnitten’’, als in der Standard-Notation der Prädikatenlogik. In mancher Hinsicht ist diese neue Einteilung intuitiv einleuchtender, als diejenige, die man in der normalen Prädikatenlogik vornimmt. Anstatt der ursprünglich eingeführten Unterteilung der logischen Aussagen in Partikularaussagen und generelle Aussagen, welche dann ihrerseits in existentielle und universelle unterteilt sind, werden die Partikularaussagen und die Existenzaussagen zusammengefasst als *Fakten*, und die Allaussagen werden als *Regeln* bezeichnet.

1. *Fakten* sind ‘‘Informationspakete’’ über *einzelne* Objekte und deren Beziehungen zueinander.
2. *Regeln* hingegen drücken *allgemeine Gesetzmässigkeiten* über die Umstände aus, unter denen bestimmte Beziehungen zwischen *beliebigen Objekten* bestehen.

Unter Benützung dieser Notationsvereinfachungen, und mit offensichtlichen Umschreibungen der logischen Quantoren und Operatoren, ist nunmehr die automatische Konversion von Prädikatenlogik in Klausellogik sehr einfach. Einige Beispiele:

Einfacher Fall:

```
all dogs are mammals

all(D):(dog(D)->mammal(D))

mammal(D) :- dog(D).
```

Etwas schwieriger:

```
there are some candies all children like

exists(C):(candy(C)&all(CH):(child(CH)->like(CH,C)))

candy(sk1).
like(CH,sk1) :- child(CH).
```

Noch etwas komplizierter (wegen des in einen Allquantor eingebettetetn Existenzquantors):

```
every man loves a woman

all(M):(man(M)->exists(W):(woman(W)&loves(M,W)))

woman(sk6(M)) :- man(M).
loves(M,sk6(M)) :- man(M).
```

Hier nun eine Skolem-*Funktion*.

Und hier folgt etwas, was sehr merkwürdig aussieht:

```
a supplier whose supplies are on time is preferred
```

```
formula in predicate logic notation:
     all(S):(supplier(S)&all(P):

supplies(S,sk5(S)); preferred(S) :- supplier(S).
preferred(S) :- supplier(S), arriveontime(sk5(S)).
```

Ableitungen im Einzelnen (0.L.3)

Leider: Auch mit Klausellogik sind effiziente maschinelle Beweise (noch) nicht möglich. Und zudem kann es auch in der Klausellogik vorkommen, dass ein Beweis nicht endet (Halbentscheidbarkeit der Logik erster Stufe).

3.4.2 Horn-Klausel-Logik, eine Teilmenge der Prädikatenlogik

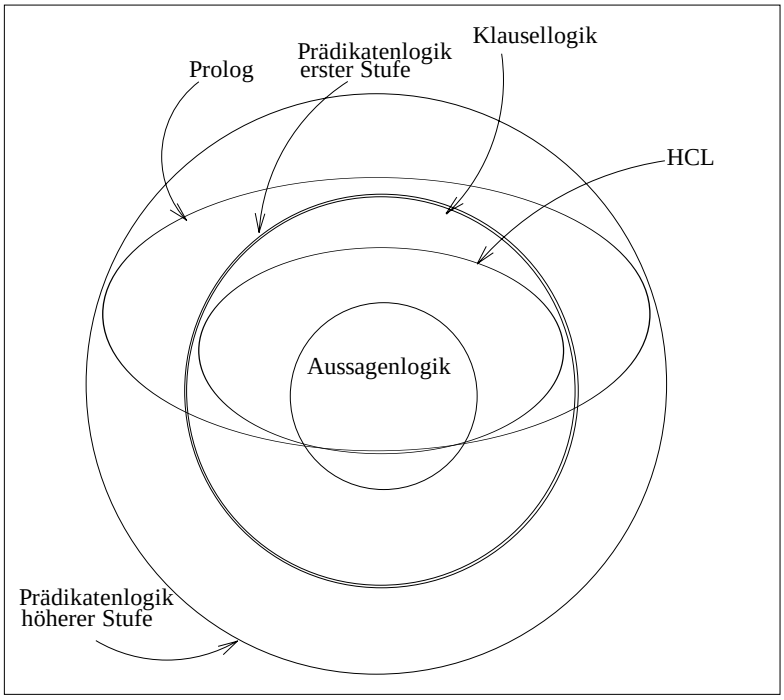
Daher: Weitere Einschränkung: Horn-Klausel-Logik.

Formal ist die Definition einer Horn-Klausel: ‘‘Eine Klausel mit höchstens einem positiven Literal’’.

HCL also nur eine *Teilmenge* der Logik erster Stufe!

Versuch, das graphisch darzustellen: [6]

6. Beachte: Beziehung zwischen Klausellogik und Logik erster Stufe ist nicht Äquivalenz (siehe oben); ist im Bild aber schlecht auszudrücken.



Also: *Keine* Horn-Klausel ist z.B.:

```
landlebend(T),
wasserlebend(T),
fliegend(T)          :- tier(T).
```

Das ist eine schmerzliche Einschränkung: Solche Dinge möchte man aussagen können (es gibt schliesslich Amphibien und Wasservögel usw.). Hingegen sind folgende zwei Klauseln Horn-Klauseln:

‘‘atypische’’ Horn-Klauseln

```
ermordet(peter,john) :- .
:- ermordet(peter,B), butler(B).
```

d.h. nur ein (einziges) positives Literal resp. *nur* negierte Literale. Der erste Fall ist einfach: Er entspricht

```
ermordet(peter,john) :- true.
```

d.h. es ist eine Aussage, die unter allen Umständen, also trivial, wahr ist.

Der zweite Fall ist anders gelagert: Der unseren Systemen zugrundliegende *Refutations*methode versucht einen *indirekten* Beweis zu führen, indem man einen *Widerspruch* zum *negierten* Theorem findet und daraus das Theorem selbst ableitet.

Deshalb wird automatisch *jedes* negierte Literal als ‘‘Frage’’ interpretiert. Obwohl man also Horn-Klauseln niederschreiben kann, die nur aus negierten Literalen bestehen, drückt man damit nicht das aus, was man eigentlich wollte, nämlich ein negiertes *Faktum*, sondern man stellt eine *Frage*. Man kann also keine elementare Negation *in der Datenbank* haben. Das ist nicht die ‘‘Schuld’’ der Klausellogik (Horn oder non-Horn), sondern die des gewählten Beweisverfahrens. Man könnte ein anderes Beweisverfahren auf die Klausellogik ansetzen, und dann könnte man die elementare Negation auch in der Datenbank haben. Aber es ist gar nicht sicher, ob das ein Vorteil wäre oder nicht vielleicht eher ein Nachteil. Wenn man nämlich die elementare Negation in der Datenbank ausdrücken könnte, *müsste* man (um die Vollständigkeit des Systems zu bewahren) von jedem Objekt nicht nur angeben, welche Attribute es *hat*, sondern auch, welche es *nicht hat*. Auf diese Weise füllt man aber jede Datenbank im Nu mit negativen Informationen. Dazu mehr unten.

Mehr zur Refutation (0.L.4)

3.4.2.1 Eingebettete Quantoren in Klausel-Logik

Da Quantoren eingeführt worden sind, um die verschiedenen möglichen Einbettungen von Ausdrücken für Allgemeinheit (und die resultierenden Skopusbeziehungen) auszudrücken, muss man in der Horn-Klausel-Logik auch dafür eine Ausdrucksmöglichkeit haben. In dieser Beziehung erlauben die Horn-Klauseln gleich viel, wie die allgemeinen Klauseln. Das Problem wird erst hier erwähnt, da es sonst die Ausführungen über das Umformulieren von FOL-Aussagen in Klausel-Form noch unverständlicher gemacht hätte.

All das ist *im Prinzip* bekannt,...

Ein Existenzquantor mit weitem Skopus über anderen Existenzquantoren ist einfach: Die Einbettung kann getrost ignoriert werden. Da nämlich

```
∃ X: (person(X) ∧ ∃ Y: (thing(Y) ∧ owns(X,Y)))
```

für ‘‘Somebody owns something’’ und

```
∃ Y: (thing(Y) ∧ ∃ X: (person(X) ∧ owns(X,Y)))
```

für ‘‘Something is owned by someone’’ logisch äquivalent sind, kann man sie beide in Horn-Klausel-Logik skopusfrei darstellen als

```
person(sk1).
thing(sk2).
owns(sk1,sk2).
```

Schwieriger ist der Fall, wo *verschiedene* Quantoren involviert sind.

aber *in der Praxis* wird vor allem das hier oft weniger verstanden.

5) Everybody loves somebody

Bekanntlich wird Beispiel 5 wird (in seiner *primären* Lesart) im Prädikatenkalkül erster Stufe dargestellt als

```
5a) ∀ X: (person(X) →
        ∃ Y: (person(Y) ∧ loves(X,Y)))
```

wo der Allquantor weiten Skopus über den Existenzquantor im Konsequens hat, im Gegensatz zur sekundären Lesart

6) There is somebody that everybody loves

Für die *primäre* Lesart müssen wir also auch in HCL zum Ausdruck bringen, dass die geliebte Person nicht (notwendigerweise) ein und dieselbe Person ist. Je nachdem, wer da liebt, kann es eine andere geliebte Person sein. Die Identitiät der geliebten Person ist eine Funktion der Identität der liebenden Person. Man darf daher diesen Tatbestand *nicht* einfach (wie man es vielleicht zu tun geneigt wäre) mit Skolem-Konstanten ausdrücken, zum Beispiel als

```
6a) loves(X,sk1) :- person(X).
    person(sk1).
```

Warum nicht? Sofern in der Datenbank irgendwo die Fakten

```
person(peter).
person(john).
```

vorkämen, würde eine komplexe Frage

```
?- loves(peter,X), loves(john,X)
```

für ‘‘Does Peter love whoever John loves?’’ nunmehr *fälschlicherweise* zu einem positiven Resultat ‘‘yes’’ (mit `X=sk1`) führen. Die gewählte Darstellung 6a ist also *falsch*; sie wäre die korrekte Darstellung für Lesart 6.

Korrekt wird die erste Lesart von Beispiel 5 in Horn-Klausel-Logik in der folgenden Art ausgedrückt:

```
5a) loves(X,sk1(X)) :- person(X).
    person(sk1(X)) :- person(X).
```

Hier liegt der allgemeinere Fall einer Skolem-*Funktion* vor (eine Skolem-Konstante kann als eine entartete Skolem-Funktion mit Null Argumenten interpretiert werden). Der *Name der Funktion* ist erfunden worden, und das Argument (allgemein: die Argumente) der Skolem-Funktion stammen von jenen Allquantoren, die weiteren Skopus als der Existenzquantor haben, der zur Schaffung der Skolem-Funktion führt. Diese Notation führt dazu, dass beim Abarbeiten der Regeln dynamisch ein kompletter Name erfunden wird. So wird zum Beispiel der Name `sk1(peter)` dynamisch geschaffen, wenn die Regel zum ersten Mal mit der Frage `loves(peter,X)` aufgerufen wird. Dieser Name wird nunmehr beim Aufruf der zweiten Frage, also `loves(john,X)`, verwendet, sodass diese Frage zu `loves(john,sk1(peter))` wird, und diese Frage wird natürlich negativ beantwortet werden.

Allquantoren, die in andere Allquantoren eingebettet sind, sind manchmal sehr einfach in Horn-Klausel-Logik zu übersetzen, und manchmal gar nicht (!).

7) Everybody loves everybody

das im Prädikatenkalkül erster Stufe dargestellt wird als

```
7a) ∀ X:(person(X) → ∀ Y:(person(Y)
                → loves(X,Y)))
```

wird in Horn-Klausel-Logik zu

```
7b) loves(X,Y) :- person(X), person(Y).
```

Wenn der Ausgangssatz hingegen ist

8) A supplier is preferred if all his supplies arrive on time

so wird das im Prädikatenkalkül erster Stufe üblicherweise dargestellt als

```
8a) ∀ X: (supplier(X)
      ∧ (∀ Y:supply(Y,X) → on_time(Y))
                                → preferred(X))
```

und das kann *nicht* in eine Horn-Klausel-Darstellung überführt werden. Es ergibt in Klauselform nämlich

```
8b) preferred(X) :- supplier(X), on_time(sk1(X)).
    preferred(X), supply(sk1(X),X) :- supplier(X).
```

und die zweite dieser Klauseln ist nicht-Horn, da ja Regeln mit disjunktivem Kopf nicht zulässig sind. *Siehe unten.*

Wie oft kommen derartige nicht-Horn-Klauseln vor? Nicht allzu oft, aber eben doch manchmal. Oben trafen wir einen ersten Fall an, hier nun einen weiteren, und ein dritter Fall, wo eine derartige nicht-Horn-Klausel herauskommt, ist die Aussage, dass

9) Alle Menschen sind entweder männlich oder aber weiblich

(exklusives oder), was im Prädikatenkalkül erster Stufe bekanntlich so aussieht:

```
∀ M:(human(M) → ((male(M) ∨ female(M)) ∧
¬(male(M) ∧ female(M))))
```

Das würde in *allgemeiner* Klausel-Logik zu

```
female(X), male(X)  :-  human(X).
:- human(X), male(X), female(X).
```

wo die erste Regel einen disjunktiven Regelkopf aufweist. Die zweite Regel hat gar keinen Regelkopf, was in allgemeiner Klausel-Logik die Negation ausdrückt (womit die zweite Zeile ausdrückt, dass kein Mensch sowohl männlich wie weiblich sein kann). Man ist vielleicht weniger überrascht, dass in diesem Fall eine nicht-Horn-Klausel herauskommt, denn hier hatten wir ja ganz explizit eine Disjunktion im Konsequens der FOL-Aussage. Unerwartet ist hingegen, dass es es auch Aussagen gibt, die in der Darstellung des Prädikatenkalküls ganz harmlos aussehen, die sich aber bei Umformung in Klauselform als nicht-Horn herausstellen: Das oben verwendete Beispiel ‘‘A supplier is preferred if all his supplies arrive on time’’ gehörte dazu.

Wie wichtig sind diese Einschränkungen wirklich? Es gibt Tricks, die es uns erlauben, die erwähnten Beschränkungen *im Einzelfall* zu umgehen, allerdings unter Opferung gewisser puristischer Ideale. Damit verlassen wir dann das Gebiet des ‘‘reinen Prolog’’. Wir werden diese Tricks daher später erwähnen.

3.4.2.2 Die Quantifikation der Variablen in (Horn-)Klausel-Logik

Wir hatten oben die Aussage gemacht, *alle* Variablen in (Horn-)Klausel-Logik seien universell quantifiziert. Aber stimmt dies wirklich? Betrachten wir eine allgemeine Aussage, welche definiert, was ein Grossvater ist:

```
∀ Alt : ∀ Jung :
  (∃ Mittel : (vater(Alt,Mittel)
  ∧ elter(Mittel,Jung)) → grossvater(Alt,Jung))
```

Diese dreifach quantifizierte Aussage wird in Horn-Klausel-Logik zu

```
grossvater(Alt,Jung)    :-      vater(Alt,Mittel),
                                elter(Mittel,Jung).
```

In beiden Versionen ist natürlich noch eine Definition des Begriffs ‘‘Elter’’ notwendig. In Horn-Klausel-Logik sehen die entsprechenden Definitionen folgendermassen aus:

```
elter(Alt,Jung)         :-      vater(Alt,Jung).
elter(Alt,Jung)         :-      mutter(Alt,Jung).
```

Die Horn-Klausel-Logik-Form der Regel ist in diesem Fall (wie meist) wesentlich übersichtlicher, als die Version in Prädikatenkalkül erster Stufe. Hier haben wir aber einen Fall, wo eine *existentiell* quantifizierte Variable (Mittel) in eine (implizit universell quantifizierte) Variable in Horn-Klausel-Logik übersetzt worden ist. Ist unsere ursprüngliche Aussage also falsch? Nein. Die existentiell quantifizierte Aussage mit engem Skopus ist nämlich logisch äquivalent mit der universellen Quantifikation mit weitestem Skopus:

```
∀ Alt : ∀ Jung :
 (∃ Mittel : (vater(Alt,Mittel)
 ∧ elter(Mittel,Jung))) → grossvater(Alt,Jung)
```

ist äquivalent mit

```
∀ Alt : ∀ Jung :
  ¬ ( ∃ Mittel : (vater(Alt,Mittel)
  ∧ elter(Mittel,Jung))) ∨ grossvater(Alt,Jung)
```

und nun kann man die Negation über den Existenzquantor hineinziehen und bekommt die äquivalente Form

```
∀ Alt : ∀ Jung :
  ∀ Mittel : ¬ (vater(Alt,Mittel)
  ∧ elter(Mittel,Jung)) ∨ grossvater(Alt,Jung))
```

und nunmehr wird die Ersetzung der Implikation durch eine Disjunktion mit Negation wieder rückgängig gemacht und man erhält

```
∀ Alt : ∀ Jung :
  ∀ Mittel : (vater(Alt,Mittel)
  ∧ elter(Mittel,Jung)) → grossvater(Alt,Jung))
```

Wir können also auch sagen, alle Variablen, die *nur* auf der rechten Seite einer Regel erscheinen, seien *existentiell* quantifiziert, und alle andern sind universell (oder, äquivalent: alle Variablen, die *auch* auf der linken Seite erscheinen, sind universell quantifiziert, alle andern existentiell). Diese Betrachtungsweise ist tatsächlich oft intuitiv einleuchtender, aber wir müssen die ursprüngliche Aussage deswegen nicht einschränken.

Das *ganze* Programm findet sich ⇒hier.

3.5 Einige Problemfälle

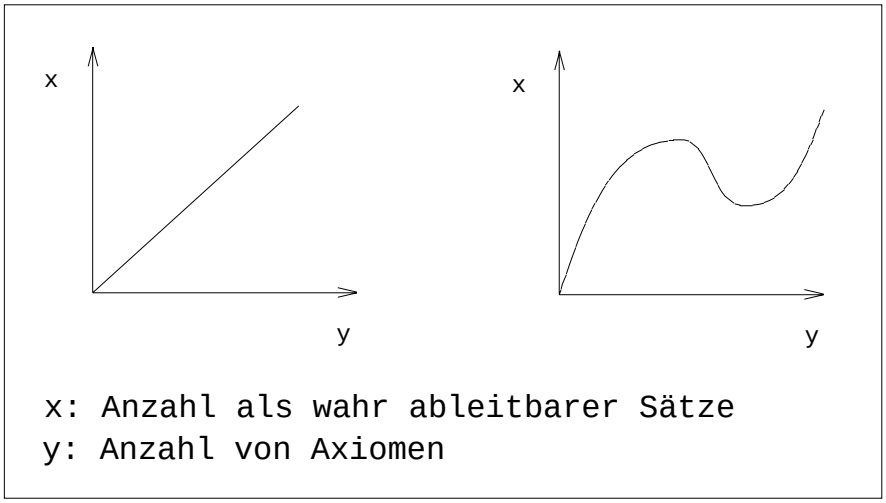
3.5.1 Negative Information

Bekanntlich kann man in der Horn-Klausel-Logik die Negation weder in Programm-Einträgen (Fakten) noch in Anfragen (Theoremen) verwenden. Für Negationen in *Fragen* gibt es immerhin eine Simulation durch ‘‘negation by failure’’:

```
not(X)  :-      X, !, fail.
not(X).
```

Dast ist aber *nicht* die logische Negation: Nicht-Beweisbarkeit ist ein *meta-logisches* Konzept. Man spricht über ganze logische Systeme, denn man sagt ja, eine bestimmte Aussage könne über einem gegebenen Axiomensystem nicht bewiesen werden.

Durch die Verwendung diese ‘‘Pseudo-Negation’’ erhält man eine *nicht-monotone* Logik. Die ist in Prolog also gewissermassen von Anfang an ‘‘eingebaut’’. Eine Logik *mit* elementarer Negation ist hingegen streng monoton: Wenn man einmal hat beweisen können, dass Hans *nicht* Bundespräsident ist, dann wird man das *immer* wieder beweisen können, und zwar auch, wenn man neue Axiome ins logische System einfügt. Die Anzahl von als wahr bewiesenen Sätzen wird also nie abnehmen, sondern nur zunehmen können (Bild links). Mit der als Nicht-Beweisbarkeit definierten Negation ist die Situation anders: Wenn man einmal abgeleitet hat, dass Hans nicht Bundespräsident sei, weil man keinen Eintrag finden konnte, *dass* er Bundespräsident ist, und wenn man nun einen neuen Eintrag in das Programm einfügt des Inhalts, *dass* Hans Bundespräsident ist, so wird die Anzahl der als wahr bewiesenen Sätze plötzlich abnehmen, und es ergibt sich ein Bild wie rechts:



3.5.1.1 Schwierigkeiten mit ‘‘Negation by Failure’’

Eine weitere Konsequenz aus dieser Art, die logische Negation zu simulieren, ist im Programmieralltag wichtiger, als die Non-Monotonität: Man bekommt keine Variablenbindungen aus einem pseudo-negierten Theorem. Wenn man an sich fragen

möchte

Wer ist *nicht* Lehrer?

und man das in der angegebenen Weise in Prolog so formuliert

```
?- \+lehrer(X).
```

so entspricht das, trotz der in Prolog eingebauten Antwortextraktion, nicht

Gibt es jemanden, der nicht Lehrer ist, und wenn ja, *wer ist es?*

In Tat und Wahrheit wird man bloss eine Ja- oder Nein-Antwort erhalten, und die Variable wird ungebunden bleiben. Das ergibt sich aus der Definition der Pseudo-Negation. Beim durch ‘‘fail’’ erzwungenen Backtracken gehen die Variablenbindungen in `P` verloren.

Zudem: (Gewisse) Variablen in ‘‘negierten’’ Termen müssen vorgängig gebunden sein, sonst bekommt man falsche Antworten: Wenn wir ein Programm haben

```
block(block1).
block(block2).
on(block1,block2).
clear(X)        :-      \+on(Y,X).
```

und dann die Anfrage stellen

```
?- clear(X), block(X).
```

(für ‘‘Gibt es einen Klotz, auf dem nichts steht?’’), *müsste* die Antwort `block1` sein, aber stattdessen erhält man ein ‘‘Nein’’: `\+on(Y,X)` ist erfolgreich, wenn `on(Y,X)` *nicht* erfolgreich ist. `on(Y,X)` ist aber erfolgreich (es findet `on(block1,block2)` als Lösung), und damit ist in der konjunktiven Frage `?- clear(X), block(X).` der erste Term ein Misserfolg, und damit die gesamte Frage auch. Wenn man hingegen die Frage umformuliert zu `?- block(X), clear(X).`, geht alles gut.

```
?- clear(X), block(X).
no                      <====== falsch !!

?- block(X), clear(X).
yes
X=block1
```

Moral: Man muss den ‘‘Generator’’ zuerst evaluieren, und erst dann den ‘‘Test’’. Es gibt Prolog-Implementationen, welche automatisch die Evaluation negierter Terme solange suspendieren, bis alle darin vorkommenden Variablen gebunden sind. In den

üblichen Implementationen muss man das aber selbst machen. Man kann auch einen *Meta-Interpreter* schreiben, der dies in mehr oder weniger sinnvoller Weise macht (siehe dazu die Vorlesung ‘‘Methoden der Künstlichen Intelligenz in der Sprachverarbeitung’’ ⇒hier).

3.5.1.2 Arten von Negationen

Manchmal trifft man auf die Auffassung, die ‘‘Negation by failure’’ sei nur ein Notbehelf, um die echte Negation zu simulieren, ein Notbehelf, der sobald wie möglich durch die echte Negation ersetzt werden sollte. Das ist nun nicht so sicher. Zumindest kann man recht plausibel die Auffassung vertreten, dass die Negation der natürlichen Sprache und dementsprechend von ‘‘common sense’’-Argumentationen keineswegs immer die elementare Negation der Logik sei. Wenn man *ausschliesslich* die explizite logische Negation benützen will, muss von jedem Objekt nicht nur angeben, welche Attribute es *hat*, sondern auch, welche es *nicht hat*. Auf diese Weise füllt man aber jede Datenbank im Nu mit negativen Informationen, und man darf füglich annehmen, dass wir unsere mentale Datenbank nicht in dieser Art benützen. Auch im praktischen Leben verwenden wir sehr oft ‘‘Negation by failure’’. **Beispiel:** Fahrplaninformation.

Dass man auch in der natürlichen Sprache zwei Arten von Negation unterscheidet, wird zumindest plausibel gemacht durch die Tatsache, dass die folgenden Sätze *nicht* synonym sind:

10) Hans ist nicht ungebildet

11) Hans ist gebildet

Wenn die durch Prädikatsnegation und durch Satznegation ausgedrückte Funktion (im Beispiel 10) identisch wären, müssten die Sätze synonym sein, da sich die Negationen in 10 dann aufheben müssten.

Wenn man nunmehr die zwei folgenden Beispiele vergleicht

12) Hans ist ungebildet

13) Hans ist nicht gebildet

so sieht man, dass man im ersten dieser Sätze sagt, dass man positiv weiss, dass Hans nicht gebildet ist, während man im zweiten Fall bloss sagt, dass man keine Anhaltspunkte dafür hat, dass Hans gebildet ist, was sich paraphrasieren lässt als

12a) Es ist der Fall dass Hans ungebildet ist

13a) Es ist nicht der Fall dass Hans gebildet ist

Man könnte also eventuell sagen:

```
Prädikatsnegation:  elementare Negation
Satznegation:        ‘‘negation by failure’’
```

Dies ist bei prädikativ verwendeten Adjektiven klarer, als bei prädikativ verwendeten Substantiven, weil es bei letzteren in natürlichen Sprachen nur selten Prädikatsnegation gibt: Es gibt z.B. keine Wortbildung ‘‘Un-Bundespräsident’’. Immerhin gibt es neben ‘‘Sinn’’ den ‘‘Unsinn’’ [7] und tatsächlich scheinen die folgenden Sätze erneut *nicht* synonymzu sein:

14) Das ist Unsinn

15) Das macht/gibt keinen Sinn

Das heisst (leider), dass ein Satz wie

16) Hans ist nicht Bundespräsident

ambig ist zwischen den zwei Lesarten der Negation.

Wie kann man das alles in Horn-Klausel-Logik repräsentieren? Nicht alles. Aber z.T. so

```
neg gebildet(hans).
neg bundespräsident(hans).
```

indem man einen Prolog-Operator, z.B. `neg` für die Prädikatsnegation verwendet.

Nun kann man unnegierte (und *prädikats*negierte) Fragen so beantworten, dass man das *Komplement* bezügl. ‘‘neg’’ bildet, und *sowohl* die Frage *wie* das Komplement zu beweisen versucht:

7. und viel andere Substantive mit Präfix ‘‘Un-’’; im Englischen gibt’s ‘‘discomfort’’, ‘‘impermeability’’, ‘‘unsteadiness’’, ‘‘non-transparency’’ etc.

Beispiel unnegierte Frage:

```
?- bundespräsident(hans).

    ==>

?-    bundespräsident(hans),
  neg bundespräsident(hans).
```

ergibt eine von vier Antworten: (cf. Covington 1994:228, Kowalski 1979:147)

1. ja (+,-)
2. nein (-,+)
3. weiss nicht (-,-)
4. Widerspruch in der Datenbank (+,+)

Bei *prädikatsnegierter* Frage: *nein*/*ja*/weiss nicht/Widerspruch

Eindeutig *satz*negierte Aussagen kann man hingegen *nicht* in HCL repräsentieren.

3.5.1.3 Disjunktive Regelköpfe

Wenn man die explizite (Prädikats-)Negation in der angegebenen Weise als Prädikat verwendet, kann man *im einzelnen Fall* auch die Beschränkung auf ein einziges Literal im Regelkopf umgehen. Wenn wir ein FOL-Aussage wie

$\forall$ M: (tier(T) $\rightarrow$ ((landlebend(T) $\vee$ wasserlebend(T))))

haben, entsprechend der **Non**-Horn-Klausel

```
landlebend(T), wasserlebend(T) :- tier(T).
```

(verinfachte Version des Beispiels oben), so können wir die FOL-Aussage umformen in

$\forall$ M: (tier(T) $\wedge$ $\neg$ wasserlebend(T)) $\rightarrow$ landlebend(T)

Die hier eingeführte Negation dürfen wir zwar in ein Prolog-‘‘not’’ überführen

```
landlebend(T)  :- tier(T), \+wasserlebend(T).
```

da hier der Generator-Term ‘‘`tier(T)`’’ vor dem negierten Term steht (und wenn das nicht der Fall wäre, könnte man es leicht entsprechend einrichten), aber wir können damit erst die Hälfte dessen beweisen, was wir aus der ursprünglichen FOL-Version ableiten konnten: Wir können die Frage ‘‘`?-wasserlebend(T)`’’ *nicht* beantworten. Da die ursprüngliche Aussage auch äquivalent mit

$\forall$ `M: (tier(T)` $\wedge$ $\neg$ `landlebend(T))` $\rightarrow$ `wasserlebend(T)`

ist, könnten wir versucht sein, einfach eine zweite Klausel

```
wasserlebend(T)  :- tier(T), \+landlebend(T).
```

anzufügen. Damit haben wir natürlich ein Programm geschaffen, das garantiert nicht terminiert, sofern wir sinnvollerweise zu erwartende Fakten der Art ‘‘`tier(bello1)`’’ und/oder ‘‘`tier(wanda1)`’’ (eins genügt) noch einfügen:

```
landlebend(T)  :- tier(T), \+wasserlebend(T).
wasserlebend(T)  :- tier(T), \+landlebend(T).
tier(bello1).
tier(wanda1).
```

Die negative Information, dass Hunde nicht Wasserlebewesen sind, kann eben nicht direkt in die Datenbank geschrieben werden.

Wenn wir hingegen die logische Negation als explizites *Prädikat* (resp. als Operator, wie oben eingeführt) ‘‘`neg`’’ repräsentieren, geht alles gut (cf. Kowalski 1979:147):

```
landlebend(T)     :- tier(T), neg wasserlebend(T).
wasserlebend(T)   :- tier(T), neg landlebend(T).
tier(bello1).
tier(wanda1).

neg wasserlebend(bello1).
neg landlebend(wanda1).
```

Weshalb funktioniert das, während wir bei der Formulierung mit ‘‘\+’’ in eine unendliche Schleife geraten? Wir müssen uns daran erinnern, dass Prolog keine funktionale Sprache ist, d.h. dass eingebettete Ausdrücke nicht etwa evaluiert werden. Weil wir `neg` als ganz normales Prädikat definiert haben, wird der Ausdruck `neg wasserlebend(hund)` in der beschriebenen Weise als unanalysierte Struktur

behandelt, d.h. der ‘‘Kern’’ `wasserlebend(hund)` wird bloss als ‘‘zusätzliche Struktur’’ behandelt, die beim Unifizieren berücksichtigt werden muss. In Tat und Wahrheit hätten wir ohne weiteres ein neues, homogenes, Prädikat `nichtwasserlebend(X)` (und analog natürlich `nichtlandlebend(X)`) verwenden können, ohne dadurch funktional etwas zu verlieren (oder zu gewinnen); beim Adjektiv ‘‘intelligent’’ haben wir ja genau das getan. In diesem Sinne ist der Operator ‘‘neg’’ reiner syntaktischer Zucker.

Die Prolog-‘‘Negation’’ hingegen ist *nicht* ein normales Prädikat. Sie ist eine funktionale Enklave in einem ansonsten relationalen Land (nicht ganz die einzige: ‘‘is’’ und einige andere arithmetische Prädikate sind auch funktional). Wenn wir

```
?- \+ wasserlebend(hund).
```

aufrufen, wird der Ausdruck syntaktisch analysiert, wobei der innerste Kern (hier also ‘‘`wasserlebend(hund)`’’) gefunden wird, und der wird nun vorgängig evaluiert, und mit dem Resultat wird etwas getan (das Resultat ist ein Wahrheitswert, und der wird umgekehrt).

Leider heisst das nun, dass wir das Problem der disjunktiven Regelköpfe nicht im allgemeinen Fall gelöst haben, sondern bloss in dem einen betrachteten Fall. Wir müssen für jede Regel mit disjunktivem Kopf die skizzierte Transformation vornehmen. Mehr über diese Probleme in Naish 1986.

3.5.2 Allaussagen als Theoreme

Mit Hilfe der Simulation der logischen Negation durch Unbeweisbarkeit wird auch die Aussage

8) A supplier is preferred if all his supplies arrive on time

die wir oben als Beispiel für eine Aussage angeführt hatten, die wir zwar in der Prädikatenlogik, nicht aber in der Horn-Klausel-Logik ausdrücken können, in (unreinem) Prolog ausdrückbar.

Was führte bei diesem Beispiel dazu, dass man es nicht in Horn-Klausel-Logik darstellen konnte? Die Tatsache, dass man eine Allquantifikation zu beweisen versuchte: Obwohl ein Aussagesatz, umfasst sie im Konditionalsatz ebenfalls eine Allaussage als zu beweisendes Theorem: ‘‘A supplier is preferred if *all* his supplies arrive on time’’. Betrachten wir deshalb einen einfacheren Fall, der aber das selbe Problem repräsentiert, nämlich die Frage ‘‘Sind alle Menschen intelligent?’’. Wir wissen, dass der *Aussage*satz ‘‘Alle Menschen sind intelligent’’ in Horn-Klausel-Logik üblicherweise dargestellt wird als

```
intelligent(M)  :-      human(M).
```

aber was ist die *Frageform* dieses Satzes in Horn-Klausel-Logik? Wenn man die Prädikatenkalkülversion der Aussage *negiert* (wie man es tun muss, um daraus ein durch Refutations-Resolution beweisbares Theorem zu erhalten) und dann in *Klausel-Logik* konvertiert, ergibt sich daraus folgendes:

```
   human(sk1).
:- intelligent(sk1).
```

Dies sollte ein zu beweisendes Theorem sein, aber in der ersten Zeile *fehlt* leider *die Negation*. Deshalb lässt sich das nicht direkt als Prolog-Frage behandeln. Wie soll man das aber sonst interpretieren?

Z.B. als Frage, ob alle individuell bekannten Menschen einzeln auch als intelligent bekannt sind, ob mit anderen Worten unsere Datenbank etwa so aussieht (hier ein Prolog-Version davon):

```
human(peter).
human(jill).
human(john).

intelligent(peter).
intelligent(jill).
intelligent(john).
```

Diese *extensionale* Interpretation der Frage ist deutlicher formuliert so:

17) Is *every* human being intelligent?

18) Ist *jeder* Mensch intelligent?

Wie müsste man eine Datenbank (resp. in unserem Kontext: ein Prolog-Programm) daraufhin untersuchen, ob zu jedem Eintrag `human(X)` auch ein Eintrag `intelligent(X)` zu finden ist?

Anstatt, wie üblich, die prädikatenlogische Ausgangsaussage zu negieren

¬ ∀ `X: human(X)` → `intelligent(X)`

und daraus dann `human(sk1).  :- intelligent(sk1).` abzuleiten, leitet man ab aus der *unnegierten* Aussage die äquivalente Aussage[8]

```
∀ X: ¬( human(X) ∧ ¬ intelligent(X)))
```

macht das zur Klausel, *ohne den Skopus der Negation zu minimieren*

```
¬( human(X), ¬ intelligent(X))
```

ab, negiert *die*

```
¬ ¬( human(X), ¬ intelligent(X))
```

und erhält so die folgende intuitiv sehr einleuchtende Prolog-Formulierung für die extensionale Version der Frage:

```
?- \+((human(X), \+intelligent(X)))
```

Nunmehr springt der Prolog-Interpretierer abwechslungsweise von `human(X)` zu `intelligent(X)` und zurück und testet für jedes Paar, ob beide Bedingungen erfüllt sind (er simuliert zwei Co-Routinen). Nur, wenn das der Fall ist, wird er mit einem Erfolg zurückkommen.

Kehren wir zurück zur Ausgangsaussage ‘‘A supplier is preferred if all his supplies arrive on time.’’ Wenn man den Konditionalsatz *extensional* auffasst, kann man diese Aussage in Prolog nunmehr so schreiben:

```
preferred(X)  :-  supplier(X),
                  \+((supply(Y,X), \+on_time(Y))).
```

8. aufgrund von:

```
p → q ↔
¬ p ∨ q ↔
¬ ¬ ( ¬ p ∨ q) ↔
¬ (p ∧ ¬ q)

(denn ¬ (p ∨ q) ↔ (¬ ∧ ¬ q))
```

3.5.3 Allaussagen vs. Regelaussagen

Die Frage ‘‘Sind alle Menschen intelligent?’’ hat aber auch eine andere Lesart. Wir können damit erfahren wollen, ob es eine *allgemeine Regel* gibt, welche aussagt, dass alle Menschen im Prinzip intelligent sind, ob Intelligenz also zu den definierenden Eigenschaften der Spezies Mensch gehört. Dies ist die *intensionale* Version der Frage. Auf Englisch könnte man diesen Aspekt verdeutlichen, indem man fragt

19) Are humans intelligent?

Hier müssen wir testen, ob unser Programm die obige *Regel* `intelligent(M) :- human(M).` enthält.

Man kann also sowohl *intensionale* wie auch *extensionale* Fragen stellen, und oftmals können wir ein und dieselbe natürlichsprachliche Frage so oder so interpretieren. Die Frage ‘‘Sind alle Menschen intelligent?’’ ist ein gutes Beispiel für eine in dieser Weise ambige Frage.

Wie sieht nun aber die intensionale Version einer Frage in Prolog aus?

Erste Idee: ‘‘schmutziger Trick’’ mit `clause(H,B)`. Dieses Prädikat gehört nicht mehr zum ‘‘reinen Prolog’’, weil es Regeln nicht in der normalen Weise interpretiert, nämlich als Inferenzregel beim Theorembeweisen verwendet (also ausführt), sondern sie gleichsam als Fakten behandelt, die man nur nachschaut. Man würde dann schreiben

```
?- clause(intelligent(M),human(M)).
```

Allerdings hat dieser Ansatz einen grossen Nachteil: Wenn man im zweiten Argument mehr als einen Eintrag hat (wenn man also nach einer Regel sucht, welche im Rumpf mehrere Terme aufweist), dann muss man beim Aufruf mit ‘‘`clause(H,B)`’’ die richtige Reihenfolge dieser Terme kennen, sonst gibt’s keine Unifikation. Diese Bedingung ist naheliegenderweise kaum je zu erfüllen.

Eine bessere Lösung ergibt sich aus der obigen Transformation der Prädikatenkalkül-Version in Non-Horn-Klausel-Logik, also aus (hier wiederholt)

```
   human(sk1).
:- intelligent(sk1).
```

sofern man die zweite Zeile wie üblich als normales Prolog-Ziel, die erste Zeile hingegen als *temporären Eintrag im Programm* interpretiert, der nur während der

Dauer des Beweises dieser einen Frage dort verbleibt. Das könnte man sehr einfach so implementieren:

```
?- assert(human(sk1)), intelligent(sk1),
   retract(human(sk1)).
```

So erhält man das erwünschte Verhalten: Da ‘‘`sk1`’’ als Skolem-Konstante einmalig ist und garantiert nirgendwo sonst im Programm vorkommt, ist die Frage ‘‘`?- intelligent(sk1)`’’ nämlich *nur* beweisbar, wenn irgendwo im Programm die Regel

```
intelligent(M)  :-      human(M).
```

vorkommt, welche *auch* für diese Skolem-Konstante funktioniert. Und das ist ja genau das, was wir mit der intensionalen Version der Frage erfahren wollten. Natürlich ist das temporäre Erweitern des Programms durch Einträge ein ausserlogischer Schritt, was nicht erstaunt, da wir ja eine Nicht-Horn-Klausel verarbeiten müssen.

Bei dieser Methode ist das obengenannte Problem der richtige Reihenfolge von RHS-Termen inexistent: Man assertiert alle diese Terme für sich, einen nach dem andern, und der Interpretierer findet sie dann beim normalen Interpretieren der zu beweisenden Regel alle, in welcher Reihenfolge man sie auch assertiert habe.

