Vorlesung

Frühlingssemester 2009

Fortgeschrittene Konzepte der Logikprogrammierung

Universität Zürich

Institut für Computerlinguistik

Prof. Dr. Michael Hess

© Michael Hess 2009

1. Beschreibung der Veranstaltung

1. *Inhalt:* Diese Lehrveranstaltung entspricht inhaltlich vorerst exakt der früheren Lehrveranstaltung ‘‘Methoden der Künstlichen Intelligenz in der Sprachverarbeitung’’.
2. *Art:* Fortgeschrittene Lehrveranstaltung. Die Veranstaltung wird *nicht* als Präsenzveranstaltung abgehalten werden, sondern als E-Learning-gestützte Selbststudiums-Veranstaltung.
3. *Voraussetzungen:*
    1. Kenntnisse in Computerlinguistik mind. auf Niveau Akzessprüfung
    2. Gute Kenntnisse von Prolog
    3. Vorteilhaft: Gute Informatikkenntnisse
4. *Ziel:*
    a. Erklärung einiger zentraler Probleme der Künstlichen Intelligenz und einige Lösungsmethoden
    b. Erklärung der spezifischen Ausprägungen dieser Methoden für computerlinguistische Fragestellungen

© Michael Hess 2009

2. Ressourcen

2.1 Literatur

Monographien:

exzellent!

1. Peter Norvig, Stuart Russell: Artificial Intelligence - A Modern Approach. The Intelligent Agent Book. 2. Auflage International Edition. Prentice Hall. Januar 2003, ISBN 0-13-790395-2.

   Sehr umfassend; leitendes Beispiel: intelligente Agenten; z.Zt. wohl die beste Einführung in die Künstliche Intelligenz.

   Deutsche Übersetzung:

   Künstliche Intelligenz: Ein moderner Ansatz (2nd Edition). München, Verlag Pearson Studium, 2005, ISBN 3-8273-7089-2

   Mit eigener, sehr reichhaltiger, Web-Seite ⇒hier.

2. Matt Ginsberg Essentials of Artificial Intelligence, Morgan Kaufmann Publishers, 1993, San Mateo, CA

Prolog-spezifische Bücher und...

3. Neil C. Rowe, Artificial Intelligence through Prolog, Prentice Hall, New Jersey, 1988
4. Yoav Shoham, Artificial intelligence techniques in Prolog, San Mateo, Calif., Morgan Kaufmann Publishers, 1994

...solche mit Allgemeinheitsanspruch

5. Günther Görz (Hrsg.), Einführung in die künstliche Intelligenz, Bonn (etc.), Addison-Wesley, 1993, ISBN 3-89319-507-6

ab hier etwas alt, aber...

6. M.R. Genesereth, N.J. Nilsson, Logical Foundations of Artificial Intelligence, Morgan Kaufmann, Los Altos, CA, 1987
7. R. Frost, Introduction to Knowledge Base Systems, Collins, London, 1986

...eben immer noch gut.

8. B. Raphael, The Thinking Computer; Mind Inside Matter, Freeman, San Francisco, 1976

Zeitschriften und Konferenzberichte (0.L.1)

© Michael Hess 2009

2.2 Informationen auf dem Internet

Eine sehr nützliches *Glossar* zu Begriffen der Künstlichen Intelligenz ist
⇒‘‘The AI Dictionary’’

von Bill Wilson.

3. Was ist Künstliche Intelligenz?

Sinnvolle Definition sehr schwer zu formulieren. Eine Auswahl aus der Literatur:

Ist die Künstliche Intelligenz... ...eine ingenieurtechnische Disziplin oder...

- Rowe 1988:1:

  > *Artificial Intelligence* is the getting of computers to do things that seem to be intelligent

  Also ist die Künstliche Intelligenz eine *ingenieurtechnische* Disziplin.

...eine Wissenschaft oder...

- Charniak 1985:6:

  > Artificial Intelligence is the study of mental faculties through the use of computational models

  Das ist etwas ganz anderes: Der Computer ist Instrument zur Analyse der *menschlichen* Intelligenz. Also: Die Künstliche Intelligenz ist ein Hilfsmittel der *Psychologie*

...von beidem etwas?

- Genesereth 1987:1:

  > Artificial Intelligence (AI) is the study of intelligent behavior. Its ultimate goal is a theory of intelligence that accounts for the behavior of naturally occurring intelligent entities and that guides the creation of artificial entities capable of intelligent behavior. Thus, AI is both a branch of science and a branch of engineering

  Also eine Art *Kompromissvorschlag*. Umfasst auch den Bereich der ‘‘Kognitiven Wissenschaft’’.

© Michael Hess 2009

Exakte maschinelle Kopien von Menschen sind gar nicht sinnvoll.

Schliesslich: Oekonomisch ist die *exakte* Simulation menschlicher Intelligenz uninteressant.

Aber:

- Computer ermüden nicht,
- machen (fast) keine Fehler,
- vergessen nichts.

Die ‘‘lineare Verlängerung’’ der menschlichen Intelligenz ist es.

Sinnvolleres Ziel: Partielle lineare Verlängerung menschlicher intellektueller Fähigkeiten

Weiterhin unbeantwortet: Was ist Intelligenz?

Statt einer Definition:

3.1 Einige Typen von KI-Systemen

‘‘prototypisch’’ ≅ besonders eingängig

Einige *prototypische* Beispiele von Systemen mit künstlicher Intelligenz:

1. Spielprogramme
2. Roboter
3. Expertensysteme
4. Systeme für die Sprachverarbeitung

Dazu:

populär, aber etwas speziell

- **Spielprogramme:** (Schach und andere Spiele): Wohl die älteste Anwendung der Künstlichen Intelligenz überhaupt. Heute schon sehr gut. Kommerziell sehr erfolgreich.
- **Roboter:** In der populären Auffassung freibewegliche Maschinen mit quasimenschlicher Form; in der industriellen Realität meist ortsfeste Systeme. ‘‘Autonome Agenten’’ sind die moderne Inkarnation, v.a. im Kontext der sog. ‘‘situativen KI’’ so genannt, wo sie nicht praktisch einsetzbare Werkzeuge, sondern Instrumente zur Erkenntnisgewinnung sind. In diesem Sinne ist die Robotik das exakte Gegenteil der Entwicklung von Spielprogrammen.

© Michael Hess 2009

kommerziell erfolgreich

- **Expertensysteme:** Wohl die kommerziell erfolgreichste Anwendung der Künstlichen Intelligenz. Aber nicht sehr ''populär'', da eher wenig spektakulär.

historisch erste Anwendung der KI!

- **Sprachverarbeitung:** Maschinelle Übersetzung war historisch erste Anwendung der Künstlichen Intelligenz. Nunmehr langsam auch kommerziell interessant. Verarbeitung gesprochener Sprache wurde in den letzten Jahren ± plötzlich kommerziell sehr relevant.

Alle diese Systeme erfordern eine Kombination verschiedener ''Intelligenz''-Funktionen. Eine der wichtigsten Aufgaben der Künstlichen Intelligenz als *Wissenschaft* ist das Isolieren von grundlegenden Einzelfunktionen der natürlichen (u.a. menschlichen) Intelligenz. Allerdings kann man das nur durch das Aufteilen entlang verschiedener Dimensionen tun:

3.2 ''Alte'' und ''Neue'' Künstliche Intelligenz

Oft wird heute von ''Neuer AI'' gesprochen. Auch dieser Begriff ist schlecht definierbar. Man könnte sagen, dass sich die zwei ''Arten'' von Künstlicher Intelligenz so unterscheiden:

''Alte'' Künstliche Intelligenz: (GOFAI: Goold Old Faith AI)

Zwar ''alt'', aber...

- Intelligenz als weitgehend isolierbares Phänomen
- explizites Instruieren (Lehren)
- explizites Wissen in Regeln und Fakten
- symbolbasiert (d.h. mit einem endlichen Inventar von im Voraus festgelegten Zeichen mit definierter Bedeutung arbeitend)

...in der Computerlinguistik immer noch wichtiger als...

- ''digital'' ausgerichtet
- Prototypisches Beispiel: Syntaxanalyse natürlicher Sprache relativ zu einer formalen Grammatik

''Neue'' Künstliche Intelligenz:

...die ''neue''.

- Intelligenz als notwendigerweise ''situiertes'' Phänomen
- implizites Instruieren (Lernen lassen)
- implizites Wissen in den Werten neuronaler Netze (d.h. Netzwerken von Einheiten, von welchen jede aufgrund der aufsummierten Stärke eingehender Impulse einen Output erzeugt, sobald ein Schwellwert überschritten wird; dieser Output wird u.U. zum Input für weitere Elemente)

© Michael Hess 2009

- subsymbolisch orientiert (d.h. mit ''rohem'' sensorischem Input arbeitend, der durch das System in Äquivalenzklassen eingeteilt wird, welch letztere dann als Bedeutungen von u.U. erst noch festzulegenden Zeichen dient)
- ''analog'' ausgerichtet
- Prototypisches Beispiel: Erkennung optischer Muster in gestörter Umgebung

Die Ansätze sind komplementär.

Natürlich kann man die beiden Ansätze *kombinieren*.

Beispiele:

1. (gute) *Vorlesesysysteme* (ab Papier): Zuerst muss man die Buchstaben *erkennen*, dann erst kann man sie im Kontext *interpretieren* (d.h. das Wort korrekt vorlesen).[1]
2. *Wahrnehmungssysteme:* Um sich in der realen Welt z.B. optisch zurechtzufinden, muss man zuerst Objekte *erkennen* und danach in ihrer relativen Position *interpretieren* (ihre Anordnung im Raum und zueinander). Siehe gleich unten. [2]

Deshalb heute vermehrt Entwurf von sog. *hybriden* Systemen.

1. Was man im Englischen in der Regel unter ''text-to-speech systems'' versteht, sind Systeme, die ab maschinenlesbarem Text arbeiten. Im vorliegenden Fall muss man das optische Bild zuerst analysieren!
2. ''Perception systems''. Im optischen Bereich muss man unterscheiden zwischen reinen ''vision systems'' und ''scene analysis systems'' (oder ''Situationserkennung''). Siehe später.

© Michael Hess 2009

3.3 Komponenten der Intelligenz

Versuch eine Kategorisierung.

Intelligenz kann *ganz allgemein* in folgende Komponenten zerlegt werden (betrifft die ''alte'' *und* die ''neue'' Künstliche Intelligenz) sind:

1. Informations*gewinnung*
2. Informations*verdichtung*
3. Informations*transformation*

3.3.1 Informationsgewinnung

Wahrnehmung

Hierunter fällt v.a. das ''Interface'' zur physischen Realität, also die *Wahrnehmung* im allgemeinsten Sinn.

Zu unterscheiden sind:

Mustererkennung...

...in verschiedenen Modi:

1. **Mustererkennung** (''pattern recognition''): aus disparaten Daten zusammenhängende Muster gewinnen. Die Muster sind von vornherein bekannt und bilden ein endliches Inventar (Standardbeispiel: Buchstaben des Alphabets).

   Wichtig v.a.:

   optische Daten...
   ...und akustische...
   ...und symbolische.

   i. *optische* Daten verarbeiten, d.h. realweltliche Objekte erkennen [3]
   ii. *akustische* Daten verarbeiten, z.B. Spracherkennung[4]
   iii. *symbolische* (''informatische'') Daten verarbeiten

Situationserkennung.

Erneut: optischer und...

2. **Situationserkennung:** komplexe *Konfigurationen* von Elementen beschreiben
   i. optische Szenen erkennen [5]

---

3. Auf gut Deutsch heisst das ''vision''
4. Also ''speech recognition''
5. Eben ''scene recognition'' oder ''scene analysis''

© Michael Hess 2009

...symbolischer Modus.

ii. ‘‘symbolische Szenen’’ erkennen

3.3.2 Informationsverdichtung

Hierzu gehören Generalisieren und insbes. auch **maschinelles Lernen:**

Konzeptlernen und...

1. **Konzeptlernen:** disparate Daten klassifizieren und *neue* Konzepte daraus ableiten
   i. visuelle Konzepte erschliessen
   ii. akustische Konzepte erschliessen
   iii. symbolische Konzepte erschliessen

...Regellernen:

2. **Regellernen:** aus einzelnen Abfolgen von Ereignissen neue Gesetzmässigkeiten ableiten
   i. aus *Rohdaten* realweltliche (z.B. physikalische) Gesetzmässigkeiten erschliessen
   ii. aus (bewerteten) *Beispielen* Gesetzmässigkeiten erschliessen

Letzlich: Ziel aller Wissenschaft.

3.3.3 Informationstransformation

Hier u.a. das Explizitmachen impliziter Information:

Alle Randbedingungen klar

1. **Schlussfolgern:** aus bekannten Fakten Schlüsse ziehen; Faktenerkennung ist kein Problem; Schlussfolgerungsregeln sind bekannt

weniger klar

2. **Problemlösen:** ein definiertes Ziel erreichen, indem man bekannte Teillösungen situationsadäquat kombiniert; Faktenerkennung u.U. problematisch, Information u.U. unvollständig

und zudem dynamisch.

3. **Planen:** Sequenzen von (die Welt verändernden) Handlungen planen, um (u.U. mehrere) Ziele (in einer u.U. sich autonom verändernden Welt) zu erreichen:

Achtung: Diese Begriffe werden nicht überall genau so verwendet.

Die meisten Anwendungen setzen mehrere, aber kaum je alle, dieser drei Komponenten (Informationsgewinnung, Informationsverdichtung, Informationstransformation) ein. Von besonderer Bedeutung für uns ist die Informationstransformation, und hier insbes. das *Schlussfolgern*. Daher mehr dazu im folgenden.

© Michael Hess 2009

4. Inferenzverfahren

Begriffe (etwas chaotische Terminologiesituation):

Schlussfolgern (‘‘Reasoning’’) ist der allgemeinste Begriff: Aus Bekanntem noch Unbekanntes ableiten. (Shapiro 1987:822)

Deduktion ist wahrheitswerterhaltende Inferenz.

1. **Inferenz** ist elementares und *formalisierbares* Schlussfolgern (Shapiro 1987, Genesereth 1987:45 sqq)

   a. **Deduktion** ist logisch korrekte, d.h. *wahrheitswerterhaltende* Inferenz (Charniak 1985:14)

   - **Resolution** ist eine von vielen *Implementationen* von Deduktion (siehe ⇒hier.)

Induktion und Abduktion sind...

...nicht wahrheitswerterhaltend!

   b. *nicht wahrheitswerterhaltende* Inferenzen:

      i. **Induktion**

      ii. **Abduktion**

   ‘‘Modellbasierte Schlussfolgerungsverfahren’’ (‘‘model based reasoning’’) sind Spielarten/Kombinationen von Induktion und Abduktion.

2. **Intuitives Schlussfolgern** (‘‘common sense reasoning’’) ist das Anstellen von oft analog vorgehenden (z.B. auf optisch/graphischen Visualisierungen abstützenden) Schlussfolgerungen, die aber oft falsch sind.

4.1 Deduktion

4.1.1 Grundsätzliches

Definitionen des Begriffs ‘‘Deduktion’’

Sehr präzis, dafür schwer verständlich:

Zwei Definitionen und...

> A form of inference such that in a valid deductive argument the joint assertion of the premises and the denial of the conclusion is a contradiction. (Edwards 1967:(5):62)

© Michael Hess 2009

Etwas weniger präzis, dafür verständlicher:

> A deductive argument — a set of premises and a conclusion inferred from them — is said to be valid iff any situation in which the premises are (assumed to be) true is (thereby) also a situation in which [the] conclusion is (assumed to be) true. (Rapaport 1987:536)

...einige Beispiele.