Wenn man versucht, diese Erkenntnis nun auf den Ausgangssatz anzuwenden, wenn man also den Konditionalsatz in ‘‘A supplier is preferred if all his supplies arrive on time’’ intensional auffasst (das könnte man vielleicht verdeutlichen, indem man sagt ‘‘A supplier is preferred if *supplies from him* arrive on time’’), könnte man folgende direkte Uebersetzung in Prolog verwenden:

```
preferred(X) :- supplier(X),
                assert(supply(sk1,X)), on_time(sk1),
                retract(supply(sk1,X)).
```

Damit wäre die Anfrage

```
?- preferred(jones).
```

nur dann positiv beantwortbar werden, wenn im Programm tatsächlich eine Regel

```
on_time(Y)      :-      supply(Y,jones).
```

vorkommt. In diesem konkreten Fall ist die intensionale Interpretation wohl etwas gezwungen, aber im folgenden Beispiel ist sie die intendierte:

Ein Mensch ist ein Genie, wenn er prinzipiell jedes Problem lösen kann

also

```
genius(X)  :-   human(X),
                assert(problem(sk1)),
                can_solve(X,sk1),
                retract(problem(sk1)).
```

Man kann sich die ganze Sache vereinfachen, indem man ein für alle Mal folgendes Prädikat für generelle Regeln (also Allquantifikation plus Implikation) definiert:

```
all(intensional,X,Y) :- skolemize_free_vars(X,Z),
                        assert(Z), prove(Y),
                        retract(Z).
all(extensional,X,Y) :- \+((prove(X), \+(prove(Y)))).
```

Beachte: Diese Definition von `all(intensional,X,Y)` würde Skolem-Konstanten erzeugen, die sich nach links und oben verbreiten, was sehr unerwünscht ist.

Lösung: Das Ganze in Doppelnegation verpacken.

Obiges Beispiel würde dann so in das Programm eingefügt

```
genius(X) :- human(X),
        all(intensional, problem(Y), can_solve(X,Y)).
```

Wenn man eventuell nicht weiss, ob eine Frage intensional oder extensional gemeint ist, kann man das erste Argument einfach offenlassen:

```
genius(X)  :-   human(X),
                all(_, problem(Y), can_solve(X,Y)).
```

Nunmehr wird zuerst die intensionale Interpretation versucht, und wenn die zu nichts führt, die extensionale. Das scheint auch die intuitiv einleuchtende Reihenfolge zu sein: Wenn man die zweideutige Frage

Sind alle Menschen intelligent?

stellt, wird der Adressat sicher zuerst prüfen, ob er eine allgemeine Regel dieses Inhalts kennt, und nur, wenn das nicht der Fall ist, wird er sich an die viel aufwendigere extensionale Durchmusterung seiner mentalen Datenbank machen.

Hintergrundinformation (0.L.5)

Also:

- Zwei Arten von Allgemeinheit
- Unterscheidung wird in der Logik selbst nicht gemacht (ist immer ∀)

Siehe unten ein wenig genauer.

4. Das Syntax-Semantik-Interface

4.1 Diskrepanz von syntaktischer und logischer Struktur

Repetition aus ECL2: Ausgangspunkt waren Sätze wie

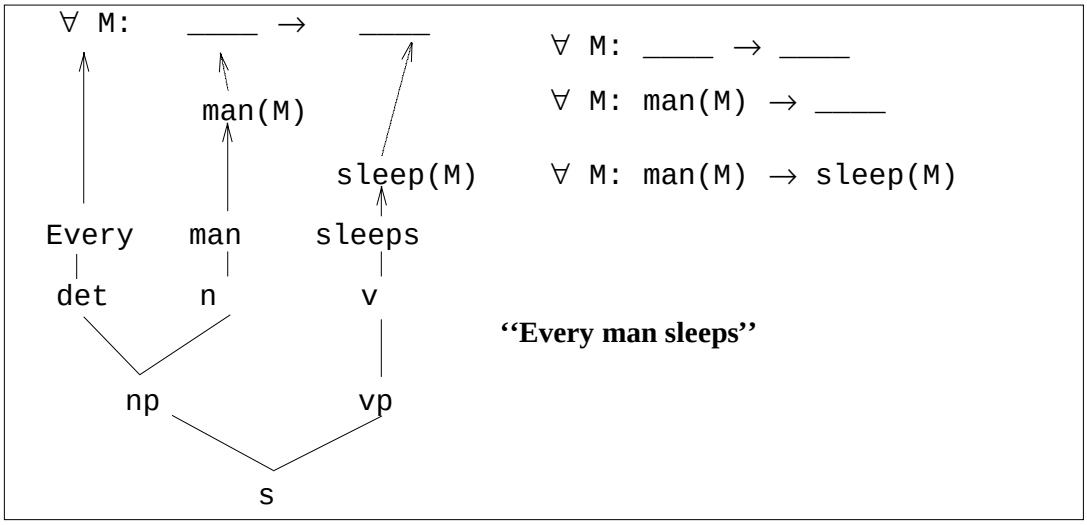
20) Every man sleeps

welche so übersetzt werden müssen:

20a) ∀ M: (man(M) → sleep(M))

sequentielles Übersetzen in Logische Formen?

Wie kommt man von der Syntaxstruktur zur LF? Einfaches sequentielles Vorgehen mit ‘‘Auffüllen von Lücken’’ scheint möglich zu sein:



Mehrere Quantoren

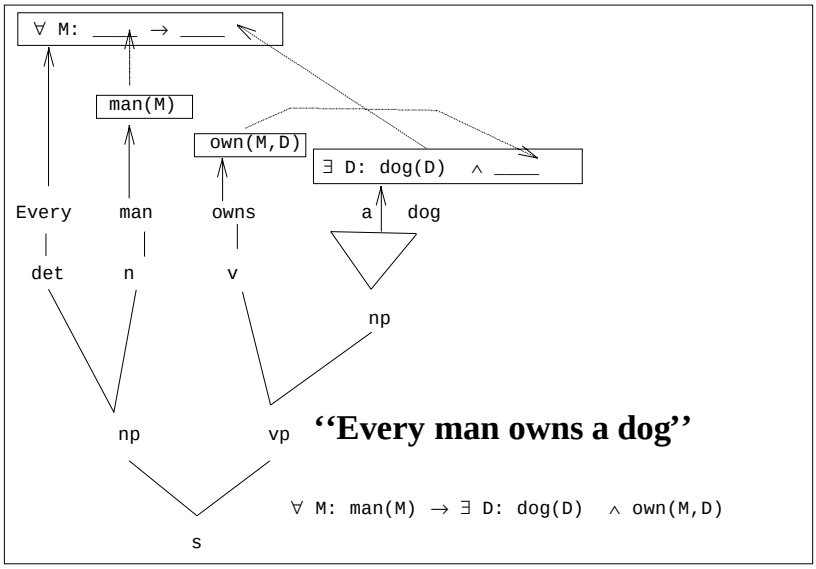
Da nun die Verbalphrase ihrerseits oft aus Verb plus Nominalphrase bestehen kann, ergeben sich aber erste Komplikationen: Wir wissen intuitiv, dass Beispiel 21

21) Every man owns a dog

als 21a

21a) ∀ M: (man(M) → ∃: (dog(D) ∧ own(M,D)))

übersetzt werden muss. In der zweiten Lücke steht nunmehr eine eingebettete Quantifikation, und das own(M,D) ist gleichsam ‘‘in sie hinein gerutscht’’.



Eigennamen

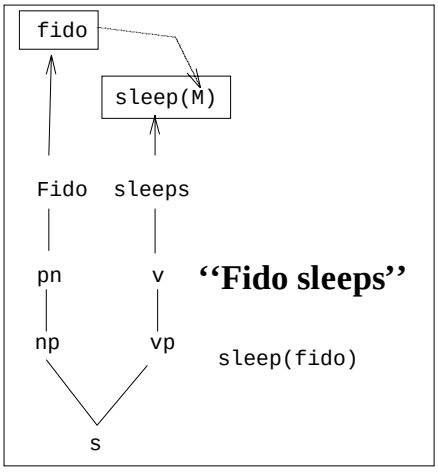
Weitere Komplikationen: Wenn das Subjekt eines Satzes ein Eigenname ist, muss ebenfalls eine (elementare) Form des ‘‘Auffüllens von Lücken’’ durchgeführt werden:

22) Fido sleeps

wird ja übersetzt als

`22a) sleep(fido)`

Aber hier wird die Variable ‘‘von hinten’’ gefüllt, d.h. es ist die Verbalphrase des Prädikats, welche einen Ausdruck mit Lücke schafft, und die Lücke muss von der Nominalphrase des Subjekts gefüllt werden. Was dies bedeutet ist, dass die Übersetzung von ‘‘Fido’’ (also `fido`) irgendwie zwischengespeichert werden muss, bis das Verb übersetzt worden ist; erst dann ist ja eine Variable vorhanden, in die man `fido` ‘‘einfüllen’’ kann:



strukturelle Ambiguitäten

Schliesslich gibt es noch eine andere und weit grundsätzlichere Komplikation: Das Beispiel

23) Everybody in this room speaks some unusual language

ist bekanntlich auf zwei verschiedene Arten interpretierbar (auch wenn die erste Interpretation deutlich bevorzugt wird):

```
23a) ∀ P: (person_in_this_room(P) →
       ∃ L: (unusual_language(L) ∧ speak(P,L)))
```

und

```
23b) ∃ L: (unusual_language(L) ∧
       ∀ P: (person_in_this_room(P) → speak(P,L)))
```

4.2 Probleme der direkten Übersetzung von Syntax in Semantik

Die fünf Schritte zur Erleuchtung

Vorgehen: Prolog ist gut im Struktur-Transformieren.

1. **Daher:** Übersetzung direkt machen, durch Einsatz aller Prolog-Tricks.
   - **Erkenntnis:** Wird sehr bald extrem komplex.
2. **Daher:** Montague-Semantik streng nach Montague implementieren.
   - **Erkenntnis:** Geht, aber ist bekanntlich auch nicht einfach
   - **Aber:** Kann durch den Einsatz der Unifikation vereinfacht werden.
3. **Daher:** Montague-Semantik mit Unifikation implementieren.
   - **Erkenntnis:** konzeptuell wesentlich einfacher
   - **Aber:** Programm wird etwas ineffizient.
4. **Daher:** Partielle Evaluation verwenden
   - **Erkenntnis:** Geht.
   - **Aber:** Termunifikation ist unnötig explizit
5. **Daher:** Man könnte Merkmals-Unifikation verwenden

Dazu siehe auch ausführlicher den oben erwähnten Text von Blackburn & Bos ⇒hier, v.a. Band 1 ab Seite 29.

4.2.1 Ein Programm zur direkten Übersetzung

Der *‘‘Let’s do it’’*-Ansatz

Unser erster Versuch, diese Gesichtspunkte (aber ohne Skopusambiguitäten) in einer Implementation zu berücksichtigen, soll möglichst kompakt sein. Wir wollen ein Programm vorstellen, das in einem einzigen Arbeitsgang einfache englische Aussagesätze in Aussagen der Logik erster Stufe übersetzt. (Es soll also ein sog. ‘‘one-pass’’-Übersetzungsprogramm sein). Im folgenden sind für einige einfache Beispiele aufgelistet je:

1. der Eingabesatz,
2. die Übersetzung in einer Logik erster Stufe
3. die (vom oben vorgestellten Klausifizierungprogramm erstellte) Version in Klausel-Logik

Die verwendete Notation für die Logik erster Stufe und ihre Entsprechungen zur Standardform sind (nur die ersten drei Einträge kommen in den Beispielen vor):

```
1.  exists(X):  <==>   ∃ X:
2.  all(X):     <==>   ∀ X:
3.  &           <==>   ∧
4.  #           <==>   ∨
5.  ->          <==>   →
6.  <->         <==>   ↔
```

Die Version in Klausel-Logik wird angeführt, um zu zeigen, dass man dieses Programm recht einfach zu einem eigentlichen Frage-Antwort-System machen könnte: Über den Klausel-Logik-Einträgen könnte man natürlich direkt Prolog-Ziele beweisen lassen (ausser, wenn eine Klausel nicht-Horn ist). Das Programm, das die Übersetzung aus Logik erster Stufe in Klausel-Logik durchführt, wird aber nicht besprochen werden, da es linguistisch nicht relevant ist[9]

Die folgenden Resultate sind nur ganz unwesentlich editiert worden (die internen Variablennamen wurden ersetzt durch die üblichen Grossbuchstaben). Das Programm in der gegenwärtigen Form verarbeitet nur deklarative Sätze. **Beispiele:**

```
1: every department employs a tutor

all(M): (department(M) -> exists(W): (tutor(W) &
employs(M,W)))

tutor(sk12(M)) :- department(M).
employs(M,sk12(M)) :- department(M).
```

9. und in Clocksin/Mellish, pp. 269 sqq., abgedruckt und kommentiert wird

```
2: every department that prospers employs a tutor

all(M): (department(M) & prospers(M) ->
        exists(W): (tutor(W) & employs(M,W)))

tutor(sk7(M)) :- department(M), prospers(M).
employs(M,sk7(M)) :- department(M), prospers(M).
```

Die Beispiel-Sätze sind etwas blöd,...

```
3: every department employs a tutor that employs a
   student

all(M1): (department(M1) -> exists(W): ((tutor(W) &
        exists(M2): (student(M2)
                & employs(W,M2))) & employs(M1,W)))

tutor(sk8(M)) :- department(M).
student(sk9(M)) :- department(M).
employs(sk8(M),sk9(M)) :- department(M).
employs(M,sk8(M)) :- department(M).
```

```
4: a department that employs a tutor prospers

exists(M): ((department(M) & exists(W): (tutor(W)
         & employs(M,W)))  & prospers(M))

department(sk10).
tutor(sk11).
employs(sk10,sk11).
prospers(sk10).
```

... und z.T. auch etwas künstlich.

```
5: every department that employs a tutor that employs
   john employs john

all(M):(department(M) & exists(W):((tutor(W) &
  employs(W,john)) & employs(M,W)) -> employs(M,john))

employs(M,john) :-  department(M), tutor(W),
                    employs(M,W), employs(W,john).
```

Beispiel 4 ist insofern interessant als seine Übersetzung in Logik intuitiv nicht sehr befriedigend ist: Sie bedeutet ja ‘‘There is a department that employs a tutor and prospers’’, und das ist eine sehr periphere Lesart des Satzes. Der Grund dafür ist, dass dieses Programm den unbestimmten Artikel ‘‘a’’ gleich behandelt wie ‘‘some’’, also als englische Version des Existenzquantors. Um zu ahnen, was da alles sonst noch mitspielt, vergleiche man die Bedeutung von Beispiel 4 mit der modifizierten Version

A department that employed a tutor (now) prospers

Der Wechsel des Tempus macht plötzlich die verwendete Übersetzung wesentlich treffender (wenn man das Tempus irgendwie ausdrückt).

Dies ist das Programm:[10]

Die Definition der Operatoren (0.L.6)

10. ursprünglich nach L.M. Pereira, F.C.N. Pereira, und D.H.D. Warren, User’s Guide to DECsystem-10 Prolog, danach oft abgedruckt.

    Das ganze Programm, zusammen mit dem Klausifikationsprogramm von oben, findet sich ⇒ hier.

```
<Hier: Definition der Operatoren>

sentence(P)         -->  noun_phrase(X,P1,P),
                         verb_phrase(X,P1).

noun_phrase(X,P1,P) -->  determiner(X,P2,P1,P),
                         noun(X,P3), rel_clause(X,P3,P2).
noun_phrase(X,P,P)  -->  name(X).

verb_phrase(X,P)    -->  trans_verb(X,Y,P1),
                         noun_phrase(Y,P1,P).
verb_phrase(X,P)    -->  intrans_verb(X,P).

rel_clause(X,P1,P1&P2)  -->  [that], verb_phrase(X,P2).
rel_clause(_,P,P)       -->  [].

determiner(X,P1,P2, all(X): (P1 -> P2)) --> [every];[all].
determiner(X,P1,P2, exists(X): (P1&P2)) --> [a];[some].

noun(X, student(X) )            -->  [student].
noun(X, department(X) )         -->  [department].
noun(X, tutor(X) )              -->  [tutor].

name(john)                      -->  [john].

trans_verb(X,Y, employs(X,Y) )  -->  [employs].
intrans_verb(X, prospers(X) )   -->  [prospers].
```

4.2.2 Kommentar zum Programm

Das Programm funktioniert offenbar, aber wie?!

Eine in diesem Programm immer wieder verwendete Möglichkeit von Prolog ist der Einsatz von (noch) teilweise oder ganz ungebundenen Variablen in Berechnungen.

Beispiel: In

```
sentence(P)             -->  noun_phrase(X,P1,P),
                             verb_phrase(X,P1).

noun_phrase(X,P1,P)     --> determiner(X,P2,P1,P),
                            noun(X,P3),
                            rel_clause(X,P3,P2).
```

wird die Variable `P1` in einer als grammatikalisches *Subjekt* verwendeten `noun_phrase` erst *nachträglich*, auf der Ebene der Regel `sentence` mit dem *erst später errechneten* Ergebnis der Interpretation der Verbalphrase gefüllt werden.

Erst dann wird sich dieser Wert auch in die schon lange geschaffene, aber noch offengehaltene Lücke in der von `determiner` geschaffenen Quantifikation ausbreiten. `P3` wird von `noun` an `rel_clause` übergeben und dort an die Interpretation des Relativsatzes angefügt, worauf das Resultat dieser Kombination (genauer: Verknüpfung durch die Konjunktion "&") an `determiner` zurückgegeben und dort (wiederum nachträglich) in den entsprechenden Teil des Quantifikations-Skeletts eingebaut wird. Die logische Variable `X` stellt sicher, dass über derselben Variable quantifiziert werden wird.

In andern Fällen werden Variablen *teilweise* gebunden und dann weiterverwendet:

```
verb_phrase(X,P)        -->  trans_verb(X,Y,P1),
                             noun_phrase(Y,P1,P).
verb_phrase(X,P)        -->  intrans_verb(X,P).
```

Hier wird die `noun_phrase` als grammatikalisches *Objekt* verwendet. Die vom Determiner dieser Nominalphrase geschaffene Quantifikation wird vom vorangehenden transitiven Verb *teilweise* aufgefüllt (`P1`).

Die Nominalphrase kreiert sich gleichzeitig eine *neue* logische Variable (`Y`; neu, da nicht mehr über dasselbe quantifiziert wird, wie im Subjekt des Satzes); hingegen muss das zweite Argument des aus dem transitiven Verb hervorgehenden Prädikats mit dieser Variable übereinstimmen: In ‘‘employs a tutor’’ wird das Prädikat `employs(X,Y)` in die logische Variable `P1` der von ‘‘a’’ geschaffenen existentiellen Quantifikation `exists(X): (P0 & P1)` eingefüllt werden. Da zu diesem Zeitpunkt `P0` schon vom Wert der Übersetzung des Substantivs, also `tutor(Y)` gefüllt sein wird, ergibt das dann `exists(Y): (tutor(Y) & employs(X,Y))`. Dass die logische Variable `Y` korrekt ist, wird ebefnalls in der Regel oben

```
verb_phrase(X,P) -->  trans_verb(X,Y,P1),
                        noun_phrase(Y,P1,P).
```

sichergestellt.

Besonders klar wird die Verwendung teilweise instantiierter Variablen in

```
determiner(X,P1,P2, all(X): (P1 -> P2) ) -->  [every];[all].
determiner(X,P1,P2, exists(X): (P1&P2) ) -->  [a];[some].
```

Hier werden die ‘‘Skelette’’ für die Grundstruktur der logischen Aussagen bereitgestellt. Die Variablen `P1` und `P2` ‘‘pumpen’’ den Inhalt von den übrigen Bestandteilen des Satzes in diese Skelette hinein. Die Variablen `X` sollen sicherstellen, dass in den verschiedenen Teilen der logischen Repräsentation die gleichen Variablen verwendet werden.

Scheinbar offesichtlich sind dann

```
noun(X, student(X) )            -->  [student].
noun(X, department(X) )         -->  [department].
noun(X, tutor(X) )              -->  [tutor].

name(john)                      -->  [john].

trans_verb(X,Y, employs(X,Y) )  -->  [employs].
intrans_verb(X, prospers(X) )   -->  [prospers].
```

Beachte aber: Hier kommen *nur* logische Variablen vor, keine meta-logischen. Siehe gleich unten.

Einige einfache Traces (0.L.7)

Vermischung von Sprache und Meta-Sprache...

Wichtige **Beobachtung:** Die Lesbarkeit des Programms ist besonders gering, weil (Objekt-)Sprache und Meta-Sprache vermischt werden müssen, um alles in einem einzigen Pass übersetzen zu können.

...ist ‘‘elegant’’, aber...

...verwirrlich.

Konkret geschieht dies, indem das Programm Prolog-Variablen verwendet sowohl, um über *logische Strukturen* zu sprechen (Variablen der Form `P, P1, P2` etc.) wie auch, um über *Objekte* der Welt zu sprechen (Variablen der Form `X, Y, Z` etc.). Die ersten Variablen sind also *meta-logisch* (sie betreffen Objekte der Sprache, d.h. der Logik, und sie können komplette logische Ausdrücke, oder unfertige Bestandteile solcher Ausdrücke, enthalten), die zweiten sind logisch (eigentlich ‘‘*objekt-logisch*’’ - ist aber ein unüblicher Begriff). Nur die logischen Variablen erscheinen als solche in der logischen Repräsentation. Dies ist die Kehrseite des eleganten Tricks, mit u.U. unter-instantiierten Variablen zu arbeiten, um logische Strukturen aufzubauen.

Die Probleme, die beim Schreiben eines derartigen Ein-Pass-Übersetzers entstehen, dürften damit zur Genüge illustriert worden sein. Die ingenieurmässigen ad hoc-Lösungen führen bei einem solchen Vorgehen innert kürzester Zeit zu völlig unverständlichen Programmen. Wer sich noch nicht geschlagen gibt, versuche, das Programm so zu erweitern, dass man auch *objektmodifizierende* Relativsätze verwenden kann.

Eine Erweiterung der naiven direkten Übersetzung (0.L.8)

Ein alter Freund: Kompositionalität

Was wir brauchen, ist ein Verfahren, das die Syntaxstruktur wenn möglich Teil um Teil in selbständige logische Komponenten übersetzt, die dann, miteinander kombiniert, automatisch die richtige Gesamtübersetzung ergeben, inklusive richtige Variablen. Auf diese Weise könnte man auch eine einmal ermittelte Übersetzung in allen möglichen Kontexten verwenden, statt immer auf die möglichen Interaktionen achten zu müssen. Dies heisst überdies, dass jeder (wohlgeformte) syntaktische Teilausdruck eine semantische Interpretation haben sollte. Dies ist das *Prinzip der Kompositionalität*.

Zur Erinnerung: Kompositionalität (0.L.9)

Und das ist exakt, was Richard Montague anfangs der Siebzigerjahre zu skizzieren versucht hat: Das Kompositionalität in der Analyse der Semantik der natürlichen Sprache konsequent anzuwenden. Ein grosser Teil der linguistischen Semantik beschäftigt sich seither damit, diese Gedanken zu erweitern.

4.3 Implementationen der Montague-Semantik

4.3.1 Repetition: Die Grundidee

Nochmals der erste Satz aus Montagues ‘‘English as a Formal Language’’[11]

> I reject the contention that an important theoretical difference exists between formal and natural languages.

11. : abgedruckt in Thomason 1974, pp. 188-221

Montague: Beziehung Syntax-Semantik ist 1:1 !

Montagues zwei Grundkonzepte:

1. *Prinzip der Kompositionalität:*

   Die Bedeutung eines Satzes lässt sich ermitteln *allein* aus

   a. der Bedeutung seiner Bestandteile und

   b. der Art ihrer Kombination (d.h. aus der Syntax des Satzes)

***Zwei* Grundkonzepte**

2. *Prinzip der funktionalen Applikation:*

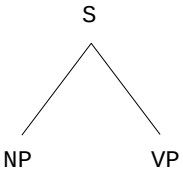
   Bedeutungen können (im wesentlichen) *nur* durch funktionale Applikation (oder ‘‘funktionale *Anwendung*’’) kombiniert werden

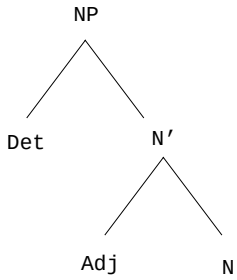
Konsequenzen:

1. Bedeutung bestimmter Konstruktionen muss nur *einmal* ermittelt werden
2. *jede* Konstituente hat eine Bedeutung
3. Übersetzung in Logik kann sogar ohne Syntaxanalyse geschehen: ‘‘Logik als Syntax’’

G

Prinzip der Kompositionalität,
angewendet auf die natürliche Sprache





[S] = [NP] ⊕ [VP]

[N'] = [Adj] ⊕ [N]

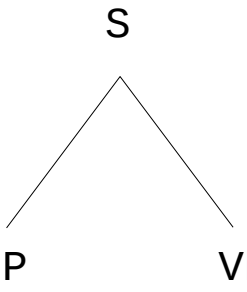
[NP] = [Det] ⊕ [N']

etc.

Gegenbeispiele: idiomatische Wendungen

ins Gras beissen
lügen, dass sich die Balken biegen
Sand in die Augen streuen

Funktionale Applikation als Interpretationsfunktion



[[S]] = [[NP]] ⊕ [[VP]]

Funktionswert = Funktor(Argument)

Funktionswert = (tnemugrA)rotknuF

Für die Bedeutung des Satzes ‘S’ stehen also zur Auswahl

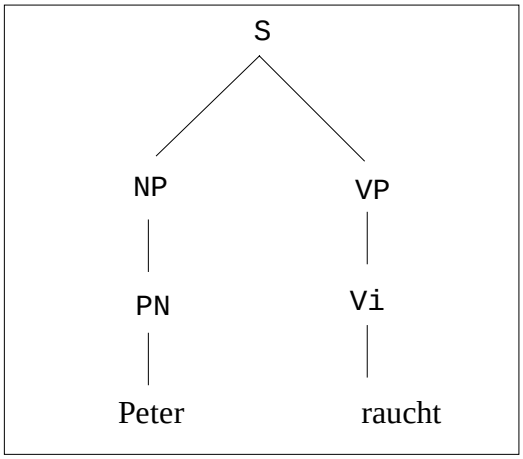
funktionale Anwendung geht entweder → oder ←

```
entweder:   1) ⟦NP⟧(⟦VP⟧)
oder:       2) ⟦VP⟧(⟦NP⟧)
```

mit der Interpretation:

```
1)  VP ist ein Element der Menge der NPs
2)  NP ist ein Element der Menge der VPs
```

Ein (scheinbar) besonders einfaches **Beispiel:**



Erste Idee war:

‘Peter raucht’’ nach Regel (2) oben zu interpretieren:
raucht′(peter′) = ‘‘Peter ist ein Element der Menge der Raucher’’

Daraus ergibt sich:

⟦VP⟧(⟦NP⟧)

Ein erster Versuch...

Wir verallgemeinern dies zu

(Tentative) allgemeine Übersetzungsregel:

A B wird übersetzt als ***B′(A′)***

mit ***A, B*** Konstituenten,
und ***X′*** die logische Übersetzung von ***X***

Leider funktioniert das nicht für *allgemeine* Nominalphrasen.

...und ein zweiter.

Deshalb brauchen wir eine andere Übersetzungsregel:

G

(Extensionale) allgemeine Übersetzungsregel:

A B wird übersetzt als ***A'(B')***

mit ***A, B*** Konstituenten,
und ***X'*** die logische Übersetzung von ***X***

Man erinnere sich: Oben hatten wir tentativ die Übersetzung

⟦VP⟧(⟦NP⟧)

verwendet. Jetzt gilt also neu

⟦NP⟧(⟦VP⟧)

Nun ergibt sich also für ‘‘Jeder Hund bellt’’:

jeder' = λ P. (λ Q. ∀ X: P(X) → Q(X))

woraus sich dann für ergäbe, der Reihe nach zuerst für ‘‘jeder Hund’’

λ P. (λ Q. ∀ X: P(X) → Q(X)) (hund') ⇒
λ Q. ∀ X: (hund'(X) → Q(X))

(die Eigenschaft, welche allen Hunden zukommt), und dann für ‘‘(jeder Hund) bellt’’

λ Q. ∀ X: (hund'(X) → Q(X) (bellt')) ⇒
∀ X: (hund'(X) → bellt'(X))

Eigennamen: ein problematischer Fall von Nominalphrase

Schliesslich musste man für den (vermeintlich besonders einfachen) Satz von oben (‘‘Peter raucht’’) eine neue Übersetzung für Eigennamen entwerfen:

λ P. P(john')

Dann wird daraus, bei der Umsetzung der Konkatenation als funktionale Applikation, für ‘‘John schläft’’:

λ P. P(john')(schläft')

und nach Beta-Reduktion dann genau

```
schläft'(john')
```

Das Verfahren der konsequent kompositionalen Analyse hat u.a. den folgenden Vorteil: Wenn man nämlich den Aufbau der logischen Strukturen exakt im Gleichschritt mit dem Analysieren der Syntax des natürlichsprachlichen Satzes vornimmt, und wenn die natürliche Sprache als (im Prinzip) direkt interpretierbar behandelt wird, dann ist auch jede logische Teil-Struktur, die aufgebaut worden ist, interpretierbar. Dies war nicht möglich bei der ‘‘naiven’’ direkten Übersetzung vom Englischen in Logik erster Stufe, die wir oben vorgestellt hatten: Dort schaffte man im Laufe der Analyse *Strukturen, die keine Interpretation* hatten. Was heisst das? Die Variable ‘‘P’’ im Term ‘‘determiner’’ in der Regel

```
noun_phrase(_,X,P1,P) -->  determiner(X,P2,P1,P),
                           noun(X,P3),
                           rel_clause(X,P3,P2).
```

wird zum Beispiel eine *unfertige Struktur* (ein ‘‘Skelett’’ einer Quantifikation) enthalten, wenn dieser Term abgearbeitet worden ist. Bei einer Nominalphrase mit unbestimmtem Artikel oder mit ‘‘some’’ wird dies die Struktur

```
exists(X):(P1 & P2)
```

sein. Dieses Skelett wird erst in den *folgenden* Schritten ‘‘aufgefüllt’’ werden. So, wie er hier steht, kann diesem Ausdruck kein Wert über der Datenbank zugewiesen werden (und das heisst, er kann semantisch nicht interpretiert werden). Erst einer vollständig ausgebildeten Quantifikation kann man einen semantischen Wert zuweisen (‘‘wahr’’ oder ‘‘falsch’’). Was ist daran störend?

Kontrolle des Parsing-Prozesses durch die Semantik: Ein mittelguter Grund für die Kompositionalität...

Man argumentiert oft, man könne beim Parsen manchmal ganze Aeste des syntaktischen Suchbaums als sinnlos ausfiltern, wenn man semantische Zwischentests einschaltet Ein Beispiel: Der Satz ‘‘Put down the red pyramid on the block behind the black box’’ ist bekanntlich mehrdeutig: Man kann ihn so verstehen, dass man jene der roten Pyramiden, die sich zur Zeit auf dem Klotz befindet, hinter die schwarze Schachtel stellen soll, oder aber, dass man die (einzige) rote Pyramide auf den schwarzen Klotz hinter der schwarzen Schachtel stellen soll. Diese syntaktische Mehrdeutigkeit lässt sich auflösen, indem man im Modell (oder, im einfacheren Fall, in der Datenbank) ‘‘nachschaut’’, ob es (genau) eine rote Pyramide auf einem Klotz gibt, oder nicht. Das heisst aber, dass man den Ausdruck ‘‘the red pyramid on the block’’ zu evaluieren versucht, und wenn man damit Erfolg hat, kann man die zweite syntaktische Analyse gleich vergessen. Wenn man sich nun die Möglichkeit offenhalten will, an jeder beliebigen Stelle der syntaktischen Analyse derartige

Evaluationstests einzuschalten, dann müssen alle Zwischenresultate der Übersetzung Syntax → Semantik über dem Modell interpretierbar sein. - Hier soll nicht verschwiegen werden, dass dieser Grund für strenge Kompositionalität zwar oft angeführt wird, aber dass fast nie die praktischen Konsequenzen daraus gezogen werden.

...und ein sehr guter Grund: Auswertung unvollständiger Syntaxanalysen

Es gibt aber einen sehr guten, sehr praktischen Grund, warum man dem Prinzip der Kompositionalität so lange nachleben sollte, wie es geht: In (allzu) vielen Fällen realer computerlinguistischer Anwendungen (z.B. Antwortextraktion) kann man einen gegebenen Satz nicht vollständig parsen. Man muss dann die Teilanalysen so weit, wie es eben geht, verwenden. Oft heisst das, einzelne Phrasen in ihre Logische Form zu übersetzen. Das kann man aber nur, wenn jede syntaktische Phrase eine klare Übersetzung in Logik hat. Eben: Kompositionalität

Um die bisher besprochenen Charakteristiken von Montagues Analyse zu verdeutlichen, soll im folgenden die Implementation einer extensionalen Version einer sehr einfachen Montague-Semantik vorgestellt werden, in der die traditionelle funktionale Applikation als Kombinationsinstrument verwendet wird.

4.3.2 Implementation mit funktionaler Applikation

Allerdings werden wir tatsächlich gewaltig vereinfachen müssen. Erstens wird der noch nicht besprochene Mechanismus zur Behandlung der Skopusambiguitäten natürlich nicht implementiert werden. Zudem wird die Trennung der syntaktischen Regeln in eine ID-Komponente und eine LP-Komponente ebenfalls nicht vorgenommen werden. Drittens werden wir, der leichteren Verständlichkeit halber, die traditionellen syntaktischen Kategorien verwenden. Dafür werden wir die Kompositionalität der Analyse ganz besonders klar darstellen, indem wir die logische Struktur gleichzeitig mit der fortschreitenden Syntaxanalyse aufbauen.

Wir müssen zuerst die Notation

prolog-kompatible Notation für ...

Prolog-kompatibel machen, d.h. die funktionalen Ausdrücke als relationale Ausdruck enkodieren:

1. die üblichen Varianten der Konnektoren und Operatoren der Logik erster Stufe

("&" statt "∧" etc.).

2. Statt

...Quantoren,...

```
exists(X): (P(X) & Q(X))
```

nunmehr

```
exists(X: (P(X) & Q(X)))
```

und analog für den Allquantor.

3. funktionale Applikation als explizites Prädikat , weil z.B.

...funktionale Anwendung,...

```
P(john)
```

syntaktisch nicht zulässig. Also ‘‘simulieren’’ wir die funktionale Applikation durch ein Prädikat @ und schreiben

```
@(P,john)
```

4. Da

...die ‘‘Doppel-argument-notation’’...

```
beat(john)(peter)
```

syntaktisch ebenfalls nicht zulässig, schreiben wir dafür

```
@(@(beat,john),peter)
```

5. Lambda-Abstraktionen ebenfalls als normale Prädikate: Statt

...und die Lambda-Abstraktion (in zwei Versionen:

```
λ X. (hund(X) ∧ rot(X))
```

können wir schreiben

etwas unhandlich und

```
lambda(X, (hund(X) & rot(X)))
```

oder (besser, weil einfacher):

deutlich ein-facher)

```
X^(hund(X) & rot(X))
```

6. Der Apostroph für die Übersetzung Englischer lexikalischer Einheiten wird unterdrückt.[12]

Nunmehr können wir zum Beispiel statt

```
λ P. λ Q. P(λ Y. (beat(Y))(Q))
```

schreiben

```
P^Q^@(P,Y^@(@(beat,Y),Q))
```

resp. (wenn als Prolog-Code zu verwenden, wegen der Präzedenzbeziehungen) so

```
P^Q^(@(P,Y^(@(@(beat,Y),Q))))
```

Das ist zwar etwas unübersichtlicher als vorhin (in der nicht ‘‘prologifizierten’’ Schreibweise), aber nicht allzu schlimm. Und für ‘‘*such that* every boy invited him/her’’ erhalten wir auf diese Weise aus

```
all(B: @(boy,B) -> @(@(invited,G),B))
```

die Lambda-Abstraktion

```
G^all(B: @(boy,B) -> @(@(invited,G),B))
```

Weiterhin gelten die folgenden Beziehungen: (**Achtung:** in der rechten Kolonne ist das Endresultat angegeben, nicht die (lambda-abstrahierte) LF des entsprechenden natürlichsprachlichen Ausdrucks)

12. Da man sowieso meist weiss, was gemeint ist. Zudem stiftet das Hochkomma, als Zitationszeichen, meist Chaos im Code. Manchmal verwendet man in Prolog den Operator ‘, z.B. `dog‘`

Beziehung
sprachliche Wortkategorien - logische Ausdrücke

```
Funktionswörter

  Determinatoren        ==>   ''Quantorenskelett''
                                von Sätzen

    ein(e)/einige/gewisse -->   ∃ V: P(V) ∧ Q(V)
    jeder/alle            -->   ∀ V: P(V) → Q(V)
    der/die (singular)    -->   ∃ V: ∀ Y: P(Y) ↔ V=Y)
                                                  ∧ Q(V)
                          (statt ↔ ist → implementiert!)

"und"                   -->     ∧
"oder"                  -->     ∨
"nicht"                 -->     ¬
"wenn ... dann"         -->     → (±!)

Eigennamen              ==>     Konstanten

Inhaltswörter           ==>     Prädikate

  Wetterverben          -->     regnet
  intransitive Verben   -->     schläft(X)
  mono-transitive V.    -->     schlägt(X,Y)
  bi-transitive Verben  -->     bevorzugt(X,Y,Z)
  tri-transitive Verben -->     transferiert(W,X,Y,Z)

  einf. Substantive     -->     mann(X)
  relationale Subst.    -->     freund(X,Y)
                        -->     vermittler(X,Y,Z)

  Adjektive (pos.)      -->     blau(X)
  Adjektive (komp.)     -->     grösser(X,Y)

  Präpositionen         -->    auf(X,Y)
                        -->     zwischen(X,Y,Z)
```

Testfragen und (echte) Ergebnisse, inkl....

Bevor wir das Programm selbst betrachten, hier einige Testfragen und ihre Ergebnisse und (in einigen Fällen) ihre Ableitungsgeschichte:

```
1: some department employs a tutor
```

```
''some'' + ''department'':

reduce(@(Q^P^exists(D: @(Q,D) & @(P,D)),department),C).
==>
C = P^exists(D: @(department,D) & @(P,D)))

''a tutor'':
R^exists(T: @(tutor,T) & @(R,T))

''employs'':
X^Y^@(X,Z^@(@(employ,Z),Y))

''employs'' + ''a tutor'':
reduce(@(X^Y^@(X,Z^@(@(employ,Z),Y)),
         R^exists(T: @(tutor,T) & @(R,T))),A).
==>
A = Y^exists(T: @(tutor,T) & @(@(employ,T),Y))

''some department'' + ''employs a tutor'':
reduce(@(P^exists(D: @(department,D) & @(P,D)),
         Y^exists(T: @(tutor,T) &
         @(@(employ,T),Y))),B).
==>
B = exists(D: @(department,D) & exists(T: @(tutor,T)
    &  @(@(employ,T),D)))

YES
```

...Auswertung über einer ''Datenbank''.