Beispiele für Deduktionen:

Ganz wichtig: Modus Ponens

Modus Ponens:

```
p → q     Wenn ich hungrig bin, bin ich schlechter Laune
p         Ich bin hungrig
________  also

q         bin ich schlechter Laune
```

Aber auch:

Für uns weniger wichtig: Modus Tollens

Modus Tollens:

$p \rightarrow q$ Wenn ich hungrig bin, bin ich schlechter Laune
$\neg q$ Ich bin ***nicht*** schlechter Laune
________   also

$\neg p$ bin ich ***nicht*** hungrig

sowie

Die folgenden der Vollständigkeit halber:

Hypothetischer Syllogismus:

$\{p \rightarrow q, q \rightarrow r\} \vdash p \rightarrow r$

Transposition:

$\{p \rightarrow q\} \vdash \neg q \rightarrow \neg p$

© Michael Hess 2009

Disjunktiver Syllogismus:

$\{p \vee q, \neg p\} \quad \vdash \quad q$

Konjunktion:

$\{p, q\} \quad \vdash \quad p \wedge q$

Simplifikation:

$p \wedge q \quad \vdash \quad p$

Zur Erinnerung: Das sind *syntaktische* Beweise, welche die entsprechenden *semantischen* Beweise ‘‘nachvollziehen’’ sollen.

Wiederholung: Semantische Beweise

Semantische Beweise sind einfach eine wortwörtliche Auslegung der Definition der logischen Konsequenz, wie sie in ECL2 gegeben worden ist (‘‘Eine Formel F ist dann eine logische Konsequenz aus einer Menge von Formeln S, wenn *alle* Wahrheitswertbelegungen, welche S befriedigen, auch F befriedigen’’).

Man generiert also alle möglichen Kombinationen von Wahrheitswertbelegungen aller propositionalen Variablen der Formeln und schaut, ob immer dann, wenn alle Antezedens-Formeln wahr sind, auch die zu beweisende Formel wahr ist, d.h. man erstellt eine Wahrheitswerttabelle für die betrachteten Sätze und das zu beweisende Theorem

(Semantischer) Beweis durch Wahrheitswerttabelle
(Beispiel: Modus Ponens).

Ein Beispiel

Variablen		Antezedens 1	Antezedens 2	Theorem
p	q	$p \to q$	p	q
T	T	T	T	T
F	T	T	F	-
T	F	F	T	-
F	F	T	F	-

© Michael Hess 2009

(Nur) in der ersten Zeile sind beide Antezedens-Formeln wahr, und das Theorem ist also wahr (wenn nicht alle Antezedens-Formeln wahr sind, interessiert uns der Rest nicht mehr). Nur: Diese Methode wird sehr bald untragbar aufwendig, sogar für einen Computer: Für *n* Variablen muss man 2^n Zeilen ausfüllen.

4.1.2 Ein allgemeines Anwendungsbeispiel: Automatisches Theorembeweisen

Automatisches Theorembeweisen: Basis der Logikprogrammierung

Man kann für das automatische Theorembeweisen Kombinationen *verschiedener* Deduktionsmethoden verwenden, aber derartige Systeme werden rasch sehr komplex. Interessanter- und erfreulicherweise kann man dazu auch den Modus Ponens als *alleinige* Deduktionsmethode verwenden.

Ursprünglich war der Modus Ponens nur für aussagenlogische Theoreme gedacht aber als ‘‘generalisierter MP’’ (Russell 1995:269) ist der MP auch auf prädikatenlogische Aussagen anwendbar

‘‘generalisierter Modus Ponens’’

```
∀ M: mensch(M) → sterblich(M)
mensch(sokrates)
------------------------------
sterblich(sokrates)
```

und zwar durch den Einsatz

Unifikation

1. der Unifikation (zur gezielten Variablenbelegung)
2. einer kanonischen Repräsentation (zur Effizienzerhöhung)
   Ein Beispiel für eine kanonische Repräsentation ist die Klausellogik
3. einer effizienten Abarbeitungsstrategie (d.h. top down und depth first)

In dieser Form dann ± effizient mechanisierbar.

© Michael Hess 2009

4.1.3 Ein computerlinguistisches Anwendungsbeispiel: ‘‘Parsing as deduction’’

Ein (ursprünglich) revolutionäres Konzept.

Die Idee, dass Parsing als eine Anwendung von Theorembeweisen aufgefasst werden kann, ist relativ jung (ab 1970) und war sehr revolutionär. Die Entwicklung von Prolog als allgemeiner Programmiersprache entstand daraus.

Einige Probleme bei der Interpretation von Parsing als Theorembeweisen:

Differenzlisten für Input

- **Frage:** Wie stellt man die (schrittweise konsumierte) Eingabekette dar?
  Antwort: Differenzlisten und das Prädikat ‘‘connects’’

Syntaxstruktur als Beweisbaum

- **Frage:** Wie bekommt man eine Syntaxstruktur?
  Antwort: beliebige Strukturen können im Verlauf eines Beweises aufgebaut werden
- **Frage:** Wie führt man zusätzliche (d.h. nicht input-konsumierende) Tests durch?
  Antwort: Durch den Beweis von Termen ohne Differenzlisten

Backtracking für Mehrdeutigkeiten

- **Frage:** Wie behandelt man Mehrdeutigkeiten?
  Eine **Antwort:** Backtracking[6]
- **Frage:** Wie berücksichtigt man Analyse *und* Generierung (u.a.)?
  Antwort: Allein durch den Bindungszustand der Variablen im Theorem
- **Frage:** Wie behandelt man (quasi-)kontextsensitive Konstruktionen?
  Antwort: zusätzliche Argumente

Gap Threading

- **Frage:** Wie behandelt man Fernabhängigkeiten?
  Eine **Antwort:** Gap Threading mit zusätzlichen Argumenten

Cf. u.a. Pereira 1980.

6. andere Antworten: z.B. Chart Parsing

© Michael Hess 2009

4.2 Abduktion

4.2.1 Grundsätzliches

Hypothesen-generierung

Abduktion ist eine Art von Schlussfolgerung, welche versucht, für eine Reihe von Fakten und Regeln eine (gemeinsame) Erklärung zu ermitteln.

Definition:

> C.S. Peirce's name for the type of reasoning that yields from a given set of facts an explanatory hypothesis for them. (Edwards 1967: 5:57)

Einfachster Fall:

Generieren von Hypothesen über Fakten

```
A → B         Regeln
B             und Fakten
_______AB
A             erklärendes Faktum
```

Konkret:

```
regnet                       → strasse_nass
strasse_nass
_____________________________________________AB
regnet
```

Aber eben auch:

```
spritzwagen_unterwegs       → strasse_nass
```

Cf. Charniak 1985:20

© Michael Hess 2009

Überzeugender in Anwendung auf *mehrere* Fakten und Regeln:

```
A → B1
A → B2
A → B3
A → B4
A → B5

B1
B2
B3
B5
_________AB
A
```

Konkret:

```
regnet      → strasse_nass
regnet      → dächer_nass
regnet      → autos_haben_scheibenwischer_an

strasse_nass
autos_haben_scheibenwischer_an
________________________________AB
regnet
```

Aber eben auch:

```
spritzwagen_unterwegs    → strasse_nass
rohrbruch_in_strasse     → strasse_nass
```

Cf. Charniak 1985:453.

Beachte:

obwohl nicht wahrheitswert-erhaltend...

- Abduktion ist ein im Prinzip *unerlaubter Umkehrschluss* (z.B. beim Aufklären eines Mords: ‘‘wem nützt es? -> er war’s!’’)

...eine sehr nützliche Art von Inferenz

- *aber:* er erlaubt Fokussieren der Aufmerksamkeit beim Testen von Hypothesen über Kausalitäten (*also:* von den vielen theoretisch möglichen Tätern konzentriert man sich zuerst auf jenen, dem der Mord etwas nützt)

© Michael Hess 2009

- *natürlich:* das Verifizieren der Hypothesen ist erforderlich, wann immer möglich (aber wenn es *nicht* möglich ist, muss man u.U. mit der besten Hypothese leben: siehe ↓)

Was tun, wenn das Verifizieren der Hypothesen *nicht* möglich ist?

Hypothesen-verifikation unmöglich: der Normalfall

- Die *beste* Hypothese suchen (''Abduction as inference to the best explanation'')
- Die Möglichkeit der *Revision* dieser Hypothese schaffen (wenn später mehr Information verfügbar wird; z.B. kann man ein ''belief maintenance system'' [auch ''truth maintenance system''] aufbauen)

D.h. im Idealfall:

```
D ist eine Kollektion von Daten (Fakten, Beobachtungen).

H erklärt D (resp. würde, wenn wahr, D erklären).

Keine andere Hypothese kann D ebenso gut erklären
________________________________________________________AB

Daher ist H wahrscheinlich wahr
```

Beachte ''wahrscheinlich''! (Cf. Josephson 1994).

Konkret:

```
regnet                     → strasse_nass
regnet                     → dächer_nass
regnet                     → autos_haben_scheibenwischer_an
spritzwagen_unterwegs      → strasse_nass
rohrbruch_in_strasse       → strasse_nass

strasse_nass
autos_haben_scheibenwischer_an

¬ spritzwagen_unterwegs
¬ rohrbruch_in_strasse
_____________________________AB
regnet
```

Was heisst aber ''die beste Hypothese'' sein?

© Michael Hess 2009

Qualität einer Hypothese

Zumindest: Die Hypothese(n) sollte(n) sein

1. sparsam:[7] möglichst *wenig* Hypothesen erforderlich, um Daten zu erklären
2. zuverlässig:[8] maximaler Konfidenzwert der einzelnen Hypothesen; ermittelt (z.B.) aus (cf. Hobbs 1988:97)
   a. Länge des Beweises
   b. ‘‘Güte’’ der verwendeten Axiome
3. konsistent

Die Verrechnung der einzelnen Werte ist sehr aufwendig; neuronale Netze sind dafür gut geeignet.

Kommensurabilität

Erfordert zudem ‘‘Kommensurabilität’’, d.h. Möglichkeit, die einzelnen Komponenten in der selben Masseinheit zu bewerten. Ist nicht trivial (s.u.).

Cf. auch ⇒http://www.cis.ohio-state.edu/lair/Projects/LLU/LLU.html

4.2.2 Ein allgemeines Anwendungsbeispiel: Expertensysteme

Einführende Literatur zu Expertensystemen: Cf. Frost 1986, Bratko 1986, Wos 1984:371.

Teilt mit der Künstlichen Intelligenz im allgemeinen das Problem, dass man den Begriff ‘‘Expertensystem’’ nicht recht definieren kann:

7. auf Englisch: ‘‘parsimonious’’
8. auf Englisch: ‘‘confident’’

© Michael Hess 2009

4.2.2.1 Was sind Expertensysteme?

Definitionsversuche:

wissenschaftlich

- Hayes-Roth:

  ''High-performance programs in specialized professional domains, a pursuit that has encouraged an emphasis on the knowledge that underlies human expertise''

pragmatisch

- Winston:

  ''Computer programs that behave like a human expert in some useful ways''

technisch

- Feigenbaum:

  ''Computer systems that can help solve complex, real-world problems in specific scientific, engineering, and medical specialties''

detailliert

- Appelrath:

  ''Informationssysteme, die

  1. (meist grosse Mengen) auch heterogenen und oft vagen Wissens eines eng begrenzten Fachgebiets in problemangepasster Weise zu repräsentieren versuchen
  2. helfen, dieses Wissen zu akquirieren und zu verändern
  3. aus solchem Wissen mit meist heuristischen Methoden Schlussfolgerungen ziehen - damit neues Wissen ableiten - und auf Abfragen eines Benutzers hin Wissen erklärend und bewertend bereitstellen''

konzis

- Anon.:

  ''Systeme, die den Benutzern von komplexen (natürlichen oder künstlichen) Systemen Auskünfte über deren Funktionieren geben''

© Michael Hess 2009

G

Anstatt einer Definition betrachte man besser eine Liste von Programmen, die im allgemeinen als Expertensysteme gelten:

Es gibt tausende!

Einige bekannte Beispiele:

1. MYCIN, EMYCIN (Blutvergiftungsdiagnose plus Therapie)
2. DENDRAL (Interpretation von Massenspektrographiewerten)
3. PROSPECTOR (Lokalisation von Mineralienvorkommen)
4. R1 (Konfiguration von Mainframe-Computern)
5. Finanzkunde: ELI (Beratung zur Legislation)

zuerst gab's ''seichte'' Expertensysteme, ...

...dann ''tiefe''

Man kann unterscheiden zwischen:

a. ''seichten'' Expertensystemen (''erster Generation''); cf. Frost 1986 421 sqq.[9]
b. ''tiefen'' Expertensysteme (''zweiter Generation'') [10]

Beachte: andere Bezeichnung für Expertensysteme oft ''wissensbasierte Systeme''. **Aber:** nicht jedes wissensbasierte System erklärt sein Verhalten!

4.2.2.2 Problemlösung in ''seichten'' Expertensystemen

regelbasierte Systeme:

Sog. ''regelbasierte Systeme'' sind, als ''modularized know-how systems'', für die Implementation seichter Expertensysteme besonders geeignet.

Andere Bezeichnungen:

- ''Pattern-directed inference systems''
- Produktionssysteme

isolierte Produktionsregeln...

Grundgedanke: Das Wissen ist festgehalten in einer Grosszahl spezifischer *Produktionsregeln* in der Form von ''Bedingung-Aktions-Regeln''.[11]

9. also: Symptom → Diagnose
10. also: Ursache → Folge
11. auch: ''Situation-Aktions-Regeln'' genannt

© Michael Hess 2009

Die Repräsentation ist grundsätzlich immer von der Form (Bratko 1986:316)

```
    wenn  <Bedingung> dann <Aktion>
```

aber im einzelnen kann sie verschieden realisiert sein:

...verschiedender Art

```
    wenn <Situation> dann <Aktion auslösen>
```

oder

```
    wenn <Voraussetzung 1 und Voraussetzung 2>
        dann <Schlussfolgerung ziehen>
```

oder

```
    wenn <Bedingung 1 und Bedingung 2>
        dann <Bedingung 3 ausschliessen>
```

Konkretes Beispiel:

```
if 1) the infection is primary bacteremia, and
   2) the site of the culture is one of sterile sites, and
   3) the suspected portal of entry of the organism is the
      gastrointestinal tract
then
   there is suggestive evidence (0.7) that the identity
   of the organism is bacteroides
```

viele Vorteile

Vorteile dieser Methode:

1. *Modularität:* jede Regel definiert ein abgeschlossenes Stück Wissen
2. *Erweiterbarkeit:* neue Regeln können individuell angefügt werden
3. *Modifizierbarkeit:* bestehende Regeln können individuell modifiziert werden
4. *Transparenz:*
    a. ''wie''-Fragen: Wie bist du auf diese Schlussfolgerung gekommen?
    b. ''warum''-Fragen: Warum bist du an dieser Information interessiert?

    sind beantwortbar aufgrund der verwendeten Regeln

© Michael Hess 2009

Implementation als Prolog-Regeln

Einfachste Implementation solcher Regeln: direkt als Prolog-Regeln! Mit einer primitiven Benutzerschnittstelle zusammen kann man so ein sehr einfaches Expertensystem implementieren. anhand einer Reihe von Indikatoren (Symptomen).

Symptome → Diagnose

```
Diagnose                      Symptome

infection(P,bacteroides)      :-  patient(P),
                                  type_of_infection(P,'primary bacteremia'),
                                  site_of_culture(P,S), sterile(S),
                                  portal_of_entry(P,O,'gastrointestinal tract'),
                                  organism(O).
```

Um Konfusionen zu vermeiden, werden wir aber im folgenden für derartige Diagnose-Regeln immer den Operator "<-" verwenden.

Sodann:

- Bei Fehlerdiagnosesystemen ist man aber meist nicht einmal 100% sicher, dass ein einziges Symptom mit Sicherheit eine bestimmte Diagnose erlaubt. (Siehe oben ‘‘suggestive evidence’’).
- Gewisse (Fehl-)Funktionen treten spontan auf

Deshalb: (cf. Rowe 1988:164.) (zweite Regel frei erfunden! Summe bewusst nicht 1.0) Absolute und bedingte Wahrscheinlichkeiten:

```
infection(P,bacteroides,0.7) <- patient(P),
                                type_of_infection(P,'primary bacteremia'),
                                site_of_culture(P,S), sterile(S),
                                portal_of_entry(P,O,'gastrointestinal tract'),
                                organism(O).

infection(P,bacteroides,0.9) <- patient(P),
                                type_of_infection(P,'secondary bacteremia'),
                                site_of_culture(P,S), not_sterile(S),
                                portal_of_entry(P,O,'respiratory tract'),
                                organism(O).

infection(P,bacteroides,0.03).
```

© Michael Hess 2009

absolute Wahrscheinlichkeit

1. ein (beliebig ausgelesener) Patient hat mit Wahrscheinlichkeit 0.03 eine Bacteroides-Infektion: absolute oder unbedingte oder a-priori- oder *Faktenwahrscheinlichkeit*

   In der Stastistik notiert als: `P(A)=n`$_1$ z.B. `P(bacteroides)=0.03`

bedingte Wahrscheinlichkeit

2. *wenn* beim Patienten alle die Bedingungen von Regel gegeben sind, *dann* hat er mit Wahrscheinlichkeit 0.7 eine Bacteroides-Infektion: Bedingte oder *Regelwahrscheinlichkeit*.

   In der Stastistik notiert als: `P(A|B)=n`$_2$
   z.B. `P(bacteroides | type_of_infection(P,'primary bacteremia')`
   `and ... )=0.2`

Hier wird immer vorausgesetzt, dass das Vorliegen der Bedingungen selbst mit 100% Sicherheit festgestellt werden kann - aber nicht einmal das ist immer realistisch.

Daher (verinfachtes Beispiel!):

```
infection(P,bacteroides,0.7) <- patient(P),
                                type_of_infection(P,'primary bacteremia',0.2)
infection(P,bacteroides,0.9) <- patient(P),
                                type_of_infection(P,'secondary bacteremia',0.
```

Evidenzwahrscheinlichkeit

Die Wahrscheinlichkeit, *dass* man korrekt feststellt, dass der erste Typ von Infektion vorliegt, ist z.B. 0.2. Das ist die sog. *Evidenzwahrscheinlichkeit* (hier im Rumpf angegeben). Im Kopf steht weiterhin die Regelwahrscheinlichkeit.

Was ist nun die Wahrscheinlichkeit, dass effektiv eine Bacteroides-Infektion vorliegt, *wenn* man glaubt, dass eine ''sekundäre Bakterämie'' vorliegt? Bei der Annahme von unabhängigen Wahrscheinlichkeiten ist es das *Produkt der Einzelwahrscheinlichkeiten* (von Regel- und Evidenzwahrscheinlichkeit):

```
P(A and B) = P(A) * P(B)
```

''Und-Wahrscheinlichkeit'

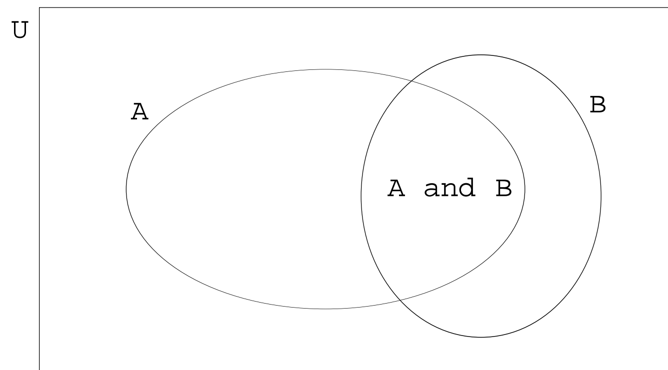
Heisst *''Und-Wahrscheinlichkeit''*.

Und was ist die *Gesamt*wahrscheinlichkeit, dass eine Bacteroides-Infektion vorliegt?

Gesamtwahrscheinlichkeit

Summe aller einschlägigen Einzelwahrscheinlichkeiten (je errechnet aus Evidenz- und Regelwahrscheinlichkeiten) *minus* Wahrscheinlichkeit, dass die Einzelwahrscheinlichkeiten gemeinsam auftreten: *''Oder-Wahrscheinlichkeit''* (Rowe 1988:168).

© Michael Hess 2009



U: Universum aller möglichen Ereignisse

NB: Subtraktion, weil man sonst den Durchschnitt zwei Mal zählen würde (cf. Russell 1995:422).

```
P(A or B) = P(A) + P(B) - P(A  and  B)

also  = P(A) + P(B) - (P(A) * P(B))
```

Am Beispiel von oben: Wenn wir *nur* festgestellt haben, dass die Infektion primäre Bakterämie ist, dann ist die Wahrscheinlichkeit, dass eine Bacteroides-Infektion vorliegt(ohne Berücksichtigung der a-priori-Wahrscheinlichkeit)

```
0.7 * 0.2 = 0.14
```

Etwas bescheiden.

je mehr Evidenz, desto höher die Zuverlässigkeit

Wenn *auch* eine sekundäre Bakterämie vorzuliegen scheint, dann

```
(0.7 * 0.2) + (0.9 * 0.5) -
        ((0.7 * 0.2) * (0.9 * 0.5)) = 0.527
```

Schon besser.

Die Berücksichtigung der a-priori-Wahrscheinlichkeiten ist heikler, weil diese meist *nicht* unabhängig von den a-posteriori-Wahrscheinlichkeiten sind (cf. Rowe 1988:174).

© Michael Hess 2009

Was ist unabhängige Wahrscheinlich-keit?

Bisher wurde immer betont, dass wir *unabhängige Wahrscheinlichkeiten* annehmen müssten. Was aber heisst ‘‘unabhängige Wahrscheinlichkeit’’?

Gegenbeispiel: Regeln oben

```
infection(P,bacteroides,0.7) <- patient(P),
                                type_of_infection(P,'primary bacteremia',0.2)
infection(P,bacteroides,0.9) <- patient(P),
                                type_of_infection(P,'secondary bacteremia',0.
```

Ermittlung der (Un)Abhängigkeit: keineswegs trivial!

ergaben Gesamtwharscheinlichkeit 0.527. Wenn primäre und sekundäre Bakterämie aber eine *gemeinsame Ursache* haben, kann der Wert prinzipiell nicht über dem höheren der zwei Einzelwerte liegen, also nicht über 0.45. Mit anderen Worten: Das Syptom ‘‘primäre Bakterämie’’ liefert unter diesem Umständen gar keine zusätzliche Bestätigung.

4.2.2.3 Problemlösung in ‘‘tiefen’’ Expertensystemen

Die sog. ‘‘tiefen’’ Expertensysteme gehen ganz anders vor, als die bisher bechriebenen ‘‘seichten’’.

Typische Expertensystemregeln waren (hier ein noch einfacheres Beispiel):

```
meningitis(0.2)  <-     stiff_neck.
meningitis(0.5)  <-     headache.

meningitis(0.03).
```

Aus dem Symptom (den Symptomen) leitet man mit einer gewissen Wahrscheinlichkeit *direkt* die Diagnose ab.

Vergleiche dagegen:

```
stiff_neck      <=      meningitis.
headache        <=      meningitis.
```

mit der Interpretation

Wenn man Meningitis hat, dann wird man *deswegen* einen steifen Hals (resp. Kopfschmerzen) haben

© Michael Hess 2009

Funktionales Modell der Realität erlaubt...

Das ist ein kausales, d.h. ein *funktionales logisches Modell* (eines Teils) der Welt:

- Meningitis *bewirkt* einen steifen Hals und Kopfschmerzen
- Ein solches Programm *simuliert* das Verhalten der modellierten Systeme.

...Simulation der Realität

Verwendungsmöglichkeiten funktionaler Modelle:

Man kann funktionale logische Modelle in *verschiedener Weise* verwenden.

(erneut:) multi-input multi-output

1. zum direkten Errechnen von Werten (also: als funktionales Modell), z.B.
   - um Resultate aus Input-Werten zu errechnen (+,-)
   - um gegebene Input-/Output-Korrelationen auf Korrektheit hin zu testen (+,+)
   - um alle möglichen Outputs zu errechnen (-,+)
   - um alle möglichen Input-/Output-Korrelationen zu generieren (-,-)

“analysis by synthesis”

2. zum Diagnostizieren von Fehlern (also: als Expertensysteme)
   a. Man macht einzelne Elemente im logischen Modell defekt (Verbindung unterbrechen, Outputwerte vertauschen, Kurzschluss etc.).
   b. Man erzeugt das Verhalten des (defekten) *modellierten* Systems.
   c. Man vergleicht es mit dem Verhalten des (defekten) *echten* Systems.
   d. Man hypothesiert, dass dasselbe Element (dieselben Elemente) im echten System defekt ist (sind).

Ein konkretes Beispiel eines funktionalen logischen Modells (0.L.2)

Ein *reales* ‘‘tiefes’’ Expertensystem: KARDIO (zu Herzrhythmusstörungen; Bratko 1989).

Diese Verwendung funktionaler Modelle zum Diagnostizieren (‘‘analysis by synthesis’’) ist

- ineffizienter, aber
- allgemeiner

als die direkten Symptom-Diagnose-Regeln (noch nie beobachtete Symptome können interpretiert werden!)

© Michael Hess 2009

Kombinationen seichter und tiefer Experten-systme

Wenn man direkte Diagnoseregeln einmal hat, sind sie effizienter als funktionale ''tiefe'' Modelle. Für die noch nie angetroffenen Fälle sollte man das logische Modell aber zumindest als ''Rückzugslinie'' bereithalten.

Zwei Fragen sind noch zu beantworten:

1. Wie kommt man zu den 100% zuverlässigen Regeln der funktionalen Modelle?
2. Wie kommt man von funktionalen Modellen zu Diagnoseregeln?

100% zuverlässige funktionale Regeln?

ad 1: Das ist eine Frage der Einzelwissenschaften, nicht der Künstlichen Intelligenz. Aber eines ist im allgemeinen Fall klar: Man weiss meist *nicht* genügend über die Welt, um funktionale Regeln mit einer Zuverlässigkeit von 100% schreiben zu können. So mag das Vorliegen von Meningitis in 50% der Fälle kausal zu steifem Hals führen, aber nicht in 100% der Fälle (siehe gleich unten). Dennoch muss man dieses partielle Wissen verwenden können. Wie? Dazu und zu Punkt 2:

Bayes'sche Regel

ad 2: Das ist eine ganze Wissenschaft für sich. Das grundlegende Prinzip ist die *Bayes'sche Regel*.

Eine einfache *Ableitung* dieser Regel geht aus von der (cf. Russell 1995:426) sog. *Produktregel:*

$$P(A \wedge B) = P(A|B)\ P(B)$$
$$P(A \wedge B) = P(B|A)\ P(A)$$

Wenn man die rechten Seiten gleichsetzt

$$P(B|A)\ P(A) = P(A|B)\ P(B)$$

und dann durch $P(A)$ teilt, ergibt sich

© Michael Hess 2009

Bayes'sche Regel

$$P(B|A) = \frac{P(A|B)\ P(B)}{P(A)}$$

von kausalen Regeln...

Anwendung:

Kausale Regel:

```
P(stiff_neck|meningitis)     = 0.5
P(headache|meningitis)       = 0.7
```

A priori Wahrscheinlichkeiten:

```
P(meningitis)                = 1/50'000
P(stiff_neck)                = 1/20
```

d.h.

1. Nur in 50% der Fälle *bewirkt* Meningitis (kausal!) einen steifen Hals (ein Beispiel für eine ''unvollkommene'' kausale Regel im funktionalen Modell).
2. Die (unbedingte) Wahrscheinlichkeit, dass ein (beliebiger!) Patient *Meningitis* hat (''Prävalenz''), ist recht klein (in der statistischen Literatur auch ''prior'' oder ''marginal probability'' genannt)
3. Die unbedingte Wahrscheinlichkeit, dass ein Patient einen *steifen Hals* hat, ist ziemlich hoch (man kann aus vielen verschiedenen Gründen einen steifen Hals haben).

Nunmehr lässt sich via Bayes'sche Regel ableiten:

...zu Diagnoseregeln

© Michael Hess 2009

Diagnoseregel

$$P(\text{meningitis}|\text{stiff_neck}) = \frac{P(\text{stiff_neck}|\text{meningitis})\ P(\text{meningitis})}{P(\text{stiff_neck})}$$

$$= \frac{0.5 \times 1/50'000}{1/20} = 0.0002$$

mit anderen Worten: Nur 2 von 10’000 Patienten mit steifem Hals haben effektiv Meningitis - *obwohl* Meningitis mit einer relativ hohen Wahrscheinlichkeit einen steifen Hals bewirkt. Aber eben: Steifer Hals kommt auch aus vielen *anderen* Gründen vor. Schönes Beispiel, wie sich die ‘‘Normalintuition’’ täuschen kann (weil wir geneigt sind, falsche Umkehrschlüsse vorzunehmen). Analog für den Kopfschmerz.

Dies waren nur die allereinfachsten Überlegungen zu diesem Thema.

Das war nur der Anfang!

Natürlich müsste man auch bedenken:

- Ein Patient könnte *sowohl* steifen Hals *als auch* Kopfschmerzen haben, was die *kombinierte Wahrscheinlichkeit* einer Meningitis erhöht (Russell 1995:428)
- Oder wir wissen, das auch ein Schleudertrauma einen steifen Hals bewirkt; dann muss man die *relative Wahrscheinlichkeit* der zwei Krankheiten/Verletzungen berechnen (Russell 1995:427).
- Und ganz wichtig: Realistischerweise hat man nicht ‘‘100% zuverlässige Beobachtungen’’ zur Verfügung (‘‘hat Kopfweh’’), sondern muss man diagnostische Tests durchführen, und die sind selbst mehr oder weniger zuverlässig. Dazu mehr im folgenden Unterdokument.

Berücksichtigung der Zuverlässigkeit von Tests (0.L.3)

Zum ganzen Themenbereich wesentlich mehr bei Russell 1995:415 ff.

© Michael Hess 2009

4.2.2.4 Verhältnis zwischen ‘‘seichten’’ und ‘‘tiefen’’ Expertensystemen

Wie verhalten sich ‘‘seichte’’ und ‘‘tiefe’’ Expertensysteme...

Man kann das Verhältnis zwischen ‘‘seichten’’ und ‘‘tiefen’’ Expertensystemen sehen als das zwischen Abduktion resp. Deduktion **über den Regeln von funktionalen logischen Modellen** der Welt:

Deduktion beim *Simulieren* (z.B. ‘‘tiefes ES’’)

```
Folge                          Ursache

symptom(stiff_neck)     <-     disease(meningitis).
symptom(headache)       <-     disease(meningitis).

symptom(stiff_neck).
_______________________________  DEDUKTION
disease(meningitis)
```

und zwar hier (unrealistischerweise) mit 100%-iger Sicherheit (also wahrheitswerterhaltend) Dagegen *Abduktion* beim direkten *Diagnostizieren* (z.B. ‘‘seichtes’’ ES)

...in *logischer* Hinsicht?

```
Diagnose                          Symptom

disease(meningitis)     <=     symptom(stiff_neck).
disease(meningitis)     <=     symptom(headache).

symptom(stiff_neck).
_______________________________  ABDUKTION
disease(meningitis)
```

Weiterführendes zur Integration von logischen Inferenzmethoden und probabilistischen Überlegungen zu unsicherem Wissen: Russell 1995:436 ff.