Die Antworten ''YES'' und ''NO'' stammen von der Evaluation über einer Datenbank: (also: der Wahrheitswert dieser Aussagesätze wird automatisch getestet; im Anwendungsfall würden also wohl nur die *negativ* evaluierenden Sätze neu assertiert, da die übrigen ja keine neue Information beitragen; dieser Teil des Programms ist unten *nicht* abgedruckt)

```
employ(d1,t1).
employ(d2,t1).
<etc.>

prosper(d1).
prosper(d2).
<etc.>

pitt(t1).
```

```
tutor(t1).
tutor(t2).
<etc.>

director(peter).

department(d1).
department(d2).
<etc.>


2: some departments employ tutors

exists(D: @(department,D) & exists(T: @(tutor,T) &
    @(@(employ,T),D)))

YES

3: the department employs the tutor

exists(X:all(D: @(department,D)->D=X) &
         exists(Y:all(T: @(tutor,T)->T=Y)&
         @(@(employ,Y),X)))

NO

4: every department employs a tutor

all(D: @(department,D) -> exists(T: @(tutor,T) &
    @(@(employ,T),D)))

YES

5: every department prospers

all(D: @(department,D) -> @(prosper,D))

YES

6: john prospers

@(prosper,john)

YES
```

```
7: john employs a tutor

‘‘a’’ + ‘‘tutor’’:
reduce(@(PˆQˆexists(T:@(P,T)&@(Q,T)),tutor),A)
==>
A = Qˆexists(T:@(tutor,T)&@(Q,T))

‘‘employs’’:
RˆWˆ@(R,Tˆ@(@(employ,T),W))

‘‘employs’’ + ‘‘a tutor’’:
reduce(@(RˆWˆ@(R,Tˆ@(@(employ,T),W)),
         Qˆexists(T:@(tutor,T)&@(Q,T))),B).
==>
B = Wˆexists(T:@(tutor,T)&@(@(employ,T),W))

‘‘john’’:
Sˆ@(S,john)

‘‘john’’ + ‘‘employs a tutor’’:

reduce(@(Sˆ@(S,john),
         Wˆexists(T:@(tutor,T)&@(@(employ,T),W))) ,C).
==>
C = exists(T:@(tutor,T)&@(@(employ,T),john))

YES

8: every department employs every tutor

all(D:@(department,D)->all(T:@(tutor,T)->
    @(@(employ,T),D)))

NO

9: peter is john

peter=john

YES

10: john is a tutor

exists(T:@(tutor,T)&john=T)
```

```
YES

11: peter is the director

exists(X:all(D,@(director,D)->D=X)&peter=X)

YES

12: john is the tutor

exists(X:all(T,@(tutor,T)->T=X)&john=X)

NO
```

Diese Implementation der beschriebenen Montague-Semantik ist *sehr* direkt (sogar ohne Umweg über Syntaxstruktur).

Die Grammatik [13] sieht so aus:

```
decl(Decl)      -->     np(Num,Np),
                        vp(Num,Vp),
                        {reduce(@(Np,Vp),Decl)}.

np(sing,Pn)     -->     proper_name(Pn).
np(Num,Np)      -->     det(Num,Det),
                        n(Num,N),
                        {reduce(@(Det,N),Np)}.

vp(Num,Vp)      -->     verb(trans,Num,Tv),
                        np(Num1,Np),
                        {reduce(@(Tv,Np),Vp)}.
vp(Num,V)       -->     verb(intrans,Num,V).

/* ********* Uebersetzung der Basis-Terme ******* */

det(sing,P^Q^all(Z: @(P,Z) -> @(Q,Z)))
                        -->     [every].
det(plur,P^Q^exists(Z: @(P,Z) & @(Q,Z)))
                        -->     []; [some].
```

13. eine stark modifizierte Fassung von Warren 1983, zu finden auch ⇒hier

```
det(sing,P^Q^exists(Z: @(P,Z) & @(Q,Z)))
                          -->     [a]; [an]; [some].
det(Nbr,P^Q^exists(Y:all(X: @(P,X) -> X=Y) & @(Q,Y)))
                          -->     [the]. % Approximation!
verb(trans,Num,P^Q^(@(P,Y^(@(@(Root,Y),Q)))))
                 -->      [V], {is_verb_form(V,Root,Num),
                          verb_type(Root,main+trans)}.
verb(intrans,Num,Root)
                 -->      [V], {is_verb_form(V,Root,Num),
                          verb_type(Root,main+intrans)}.
verb(_,Num,P^X^(@(P,Y^(X=Y))))
                 -->      [V], {is_verb_form(V,Root,Num),
                 verb_type(Root,aux+be)}.
proper_name(P^(@(P,Pn)))
                 -->      [Pn], {is_proper_name(Pn)}.
n(Num,Root)      -->      [N], {is_noun_form(N,Root,Num)}.

/* ***************** Lexikon/Morphologie ************ */

is_noun_form(tutors,tutor,plur).
is_noun_form(tutor,tutor,sing).
<etc.>

is_verb_form(employs,employ,sing).
is_verb_form(employ,employ,plur).
<etc.>

verb_type(employ,main+trans).
verb_type(be,aux+be).
<etc.>

is_adj(rich).

is_proper_name(john).
is_proper_name(peter).
```

Beta-Reduktion in Prolog für...

Der *Kern* der Beta-Reduktion kann mit einer einzigen Prolog-Klausel implementiert werden:

```
reduce(@(V^P,V),P).
```

...einfache und...

entsprechend:

```
λ V.P(V) ==> P
```

Anhand eines Beispiels:

```
λ V.  P    (   V  ) ==>     P
    --^---- ---^--      ----^------
λ V.rich(V)(peter') ==> rich(peter')
```

...für komplizierte Fälle...

Für kompliziertere Fälle (mit eingebetteten Strukturen) brauchen wir zusätzlich noch die Regeln

```
reduce(@(V^P,V),Q)              :-  nonvar(P),
                                    reduce(P,Q), !.

reduce(P^Q,P^R)                 :-  nonvar(Q),
                                    reduce(Q,R).

reduce(all(X:Y -> Z),all(X:A -> B))
                                :-  reduce(Y,A),
                                    reduce(Z,B).

reduce(exists(X:Y & Z),exists(X:A & B))
                                :-  reduce(Y,A),
                                    reduce(Z,B).

reduce(P&Q,R&S)                 :-  reduce(P,R),
                                    reduce(Q,S).

reduce(@(Y,Z),@(A,B))           :-  nonvar(Y),
                                    nonvar(Z),
                                    reduce(Y,A),
                                    reduce(Z,B).
```

sowie die ‘‘catch-all’’-Regel, welche für alle unreduzierbaren (oder unreduzierbar gewordenen) Fälle dient:

```
reduce(O,O).
```

4.3.3 Implementation mit Unifikation

Beobachtung:

- Die Prinzipien von Kompositionalität und funktionaler Applikation *zusammen* sind sehr kompliziert.
- *aber:* Die Prinzipien sind *nicht* siamesische Zwillinge.
- ***daher:*** Die funktionale Applikation wird ersetzt.

Siehe dazu auch Moore 1995:174.

4.3.3.1 Die grundsätzliche Idee

Funktionale Applikation ist...

Daher: Wir verwenden Unifikation statt funktionale Applikation als Instrument zur Kombination von Bedeutungen.

Gründe:

...sequentiell und uni-direktional

1. funktionale Applikation ist streng sequentiell und uni-direktional (d.h. sie erfordert, dass eine Komponente voll instantiiert ist, wenn sie mit einer anderen Komponente kombiniert wird): Die Errechnung der Satzbedeutung gemäss

   ```
   SENT' = NP'(VP')
   ```

   setzt voraus, dass die Bedeutung `VP'` voll errechnet ist, wenn (ein ebenfalls voll errechnetes) `NP'` darauf funktional angewendet wird.

Unifikation erlaubt...

2. Unifikation ist (potentiell) parallel und multi-direktional (d.h. sie kann die ‘‘logische Variable’’ verwenden, um unterspezifizierte Strukturen zu verwenden, die danach ‘‘rückwirkend’’ aufgefüllt werden):

   ```
   @(NP',VP',SENT')
   ```

   ...unter-intantiierte Ausdrücke.

   lässt (prinzipiell) durchaus zu, dass `NP'` und/oder `VP'` nicht oder nur teilweise instantiiert sind (und dafür z.B. SENT’ voll instantiiert)

Vorteile der unifikationsbasierten Komposition:

einfacher, aber...

1. Übersetzungen der Basisterme (z.T. wesentlich) einfacher

...dennoch allgemein.

2. Übersetzungen einzelner Konstituenten dennoch allgemein (d.h. unabhängig vom Verwendungskontext ein für allemal definierbar)
3. im Rahmen der Logik-Programmierung besonders leicht realisierbar

Idee: Man muss nicht mehr die Abfolge von funktionalen Applikationen in der Struktur der Übersetzungen der Basis-Terme voraussehen.

Skizze einer Implementation: (siehe Pereira 1987:98 ff.; dort allerdings zusammen mit Entfaltung (‘‘partielle Ausführung’’) behandelt, was hier nicht getan wird, da sehr verwirrlich)

transitive Verben *viel* einfacher

Ausgangspunkt: transitive Verben.

- als LF von ***‘‘employs’’*** bisher

  ```
  P^Z^@(P,Y^@(@(employs,Y),Z))
  ```

  also

  ```
  λ P. λ Z. P(λ Y. employs(Z,Y))
  ```

  ist sehr kompliziert; intuitiver wäre

  ```
         Y^  Z^ @(@(employs,Y),Z),
  d.h. λ Y.  λ Z.   employs(Z,Y)
  ```

- umgekehrt für intransitive Verben:

  ```
  prosper
  ```

  (*ohne* Lambda) als LF für ***‘‘prospers’’*** ist ‘‘zu wenig kompliziert’’; intuitiver wäre

  ```
  X^@(prosper,X)
  ```

- Eigennamen:

  ```
  P^@(P,peter)
  ```

  für ***‘‘Peter’’*** ist ‘‘zu kompliziert’’; intuitiver wäre

  ```
  peter
  ```

- Substantive (‘‘common nouns’’):

  `haus`

  (*ohne* Lambda) für ***‘‘Haus’’*** ist ‘‘zu einfach’’; [14] intuitiver wäre

  `X^@(haus,X)`

Nunmehr:

Phrasen wie z.B. ‘‘some department’’

Determinatoren: unverändert

- müssen (weiterhin) die folgende LF erhalten

  `Q^exists(D: @(department,D)& @(Q,D))`

- Diese LF enthält als einzige nicht durch einen Quantor gebundene Variable *zuäusserst* die Variable *Q*.
- Daher kann man sie *direkt* kombinieren mit der LF eines Ausdrucks, der diese Variable instantiieren soll.

  Das kann (in der obigen extrem einfachen Grammatik!) nur ein Verb sein, zu dem die Nominalphrase

  i. das *Subjekt* ist (‘‘some department prospers’’)
  ii. ein *Objekt* ist (‘‘(some department) employs a tutor’’)

Für den Fall i) mit

`X^(@(prosper,X))`

ändert sich daher nichts an der Regel

14. Hier und früher wurden sowohl Substantive wie intransitive Verben als Basisausdrücke übersetzt, wie dies der ‘‘klassischen Lehre’’ entspricht (cf. z.B. Dowty 1981:196). Dass man in beiden Fällen auch im Rahmen einer Montague-Semantik mit funktionaler Anwendung Lambda-Abstrakte als LF solcher Wörter verwenden kann (also `λ H.haus(H)` resp. `λ H.schlafen(H)`), sieht man z.B. in Lohnstein 1996:158 und eine enstprechende Ableitung S. 179). Die *transitiven* Verben müssen allerdings auch von ihm so kompliziert analysiert werden, wie in der ‘‘klassischen Lehre’’; siehe Lohnstein 1996:185 ff.

```
decl(S)         -->     np(Num,NP),
                        vp(Num,VP),
                        {reduce(@(NP,VP),S)}.
```

Die LF des Prädikats kann direkt in diese Variable (unten **Q**) ‘‘eingefüllt’’ werden:

```
Call: np(Num, NP,                                   ,...)
Exit: np(sing,Qˆexists(D: @(department,D)& @(Q,D)),...)

Call: vp(sing,VP,                       [prospers],[])
Exit: vp(sing,Yˆ @(prosper,Y),          [prospers],[])
```

Sodann:

```
reduce(@(Yˆ@(prosper,Y), X), R)

Call: reduce(@( Q               ^exists(X: @(department,X)&
Exit: reduce(@((Xˆ@(prosper,X))ˆexists(X: @(department,X)&

            @(Q             ,X)),Yˆ@(prosper,Y)),
            @(Xˆ@(prosper,X),X)),Xˆ@(prosper,X)),

                    S)
                    exists(X: @(department,X)& @(prosper,X)))
```

und da die Variable *Y* in der LF der Verbalphrase `Yˆ@(prosper,Y)` in der äussersten Position ist und (pseudo-)appliziert wird auf die von der Nominalphrase gelieferte Variable *X* (und dann beta-reduziert wird, und zwar im Innern von `exists(...)`, mit `X=Y`)

```
reduce(@(Yˆ@(prosper,Y),X), R)
```

wird effektiv über das Subjekt des Satzes prädiziert.

Anders ist Fall ii): Phrasen wie ‘‘employs a tutor’’

- müssen zwar (weiterhin) die folgende LF erhalten

  ```
  24) Zˆexists(T: @(tutor,T) & @(@(employs,T),Z))
  ```

  (die gleiche äussere Form wie die eines intransitiven Verbs: ‘‘schläft’’)

- Die LF des transitiven Verbs ist aber

  ```
  X^Y^(@(@(employ,Y),X))
  ```

  und hier muss die vom Objekt (‘‘a tutor’’) stammende Variable mit der an *zweit*äusserster Stelle stehenden Variable *Y* des Verbs unifiziert werden.

Grammatik z.T. komplizierter

Daher muss die entsprechende Regel etwas komplizierter werden:

```
vp(Num,Z^VP)    --> verb(trans,Num,TV),
                    np(Num1,NP),
                    { reduce(@(TV,Z),IV),
                      reduce(@(NP,IV),VP)}.
```

(mit ‘‘IV’’ für die LF, die der eines **in**transitiven Verbs entspricht: ‘‘employs_a_tutor’’).

Das ist der erste Ort, wo die Grammatik (etwas) komplizierter wird, als in der vorhergehenden Version. Hier wird Struktur-Matching verwendet, um Komplexität von den Übersetzungen gewisser Basisterme in die Regeln zu verschieben.

Vergleiche den Trace von ‘‘employs a tutor’’:

Durch das *erste* ‘‘reduce’’ wird die äussere (Objekt-)Variable in der LF des transitiven Verbs ‘‘employ’’ (hier: *Z* genannt) ‘‘abgeschält’’, sodass der resultierende Ausdruck (oben `IV` genannt) jene Variable zuäusserst trägt, die man zur Kombination mit dem direkten Objekt braucht (also `X`):

```
1.    reduce(@(TV,Z),IV)

Call: reduce(@(Y^X^ @(@(employ,X),Y),Z),IV                )
Exit: reduce(@(Z^X^ @(@(employ,X),Z),Z),X^ @(@(employ,X),Z))
              ↑                  ↑
              X=Y
```

Da die ‘‘abgeschälte’’ Variable Z aber für die Bindung an die Variable des Satz*subjekts* erforderlich sein wird (‘‘*who* employs?’’), wird sie im Kopf der DCG-Regel wieder ‘‘angeklebt’’:

```
vp(Num,Z^VP)       --> verb(trans,Num,TV),
                                  ...
                                  ...
```

Dort kann sie mit mit der externen, also von aussen (vom Verb) ‘‘gelieferten’’

Variablen *Y* unifiziert werden (↑), und zwar, *bevor* für *Y* eine Bindung (durch Quantor, Lambda oder Konstantenzuweisung) bekannt ist (siehe gleich unten).

Im *zweiten* ‘‘reduce’’ wir aber zuvor die ‘‘eigentliche’’ Arbeit zum Aufbau der Transitiv-Konstruktion geleistet

```
2.    reduce(@(NP,V),VP)

Call: reduce(@( Q                  ^exists(T: @(tutor,T)&
Exit: reduce(@((X^@(@(employ,X),Z))^exists(X: @(tutor,X)&

          @(Q                 ,T)),X^ @(@(employ,X),Z)),
          @(X^ @(@(employ,X),Z),X)),X^ @(@(employ,X),Z)),

             VP                                          )
             exists(X: @(tutor,X)& @(@(employ,X),Z))       )
```

d.h.

```
VP = exists(X: @(tutor,X)& @(@(employ,X),Z))
```

und daher natürlich: (Z wieder ‘‘angeklebt’’)

```
Z^VP = Z^exists(X: @(tutor,X)& @(@(employ,X),Z))
```

Für den ganzen Satz ‘‘Some department employs a tutor’’ wird dann die Variable *Z* durch die (unveränderte) Satz-Regel (hier wiederholt)

```
decl(DECL)      -->     np(Num,Np),
                        vp(Num,Vp),
                        {reduce(@(Np,Vp),DECL)}.
```

unifiziert mit der ‘‘externen’’ Variablen im vom Satzsubjekt stammenden Term (unten: *D*):

```
Call: reduce(@( P^
Exit: reduce(@((D^exists(X: @(tutor,X)& @(@(employ,X),D)))^

        exists(D: @(department,D)&
        exists(D: @(department,D)&

          @(P,                       D)),
          @(D^exists(X: ... & @(...)),D)),
```

```
                Zˆexists(X: @(tutor,X)& @(@(employ,X),Z))),
                Dˆexists(X: @(tutor,X)& @(@(employ,X),D))),
                ↑                                    ↑
                  S                                          )
                  exists(D: @(department,D)&exists(X:@(tutor,X)&
                                          @(@(employ,X),D))))
```

weitere Modifikation der Grammatik

Der *zweite* Ort, wo die Grammatik modifiziert werden muss, ist die Regel für *Eigennamen:*

```
np(sing,Pˆ NP)    -->     proper_name(PN),
                          {reduce(@(P,PN),NP) }.
```

Die war vorher einfach

```
np(sing,Pn)       -->     proper_name(Pn).
```

Diese modifizierte Regel konstruiert ‘‘on the fly’’ jene LF für Eigennamen, die wir schon kennen:

```
Call: np(Num,NP0,[john,prospers]
Call: proper_name(PN,[john,prospers],_981)
Exit: proper_name(john,[john,prospers]
Call: reduce(@(P,john),NP)
Exit: reduce(@(P,john),@(P,john))
Exit: np(sing,Pˆ @(P,john),[john,prospers],[prospers])
```

Die Grammatik für quantifizierte Nominalphrasen bleibt unverändert (da wir keine Notwendigkeit sahen, die Übersetzung von Determinatoren intuitiver zu machen):

```
np(Num,NP)        -->     det(Num,DET),
                          n(Num,N),
                          {reduce(@(DET,N),NP)}.
```

Der *Kern* des Programms ist also:

```
decl(DECL)        -->     np(Num,NP),
                          vp(Num,VP),
                          {reduce(@(NP,VP),DECL)}.

np(sing,Pˆ NP)    -->     proper_name(PN),
                          {reduce(@(P,PN),NP)}.
```

```
np(Num,NP)       -->     det(Num,DET),
                         n(Num,N),
                         {reduce(@(Det,N),NP)}.

vp(Num,Z^VP)     -->     verb(trans,Num,TV),
                         np(Num1,NP),
                         {reduce(@(TV,Z),V),
                         reduce(@(NP,V),VP)}.

vp(Num,IV)       -->     verb(intrans,Num,IV).
```

Das *ganze* Programm [15] findet sich ⇒hier.

Nachteile:

1. nicht alle Teilstrukturen sind interpretierbar
2. uneingeschränkte Unifikation ist zu permissiv (siehe gleich unten)
3. das Verfahren ist *(in der vorgestellten Form)* notationell unsauber (da man die Prolog-Variablen erneut für Objekt- *und* Meta-Variablen verwendet; das liesse sich bei Bedarf aber sauberer machen, indem man Objekt-Variablen während der Erzeugung der Logische Formen als Atome darstellt (z.B. als `var-1`, die ganz am Schluss in entsprechende Prolog-Variablen umgewandelt werden)
4. die Abarbeitung ist nicht maximal effizient

15. Beachte: Im vollständigen Programm haben wir die Übersetzungen der Determinatoren so dargestellt

```
det(sing,R^S^all(X: P -> Q))
                         -->     [every],
                                 { reduce(@(R,X),P),
                                   reduce(@(S,X),Q) }.
```

und nicht mehr so (wie oben)

```
det(sing,P^Q^all(X: @(P,X) -> @(Q,X)))
                         -->     [every].
```

Die zweitgenannte (und ersteingeführte) Notation ist einfacher, aber konzeptuell etwas weniger sauber (eine gleich unten motivierte Vereinfachung, die sog. Entfaltung, ist hier nämlich hinterrücks schon durchgeführt worden).

4.3.3.2 Effizienzgewinn durch Entfaltung

Die Resultate der Anwendungen von ‘‘reduce’’ lassen sich durch *Entfaltung* ‘‘verallgemeinert vor-berechnen’’.

4.3.3.2.1 Entfaltung von Programmen in der Logikprogrammierung

Eine allgemein nützliche Technik

Was ist ‘‘Entfaltung? (manchmal auch ‘‘partielle Evaluation’’ genannt - etwas unklare Terminologie; siehe dazu mehr ⇒hier).

Beispiel: DCG-Notation einer Grammatik und daraus abgeleiteter Parser:

Grammatik:

```
sent            -->     np,
                        vp.

np              -->     det,
                        adj,
                        n.

vp              -->     iv.

det             -->     [the].
n               -->     [dog].
adj             -->     [brown].
iv              -->     [barks].
```

Daraus abgeleiteter Parser:

Das ist bekannt.

Programm:

```
sent(S0,S)      :-      np(S0,S1),
                        vp(S1,S).

np(S0,S)        :-      det(S0,S1),
                        adj(S1,S2),
                        n(S2,S).

vp(S0,S)        :-      iv(S0,S).
```

```
det(S0,S)       :-      c(S0,the,S).
n(S0,S)         :-      c(S0,dog,S).
adj(S0,S)       :-      c(S0,brown,S).
iv(S0,S)        :-      c(S0,barks,S).
```

Die Definition von ‘c’...

```
c([Word|Rest],Word,Rest).
```

Partiell evaluiertes Programm:

```
sent(S0,S)      :-      np(S0,S1),
 <etc. etc.>
vp(S0,S)        :-      iv(S0,S).
```

...wird ‘‘entfaltet’’... d.h. wegcompiliert.

```
det([the|S],S).
n([a|S],S).
adj([brown|S],S).
iv([barks|S],S).
```

ohne `c([Word|Rest],Word,Rest).`

Resultat der ‘‘Offline-Verwendung’’ des Prädikats ‘‘c’’. D.h.: ‘‘Partiell evaluiert relativ zum Prädikat `c`’’!

4.3.3.2.2 Anwendung auf die Beispiel-Grammatik

Ein komplizierteres Beispiel:

Angewandt auf das Prädikat `reduce` im obigen Beispiel.

Beispiel

```
decl(DECL)      -->     np(Num,NP),
                        vp(Num,VP),
                        {reduce(@(NP,VP),DECL)}.
```

`reduce` ist ja (im einfachsten Fall)

Die Definition von ‘‘reduce’’...

```
reduce(@(V^P,V),P).
```

`reduce` mit lauter Variablen aufrufen:

```
Call: reduce(@(NP     ,VP),DECL)
Exit: reduce(@(VP^DECL,VP),DECL)

also:

NP      =   VP^DECL
VP      =   VP
DECL    =   DECL
```

Daher kann man obige Regel *ersetzen* durch

...wird weg-compiliert

```
decl(DECL)      -->     np(Num,VP^DECL),
                        vp(Num,VP).
```

(ohne ‘‘reduce’’!)

Kann man sich so merken:

```
reduce(@(NP,    VP),      DECL)
         NP => VP    ^    DECL
```

(mit => für ‘‘wird ersetzt durch’’)

Dasselbe für alle andern Vorkommen von `reduce:` In

```
vp(Num,X^VP)    --> verb(trans,Num,TV),
                    np(Num1,NP),
                    {reduce(@(TV,X),IV),
                     reduce(@(NP,IV),VP)}.
```

gelten

```
reduce(@(TV,X),IV):    TV => X^IV
reduce(@(NP,IV),VP):   NP => IV^VP
```

also:

```
vp(Num,X^VP)  -->    verb(trans,Num,X^IV),
                     np(Num1,IV^VP).
```

Bei systematischer (vorzugsweise automatischer) Anwendung ergeben sich sehr kompakte unifikationsbasierte Montague-Grammatiken, z.B. aus der obigen Grammatik:

```
decl(DECL)              --> np(Num,VP^DECL),
                            vp(Num,VP).

np(sing,(PN^NP)^NP)     --> proper_name(PN).

np(Num,NP)              --> det(Num,N^NP),
                            n(Num,N).

vp(Num,X^VP)            --> verb(trans,Num,X^IV),
                            np(Num1,IV^VP).

vp(Num,VP)              --> verb(intrans,Num,VP).


det(sing,(X^P)^(X^Q)^all(X: P -> Q)) --> [every].
<etc.>
```

In dieser Form der Grammatik wird besonders klar: Verwendung ‘‘logischer Variablen’’ für unterspezifizierte logische Formen (cf. Pereira 1987:103 ff.; ‘‘?’’ statt

Es bleibt eine ungebundene Variable im Resultat!

ungebundener Variablen für Satzrest)

```
2 Call: decl(DECL,[some,department,prospers],[])

 3 Call: np(Num,VPˆDECL,[some,department,prospers],?)

  4 Call: det(Num,NˆVPˆDECL,[some,department,prospers],?)
  4 Exit: det(sing,(XˆP)ˆ(XˆQ)ˆexists(X: P & Q),...)

  4 Call: n(sing,XˆP,[department,prospers],?)
  4 Exit: n(sing,Xˆ @(department,X),[department,prospers],[prospers])

 3 Exit: np(sing,(XˆQ)ˆexists(X: @(department,X)
           & Q),[some,department,prospers],[prospers])
             ↑

 3 Call: vp(sing,XˆQ,[prospers],[])

  4 Call: verb(trans,sing,XˆIV,[prospers],?)
  4 Exit: verb(intrans,sing,Xˆ @(prosper,X),[prospers],[]) ?

 3 Exit: vp(sing,Xˆ @(prosper,X),[prospers],[]) ?

2 Exit: decl(exists(X: @(department,X)
          & @(prosper,X)),[some,department,prospers],[])
                ↑
```

Der Term `np(Num,VPˆDECL,[some,...],?)`(2. Zeile von oben) wird zu Ende abgearbeitet, *obwohl* die Variable *Q* (beim ersten ↑) noch ungebunden ist.

Man erspart sich damit also das ‘‘Umkehren’’ der Abarbeitungsreihenfolge von Nominalphrase und Verbalphrase via Lambda-Abstraktion über einer Prädikatsvariablen (*P*):

```
λ P. λ X. P(λ Y. employs(X,Y)
```

d.h. obwohl die LF der Nominalphrase in die LF der erst *nachher* erscheinenden Verbalphrase gesteckt werden muss, können wir die Nominalphrase (in der Reihenfolge des Erscheinens) *zuerst* konvertieren, weil wir das ‘‘Loch’’ für die LF der erst nachher errechneten LF der Verbalphrase offen lassen können. Dies ist einer der *zentralen Vorteile* beim Verwenden der Unifikation statt der funktionalen Anwendung.

4.3.3.2.3 Direkte Programme vs. entfaltete Montague-basierte Programme

Nun zeigt sich noch etwas sehr Interessantes und Unerwartetes: Die ‘‘direkten’’ Programme entsprechen weitgehend ‘‘von Hand’’ entfalteten Programmen, wie wir sie oben besprochen hatten (hier nur leicht vereinfacht: ‘‘rel_clause’’ entfernt, da wir das in den Montague-Grammatiken nicht hatten):

‘‘Direktes’’ Programm:

```
sentence(P)          -->     noun_phrase(X,P1,P),
                             verb_phrase(X,P1).

noun_phrase(X,P1,P) -->      determiner(X,P2,P1,P),
                             noun(X,P2).
<etc.>
```

Entfaltetes Montague-basiertes Programm:

```
decl(DECL)               --> np(Num,VP^DECL),
                             vp(Num,VP).

np(Num,NP)               --> det(Num,N^NP),
                             n(Num,N).
<etc.>
```

Die verschiedenen Konstituentennamen wurden beibehalten.

Die Entsprechungen der Variablen sind:

- in `sentence` resp. `decl: P1=VP, P=DECL`
- in `noun_phrase` resp. `np: P2=N, P=NP.` (P1 ist hier redundant; diente im ‘‘direkten Programm’’ zur Behandlung von Relativsätzen, die in dem nachfolgenden Programme nicht berücksichtigt wurden)

Beobachtungen:

1. Die Variable *Num* fehlt in der direkten Version (echter Mangel) ; hat aber nichts mit dem Problem der Kompositionalität zu tun.
2. Terme wie `X^Y` in der entfalteten Version entsprechen zwei (oder mehr - siehe gleich unten) verschiedenen Argumenten in der direkten Version.

3. Die Variable *X* fehlt in der entfalteten Version. Sie wird in den entsprechenden Werten der Variablen *N* und *VP* eingebettet nach oben diffundiert; siehe nochmals den Trace von oben, hier wiederholt:

```
3 Call: np(Num,VPˆDECL,[some,department,prospers],?)

 4 Call: det(Num,   N  ˆ VP  ˆDECL,[some,department,prospers],?)
 4 Exit: det(sing,(Xˆ P)ˆ(Xˆ Q)ˆexists(X: P & Q),...)
```

4.3.3.3 Probleme bei der Simulation der Beta-Reduktion durch Unifikation

Unifikation ist zu permissiv.

Beispielhaft für die Probleme bei der Simulation der Beta-Reduktion durch Unifikation sind koordinierte Konstruktionen, wie z.B. der Satz ‘‘John and Peter smoke’’. ‘‘smoke’’ wird, wie bekannt, als Lambda-Ausdruck `λ X.smoke(X)` dargestellt. Die LF der Nominalphrase ‘‘John and Peter’’ muss `λP.P(john)&P(peter)` sein. Diese LF kann erzeugt werden durch eine neue Grammatikregel

```
np(plur, Pˆ(Conj1&Conj2) )
                        -->     np(_, Pˆ Conj1),
                                [and],
                                np(_, Pˆ Conj2).
```

Nun gilt [16]

16. wobei FA ‘‘funktionale Anwendung’’ heisst

```
John and Peter            smoke

λ P.P(john)&P(peter)      λ X.smoke(X)
-------------------------------------------  FA
λ X.smoke(X)(john) & λ X.smoke(X)(peter)
-------------------------------------------  FA
smoke(john) & smoke(peter)
```

Lambdas *binden* Variablen.

wo die koordinierte Nominalphrase das Prädikat `smoke` auf zwei *verschiedene* Konstanten anwendet. Dies ist möglich, weil die beiden Vorkommen der Variablen *X* jeweils durch das Lambda *gebundene* Vorkommen sind, die etwas Veschiedendes bedeuten und die auch getrennt verarbeitet werden (obwohl sie, rein typographisch, gleich repräsentiert werden).

Das geriet unters Eis.

Bei der Simulation mit Unifikation entsteht hingegen keine Variablenbindung durch das ^ (was ein ganz normaler Operator ist). Daher breitet sich jede Bindung der Variable *X* in der *ganzen* Klausel aus. Daher ergeben die soeben eingeführte Grammatikregel und (bisherige Regel)

```
decl(S) -->     np(Num,NP),
                vp(Num,VP),
                {reduce(@(NP,VP),S)}.
```

vorerst

```
Call: reduce(@(P^(@(P,john)& @(P,peter)),X^ @(smoke,X)),S)

  Call: reduce(@(X^    @(smoke,X),   john),R1)
  Exit: reduce(@(john^ @(smoke,john),john),@(smoke,john))
```

aber dann

```
  Call: reduce(@(john^ @(smoke,john),peter),R2)
```

was nicht funktioniert.

Grund: Die Vorkommen von P in `P^(@(P,john)&@(P,peter))` erhalten bei der Reduktion *identische* Variablen *X*, aber *X* kann nicht sowohl mit ‘‘john’’ als auch mit ‘‘peter’’ instantiiert werden. Die Ausbreitung der Variablenbindungen in Prolog ist hier zu permissiv (sie wird durch die ‘‘Pseudo-Lambdas’’ ^ nicht abgeschirmt). Das ist der Grund für diesen Fehlschlag. (Ein ähnliches Problem ergibt sich bei Ellipsen.)

Lösung: Man kann die (in der ursprünglichen Version *effektiv* lambda-gebundenen) Variablen in der Unifikationn-Version *umbenennen*. (d.h. pseudo-lambda-gebundene Vorkommen in den Prolog-Termen durch neue Variablen ersetzen; das kann z.B. durch die Definition von `rename/2` durch `copy_term/2` sehr einfach geschehen; es werden dadurch *alle* Variablen umgetauft).

```
reduce(P&Q,R&S)         :-      rename(P,P1),
                                rename(Q,Q1),
                                reduce(P1,R),
                                reduce(Q1,S),
                                !.
```

Jetzt:

```
4  Call: reduce(@(U^ @(prosper,U),john)&
                   @(U^ @(prosper,U),peter),_175)

 5  Call: rename(@(U^ @(prosper,U),john),_5013)
 5  Exit: rename(@(U^ @(prosper,U),john),
                @(V^ @(prosper,V),john))

 5  Call: rename(@(U^ @(prosper,U),peter),_5007)
 5  Exit: rename(@(U^ @(prosper,U),peter),
                @(W^ @(prosper,W),peter))

 5  Call: reduce(@(V^ @(prosper,V),john),_5018)
 5  Exit: reduce(@(john^ @(prosper,john),john),
                @(prosper,john)) ?

 5  Call: reduce(@(W^ @(prosper,W),peter),_5019)
 5  Exit: reduce(@(peter^ @(prosper,peter),peter),
                @(prosper,peter))

4  Exit: reduce(@(U^@(prosper,U),john)& @(U^@(prosper,U),peter),
               @(prosper,john)& @(prosper,peter))
```

Allerdings: Jetzt hat man die Logik erster Stufe verlassen und sich in den Bereich der Metalogik begeben, in der man über die syntaktische Gestalt (Form der Variablen) von Formeln reden kann.

4.4 Grenzen der traditionellen Verfahren

Auch wenn die Logik erster Stufe als Zielsprache und das Prinzip der Kompositionalität als Abbildungsprinzip zwischen Syntax und Semantik sehr schön und klar sind, stösst man gelegentlich an Grenzen. Das Prinzip der Kompositionalität muss aus Gründen der Praktikabilität etwas gelockert werden, und die Logik erster Stufe ist nicht mächtig genug, um bestimmte Dinge auszudrücken. Daher müssen diese zwei Konzepte etwas modifiziert werden:

1. *Lockerung* des Prinzips der Kompositionalität
2. *Erweiterung* der semantischen Basis

5. Grenzen der Kompositionalität

5.1 Stukturelle Ambiguitäten

5.1.1 Implementation von Montagues Analyse von Skopus-Ambiguitäten

Wie erinnerlich (ECL II), analysiert man Skopus-Ambiguitäten in der Montague-Grammatik durch eine Art ‘‘umgekehrte Topikalisierung’’:

25) Every girl invited some boy

wird analysiert, als wäre zuerst

25a) Some boy$_n$, every girl invited he$_n$

generiert worden, worin dann he_n ersetzt wird durch ‘‘some boy’’. Damit kann man das ‘‘some boy’’ auch als nach links verschobenen Konstituente betrachten, ganz analog zu Fragewörtern, Relativpronomina und topikalisierten Konstituenten.

Ist aber alles sehr mühsam. Die Pseudo-Ambiguitäten werden jeden Parser in die Knie zwingen.

Zu Montagues Analyse von Skopus-Ambiguitäten (0.L.10)

5.1.2 Alternativen für die Behandlung von strukturellen Ambiguitäten

Da dies alles andere als elegant ist, anerkennt man heute in der Regel an, dass strukturelle Ambiguitäten nicht mehr zum kompositionalen Kern der Sprache gehören. Daher Analyse in *mehreren* Schritten:[17] Ausgangspunkt:

17. im folgenden Beispiele aus Covington 1994:212 ff.

28) A dog chased every cat

Zwei Lesarten

```
A dog chased every cat =

(1)  ∃ X: dog(X) ∧ ∀ Y: cat(Y) → chased(X,Y)

     'There is a dog that chased all cats'

(2)  ∀ Y: cat(Y) → ∃ X: dog(X) ∧ chased(X,Y)

    'Each cat was chased by some dog
     (not necessarily the same dog)'
```

Wir betrachten hierzu zwei Techniken: Die (''klassische'') Quantorenanhebung und die (modernere) Verwendung unterspezifizierter Zwischenstrukturen. Die Technik des sog. ''Storage'' (Cooper-Storage, Keller-Storage) hingegen, welche als Vorgängerin der Technik unterspezifizierter Zwischenstrukturen betrachtet werden kann, wollen wir nicht näher untersuchen (siehe dazu ⇒hier und v.a. Blackburn 1999:70-82.

5.1.2.1 Quantorenanhebung

Bei der Methode der Quantorenanhebung (''Quantifier Raising'') erzeugt man (kompositional) eine voll-quantifizierte Repräsentation für eine erste Lesart und leitet dann, in einem zweiten Schritt, durch Transformation dieser Repräsentation alle anderen Lesarten daraus ab:

```
   Quantor(Variable,Restriktor,Prädikation             )

(1)  some(X,       dog(X),    all( Y,cat(Y),chased(X,Y)))
(2)  all( Y,       cat(Y),    some(X,dog(X),chased(X,Y)))
```

Erste Lesart wird kompositional generiert.

Zweite Lesart aus der ersten abgeleitet durch Quantorenanhebung'':

```
some(X, dog(X), all(Y, cat(Y), chased(X,Y)))
                ⇓
all(Y, cat(Y), some(X, dog(X), chased(X,Y)))
```

Allgemein:

`Quantorenanhebung aus der Prädikation`

```
Ql(V1, R1, Q2(V2, R2, S2))
                ⇓
Q2(V2, R2, Ql(V1, R1, S2))
```

`Q1` und `Q2` stehen für die Quantoren und `R2` enthält keine andere Variable als `V2`.

Auch möglich: Quantorenanhebung aus dem *Restriktor:*

```
Everybody who loves a dog is happy

 (1)  all(X, some(Y, dog(Y), loves(X,Y)), happy(X))

 (2)  some(Y, dog(Y), all(X, loves(X,Y), happy(X)))
```

Hier ist folgendes Transformationsschema erforderlich:

`Quantorenanhebung aus dem Restriktor`

```
Ql(V1, Q2(V2, R2, S2), S1)
                ⇓
Q2(V2, R2, Ql(V1, S2, S1))
```

Erste Verallgemeinerung des Prinzips der Quantorenanhebung: Covington 1994:213

- Bestimmte die anzuhebende quantifizierte Struktur; nenne sie *Q(V, R, S)*.
- Ersetze *Q(V, R, S)* durch *S*. Nenne die resultierende Formel *F*.
- Das Resultat der Anhebung is dann *Q(V, R, F)*.

Beispiel:

```
1. all(X, some(Y, dog(Y), loves(X,Y)), happy(X))
             Q(V, R,      S        )
           ------------------------
2.                  ⇓
   all(X,        loves(X,Y),          happy(X) )  = F
   ---------------------------------------------
                    ⇓
3.    Q(V, R,        F                            )

   some(Y, dog(Y), all(X, loves(X,Y),  happy(X)))
                  ----------------------------
```

Rekursiv anwendbar! Ein Satz mit *n* Quantoren kann so im Prinzip *n !* (n Fakultät) Lesarten bekommen.

Beachte: Das geht *so* nur in einfachen Fällen. Allein schon eine Formel wie

`all(X, (`**`person(X),`** `some(Y, dog(Y), loves(X,Y))), happy(X))`

die eine korrektere Wiedergabe von ‘‘Ever*body* who loves a dog is happy’’ wäre, kann nicht mehr transformiert werden.