© Michael Hess 2009

G

4.2.3 Ein computerlinguistisches Anwendungsbeispiel: Interpretation durch Abduktion

Abduktion ist (wäre?) sehr wichtig in der Computerlinguistik.

Auch bei der Interpretation natürlicher Sprache ist es oft erforderlich, die *beste* von mehreren Erklärungsmöglichkeiten für eine sprachliche Äusserung finden. Die sprachlichen Äusserungen werden also als gewissermassen *Indizien* dafür genommen, was der Sprecher/Schreiber sagen *wollte*.

Hierzu u.a. kurz: Hobbs 1988, Norvig 1990, Hobbs 1993.

Dazu gehören zumindest:

1. Referenzauflösung
2. Kompositaauflösung
3. Desambiguierung

4.2.3.1 Referenzauflösung durch Abduktion

Referenzauflösung: Einfachster Fall sind definite Nominalphrasen.

Akkommodation ist (ein Form von) Abduktion

Wenn kein Antezedens gefunden, dann *akkommodieren* wir es:
***Der* Präsident der Universität Zürich ernennt *die* Vizepräsidenten der Universität Zürich und *ihre* Stellvertreter.**
''Wie müsste die Welt aussehen, dass sich diese Aussage interpretieren lässt?''

Regel: *Wenn* 1. ein Objekt existiert, *und* 2. der Sprecher *und* 3. der Hörer davon wissen, *dann* ist eine definite Nominalphrase zur Referenz angebracht.

Beobachtung: Es wird eine definite Nominalphrase zur Referenz verwendet, aber der Hörer weiss nichts von der Existenz des Objekts.

Also Abduktion: Existenz *und* Wissen des Hörers werden akkommodiert.

Aber: Akkommodieren ist nur *eine* Erklärungsmöglichkeit. In Konkurrenz dazu z.B.:

© Michael Hess 2009

1. Sprecher/Autor ist schlecht informiert
2. Sprecher/Autor will irreführen
3. generische Lesart von Nominalphrasen ist intendiert

Eventuell sind auch *mehrere* solcher Erklärungen gleichzeitig denkbar. Daher liegt hier ein (einfacher) Fall von Abduktion vor.

4.2.3.2 Kompositaauflösung durch Abduktion

Ein bekanntes und sehr schwieriges Problem in der Computerlinguistik ist die Auflösung von Komposita (resp. von Verkettungen im Englischen)

Was bedeuten (Nominal)-Komposita?

Beispiel:

1) Disengaged compressor after *lube-oil alarm*

[12] Was heisst das? ‘‘Alarm *aus* Schmieröl *gemacht*’’? Oder (in Analogie zu ‘‘Personenalarm’’) ‘‘Alarm *gerichtet an* Schmieröl’’? Der Kontext (Serviceberichte über Fehlfunktionen von Maschinen) macht klar, dass es ein Alarm *gerichtet an jemanden* ist (wonach *irgendetwas* mit dem Schmieröl nicht stimmt).

“bezüglich” ist fast bedeutungsleer.

Aber ohne sehr viel Weltwissen können wir nur die (fast leere) Interpretation ‘‘Alarm *bezüglich* Schmieröl’’ ableiten:

```
lubeoil(o) ∧ alarm(a) ∧ nn(o,a)
```

nn ist die semantische Repräsentation von ‘‘bezüglich’’ - also: man weiss es nicht genau.

Eine manchmal vertretene Erklärung der Bedeutung von Komposita ist, dass ihre Komponenten in einer *beliebigen* Beziehung stehen können. Das ist aber eine wenig hilfreiche ‘‘Erklärung’’, denn im *konkreten* Fall wissen wir Menschen meist recht genau, welche Beziehung vorliegt.

12. Quelle: Ein Servicebericht.

© Michael Hess 2009

Bedeutungs-postulate ≅ Begriffs-definitionen

Deshalb kann man versuchen, Bedeutungspostulate für die Relation *nn* zu verwenden:

$\forall$ `X,Y: component_of(Y,X)` $\rightarrow$ `nn(X,Y)` (‘‘filter element’’)
$\forall$ `X,Y: portion_of(X,Y)` $\rightarrow$ `nn(X,Y)` (‘‘oil sample’’)
$\forall$ `X,Y: for(Y,Z)` $\rightarrow$ `nn(X,Y)` (‘‘lube-oil alarm’’)

Dabei soll `component_of(Y,X)` heissen, dass `Y` ein eigenständiges Objekt als Teil eines andern, eigenständigen Objekts `X` ist, während `portion_of(X,Y)` sagen soll, dass `X` ein willkürlich abgrenzbarer Ausschnitt von `Y` ist.

konkurrierende Erklärungen

Das sind alles mögliche, aber *konkurrierende* Erklärungen (also: Hypothesen) für das Auftreten von *nn* (resp. des Nominalkompositums) in einem konkreten Fall. Um nun die *beste* Hypothese zu ermitteln, kann man Abduktion (d.h. Akkommodation von Hypothesen via die obigen Regeln) und komponentenweises Gewichten der Erklärungen verwenden.

Deshalb: Gewichtung der Erklärungen

Zur Gewichtung der Erklärungen:

- Da ‘‘lube-oil’’ und ‘‘alarm’’ als syntaktisch indefinit betrachtet werden können (im gegebenen Kontext werden fast nie *explizit* indefinite Nominalphrasen verwendet), sind die Kosten ihrer ‘‘Annahme’’ gering (sie liefern *neue* Information, die *definitionsgemäss* assertiert [und nicht akkommodiert] werden muss).
- Da aber ‘‘nn’’ eine *schlechte* (da massiv unterspezifizierte) Erklärung ist, ist sie teuer. Die Erklärungen durch die diversen Bedeutungspostulate müssen daher billiger gemacht werden

Also z.B.:

$\texttt{lubeoil(o)}^{\$5} \wedge \texttt{alarm(a)}^{\$5} \wedge \texttt{nn(o,a)}^{\$20}$

$\forall$ `X,Y: component_of(X,Y)` $\rightarrow$ `nn(X,Y)` $^{\$10}$
$\forall$ `X,Y: portion_of(X,Y)` $\rightarrow$ `nn(X,Y)` $^{\$10}$
$\forall$ `X,Y: for(Y,Z)` $\rightarrow$ `nn(X,Y)` $^{\$10}$

(Die Superskripte geben Kosten an im Sinn von ‘‘Erklärungsqualität’’ - je schlechter, desto teurer; man wählt die billigste Interpretation). Das heisst also: Statt der 20$ teuren ‘‘Erklärung’’ `nn` kann man eine der drei nur 10$ teuren Erklärungen via Bedeutungspostulat nehmen).

Damit hat man vorerst nur die Entscheidung getroffen, *dass* akkommodiert werden sollte (aber nicht, *wie* - denn alle Bedeutungspostulate haben vorerst dieselben Kos-

© Michael Hess 2009

Erklärungsqualität

ten).

Daher (z.B.):

```
∀ X,Y: component_of(X,Y) → nn(X,Y) $10    if artefact(X) ∧ artefact(Y)
∀ X,Y: component_of(X,Y) → nn(X,Y) $20    if mass(X) ∧ mass(Y)

∀ X,Y: portion_of(X,Y) → nn(X,Y) $10    if mass(X) ∧ mass(Y)
<etc.>
```

Das soll heissen, dass `component_of(Y,X)` die wesentlich wahrscheinlichere Erklärung ist, wenn `X` und `Y` Artefakte sind, als wenn sie unstrukturierte Substanzhaufen sind. Umgekehrt sollte ‘‘oil sample’’ eher als ‘‘a sample which is a portion of oil’’ interpretiert werden denn als ‘‘a sample which is a *component* of oil’’.

auch zu beachten: Erkennungs-Wahrscheinlichkeit

Zusätzlich muss man berücksichtigen: Die *Erkennungs*-Wahrscheinlichkeit(also dass ‘‘lube-oil alarm’’ überhaupt eine Nominalverkettung ist) ist sehr hoch.

Das ist nicht immer so einfach:

oil and water pressure OK

oft problematisch!

Ist syntaktisch ambig:

1. (oil and water) pressure OK
2. oil and (water pressure) OK
3. oil [pressure] and water pressure OK

Rein syntaktisch ist 1. am wahrscheinlichsten (am ‘‘billigsten’’), aber die Qualität der Erklärung ist gering.

Moral: Man muss unterscheiden zwischen:

1. Kosten der Annahme
2. Qualität der Erklärung

Die zwei Werte müssen irgendwie verrechnet werden. Zwei Probleme:

1. Art der Verrechnung [13]

13. Siehe unten: Kostenfunktion und Evaluationsfunktion im A*-Suchverfahren

© Michael Hess 2009

2. (erneut)

Kommensurabilität

Anderes Beispiel (konvers zum ‘‘lube-oil alarm’’): Zauberer zieht Kaninchen aus dem Hut. Eine perfekte Erklärung ist, dass das Kaninchen im Hut entstanden ist (perfekt: die *Erscheinung* wird perfekt erklärt). Bloss: Sehr hohe Kosten der Annahme (man muss eine neue *Kausalität* postulieren).

4.2.3.3 Desambiguierung struktureller Mehrdeutigkeiten durch Abduktion

Desambiguierung struktureller Mehrdeutigkeitenen kann analog zur Kompositaauflösung betrieben werden. Insbes. muss die (semantische) Repräsentation der unterspezifizierten (syntaktischen) Strukturen teuer sein:

1) Disengaged compressor *after* lube-oil alarm

heisst das ‘‘*disengaged **after*** lube-oil alarm [sounded]’’ oder ‘‘[the] *compressor **after*** lube-oil alarm’’?

Zur Not kann man das so darstellen:

```
∃ E,C,Y,A: disengaged(E,C) ∧ alarm(A) ∧ compressor(C)
           ∧ after(Y,A) ∧ Y ∈ {C,E}
```

`Y ∈ {C,E}` heisst nichts anderes, als (exklusive) Disjunktion in der logischen Datenbank repräsentieren. Macht die Sache 1. rechenintensiv und 2. unpräzis.

Die (syntaktische) Wahrscheinlichkeit eines bestimmten Anschlusses und die (semantische) Güte der dadurch erreichten Interpretation müssen sinnvoll kombiniert werden.

Hier *könnte* die Computerlinguistik von den Expertensystemen lernen!

Viele dieser Überlegungen (z.B. Verrechnen der Stärke von Indizien für ein bestimmtes Phänomen, Unterscheidung Erklärungswahrscheinlichkeit vs. Erscheinungswahrscheinlichkeit) wurden in der Literatur zu Expertensystemen schon ausführlich untersucht.

Daher wurde oben relativ ausführlich auf Expertensysteme eingegangen (obwohl sie

© Michael Hess 2009

nicht mehr die modernste Anwendung der Künstlichen Intelligenz sind).

4.3 Induktion

Definitionen

1. Inferenz von Beobachtungsdaten auf *Regel*
2. Inferenz von Zusammensetzung einer Teilgruppe auf Zusammensetzung der *gesamten* Gruppe; cf. Edwards 1967 (5):66

```
Beispiel (für 1):

                 hund(fido)
                 bellt(fido)

                 hund(bello)
                 bellt(bello)

                 bellt(chico)
                  -

                 hund(damian)
                 bellt(damian)

                 bellt(elfie)
                  -
                 ________________________
                 ∀ X: hund(X) → bellt(X)
```

mehr: Shapiro 1987, Charniak 1985:609).

Induktion ist ebenfalls nicht logisch korrekt im Sinn von ‘‘wahrheitswerterhaltend’’ - sonst wäre die Wissenschaft an und für sich überflüssig, weil sie automatisiert werden könnte.

regelhafte Zusammenhänge finden

Aber: Induktion ist der zentrale Schritt in jeder Wissenschaft beim Aufstellen von Hypothesen über regelhafte Zusammenhänge (‘‘Gesetze’’).

Zwei Beispiele:

1. maschinelles Lernen
2. Erschliessen von Grammatiken aus Corpora

© Michael Hess 2009

4.4 Deduktion, Abduktion und Induktion im Vergleich

```
∀ H: hund(H) → bellt(H)             Regel
hund(barry)                         Faktum
__________________ DEDUKTION
bellt(barry)                        Faktum


∀ H: hund(H) → bellt(H)             Regel
bellt(bello)                        Faktum
___________________ ABDUKTION
hund(bello)                         hypothet. Fakten


hund(fido)                       Fakten
bellt(fido)

hund(bello)
bellt(bello)

bellt(chico)
 -

hund(damian)
bellt(damian)

bellt(elfie)
 -
____________________ INDUKTION

∀ H: hund(H) → bellt(H)             hypoth. Regeln
```

© Michael Hess 2009

5. Problemlösungsstrategien

5.1 Kontrolle von Inferenzprozessen

Logik vs. Kontrollwissen: Wenn man sich für die Verwendung einer bestimmten *Art von Inferenz* (oder auch von mehreren) entschieden hat, kann man immer noch ganz verschiedene *Strategien bei der Verwendung* der entsprechenden Axiome anwenden.

Es gibt viele Beweisstrategien,...

- **Bisher** wurde stillschweigend angenommen: Als *Inferenzmethode* die Deduktion und als *Art der Verarbeitung* die ‘‘Prolog-Beweisstrategie’’.

...nicht nur die von Prolog.

- **Aber:** Das ist nicht die einzige Möglichkeit!
- **Mehr noch:** Bei *allen* Arten von Inferenz stellt sich die Frage nach der Art der Verarbeitung, d.h. nach der *Kontrolle*. Die Kontrolle ist also ein Schema, das die Reihenfolge der Abarbeitung der Ausdrücke eines Programmtexts festlegt.

deklarative vs. prozedurale Beschreibung

Konkret: Ein Programm in (‘‘reinem’’) Prolog ist eine rein *deklarative* Beschreibung einer Situation, im Gegensatz zur *prozeduralen* Beschreibung des Verhaltens in einer sog. imperativen Sprache.

Was ist ‘‘reines’’ resp. ‘‘unreines’’ Prolog? (0.L.4)

Kontrollwissen und...

Erst das Anwenden der *Abarbeitungsprozedur* auf den Programmtext ergibt ein Verhalten. Die Information, welche diese Prozedur steuert, nennt man ‘‘*Kontrollwissen*’’ oder Kontrollinformation.

Schlagwortartig: ‘‘Logik + Kontrollfluss = Algorithmus’’

...Kontrollfluss

Anhand eines Beispiels aus der Prolog-Welt: Um das Ziel im untenstehenden Programm zu beweisen, muss man jeden Knoten auf jedem Pfad vom Ziel zu den *relevanten* Fakten (mindestens) einmal abklappern (das ist die Logik) - aber die Reihenfolge, in der das geschieht, kann man im Prinzip frei wählen, und das ergibt den *Kontrollfluss*.

Beispiel:

© Michael Hess 2009

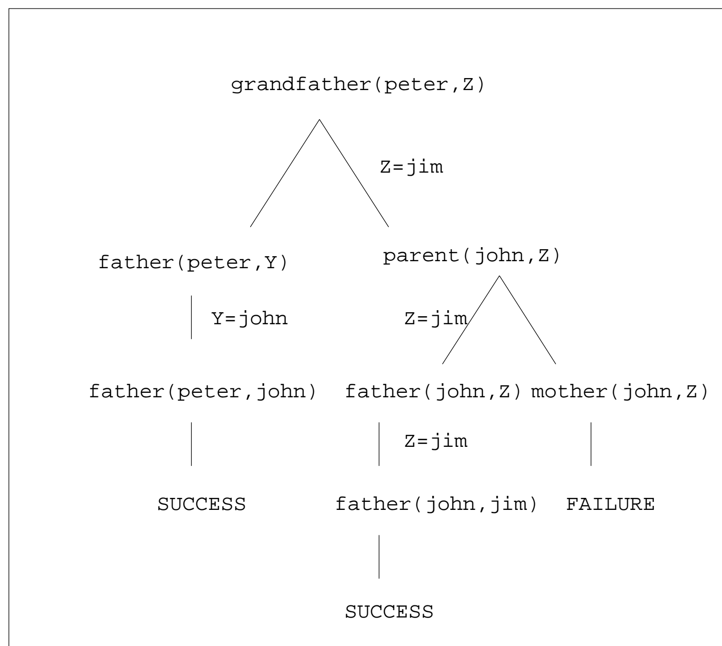
Ziel:

```
?- grandfather(peter,X).
```

Programm:

```
grandfather(X,Z)     :-   father(X,Y),parent(Y,Z).
parent(X,Y)          :-   father(X,Y).
parent(X,Y)          :-   mother(X,Y).

father(peter,john).
father(john,jim).
father(bob,mary).
mother(mary,jill).
```

Das sieht man am leichtesten anhand des sog. Beweisbaums:



Viele Wege führen zum Ziel.

In diesem (einfachen) Fall (mit nur einer einzigen Lösung) ist der Baum denkbar einfach. Aber man sieht immerhin, dass man das Ziel auf ganz verschiedenen Wegen erreichen kann.

© Michael Hess 2009

Wie ermittelt man einen Beweisbaum? (0.L.5)

Diese Erwägungen über Kontrollwissen sind auch in *unreinem* Prolog wichtig - im Grund sogar wichtiger!

Zu unterscheiden sind *zwei Arten von Kontrollwissen:*

1. *allgemeines* Kontrollwissen (''Abarbeitungsstrategien'')
2. *bereichsspezifisches* Kontrollwissen (oft ''Heuristik'' genannt); siehe aber unten; cf. Jackson 1985, Bratko 1986:265

Die Geschichte der Expertensysteme ist ± die Geschichte von der Ersetzung (zumindest: der Ergänzung) der ersten Sorte Wissen durch die zweite.

Diese Überlegungen zum automatischen Beweisen lassen sich verallgemeinern auf Problemlösungsstrategien in der Künstlichen Intelligenz allgemein.

5.2 ''Suche'' als Abstraktion von Problemlösungsverfahren

eine gemeinsame ''Sprache'' fürs Problemlösen

Um verschiedene Problemlösungsstrategien zu vergleichen, brauchen wir gemeinsame ''Sprache''. Man verwendet dazu das Konzept der ''Suche'' in einem ''Zustandsraum''

Zustände

1. Jede mögliche/legale Konfiguration ein *Zustand*.
   a. Davon sind einer/mehrere *Ausgangszustand*/zustände
   b. Davon sind einer/mehrere *Zielzustand*/zustände

Operatoren

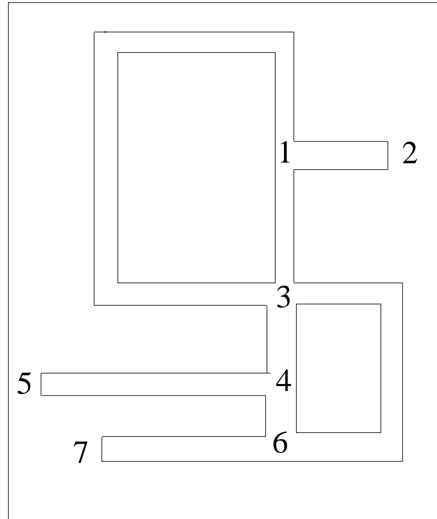
2. Es gibt eine Menge von *Operatoren*, welche einen Zustand in einen anderen überführen

Beispiele:

wohl einfachstes Beispiel

1. **Wegsuche** (cf. Rowe 1988:195)

© Michael Hess 2009



Zustände sind die jeweilige Position des Suchenden im Raum; insbes.

Problem der Konzeptualisierung

i. ein Ausgangszustand (z.B. 1)

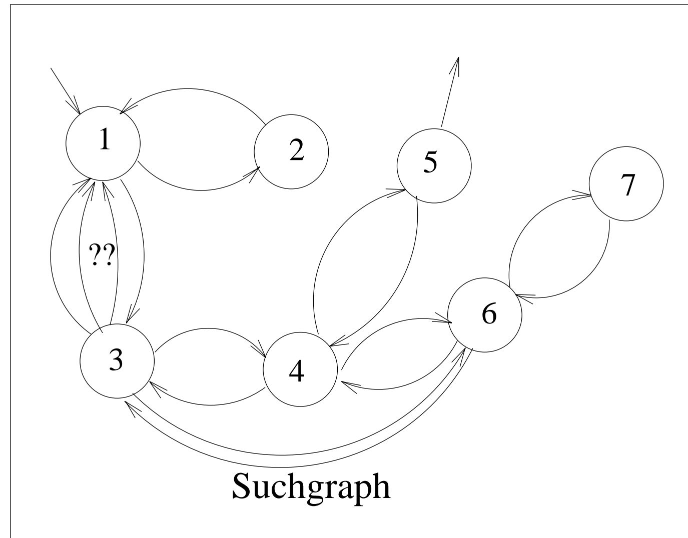
ii. ein Zielzustand (z.B. 5)

iii. *Operatoren:* sich von einem ‘‘Kontrollpunkt’’ zum andern bewegen (z.B. Kreuzung od. Verzweigung, Sackgasse)

iv. besonderes Merkmal: einzige Veränderung der Welt ist die eigene Position; einfachster Fall

v. Darstellung als Suchgraph:

© Michael Hess 2009

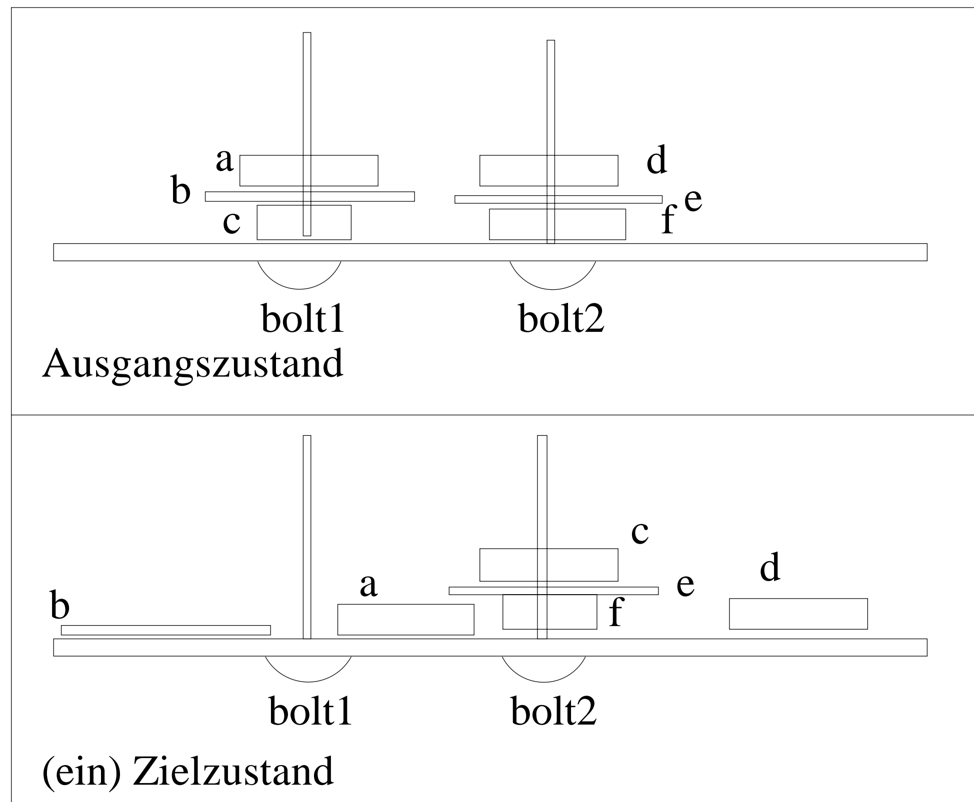


Suchgraph

Die Frage, ob man zwischen den Zuständen 1 und 3 *zwei* Pfadpaare einfügen muss (wegen der zwei möglichen Wege zwischen den zwei Punkten) ist eine typische Kozeptualisierungsfrage: Es kommt ganz drauf an, was das letztendliche Ziel der Formalisierung ist. *Wenn* z.B. weder Länge des Wegs noch Anzahl von Ecken eine Rolle spielen sollen, dann sind die zwei Routen zwischen 1 und 3 ununterscheidbar und müssen daher im Suchgraphen auch nicht unterschieden werden.

2. **Konfigurieren** (hier: Bauteile zusammensetzen; cf. Rowe 1988:227)

© Michael Hess 2009



Zielbedingung
≠
Zielzustand

i. ein Ausgangszustand

ii. mehrere Zielzustände (die einzige Ziel-*Bedingung* könnte z.B. sein: ''c'' muss auf ''e'' muss auf ''bolt2'' sein; in Benutzersicht ist das ein *einziges* (unterspezifiziertes!) Ziel, das aber durch *mehrere* verschiedene Ziel-*Zustände* instantiiert werden kann)

iii. (einziger) Operator: etwas von A nach B verschieben

iv. besonderes Merkmal: eines von mehreren Zwischenzielen kann von späteren Operatoren wieder zerstört werden

3. **Brettspiele** (hier: Schach)

i. ein Ausgangszustand

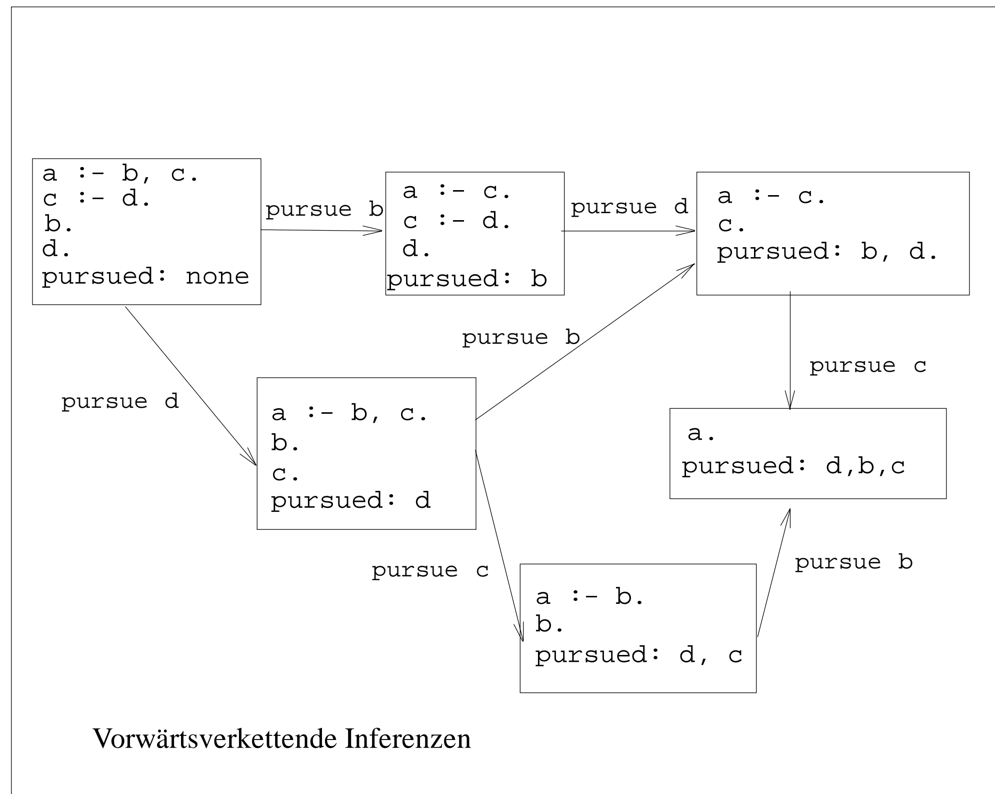
ii. mehrere Zielzustände (keine Bewegungsmöglichkeiten mehr; Schachmatt)

iii. (mehrere) Operatoren: legale Bewegungen der Figuren

iv. besonderes Merkmal: ein intelligenter Gegner

© Michael Hess 2009

4. **Kontrolle** (hier: Steuerung von Inferenzprozessen; cf. Rowe 1988:196)



Vorwärtsverkettende Inferenzen

i. ein Ausgangszustand: gegebene Regeln und Fakten
ii. (potentiell) mehrere (implizite) Zielzustände: alle Situationen, in denen keine Regeln mehr existieren; (also wie im Konfigurationsbeispiel: mehrere Zielzustände, definiert durch eine einzige Bedingung)
iii. ein Operator: Regel via Fakten reduzieren
iv. besonderes Merkmal: Suchprozedur ist ein Meta-Algorithmus (Regeln [Operatoren] steuern die Verwendung von Regeln)

5. **Syntaxanalyse**
   i. ein Ausgangszustand: gegebene Grammatik (Regeln) und Lexikon (Fakten)
   ii. ein Zielzustand: ein daraus abzuleitender Satz (Theorem)
   iii. (mehrere) Operatoren: ein neues Faktum (Analyse einer Teil-Phrase) aus Regeln und Fakten ableiten

© Michael Hess 2009

iv. besonderes Merkmal: Lexikon kann auch Analysen ganzer Teil-Phrasen enthalten (z.B. Idiome)

6. usw. usf. (im Prinzip jedes denkbare Problem)

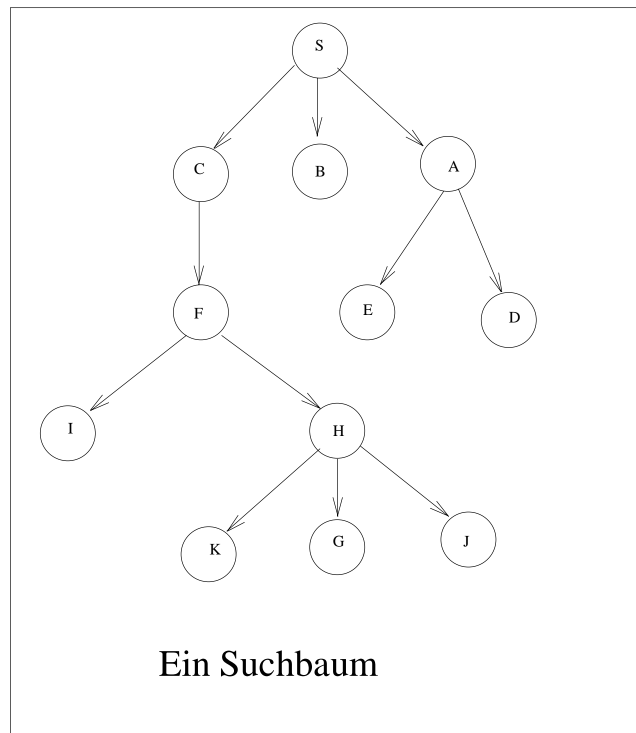
Suchraum

Suchgraph

Suchbaum

Darstellung von Suchräumen:

- Im *allgemeinen* Fall: Such*graph* (Rowe 1988:198)
- Im *einfachsten* Fall: Such*baum*.
- Der Graph (Baum) ''wächst'' *während* der Suche (nicht der ganze Suchraum ist von Anfang an sichtbar). Das wird oft ein wenig vergessen!



Ein Suchbaum

Ein Suchbaum stellt alle möglichen Abfolgen von Operatoren (also: Berechnungen) dar. Ein Suchbaum ist immer relativ zu einem *Ziel* und zu einem *Programm* definiert. Das Ziel ist die ''Wurzel'' des Baums.