In der Praxis gibt es weniger Lesarten:

1. wenn verschiedene Lesarten logisch äquivalent sind ; *formale* Erwägung

   ```
   all(D, dog(D), all(C, cat(C), chased(D,C)))
                  ↔
   all(C, cat(C), all(D, dog(D), chased(D,C)))
   ```

2. wenn ungebundene Variablen erzeugt werden

   `A man killed every one of his dogs`

   `some(M, man(M), all(D, (dog(D),` **`of(D,M)`**`), killed(M,D)))`

   `* all(D, (dog(D), of(D,`**`M`**`)), some(M, man(M), killed(M,D)))`

   ‘‘for every dog of him$_1$ there was a man$_1$ who killed it’’

   formale Erwägung

3. wenn (empirische) Präzedenzbedingungen verletzt werden:

```
each > {some|all|every} > any > {one|two|...|many}
```

z.B.

```
Does a river in Europe flow through each country?

* some(R, river(R), all(C, country(C), flows(R,C)))
  all(C, country(C), some(R, river(R), flows(R,C)))}
```

Dagegen:

```
Does a river in Europe flow through every country?

  some(R, river(R), all(C, country(C), flows(R,C)))
  all(C, country(C), some(R, river(R), flows(R,C)))
```

und schliesslich

```
Is there a river that flows through each country?

  some(R, river(R), all(C, country(C), flows(R,C)))
* all(C, country(C), some(R, river(R), flows(R,C)))
```

Beachte, dass gemäss obigen Quantorenanhebungsregeln ein Quantor nie zwischen die aus

1. Kopfnomen
2. Modifikator

abgeleiteten Ausdrücken zu stehen kommt (Hobbs and Shieber (1987))

```
Every representative that was sent by a company
saw every sample

                ⇓
  all(R,(representative(R),
         some(C,company(C),sent(R,C))),
      all(S,sample(S),saw(R,S)))

                ⇓
  all(R, representative(R),all(S,(sample(S),
         some(C,company(C),sent(R,C))),saw(R,S)))

* Every representative saw every sample a company
  that sent him
```

Die QR-Regeln erlauben diese Ableitung nicht, da sie die Konjunktion nicht als ‘‘Trennfuge’’ erkennen; die Englische Grammatik erlaubt es auch nicht.

Beachte: Auch sehr ‘‘ortsfeste’’ Determinatoren (wie Zahlausdrücke) müssen u.U. verschoben werden. (nämlich dann, wenn ein Satz direkt benachbarte Determinatoren diesen Typs enthält: ‘‘Three boys invited two girls’’).

5.1.2.2 ‘‘Quasi-Logische Formen’’

Nachteil der obigen Methode: Ausdrücke sind vollquantifiziert (mit explizitem Skopus), obwohl korrekte Skopusbeziehungen eigentlich gerade *nicht* bekannt.

Daher ist sauberer: Einführung einer *Zwischenebene*, z.B.:

1. Erzeuge eine *skopusneutrale* Zwischenrepräsentation, in der alle quantifizierenden Ausdrücke *in situ* belassen sind (d.h. ohne den Skopus explizit zu machen)
2. Diese Repräsentation enthält *sowohl* Elemente der Syntaxstruktur *wie auch* solche der LF; sie wird daher oft ‘‘Quasi-Logische Form’’ genannt, wobei dieser Begriff ursprünglich von SRI Cambridge für die Repräsentation in der ‘‘Core Language Engine’’ geprägt worden ist, aber auch als allgemeiner Begriff taugt.
3. Erzeuge die endgültige Repräsentation durch systematisches Quantorenanheben aus *dieser* Struktur (unter Beachtung der genannten Restriktionen)

Heute die ± allgemein benutzte Methode.

Beispiel: ‘‘Chat-80’’ (und danach z.B. CLE und Alvey)

Repräsentation von quantifizierten Ausdrücken z.B. so:

```
quant(  Det,
        Type+Variable,
        Range)
```

Ergibt z.B. (vereinfacht)

```
‘‘every country in Europe’’:

quant(det(every),
      feature&place&X-C,
      country(C),in(C,europe))
```

(beachte: das ist *nicht* die dreiteilige Quantorendarstellung von oben!) und für einen kompletten Satz (*vereinfacht!*)

```
Does a river flow through every country in Europe?

pred(   quant(  det(a),
                feature&river-R,
                river(R)),
        flow_through(R,C),
        quant(  det(every),
                feature&place-C,
                country(C),in(C,europe)) )
```

wo die Syntaxstruktur des Satzes teilweise erhalten ist (als `pred(Subjekt,Verb,direktes_Objekt)`), wo aber auch schon viele Elemente der LF vorkommen (z.B. die Übersetzung der lexikalischen Einheiten mit korrekten gemeinsamen Variablen, wie `country(C),in(C,europe)`, und zusätzlich noch Sorteninformation wie `feature&place-C`).

Über dieser Struktur wird dann die Quantorenanhebung ansetzen und die korrekte(n) Lesart(en) erzeugen.

Die unvereinfachte Darstellung (0.L.11)

Andere Vorteile einer skopusneutralen Repräsentation sind:

1. Die vollständige Desambiguierung ist oft *unnötig* .
2. Die vollständige Desambiguierung ist oft *unmöglich innerhalb eines Satzes* .
3. Die vollständige Desambiguierung ist oft *überhaupt unmöglich* .

Man vergleiche die QLFs mit anderen unterspezifizierten Repräsentationen, u.a. im syntaktischen Bereich ⇒hier.

Hintergrundinformation (0.L.12)

5.2 Komposita

Ein weiteres Gebiet, wo Ambiguitäten kaum durch rein kompositional verfahrende Methoden behandelt werden können, sind *Komposita*, die v.a. in technischer Sprache sehr häufig sind. Hier muss man in manchen Fällen mit viel einzelsprachlichem Wissen (und Weltwissen) zu Werke gehen, um überhaupt Aussicht auf Erfolg zu haben.

Formal unterscheidbar sind: (nach: Kowalewski 1997)

1. `N + N → N: Fabriktor`
2. `V + N → N: Reitschule`
3. `A + N → N: Schwarzgeld`
4. `P + N → N: Nachgebühr`
5. `N + A → A: wetterfest`
6. `P + V → V: aufschreiben`
7. `N + V → V: ballspielen`

Besonders verbreitet sind (reine) Nominalkomposita (oben: 1.), auf die wir uns in der Folge konzentrieren wollen.

5.2.1 Nominalkomposita

Die Situation mit Nominalkomposita ist im *Englischen* und *Deutschen* äusserst problematisch, weil in diesen Sprachen keine Information über die zwischen den Bestandteilen bestehenden Beziehungen ausgedrückt wird. Im *Französischen* ist die Lage etwas weniger schwierig, weil man da oft Präpositionalphrasen statt Komposita verwendet:

Computertyrannei
computer tyranny → tyrannie *par* l'ordinateur

Computerspeicher
computer memory → mémoire *de* l'ordinateur

Aber auch im Französischen gibt es echte Nominalkomposita:
installation configuration → configuration *d'*installation

analog:

danger *de* mort
danger *d'*avalanches

Im folgenden konzentrieren wir uns auf das Deutsche.

Funktional sind zu unterscheiden (zumindest) die folgenden Nominalkomposita.

5.2.1.1 Relationale Komposita

Relationale Komposita

1. relationales Wort + 1 Argumentfüller
2. ''Autoverkauf'', ''Überführungskosten''
3. kompositional analysierbar (morphologisches Wissen genügt meist) (zusätzlich zur Angabe über Relationalität)

Die Probleme hier sind insbes.:

- Die alte Frage, welche Argumente (d.h. welche themastischen Rollen) es denn gibt, ist von der Linguistik noch nicht beantwortet.
- Da man offenbar nur einen einzigen Bestandteil eines Kompositums als Argumentfüller verwenden kann, ist es bei *mehrstelligen* relationalen Wörtern

(z.B. ‘‘Übersetzung’’) nicht klar, *welche* der Argumentspositionen vom Argumentfüller gefüllt wird. Beispiel: ‘‘Die Französischübersetzung ist mir völlig missraten.’’

- Umfassende Listen der relationalen Substantive verschiedener Sprachen sind kaum erhältlich. Meist sind relationale Substantive *Derivationen von Verben* (aber es gibt eben auch andere).

  Diese Listen müssen nicht nur die Wörter an sich enthalten, sondern auch deren Stelligkeit und, wenn möglich, Selektionsrestriktionen über diesen Argumentspositionen, um das zweite der genannten Probleme (welches Argument wird gefüllt?) zu mildern.

5.2.1.2 Stereotyp-Komposita

Stereotyp-Komposita:

1. ‘‘implizit relationales’’ Wort + 1 Argumentfüller
2. Beispiel: ‘‘Schuhfabrik’’
3. kompositional analysierbar (sofern entsprechendes lexikalisches Wissen vorhanden ist) (der Art ‘‘Fabriken produzieren Artefakte’’)

Problem:

- Listen von ‘‘implizit relationalen’’ Wörtern sind kaum erhältlich.

5.2.1.3 Komposita mit Grundrelation

Komposita mit Grundrelation:

1. (klar *nicht-relationales*) Substantiv + 1 ‘‘Argument’’-Füller;

   Die durch das Kompositum ausgedrückte Relation stammt aus einer kleinen und endlichen Menge von Relationen, wie

   a. Ort
   b. Ähnlichkeit
   c. Material

2. Beispiele: ‘‘Seidenkleid’’, ‘‘Samtstimme’’

3. kompositional *nicht* mehr *zuverlässig* analysierbar

Problem:

- Das funktioniert offenbar für bestimmte Substantive. Für welche?
- Die ‘‘Argument’’-Füller füllen nicht mehr die ± offensichtlichen *Argumente* von oben, sondern sind eher Umstandsbeschreibungen. Welche gibt es? Sind sie von Sprache zu Sprache verschieden?

Typen von Umstandsbeschreibungen im Englischen (0.L.13)

5.2.1.4 Kontextabhängige Komposita

Kontextabhängige Komposita:

1. beliebige Substantive und völlig heterogene Relationen
2. Beispiel: ‘‘Blumenadler’’
3. kompositional *überhaupt nicht* mehr analysierbar

5.2.1.5 Kombination der Probleme

Leider:

1. Die meisten relationalen Substantive können auch nicht-relational verwendet werden.
2. Es gibt oft *verschiedene* Argumentspositionen, die gefüllt werden können (siehe oben).
3. Dazu gibt es oft *Umstandsbestimmungen*
4. und manchmal noch schwer definierbare *zusätzliche Informationen*.

Das heisst:

- ‘‘Wiesenverkauf’’ kann auch ‘‘Verkauf *auf* einer Wiese’’ bedeuten (‘‘einfaches’’ relationales Kompositum in ‘‘atypischer’’ Weise verwendet)
- ‘‘computer design’’ kann (mindestens) heissen:
    1. Entwurf *des* Computers (thematische Rolle ‘‘affected entity’’)

2. Entwurf *durch* den Computer (thematische Rolle ‘‘agent’’)
3. Entwurf *für den* Computer (thematische Rolle ‘‘beneficiary’’)
4. Entwurf *mit dem* Computer (Umstandsbestimmung ‘‘instrument’’)
5. (?) Entwurf *wie ein* Computer (zusätzliche Information ‘‘Ähnlichkeit’’)

Extremfall:

Fischfrau

kann heissen

```
Frau des Fisches
Frau, die Fisch verkauft
Frau, die im Sternbild des Fisches geboren ist
Frau und Fisch (=Nixe)
Frau, die Fisch isst
Frau, die Fisch produziert
Frau, die von einem Fisch abstammt
Frau, die kühl wie ein Fisch ist
Frau, die den Fisch gebracht hat
Frau, die beim Fisch steht
Frau, die wie ein Fisch aussieht
etc. etc.
```

6. Grenzen der Semantik erster Stufe

6.1 Die Theorie der Generalisierten Quantoren

Problem: Plurale. Zwar werden

29) All seats are taken

30) Some seats are taken

üblicherweise übersetzt als

29a) $\forall$ S: seat(S) $\rightarrow$ taken(S)

30a) $\exists$ S: seat(S) $\wedge$ taken(S)

aber damit hat man im Grunde eben nicht über eine Pluralität von Sitzen gesprochen, sondern gesagt, dass jeder *einzelne* Sitz besetzt ist: Jedes *einzelne* Objekt, welches ‘‘seat(S)’’ erfüllt, erfüllt auch ‘‘taken(S)’’. (analog für ‘‘some’’).

Konkret heisst das, dass der Wahrheitswert von 29 und 30 ermittelt werden kann, ohne dass man die Menge aller Sitze *als Menge* kennt; insbesondere muss man die Kardinalität dieser Menge nicht kennen. Daher: 29a und 30a sind eigentlich Übersetzungen von

31) Every seat is taken

32) Some seat is taken

(mit starker Betonung auf ‘‘some’’!)

Viele andere Determinatoren lassen sich so nicht mehr behandeln. In 33

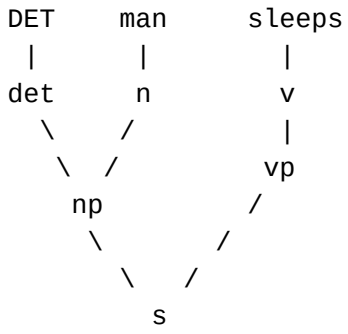
33) Most seats are taken

muss man die Kardinalität der Menge aller Sitze kennen, und die Kardinalität aller belegten Dinge, und dann muss man sie vergleichen. Die sog. nicht-klassische Quantoren lassen sich daher *prinzipiell* nicht in Prädikatenkalkül erster Stufe abbilden; Logik erster Stufe operiert nur über *Individuen*, nie über *Mengen* von Individuen. Interessanterweise liefert die Montague-Grammatik auch hierzu eine in vielen wichtigen Fällen sehr gut funktionierende Antwort. Sie wurde allerdings erst in den letzten ca. 20 Jahren ausgearbeitet[18] und zwar unter dem Namen der Theorie der

generalisierten Quantoren. Montague selbst hat keine Beispiele mit nicht-klassischen Quantoren verwendet. Die Theorie der generalisierten Quantoren ist gleichzeitig ein gutes Beispiel dafür, dass die Einsichten von Montague oft auf den extensionalen Bereich der natürlichen Sprache angewendet werden können. Wir beschränken uns daher hier und im folgenden immer auf eine extensionale Version der Montague-Grammatik.

6.1.1 Nominalphrasen als Quantoren

Die erste Erkenntnis: Es sind in allen Fällen ganze *Nominalphrasen* und nicht Artikel, welche die Funktion von Quantoren in der natürlichen Sprache erfüllen. Das ergibt sich implizit zwar schon aus dem Grundprinzip der strikten Kompositionalität, aber es wurde von Barwise und Cooper zum ersten Mal so klar ausgesprochen. Damit kann man die Parallelität zwischen Syntax und Semantik in diesem Bereich ganz konsequent durchziehen:



‘‘DET’’ kann sein ‘‘every’’, ‘‘some’’, ‘‘most’’, ‘‘more than 11 but fewer than 24’’ etc. In jedem Fall wird von der gesamten Nominalphrase eine Menge von Objekten denotiert: Die Menge, welche jeden Mann, einige Männer, die meisten Männer oder mehr als 11 aber weniger als 24 Männer umfasst.

Da man in der GQT direkt über Mengen spricht, hat man also auch keine Mühe, nicht-klassiche Quantoren wie ‘‘most’’ auszudrücken (s.u.). Das ist aber nicht eine Leistung von Montague oder der GQT, sondern einfach das Ergebnis der höheren Leistungsfähigkeit der Logik zweiter (und höherer) Stufe.

18. , Barwise 1981 war der eigentliche Startschuss dazu (obwohl die Mathematiker generalisierte Quantoren schon früher erforscht hatten). Hier wird fast ausschliesslich auf dieses Papier abgestützt.

Was aber ein *genuiner Beitrag* dieser Formulierung ist, sind eine Reihe davon abgeleiteter interessanter Beobachtungen. So kann man zum Beispiel erwarten, dass Skopusbeziehungen ausschliesslich zwischen ganzen Nominalphrasen stattfinden, und nicht zwichen ihren einzelnen Bestandteilen. Eine besondere Art von Skopusbeziehung entsteht zum Beispiel bei Quantorenambiguitäten durch Quantorenanhebung. Man müsste daher annehmen, dass in allen derartigen Fällen nur ganze Nominalphrasen, nicht aber isolierte Artikel oder Adjektive, ‘‘angehoben’’ werden können. Das scheint auch tatsächlich der Fall zu sein:

34) Every male person in this room speaks some unusual language

erhält die folgenden zwei Interpretationen

```
∀ P: (person_in_this_room(P) ∧ male(P) →
   ∃ L: (language(L) ∧ unusual(L) ∧ speaks(P,L)))

∃ L: (language(L) ∧ unusual(L)  ∧
    ∀ P: (person_in_this_room(P) ∧ male(P) →
    ∧ speaks(P,L)))
```

aber *nicht* zum Beispiel

```
∀ P: (person_in_this_room(P) → ∃ L: (language(L)
     ∧ unusual(L) ∧ speaks(P,L) ∧ male(P)))
```

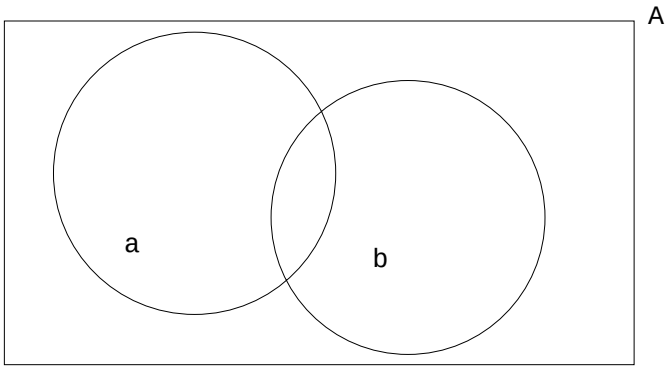
was ja paraphrasiert werden könnte als

34a) Every person in this room speaks some unusual language and is male

und das heisst etwas ganz anderes als 34.

6.1.2 Determinatoren als Relation zwischen Mengen

Für viele Zwecke einfacher die relationale Sichtweise: Der Determinator ist eine Relation zwischen zwei Mengen:

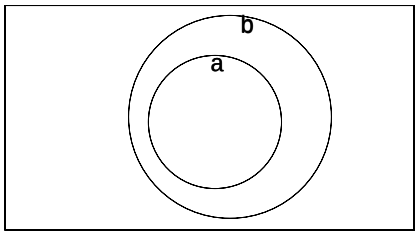


some a b: $a \cap b \neq \emptyset$

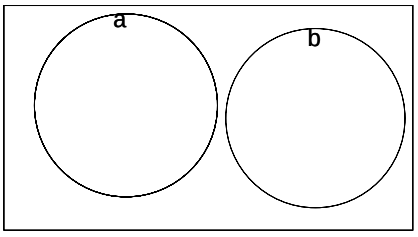
A: Grundmenge

Augenmerk nicht auf der Nominalphrase, sondern dem Determinator.

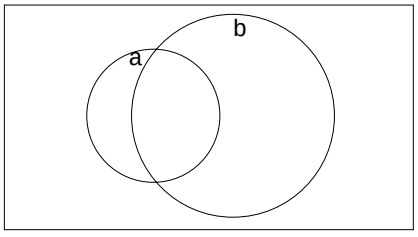
In dieser Sichtweise wird die Interpretation vieler Quantoren mengentheoretisch sehr transparent, und man kann die zwei klassischen Quantoren genau gleich behandeln:



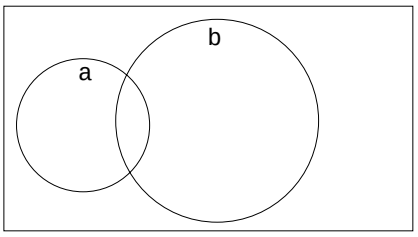
all a b



no a b



most a b



few a b (??)

Determinator	Interpretation

für $N \subseteq A$

Every N	$\{X \subseteq A \mid N \subseteq X\}$
Some N	$\{X \subseteq A \mid N \cap X \neq \emptyset\}$
No N	$\{X \subseteq A \mid N \cap X = \emptyset\}$
Most N	$\{X \subseteq A \mid card(N \cap X) > card(N)/2$
Exactly M N	$\{X \subseteq A \mid card(N \cap X) = M\}$
M N	$\{X \subseteq A \mid card(N \cap X) \geq M\}$
	(?; M: Numeral)

Ergibt dann z.B.

some students $\{X \subseteq A \mid [\![student]\!] \cap X \neq \emptyset\}$

mit $[\![student]\!]$ die Denotation von ‘‘student’’ (d.h. die Menge aller Studenten).

Daraus dann:

some students smoke: $\{[\![student]\!] \cap [\![smokes]\!] \neq \emptyset\}$
every student smokes:$\{[\![student]\!] \subseteq [\![smokes]\!]\}$

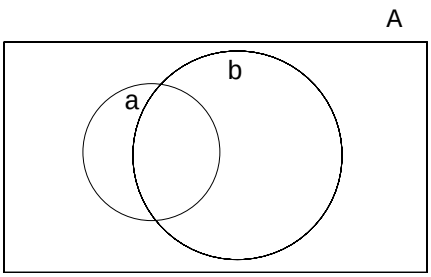
Oder noch einheitlicher:

Every N	$\{X \subseteq A \mid card(N \cap X) = card(N)\}$
Some N	$\{X \subseteq A \mid card(N \cap X) > 0$
No N	$\{X \subseteq A \mid card(N \cap X) = 0$
Most N	$\{X \subseteq A \mid card(N \cap X) > card(N)/2$
Exactly M N	$\{X \subseteq A \mid card(N \cap X) = M\}$
M N	$\{X \subseteq A \mid card(N \cap X) \geq M\}$

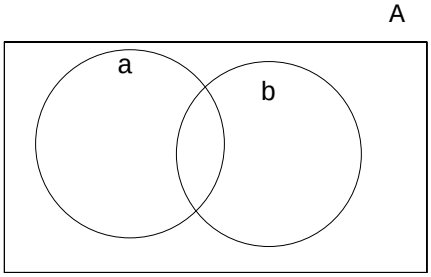
ergibt:

some students smoke:
$card([\![student]\!] \cap [\![smokes]\!]) > 0$

Problem: ‘‘few’’, ‘‘many’’. Kontextuell parametrisiert:



many a b



few a b

35) Viele Studenten sind beim Examen durchgefallen

36) Viele Studenten sind an der Grippe gestorben

Beispiel 35: 30%? 50%? Beispiel 36: 1%? 3%?

6.1.3 Monoton zu- und abnehmende Quantoren

Weiterhin kann man unterscheiden zwischen monotonen und nicht-monotonen Quantoren, und bei den monotonen zwischen monton zunehmenden und monton abnehmenden.

Regeln das Verhalten von Quantoren, wenn deren Argumente verkleinert oder vergrössert werden. Diese Beziehungen sind sher wichtig für *Inferenzbeziehungen*, die in computerlinguistischen Anwendungen von zentraler Bedeutung sind.

Monotone Quantoren sind solche Nominalphrasen, bei denen gilt

37) $NP\ VP_1 \rightarrow NP\ VP_2$ (NP ist monoton zunehmend)

38) $NP\ VP_2 \rightarrow NP\ VP_1$ (NP ist monoton abnehmend)

mit $[\![VP_1]\!] \subset [\![VP_2]\!]$

Beispiele:

Monoton zunehmend:

```
some Republican entered the race early
        → some Republican entered the race
many men are tall and blond
        → many men are are tall
every linguist          ...
John                    ...
most peanut farmers     ...
```

Das scheint einfach:

```
republican(sk1).
enter(e1,sk1,sk2).
race(sk2).
time(e1,past).
early(e1).
_____________________

?- republican(R), enter(_,R,S), race(S).
yes
R=sk1
S=sk2
```

Aber hier ist es eben anders:

Monoton abnehmend:

```
no plumber entered the race
        → no plumber entered the race early
few linguists are tall
        → few linguists are tall and blond
neither Democrat        ...
```

Das hat schon manchen Computerlinguisten aufs Glatteis geführt.

Zudem erweist es sich, dass die Negation die Richtung der Monotonität ändert:

```
negiert: monton ↓ | positiv: monoton ↑
_____________________________________________
no man/men         | some man/men      (no = not some)
few men            | many/several men  (few = not many)
```

Die zwei Unterscheidungen scheinen zusammenzuhängen: Es scheint empirisch zu stimmen, dass in allen natürlichen Sprachen positiv starke (siehe gleich unten) Determinatoren monoton zunehmend sind, und negativ starke Determinatoren monoton abnehmend. Das ist der sog. ‘‘*strong determiner constraint*’’.

Es ist also klar, dass derartige Erkenntnisse über die verschiedenen Typen von Quantoren von unmittelbarem Interesse auch bei der machinellen Verarbeitung von Sprache sind: Je nach Quantor sind von einer gegebenen Aussage aus ganz verschiedene Schlussfolgerungen erlaubt.

Etwas weniger Probleme schaffen die nicht-monotonen Quantoren (aber sie sind auch viel seltener). Beispiele für *nicht-monotone* Quantoren:

- exactly two men
- exactly half the men

6.1.4 Absolute und relative, proportionale und kardinale Determinatoren

Eine weitere Unterscheidung, welche in GQT getroffen wird, ist die zwischen ‘‘starken’’ und ‘‘schwachen’’ Quantoren (resp. Determinatoren). Schon früher wurde beobachtet, dass es Quantoren gibt, welche man in ‘‘there is’’-Sätzen verwenden kann, und andere. So sind folgende Fügungen ganz unproblematisch:

39) There are {some|many|seven|several|no} boys in the garden

und die folgenden ganz schlecht:

40) ?? There are {the|the two|all|every|each|both|most} boys in the garden

Die Quantoren im Fall 39 heissen (meist) ‘‘schwach’’, die anderen ‘‘stark’’. Oft wurde daher gesagt, die starken Quantoren könnten nicht als ‘‘referierende Ausdrücke’’ verwendet werden, wohl aber die schwachen. Die Theorie der generalisierten Quantoren versucht, derartige Intuitionen klarer zu machen. Ein Kriterium,

das sich daraus zum Beispiel ergab, ist (streng nach Barwise 1981):

1. Ein Determinator *D* ist *positiv stark*, wenn für jedes Modell gilt

   $A \in [\![D]\!](A)$

   ‘‘*Every* gnu is a gnu’’ ist eine Tautologie

2. Ein Determinator *D* ist *negativ stark*, wenn für jedes Modell gilt

   $A \notin [\![D]\!](A)$

   ‘‘*Neither* gnu is a gnu’’ ist immer falsch

3. Ein Determinator *D* ist *schwach*, wenn eine Aussage ‘‘*D N* is a *N*’’ *kontingent* ist. ‘‘*Many* gnus are gnus’’ ist nur wahr, wenn es viele Gnus gibt

Diese Darstellung ist aber reichlich merkwürdig; sie wurde seit Barwise&Cooper in der Form auch kaum mehr verwendet. Eine m.E. einleuchtendere Formulierung wäre:

- ‘‘*Every* gnu is a gnu’’ ist eine Tautologie
- ‘‘*Neither* gnu is a gnu’’ kann nie wahr sein, weil seine Aussage der existentiellen Präsupposition widerspricht (also nicht ‘‘ist nie wahr’’, sondern ‘‘ist nie wahrheits*fähig*’’).
- ‘‘*Many* gnus are gnus’’ ist wahrheits*fähig*, sofern es viele Gnus gibt, und sonst nicht (also nicht ‘‘kontingent’’ im Sinne von ‘‘manchmal wahr/manchmal falsch’’, sondern im Sinn von ‘‘manchmal wahrheitsfähig/manchmal nicht wahrheitsfähig’’).

Eine wichtige Unterklasse der starken Determinatoren sind die definiten Determinatoren: Sie setzen (bekanntlich) die Existenz und Eindeutigkeit der ‘‘Basismenge’’ voraus. Man kann einen definiten Quantor *D N* daher immer verwenden in den folgenden Kontexten *all of D*, *most of D*,, *some of D* and *many of D*, zum Beispiel ‘‘all men’’ und ‘‘most of all men’’, aber *nicht* ‘‘most of many men’’. Ob man hier den Intuitionen von Barwise und Cooper folgen will, ist ein andere Frage.

Die Behandlung der in definiten Determinatoren enthaltenen Präsuppositionen ist in der GQT in der Regel weniger unbefriedigend als sonst: Man anerkennt zumindest deren Existenz. Meist definiert man die Interpretationsfunktion als partiell, d.h. ein Satz hat keinen Wahrheitswert, wenn eine Präsupposition nicht erfüllt ist.

Auch wenn die Theorie der generalisierten Quantoren manche Probleme im Zusammenhang mit Quantoren aufgehellt hat, sind keineswegs alle Schwierigkeiten ausgeräumt. Dies betrifft nicht zuletzt die Terminologie. So wird statt der Unterscheidung ‘‘stark/schwach’’ oft auch ‘‘quantifikationell/kardinal’’ verwendet, oder ‘‘proportional/kardinal’’ oder sogar ‘‘definit/indefinit’’. Dieses Terminologie-Chaos verbirgt die Tatsache, dass gewisse Unterscheidungen auch in GQT nicht klar genug getroffen werden. Ein Versuch, die Situation wenigstens in bezug auf die Terminologie ein wenig zu ändern, folgt.

Der Hauptunterschied ist zweifellos der zwischen Quantoren, welche zwei Mengen (resp. deren Kardinalitäten) vergleichen, und den anderen (also der Unterschied, der bei Barwise/Cooper ‘‘stark/schwach’’ heisst). Es wäre aussagekräftiger, hier zwischen ‘‘*relativen*’’ und ‘‘*absoluten*’’ Determinatoren (und daraus gebildeten Quantoren) zu unterscheiden. Die relativen Quantoren können immer durch eine Partitivkonstruktion paraphrasiert werden. Nennen wir die Menge, aus der die zu vergleichende Untermenge herausgegriffen wird, ‘‘Basismenge’’, und die andere die ‘‘Vergleichsmenge’’: Bei ‘‘Most men’’ wäre ‘‘men’’ die Basismenge, ‘‘most (of the men)’’ die Vergleichsmenge.

Absolute Determinatoren können präzis oder vag sein. Präzis absolute Determinatoren sind: ‘‘a’’, *unbetonte* Zahlwörter (‘‘svn’’) und ‘‘no’’. Vag absolute Determinatoren sind *unbetontes* ‘‘some’’ (‘‘/sm/’’), *unbetontes* ‘‘several’’ (‘‘svrl’’) und ‘‘many’’ (‘‘/mny/’’; ein bisschen fraglich), ev. ‘‘a few’’.

Bei den relativen Quantoren gibt es proportionale und kardinale; bei den proportional relativen wird die Vergleichsmenge als Proportion der Basismenge angegeben: ‘‘most’’, ‘‘few’’ usw., aber auch das *betonte* ‘‘some’’. Die Paraphrase ‘‘most of the’’, ‘‘SOME of the’’ etc. ist problemlos. Bei den kardinal relativen Quantoren ist die Kardinalität explizit angegeben. Sie kann *präzis* oder *vag* angegeben werden. Zu den präzis kardinal relativen Determinatoren gehören die *betonten* Zahlwörter (‘‘one’’, ‘‘seven’’), zu den vag kardinal relativen zum Beispiel die betonten Versionen von ‘‘several’’ und ‘‘many’’. Erneut ist die Paraphrase mit dem Partitiv leicht möglich: ‘‘SEVEN of the’’, ‘‘SEVERAL of the’’ usw.

	indefinit	definit
absolut		
präzis	a, /svn/, no	-
vag	/svrl/, /sm/ /mny/, a few (?)	-
relativ		
kardinal		
präzis	one, seven	the one, the seven
vag	several, some many	the several, the many (?? the some)
proportional		
präzis	all, none	the (all the, ?? the none)
vag	most, few	the few (?? the most)

Was hier besonders auffällt ist die Tatsache, dass viele lexikalische Elemente (d.h. Wortformen) sowohl absolut wie relativ verwendet werden, dass aber die Aussprache eine klare Unterscheidung macht. Obwohl natürlich die Aussprache selbst in der Schrift nicht zum Ausdruck kommt, gibt es explizite Topikalisiserungen, welche den selben Zweck erfüllen (‘‘It was only some girls that ...’’). Meist reicht aber die Position im Satz, um dasselbe zu erreichen: Im Englischen (und Deutschen) ist der Satzanfang stark betont, und ein ‘‘some’’ am Satzanfang ist daher meist auch proportional verwendet.

Hier sollte auch erwähnt werden, dass zum Beispiel ‘‘any’’ je nach Betonung eine ganz andere Bedeutung annimmt. Wenn wir in

41) I don’t lend my books to anybody

das ‘‘anybody’’ aussprechen als ‘‘/a$^{\backslash}$nybody/’’ , so heisst das ‘‘niemandem’’, ‘‘nicht einem einzigen’’:

41a) $\neg \exists$ P: person(P) $\wedge$ i_lend_my_books_to(P)

Wenn wir es aber ‘‘/a$^{\vee}$nybody/’’ aussprechen, so heisst das ‘‘nicht einfach jedem’’:

41b) $\neg \forall$ P: person(P) $\rightarrow$ i_lend_my_books_to(P)

Diese, und manche andere Beispiele, zeigen, dass die Betonung (insbes. die Satzbetonung) keineswegs bloss ein stilistisches Mittel ohne semantische Relevanz ist.

Während die Theorie der generalisierten Quantoren v.a. die Unterscheidung zwischen relativen und absoluten Determinatoren (und Nominalphrasen) also durchaus vornimmt, sitzen die absoluten Nominalphrasen irgendwie etwas unbehaglich im ganzen System. Einer der Gründe ist der, dass bei absoluten Nominalphrasen bestimmte Lesarten des Plurals möglich sind, welche von GQT nicht erfasst werden können, während sie (aus nicht ganz klaren Gründen) bei den relativen Nominalphrasen ausgeschlossen sind, weshalb die für GQT auch weniger problematisch sind.

6.1.5 Zur Implementierung der Theorie der Generalisierten Quantoren

6.1.5.1 Generelle Implementationen

Ein erstes Problem bei der Implementierung der Theorie der generalisierten Quantoren betrifft die Grundidee der modelltheoretischen Interpretation von Plural-Nominalphrasen, und als solches ist es nicht auf GQT beschränkt. Um den Wahrheitswert eines so einfachen Satzes wie

42) John smokes

zu testen, muss man die Übersetzung

```
λ P. P(john')(smoke')
```

interpretieren (oder die entsprechende direkte Interpretation vornehmen), d.h. ermitteln, ob die Menge aller Raucher ein Element ist in der Menge aller Mengen, von denen John ein Element ist. Man müsste also (schon bei einer Analyse, die keine intensionalen Aspekte berücksichtigt)

1. die Menge aller Mengen ermitteln, zu denen u.a. John gehört
2. die Menge aller Raucher ermitteln
3. testen, ob letztere ein Element ersterer ist

Dass dies sicher nicht die Methode ist, die wir in unserem Kopf verwenden, ist klar.

Dasselbe gilt auch für Sätze mit allgemeinen Ausdrücken wie ‘‘Most men smoke’’. Auch hier ist nicht anzunehmen, dass wir ermitteln, ob die Menge aller Raucher ein Element der Menge aller Mengen ist, welche eine Mehrheit aller Männer enthalten.

Deshalb wird man wieder auf die relationale Auffassung zurückgreifen

43) Most men smoke

Dies ist nicht nur einfacher zu verstehen, sondern auch recht einfach zu implementieren.

Wenn man möglichst nur Standard-Prolog Konstrukte verwenden will, könnte man die *Frage*

44) Do *Q* men smoke?

als

```
quant(Q,M,man(M),smoke(S))
```

übersetzen, was dann zu

```
?-  setof(M,man(M),Ms),
    setof(S,smoke(S),Ss),
    intersection(Ms,Ss,Is),
    card_compare(Q,Is,Ms).

mit Q = {all,some,no,most, etc.}
```

resp. allgemein

```
quant(Q,V,P,R)  :-  setof(V,P,Ms),
                    setof(V,R,Ss),
                    intersection(Ms,Ss,Is),
                    card_compare(Q,Is,Ms).
```

mit entsprechenden Definitionen (die numerischen Werte sind ± willkürlich)

```
card_compare(most,Is,Ms)   :-   length(Is,CIs),
                                length(Ms,CMs),
                                CIs/CMs > 0.5.
card_compare(no,Is,Ms)     :-   length(Is,CIs),
                                CIs = 0.
card_compare(few,Is,Ms)    :-   length(Is,CIs),
                                length(Ms,CMs),
                                CIs/CMs < 0.1.

card_compare(all,Is,Ms)    :-   length(Is,CIs),
                                length(Ms,CMs),
                                CIs = CMs.
card_compare(some,Is,Ms)   :-   length(Is,CIs),
                                CIs > 0.
<etc.>
```

Ausführliche Aufstellung (0.L.14)

6.1.5.2 Spezialisierte Implementationen

Aber: Diese generellen Implementationen sind extrem ineffizient.

Vergleiche z.B.

```
?-  setof(M,man(M),Ms),
    setof(S,smoke(S),Ss),
    intersection(Ms,Ss,Is),
    card_compare(all,Is,Ms).
```

mit

```
?- \+((man(M), \+ smokes(M)))
```

Und noch extremer bei ‘‘some’’:

```
?- man(M), smokes(M).
```

Daher:

```
quant(all,V,P,Q)    :-  \+((P, \+Q)).
quant(some,V,P,Q)   :-  P, Q.
```

Und *noch* einfacher mit ‘‘no’’:

```
quant(no,V,P,Q)     :-  \+((P, Q)).
```

Eine *einzige* Ausnahme genügt!

Das ist die ‘‘Rechtfertigung’’ der Methoden im Kapitel ‘‘Allaussagen als Theoreme’’.

Aber: Präsuppositionen!

Daher eventuell besser:

```
quant(all,V,P,Q)      :-    P, \+((P, \+Q)).

quant(no,V,P,Q)       :-    P, \+((P, Q)).
```

oder sogar

```
quant(all,V,P,Q)        :-      prsp(P), \+((P, \+Q)).
quant(no,V,P,Q)         :-      prsp(P), \+((P, Q)).
quant(some,V,P,Q)       :-      prsp(P), Q.
```

mit

```
prsp(P) :-      \+(\+(P)), !.
prsp(P) :-      error_message(prsp_violation,P),
                fail.

error_message(prsp_violation,P)
        :-  nl, write('There are no instances of '),
            write(P),
            write(' in the first place.'), nl.
```

Doppelnegation: Um keine dauernden Bindungen zu erzeugen. Beachte (doppelte) Klammerung!

Auch bei echt proportionalen Determinatoren geht’s (etwas) einfacher:

```
‘‘Do most men smoke?’’:

?-  prsp(man(M)),
    setof(M,man(M),Ms),
    setof(X,(member(X,Ms),smoke(X)),Xs),
    card_compare(most,Xs,Ms).
```

Da muss man zumindest nicht *alle* Raucher zusammensuchen.

Allgemein:

```
quant(most,V,P,Q):-     prsp(P),
                        setof(V,P,Vs),
                        setof(X,(member(X,Vs),Q),Xs),
                        card_compare(most,Xs,Vs).
```

Analog ‘‘few’’ etc. Für Präsuppositionen erneute spezielle Behandlung erforderlich.

Sind aber alles bloss *effizienzmässig* relevante Umformungen; semantisch sind sie irrelevant.

Nunmehr: Auch definite Nominalphrasen können analog behandelt werden.

Singular:

45) Is the king of France bald?

```
quant(sg_the,K, king(K,france), bald(K))

quant(sg_the,V,P,Q)  :- prsp(P),
                        setof(V,P,[Vs]),
                        Q.
```

Plural:

46) Are the viscounts of France bald?

```
quant(pl_the,V, viscount(V,france), bald(V))

quant(pl_the,V,P,Q)  :- prsp(P),
                        setof(V,P,Vs),
                        forall(member(X,Vs), Q).
```

mit

```
forall(P,Q) :-  \+((P, \+Q)).
```

d.h. die Interpretation von ‘‘all’’ ohne Präsuppositionstest.

Beachte: Eingebette Quantoren schaffen keine neuen Probleme:

47) Do most smokers like all brands of cigarettes?

ergibt:

```
?- quant(most,S,smoker(S),
         quant(all,B,cigarette_brand(B),likes(S,B))).
```

wird reduziert zu:

```
?-  prsp(smokes(S)),
    setof(S,smokes(S),Ss),
    prsp(cigarette_brand(B)),
    setof(X,(member(X,Ss),
             \+((cigarette_brand(B), \+likes(X,B)))),Xs),
    card_compare(most,Xs,Ss).
```

so wie’s dasteht würde es ein Compiler erzeugen. Meist aber interpretiert.