© Michael Hess 2009

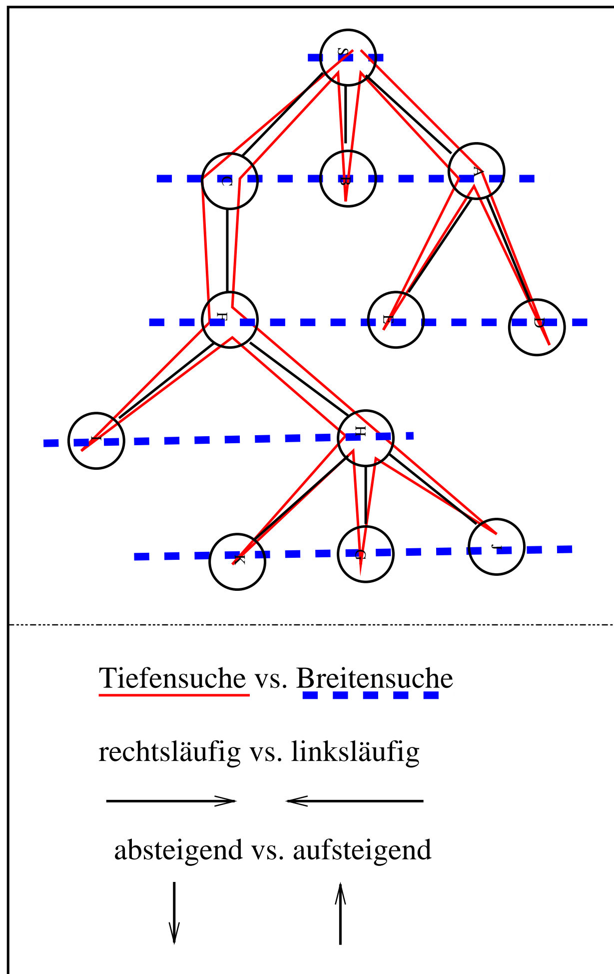
5.3 Allgemeine Suchstrategien

‘‘Allgemeine Suchstrategien’’ sind Strategien, welche *ohne* Verwendung von anwendungs- resp. bereichsspezifischem Wissen funktionieren. Oft ‘‘uninformiert’’, ‘‘blind’’, ‘‘brute force’’ genannt.

drei Dimensionen von Strategien

Man kann *drei* Dimensionen (nicht: Typen) von Strategien unterschieden: Man kann einen Suchraum abschreiten

1. mit Tiefensuche (depth-first) *oder* Breitensuche (breadth-first)
2. absteigend (top-down; auch: backward chaining, goal-directed reasoning) *oder* aufsteigend (bottom-up; auch: forward chaining, data-driven reasoning)
3. rechtsläufig (left-right) *oder* linksläufig (right-left)



© Michael Hess 2009

acht reine Strategien...

In *jeder* Beziehung muss man sich für eines der Regimes entscheiden, d.h. es gibt acht verschiedene ''reine'' Kombinationen (d.h. konkrete Strategien). Zudem kann man das Kontrollregime je nach erreichten Zwischenergebnissen ''fliegend'' wechseln, d.h. eine ''gemischte'' Strategie anwenden:

...und viele gemischte.

1. mitten in der Analyse von aufsteigend zu absteigend übergehen (z.B. ''left-corner'')
2. z.B. anhand von wiederholten Durchgängen reiner Tiefensuche den Effekt von Breitensuche erzielen (''iterative deepening'')
3. irgendwo in der Mitte anfangen und sich von dort quasi-simultan nach beiden Richtungen vorarbeiten (''middle-out'')

und zusätzlich: parallel vs. sequentiell

Schliesslich kann man ein Programm *sequentiell* abarbeiten (heute noch der Normalfall) oder aber *parallel* (wobei es verschiedene Arten Parallelverarbeitung von Logik-Programmen gibt; siehe unten). Das ist aber eine andere Art der Unterscheidung zwischen Abarbeitungsstrategien! **NB:** Breitensuche lässt sich zwar leichter parallelisieren, aber sie ist *an sich* auch sequentiell.

5.3.1 Einige reine Strategien

5.3.1.1 Absteigende rechtsläufige Tiefensuche

Die Prolog-Beweisstrategie...

Prolog verwendet:

1. rechtsläufig
2. absteigend
3. Tiefensuche mit chronologischem Backtracken

...ist oft brauchbar und zudem...

Warum? In den meisten Fällen die einfachste Strategie:

1. *Rechtsläufigkeit:*
   - Im Bereich der Sprachverarbeitung sind natürliche und formale Sprachen in Europa/USA so in der Regel einfacher zu erfassen
   - allerdings: wenn man in der morphologischen Analyse die Suffixe im gleichen Arbeitsgang wie die Syntax erfasst, nicht mehr

© Michael Hess 2009

- im folgenden wird aber immer Rechtsläufigkeit angenommen

2. *absteigende* Strategie:

   - günstig, wenn wenige (und wohldefinierte) Ziele und meist viele Fakten.

     Beispiele:

     1. Einen Verdächtigen eines Mordes überführen. Ein einziges Ziel (‘‘Er ist der Mörder!’’), viele (z.T. noch unbekannte) Fakten (wo war er zur Tatzeit? Was ist seine Blutgruppe?)
     2. Syntaxanalyse: *ein* Satz, den man analysieren will

   - erlaubt besonders sinnvoll das Verwenden von ‘‘virtuellen Fakten’’ (Fakten, die durch Befragen der Umwelt, z.B. des Benutzers, ermittelt werden; ein Beispiel dafür trafen wir oben an: `askuser`.)

3. *Tiefensuche:*

   - ‘‘monomanes’’ Verfolgen einer Hypothese (d.h. ein Unterziel erst dann aufgeben, wenn es erwieseneermassen aussichtslos ist - ohne Rücksicht auf die Kosten)
   - rein chronologisches Backtracken bei Fehlschlag
   - **Vorteil:** besonders *einfach* zu implementieren (siehe unten)
   - **Nachteile:**

     a. *nicht vollständig* (d.h. vorhandene Lösung wird nicht garantiert gefunden)

     b. in vielen Anwendungen (z.B. Computerlinguistik!) sehr *ineffizient*

...sehr einfach zu implementieren.

Aber: nicht vollständig

Beachte: Auswahl der Regeln von oben nach unten ist Resultat von Rechtsläufigkeit, nicht von Tiefensuche

© Michael Hess 2009

Beispiel für diese Kombination (d.h. exakt Prolog-Strategie):

```
p          :-     a.
p          :-     b, c(X), d(X).
a          :-     d(X).
c(X)       :-     g(X).
d(X)       :-     h, i(X).
d(X)       :-     k, l.
h          :-     b.
h          :-     x.
i(X)       :-     y(X).
i(X)       :-     g(X).
b.
g(2).
k.
l.
```

Ein Versuch, das graphisch darzustellen (0.L.6)

Drei Fragen zur absteigenden rechtsläufigen Tiefensuche:

1. inwiefern ‘‘*chronologisches*’’ Backtracken?
2. inwiefern ‘‘*nicht vollständig*’’*?*
3. inwiefern *ineffizient?*

‘Chronologisches’ Backtracken ist dumm,...

***ad:* chronologisches Backtracken:** Im Programm von oben

```
<...>
d(X)       :-     h, i(X).
d(X)       :-     k, l.
h          :-     b.
h          :-     x.
<...>
```

... ‘‘dependency directed backtracking’’ ist besser.

würde ‘chronologisches’ Backtracken nach einem Misserfolg von ‘i(X)’ zu einem zweiten Versuch führen, ‘h’ zu beweisen (via die zweite Regel für ‘h’). ‘Intelligentes’ Backtracking (z.B. ‘‘dependency directed backtracking’’) würde das nicht tun, da sinnlos (keine neuen Variablenbindungen; siehe unten).

© Michael Hess 2009

Tiefensuche kann...

***ad:* Unvollständigkeit der Tiefensuche:** Bei dieser Kombination (absteigend rechtsläufig) führen *linksrekursive Regeln* in unendliche Schlaufen:

```
adjp(adjp(Adjp,Adj))          -->     adjp(Adjp),
                                      adjektiv(Adj).
adjp(adjp([]))                -->     [].

adjektiv(adj(blau))           -->     [blau].
adjektiv(adj(gross))          -->     [gross].
adjektiv(adj(schwer))         -->     [schwer].

?-  adjp(Adjp,[blau,gross,schwer],[]).
```

...in Endlos-schlaufen führen.

```
<........  memory overflow>
```

Logisch äquivalent (Konjunktion ist symmetrisch):

```
adjp(adjp(Adj,Adjp))          -->     adjektiv(Adj),
                                      adjp(Adjp).
adjp(adjp([]))                -->     [].
```

Jetzt funktioniert's.

Ärgerlich, aber nicht schlimm, denn:

- jedes linksrekursive Prädikat lässt sich in ein rechtsrekursives umschreiben
- Resultat allerdings oft weniger intuitiv (künstliche Zwischenprädikate)
- den Output (z.B. beim Parsen) kann man aber so ''normalisieren'', dass die ''unnatürliche'' Reihenfolge nicht mehr sichtbar ist. Z.B.: statt

```
adjp(adj(blau),
     adjp(adj(gross),
          adjp(adj(schwer),
               adjp([]))))
```

also (z.B.)

© Michael Hess 2009

```
adjp([adj(blau),adj(gross),adj(schwer)]).
```

oder ggf. sogar so darstellen, wie wenn er bei Linksrekursivität entstanden *wäre* (nicht ganz trivial)

Prologs Beweisstrategie ist oft nicht effizient.

ad: **Effizienz:** Besonders deutlich bei der Syntaxanalyse.

Beispiel

the dog bites the man

über der Grammatik (DCG-Notation)

```
sent     -->     np, vp.
np       -->     det, adjp, cn.

det      -->     [the]
det      -->     [a].

cn       -->     [man].
cn       -->     [woman].
cn       -->     [rock].
cn       -->     [dog].
<...>
```

Hier wird die Hypothese 'dog = cn' zu beweisen versucht, indem man die *ganze* Liste von Substantiven von oben nach unten durchsucht, obwohl uns ja das Substantiv 'dog' vor Augen vorliegt - und dasselbe für jede einzelne Wortform!

Absteigende Verfahren sind hier ''blind für das Offensichtliche''.

5.3.1.2 Aufsteigende rechtsläufige Tiefensuche

Bottom-Up-Strategie: Günstig, wenn viele (oder schlecht definierte) mögliche Ziele und u.U. wenige (evtl. irrelevante) Fakten. Siehe auch das Beispiel oben.

Beispiele:

© Michael Hess 2009

1. Einen Todesfall aufklären
2. Objektidentifikation auf Distanz, Fehlerdiagnose aufgrund einer Beschreibung der Systemverhaltens
3. Syntaxanalyse: grosse Lexika

Prinzipielles Vorgehen:

'Saturieren' eines Systems

- entweder zyklisches 'Generieren von neuen Fakten' bis ''nichts mehr passiert'', d.h. 'Saturieren' des Systems (Rowe 1988:102)
- oder Definieren eines Zieles

Beispiel aus dem Bereich der regelbasierten Expertensysteme:

```
goal1      :-     fact1.              % R1
goal1      :-     a, b.               % R2
goal2      :-     c(X).               % R3
a          :-     not(d).             % R4
b          :-     d.                  % R5
b          :-     e.                  % R6
c(2)       :-     not(e).             % R7
d          :-     fact2, fact3.       % R8
e          :-     fact2, fact4.       % R9


fact2.
fact3.
```

Vorgehen im einzelnen beim Saturieren:

1. Für jedes vorhandene Faktum jene Regeln suchen, in deren RHS (irgendwo) Prädikate vorkommen, welche mit dem Faktum übereinstimmen (unifizieren).
2. Daraus eine zusätzliche, verkürzte Regel generieren und assertieren.
3. Wenn alte Regel redundant wird, entfernen.
4. Faktum als gebraucht markieren.
5. Negierte Terme für zuletzt aufsparen.

'fact2' (*erstes* Faktum) + Regeln R8, R9 → neue Regeln (''fact2'' gilt wegen der Rechtsläufigkeit als erstes Faktum)

© Michael Hess 2009

```
d          :-     fact3.     % R10
e          :-     fact4.     % R11
```

Beachte: Sind *zusätzliche* Regeln. Hier allerdings: Regeln 8 und 9 sind nun redundant; entfernen! ‘‘Focus of attention’’-Strategie: neues Material *oben* einfügen (d.h. Tiefensuche). ‘fact2’ als *gebraucht* markieren.

‘fact3’ (nächstfolgendes Faktum) + Regel R10 → neues Faktum

```
d
```

Regel 10 redundant; entfernen!

Dann:

‘d’ (nunmehr erstes Faktum) + Regel R5 → neues Faktum

```
b
```

Regeln 5 und 6 entfernen! **Beachte:** *alle* Regeln, deren LHS bewiesen worden sind, entfernen.

‘b’ + Regel 2 → neue Regel; **beachte:** ‘‘b’’ ist in Regel 2 in der RHS *nicht* an erster Stelle

```
goal1      :-     a.         % R12
```

Regel 2 als redundant entfernen!.

Zustand jetzt:

```
goal1      :-     a.                 % R12
e          :-     fact4.             % R11
goal1      :-     fact1.             % R1
goal2      :-     c(X).              % R3
a          :-     not(d).            % R4
c(2)       :-     not(e).            % R7

b
d
```

© Michael Hess 2009

Genauer: Rowe 1988:104

Nun die Negationen:

1. Regel 4 kann nie gelingen (‘d’ ist wahr)
2. da ‘‘e’’ *nicht* bewiesen ist, wird ‘not(e)’ als neues Faktum assertiert
3. daraus via Regel 7 ‘c(2)’ ableiten und assertieren. Regel 7 entfernen
4. via Regel 3 ‘goal2’ assertieren. Regel 3 *nicht* entfernen (enthält Variable)
5. nichts Neues kann abgeleitet werden

Bewiesene Fakten:

```
goal2.
c(2).
not(e).
b
d
```

Implementation: Rowe 1988:137

5.3.1.3 Aufsteigende rechtsläufige Breitensuche

Breitensuche ist

Vollständigkeit ist wichtig,...

... aber nicht gratis.

- *vollständig* (vergleiche dagegen Prolog-Strategie!)
- mit entsprechenden Verfeinerungen optimal (der *kürzeste* Weg wird gefunden)
- viel *schwieriger zu implementieren* (man muss Buch führen über alle angefangenen, aber noch offenen Äste des Suchbaums)
- Grundlage für viele Strategien, welche *Bereichswissen* verwenden, obwohl natürlich nicht darauf beschränkt (Bereichswissen beeinflusst oft die lokalen Entscheidungen)

Agenda...

Buchführen über die angefangenen Äste des Suchbaums (d.h. alle Zustände, deren Nachfolgezustände noch nicht generiert worden sind, schreibt man in eine *Agenda:*))

...als Schlange.

1. Für *Breitensuche* muss die Agenda als *Schlange* (queue) aufgebaut werden.

© Michael Hess 2009

2. *Implementation* der Agenda in Prolog:

"quick and dirty": Agendeneinträge assertieren

   a. Einfach (aber etwas gefährlich) ist es, jeden Eintrag zu assertieren, und zwar (für eine Schlange) am Ende der Datenbank (`assertz`);
   b. sauber: Liste durch alle Prädikate

Zyklen vermeiden!

3. Jede Assertion hat die Form `agenda(State,Path)`. `Path` wird verwendet, um
   a. zirkuläre Pfade zu vermeiden
   b. letztlich die Antwort zur Verfügung zu haben

4. zusätzlich eine Agenda ''ausgeschöpfter'' Zustände
   a. verhindert weitere zirkuläre Pfade
   b. stellt sicher, dass man den *kürzesten* Weg findet
   c. Implementation: `oldagenda(State,Path)`

5. Beschreibung der Welt: ebenfalls assertieren
6. Beschreibung des Zielzustandes: dito
7. in beiden Fällen: einfach, aber unschön; andere Lösungen?

Konkrete Implementation (0.L.7)

Man kann den gleichen Mechanismus auch für die aufsteigende Teifensuche verwenden:

Agenda als Stapel.

1. Agenda nunmehr als *Stapel* (stack) aufgebaut.
2. *Implementation* der Agenda in Prolog:
   a. ''quick and dirty'': Agendeneinträge am Anfang assertieren (`asserta`);
   b. sauber: Liste durch alle Prädikate

© Michael Hess 2009

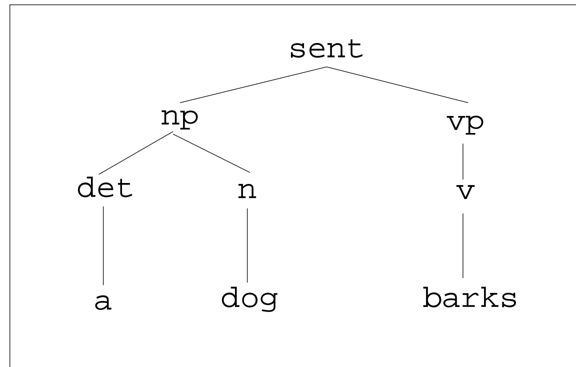
5.3.1.4 Absteigende rechtsläufige Breitensuche

Left to the reader as an exercise.

5.3.2 Einige gemischte Strategien

5.3.2.1 ‘‘Left-corner’’-Analyse

Eine in der Computerlinguistik beliebte Parsing-Strategie, das ‘‘Left-corner’’-Parsen (Pereira 1987, Covington 1994:158 ff.), ist ein typischer Vertreter einer gemischten Strategie. Als ‘‘linke Ecke’’ der Konstituente `sent` bezeichnet man in einer *Syntaxstruktur* wie



‘‘Linke Ecke’’ heisst zwei Dinge.

die Konstituenten `np` *und* `det` *und* das Terminalelement `a`. Ebenfalls als ‘‘linke Ecke’’ der Konstituente `sent` bezeichnet man die entsprechenden Elemente in den Regeln der *Grammatik*, also die fettgedruckten Terme in

```
sent(sent(Np,Vp))                   --->  np(Np,Number),
                                          vp(Vp,Number).

np(np(Number,Det,Adjp,N),Number)--->  det(Det,Number),
                                          adjp(Adjp),
                                          n(N,Number).

det(det(the),_)                     --->  [the].
```

© Michael Hess 2009

```
det(det(a),sing)                 --->  [a].

adjp([])                         --->  [].
adjp(adjp(Adj,Adjp))             --->  adj(Adj),
                                       adjp(Adjp).

vp(vp(Number,V,Np),Number)       --->  v(V,tr,Number),
                                       np(Np,_).
vp(vp(Number,V),Number)          --->  v(V,intr,Number).

n(n(dog),sing)                   --->  [dog].

n(n(man),plur)                   --->  [men].

v(v(eat),tr,sing)                --->  [eats].
v(v(eat),tr,plur)                --->  [eat].

adj(brown)                       --->  [brown].
```

Beachte: '--->' ist nicht '-->'.

"syntaktischer Zucker"

Es macht die Arbeit einfacher für den Parser, wenn man die Grammatik intern etwas anders repräsentiert, und zwar in der Form

```
sent(sent(A,B))      ==>     [np(A,C),vp(B,C)].
np(np(A,B,C,D),A)    ==>     [det(B,A),adjp(C),n(D,A)].
det(det(the),_)      ==>     [the].
<etc.>
```

Einige technische Details (0.L.8)

Der Parser sieht so aus; (man beachte: der Parser *selbst* ist im Format der DCG geschrieben - er ist eben ein Interpreter, während die üblichen DCG-Grammatiken intern zu einem Prolog-Programm *compiliert* werden):

© Michael Hess 2009

```
parse(Phrase)    -->  leaf(SubPhrase),
                      {link(SubPhrase,Phrase)},
                      lc(SubPhrase,Phrase).

leaf(Cat)        -->  [Word], {Cat ==> [Word]}.
leaf(Phrase)     -->  {Phrase ==> []}.

lc(Phrase,Phrase)--> [].
lc(SubPhrase,SuperPhrase)
                 -->  {(Phrase ==> [SubPhrase|Rest]),
                      link(Phrase, SuperPhrase)},
                      parse_rest(Rest),
                      lc(Phrase,SuperPhrase).

parse_rest([])   -->  [].
parse_rest([Phrase|Phrases])
                 -->  parse(Phrase),
                      parse_rest(Phrases).
```

Der vollständige Code ist ⇒hier.

Eine Ableitung (0.L.9)

5.3.2.2 ‘‘Iterative Deepening’’

Ebenfalls eine Art gemischter Strategie ist das Iterative Deepening.

Motivation: Absteigende Breitensuche ist zwar ‘‘besser’’ als absteigende Tiefensuche in dem Sinne, dass sie

1. *garantiert* eine Lösung findet, wenn es eine gibt (keine unendliche Schleifen)
2. den *kürzesten* Weg zur Lösung findet
3. uns erlaubt, *bereichsspezifisches Wissen* besonders einfach in der Suche zu verwenden

Aber sie ist sehr aufwändig (an Speicherplatz und Zeit und Programmkomplexität).

Da sich Tiefensuche so einfach implementieren lässt (und oft ja schon in den Interpreter ‘‘eingebaut’’ ist), kann man versuchen, das Beste beider Welten zu

© Michael Hess 2009

kombinieren:

1. man definiert eine *Suchtiefe:* T = 1 (z.B.)
2. man verwendet Tiefensuche *bis Tiefe T* ; wenn die Lösung gefunden wird, ist man fertig
3. sonst T um 1 erhöhen und zurück zu Schritt 2

Diese Methode simuliert gewissermassen Breitensuche mittels Tiefensuche und

"Simulation" der Breitensuche

- hat alle oben erwähnten Vorteile der Breitensuche

gar nicht so teuer

- ist nicht annähernd so verschwenderisch, wie es aussieht; ist sogar die *optimale* unspezifische Suchmethode
- kann ebenfalls mit der Verwendung bereichsspezfischen Wissens kombiniert werden

Näheres (mit einfacher Abbildung): Rich 1991:322

Weitere gemischte Strategien (0.L.10)

5.3.3 Strategien ohne Backtracking

Wenn beim Suchen an einer bestimmten Stelle mehrere Lösungen möglich sind, muss man sich darüber klar werden, was man tun will. Zwei Möglichkeiten wurden schon erwähnt:

zwei schon bekannte Methoden und...

1. eine *einzige* Lösung errechnen und im Bedarfsfall *backtracken* (übliche Methode bei Tiefensuche)
2. *alle* Lösungen errechnen und die ganze Menge von Lösungen zur Weiterverarbeitung weiterreichen (übliche Methode bei Breitensuche)

Beide Methoden sind aber nicht sehr effizient. Daher kann man eine der zwei folgenden Alternativmethoden versuchen:

...zwei neue.

1. eine (einzige) *kanonische* Lösung errechnen und ggf. lokal umformulieren
2. eine *unterspezifizierte* Lösung errechnen und ggf. inkrementell spezifizieren

   Beim Verwenden unterspezifizierter Lösungen kann man weiterhin unterscheiden

© Michael Hess 2009

a. Verwendung gepackter Strukturen

b. ‘‘Constraint’’-basierte Verfahren

5.3.3.1 Verwendung kanonischer Lösungen

Beispiel: Anschlussmehrdeutigkeit von Präpositionalphrasen. Die zwei Analysen von

2) The woman sees the man with the telescope

d.h. :

```
Parse 1 of 2:

sent(np(det(the)
        cnp([]
            noun(woman)
            []
            []
            []
            []))
     vp(cvp(verb(see)
            np(det(the)
               cnp([]
                   noun(man)
                   []
                   []
                   []
                   []))
            []
            []
            3+sing)
        pp(prep(with)
           np(det(a)
              cnp([]
                  noun(telescope)
                  []
                  []
                  []
                  [])))
        []
        3+sing))
```

© Michael Hess 2009

```
Parse 2 of 2:

sent(np(det(the)
        cnp([]
            noun(woman)
            []
            []
            []
            []))
     vp(cvp(verb(see)
            np(det(the)
               cnp([]
                   cnp([]
                       noun(man)
                       []
                       []
                       []
                       [])
                   pp(prep(with)
                      np(det(a)
                         cnp([]
                             noun(telescope)
                             []
                             []
                             []
                             []))))))
            []
            []
            3+sing)
        []
        []
        3+sing))
```

Anmerkung: `cnp` und `cvp` sind alternative Benennungen von `N'` und `V'` (Montague-inspiriert und typographisch einfacher).

Man kann nun entweder

‘‘kanonische’’ Lesart und backtracken

‘‘kanonische’’ Lesart und umformen

1. *eine* der extremen Lesarten zur kanonischen Lösung erklären (z.B. die zweite: maximal tief und maximal rechts) und ggf. backtracken
2. oder eine Regel definieren, wie man aus der kanonischen Struktur alle anderen zugelassenen Analysen ableiten kann

effizient, aber nicht allgemein

Beachte: Das Umformen geschieht *ohne* Backtracking. Aber: Es muss natürlich für jeden Typ von struktureller Ambiguität eine spezifische Regel vorhanden sein (eine für Pp-Attachment, eine für Nominalkomposita, eine für ... etc.).

© Michael Hess 2009

5.3.3.2 Gepackte Strukturen

Repräsentation von Unterspezifiziertheit:

Alternative: Man verschiebt die Entscheidung und stellt die Lösung *unterspezifiziert* dar.

''gepackte Strukturen'' oder constraint-basierte Verfahren

Wie man die Unterspezifiziertheit darstellt, unterscheidet die Verfahren mit gepackten Strukturen von den constraint-basierten Verfahren. Man kann die constraint-basierten Verfahren als Verallgemeinerung der Verfahren mit gepackten Strukturen ansehen.

Ein erster Ansatz beim Repräsentieren unterspezifizierter Lösungen, der v.a. im Bereich der Syntaxanalyse verfolgt wird, verwendet sog. ''gepackte Strukturen'' . Hierzu ein ± künstliches, aber einfaches und instruktives Beispiel (Russell 1995:701 ff.):

Das ''Ausmultiplizieren'' von Ambiguitäten...

3) Fall leaves fall and spring leaves spring

Hat vier Analysen: [14]

```
1. s(s(np(n(fall)
           n(leaves))
      vp(v(fall)))
    cj(and)
    s(np(n(spring)
           n(leaves))
      vp(v(spring))))

2. s(s(np(n(fall))
      vp(v(leaves)
           n(fall)))
    cj(and)
    s(np(n(spring))
      vp(v(leaves)
           n(spring))))
```

...kann sehr aufwändig werden.

14. Beachte: `S(Np(N(fall)),Vp(V(leaves),N(fall)))` ist die Lesart ''Autumn abandons autumn''

© Michael Hess 2009

```
3. s(s(np(n(fall)
           n(leaves))
        vp(v(fall)))
      cj(and)
      s(np(n(spring))
        vp(v(leaves)
           n(spring))))

4. s(s(np(n(fall))
        vp(v(leaves)
           n(fall)))
      cj(and)
      s(np(n(spring)
           n(leaves))
        vp(v(spring))))
```

Jeder der 2 Teilsätze ist 2-fach ambig. Bei *n* Teilsätzen ergäbe das 2^n Analysen (wenn man die Ambiguität der Konjunktion mal ausser Acht lässt).

Das kann kompakter dargestellt werden, und zwar als *nicht* ''ausmultiplizierte'' (eben: gepackte) Struktur:

''Nicht ausmultipliziert'' heisst ''gepackt''.

```
      -----------------               -------------------
     | S                 |           | S                   |
      -----------------               -------------------
     | s(np(n(fall)      |  -------  | s(np(n(spring)      |
     |      n(leaves))   | |Cj     | |      n(leaves))     |
     |   vp(v(fall)))    | |-------  |   vp(v(spring))))   |
s(   |-------------------| |cj(and)| |---------------------|   )
     | s(np(n(fall))     |  -------  | s(np(n(spring))     |
     |   vp(v(leaves)    |           |   vp(v(leaves)      |
     |      n(fall)))    |           |      n(spring))))   |
      -----------------               -------------------
```

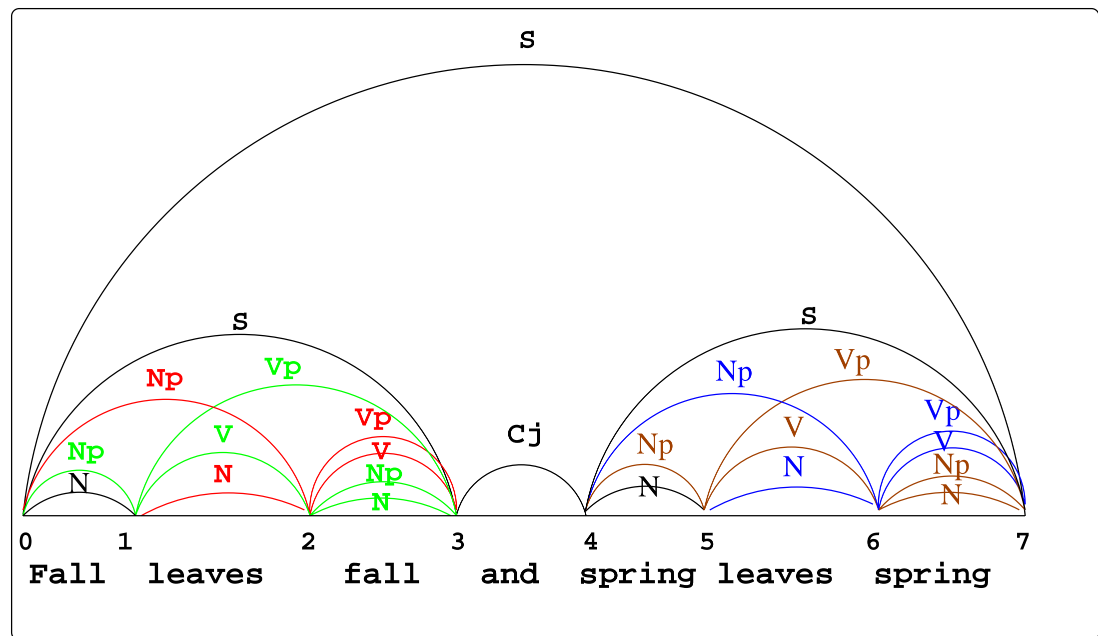
Die vier Bäume sind in eine *einzige* Struktur gepackt (einen sog. ''Wald'', da es ja vier Bäume sind). Man gibt also einerseits an, dass die *Gesamtstruktur* des Satzes `s( S Cj S )` ist, und anderseits, was der *innere Aufbau* jedes Teilsatzes `S` (und grundsätzlich auch von `Cj`) sein kann.

Hier ist noch kein besonders grosser Unterschied in der Komplexität zu sehen (immerhin: weniger als halb so umfangreich!), aber mit steigender Ambiguität der Teilsätze wird der ''Spareffekt'' immer deutlicher.