Aber: Die korrekte Einbettung (Skopus!) muss zuerst erreicht werden Siehe oben.

6.1.5.3 Behandlung von Aussagesätzen

Ein **fundamentales Problem:** Behandlung von *Aussagesätzen* in einem *kommunikativen* Kontext:

43) Most/Few men smoke.

Soll z.B. Datenbankeinträge *generieren*.

Bei universell quantifizierten Aussagen wie
Every control code invokes a terminal command
ist das einigermassen einfach:

```
declare(quant(every,V,P,Q))      :-
        prsp(P), \+((P, \+assert(Q))).
```

Also: Pro erfüllte Bedingung ‘P’ entsprechende Folgerung ‘Q’ assertieren. `declare` macht klar, dass ein Aussagesatz vorliegt. Man müsste dann, konsequenterweise, für Fragesätze ein entsprechendes Prädikat (`question` od.dgl.) verwenden.

Aber: Bei ‘‘most/few/etc.’’ (‘‘most students smoke’’) kennt man die Individuen (‘‘students’’) nicht.

Mögliche Lösung:

```
declare(quant(Qu,V,P,Q)) :- prsp(P),
                            setof(V,P,PQs),
                            card_compare(Qu,Xs,PQs), !,
                            forall(member(X,Xs),
                                   (skolemize(V,V),
                                    assert(P),
                                    declare(Q))).
```

Findet z.B. für `Qu = most` die *Mindestzahl*, welche eine Majorität ist, und

1. erzeugt künstliche Namen (`skolemize`)
2. assertiert entsprechende Einträge für die *Prädikation* (‘‘smoker’’)
3. *und* assertiert Einträge für die *Restriktion* (‘‘student’’)

`assert(P)` ist noch nicht allgemein genug: Könnte ja ein komplexer Ausdruck sein (z.B. für ‘‘every student *of chemistry*’’)

Beispiel: Datenbank sei:

```
student(a).     student(f).
student(b).     student(g).
student(c).     student(h).
student(d).     student(i).
student(e).     student(k).
```

10 Studenten. 6 ist die kleinste Mehrheit:

Nunmehr:

```
‘‘Most students smoke.’’:

2  2  Call: declare(quant(most,S,student(S),smoker(S)))
3  3  Call: prsp(student(S))
3  3  Exit: prsp(student(S))
4  3  Call: setof(S,student(S),_553)
4  3  Exit: setof(S,student(S),[a,b,c,d,e,f,g,h,i|...])
5  3  Call: card_compare(most,Qu,[a,b,c,d,e,f,g,h,i|...])
5  3  Exit: card_compare(most,[A,B,c,D,E,F],[a,b,c,d,e,f,g,h,i|...])
3  3  Call: member(X,[A,B,C,D,E,F])
3  3  Exit: member(X,[X,B,C,D,E,F])
4  3  Call: skolemize(S,S)
4  3  Exit: skolemize(sk-1,sk-1)
8  3  Call: assert(student(sk-1))
8  3  Exit: assert(student(sk-1))
9  3  Call: assert(smoker(sk-1))
9  3  Exit: assert(smoker(sk-1))
3  3  Redo: member(X,[X,B,C,D,E,F])

<etc.>
```

Bemerkungen:

1. `sk-N` sind Skolem-Konstanten
2. `card_compare(most,M,[a,b,c,d,e,f,g,h,i,k])` *erzeugt* (in M) eine Liste der erforderlichen Länge (mit Variablen!)
3. das Prädikat

```
forall(member(X,Xs),
      (skolemize(V,V),
       assert(P),
       declare(Q))).
```

iteriert durch die Einträge

```
forall(Generator, Goal) :-
    \+((Generator, \+Goal)), !.
```

Resultat: Es werden *neu* assertiert (mit obiger Datenbank)

```
smoker(sk-1).   student(sk-1).
smoker(sk-2).   student(sk-2).
smoker(sk-3).   student(sk-3).
smoker(sk-4).   student(sk-4).
smoker(sk-5).   student(sk-5).
smoker(sk-6).   student(sk-6).
```

6 ist die kleinste Mehrheit von 10.

Warum eigentlich Skolem-Konstanten generieren und nicht `smoker(a). smoker(b).` assertieren? Weil wir natürlich nicht wissen, *welche* der Studenten rauchen.

Bei eingebetteten Ausdrücken: (aber wieder der einfachere Fall von ‘‘every’’ and ‘‘some’’)

Every smoker likes some brand (of cigarette).

wird übersetzt in

```
declare(quant(every,S,smoker(S),
              quant(some,B,brand(B),
                    likes(S,B))))
```

und evaluiert wie folgt

```
2  2  Call: declare(quant(every,S,smoker(S),
                                  quant(some,B,brand(B),
                                        likes(S,B))))
4  3  Call: prsp(smoker(S))
4  3  Exit: prsp(smoker(S))

3  3  Call: forall(smoker(S),
```

```
                    declare(quant(some,B,brand(B),
                                          likes(S,B))))
4  4  Call: smoker(S)
4  4  Exit: smoker(a)

6  4  Call: declare(quant(some,B,brand(B),likes(a,B)))

 8  6  Call: prsp(brand(B))
 8  6  Exit: prsp(brand(B))

 9  6  Call: setof(B,brand(B),_2393)
 9  6  Exit: setof(B,brand(B),[1,2,3,4])
```

(offenbar 4 Marken)

```
10  6  Call: card_compare(some,_2382,[1,2,3,4])
10  6  Exit: card_compare(some,[_3464],[1,2,3,4])

 8  6  Call: forall(member(_2372,[_3464]),
                  (skolemize(B,B),
                   assert(brand(B)),
                   declare(likes(a,B))))

  10  8  Call: member(_2372,[_3464])
  10  8  Exit: member(_2372,[_2372])

  11  7  Call: (skolemize(B,B),assert(brand(B)),
                declare(likes(a,B)))

  12  8  Call: skolemize(B,B)
  12  8  Exit: skolemize(sk-1,sk-1)

  16  8  Call: assert(brand(sk-1))
  16  8  Exit: assert(brand(sk-1))

  17  8  Call: declare(likes(a,sk-1))

   18  9  Call: new_assert(likes(a,sk-1))

   19  10  Call: likes(a,sk-1)
   19  10  Fail: likes(a,sk-1)

   19  10  Call: assert(likes(a,sk-1))
   19  10  Exit: assert(likes(a,sk-1))

   18  9  Exit: new_assert(likes(a,sk-1))
```

```
  17  8  Exit: declare(likes(a,sk-1))

  11  7  Exit: call((skolemize(sk-1,sk-1),
                    assert(brand(sk-1)),
                    declare(likes(a,sk-1))))


   9  7  Redo: call(member(_2372,[_2372])) l

<etc.>
```

Resultat:

```
<keine neuen Einträge zu ‘‘smoker’’!>

likes(a, sk-1).   brand(sk-1).
likes(b, sk-2).   brand(sk-2).
likes(c, sk-3).   brand(sk-3).
likes(d, sk-4).   brand(sk-4).
likes(e, sk-5).   brand(sk-5).
likes(f, sk-6).   brand(sk-6).
likes(g, sk-7).   brand(sk-7).
```

Aber:

1. Problem der redundanten Information
2. Problem der ‘‘Pseudogenauigkeit’’
3. Problem der fehlenden Information über Identität

Ad 1: Nach Assimilation von ‘‘Most students smoke’’ umfasst die Datenbank (u.a.): (angenommen, Studenten a bis g sind in der Datenbank als Raucher ausgewiesen)

```
smoker(a).  smoker(sk-1).
smoker(b).  smoker(sk-2).
smoker(c).  smoker(sk-3).
smoker(d).  smoker(sk-4).
smoker(e).  smoker(sk-5).
smoker(f).  smoker(sk-6).
smoker(g).
```

10 Studenten, Mehrheit davon 6: `smoker(sk-1).` bis `smoker(sk-6).`.

- Frage: ‘‘Wie viele Raucher gibt es? ’’
- Antwort: ‘‘13’’

Antwort ist *nicht korrekt:* Das Verfahren generiert ja *zusätzliche* Einträge `smoker(sk-1).` bis `smoker(sk-6).` Aber es sind in der Ausgangslage ja schon Einträge zu 7 rauchenden Studenten vorhanden!

Also: Redundanz der Aussage testen. (d.h. ermitteln, was aus der schon in der Datenbank enthaltenen Information abgeleitet werden kann), und nur das assertieren, was nicht schon bekannt ist. Hier wäre also richtig gewesen: *Nichts* hinzufügen.

Was, wenn in der Datenbank nur 4 Studenten als Raucher ausgewiesen wären? Die *teilweise* Redundanz berücksichtigen, d.h. soweit, wie erforderlich, ‘‘auffüllen’’(hier mit zwei Einträgen: `smoker(sk-1).` und `smoker(sk-2).`).

```
student(a).
student(b).
student(c).
student(d).
student(e).
student(f).
student(g).
student(h).
student(i).
student(k).
student(sk-1).
student(sk-2).
```

```
smoker(a).
smoker(b).
smoker(c).
smoker(d).
smoker(sk-1).
smoker(sk-2).
```

Ad 2: Nunmehr also:

- Frage: ‘‘Wie viele Raucher gibt es? ’’
- Antwort: ‘‘6’’

Auch das ist falsch. Korrekter wäre ‘‘*mindestens* 6’’. Wenn wir ‘‘most’’ wie dargestellt behandeln, erzeugen wir ‘‘Pseudogenauigkeit’’. **Grund:** Wir haben die Vagheit von ‘‘most’’ eben gerade *nicht* als solche repräsentiert.

Ad 3: Auch die Antwort ‘‘mindestens 6’’ ist falsch!

Grund: Dasselbe Objekt kann mehrere Namen haben: `a` und `b` in `smoker(a)` und `smoker(b)` könnten sich durchaus auf dasselbe Individuum beziehen. (und `a` und `sk-1` *a fortiori* auch).

Man müsste die Identität eventuell explizit festhalten:

```
student(a).            smoker(a).
student(b).            smoker(b).
student(c).            smoker(c).
student(d).            smoker(d).
student(e).            smoker(sk-1).
student(f).            smoker(sk-2).
student(g).
student(h).
student(i).
student(k).
student(sk-1).
student(sk-2).
ident(a,b).
ident(e,h).
```
(und sonst keine - auch nicht sk-...)

Nun sind's 10 Studenten. Das ist möglich, wird aber teuer, da man bei jeder Operation, welche eine Konstante involviert, auf Identität hin testen muss! Und: Wir werden *nie* die Garantie haben, über alle Identitäten Bescheid zu wissen.

Schliesslich:

- Aussage: ''470 students smoke''

Soll das *wirklich* wie folgt dargestellt werden?!

```
student(sk-1).         smoker(sk-1).
student(sk-2).         smoker(sk-2).
student(sk-3).         smoker(sk-3).
  ...                    ...
student(sk-470).       smoker(sk-470).
```

Explizite Repräsentation ist extrem aufwendig und bringt wenig (da wir die Individuen ja nicht wirklich individuell kennen).

Und:

- Aussage: ''Wenige Studenten rauchen.''
- Frage: ''Rauchen die meisten Studenten?''
- Antwort: ''Nein.''

Die Antwort kann man ermitteln, ohne in einer expliziten Repräsentation

nachzuzählen.

Daher ev. besser ein Eintrag zur ganzen *Gruppe:*

```
set(student(sk-1)).

subset(sk-2,sk-1).
dstr(smoker(sk-2)).

card(sk-1,nr-1).
card(sk-2,nr-2).
minority(nr-2,nr-1).
```

Bemerkungen:

1. `set(student(sk-1))` entspricht dem Plural-Operator

   Ist auch nicht ideal: Man kommt nicht an die Skolem-Werte heran. Daher ev. geeigneter: `set(X,student(X),Xs)` (mit z.B. `Xs=sk-1`).

2. `dstr` soll heissen ‘‘distributiv über den Elementen der Gruppe’’

3. `sk-N` ist Skolem-Konstante für Objekt, `nr-N` für Zahlwert

Dann würde man über den explizit dargestellten *Mengenrelationen* (Mehrheit, Minderheit etc.) inferieren.

Zum Problem der Darstellung von ‘‘Pluralitäten’’ *siehe unten!*

6.2 Extensionale und intensionale Interpretation von Sätzen

Auch die oben (p. 63) unter dem Titel ‘‘Allaussagen vs. Regelaussagen’’ behandelte Problematik der extensionalen und intensionalen Antworten muss noch präziser betrachtet werden.

Dort (p. 61) nur erwähnt der Unterschied zwischen einer extensionalen und einer intensionalen Interpretation von Fragen wie ‘‘Sind alle Menschen intelligent?’’

Zwei Präzisierungen: Zur

1. intensionalen Interpretation universell quantifizierter Sätze
2. intensionalen Interpretation von Wh-Fragen

6.2.1 Intensionale und extensionale Interpretation universell quantifizierter Sätze

Der Begriff ‘‘universell quantifizierte Sätze’’ ist besonders im Englischen mehrdeutig: *every, all, each, 0.*

Unterscheiden sich u.a. bezügl. ihrer Intensionalität. Am unproblematischsten ist *‘‘every’’:* Klar extensional.

Fragen: mit *‘‘every’’*
Does every control code invoke a terminal command?
Ruft jeder Kontroll-Code einen Terminal-Befehl auf?

Daher wie gehabt: *Pro erfüllte Bedingung ‘P’* entsprechende Folgerung ‘Q’ *beweisen:*

```
∀ V: (P → Q)

beweise: P    beweise: Q
beweise: P    beweise: Q
beweise: P    beweise: Q
       ...
```

Implementation (prinzipiell): Doppelnegation mit Präsuppositionstest.

Aussage mit *‘‘every’’*
Every control code invokes a terminal command
Jeder Kontroll-Code ruft einen Terminal-Befehl auf

Bevorzugte Lesart erfordert hier:

Pro erfüllte Bedingung ‘P’ entsprechende Folgerung ‘Q’ assertieren:

```
∀ V: (P → Q)

beweise: P    assertiere: Q
beweise: P    assertiere: Q
beweise: P    assertiere: Q
       ...
```

Implementation (prinzipiell):

```
\+((P, \+(assert(Q))))
```

Details: siehe oben.

Etwas weniger klar:

Aussagen mit ‘‘*all*’’ resp. mit ‘‘bare plural’’:

(All) control codes invoke a terminal command
(Alle) Kontroll-Codes rufen einen Terminal-Befehl auf

Analog zur Idee bei Fragesätzen: Allaussage *als solche* assertieren, ev. in 2. Linie extensional assertieren.

Erste Idee:

```
declare(quant(all,V,P,Q)) :-
           assert(quant(all,V,P,Q)).
declare(quant(all,V,P,Q))   :-
           prsp(P), !, forall(P, declare(Q)).
```

Dann erforderlich ebenfalls: Modifikation des Beweisers.

Im Prinzip:

```
prove(Q)        :-  quant(all,V,P,Q),
                    prove(P).
```

Aber: *Q* kann komplex sein (inkl. eingebettete quantifizierte Aussage!). Das war der Grund zur Einführung der Klausel-Logik!

Also: Zweite Idee:

```
declare(quant(all,V,P,Q)) :-
            clausify((Q:-P), Clauses),
            assert_clauses(Clauses).
<plus 2. Regel oben>
```

Heisst natürlich, dass gewisse Aussagen *nicht* assertiert werden können.

Fragen mit ‘‘*all*’’ resp. mit ‘‘bare plural’’:

Do (all) control codes invoke a terminal command?
Rufen (alle) Kontroll-Codes einen Terminal-Befehl auf?

Oben: Allaussage *als solche* finden, ev. in 2. Linie extensional beweisen.

Daher jetzt (praktisch unverändert):

```
prove(quant(all,V,P,Q)) :-
            skolemize(V,V),
            assert(P), prove(Q),
            retract(P).
prove(quant(all,V,P,Q)) :-
            prsp(P), !, forall(P,   prove(Q)).
```

± **Offene Fragen:**

1. Bedeutung von ‘‘each’’?

   48) Marge admired each of her husbands

   49) ? Marge admired each of her uncles

   50) Marge admired each of her uncles *in a different way*

2. starke Interferenzen mit der Unterscheidung kollektiv/distributiv

   51) John saw every person in the room leave

   52) John saw each person in the room leave

3. und dasselbe kombiniert mit Generizität

   53) All doctors will tell you that Stopsneeze helps

54) Every doctor will tell you that Stopsneeze helps

55) Any doctor will tell you that Stopsneeze helps

56) Each doctor will tell you that Stopsneeze helps

4. Massivste Probleme mit Sätzen wie
   All smokers like most brands they smoke
   welche realistischerweise intensional (als Regel) interpretiert werden sollten. Soll man das übersetzen als

```
quant(most,B,(brand(B),smokes(S,B)),likes(S,B))
                                    :- smoker(S).
```

Hintergrundinformation (0.L.15)

Diese Probleme, zusammen mit den oben genannten 2 Gründen (Unsinnigkeit, 120 Einzeleinträge für ‘‘120 Studenten’’ zu erzeugen; Möglichkeit, aus ‘‘Wenige Studenten rauchen.’’ *intensional* abzuleiten, dass ‘‘Rauchen die meisten Studenten?’’ falsch ist) zwingen uns faktisch, Pluralitäten als solche zu repräsentieren. Dazu ein wenig mehr weiter unten.

Die Unterscheidung zwischen intensionaler und extensionaler Interpretation kann gemacht (und sogar explizit ausgedrückt werden) bei Wh-Fragen.

6.2.2 Intensionale Antworten auf Wh-Fragen

Einfachster Fall:

- Frage: ‘‘*Welche* Mitarbeiter verdienen mehr als 4000.- Fr. im Monat?’’
- erwartete Antwort: eine Liste von Namen ‘‘Müller, Meier und Schulze’’

Hingegen:

- Frage: ‘‘*Was für* Mitarbeiter verdienen mehr als 4000.- Fr. im Monat?’’
- erwartete Antwort: ‘‘Alle Mitarbeiter, welche Verkaufsleiter oder höher sind’’

Im ersten Fall will man den Begriffsumfang, die Extension, von ‘‘Mitarbeiter, welche...’’, im zweiten Fall die Begriffsdefinition, die Intension, die Regel, anhand derer man die Einzelfälle errechnen kann.

Die selbe Unterscheidung wird im Englischen gemacht. So fragt man mit

***Which* managers can hire personnel without the consent of the Personnel Department?**

direkt nach der Angabe der konkreten Einzelfälle, also nach der *Extension*, und man erwartet eine Antwort wie

Miller, Hart, and Jones.

Die Frage hingegen

***What* managers can hire personnel without the consent of the Personnel Department?**

fragt danach, welcher *Typ* von Manager von seiner Stellung her diese Kompetenz hat, und man erwartet eine Antwort wie zum Beispiel

All managers with the rank of Vice President or above.

Hier geht es also erneut um die *generelle Regel*, um die *Intension*, aus der man gegebenenfalls in einer konkreten Situation die Extension, die konkreten Einzelfälle, errechnen könnte.

Manchmal ist eine extensionale Antwort aber ungeeignet: Wenn man eine Frage stellt, die weder explizit intensional noch explizit extensional ist

- Frage: ‘‘*Wer* verdient mehr als 4000.- Fr. im Monat?’’
- extensionale Antwort: <eine Liste von 150 Namen>

Besser:

- Antwort: ‘‘30 der Sekretärinnen und 120 der Verkaufsleiter’

Man versucht, die Liste in Äquivalenzklassen aufzuteilen: Kriterien, nach denen man die ganze Liste in sinnvolle Teillisten aufspalten kann.

Noch sinnvoller (und viel, viel schwieriger) ist es, aus der Extension eine *Intension* zu *inferieren*. Am einfachsten ist es noch, wenn *alle* Objekte einer bestimmten Kategorie in einer extensionalen Antwort erwähnt sind. Dann kann man auf die obige Frage zum Beispiel antworten mit

- Antwort: ‘‘Alle Direktoren’’

Schwieriger, wenn man eine komplexere Inferenzregel erschliessen will.

Einfachere Beispiele:

Gegeben sei eine Menge von spezifischen Instanzen:

```
E1(a,[c],[a,c])
E2(b,[a,b],[b,a])
E3(c,[a,b],[c,a,b])
```

Finde (durch Termgeneralisierung) eine generelle Form:

```
E(X,[Y|Ys],[X,Y|Xs])
```

Noch schwieriger: ‘‘aktive Generalisierung’’ oder ‘‘Relationenlernen’’:

Gegeben seien die spezifischen Instanzen

```
E1(1,2)
E2(2,4)
E3(3,6)
```

Finde eine ‘‘constraint general form:’’

```
E(X,Y) :- Y=2*X
```

Beispiele aus:

Hintergrundinformation (0.L.16)

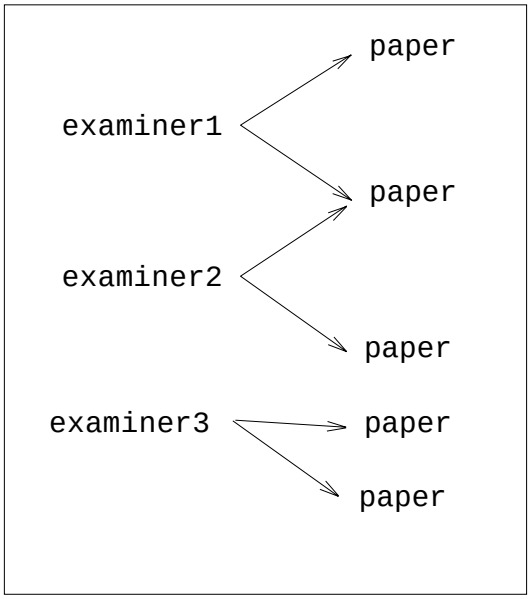
Das ist eine ‘‘höhere Art’’ von Datenkompression. Schwierig ist das Ermitteln einer Intension aus der Extension vor allem aus folgendem Grund: Während es (bei der üblichen Auffassung des Begriffs ‘‘Intension’’) zu jeder Intension in einer gegebenen Situation genau eine Extension gibt, ist das umgekehrte nicht der Fall: Man kann ein und dieselbe Liste von Einzelfällen in verschiedener Art kondensieren (daher: ‘‘*eine* Intension errechnen’’). So könnte ein und dieselbe Liste von Vielverdienern sowohl ‘‘komprimiert’’ werden zur Intension ‘‘alle Mitarbeiter, deren Vorname mit ‘B’ anfängt’’, oder aber zu ‘‘alle Sekretärinnen und der Direktor’’. Die erste Intension ist offensichtlich ‘‘nicht relevant’’ (d.h. die entdeckte Gemeinsamkeit beruht auf purem Zufall), während man im zweiten Fall annehmen kann, dass ‘‘dahinter System steckt’’. Aber wie soll das ein Programm merken?

6.3 Die verschiedenen Lesarten von Pluralen

Erst in den letzten Jahren hat man den Zahlwörtern mehr Beachtung geschenkt. Am klarsten werden die verschiedenen Lesarten von Plural-Nominalphrasen bei den Zahlwörtern:

57) Three examiners marked two papers

Die bisher immer unterstellte Lesart war die, dass jeder der drei Examinatoren zwei Papiere korrigierte, dass es also drei Examinatoren und maximal sechs Papiere gibt:



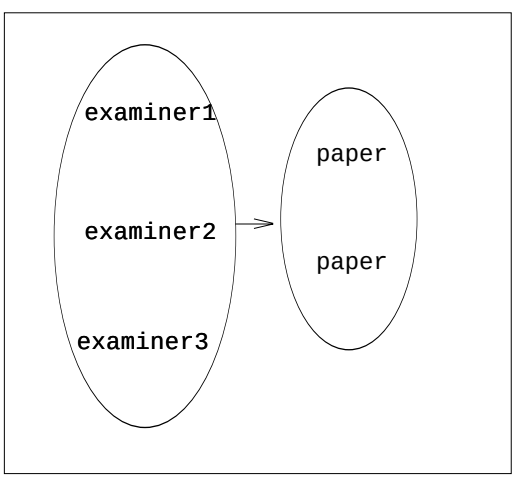
oder, unter der sehr unwahrscheinlichen zweiten Lesart, dass jedes der zwei Papiere von je drei Examinatoren korrigiert wurde: zwei Papiere, maximal sechs Examinatoren. Dies ist die *distributive* Lesart. Bei eindeutig relativen Determinatoren die einzig mögliche:

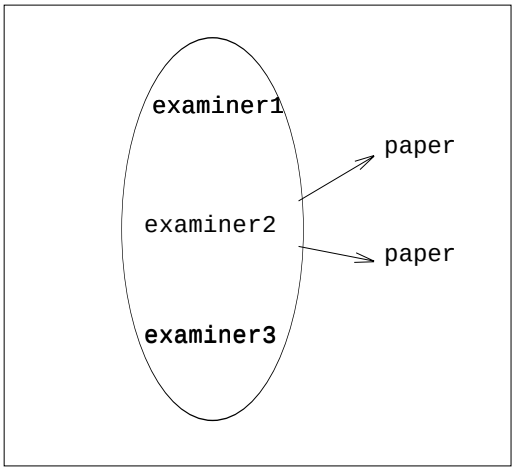
58) *Every* examiner marked two papers

kann nur heissen, dass es für jeden Examinator zwei Papiere (aber nicht notwendigerweise zwei verschiedene) gab.

Aber 57 kann eben auch heissen, dass das Kollektiv der drei Examinatoren zusammen korrigierte, und zwar zwei Papiere. Von keinem der Examinatoren kann man

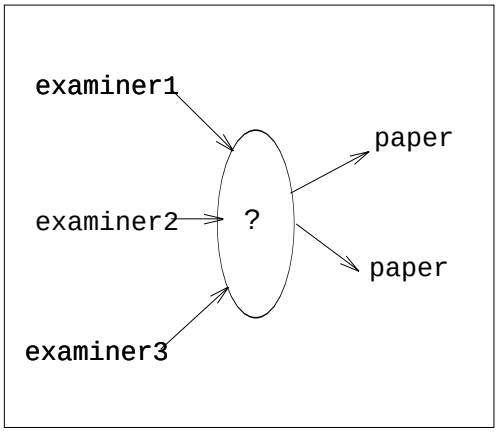
sagen, er habe ein Papier korrigiert (er hat bloss *an* einem Papier korrigiert - vielleicht hat er ausschliesslich die Korrektheit der Beweise kontrolliert, oder bloss den sprachlichen Stil). Die *kollektive* Lesart:





Das Kollektiv der Examinatoren kann dabei die zwei Papiere je einzeln korrigieren (Illustration rechts), oder aber es kann das Kollektiv der Papiere korrigieren (Illustration links); in diesem Fall würde keines der Papiere für sich bewertet (vielleicht, weil sie die zwei Teile einer Gruppenarbeit sind - nur die Gesamtarbeit erhält eine Note). *Jede* Argumentsposition eines Verbs kann also kollektiv interpretiert werden. In jedem Fall ist die Gesamtzahl von Examinatoren drei, und von Papieren zwei. Deshalb behandelt man die kollektive Lesart oft auch als einen der zwei Fälle von *‘‘Gruppenlesart’’*, eben jenen Lesarten, wo die Kardinalität der von den Nominalphrasen bezeichneten Mengen die explizit genannte ist.

Aber 57 kann noch etwas drittes heissen: Wiederum sind insgesamt drei Examinatoren und zwei Papiere involviert. Jeder Examinator hat mindestens einmal korrigiert, und jedes Papier wurde mindestens einmal korrigiert, aber wie genau die Zuordnung von Examinatoren und Papieren ist, wird offengelassen. Die *‘‘kumulative’’* Lesart:



Die Entscheidung, welche Lesart für eine Plural-Nominalphrase intendiert ist, ist oft nicht einfach. Wie schon erwähnt, scheinen die relativen Determinatoren obligatorisch distributiv zu sein:

59) Many/seven/some/several/all/most/few examiners marked two papers

scheint zumindest zuzulassen, dass mehr als zwei Papiere involviert sind (es wird zwar nicht zwingend verlangt - es könnten die gleichen Papiere wiederholt korrigiert worden sein - aber es wird sehr wahrscheinlich gemacht). In vielen Fällen kann die Bedeutung des *Verbs* zur Entscheidung beigezogen werden: So gibt es

1. *Verben*, die in ihrer *Subjektsposition* obligatorisch kollektiv sind:

   60) *The boys* gathered outside

   61) *The boys* dispersed quickly

   62) * The boy gathered/dispersed

2. Verben, die in ihrer *Objektsposition* obligatorisch kollektiv sind:

   63) The boys collected *the pebbles*

3. Verben, die in ihrer Subjekts- *und* Objektsposition obligatorisch kollektiv sind:

   64) *The boys* herded *the girls* together

4. *Adjektive*, deren modifiziertes *Objekt* kollektiv sein muss

   65) The boys were numerous

   66) Numerous boys were present

Manchmal hilft ein Pronomen, die Entscheidung zu treffen: Wenn wir 59 fortsetzen

mit

58a) All six had to be rejected because of faulty spelling

dann wissen wir, dass es sich um die distributive Lesart (und zwar distributiv über ‘‘three examiners’’) handeln muss. Die Fortsetzung

58b) Both had to be rejected because of faulty spelling

hilft hingegen nur wenig weiter; sie schliesst bloss aus, dass dieser Fall vorliegt (in allen andern Fällen *können* es zumindest zwei Papiere sein).

Oft wird die Situation auch durch adverbielle Ausdrücke klargemacht:

67) Three examiners *each* marked two papers

68) Three examiners *together* marked two papers

69) Three examiners *between them* marked two papers

wo man von der Gruppe der Examinatoren sagt, wie sie aufzufassen sind. Man kann das auch (in beschränktem Ausmass) kombinieren mit Präzisierungen zu den Papieren:

70) Three examiners *together* marked *each of* two papers

Hier sagt man offensichtlich, dass das Kollektiv der Examinatoren in zwei Arbeitsgängen beide von zwei Papieren korrigierte. *Siehe unten!*

Hintergrundinformation (0.L.17)

7. Konzeptualisierung und Ontologie

Zu diesem Thema ist einschlägig das Web-Buch ‘‘Wissensrepräsentation und Inferenz’’ von W. Bibel zusammen mit S. Hölldobler und T. Schaub ⇒hier.

‘‘Logik als Repräsentationssprache’’: bloss Wahl eines *formalen* Werkzeugs

Eine fundamentalere Entscheidung: Eine geeignete *Ontologie* (Konzeptualisierung):

- was für *Objekte*?
- was für *Relationen* zwischen ihnen?

Was sagt zu *Typen von Objekten* die

- traditionelle formale Semantik: ‘‘offensichtlich’’
- deskriptive Linguistik und Sprachphilosophie: ‘‘schwierig’’
- Tendenz in der formalen Semantik: wachsende ‘‘ontologische Promiskuität’’:
  1. Zustände
  2. Aktivitäten, Prozesse, Ereignisse
  3. Eigenschaften
  4. Typen von Objekten und ‘‘Zustände’’ (stages) von Objekten
  5. Materie und Masse
  6. Situationen
  7. u.v.a.m.

Was sagt zu *Typen von Relationen* die

- traditionelle formale Semantik: ‘‘offensichtlich’’
- deskriptive Linguistik: allgemeine *Rollen* für die Teilnehmer von Aktionen und Ereignissen (‘‘Tiefenkasus’’, ‘‘thematische Rollen’’)

Wozu sich entscheiden?

- kein Konsens absehbar
- die zwei Probleme hangen zusammen

Hier *viel* Sachen dazu (Ontolingua etc.):

Hintergrundinformation (0.L.18)

Hintergrundinformation (0.L.19)

7.1 Typen von Relationen

7.1.1 Tiefenkasus und thematische Rollen: Das Ausgangsproblem

Spätens seit 1968, als Fillmore sein ‘‘The Case for Case’’ veröffentlichte, sind Tiefenkasus oder thematische Rollen mit mehr oder weniger Intensität immer wieder behandelt worden. Sie haben sehr direkte Relevanz für jede auch einfache Implementation.

Als Ausgangspunkt kann man die Erkenntnis nehmen, dass die folgenden Sätze sich nur in ihrer ‘‘stilistischen Färbung’’ unterscheiden:

71) Mary hit Peter

72) Peter was hit by Mary

‘‘Mary’’ in 71 ist das grammatikalische Subjekt und ‘‘Peter’’ das grammatikalische direkte Objekt, während ‘‘Peter’’ in 72 das grammatikalische Subjekt ist (und ‘‘Mary’’ in einer Art Instrumental-Konstruktion steht). Dennoch sind die Sätze so synonym, wie man es sich wünschen kann. Es liegt daher nahe, *semantische* Tiefenkasus von den *syntaktischen* Kasus zu unterscheiden, also z.B. ‘‘Agent’’ (Mary) und ‘‘Betroffener’’ (Peter), und damit die verschiedenen syntaktischen ‘‘Oberflächen’’-Manifestationen ein und derselben zugrundliegenden Proposition ‘‘einzuebnen’’.

Es scheint ± klar, dass es bei einer Aktion genau einen Agenten gibt und genau einen Betroffenen. Insbesondere scheint zu gelten, dass ein Tiefenkasus, der nicht explizit genannt wird, automatisch ergänzt wird:

73) Peter was hit

ist fast synonym mit

73a) Peter was hit *by someone/something*

Man kann dann feststellen, dass gewisse Tiefenkasus (in der Wissensrepräsentation) obligatorisch realisiert sein müssen (insbes. der Agent), andere nicht.

Während diese Auffassung in sehr einfachen Fällen recht einleuchtend ist, wird sie bei etwas komplizierteren Fällen schnell problematisch. Die *Zahl und Art von Tiefenkasus* ist dabei das Hauptproblem:

Ist das Buch in 74 und 75

74) Mary gave Peter a book

75) A book was given to Peter by Mary

tatsächlich im gleichen Sinne ‘‘betroffen’’ wie Peter in 71 und 72? Es verändert zwar seine geographische und ev. juristische Position, aber es selbst verändert sich nicht und empfindet sicher auch nichts.

Und was ist die Rolle von Peter? Ist er der Begünstigte der Aktion (er hat etwas vom Buch) oder vielleicht nur ihr *Ziel* (er hat nichts davon)?

Und wenn er bloss das Ziel ist, ist dann Mary nicht vielleicht der Ausgangspunkt (source)?

Oder kann man gleichzeitig der Agent und der Ausgangspunkt sein? Oder der Agent und das Ziel (sich etwas geben lassen)?

Und auch, was die Obligatheit von Tiefenkasus betrifft, ist die Situation schwierig: Ohne besonderen Kontext ist

74a) * Mary gave a book

nicht grammatikalisch, während

75a) A book was given to Peter

korrekt ist, obwohl der Agent fehlt.

Und dann gibt es eine ganze Anzahl von Dingen, die manchmal, aber nicht sehr oft von einem Ereignis ausgesagt sind: Zeitpunkt, Art und Weise, Ort, Grund, usw.:

74) Mary gave Peter a book

76) Mary gave Peter a book secretly

77) Mary gave Peter a book behind the shed

78) Mary gave Peter a book in order to please him

Sind das nun alles semantische Rollen, die ebenso fundamental sind, wie Agent, Betroffener, Begünstigter usw., oder sind das irgendwie blosse ‘‘Hilfsrollen’’?

7.1.2 Zwei mögliche Ansätze: Argumentswerte oder explizite Rollen?

Die Theorien der Relationen fallen in zwei Hauptkategorien:

1. Theorien der geordneten Argumentswerte *(ordered argument theories)*
2. Theorien der expliziten Rollenangaben *(explicit role theories)*

Gemeinsamer Ausgangspunkt: ± unkontroverse konkrete Hauptverben

```
X sleeps        →    sleep(X)
X beats Y       →    beat(X,Y)
X gives Y to Z  →    give(X,Y,Z)
```

Argumentspositionen: *syntaktische* Rollen

Dann:

1. über syntaktischen Argumenten generalisieren
2. gemeinsame semantische Rollen für die Füller ableiten
3. semantische Rollen sollten auch aussersprachlich identifizierbar sein

Uneinigkeit fängt an:

A. Theorien der geordneten Argumentswerte sagen
   1. benutze *Argumentspositionen* für semantische Rollen
   2. semantische Rollen sind üblicherweise:
      - *agent* (experiencer)
      - *affected entity* (theme, patient, ...)
      - *goal* (beneficiary)

   bisher (stillschweigend) schon immer verwendet:

```
74b) give(mary,peter,book1).
     book(book1).
```

Agent, Betroffener und Begünstigter in dieser *Reihenfolge* als Argumente

1. nur je einen einzigen Agenten, Betroffenen und Begünstigten geben : Einmaligkeit
2. Ebenfalls festgelegt, dass jedes dieser Argumente obligatorisch einen Wert erhalten muss : Obligatheit
3. festgelegt, dass es eine endliche (und bekannte!) Anzahl von Tiefenkasus gibt : Abgeschlossenheit

Aber:

79) John gave Mary an apple *on Tuesday in the courtyard*

Eine neue Argumentsposition für jede neue Prädikation?

79a) `gave(john,apple1,mary,`**`tuesday1,courtyard1`**`)`
`∧ apple(apple1)`

Und:

1. *Art und Weise* (‘John is beating Mary *vigorously’*)
2. *Ursache* und/oder *Grund* (‘...*because* he hates her’)
3. *Adversative* (‘...*although* he was told not to’)

Wann aufhören?

B. **Alternative:** Theorien der expliziten Rollenangaben

1. Verwendung von *expliziten* Rollenprädikaten
2. kodiere ein *n*-stelliges Prädikat als *n+1* 2-stellige Prädikate
3. das zusätzliche Prädikat kann als eindeutiger Identifikator der Aktion etc. interpretiert werden (hier:sk-5)

```
79b) action(gave,sk-5).
                              agent(john,sk-5).
                              affected_entity(apple1,sk-
                        5).
                              goal(mary,sk-5).
                              time(tuesday1,sk-5).
                              location(courtyard1,sk-5).
                              apple(apple1).
```

Oft genannt der ‘‘neo-Davidson’sche’’ Ansatz; Davidson selbst schlug (vorher) eine Repräsentation wie in 93a vor, die mir vernünftiger scheint.

Auf den ersten Blick sieht das aus wie eine reine Alternativ-Notation: Man hat die Arguments-Positionen nicht mehr numeriert, sondern explizit genannt. Das hat eine Reihe klarer praktischer Vorteile:

1. man muss sich nicht an diese Positionen erinnern,
2. man muss nur das erwähnen, was man in einem konkreten Fall benötigt und
3. man muss sich nicht auf eine fixe Anzahl von Argumenten festlegen

Ist aber *nicht* eine rein ingenieurmässige Entscheidung!

Die zwei Notationen sind keineswegs äquivalent, und mit der Wahl einer Notation nimmt man eindeutig Stellung in den oben genannten Fragen. Bei der Wahl der Theorie der thematischen Rollen heisst das:

1. Man geht nicht mehr davon aus, dass jeder Tiefenkasus einen ‘‘Träger’’ haben *muss*
2. Ebenfalls wird die Einmaligkeit von Rollenträgern nicht mehr vorausgesetzt;
3. Schliesslich Art und Zahl der Rollen (im Prinzip) unbekannt

In allen Fällen könnte man natürlich die entsprechenden Einschränkungen explizit auf die Notation ‘‘aufpropfen’’, aber das wäre ein zusätzlich zu leistender Schritt, während sie bei der Notation der ‘‘geordneten Argumente’’ durch die Syntax ‘‘automatisch’’ erzwungen werden.

Besonders wichtig: Die Theorie der expliziten Rollenangaben nimmt eine Welt an, in der gilt:

1. Man legt eine *ereignisbasierte Semantik* zugrunde: Verbalphrasen denotieren Ereignisse

2. alle Rollenfüller sind *gleich wichtig*

7.1.2.1 Elemente einer umfassenderen Theorie: Ereignisbasierte Semantik

Ad 1: Handlungen und Ereignisse werden als *selbständige Objekte* in der Welt angenommen, d.h. sie sind *reifizierbar:* sk-5 bezeichnet eine Handlung (dies ist der Punkt, wo die Theorie der Relationen und die Theorie der Objekte interagieren)

Ist das sinnvoll? (Mindestens) folgende Gründe dafür:

1. Verbalphrasenmodifikation
2. Wahrnehmungsaussagen
3. Referenz auf Ereignisse
4. Quantifikation über Ereignisse

ad Verbalphrasenmodifikation:

80) Brutus stabbed Caesar in the back with a knife

81) Brutus stabbed Caesar in the back

82) Brutus stabbed Caesar with a knife

Cf. Parsons 1990:13ff.

Aus 80 folgen je 81 und 82, aber aus der Konjunktion von 81 und 82 folgt *nicht* 80:

B. mag C. bei zwei verschiedenen Gelegenheiten an zwei verschiedenen Orten gestochen haben, und dann sind 81 und 82 wahr, 80 aber nicht.

Auch ist der koordinierte Satz ‘‘Brutus stabbed Caesar in the back *and* Brutus stabbed Caesar with a knife’’ keineswegs synonym mit 80.

ad Wahrnehmungsaussagen:

83) Mary saw Brutus stab Caesar

84) Mary saw *that* Brutus stabbed Caesar

Diese Konstruktion funktioniert nur mit Wahrnehmungsverben! Auch ‘‘Neutral Perception Report’’ genannt.

Hier zu beachten: ‘‘Brutus stab Caesar’’ in 83 ist *nicht* in einem opaken Kontext, und tatsächlich folgt aus 83 und der Tatsache, dass Mary weiss, dass C. der Kaiser ist, dass Mary Brutus den Kaiser erstechen sah. Dasselbe gilt *nicht* bei 84.

In einer ereignisbasierten Semantik leicht zu erklären: Mary sah das Ereignis *selbst*.

ad Referenz auf Ereignisse:

85) They sang the Marseillaise. *It* was a moving experience

Ist eine sehr elementare Beobachtung, und m.E. unabweisbare Evidenz für eine ereignisbasierte Semantik.

Ebenfalls interessant: Beziehungen zwischen *impliziter* und *expliziter* Referenz auf Ereignisse. Vergleiche:

86) After the singing of the Marseillaise they saluted the flag

87) After the Marseillaise was sung they saluted the flag

Der einzige Unterschied an Bedeutung scheint die Einmaligkeitspräsupposition von 86 zu sein - genau, was man von einer definiten Nominalphrase erwartet. Geht aber nur, wenn man das Ereignis als Objekt betrachtet.

ad Quantifikation über Ereignisse:

88) In every burning, oxygen is consumed

89) Agatha burned the wood

90) Oxygen was consumed

Intuitiv ist `{88, 89} |- 90` eine klare Schlussfolgerung, aber nach klassischer logischer Auffassung gibts in 89 und 90 keine Quantifikationen.

7.1.2.2 Elemente einer umfassenderen Theorie: Relative Wichtigkeit thematischer Rollen

Ad 2 oben: Sind alle Rollenfüller gleich wichtig?

Beachte: Nur *einige* Argumentswerte sind

1. obligatorisch

2. einmalig

Vergleiche:

91) John gives Mary apples

92) * John gave Mary on Tuesday in the courtyard

In 91 überhaupt keine Zeitangabe!

Also:

- ‘‘Agent’’, ‘‘affected’’ entity, und ‘‘goal/beneficiary’’ *sind* obligatorisch und einmalig
- Zeit und Ort, Art und Weise etc. sind es *nicht*

Daher: Wünschbar eine Kombination

1. spezielle Argumentspositionen für spezielle Werte
2. plus ein zusätzliches Argument für den Handlungsidentifikator
3. andere Modifikatoren werden als zusätzliche Prädikate repräsentiert

93) John is beating Mary vigorously

wird zu

```
93a) beat(sk-6,john,mary).
      time(now,sk-6).
      manner(vigorous,sk-6).
```

Beachte das ‘‘now’’. Müsste zum Sprecherzeitpunkt ermittelt werden: Vergleiche in Unix `‘date‘`.

Um über dem Typ von Objekten und Handlungen quantifizieren und prädizieren zu können:

```
93b) action(give,sk-5,john,apple1,mary).
      time(tuesday1,sk-5).
      location(courtyard1,sk-5).
      object(apple,apple1).
```

Wie viele Prädikate für Umstandsbestimmungen? Mögliche Lösung:

1. ein paar für ± klare Fälle
   - ‘‘purpose’’
   - ‘‘method’’/‘‘tool’’
   - ‘‘location’’
   - ‘‘time’’

2. weniger weit abstrahiert für die andern :

   91) John gave Mary an apple *against Peter’s advice*

```
91a) action(gave,sk-5,john,sk-4,mary).
                          object(apple,sk-4).
                          object(advice,sk-3,peter,sk-6).
                          against(sk-5,sk-3).
```

   oder (*etwas* verallgemeinert)

```
relationship(against,sk-5,sk-3).
```

q Nun kann man, je nach Bedarf, neue Rollenprädikate einführen.

Wie kann man den Mangel an Allgemeinheit (teilweise) kompensieren?

1. Abhängigkeitsbeziehungen (entailments) zwischen diesen Relationen und anderen Fakten der Wissenbasis (siehe unten).
2. tiefenvariable Evaluation (variable-depth evaluation)

7.1.2.3 Elemente einer umfassenderen Theorie: Relationale Substantive

Thematische Rollen sind ebenfalls wichtig für die Modellierung relationaler Substantive:

```
student (of)                  → student(I,A)
characterization (by/of)      → characterization(I,A,O)
preference (by/of/over)       → preference(I,A,O,G)
conversion (by/of/into/from)→ conversion(I,A,O,G,S)
```

Vorteil: Generalisierungen über von solchen Substantiven bezeichneten Handlungen und Objekten wird einfach

Anfrage:

94) Characterizing preferences

gives

```
94a) ?- action(characterize,C,A1,P),
         object(preference,P,A2,O1,O2).
```

Dokument:

95) A new characterization of attachment preferences in English

```
95a) property(new,sk-2).
      object(english,sk-3).
      relationship(in,sk-4,sk-3).
      object(characterization,sk-2,sk-5,sk-4).
      object(attachment,sk-6,sk-7,sk-8,sk-9).
      object(preference,sk-4,sk-10,sk-6,sk-11).
```

Zwei Probleme:

1. es sollten unifizieren:

   ```
   action(characterize,C,A1,P)
   object(characterization,sk-2,sk-5,sk-4)
   ```

2. ‘‘characterization’’ ist vag:
   - ‘‘the *process* or *state* of characterizing’’

- ‘‘the *product* of characterizing’’

7.1.2.4 Elemente einer umfassenderen Theorie: Parametrisierung nach Prädikaten?

Bisher: ± unausgesprochene Annahme, *dass* es einen gemeinsamen semantischen Kern der Oberflächenkasus gibt.

Aber: Ist aber grundsätzlich fraglich.

Schon in den einfachsten Fällen, d.h. den Rollen ‘‘agent’’, ‘‘affected entity’’ und ‘‘goal/beneficiary’’ wird die Situation bald unklar. Ausser im allereinfachsten Fall, d.h. 96, gibt’s Probleme:

96) John sold Mary an apple

97) John bought an apple from Mary

98) The door opened

99) Mary wrote a book

100) Mary opened the door with a hammer

In 97 ist es unklar, ob John der Agent sein sollte oder ‘‘goal/beneficiary’’, oder beides. Und Mary? Ist sie auch ein ‘‘goal/beneficiary’’? (da sie ja Geld erhält).

In 98, ob die Tür der ‘‘agent’’ sein sollte (unbelebt?!), oder eine ‘‘affected entity’’, oder beides.

In 99, ob das Buch ‘‘affected entity’’ (noch nicht existierend!) oder ‘‘goal/beneficiary’’ (Ziel einer bewussten Aktion!) sein sollte

In 100, ob Mary oder der Hammer der Agent sein sollte, und ob die Tür oder der hammer die ‘‘affected entity’’ sein sollte (und die offen Tür vielleicht ‘‘goal/beneficiary’’?) etc.

Auch die Sprachunabhängigkeit ist fraglich.

Beispiele:

101) to enter *something*

102) entrer *dans* qch.

Auch die vermeintlich klare Beziehung Komplement:Tiefenkasus = Adjunkt/Modifikation ist schwierig:

103) Nick smashed the vase with a hammer yesterday

‘‘Nick’’ und ‘‘vase’’ sind klare Komplemente, ‘‘yesterday’’ ein klares Adjunkt, aber was ist ‘‘with a hammer’’?!

Somers ist zum Schluss gekommen (Somers 1986:200ff., auch Rosner 1980), dass es *keine* kleine, geschlossene Liste von thematischen Rollen gibt. Dagegen: Es gibt

1. vier ‘‘*inner roles*’’
   i. ‘‘source’’
   ii. ‘‘path’’
   iii. ‘‘goal’’
   iv. ‘‘local’’ (± affected entity)

2. sechs *Parameter*
   i. ‘‘locative’’
   ii. ‘‘temporal’’
   iii. ‘‘active’’
   iv. ‘‘objective’’
   v. ‘‘dative’’
   vi. ‘‘ambient’’

3. Resultat: Ein ‘‘case grid’’ von 24 thematischen Rollen

Beachte: Die Parameter entsprechen Typen von Eventualitäten. Also wohl keine echten semantischen Gemeinsamkeitn mehr (und keine gemeinsamen Inferenzen mehr). Letzteres könnte vielleicht sogar als Kriterium für das Ermitteln von Paramtern behandelt werden: Gemeinsame Informationmuster definieren Eventualitätstypen.

7.2 Typen von Zuständen und Aktionen

Die Unterscheidung von Verbtypen hat sich zu einer eigenen Wissenschaft entwickelt. Derartige Unterscheidungen sind aber keineswegs von rein theoretischem Interesse: Für computerlinguistische Anwendungen sind diese Dinge sehr relevant, da sie für die *Ableitbarkeit impliziter Information* direkt relevant ist.

Eine der frühesten Unterscheidungen stammt von Vendler (Vendler 1967). Er schlug eine Vierteilung vor:

1. Activities

   ```
   run (around, all over)
   walk
   swim (along, past)
   push (a cart)
   ```

2. Accomplishments

   ```
   run-a-mile
   paint-a-picture
   grow up
   recover from illness
   ```

3. Achievements

   ```
   recognise
   find
   win (the race)
   start/stop/resume
   be born/die
   ```

4. States

   ```
   desire
   want
   love
   hate
   dominate
   ```

Activities sind homogen: ‘‘Any part of the process is of the same nature as the whole.’’ Man kann überdies zu jedem Zeitpunkt sowohl sagen ‘‘Jones is running’’ sagen als auch ‘‘John has run’’.

Accomplishments drücken aus, dass man ‘‘etwas erreicht’’ damit. Sie sind deshalb nicht homogen. Sie haben eine eigene Zeitdauer. Wenn man weiss, dass A ein Bild gemalt hat, so kann man daraus schliessen, dass es nunmehr ein Bild *gibt* (∃!), oder dass ein bestimmter Wert sich *sichtbar verändert* hat (man ist nun erwachsen).

Achievements erfassen entweder den Anfang oder die Kulmination einer Aktion. Sie können selbst aber nicht *während* einer gewissen Zeitspanne ablaufen. Sie greifen bloss einen Abschnitt aus einer Handlung heraus. Man kann also kaum mehr in jedem Fall auf die Existenz von Objekten oder sichtbare Veränderungen schliessen (man hat etwas realisiert, aber das ist von aussen schwer feststellbar).

States sind keine Aktionen. Syntaktisch haben sie auch kein ‘‘progressive’’. ‘‘Though it may arise, or be acquired, as a result of a change, and though it may provide the potential of change, the state itself does not constitute a change.’’

Kenny (Kenny 1963) hingegen teilte das Feld in *drei* Kategorien auf:

1. Activities
2. Performances: sie fassen Vendlers ‘‘achievements’’ und ‘‘accomplishments’’ zusammen, d.h. ‘‘discover’’ und ‘‘find’’ ebenso wie ‘‘grow up’’ und ‘‘build a house’’.
3. States

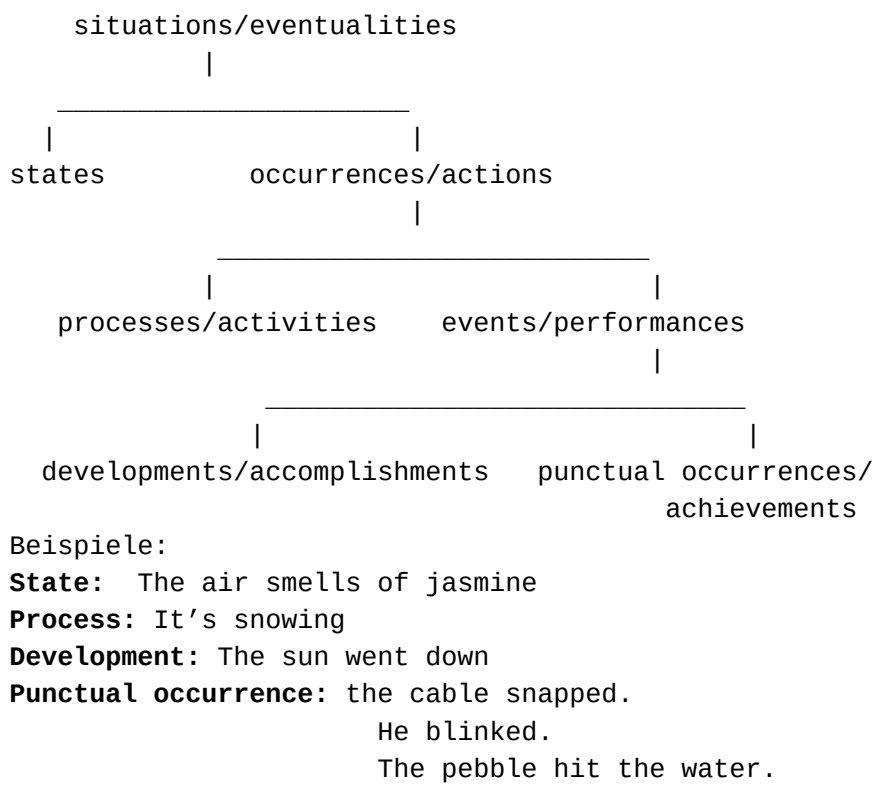
Der Grund, weshalb ‘‘accomplishments’’ und ‘‘achievements’’ zusammengefasst werden, liegt nach Kenny darin, dass beide ein Produkt, ein Resultat oder einen Ausgang umfassen. Es gibt kein ‘‘accomplishment’’ ohne den eng damit verknüpften Endpunkt eines ‘‘achievement’’; man kann nicht sagen ‘‘I wrote the letter’’ wenn man nicht auch sagen kann ‘‘I finished the letter’’. Sowohl ‘‘accomplishments’’ wie auch ‘‘achievements’’ benötigen Zeit, d.h. man kann in beiden Fällen sagen ‘‘it took him N minutes to X’’.

Was in beiden Kategoriensystemen nicht erfasst wird, sind die Aspektunterscheidungen, die wir (mehr oder weniger) in allen indoeuropäischen Sprachen haben (oder hatten)[19]

19. Vendler 1967:421

> The function of aspect is not limited to providing assignment to one or another of the categories activity-performance (accomplishment/achievement)-state. One of the major functions that lies outside the Kenny-Vendler typology is the encoding of patterns of frequency or habituation. [...] A wide variety of other points of information, also unrelated to the Kenny-Vendler typology, may be encoded through aspect - e.g.
>
> 1. endeavor
> 2. serialization
> 3. spatial distribution
> 4. temporary (contingent) state
>
> Moreover, even in those cases where the predication *is* classifiable under one or another of the Kenny-Vendler categories, the verb's aspectual marking does not by itself specify the relevant category. In all cases a total of six factors are involved:
>
> 1. the verb's inherent meaning
> 2. the nature of the verb's arguments, i.e. of the subject and the object(s), if any
> 3. adverbials, if any
> 4. aspect
> 5. tense as phase, e.g. the perfect
> 6. tense as time reference to past, present, or future.[20]

Daraus ergibt sich die folgende Aufteilung (der Ausdruck vor dem Schrägstrich ist vorgesehen für die Verwendung in Fällen, wo der ‘‘Handelnde’’ keine Person o.ä. ist):



Beispiele:
State: The air smells of jasmine
Process: It's snowing
Development: The sun went down
Punctual occurrence: the cable snapped.
He blinked.
The pebble hit the water.

Beispiel: Das Objekt des Verbs ''lends its character to the predication as a whole'':

104) He played a Mozart sonata

ist ein ''accomplishment'', während

105) He played a little Mozart

eine Aktivität ist. Aber die Prädikation *selbst* hat die gleichen Charakteristiken wie Massen- oder Zählbegriffe. Dies wird besonders klar, wenn wir die Nominalisierungsmöglichkeiten von Verbalphrasen mittels Suffixen betrachten (-ion, -ment, -al, -ure)[21] Man kann

106) John pushed the cart

21. Vendler 1967:425

(event/performance) paraphrasieren als

107) There was a pushing of the cart by John

und

108) Jones was painting the Nativity

(process/activity) als

109) There was painting of the Nativity by Jones

ohne indefiniten Artikel, ganz parallel zu ‘‘There was snow on the roof’’. Zustandsprädikationen hingegen können nicht als ‘‘gezählt-quantifizierte’’ Aussagen paraphrasisert werden, sondern nur als ‘‘gemessen-quantifizierte’’ Aussagen:

110) John hates liars

111) * There is a hating by John of liars

Das folgende, 112, hingegen ist o.k.

112) There is hate by John of liars

Nur Ereignis-Prädikationen (event predications) sind äquivalent mit ‘‘gezählt-quantifizierten’’ Konstruktionen. Ereignisse sind Situationen, welche direkt gezählt werden können. Also gibt es eine zweite Parallele: Ereignisse und Aktivitäten einerseits, und Objekte/Dinge/Substanzen und Eigenschaften/Qualitäten/Substanzen anderseits. Ein Objekt ist nicht homogen (eine Uhr besteht nicht aus Uhren), im Gegensatz zu Substanzen (alle Teile eines Klotzes Gold sind Gold), und man kann eine Substanz in Einzelteile ‘‘individuieren’’ (in Stücke, Flaschen, Masse usw.) und kann sie anderseits auch zusammenfassen in Abschnitte, Phasen usw.

Wenn man diese (und ähnliche) Unterscheidungen repräsentieren will, muss man weitere Sorten von Entitäten in den Objektsbereich einführen, und man muss ihm auch eine innere (algebraische) Struktur geben (Entaltensein z.B.). Anstatt näher darauf einzugehen, soll noch ein Thema zumindest erwähnt werden, das durch die letzten Paraphrasen der letzten Beispiele besonders aktuell wird.

7.3 Typen von Objekten

In den Fällen von kollektiver (und kumulativer) Lesart ist es unerlässlich, dass wir eine Menge von Objekten *als Menge* behandeln können, denn hier handelt es sich um *essentielle Plurale*. Für die Darstellung dieser Lesarten ist also eine Sprache erforderlich, die über die Logik erster Stufe hinausgeht. Allerdings ist das weniger harmlos, als es auf den ersten Blick aussehen mag: Die Interpretation der entsprechenden logischen Aussagen erfordert, dass wir *Mengen als elementare Objekte in der Welt* (und damit auch im Modell) verwenden. Solche essentielle Mengen werden weder von der Montague-Grammatik noch von der daraus entwickelten Theorie der generalisierten Quantoren erfasst. In beiden Theorien wendet man die nicht-logischen Prädikate (‘‘sleep’’ etc.) immer nur auf individuelle Objekte an, auch wenn man zuerst mit ganzen Mengen operiert hat (um deren Kardinalitäten zu vergleichen etc.).

Eine starke Tendenz der letzten 15 Jahre geht dahin, die Modelle der natürlichsprachlichen Semantik durch immer neue Typen von Objekten zu erweitern. Plural-Objekte sind hier nur eines von vielen Beispielen. Damit verliert aber der jeder Modelltheorie zugrundeliegende radikale Extensionalismus zunehmend von seiner intuitiven Berechtigung: Man mag noch akzeptieren, dass konkrete Objekte wie Äpfel und Tische ‘‘unmittelbar erkennbar’’ sind, dass man sich also nicht über deren Merkmale etc. unterhalten muss, aber bei diesen neuen Objekten ist das zunehmend weniger der Fall. Es scheint, dass man sich bei diesen Objekten zunehmend von der ‘‘realen Welt’’ entfernt und einer ‘‘mentalen Welt’’ von Konzepten annähert: Die Ausdrücke der Sprache beziehen sich dann nicht mehr auf die Objekte der Aussenwelt, sondern auf die Objekte der Innenwelt.

7.3.1 Mengen, Kollektionen und andere Pluralitäten

Man kann für Mengen ein besonderes Prädikat verwenden und die Kardinalitäten gesondert angeben:

```
set(Variable,Intension,Extension).
card(Extension,Card).
```

Dann kann man die bi-kollektive Lesart von 70 oben, also

70) Three examiners *together* marked *each of* two papers

so darstellen:

```
113) ∃ Es: set(E,examiners(E),Es) ∧ card(Es,3) ∧
     ∃ Ps: set(P,papers(P),Ps) ∧ card(Ps,2) ∧
     marked(Es,Ps)
```

Allerdings wird hier die *Basismenge* assertiert und nicht präsupponiert, was sicher unangebracht ist.

Einfach sind in dieser Notation die distributiven Lesarten (aber sie machen schliesslich auch sonst keine Probleme):

Every examiner marked two papers each

```
114) ∃ Es: (set(E,examiners(E),Es) ∧ card(Es,3)
      ∧ ∀ E: (element(E,Es) →
      ∃ Ps: (set(P,papers(P),Ps) ∧ card(Ps,2)
       ∧ ∀ P: (element(P,Ps) →  marked(E,P)))))
```

und die (unwahrscheinliche) zweite distributive Lesart

Every paper was marked by three examiners each

```
115) ∃ Ps: (set(P,papers(P),Ps) ∧ card(Ps,2) ∧
    ∀ P: (element(P,Ps) →
     ∃ Es: (set(E,examiners(E),Es) ∧ card(Es,3)
     ∧ ∀ E: (element(E,Es) → marked(E,P)))))
```

Wenn die Lesart distributiv über dem Subjekt und kollektiv über dem Objekt ist (‘‘Each of the three examiners marked one set of two papers’’ - but the papers of each set were the result of a collaborative effort and got no individual marks) gibt das:

```
116) ∃ Es: (set(E,examiners(E),Es) ∧ card(Es,3)
    ∧ ∀ E: (element(E,Es) →
     ∃ Ps: (set(P,papers(P),Ps) ∧ card(Ps,2) ∧
       marked(E,Ps))))
```

oder kollektiv über dem Subjekt und distributiv über dem Objekt (‘‘The examiners, as a collective, marked each of the two papers’’):

```
117) ∃ Ps: (set(P,papers(P),Ps) ∧ card(Ps,2)
     ∧ ∀ P: (element(P,Ps) →
     ∃ Es: (set(E,examiners(E),Es) ∧ card(Es,3)
     ∧  marked(Es,P))))
```

Schwieriger wird die kumulative Lesart:

```
118) ∃ Es: (set(E,examiners(E),Es) ∧ card(Es,3)
    ∧ (∃ Ps: (set(P,papers(P),Ps) ∧ card(Ps,2)
    ∧ (∀ E: (element(E,Es) →
       (∃ X: (element(X,Ps) ∧ marked(E,X)))))
    ∧ (∀ P: (element(P,Ps) →
       (∃ Y: (element(Y,Es) ∧ marked(Y,P))))))))
```

Ein Sonderfall der kumulativen Lesart, die sog. vollständige kumulative Lesart, ist die, wo jeder Examinator mindestens ein Papier korrigierte, und wo jedes Papier von jedem Examinator korrigiert wurde:

```
119) ∃ Es: (set(E,examiners(E),Es) ∧ card(Es,3)
    ∧ (∃ Ps: (set(P,papers(P),Ps) ∧ card(Ps,2)
    ∧ (∀ E: (element(E,Es) →
    (∀ P: (element(P,Ps) →  marked(E,P))))))))
```

Diese z.T. etwas komplizierten Darstellungen werden klarer, wenn man sie in Horn-Klausel-Logik übersetzt. Einige Beispiele:

```
113a) set(E,examiner(E),sk1).
      card(sk1,3).
      set(P,paper(P),sk2)).
      card(sk2,2).
      marked(sk1,sk2).
```

(kollektiv-kollektiv)

```
114a) set(E,examiner(E),sk1)).
      card(sk1,3).
      set(P,paper(P),sk2(E))) :- element(E,sk1).
      card(sk2(E),2)          :- element(E,sk1).
      marked(E,P)             :- element(E,sk1),
                                 element(P,sk2(E)).
```

(distributiv-distributiv)

```
117a) set(E,examiner(E),sk1)).
      card(sk1,3).
      set(P,paper(P),sk2)).
      card(sk2,2).
      marked(sk1,G)     :-     element(G,sk2).
```

(kollektiv-distributiv)

Man beachte, dass wir in allen diesen Fällen über Mengen sprechen, deren Identität uns unbekannt ist; das ist auch korrekt, wenn man vom Satz 70 ausgeht. Aber wir möchten auch die Möglichkeit haben, über Mengen zu sprechen, deren Zusammensetzung uns bekannt ist. Wenn Hart, Miller und Burt gemeinsam ein Paar von Papieren (die als Gemeinschaftsarbeit bewertet werden) korrigerien, so kann man das zum Beispiel so ausdrücken:

```
113b) set(E,examiner(E),sk1).
      set(P,paper(P),sk2)).
      card(sk2,2).
      marked(sk2,sk2).

      element(hart,sk1).
      element(miller,sk1).
      element(burt,sk1).
```

Es stellt sich aber die Frage, ob der mathematische Mengenbegriff überhaupt das ist, was wir brauchen. Zwar ist es sinnvoll, anzunehmen, dass wir Elemente nur einmal in einer Menge wollen (‘‘Peter, Mary, Sue, and Peter’’ ist eine zumindest sehr merkwürdige Art, sich auf eine Gruppe zu beziehen, wenn es *derselbe* Peter ist), aber eine andere Einschränkung ist weniger sinnvoll: So ist zum Beispiel die Menge

```
{a}
```

nicht dasselbe wie das Element

```
a
```

Und dementsprechend ist die Menge

```
{{a,b},c,{d,e}}
```

etwas völlig anderes als die Menge

```
{a,b,c,d,e}
```

und wenn man zwei Mengen ‘‘addiert’’

```
{a,b}
{c,d}
```

bekommt man eine Menge aus *zwei* Elementen (je eine Teilmenge)

```
{{a,b},{c,d}}
```

Aber das stimmt sicher nicht mit unserem intuititven Verständnis von ‘‘Mengenaddition’’ überein: Wenn wir sagen

120) John and Jill went to the zoo. There they met Peter and Mary

dann können wir *nicht* weiterfahren mit

120a) They *both* liked it there

im Sinne von ‘‘Beiden Paaren gefiel es je’’, und wir können damit sicher nicht nicht zum Ausdruck bringen, dass es allen vier gefiel. Zudem gibt es ganz einfach keine Möglichkeit in der natürlichen Sprache, den Unterschied zwischen einem Element und einer Menge aus diesem Element zum Ausdruck zu bringen.

Deshalb haben gewisse Leute eine Art ‘‘abgeschwächten’’ Mengenbegriffs vorgeschlagen, zum Beispiel sog. ‘‘Kollektionen’’[22] welche aus einmaligen, ungeordneten Elementen bestehen, wo aber ein Element und eine Kollektion aus diesem Element dasselbe ist. Man kann das in Prolog annähernd als sog. ‘‘runde Listen’’[23]

22. cf. Moore 1981:18
23. Ausdruck von Amble 1987:100

modellieren. In Standard-Prolog ist die runde Liste

```
((a,b),c,(d,e))
```

tatsächlich äquivalent mit

```
(a,b,c,d,e)
```

und ‘‘(a)’’ ist dasselbe wie ‘‘a’’. Allerdings wird natürlich ‘‘(a,b)’’ immer noch nicht mit ‘‘(b,a)’’ unifizieren; deshalb ist das bloss eine Annäherung.

Aber es gibt ein wichtiges Argument gegen die ‘‘naive’’ Abbildung von Pluralen als Mengen irgendwelcher Art (abgeschwächt oder nicht): Der einheitliche semantische Typ von Singular-Nominalphrasen war ja (wenn man die Intension einmal ignoriert)

```
John:               <<e,t>,t>
every man:          <<e,t>,t>
```

Wenn man die Gruppenlesart von Plural-Nominalphrasen und Konstruktionen wie ‘‘Peter and Mary’’ aber als Mengen darstellen würde, ergäbe das für die entsprechenden Nominalphrasen einen Typ

```
Three examiners:    <<<e,t>,t>,t>
Peter and Mary:     <<<e,t>,t>,t>
```

Wegen der Parallelität zwischen Syntax und Semantik würde dies aber erfordern, dass man z.B. für alle Pluralformen von Verben ebenfalls einen zusätzlichen syntaktischen Typ schaffen müsste, und dann für die Adverbien, welche mit den Verben kombiniert werden usw.

Mit anderen Worten: Die Einführung von Mengen mit entsprechend neuem logischen Typ würde sich durch das ganze logische und syntaktische System hindurch ausbreiten, was heisst, dass man eine offensichtliche Generalisierung nicht mehr macht (nämlich die, dass Plural und Singular zwei Erscheinungsformen ein und desselben sprachlichen Objektes sind).

Ein Verfahren (ev. ein Trick), das zu umgehen, besteht darin, *alle* Nominalphrasen in Mengen abzubilden, und wenn die Nominalphrase singular ist, ist’s halt eine Menge mit einem einzigen Element:

```
one dog:            set(D,dog(D),sk1), card(sk1,1).
two dogs:           set(D,dog(D),sk1), card(sk1,2).
```

Das hat zwar den Vorteil, einfach implementierbar zu sein, aber es ist wirklich nur

ein temporärer Ausweg, denn die Diskussion darüber, was genau diese Plural-Objekte sein wollen und wie sie intern strukturiert sein sollen, ist noch ganz offen[24] So scheint es schon so zu sein, dass Mengen (oder Kollektionen) weniger konkret sind, als individuelle Objekte: Wenn die drei Kinder von Peter sein Wohnzimmer in Unordnung gebracht haben, so kann man auf die Frage, wieviele Entitäten dabei beteiligt gewesen seien, wohl mit ‘‘drei’’ antworten, eventuell auch mit ‘‘eine’’ (das Kollektiv der Kinder), aber sicher nicht mit ‘‘sieben’’ (obwohl das die Anzahl von Teilmengen ist, die man aus den drei Elementen bilden kann).

Sodann wurde darauf hingewiesen, dass es Objekte gibt, die sozusagen definitionsgemäss Pluralitäten sein müssen: Komitee, Regierung etc. (und sie werden, im Britischen Englisch, in der Regel auch als Plural konstruiert). Ein Komitee ist aber nicht einfach die Zusammenfassung seiner Mitglieder; es hat u.U. ‘‘qua Komitee’’ Kompetenzen, welche keines seiner Mitglieder hat (also nicht einfach eine Summation von Macht/Einfluss/... der Einzelmitglieder, sondern etwas qualitativ Neues).

7.3.2 Masse und Substanz

Ein weiteres Phänomen, das zu einer zunehmenden Aufblähung des Modells zu führen scheint, sind Terme für Massen und Substanzen. Massenterme beziehen sich auf Dinge, welche sich (wie oben kurz erwähnt) ganz und gar nicht wie Objekte verhalten, und das spiegelt sich auch in der sprachlichen Verwendung von Massentermen wider. So kann man z.B. ohne weiteres sagen

121) The beer in my glass and the beer in your glass *was* poured from the same bottle

(mit einem Singular!), während wir bei abzählbaren Begriffen denPlural verwenden müssen:

122) The Queen and the Prime Minister *disagree* on this point

Die Verwendung des Singulars im Fall 121 spiegelt die Tatsache wider, dass man aus zwei Wassertropfen (oder dem Inhalt von zwei Biergläsern) *einen* Wassertropfen (den Inhalt *eines* Biergalses) machen kann.

Es wurde beobachtet, dass Massenterme und Pluralterme (resp. die von ihnen denotieren Phänomene) manches gemeinsam haben. So kann man bei beiden Phänomenen *Teile* des Gesamten erkennen, die nicht ‘‘Bestandteile’’ im gleichen

24. Landman 1989

Sinn sind, wie meine Hand ein ‘‘natürlicher’’ Bestandteil von mir ist. In einer Gruppe von drei Personen ‘‘1, 2, 3’’ kann man in beliebiger Weise (maximal sieben) Untergruppen bilden

```
{1 2 3}
{1 2}   {1 3}   {2 3}
{1}     {2}     {3}
```

und aus einer Masse von z.B. Wasser man kann in beliebiger Weise (beliebig viele) ‘‘Untermassen’’ bilden (man kann aus einem Menschen aber nicht beliebig viele Hände machen). Der Unterschied zwischen Plural-Termen und Massentermen besteht darin, dass es bei den Denotaten von Plural-Termen eben eine untere Grenze des Aufsplitterns gibt (es gibt *Atome*), während das (im Erlebnisbereich des Alltags) bei den Denotaten von Massentermen nicht der Fall ist. Sprachlich äussert sich der Unterschied darin, dass man bei Massentermen Masseinheiten braucht (‘‘zwei *Liter* Bier’’), während man bei den zählbaren Begriffen mit reinen Zahlwörtern auskommt. Daraus ergab sich die Hypothese, dass der Massenbegriff elementarer ist als der Zahlbegriff, eine Hypothese, welche für viele Linguisten sehr attraktiv ist.

Ohne hier näher in die (immer noch heiss umstrittenen) Details zu gehen können wir also feststellen, dass der ‘‘Objektezoo’’ der modelltheoretischen Modelle auch an dieser Front am Wachsen ist. Als anderes Beispiel dafür sind schon die Terme für Aktionen genannt worden.

7.4 Die ‘‘subatomare’’ Struktur lexikalischer Einheiten

7.4.1 Lexikalische Dekomposition oder Bedeutungspostulate?

Daher: Wir müssen:

- den *lexikalischen Einheiten* der Sprache eine Interpretation geben
- Inferenzregeln darüber definieren
- d.h. ein *semantisches Kalkül* definieren

Dazu prinzipiell zwei Methoden:

1. lexikalische Einheiten in (wohl wenige) Bedeutungsprimitive *zerlegen* (Dekomposition)

2. *Bedeutungspostulate* definieren, welche Abhängigkeitsbeziehungen zwischen Aussagen der Sprache definieren

Unterschiedliche ontologische Konsequenzen:

- der dekompositionelle Ansatz: nur wenige Typen von Entitäten
- der Bedeutungspostulats-Ansatz: erlaubt eine (prinzipiell beliebige) Ontologie

Dokumentretrieval: Bedeutungspostulats-Ansatz ist vorzuziehen:

1. inkrementell erweiterbar
2. weniger angreifbar ...
3. erlaubt einseitige Inferenzen (statt nur Äquivalenzen)

Drei Typen von Bedeutungspostulaten:

1. konzeptuelle Hierarchien
2. vollständige Begriffsdefinitionen
3. Inferenzregeln

konzeptuelle Hierarchien repräsentieren Typ/Subtyp-Information:

‘‘a concept is an idea’’

```
concept is_a idea.
```

vollständige Begriffsdefinitionen:

‘‘to design something’’ is
‘‘to create the concept of something in one’s mind’’

```
action(design,Ds,Ag,DO) <=  object(concept,C,DO),
                            action(develop,Dp,Ag,C),
                            tool(M,Dp),
                            object(mind,M,Ag).
```

‘‘Prolog’’ is
‘‘a language for computers that uses logic’’

```
object('Prolog',Lg)     <=  object(language,Lg),
                            object(computer,C),
                            beneficiary(Lg,C),
                            object(logic,Lc),
                            action(use,U,Lg,Lc).
```

Inferenzregeln: Fokus auf Relationen

Dokument:

123) Natural language question answering systems

Das *System* funktioniert (‘‘realized’’) *mittels* natürlicher Sprache

```
property(natural,sk-28).
object(language,sk-28).
object(question,sk-29).
object(system,sk-30).
relationship(by_with_for,sk-30,sk-28).
action(answer,sk-31,sk-30,sk-29).
```

Anfrage:

124) Natural language questions

Die *Fragen* sind in natürlicher Sprache formuliert (‘‘ realized’’)

```
?- property(natural,L),
   object(language,L),
   object(question,Q),
   relationship(by_with_for,Q,L).
```

Entweder: Extrem spezifische Regeln

```
If a system answering something functions by
means of natural language, then this something
is realized through natural language
```

oder Generalisierungen auf verschiedenen Ebenen:

```
If the agent Ag of action Ev does X by_with_for Y,
then the target X of Ev is realized by_with_for Y

relationship(by_with_for,X,Y)
                <-  action(Action,Ev,Ag,X),
                    relationship(by_with_for,Ag,Y).
```

8. Zitierte Literatur

[Amble 1987] Amble, T., *Logic Programming and Knowledge Engineering*, International Computer Science Series, Addison-Wesley, Wokingham etc., 1987.

[Barwise 1981] Barwise, J. and Cooper, R., ‘‘Generalized Quantifiers and Natural Language,’’ *Linguistics and Philosophy*, no. 4, pp. 159-219, 1981.

[Blackburn 1999] Blackburn, Patrick and Bos, Johan, *Representation and Inference for Natural Language A First Course in Computational Semantics*, Saarbrücken, 1999.

[Covington 1994] Covington, M.A., *Natural Language Processing for Prolog Programmers*, Prentice Hall, Englewood Cliffs, N.J., 1994.

[Dowty 1981] Dowty, D.R., Wall, R.E., and Peters, S., *Introduction to Montague Semantics*, Synthese Language Library, 11, Reidel, Dordrecht/Boston/London, 1981.

[Kenny 1963] Kenny, A., *Action, Emotion and Will*, New York, 1963.

[Kowalewski 1997] Kowalewski, S., ‘‘Konzeptorientierte Kompositabildung in der Textgenerierung,’’ Universität Koblenz-Landau Dissertation , April 1997.

[Kowalski 1979] Kowalski, R., *Logic for Problem Solving*, The Computer Science Library, 7, North Holland, New York/Oxford, 1979.

[Landman 1989] Landman, F., *Groups*, 1989. Manuscript

[Lohnstein 1996] Lohnstein, Horst, *Formale Semantik und natürliche Sprache: Einführendes Lehrbuch*, Westdeutscher Verlag Wiesbaden, 1996.

[Loveland 1978] Loveland, D.W., *Automated Theorem Proving: A Logical Basis*, Fundamental Studies in Computer Sciences, 6, North-Holland, Amsterdam/New York/Oxford, 1978.

[McCawley 1981] McCawley, J.D., *Everything that Linguists have Always Wanted to Know about Logic*, Blackwell, Oxford, 1981.

[Moore 1981] Moore, R.C., ‘‘Practical Natural-Language Processing by Computer,’’ SRI International Technical Note 251 , October 1981.

[Moore 1995] Moore, Robert C., *Logic and Representation,* CSLI lecture notes, Center for the Study of Language and Information, 1995.

[Naish 1986] Naish, L., *Negation and Control in Prolog,* Lecture Notes in Computer Science, 238, Springer, Berlin etc., 1986.

[Olsen 1986] Olsen, S., *Wortbildung im Deutschen,* Kröner, Stuttgart, 1986.

[Parsons 1990] Parsons, Terence, ‘‘Events in the Semantics of English: A Study in Subatomic Semantics,’’ Current Studies in Linguistics, The MIT Press, Massachusetts, 1990.

[Pereira 1987] Pereira, F.C.N. and Shieber, S.M., *Prolog and Natural Language Analysis,* CSLI Lecture Notes, 10, Center for the Study of Language and Information, Menlo Park/Stanford/Palo Alto, 1987.

[Rosner 1980] Rosner, M. and Somers, H., ‘‘Case in Linguistics and Cognitive Science,’’ *UEA Papers in Linguistics*, no. 13, pp. 1-29, June 1980.

[Russell 1995] Russell, Stuart J. and Norvig, Peter, *Artificial Intelligence: A Modern Approach*, Prentice Hall, Englewood Cliffs, N.J., 1995.

[Somers 1986] Somers, Harold L., *Valency and Case in Computational Linguistics,* Edinburgh University Press, Edinburgh, 1986.

[Tennant 1981] Tennant, H., ‘‘Evaluation of Natural Language Processors,’’ Ph.D. Thesis, University of Illinois, Urbana, Illinois, 1981.

[Thomason 1974] Thomason, R.H. (ed.), *Formal Philosophy; Selected Papers of Richard Montague,* Yale U.P., New Haven and London, 1974.

[Vendler 1967] Vendler, Z., *Linguistics in Philosophy*, Ithaka, 1967. One chapter only in xerocopy

[Warren 1983] Warren, D.S., ‘‘Using Lambda-Calculus to Represent Meaning in Logic Grammars,’’ in: *Proceedings of the 21st Annual Meeting of the ACL*, pp. 51-56, MIT, Cambridge, Mass., June 1983.

[Winograd 1972] Winograd, T., *Understanding Natural Language*, Edinburgh U.P., Edinburgh, 1972.

G

9. Inhaltsverzeichnis

Der Titel wird zuerst *syntaktisch* analysiert als (hier als gestaffelte Struktur; `cnp` ≈ `N'`)

```
np(det(a)
   cnp(cnp(adjp([]
                adjective(formal))
           cnp([]
               noun(specification)
               []
               []
               []
               []))
       cnp([]
           cnp([]
               noun(language)
               []
               []
               []
               [])
           pp(prep(for)
              np(det(the)
                 cnp(adjp([]
                          adjective(automatic))
                     cnp([]
                         noun(design)
                         np([]
                            cnp([]
                                noun(computer)
                                []
                                []
                                []
                                []))
                         np([]
                            cnp([]
                                noun(chip)
                                []
                                []
                                []
                                []))
                         []
                         []))))))
       []))
```

Versuch einer Rekonstruktion von Montagues Analyse von Skopus-Ambiguitäten

Wenn man statt von der Synthese eines Satzes von seiner Analyse ausgeht, kann man die skizzierten Grundideen zu Montagues Analyse von Skopus-Ambiguitäten vielleicht etwas verständlicher machen, indem man auf eine Technik der Syntaxanalyse zurückgreift. Insbesondere kann man die Rolle der ‘‘künstlichen Pronomina’’ etwas klarer beschreiben, wenn man sich an die ‘‘ungebundenen Verschiebungen’’ erinnert, u.a. an einen der einfachsten Fälle, die Wh-Bewegung. Der Satz

26) Who does the dog bite?

lässt sich auffassen als das Resultat einer (nicht-lokalen) *Linksextraposition* des Frageworts aus einer ‘‘Tiefenstruktur’’ sowie einer (lokalen) Transposition von Hilfsverb und Subjekt:

26a) The dog does bite who?

Um diese Verschiebungen besonders einfach zu erfassen, hat Pereira den Formalismus der Extrapositionsgrammatiken (‘‘XG’’) eingeführt:

```
q_word ... np(_)                --->     [who].
preposed_aux ... v(do,_,Nbr)    --->     v(do,_,Nbr).
```

Die Interpretation ist die, dass ein ‘‘q_word’’ als erkannt betrachtet wird, wenn ein ‘‘who’’ angetroffen worden ist, dass man sich aber gleichzeitg merkt, dass später einmal die Nominalphrase (die man nach dem Verb erwarten würde), fehlen darf. Zu diesem Zweck speichert man die Nominalphrase ‘‘np(_)’’ in einer speziellen Differenzliste, von wo man sie dann ‘‘bezieht’’, wenn sie scheinbar fehlt. Damit kann man sehr konzis Grammatiken schreiben, welche Aussage- *und* Fragesätze erfassen (ohne Aufbau von Syntaxstrukturen):

```
sent(decl)          -->     s.
sent(interr)        -->     q_word,
                            preposed_aux,
                            s.
sent(interr)        -->     preposed_aux,
                            s.

s                   -->     np(Num),
                            vp(Num).

np(Num)             -->     det(Number),
                            adjp,
                            n(Number).

q_word ... np(_)    -->     [who].
preposed_aux ... v(do,_,Nbr)
                    -->     v(do,_,Nbr).

vp(Number)          -->     verb_form(tr,Number),
                            np(_).
vp(Number)          -->     verb_form(itr,Number).

verb_form(Tr,Nbr)   -->     v(do,_,Nbr),
                            v(_,Tr,infin).
verb_form(Tr,Nbr)   -->     v(_,Tr,Nbr).

<etc.>

t1      :-      sent(X,[does,the,brown,dog,sleep],[],[],[]).
t2      :-      sent(X,[who,does,the,man,beat],[],[],[]).
```

Die Interpretation des ‘‘...’’-Infix-Operators erfolgt durch eine Differenzliste, welche das ‘‘gap-threading’’ implementiert. Die entsprechende Abarbeitung solcher Grammatiken erfolgt entweder interpretiert oder (normalerweise) compiliert.

Es gibt bekanntlich auch *Rechts*-Extrapositionen, wie z.B. die ‘‘heavy NP-Shift’’ in

the man t_1 arrived [who I told you about]$_1$

Dies ist im Formalismus der XG *nicht* auszudrücken. Daher könnten wir folgende Notation einführen, um diesen Beispielsatz erfassen zu können (nunmehr mit Aufbau von Syntaxstrukturen):

```
s(s(Np,Vp))          --> np(Np), vp(Vp).
np(np(Det,N,RC))         det(Det), n(N),
                         ...rel_cl(RC).
vp(vp(V))            --> v(V,intransitiv).
vp(vp(V,Np))         --> v(V,transitiv),
                         np(Np).
rel_cl(RC)           --> <etc.>
```

Der neue Präfix-Operator ‘‘...’’ sagt aus, dass die Konstituente ‘‘rel_cl’’ beliebig viel später auftreten darf (aber irgendwann auftreten muss). Man müsste sicherstellen, dass eine derartige Konstituente in einer weiteren Differenzliste gespeichert würde, und dass vor jeder *folgenden* Konstituente getestet würde, ob dort nicht vielleicht die noch ausstehende rechtsextraponierte Konstituente steht. Wenn ja, würde sie aus der Rechts-Extrapositions-Liste gestrichen, wenn nein, würde sie weitergeboten. Ein Satz wäre nur korrekt, wenn am Ende diese Liste leer ist.

Diesen neuen Formalismus kann man nun verwenden, um Montagues Lösung des Skopusamibguitätsproblems direkt zu implementieren. Man kann dazu folgende Regeln für die Konstruktion eines deklarativen Satzes verwenden: (***alles folgende ist ungetestet!***)

```
sent(decl,Sent) -->     s(Sent).
sent(decl,Sent) -->     xnp(NP,X),
                        s(S),
                        {reduce(@(NP,X^S),Sent)}.

xnp(NP,X)  ... [he(R^ @(R,X)))] -->     ...np(NP).
```

Die erste Regel sagt, dass Sätze wie bisher aufgbaut werden können (*direkte* Interpretation). Die zweite (neue) Regel sagt, dass Nominalphrasen ‘‘normalerweise’’ am Anfang eines Satzes stehen, dass sie aber eventuell rechtsexptraponiert sein können. Bei der Verwendung der zweiten Regel wird ein Pronomen in die Input-Liste eingeschoben. Die ‘‘künstliche’’ Konstituente `xnp` ist erforderlich

wegen der Eigenarten der XG (Einschübe erst nach Abarbeitung der gesamten Regel, hier: sent).

Man nimmt also an, dass beide folgenden Typen von Sätzen grammatikalisch korrekt sind:

27) Every boy invited a girl

27a) A girl, every boy invited her

Satz 27a entspricht ± der Version bei Montague mit syntaktischer Variablen.

Natürlich kommt ein Satz wie 27a nie vor, aber indem man ihn zulässt, lässt man eben auch zu, dass Beispiel 27 eine zweite Lesart erhält, *wie wenn* 27 durch Rechts-Extraposition aus 27a abgeleitet worden wäre. D.h. es wird angenommen, *vor* ‘‘Every boy’’ fehle etwas, von dem dann angenommen wird, dass es rechtsextraponiert worden ist. Man versucht dann, schon das ‘‘Every boy’’ als diese extraponierte Konstituente zu interpretieren, was aber nicht aufgeht, da dann das Subjekt von `s(S)` fehlen wird. Also wird weitergelesen, es wird ‘‘a girl’’ angetroffen und als extraponierte Konstituente interpretiert und als erledigt abgebucht. Dann wird das von der Regel eingefügte Pronomen `he(X)` als die an dieser Stelle zu erwartende Nominalphrase interpretiert.

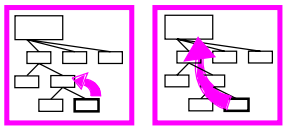
Der Aufbau der LF erfolgt dann exakt nach Montague, wobei die Lambda-Abstraktion von Montagues Regel 8 ($\mathbf{T}_{n,8}$(P,Q) = **P**(λ X_n. **Q**)) übernommen wird vom an verschiedenen Orten eingefügten *X*.

‘‘Every boy invited a girl’’

```
‘‘invited + her’’:
reduce(@( P^Q^ @(P, G^ @(@(invited,G),Q)),
          R^ @(R,G)), X).
==>
X =  Q^ @(@(invited,G),Q)

‘‘every + boy’’:
reduce(@( P^Q^all(B,@(P,B),@(Q,B)), boy), X).
==>
X =  Q^ all(B,@(boy,B),@(Q,B))
```

```
‘‘every boy + invited her’’:
reduce(@( Sˆall(B,@(boy,B),@(S,B)),
          Qˆ @(@(invited,G),Q)),X).
==>
X = all(B,@(boy,B),@(@(invited,G),B))

    von der Übersetzungsregel eingefügt
              ↓
‘‘a girl + (such that) every boy invited her’’:
reduce(@( Qˆexists(G, @(girl,G),@(Q,G)),
          Gˆall(B, @(boy,B), @(@(invited,G),B))),X).
==>
X = exists(G,@(girl,G),all(B,@(boy,B),
    @(@(invited,G),B)))
```

Ende des Unterdokuments



In voller Pracht sind diese Strukturen um einiges komplizierter, wie man im folgenden sieht. Die Frage

```
Question: Is there a capital in each country?
```

wird erwartungsgemäss so interpretiert, dass in jedem Land eine Hauptstadt gesucht wird (und nicht eine einzige Hauptstadt für alle Länder):

```
answer([]) :-
   { \+
       exists A
         country(A)
       & { \+
             exists B C
               in(B,A)
             & {capital(C,B)} } }
```

Die Quantorenanhebung (von *each*) ergibt:

```
pred(quant(void,X,‘true,‘true,[],feature&city-C),id,C=C1,
           [quant(det(a),
                  feature&city-CI,
                  ‘capital(C2,CI),
                  ‘true,
                   [quant(det(each),
                          feature&place&country-C3,
                          ‘country(C3),
                          ‘true,[],
                          feature&place&country-C4) & ‘in(CI,C4),
                          quant(void,feature&place&country-C2,‘true,‘true,[],
            feature&city-C1)]),Y,[],V)
```

und ergibt

```
quant(void,X,((true,true),true),feature&city-C),
      quant(det(each),
      feature&place&country-C3,
      ((country(C3),true),true),
      feature&place&country-C4),
      quant(det(a),
            feature&city-CI,
            (C2^(((true,true),true),
            capital(C2,CI),(true,in(CI,C4)),true,true),true),feature&cit
```

Die Details dieser Darstellung sind ohne umfangreiche Erläuterungen kaum verständlich zu machen; sie sind aber auch nicht sehr wichtig. Wichtig ist nur die Beobachtung, wie die Quantorenanhebung über derartigen *Zwischenrepräsentationen* durchgeführt wird.

Ende des Unterdokuments

Wilfried Meyer Viol, Movement in First-Order Logic

In Hilbert’s Epsilon Calculus every first-order formula has a quantifier-free equivalent. Such an equivalent consists of a formula in a term-logic with structured terms. The quantificational complexity of the original first-order formula is mirrored in the structure of epsilon terms occurring in quantifier-free equivalent. By the above translation every first-order formula can be parsed into a a pair consisting of logical form , a simple term logical formula, and a dependency structure : a finite strict partial order on the terms.

In this lecture we will discuss the parsing process, identify the class of formulas in the epsilon calculus that have first-order equivalents and consider the effect logical proof rules have on this paired structure. Proof rules will be seen to be either structure preserving , or structure transforming . The first type of rules leaves the dependency structure on terms invariant, but changes the logical form. The rules of the second type do not change the logical form but transform the dependency structure. Quantifier interaction principles will be seen to be of the second kind.

Changes in the dependency structure of a formula, without a change in the logical form will be related to the linguistic notion of movement . Various movement principles will be discussed together with their restrictions.

presented at: Logic, Structures and Syntax, 26th-28th September 1994, CWI, Amsterdam (http://www.cwi.nl/~mdr/lss.html)

Ende des Unterdokuments

Komposita mit Grundrelation

[Typen von Umstandsbeschreibungen im Englischen]

(0.L.13)

Weitere Beispiele von Umstandsbeschreibungen im Englischen sind zu finden in ⇒ Johannessons Englischkurs:

made of/from: bronze medal, rubber bullet, silver spoon; fish meal

made using: cherry brandy, ginger ale, olive oil; fish stick

consisting of: mountain range; pine grove, student group

similar to: cauliflower ear; spider plant

belonging to: computer screen

having ... in/at it: goldfish pond

occurring in: city centre; sea serpent

which can be kept in: pocket calculator; pocket knife

coming from: Labrador retriever; hospital bill

caused by: hay fever, virus disease

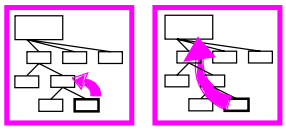
affecting: air pollution, swine fever

affected by: tennis elbow, shotgun wedding

intended for: dog biscuit, lobster crate, oboe reed

producing: oil well, toy factory

Ende des Unterdokuments



Generelle Implementationen
[Ausführliche Aufstellung]
(0.L.14)

Das lässt sich nun alles ins Lexikon aufnehmen, unter Verwendung der bekannten Techniken für die Implementation von Montague-Grammatiken (andere Notation!)[1]

1. : Quelle: Lucia Tovena

Zweistellige Determinatoren:

```
det(det(a),agr(sg,no),
  (X^X)^((P1^A)^(P2^B)^lf([A,B,intersect(P1,P2,[_])])))
                                                --> [a].
det(det(all),agr(pl,no),
  (X^X)^((P1^A)^(P2^B)^lf([A,B,subset(P1,P2)])))
                                                --> [all].
det(det(both),agr(pl,no),
  (X^X)^((P1^A)^(P2^B)^lf([A,B,subset(P1,P2,R),length(R,2)])))
                                                --> [both].
det(det(each),agr(sg,no),
  (X^X)^((P1^A)^(P2^B)^lf([A,B,subset(P1,P2)])))
                                                --> [each].
det(det(every),agr(sg,no),
  (X^X)^((P1^A)^(P2^B)^lf([A,B,subset(P1,P2)])))
                                                --> [every].
det(det(neither),agr(pl,no),
  (X^X)^((P1^A)^(P2^B)^lf([A,B,intersect(P1,P2,[]),
        length(P1,2)])))                     --> [neither].
det(det(no),agr(_n,no),
  (X^X)^((P1^A)^(P2^B)^lf([A,B,intersect(P1,P2,[])])))
                                                --> [no].
det(det(only),agr(pl,no),
  (X^X)^((P1^A)^(P2^B)^lf([A,B,subset(P2,P1)])))
                                                --> [only].
det(det(some),agr(_n,no),
  (X^X)^((P1^A)^(P2^B)^lf([A,B,intersect(P1,P2,[_|_])])))
                                                --> [some].
det(det(three),agr(pl,no),
  (Z^(LR is 3))^((P1^A)^(P2^B)^lf([A,B,intersect(P1,P2,R),
        length(R,LR),Z])))                       --> [three].
det(det(half),agr(pl,no),
  (Z^(LP1 is LR*2))^((P1^A)^(P2^B)^lf([A,B,intersect(P1,P2,R),
      length(R,LR),length(P1,LP1),Z])))      --> [half].
```

Definition des Operators ‘‘^’’ noch erforderlich.

Beispiele:[2]

‘‘A ballerina came’’:

```
s(np(detp(det(a)),n(ballerina)),vp(v(came)))

setof2(A,ballerina(A),B),setof2(C,come(C),D),
intersect(B,D,[E])
```

‘‘All ballerinas came’’:

```
s(np(detp(det(all)),n(ballerinas)),vp(v(came)))

setof2(A,ballerina(A),B),setof2(C,come(C),D),subset(B,D)
```

‘‘No ballerina came’’:

```
s(np(detp(det(no)),n(ballerina)),vp(v(came)))

setof2(A,ballerina(A),B),setof2(C,come(C),D),
intersect(B,D,[])
```

‘‘Both ballerinas came’’:

```
s(np(detp(det(both)),n(ballerinas)),vp(v(came)))

setof2(A,ballerina(A),B),setof2(C,come(C),D),
subset(B,D,E),length(E,2)
```

Analog lassen sich auch *komplexe Determinatoren* erfassen:

2. ‘‘In case of failure of setof, setof2 returns the null set’’

Dreistellige Determinatoren:

```
det(det(more),agr(pl,than),
  ((Zˆ(L > L1))ˆ(P1ˆA)ˆ(P0ˆC)ˆ(P2ˆB)ˆlf([A,C,B,
    intersect(P1,P2,R),intersect(P0,P2,R1),length(R,L),
                        length(R1,L1),Z])))
                                --> [more].
det(det(fewer),agr(pl,than),
  ((Zˆ(L < L1))ˆ(P1ˆA)ˆ(P0ˆC)ˆ(P2ˆB)ˆlf([A,C,B,
    intersect(P1,P2,R),intersect(P0,P2,R1),length(R,L),
                        length(R1,L1)])))
                                --> [fewer].

det(det(as,many),agr(pl,as),
  ((Zˆ(L is L1))ˆ(P1ˆA)ˆ(P0ˆC)ˆ(P2ˆB)ˆlf([A,C,B,
    intersect(P1,P2,R),intersect(P0,P2,R1),length(R,L),
                        length(R1,L1),Z])))
                               --> [as,many].
```

Beispiele:

‘‘More ballerinas than directors came’’:

```
s(np(detp(detp(det(more))),n(ballerinas),
postdet(particle(than),n(directors))),vp(v(came)))

setof2(A,ballerina(A),B),setof2(C,director(C),D),
setof2(E,come(E),F),intersect(B,F,G),intersect(D,F,H),
length(G,I),length(H,J),I>J
```

‘‘As many ballerinas as directors came’’:

```
s(np(detp(detp(det(as,many))),n(ballerinas),
postdet(particle(as),n(directors))),vp(v(came)))

setof2(A,ballerina(A),B),setof2(C,director(C),D),
setof2(E,come(E),F),intersect(B,F,G),intersect(D,F,H),
length(G,I),length(H,J),I is J
```

‘‘Fewer than three ballerinas came’’:

```
s(np(detp(detmod(fewer,than),detp(det(three))),
n(ballerinas)),vp(v(came)))

setof2(A,ballerina(A),B),setof2(C,come(C),D),
intersect(B,D,E),length(E,F),F<3
```

‘‘Half as many ballerinas as directors came’’:

```
s(np(detp(detp(detmod(half),detp(det(as,many)))),n(ballerinas),
postdet(particle(as),n(directors))),vp(v(came)))

setof2(A,ballerina(A),B),setof2(C,director(C),D),
setof2(E,come(E),F),intersect(B,F,G),intersect(D,F,H),
length(G,I),length(H,J),I*2 is J
```

Ende des Unterdokuments

LINGUIST List: Vol-9-804. Fri May 29 1998. ISSN: 1068-4875.

---------- Message 1 ----------

Date: Fri, 29 May 1998 12:11:29 +0100
From: Alice Drewery <alice@cogsci.ed.ac.uk>
Subject: all-every

Dear Linguist-list readers,

This is a very belated summary of the responses I received to my query about ‘all’ and ‘every’, which I posted back in February. Apologies for the length of time this has taken; I got distracted with other work.

My query was:

Can anyone point me to work done on the semantic differences between the quantifiers "all" and "every"? Clearly they don’t mean exactly the same, but has anyone studied exactly what the differences are?

First, many thanks to all those who responded:

Will Lowe, Massimo Poesio, Henry Thompson, Colin Matheson, Hiroaki Tanaka, Ash Asudeh, Line Mikkelsen, Ian Mackenzie, Christine Brisson, Jean-Francois Joubert, Larry Horn, Suzanne E Kemmer, Jean-Charles Khalifa, Georges Rebuschi, Michael B. Smith, Kate Kearns, Andrew McMichael, Rodger Kibble, Louise McNally, Yishai Tobin, Marie-Odile Junker, Retta Whinnery, David Houghton, Ananbel Cormack.

The canonical paper which many people referred to was:

Zeno Vendler (1967) "Each and Every, Any and All." in his book of essays, "Linguistics in Philosophy", Cornell University Press.

Other general references on all/every:

Aldridge, M.V. (1982) English Quantifiers: A Study of Quantifying Expressions in Linguistic Science and Modern English. London: Avebury.

Yishai Tobin, 1994. Invariance, markedness and distinctive feature analysis: A contrastive study of sign systems in English and Hebrew. Amsterdam: John Benjamins

Jim McCawley, 1981, "Everything Linguists Always Wanted to Know About Logic (but were Ashamed to Ask)"

Marie-Odile Junker, 1994, "French Universal Quantifiers in Conceptual Semantics", Linguistics, 32-2: 213-239.

Marie-Odile Junker, 1995, Syntaxe et smantique des quantifieurs flottants tous et chacun, Distributivit en smantique conceptuelle, Droz: Geneve, 1995: 183 p.

Many people pointed out that ‘every’ is distributive but ‘all’ often prefers a non-distributive reading which is not possible with ‘every’. References on this:

The chapters by David GIL & Martin HASPELMATH in Emmon BACH et al. (eds.), _Quantification in Natural Languages_, Dordrecht: Kluwer, 1995.

Beghelli and Stowell "Negation and Distributivity", in Szabolcsi (ed.) _Ways of Scope Taking_, Kluwer.

Godehard Link (1987) "Generalized Quantifiers and Plurals" in P. Gardenfors, Ed. "Generalized Quantifiers" Reidel Publishers.

David Dowty (1987) "Collective Predicates, Distributive Predicates, and All" in the proceedings of the 3rd ESCOL (Eastern States Conference on Linguistics).

Taub, Alison (1989) "Collective Predicates, Aktionsarten, and all" in Bach, Kratzer, Partee, eds., "Papers on Quantification" University of Massachusetts (not formally published).

Christine Brisson writes that her dissertation claims that ‘all’ is

not a quantifier at all, contrary to what is usually assumed, and that part of this was presented at SALT 7 last year.

Larry Horn sent me his recent CLS paper: "All John's Children are as Bald as the King of France: Existential Import and the Geometry of Opposition." from CLS 33, 1997, treating existential import of universally quantified statements, among other things. Further references on this are:

P F Strawson, 1952, Introduction to Logical Theory, London Methuen

J Moravcsik, 1991. "All A's are B's": Form and Content. Journal of Pragmatics 16, 427-441.

Several people referred me to the cognitive grammar standpoint, particularly Ron Langacker's work. References:

Ron Langacker, 1991, "Foundations of Cognitive Grammar", Stanford University Press.

Jean-Remi LAPAIRE and Wilfrid ROTGE (1991) "Linguistique et Grammaire de l'Anglais". Toulouse: Presses Universitaires du Mirail. (in French)

Jean-Charles Khalifa says:

Yours is now a standard problem for French linguists working under different frameworks, mostly enunciative theories inspired by Benveniste, Antoine Culioli or many others. I'm afraid their work is very little known outside this country, and written in French most of the time...

Anyway, to make a very long story short, "all" conveys the idea of totality and exhaustivity. In a mathematical-like set theory, this amounts to considering the whole set rather than the members of the said set. On the other hand, "every" is etymologically "ever-each". It involves an operation which is a lot more complex, beginning with

scanning EACH member of the set before considering the whole. This explains in particular why plurality is inherent in "all", when singularity remains at the very heart of "every".

I'm posting just this list of references now as it's so long since the original request, but I'd be keen to discuss the issues further with anyone who's interested.

Yours,

Alice Drewery.

- ----------
Alice Drewery Centre for Cognitive Science University of Edinburgh
alice@cogsci.ed.ac.uk tel: +44 (0)131 650 4436
- ----------

Ende des Unterdokuments

```
From js10@doc.ic.ac.uk Tue Jun 18 12:59:00 1996
Path: ifinews!rzunews.unizh.ch!swsbe6.switch.ch!swidir.switch.ch!in2p3.fr!oleane!jussieu.fr!math
From: js10@doc.ic.ac.uk (Joachim Schimpf)
Newsgroups: comp.lang.prolog
Subject: Re: references on term generalization ?
Date: 18 Jun 1996 10:59:00 GMT
Organization: Dept. of Computing, Imperial College, University of London, UK.
Lines: 72
Message-ID: <4q625k$fto@penguin.doc.ic.ac.uk>
References: <31BD8E98.2D8B@six.clo.tour.ist>
NNTP-Posting-Host: triumph.doc.ic.ac.uk
X-Newsreader: knews 0.9.6

Arch-image <aramante@six.clo.tour.ist> writes:
>I am looking for references in the field of logic programming.
>
>First, on term generalization;
>I.e
>               Given a set of ground examples:
>               E1(a,[c],[a,c])
>               E2(b,[a,b],[b,a])
>               E3(c,[a,b],[c,a,b])
>
>               Find a general form (by term generalization):
>               E(X,[Y|Ys],[X,Y|Xs])
>
>
>Next, and more particularly, I am interested by works on
>"active generalization" or "relation learning";
>I.e
>               Given a set of ground examples:
>               E1(1,2)
>               E2(2,4)
>               E3(3,6)
>
>               Find a constraint general form:
>               E(X,Y) :- Y=2*X
>
```

You might be interested to look at the work on Generalised Propagation.
It is about inferring (possibly approximate) generalisations from
disjunctions. There exists an implementation called Propia, which
comes as a library of the ECLiPSe system. Here is a small example
of how it works:

```
    [eclipse 2]: [user].
     e(1,2).
     e(2,4).
     e(3,6).

    yes.
    [eclipse 3]: e(X,Y) infers most.

    X = X{[1..3]}
    Y = Y{[2, 4, 6]}

    yes.
```

The generalisations are represented in terms of the constraint
system available, in this case the finite domains.

Further reading:

T. Le Provost and M. Wallace. Domain-independent propagation (or
    Generalised Propagation). In Proceedings of the International
    Conference on Fifth Generation Computer Systems (FGCS'92),
    pages 1004--1011, June 1992.

T. Le Provost and M. Wallace. Constraint satisfaction over the CLP
    Scheme. Journal of Logic Programming, 16(3-4):319--359, July 1993.
    Special Issue on Constraint Logic Programming.
    ftp://ftp.ecrc.de/pub/ECRC_tech_reports/reports/ECRC-92-1.ps.Z

Description of the Propia library:
    http://www.ecrc.de/eclipse/html/extroot/node68.html

About the ECLiPSe system in general:
    http://www.ecrc.de/eclipse/eclipse.html

```
--------------------------------------------------------------------------
 Joachim Schimpf                        /   phone:      +44 171 594 8187
 IC-Parc, William Penney Laboratory    /   mailto:J.Schimpf@doc.ic.ac.uk
 Imperial College, London SW7 2AZ     /   http://www-icparc.doc.ic.ac.uk


From js10@doc.ic.ac.uk Tue Jun 18 12:59:00 1996
Path: ifinews!rzunews.unizh.ch!swsbe6.switch.ch!swidir.switch.ch!in2p3.fr!oleane!jussieu.fr!math
From: js10@doc.ic.ac.uk (Joachim Schimpf)
Newsgroups: comp.lang.prolog
Subject: Re: references on term generalization ?
Date: 18 Jun 1996 10:59:00 GMT
Organization: Dept. of Computing, Imperial College, University of London, UK.
Lines: 72
Message-ID: <4q625k$fto@penguin.doc.ic.ac.uk>
References: <31BD8E98.2D8B@six.clo.tour.ist>
NNTP-Posting-Host: triumph.doc.ic.ac.uk
X-Newsreader: knews 0.9.6

Arch-image <aramante@six.clo.tour.ist> writes:
>I am looking for references in the field of logic programming.
>
>First, on term generalization;
>I.e
>               Given a set of ground examples:
>               E1(a,[c],[a,c])
>               E2(b,[a,b],[b,a])
>               E3(c,[a,b],[c,a,b])
>
>               Find a general form (by term generalization):
>               E(X,[Y|Ys],[X,Y|Xs])
>
>
>Next, and more particularly, I am interested by works on
>"active generalization" or "relation learning";
>I.e
>               Given a set of ground examples:
>               E1(1,2)
>               E2(2,4)
```

```
>               E3(3,6)
>
>               Find a constraint general form:
>               E(X,Y) :- Y=2*X
>
```

You might be interested to look at the work on Generalised Propagation. It is about inferring (possibly approximate) generalisations from disjunctions. There exists an implementation called Propia, which comes as a library of the ECLiPSe system. Here is a small example of how it works:

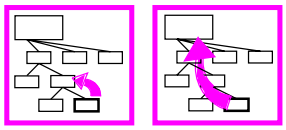
```
[eclipse 2]: [user].
 e(1,2).
 e(2,4).
 e(3,6).

yes.
[eclipse 3]: e(X,Y) infers most.

X = X{[1..3]}
Y = Y{[2, 4, 6]}

yes.
```

The generalisations are represented in terms of the constraint system available, in this case the finite domains.

Further reading:

T. Le Provost and M. Wallace. Domain-independent propagation (or Generalised Propagation). In Proceedings of the International Conference on Fifth Generation Computer Systems (FGCS'92), pages 1004--1011, June 1992.

T. Le Provost and M. Wallace. Constraint satisfaction over the CLP Scheme. Journal of Logic Programming, 16(3-4):319--359, July 1993. Special Issue on Constraint Logic Programming. ftp://ftp.ecrc.de/pub/ECRC_tech_reports/reports/ECRC-92-1.ps.Z




```
Description of the Propia library:
    http://www.ecrc.de/eclipse/html/extroot/node68.html

About the ECLiPSe system in general:
    http://www.ecrc.de/eclipse/eclipse.html


------------------------------------------------------------------------
 Joachim Schimpf                       /   phone:      +44 171 594 8187
 IC-Parc, William Penney Laboratory    /   mailto:J.Schimpf@doc.ic.ac.uk
 Imperial College, London SW7 2AZ      /   http://www-icparc.doc.ic.ac.uk
```

Ende des Unterdokuments

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2//EN">
<HTML>
<HEAD>
    <TITLE>LINGUIST List 5.234: Quantifiers</TITLE>
    <META name="GENERATOR" content="iml2html">
    <META name="ORGANIZATION" content="The LINGUIST Network">
</HEAD>
<BODY BGCOLOR="#FFFFFF" LINK="#CC0000" VLINK="#0066CC" ALINK="#FF3300">
<H1 ALIGN=center>LINGUIST List 5.234</H1>
<H2 ALIGN=center>Sun 27 Feb 1994</H2>
<H2 ALIGN=center>Disc: Quantifiers</H2>
<P ALIGN=center><STRONG>Editor for this issue:</STRONG>  &lt;<A HREF="mailto:"></A>&gt;
</P>
<HR>
<A NAME="dir"> </A>
<H2>Directory</H2>
<OL>
<LI>Carpenter, <A HREF="#1">quantifiers</A>
</OL>
<HR>
<A NAME="1"><H3>Message 1: quantifiers</H3></A>
<STRONG>Date:</STRONG> Sun, 27 Feb 94 15:57:05 esquantifiers<BR>
<STRONG>From:</STRONG> Carpenter &lt;<A HREF="mailto:carp@lcl.cmu.edu">carp@lcl.cmu.edu</A>&gt;<
<STRONG>Subject:</STRONG> quantifiers<BR>
<PRE>


This message was originally constructed in response to David Gil's
posting about why people try to come up with 'complex' theories of
quantification.  Hopefully, this will be interesting to a wider
audience than just those who are following Gil's thread.  The points
address those raised in Gil's original message.

Point 1)
There is an added complication with 'two men love three women' in that
you have plural noun phrases.  This leads to interactions of scope
with distributive and collective readings.  The 'standard wisdom' for
these can be found in the non-theory-bound paper of Davies
(_Linguistics and Philosophy_ 1989, 293--324), &quot;Three examiners marked
six scripts&quot;, the Heim, Lasnik and May _Linguistic Inquiry_ paper on
```

reciprocity (1991), containing a GB version, or I could send anyone who's interested my own unpublished CG account. Davies most clearly states the possible readings for such a sentence, based on TWO quantifiers for each noun phrase:

```
   Exists X, a set of two men
   Exists Y, a set of three women
   ( forall x in X (distributive)
     OR for some x formed of a group of X (collective))
   ( forall y in Y (distributive)
     OR for some y formed of a group of Y (collective))
```

alternated so that there are no free variables (there are well-founded mechanisms for guarnteeing this in both Cooper-storage, Montagovian, GB and HPSG accounts). These possibilities account for the four readings you suggest, along with the other possibilities, most clearly seen in cases such as Davies'. These readings (with your possibilities in brackets):

```
  Exists X, forall x in X, Exists Y, forall y in Y  love(x,y)  [I]
  Exists X, forall x in X, Exists Y, some group y of Y  love(x,y)
  Exists X, some group x of X, Exists Y, forall y in Y  love(x,y)
  Exists X, some group x of X, Exists Y, some group y of Y  love(x,y) [IV]

  Exists Y, forall y in Y, Exists X, forall x in X  love(x,y)  [II]
  Exists Y, forall y in Y, Exists X, some group x of X  love(x,y)
  Exists Y, some group y of Y, Exists X, forall x in X  love(x,y)

  Exists X, Exists Y, forall x in X, forall y in Y  love(x,y)  [III]
```

The other readings, from different scopings, are all logically equivalent to one of the ones above.

(Note that this assumes the hypothesis of Partee (ms) and later Roberts (1989) that the cumulative readings of Scha (1984?) are group-group readings, your reading IV.) Also note that none of these readings require branching quantifiers, but rather stem from grouping rather than distributing. Davies used branching quantifiers for the last case above, your III, as you suggest. But the analyses I just

listed, basically those of Heim et al., show that they aren't necessary.

Point 2)
What's your empirical evidence that humans generate limited readings of quantifier scopings as opposed to those proposed in the theoretical semantics literature? Is it psycholinguistic, or did you just ask people in the null context?

I'd suggest looking at:

Howard Kurtzman and Maryellen MacDonald (1993) "Resolution of quantifier scope ambiguities", _Cognition_, 48:243--279.

They conclude that scope preferences for readings are NOT structurally determined, but rather stem from preferences having to do with lexical semantics of the terms involved. And furthermore, they cite reaction time and priming evidence which indicates that multiple readings are computed in parallel on-line.

Point 3)
I didn't see the relevance of branching quantifiers. The only motivation for branching quantifiers with which I'm familiar (and I also asked Jon Barwise, albeit he was responding off the top of his head at a conference) involves the sentences above which Heim et al. demonstrated don't need that additional device. So as far as I can tell, branching's inessential.

Point 4)
This is also undercut by the above discussion. As you claim that there are two equivalent readings, one with branching, and one with wide-scope existential, I don't see that you can conclude that:
"(1) should accordingly be represented not with wide scope for the existential quantifier (as per alternative (a) Point 3), but rather with branching universal and existential quantifiers (as per alternative (b) Point 3) -- contrary to Claim B above."

End of response to your arguments. And by the way, Davies does respond to Kempson and Cormack (1981: Linguistics and Philosophy), who

as far as I can understand Davies' portrayal of K&C and your portrayal of your own arguments, are making similar points, and Davies also cites Tennant (Linguistics and Philosophy 1981) as responding to K&C's claims.

I'm not sure your argument is coherent (reason (a)), because it's not stated precisely enough (which I guess is your reason (b)). For instance, what principle do you provide that will do the right thing for the following two sentences?

A kid likes every toy.
Every kid likes a toy.

Here the problem is in the syntax/semantics mapping and making sure that you get the same scope altenations for both sentences (one branching, one wide-scope universal, if I understand you correctly).

At this point in time, I'd also say that your argument is fairly irrelevant, given the analyses of Heim et al., which can be easily transferred from GB to your favorite syntactic theory.

Note that there is no 'elaborate theoretical edifice' which has been built to get the above readings. They're very natural -- just look at the Heim et al. article. And I don't see how you can get around something similar for quantifier scoping and still get all the possible readings, even if you do allow branching, which itself goes beyond classical first-order logic, if that's your base-line for determining what's an 'elaborate theoretical edifice'.

Furthermore, how do you account for the interaction of quantifiers with control, unbounded dependencies, extraction islands, coordination, de-dicto/de-re intensional verbs, negation, adverbs, adjuncts, embedded sentences, etc.? My own theory, in logical Categorial Grammar (Lambek style + scope) minus the plurals, can be gotten via anonymous ftp in compressed format (use binary mode, then uncompress) on j.gp.cs.cmu.edu by cd-ing to /usr1/carp/ftp/ and getting the file quants.ps.Z (in binary mode, of course).

I have to admit that I haven't read Aoun and Li's book. I find GB (or

whatever acronymned theory has descended from it lately) pretty
impenetrable at best, and assume it's in that framework. If it's as
good as Heim et al., though, I'll gladly wade through it. I would
assume they at least address the problems stated in the previous
paragraph.

Please send responses to me directly rather than to the list. I'll
summarize and repost if there are multiple comments.

- Bob Carpenter
  Computational Linguistics Program, Philosophy Department
  Carnegie Mellon University, Pittsburgh, PA 15213
  Net: <A HREF="mailto:carp@lcl.cmu.edu">carp@lcl.cmu.edu</A> Phone: (412) 268-8043   Fax: (4

</PRE>
<FONT SIZE=-1>
<A HREF="mailto:carp@lcl.cmu.edu">Mail to author</A>|<A HREF="mailto:linguist@linguistlist.org">
<HR>
</BODY>
</HTML>

Ende des Unterdokuments

```
<TITLE>Sharable Ontologies Library</TITLE>
<H1>Sharable Ontology Library</H1>

<BLOCKQUOTE>
<B>
<font size=+1>
 This used to be a static
library but is now available through the  <P>
<a href="http://www-ksl-svc.stanford.edu:5915/">KSL Interactive Ontology
Server</A> <P>  <B>The Interactive Ontology Server has many new and updated ontologies; the stat
</B>
</font>
</BLOCKQUOTE>
<hr>

 <P>

This is an electronic library of
public ontologies.
<a href="http://www-ksl.stanford.edu/kst/what-is-an-ontology.html">Ontologies</A> are specificat
used to help programs and humans share knowledge <A HREF="#2">[2]</A>.
In practice they consist of definitions of representational
vocabulary, some including axiomatic theories.
Most of the files here are in a machine-readable form:
<A HREF="../kif/">KIF</A> <A HREF="#1">[1]</A> sentences in forms that <A HREF="../ontolingua/or
into implemented representation languages.

 <P>
<H2>Browsing the Ontology Library</H2>

The ontologies are
available in the form of <B>cross-indexed hypertext documents</B>.

The web comprising all the available ontologies and accompanying documentation
is found in the /html subdirectory, starting with the
<BLOCKQUOTE>
<A HREF="../ontologies/html/index.html"><B>Ontology Library Table of Contents</B></A>
</BLOCKQUOTE>
Look at
the <B>Reference Documents</B> section of the index first; these documents introduce and link ir
```

```
<H2>Downloading the Ontologies</H2>
The ontologies are also available in this directory by anonymous ftp.

The root of the ftp ontology library is
<BLOCKQUOTE>
<A HREF="ftp://ksl.stanford.edu/pub/knowledge-sharing/ontologies/">
ksl.stanford.edu: /pub/knowledge-sharing/ontologies/</A>
</BLOCKQUOTE>
 <P>
It is suggested that you browse the ontologies in their nice, hypertext form first (see above) a
The following files contain the ontology library
in various languages:
<UL>
<LI> <A HREF="ol-ontologies.tar.Z">ol-ontologies.tar.Z</A> Ontolingua source

<LI> <A HREF="kif-ontologies.tar.Z">kif-ontologies.tar.Z</A>
<LI> <A HREF="loom-ontologies.tar.Z">loom-ontologies.tar.Z</A>
<LI> <A HREF="clips-ontologies.tar.Z">clips-ontologies.tar.Z</A>
<LI> <A HREF="generic-frame-ontologies.tar.Z">generic-frame-ontologies.tar.Z</A>
<LI> <A HREF="epikit-ontologies.tar.Z">epikit-ontologies.tar.Z</A>
</UL>

 <P>
<BLOCKQUOTE>

<B>Note:</B> <EM>It is very possible that the .tar.Z files will be older than
the latest offerings in hypertext form.  Check the file dates carefully.  To
get the latest individual ontolingua source document, you can view the source
file from the hypertext interface and then use your WWW browser to save it as
a plain text file.</EM>

</BLOCKQUOTE>

 <P>
<B>If you live behind a firewall,</B> (i.e., you can't access internet addresses outside your si
It contains the ontology web as a self-contained directory of files that you can browse locally
 <P>

The ontologies posted here may change and no claims or
```

```
warrants are made.  They may be copyrighted, but the authors
give permission to freely distribute the files with the copyright
label intact.
 <P>
<H2>Contributing to the Ontology Library</H2>
We welcome additions to the ontology library. There is now a reasonable body
of examples to work from.  The procedure for
contributing an ontology is described in <A HREF="how-to-submit-an-ontology.html">how-to-submit-


<H2>For More Information</H2>

<DL COMPACT>


<DT>
<A NAME="1">[1]</A>
<DD>
M.  R.  Genesereth, R.  E.  Fikes (Editors).  Knowledge
Interchange Format, Version 3.0 Reference Manual.  Computer Science
Department, Stanford University, Technical Report Logic-92-1, March
1992. <A HREF="../papers/README.html#kif">Available on line.</A>
 <P>

<DT>
<A NAME="2">[2]</A>
<DD>
T.  R.  Gruber.  A Translation Approach to Portable
Ontology Specifications.  <EM>Knowledge Acquisition</EM>, <B>5</B>(2):199-200,
1993.  <A HREF="../papers/README.html#ontolingua-intro">Available on line.</A>
 <P>
</DL>
 <P>
<I>See also the WWW-resident</I>
<UL>
<LI>
<a href="http://www-ksl.stanford.edu/kst/what-is-an-ontology.html">What is an ontology?</A>
<LI>
<A HREF="../ontolingua/">All About Ontolingua</A> with pointers to documentation and  software.
```

```
<LI>
<A HREF="../kif/">All About KIF</A>

<LI> <A HREF="../index.html">[up] The Knowledge Sharing Effort Library</A>
</UL>

<HR>
<ADDRESS>
<A HREF="http://www-ksl.stanford.edu/people/gruber/">Tom Gruber</A>
</ADDRESS>
 <P>
```

Ende des Unterdokuments

DIMENSIONS OF THE ONTOLOGICAL ANALYSIS

Lecture 1. Introductory notes. Ontology in philosophy and in the theory of basis of data. An Aristotelian problem: Categories vs Metaphysics. Esternal or classificatory categories

Lecture 2. Internal categories. The layered structure of reality. Layers, levels and echelons. Dependences of the bearer/borne type vs dependences of the carrier/carried type. Laws of dependence among layers. Laws of authonomy among layers

Lecture 3. Overall architectonic and an example of ontological categorization. General, Regional, Domain and Application Ontology. The case of artifacts

Lecture 4. An overview and a comparison with other ontological projects. Cyc. Generalized Upper Model. Kosmos. Kactus

****

Roberto Poli
Department of Sociology and Social Research
26, Verdi street
38100 Trento -- Italy
Tel. ++39-461-881-403
Fax: ++39-461-881-348
e-mail: poli@risc1.gelso.unitn.it
home-page: http://www.gelso.unitn.it/~poli/

Ende des Unterdokuments

From newshost.uni-koblenz.de!xlink.net!howland.reston.ans.net!pipex!sunic!sics.se!torkel Thu Nov
Article: 8126 of comp.lang.prolog
Newsgroups: comp.lang.prolog
Path: newshost.uni-koblenz.de!xlink.net!howland.reston.ans.net!pipex!sunic!sics.se!torkel
From: torkel@sics.se (Torkel Franzen)
Subject: Re: FOL and Horn Clauses
In-Reply-To: govin-k@cs.buffalo.edu's message of Wed, 17 Nov 1993 22:17:19 GMT
Message-ID: <TORKEL.93Nov18011656@anhur.sics.se>
Lines: 18
Sender: news@sics.se
Organization: Swedish Institute of Computer Science, Kista
References: <CGnpwy.5A5@acsu.buffalo.edu>
Date: Thu, 18 Nov 1993 00:16:58 GMT

In article <CGnpwy.5A5@acsu.buffalo.edu> govin-k@cs.buffalo.edu
 (Kannan Govindarajan) writes:

   >In other words, can every first order formula be
   >equivalently written as a conjunction of Horn Clauses?

  First note that not every formula is equivalent to a conjunction of
clauses. (The Skolemization of A is in general strictly stronger than
A.)

  The clause p V q is not equivalent to any conjunction of Horn clauses.

  Whether a clause is equivalent to a conjunction of conjunction of clauses is
recursively undecidable.

  A formula A is equivalent to a conjunction of Horn clauses iff every
substructure of a model of A is a model of A and every direct product
of models of A is a model of A.

From newshost.uni-koblenz.de!xlink.net!rz.uni-karlsruhe.de!news.uni-stuttgart.de!news.belwue.de!
Article: 8127 of comp.lang.prolog
Path: newshost.uni-koblenz.de!xlink.net!rz.uni-karlsruhe.de!news.uni-stuttgart.de!news.belwue.de
From: lhe@kant.logik.informatik.uni-tuebingen.de (Lars-Henrik Eriksson)
Newsgroups: comp.lang.prolog
Subject: Re: FOL and Horn Clauses

Date: 18 Nov 1993 07:58:42 GMT
Organization: Lehrstuhl fuer Technische Inforrmatik, Uni Tuebingen
Lines: 31
Message-ID: <LHE.93Nov18085842@kant.logik.informatik.uni-tuebingen.de>
References: <CGnpwy.5A5@acsu.buffalo.edu>
NNTP-Posting-Host: kant.informatik.uni-tuebingen.de
In-reply-to: govin-k@cs.buffalo.edu's message of Wed, 17 Nov 1993 22:17:19 GMT

In article <CGnpwy.5A5@acsu.buffalo.edu> govin-k@cs.buffalo.edu (Kannan Govindarajan) writes:
   I have a very basic query about first order logic and horn clauses.

   It is well known that one can write every first order formula in
   prenex normal form (the "matrix" has no quantifiers). Further
   every existential quantifier can be eliminated by appropriate skolemization.

To elaborate on Torkel's reply:

You can eliminate existential quantifiers by skolemization, but the
skolemized formula will not be equivalent to an unskolemized one. The
relationship is that skolemized(P) implies P, but generally not the
other way around. To see this, consider the simple formula

(1)   some x p(x)

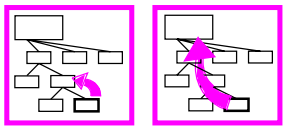
After skolemization this becomes

(2)   p(a)

where a is a Skolem constant. Clearly (2) implies (1), but not the
other way around.

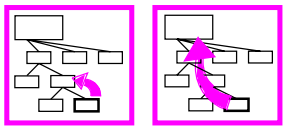
For use with a resolution proof procedure, such as the Prolog
execution mechanism, skolemizing input clauses is sound anyway, but in
other cases it might not be.
--
Lars-Henrik Eriksson, Wilhelm-Schickard-Institut, Tuebingen University
On leave from the Swedish Institute of Computer Science until Dec. 8, 1993.
Internet: lhe@logik.informatik.uni-tuebingen.de   Phone: +49 7071 294284
At SICS: lhe@sics.se   Phone: +46 8 752 15 09

Ende des Unterdokuments



Einfacher Fall:

```
all dogs are mammals

formula in predicate logic notation:
     all(D):(dog(D)->mammal(D))
implications replaced:
     all(D):(~dog(D)#mammal(D))
negations moved in:
     all(D):(~dog(D)#mammal(D))
existentials skolemized:
     all(D):(~dog(D)#mammal(D))
universals removed:
     ~dog(D)#mammal(D)
conjunctive normal form:
     ~dog(D)#mammal(D)
clausal form:
     cl([mammal(D)],[dog(D)])

mammal(D) :- dog(D).
```



Einige notationelle Vereinfachungen
[Ableitungen im Einzelnen]
(0.L.3)

Etwas schwieriger:

```
there are some candies all children like

formula in predicate logic notation:
     exists(C):(candy(C)&all(CH):(child(CH)->like(CH,C)))
implications replaced:
     exists(C):(candy(C)&all(CH):(~child(CH)#like(CH,C)))
negations moved in:
     exists(C):(candy(C)&all(CH):(~child(CH)#like(CH,C)))
existentials skolemized:
     candy(sk1)&all(CH):(~child(CH)#like(CH,sk1))
universals removed:
     candy(sk1)& ~child(CH)#like(CH,sk1)
conjunctive normal form:
     candy(sk1)& ~child(CH)#like(CH,sk1)
clausal form:
     cl([candy(sk1)],[])
     cl([like(CH,sk1)],[child(CH)])

candy(sk1).
like(CH,sk1) :- child(CH).
```

Noch etwas komplizierter (wegen des in einen Allquantor eingebettetetn Existenzquantors):

```
every man loves a woman

formula in predicate logic notation:
     all(M):(man(M)->exists(W):(woman(W)&loves(M,W)))
implications replaced:
     all(M):(~man(M)#exists(W):(woman(W)&loves(M,W)))
negations moved in:
     all(M):(~man(M)#exists(W):(woman(W)&loves(M,W)))
existentials skolemized:
     all(M):(~man(M)#woman(sk6(M))&loves(M,sk6(M)))
universals removed:
     ~man(M)#woman(sk6(M))&loves(M,sk6(M))
conjunctive normal form:
     (~man(M)#woman(sk6(M)))& ~man(M)#loves(M,sk6(M))
clausal form:
     cl([woman(sk6(M))],[man(M)])
     cl([loves(M,sk6(M))],[man(M)])

woman(sk6(M)) :- man(M).
loves(M,sk6(M)) :- man(M).
```

Hier nun eine Skolem-*Funktion*.

Nicht in HCL überführbar:

```
a supplier whose supplies are on time is preferred

formula in predicate logic notation:
     all(S):(supplier(S)&all(P):
        (supplies(S,P)->arriveontime(P))->preferred(S))
implications replaced:
     all(S):(~ (supplier(S)&all(P):
        (~supplies(S,P)#arriveontime(P)))#preferred(S))
negations moved in:
     all(S):((~supplier(S)#exists(P):
        (supplies(S,P)& ~arriveontime(P)))#preferred(S))
existentials skolemized:
     all(S):((~supplier(S)#supplies(S,sk5(S))&
        ~arriveontime(sk5(S)))#preferred(S))
universals removed:
     (~supplier(S)#supplies(S,sk5(S))&
        ~arriveontime(sk5(S)))#preferred(S)
conjunctive normal form:
     ((~supplier(S)#supplies(S,sk5(S)))#preferred(S))&
        (~supplier(S)# ~arriveontime(sk5(S)))#preferred(S)
clausal form:
     cl([supplies(S,sk5(S)),preferred(S)],[supplier(S)])
     cl([preferred(S)],[supplier(S),arriveontime(sk5(S))])

supplies(S,sk5(S)); preferred(S) :- supplier(S).
preferred(S) :- supplier(S), arriveontime(sk5(S)).
```

Ende des Unterdokuments

From newshost.uni-koblenz.de!xlink.net!sol.ctr.columbia.edu!howland.reston.ans.net!pipex!sunic!s
Article: 8860 of comp.lang.prolog
Newsgroups: comp.lang.prolog
Path: newshost.uni-koblenz.de!xlink.net!sol.ctr.columbia.edu!howland.reston.ans.net!pipex!sunic!
From: lhe@sics.se (Lars-Henrik Eriksson)
Subject: Re: Warren's OLDT, some questions
In-Reply-To: atwoodj@storm.cs.orst.edu's message of 18 Feb 1994 12:56:50 GMT
Message-ID: <LHE.94Feb19205614@anhur.sics.se>
Lines: 56
Sender: news@sics.se
Organization: Swedish Institute of Computer Science, Kista
References: <2k2duiINNq8a@flop.ENGR.ORST.EDU>
Date: Sat, 19 Feb 1994 19:56:16 GMT

In article <2k2duiINNq8a@flop.ENGR.ORST.EDU> atwoodj@storm.cs.orst.edu (John Atwood) writes:

> 3) I confused by the seeming interchangeable use of 'refutation'
> and 'resolution'. Is the former a logic term while the
> latter is a computing term?

Both are logic (or perhaps rather theorem proving) terms. "Refutation"
means proving that a formula P is a logical consequence of another
(set of) formula(e) Q, by showing that ~P together with Q is
inconsistent, i.e. implies absurdity. This works if Q itself is known
not to be inconsistent. That is the case e.g. if Q is a set of Horn
clauses.

"Resolution" is a logical inference step, i.e. a rule that can be used to
infer facts from known facts. If we ignore logical variables, resolution
says that from two formulae of the forms:

R or P1 or P2 or ... or Pnand
~R or Q1 or Q2 or ... or Qmwe can infer
P1 or P2 or ... or Pn or Q1 or Q2 or ... or Qm.

If n=0 and m=0, the inferred formula is empty. This is another way of
representing absurdity (since absurdity is neutral with respect to
disjunction).

>From a logic programming point of view, an important special case of

resolution is where R is an atom and all Pi and Qi are negated atoms.
In that case, the first formula can be rewritten as (B1&B2&...&Bn)->
R, (where ~Bi=Pi) which is recognisable as the Prolog *clause*
R:-B1,B2,...,Bn written with logical symbols. The second formula can
be written ~(Q1&Q2&...&Qm) - this corresponds to the Prolog *goal*
(Q1,Q2,...,Qm).

In Prolog, each resolution corresponds to a predicate call.

So if we have the program:

```
a :- b,c.    logically equivalent to (b&c)->a as well as
             to (a or ~b or ~c).
b.
c.
```

And give Prolog the goal formula a, we can show that a is a logical
consequence of the program by a refutation proof - showing that the we
can derive absurdity from the set of formulae ~a, b, c and (a or ~b or
~c)

First resolution: ~a and (a or ~b or ~c) gives (~b or ~c).
Second resolution: (~b or ~c) and b gives ~c.
Third resolution: ~c and c, gives an empty disjunction, which is another
way of writing absurdity
--

Lars-Henrik Eriksson
Swedish Institute of Computer Science
Box 1263
S-164 28 KISTA, SWEDEN

Internet: lhe@sics.se
Phone (intn'l): +46 8 752 15 09
Telefon (nat'l): 08 - 752 15 09
Fax: +46 8 751 72 30

Ende des Unterdokuments

Der bekannte Syllogismus (Inferenzregel) des *Modus Ponens*

```
p    →    q
p
_____________
q
```

hat den Nachteil, dass man nur ‘‘Fakten’’ (wie eben q) ableiten kann.

Besser ist daher die Inferenzregel der *Resolution* (die man als Verallgemeinerung des Modus Ponens betrachten kann), die man (in äquivalenter) Weise darstellen kann (vorläufig für den propositionalen Fall) als

```
   P ∨ Q                          ¬ P → Q
¬ Q ∨ S                           Q → S
_____________      oder       _____________
P ∨ S                           ¬ P → S
```

wo man auch *Regeln* ableiten kann (und natürlich, *a fortiori*, auch Fakten, da solche ja als Regeln mit ‘‘true’’ als Antezedens aufgefasst werden können).

Diesen Syllogismus kann man mittels verschiedener *Abarbeitungstrategien* benützen.

1. Die einfachste davon ist die Vorwärts- oder Rückwärtsverkettung, also z.B. (hier: Rückwärtsverkettung):

   ```
   ?- a.
   ```

   über einer Datenbank

   ```
   a ∨ b.
   ¬ b.
   ```

   in einem einzigen Schritt

```
?- a.
```

```
    ?- a ∨ b
    ?- ¬ b
```

```
yes
```

```
a=P
S=[]
b=Q
```

Aber: Resolution via Verkettung ist leider *nicht vollständig*. So kann man z.B. *nicht* ableiten

```
?- a ∨ ¬ a.
```

über einer leeren Datenbank

```
[ ]
```

obwohl das Theorem unzweifelhaft wahr ist.

2. Deshalb verwendet man die Abarbeitungstrategie der *Refutation:* Um `P` zu beweisen über ein Datenbank `DB`,
   a. fügen wir das *negierte* Theorem der Datenbank hinzu
   b. und beweisen, dass daraus ein *Widerspruch* folgt
   c. und schliessen daöraus, dass das *unnegierte* Theorem *ableitbar* ist.

also:

```
(DB ∧ ¬P  False) ↔ (DB  P)
```

Diese Strategie ist etwas aufwendiger, als Verkettung, aber dafür vollständig.

Genaueres dazu: Russell 1995:276 ff.

Schliesslich müssen wir die Resolution noch auf den prädikatenlogischen Fall erweitern, wo nicht mehr atomare Symbole vollständig ersetzt werden können. Dazu wird bekanntlich das Konzept der *Unifikation* verwendet (siehe hierzu auch Russell 1995:270).

Hintergrundinformation (0.L.4.L.1)

Ende des Unterdokuments

```
From newshost.uni-koblenz.de!xlink.net!rz.uni-karlsruhe.de!news.uni-ulm.de!news.belwue.de!iptc!r
Article: 8104 of comp.lang.prolog
Path: newshost.uni-koblenz.de!xlink.net!rz.uni-karlsruhe.de!news.uni-ulm.de!news.belwue.de!iptc!
From: lhe@kant.logik.informatik.uni-tuebingen.de (Lars-Henrik Eriksson)
Newsgroups: comp.lang.prolog
Subject: Re: imply(A,B)
Date: 16 Nov 1993 11:15:21 GMT
Organization: Lehrstuhl fuer Technische Inforrmatik, Uni Tuebingen
Lines: 38
Message-ID: <LHE.93Nov16121522@kant.logik.informatik.uni-tuebingen.de>
References: <zeppelin.753407516@login.dkuug.dk>
NNTP-Posting-Host: kant.informatik.uni-tuebingen.de
In-reply-to: zeppelin@login.dkuug.dk's message of Mon, 15 Nov 1993 23:51:56 GMT

In article <zeppelin.753407516@login.dkuug.dk> zeppelin@login.dkuug.dk (Thomas B. Pedersen) writ

   How would you write a function 'imply(A,B)' in Prolog which implements
   the logical implication A ==> B?

That depends on what you want to achieve, *operationally*.

If we ignore the issue of logical variables in A, there are two obvious ways:

1) imply(A,B) :- B.
   imply(A,B) :- + A.

2) imply(A,B) :- assert(A),
                 B,
                 (retract(A); assert(A), fail).
   imply(A,B) :- retract(A), fail.

The first case uses the logical equivalence A->B == not A or B. This is
easiest, but likely not what one would usually want.

The second case exploits the deduction theorem, i.e. the fact that if
B can be derived, assuming A, then it is correct to derive A->B. I
guess this is what one would want in most cases. This is also the
implication implemented in the hereditary Harrop-formula family of
logic programming language (e.g. Lambda-prolog). The code is rather
tricky, since you must make sure that it always does the right thing
```

on backtracking. Also, my example only works when A is an atomic goal.

If logical variables are permitted in A, both cases become
considerably more complex. In the first case, sound negation must be
used instead of +. In the second case, I don't know if it is even
possible to implement correctly in Prolog.

--
Lars-Henrik Eriksson, Wilhelm-Schickard-Institut, Tuebingen University
On leave from the Swedish Institute of Computer Science until Dec. 8, 1993.
Internet: lhe@logik.informatik.uni-tuebingen.de Phone: +49 7071 294284
At SICS: lhe@sics.se Phone: +46 8 752 15 09

From newshost.uni-koblenz.de!xlink.net!howland.reston.ans.net!pipex!uknet!warwick!zaphod.crihan.
Article: 8108 of comp.lang.prolog
Path: newshost.uni-koblenz.de!xlink.net!howland.reston.ans.net!pipex!uknet!warwick!zaphod.crihar
From: devienne@lifl.fr (devienne_philippe)
Newsgroups: comp.lang.prolog
Subject: Re: imply(A,B)
Date: 16 Nov 1993 13:52:15 GMT
Organization: Laboratoire d'Informatique Fondamentale de Lille - France
Lines: 11
Distribution: world
Message-ID: <2caluf$68r@netserver.univ-lille1.fr>
References: <zeppelin.753407516@login.dkuug.dk>
Reply-To: devienne@lifl.fr
NNTP-Posting-Host: budweiser.lifl.fr

Indeed, such a property is almost undecidable. The only decidable case that I know
supposes that the Horn Clause A is binary (that is whose body contains only one
literal) [Schmidt-Schauss in TCS journal , Vol 59, 1988].
As soon as the Horn clause A is ternary (whose body has two litterals),
the set of implied Horn clauses is recursively enumerable (see Pacholsky and Marcinkowsky in FOC

All the best.
Philippe DEVIENNE.

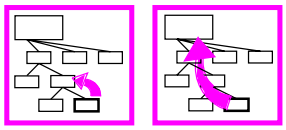
```
From newshost.uni-koblenz.de!xlink.net!howland.reston.ans.net!wupost!gumby!destroyer!nntp.cs.ubc
Article: 8124 of comp.lang.prolog
Newsgroups: comp.lang.prolog
Path: newshost.uni-koblenz.de!xlink.net!howland.reston.ans.net!wupost!gumby!destroyer!nntp.cs.ub
From: tarau@cs.sfu.ca (Paul Tarau)
Subject: Re: imply(A,B) again
Message-ID: <1993Nov17.210009.27635@cs.sfu.ca>
Organization: CSS, Simon Fraser University, Burnaby, B.C., Canada
References: <zeppelin.753534369@login.dkuug.dk>
Date: Wed, 17 Nov 1993 21:00:09 GMT
Lines: 131

In article <zeppelin.753534369@login.dkuug.dk> zeppelin@login.dkuug.dk (Thomas B. Pedersen) writ
>
>Given the clauses
>
>    person(john).
>    person(peter).
>    person(anna).
>    male(john).
>    male(peter).
>    female(anna).
>
>is it then possible to implement the logical implication A -> B, so that the
>following means: is every person a male? (i.e., returning yes or no).
>
>    imply(person(X), male(X))          (or something like that)
>


/*
It can be done reasonably with a not-too-fussy-but-sound negation
as failure, at least in the case of your example:
*/

imply(X,Y):- negation((X, negation(Y))).

negation(X):- + X,!.
negation(X):-ground(X),!,fail.
```

```
negation(X):-write(ground_expected_in_negation_of(X)),nl,error.

    person(peter).
    person(john).
    person(anna).
    male(peter).
    male(john).
    female(anna).

/*
Note that groundness is not necessarily required for negation/1
(i.e. Goals are presumed innocent unless proven guilty :-) ).
The usual implementation of sound non-delayed negation
(as it can be found for instance in library(not) in Quintus Prolog)
seems too restrictive because it requires groundness too soon.
*/

fussy_negation(X):-ground(X),!,+ X.
fussy_negation(X):-write(ground_expected_in_fussy_negation_of(X)),nl,error.

fussy_imply(X,Y):- fussy_negation((X, fussy_negation(Y))).

error:-abort.

% You can try out the differences in behaviour with the following:

test(1,imply(male(X),person(X))).
test(2,imply(female(X),person(X))).
test(3,imply(female(X),male(X))).
test(4,imply(male(X),male(X))).

test(5,fussy_imply(male(X),person(X))).
test(6,fussy_imply(female(X),person(X))).
test(7,fussy_imply(female(X),male(X))).
test(8,fussy_imply(male(X),male(X))).

go(N):-test(N,X),nl,write([N]),write(X),write(('=>')),
    (X->write(true);write(false)),nl.
```

```
/*
The output (in Aquarius Prolog version 1.0 top-level (SPARC, SunOS)) is:

| ?- go(1).

[1]imply(male(_480),person(_480))=>true

yes
| ?- go(2).

[2]imply(female(_480),person(_480))=>true

yes
| ?- go(3).

[3]imply(female(_480),male(_480))=>ground_expected_in_negation_of((female(_480),negation(male(_4

| ?- go(4).

[4]imply(male(_480),male(_480))=>true

yes

| ?- go(5).

[5]fussy_imply(male(_480),person(_480))=>ground_expected_in_fussy_negation_of
((male(_480),fussy_negation(person(_480))))

| ?- go(6).

[6]fussy_imply(female(_480),person(_480))=>ground_expected_in_fussy_negation_of
((female(_480),fussy_negation(person(_480))))

| ?- go(7).

[7]fussy_imply(female(_480),male(_480))=>ground_expected_in_fussy_negation_of
((female(_480),fussy_negation(male(_480))))

| ?- go(8).
```

```
[8]fussy_imply(male(_480),male(_480))=>ground_expected_in_fussy_negation_of
((male(_480),fussy_negation(male(_480))))

Clearly, the behaviour of negation/1 is close to what we
really expect it to do.

My intuition is that negation/1 is nice basically because if
negation(G) succeeds (i.e. if G fails), instantiating later a variable in
G cannot make G succeed and therefore change the outcome of negation(G).

On the other hand, if negation(G) fails it may happen that instantiating
its variables later would change the outcome so this is an error
(or delay) condition.

I know a few papers/books written about fussy_negation/1 but no one
about negation/1.  Is there a deep reason for that?
Did someone use/implement it in Prolog?

Note that negation/1 is not only less fussy but also
faster because it avoids the expensive call to ground/1
when possible.

Paul Tarau
tarau@cs.sfu.ca
*/
```

Ende des Unterdokuments



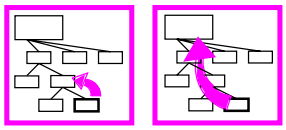
[Die Definition der Operatoren]

(0.L.6)

Die Definition der Operatoren in Sicstus-Prolog:

```
:-op( 300,xfx,:).
:-op( 450,fx,~).
:-op( 800,xfy,#).
:-op( 800,xfy,&).
:-op( 850,xfy,->).
:-op( 850,xfy,<->).
```

Ende des Unterdokuments



Kommentar zum Programm
[Einige einfache Traces]
(0.L.7)

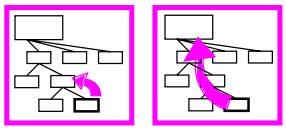
Zur Illustration für den Informationsfluss beim Abarbeiten des Programms folgen hier einige einfache Traces.

```
1: every department employs a tutor

  (5) 1 Call: sentence(P0,[every,department,employs,a,tutor],[]) ?
   (6) 2 Call: noun_phrase(X0,P1,P0,
    [every,department,employs,a,tutor],_) ?
   (7) 3 Call: determiner(X0,P2,P1,P0,
    [every,department,employs,a,tutor],_) ? s
 > (7) 3 Exit: determiner(X0,P2,P1,all(X0): (P2->P1),
    [every,department,employs,a,tutor],[department,employs,a,tutor]) ?
   (9) 3 Call: noun(X0,S,
    [department,employs,a,tutor],_) ? s
 > (9) 3 Exit: noun(X0,department(X0),
    [department,employs,a,tutor], [employs,a,tutor]) ?
   (12) 3 Call: rel_clause(X0,department(X0),P2,
    [employs,a,tutor],_) ? s
 > (12) 3 Exit: rel_clause(X0,department(X0),department(X0),
    [employs,a,tutor], [employs,a,tutor]) ?
   (6) 2 Exit: noun_phrase(X0,P1,all(X0): (department(X0)->P1),
    [every,department,employs,a,tutor],[employs,a,tutor]) ?
   (14) 2 Call: verb_phrase(X0,P1,
    [employs,a,tutor],[]) ?
   (15) 3 Call: trans_verb(X0,Y0,S1,
    [employs,a,tutor],_) ? s
 > (15) 3 Exit: trans_verb(X0,Y0,employs(X0,Y0),
    [employs,a,tutor],[a,tutor]) ?
   (17) 3 Call: noun_phrase(Y0,employs(X0,Y0),P1,
    [a,tutor],[]) ?
   (18) 4 Call: determiner(Y0,P4,employs(X0,Y0),P1,
    [a,tutor],_) ? s
 > (18) 4 Exit: determiner(Y0,P4,employs(X0,Y0),
                        exists(Y0): (P4&employs(X0,Y0)),
    [a,tutor],[tutor]) ?
   (22) 4 Call: noun(Y0,S3,[tutor],_) ? s
 > (22) 4 Exit: noun(Y0,tutor(Y0),[tutor],[]) ?
   (26) 4 Call: rel_clause(Y0,tutor(Y0),P4,[],[]) ? s
 > (26) 4 Exit: rel_clause(Y0,tutor(Y0),tutor(Y0),[],[]) ?
```

```
   (17) 3 Exit: noun_phrase(Y0,employs(X0,Y0),
                    exists(Y0): (tutor(Y0)&employs(X0,Y0)),
    [a,tutor],[]) ?
   (14) 2 Exit: verb_phrase(X0,exists(Y0): (tutor(Y0)&employs(X0,Y0)),
    [employs,a,tutor],[]) ?
   (5) 1 Exit: sentence(all(X0): (department(X0)->
                    exists(Y0): (tutor(Y0)&employs(X0,Y0))),
    [every,department,employs,a,tutor],[])


2: every department that prospers employs a tutor

  (5) 1 Call: sentence(P0,[...],[]) ?
   (6) 2 Call: noun_phrase(?,_272,P0,[...],_275) ?
   (7) 3 Call: determiner(?,_316,_272,P0,[...],_320) ? s
 > (7) 3 Exit: determiner(?,_316,_272,all(?): (_316->_272),[...],[...]) ?
   (9) 3 Call: noun(?,_329,[...],_331) ? s
 > (9) 3 Exit: noun(?,department(?),[...],[...]) ?
   (12) 3 Call: rel_clause(?,department(?),_316,[...],_275) ? s
 > (12) 3 Exit: rel_clause(?,department(?),
                    department(?)&prospers(?),[...],[...]) ?
   (6) 2 Exit: noun_phrase(?,_272,all(?):
                                        (department(?)&
                                         prospers(?)->_272),[...],[...]) ?
   (19) 2 Call: verb_phrase(?,_272,[...],[...]) ?
   (20) 3 Call: trans_verb(?,_570,_571,[...],_573) ? s
 > (20) 3 Exit: trans_verb(?,_570,employs(?,_570),[...],[...]) ?
   (22) 3 Call: noun_phrase(_570,employs(?,_570),_272,[...],[...]) ?
   (23) 4 Call: determiner(_570,_651,employs(?,_570),_272,[...],_655) ? s
 > (23) 4 Exit: determiner(_570,_651,employs(?,_570),
                    exists(_570): (_651&employs(?,_570)),[...],[...]) ?
   (27) 4 Call: noun(_570,_664,[...],_666) ? s
 > (27) 4 Exit: noun(_570,tutor(_570),[...],[...]) ?
   (31) 4 Call: rel_clause(_570,tutor(_570),_651,[...],[...]) ? s
 > (31) 4 Exit: rel_clause(_570,tutor(_570),tutor(_570),[...],[...]) ?
   (22) 3 Exit: noun_phrase(_570,employs(?,_570),
                    exists(_570): (tutor(_570)&
                    employs(?,_570)),[...],[...]) ?
   (19) 2 Exit: verb_phrase(?,exists(_570):
```

```
                                        (tutor(_570)&
                                         employs(?,_570)),[...],[...]) ?
   (5) 1 Exit: sentence(all(?):
                            (department(?)&prospers(?)->
                            exists(_570): (tutor(_570)&
                            employs(?,_570))),[...],[...]) ?


3: every department employs a tutor that employs a student

   (5) 1 Call: sentence(P0,[...],[...]) ?
   (6) 2 Call: noun_phrase(_275,_276,P0,[...],_279) ? s
 > (6) 2 Exit: noun_phrase(_275,_276,all(_275):
        (department(_275)->_276),[...],[...]) ?
   (14) 2 Call: verb_phrase(_275,_276,[...],[...]) ?
   (15) 3 Call: trans_verb(_275,_486,_487,[...],_489) ? s
 > (15) 3 Exit: trans_verb(_275,_486,employs(_275,_486),[...],[...]) ?
   (17) 3 Call: noun_phrase(_486,employs(_275,_486),_276,[...],[...]) ? s
 > (17) 3 Exit: noun_phrase(_486,employs(_275,_486),
        exists(_486): ((tutor(_486)&exists(_756): (student(_756)&
            employs(_486,_756)))&employs(_275,_486)),[...],[...]) ?
   (14) 2 Exit: verb_phrase(_275,
        exists(_486): ((tutor(_486)&exists(_756): (student(_756)&
            employs(_486,_756)))&employs(_275,_486)),[...],[...]) ?
   (5) 1 Exit: sentence(all(_275): (department(_275)->
                            exists(_486): ((tutor(_486)&
                                exists(_756): (student(_756)&
                                    employs(_486,_756)))&
                                    employs(_275,_486))),[...],[])
```

Ende des Unterdokuments



Kommentar zum Programm

[Eine Erweiterung der naiven direkten Übersetzung]

(0.L.8)

Ziel nunmehr also: Objektmodifizierende Relativsätze wie in

Every department that employs a student *that a tutor pays* prospers

Dieser Satz sollte folgende logische Repräsentation erzeugen:

```
all(D): (department(D) & exists(S): (student(S) &
    employs(D,S) & exists(T): (tutor(T) & pays(T,S)))
    -> prospers(D))
```

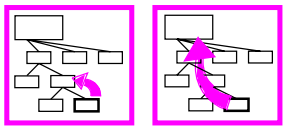
Und natürlich möchte man auch einen Satz wie den folgenden korrekt übersetzt erhalten:

Every department that employs a student that *it* pays prospers

```
all(D): (department(D) ∧ exists(S): (student(S)
    ∧ employs(D,S) ∧ pays(D,S)) -> prospers(D))
```

Diese vermeintlich harmlosen Modifikationen (die syntaktisch nicht besonders schwer zu erfassen ist), führen beim gewählten ‘‘naiven’’ Ansatz (direkte Übersetzung) zu ganz unverhältnismässigen Komplikationen in der semantischen Komponente des Programms.

Wie müsste das Programm modifiziert werden, dass es auch diese Sätze verdauen kann? Hier ist das modifizierte Programm:



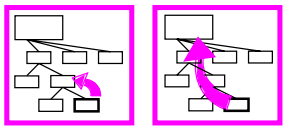
Kommentar zum Programm

[Eine Erweiterung der naiven direkten Übersetzung]

(0.L.8)


```
sentence(P) --> noun_phrase(W,X,P1,P), verb_phrase(X,P1).

noun_phrase(X,X,P,P) --> [he];[she];[it].
noun_phrase(_,X,P1,P) -->
   determiner(X,P2,P1,P), noun(X,P3), rel_clause(X,P3,P2).
noun_phrase(_,X,P,P) --> name(X).

verb_phrase(X,P) --> intrans_verb(X,P).

verb_phrase(X,P2) --> trans_verb(X,Y,P1),
                      noun_phrase(X,Y,P1&P3,P2),
                      obj_rel_clause(X,Y,P2,P3).

obj_rel_clause(W,X,P1,P2) --> [that],
                              noun_phrase(W,Y,P3,P2),
                              trans_verb(Y,X,P3).
obj_rel_clause(_,_,_,true) --> [].

rel_clause(X,P1,P1&P2) --> [that], verb_phrase(X,P2).
rel_clause(_,P,P) --> [].

determiner(X,P1,P2, all(X): (P1->P2) ) --> [every];[all].
determiner(X,P1,P2, exists(X): (P1&P2)) --> [a];[some].

noun(X, department(X)) --> [department].
noun(X, tutor(X)) --> [tutor].
noun(X, student(X)) --> [student].

name(john) --> [john].

trans_verb(X,Y, pays(X,Y)) --> [pays].
trans_verb(X,Y, employs(X,Y)) --> [employs].
intrans_verb(X, prospers(X)) --> [prospers].
```

Kommentar zum Programm:




```
sentence(P) --> noun_phrase(W,X,P1,P), verb_phrase(X,P1).

noun_phrase(X,X,P,P) --> [he];[she];[it].
% das liesse sich, was die Syntax angeht,  eleganter machen;
% zur Rolle des ersten "X" siehe unten
noun_phrase(_,X,P1,P) -->
   determiner(X,P2,P1,P), noun(X,P3), rel_clause(X,P3,P2).
noun_phrase(_,X,P,P) --> name(X).

verb_phrase(X,P) --> intrans_verb(X,P).

verb_phrase(X,P2) --> trans_verb(X,Y,P1), noun_phrase(X,Y,P1&P3,P2),
                obj_rel_clause(X,Y,P2,P3).
```

Das `X` in der Nominalphrase ist allein deshalb hier, um dann, wenn die in der in `obj_rel_clause` enthaltene Nominalphrase ein Personalpronomen ist, den Wert der Variablen dort nicht lokal zu erzeugen, sondern von der diese Regel hier dominierenden Nominalphrase (via Verbalphrase!) zu übernehmen (in der Pronomen-Regel X=X); dann wird dieser Wert an `obj_rel_clause` weitergeboten:

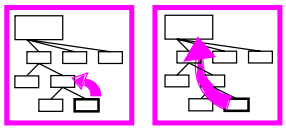
```
obj_rel_clause(W,X,P1,P2) --> [that], noun_phrase(W,Y,P3,P2),
                                      trans_verb(Y,X,P3).
```

Hier enthält das `W` den Wert der Variablen der zwei (!) Ebenen höher gelegenen Nominalphrase für den Fall dass diese Nominalphrase hier ein Personalpronomen ist und deshalb keine eigenen Variable erzeugen darf

```
obj_rel_clause(_,_,_,true) --> [].
```

das ‘‘true’’ könnte vermieden werden, indem man die Vp-Regel modifiziert für den Fall, dass keine obj_rel_clause vorliegt.

Ende des Unterdokuments



Zur Erinnerung: Kompositionalität

Dieses Prinzip, das oft ‘‘Freges Prinzip’’ genannt wird, obwohl er es selbst nie ganz explizit formuliert hatte, verlangt, dass man die Semantik eines Satzes rein syntaxgesteuert errechnen können sollte. Das heisst insbesondere, dass man für jede wohlgeformte Komponente der Syntaxstruktur die entsprechende semantische Übersetzung errechnen kann, ohne auf den (noch nicht analysierten) Kontext der Struktur zu achten. Idiome sind ein gutes Beispiel für (prinzipiell) nicht kompositional analysierbare Konstruktionen. Idiome unterscheiden sich von ‘‘normalen’’ Konstruktionen dadurch, dass man ihre Bedeutung *nicht* aus der Bedeutung der Einzelteile errechnen kann. Ein Beispiel: Da die Bedeutung des Idioms ‘‘ins Gras beissen’’ nicht kompositional ermittelt werden kann, muss man das ganze Idiom wie ein neues Wort lernen. Das Prinzip der Kompositionalität fordert, dass *nur* Idiome in dieser Weise behandelt werden, alle ‘‘normalen’’ Konstruktionen aber rein syntaxgesteuert. Gerade das ist aber im vorliegenden Fall (in der beschriebenen Weise) nicht möglich: Eine Syntaxstruktur ‘‘S(NP(PN),VP)’’ und eine Struktur ‘‘S(NP(DET,N),VP)’’ würden, um beim obigen Beispiel zu bleiben, wie zwei verschiedene Wörter behandelt (man müsste sie im Lexikon nachschauen).

Ende des Unterdokuments