© Michael Hess 2009

Ein alter Bekannter: Chart

Dass dieses Verfahren, ambige Analysen in kompakter Weise darzustellen, überhaupt nicht ungewöhnlich ist, sieht man daran, das man die Analysen sehr übersichtlich auch als (passive) Chart-Struktur darstellen kann (die sich konkurrenzierenden Kanten sind in Farbe dargestellt; die Kanten sind nicht mit den Regeln etikettiert, durch die sie abgeleitet wurden):



Warum ‘‘packed forest’’? (0.L.11)

Implementation (0.L.12)

Wichtig: Damit spart man sich die explizite Auflistung aller Lesarten. Weil die meiste Zeit beim Parsen mit dem Aufbau der Strukturen verwendet wird, ist ein Algorithmus, der nur gepackte Wälder erzeugt, viel effizienter: Laufzeitverhalten bis zu diesem Punkt im schlimmsten Fall nur *polynomial:* $O(n^3)$ mit *n*=Inputlänge (und *O* konstant), statt exponentiellem Verhalten ohne gepackte Strukturen (wegen der $O(2^n)$ Syntaxstrukturen). $O(n^3)$ ist das beste, was man mit kontextfreien Sprachen erreichen kann. Also: Erst das Extrahieren *aller* Lösungen wird *exponentiell*.

© Michael Hess 2009

Direkte Verarbeitung gepackter Strukturen

Das heisst aber auch: Wirklich sinnvoll ist das Verfahren nur, wenn man die weitere Verarbeitung *direkt* über derartigen gepackten Strukturen operieren lässt.

Beispiele dafür:

- Wenn man z.B. weiss, dass die Kombination 'v+np' viel *unwahrscheinlicher* ist als die Kombination 'vp+pp', kann man die erste Lösung zur semantischen Auswertung verwenden, ohne die zweite ''auszumultiplizieren'' cf. dazu Carroll 1992
- Wenn man bestimmte Ambiguitäten nicht auflösen *muss* z.B. bei einer Übersetzung zwischen eng verwandten Sprachen, wo man sie unverändert übernehmen kann, kann man direkt die gepackten Strukturen verarbeiten. Hierzu wichtig (zur sog. ''Chart Translation''): Kay 1999.
- Wenn man Ambiguitäten nicht auflösen *kann*, kann man versuchen, sie direkt in der Semantik zu repräsentieren

5.3.3.3 ''Constraint''-Sprachen

Verallgemeinerung gepackter Strukturen

Im Umfeld v.a. der Informatik (und besonders der Logikprogrammierung) hat man Verfahren entwickelt, die man als Verallgemeinerung der spezifisch computerlinguistischen gepackten Strukturen betrachten kann.

Beispiel: Irgendwie möchte man das schreiben können

```
sent(np(det(the),np(noun(woman))),
     vp(verb(see),np(det(the),noun(man),W)),X),

    ( W = pp(prep(with),np(det(a),noun(telescope))), X=[]
    ;
      X = pp(prep(with),np(det(a),noun(telescope))), W=[] )
```

und die Entscheidung, welche der ''Abzweigungen'' das abarbeitende Programm nimmt, suspendieren können.

Das ist ein erster Schritt zum ''Constraint Satisfaction Programming''. Was ist das?

bisher: generate - test - backtrack

Bis jetzt war das Vorgehen beim Beweisen immer:

- sofort konkrete Werte für einzelne Variablen erraten (*generate*)
- und dann schauen, ob man die Lösung erhalten hat (*test*)

© Michael Hess 2009

- und wenn falsch: zurücksetzen (*backtrack*) (und neuen Wert erraten)

D.h. die *Operatoren* waren *Wertezuschreibungen*, der *Suchraum* war ein Raum von *Variablenbelegungen*.

Alternative:

nunmehr: propagate - distribute - backtrack

- schrittweise enger werdende *Einschränkungen* (''constraints'') für Variablenwerte aus den Ausgangsbedingungen erschliessen: **propagate**
- erst dann, wenn das nicht mehr möglich ist, blind raten und den geratenen Wert als neue Einschränkung behandeln: **distribute**
- nur innerhalb dieses (viel beschränkteren) Rahmens an geratenen Werten ggf. **backtrack**

D.h. die *Operatoren* sind derartige *Ableitungsschritte*, und der *Suchraum* ist der Raum von *Einschränkungen*.

Die Einschränkungen können

- extensional formuliert werden (*Mengen* von Variablenwerten) (cf.Rowe 1988:312)
- oder intensional (*Regeln*, d.h. Gleichungen, Ungleichungen etc., welche zwischen einzelnen Variablen bestehen müssen) (ausführliches Beispiel: Rich 1991:88)

Zum Konzept des ''Constraint programming'' (0.L.13)

Diese Überlegungen führten zur Entwicklung einer ganzen Reihe von neuen Sprachen. Prolog III und ⇒Prolog IV erweitert Prolog zu einer Constraint Programming Language (cf. Colmerauer 1987, Jourdan 1988). Auch Sicstus-Prolog hat eine CLP-Erweiterung.

Besonders wichtig ist die Möglichkeit, durch die Verwendung der Sprache ''CHR'' (''Constraint Handling Rules'') selbst einen Constraint Solver zu schreiben (statt sich auf einen der mitgelieferten beschränken zu müssen, wie bei den anderen Systemen). CHR baut auf Sicstus-Prolog auf und wird mit diesem mitgeliefert. Siehe dazu ⇒hier.

Weitere, ± eigenständige, Constraint-Sprachen sind Oz, Eclipse, CLP(Q), CLP(Q,R), CHIP u.a. Mehr dazu (allerdings nicht mehr ganz neu)
⇒hier (entfernt) und
⇒hier (lokal).

© Michael Hess 2009

Ein einfaches **Beispiel** für das Verhalten eines (in CHR geschriebenen) Constraint Solvers ist [15]

Constraints kombinieren sich zu...

```
Query:

X::[1,2,3,4,5,6,7], Y::[5,6,7,8,9], Y lt X.

Answer(s)
```

engeren Constraints und...

```
X::[6,7]
Y::[5,6]
Y lt X

yes
```

(Beispiel von der on-line Demo ⇒hier)

...ohne jedes Backtracken...

Hier wurden korrekterweise die Zahlen bis 5 aus der Liste *X* und die Zahlen ab 7 aus der zweiten Liste entfernt: Nur die Zahlen 5 und 6 für die Variable *Y* können die Constraints erfüllen (und nur die Zahlen 6 und 7 aus der ersten Liste werden dabei eine Rolle spielen) Man beachte: Es wird keine Sequenz von möglichen Werten der zwei Variablen ermittelt (wie das Prolog mit erzwungenem Backtracking tun würde), sondern eine neue Anzahl von (engeren) Constraints. Natürlich *könnte* man aus diesen drei Constraints die drei expliziten Lösungen, d.h. Variablenbelegungen ($Y=5 \wedge X=6$, $Y=5 \wedge X=7$, $Y=6 \wedge X=7$) errechnen, aber man muss es nicht.

Das kann man als ganz analog einer gepackten Struktur resp. einer Anzahl von Chart-Kanten betrachten. Wir hatten die Ambiguität eines Satzes oben so dargestellt:

```
edge(s,[vp,np],[],0,7)
edge(vp,[np,vt],[],1,7)
edge(vp,[pp,np,vt],[],1,7)
```

Daraus *könnte* man die zwei expliziten Lösungen

```
s(vp(np,vt),np)
s(vp(pp,np,vt),np)
```

15. Interpretation des Operators ‘‘::’’ ist ‘‘Element von (endlicher) Liste’’; ‘‘lt’’ heisst ‘‘less than’’

© Michael Hess 2009

errechnen, aber man muss es nicht.

Bei einer entsprechend weiter eingeschränkten Anfrage ergibt sich für eine der Variablen ein Constraint, der eine eindeutige Zuordnung *Y=5* erlaubt: [16]

...schliesslich zu konkreten Werten.

```
Query:

X::[1,2,3,4,5,6,7], Y::[5,6,7,8,9], Y lt X, Y ne 6.

Answer(s)

X::[6,7]
5 lt X
5 ne 6
Y=5
```

(Un-)Gleichungs systeme als Ergebnis

Wichtig ist also, dass als Ergebnis (Un-)Gleichungssysteme ausgegeben werden können, auf die man inkrementell immer engere Constraints ansetzen kann, bis am Schluss entweder nur noch eine einzige Lösung möglich ist oder man aus den verbleibenden Constraints die Lösungen ‘‘ausmultipliziert’’, wenn gewünscht.

CLP in der Computerlinguistik

Constraint Logic Programming ist für NLP allgemein sehr interessant, weil Ambiguitäten in der Sprache oft nur lokal sind:

Holzwegsätze

1. Beispiel Holzwegsätze

4) **The man raced past the barn complained**

Klassischer Fall lokaler Ambiguität: Zwei wohlgeformte Sätze, die aber nicht die ganze Kette abdecken (‘‘The man raced’’, ‘‘The man raced past the barn’’). Siehe auch weiter unten.

2. Beispiel Subkategorielle Unterspezifikation: (cf. Lehner 1990)

5) **Der Trainer schreit auf**

6) **Der Trainer schreit den Schiedsrichter an**

7) *** Der Trainer schreit an**

16. ‘‘ne’’ ist ‘‘not equal’’.

© Michael Hess 2009

Abtrennbare Verbsuffixe

Der Subkategorisierungsrahmen (d.h. obligatorische Komplemente) für ''schreit'' ist erst bekannt, wenn das abgetrennte Präfix erkannt ist. Dann aber ist er *eindeutig* bekannt. Statt blind eine vollständig spezifizierte Lösung zu erraten und ggf. zu backtracken, *verschiebt* man die Entscheidung, ob das Verb ''schreien'' oder ''aufschreien'' ist und erzeugt nur den Constraint-Raum, den man an die nachfolgenden Komponenten weiterprogagiert.

3. Beispiel: Strukturelle Unterspezifikation (Koller 1999, Pinkal 1999)

Strukturelle Unterspezifikation:

8) Every man loves a woman

mit den bekannten zwei Lesarten

(Vorläufig unauflösbare) Ambiguitäten...

9) $\forall$ M: man(M) $\rightarrow$ $\exists$ W: woman(W) $\wedge$ loves (M,W)

10) $\exists$ W: woman(W) $\wedge$ $\forall$ M: man(M) $\rightarrow$ loves (M,W)

Nun kann man aber schreiben:

X_1: $\forall$ M: man(M) $\rightarrow$ X_3
X_2: $\exists$ W: woman(W) $\wedge$ X_4
X_5: loves (M,W)

{X_0 *> X_1
X_0 *> X_2
X_3 *> X_5
X_4 *> X_5}

mit *> für ***Dominanz*** ($\approx$ enthält strukturell)
und X:A ''X hat Form A''

Die Meta-Variablen ***X*** werden oft ''handles'' (oder ''holes'') genannt. Sie rangieren über Knoten in der Struktur der LF.

Später mag desambiguierende Information einlaufen:

...und deren (spätere) Auflösung...

11) She is just gorgeous

Dann fügt man *inkrementell* hinzu:

...durch inkrementelles Spezifizieren

{X_4 *> X_1}

d.h. die LF von ''every man'' ist *enthalten* im zweiten Konjunkt der LF von ''a woman''.

''Inkrementell' heisst also, dass man keine vorher aufgebauten Strukturen (hier:

© Michael Hess 2009

Logische Formen) wegwirft. Die Menge der Logischen Formen wächst daher *monoton*.

Eine weitgehend analoge Art der Repräsentation unterspezifizierter Strukturen ist im Rahmen der sog. ''Minimal Recursion Semantics'' möglich. Hierzu siehe u.a. Copestake 1999.

Schliesslich sei auf die Parallelen dieser Darstellungen mit den ''Quasi-Logical Forms'' (QLF) hingewiesen. Die QLFs stellen ⇒bekanntlich ebenfalls unterspezifizierte Skopusstrukturen (aber nur für Quantoren) dar. Dort wird als Meta-Sprache die *Syntax* der natürlichen Sprache verwendet: Die syntaktischen Elemente beschreiben die möglichen Beziehungen zwischen den *(objekt-)logischen* Sprachelementen.

Lexikalische Ambiguitäten

4. Beispiel: Lexikalische Unterspezifikation (Pinkal 1999:2)

12) John began the book

kann heissen

1. began to read
2. began to write
3. began to paint (?)
4. began to eat (??)
5. etc.

Daher *vorerst:*

X_0: begin(john,X_1), X_2: the_book

{X_1 *> X_2}

und dann evtl. eines von

X_1: writing_of(X_2)
X_1: reading_of(X_2)
<etc.>

© Michael Hess 2009

G

5.4 Bereichsspezifische Suchstrategien

Universalität steht meist im Widerspruch...

Nachteil aller *universellen* Strategien:

- zwar relativ einfach zu implementieren
- aber potentiell ineffizient (z.T. nicht einmal vollständig! Beispiel: Prolog)

...zur Effizienz.

Beispiel: Weg ins Zentrum einer unbekannten Stadt finden.

- Universelle Strategie: Bei Abzweigungen *immer* die linke nehmen (linksläufig), und vorwärtsschreiten, solange es irgendwie geht (Tiefensuche)
- Bereichsspezifische Strategie: Jene Abzweigung nehmen, welche tendenziell in Richtung Hochhäuser geht

Universelle Suchstrategien sind oft manifest schlecht.

Ziemlich klar:

- Das unspezifische Vorgehen wird hier fast immer *schlechter* sein (und meist sogar in unendliche Schlaufen steuern).
- Das spezifische Vorgehen wird manchmal vom Idealpfad abweichen, aber nur *vorübergehend*.

Kontrollwissen

Heuristisches Wissen ist...

...nicht-numerisch und...

...lokal.

Terminologisch recht unklare Verhältnisse:

1. bereichsspezifisches *Kontroll*wissen heisst oft ‘‘heuristisches Wissen’’. Bereichsspezifische Verfahren werden oft ‘‘informierte’’ Suchverfahren genannt.
2. oft wird der Begriff ‘‘heuristisches Wissen’’ aber auch allgemein im Sinne von ‘‘Faustregel’’ verwendet (‘‘oft, aber nicht immer erfolgreich’’)
3. heuristisches Wissen ist in der Regel *nicht-numerisch* (also z.B. ‘‘Operator A ist besser als B’’)
4. heuristisches Wissen ist *lokal:* betrifft nur den nächsten Zug

Heuristiken machen oft widersprüchliche oder unklare Empfehlungen:

```
''Operator B ist besser als A und
  Operator C ist besser als A''.
```

Dann sind numerische Vergleiche erforderlich:

© Michael Hess 2009

```
A = 5, B = 3, C = 1, daher: A > B > C
```

(*kleiner* heisse hier *besser*)

Deshalb: andere Arten bereichsspezifischen Wissens, z.B.

Evaluationsfunktion

Kostenfunktion

a. Evaluationsfunktionen (lokal: noch vermutete Entfernung vom Ziel)
b. Kostenfunktionen (lokal: Kosten des schon Geleisteten)
c. **auch:** Typologie von Problemen (global: dient zur Aufspaltung in Teilprobleme)

Im folgenden hauptsächlich bereichsspezifisches *Kontroll*wissen.

5.4.1 Tiefensuche + Evaluationsfunktion = ‘‘Hill-Climbing’’

Eine wichtige Sorte numerischer Bewertung ist die Evaluationsfunktion: Um wieviel besser ist ein Folgezustand *vermutlich* im Vergleich zum gegenwärtigen Zustand?

Beachte: ‘‘vermutlich’’. Sonst wäre Suche unnötig.

Scheinbar naheliegende Strategie

Heisst ‘‘Hill-climbing’’, weil man immer den steilsten Weg (den scheinbar am direktesten nach oben, zum Ziel führenden Weg) versucht. Auch: ‘‘diskrete Optimierung’’ (schrittchenweise optimieren).

Beispiele:

1. Bergsteigen im Nebel: exakt die obige Strategie des steilsten Anstiegs verfolgen
2. Wegsuche in unbekannter Stadt: jenen Nachfolgezustand wählen, der bezügl. der Luftlinien-Distanz *näher* beim Zielzustand ist
3. Computerlinguistik: immer die lokal wahrscheinlichste Lesart einer Mehrdeutigkeit wählen:

Funktioniert oft recht gut...

13) The horse raced ...

‘‘to race’’ ist syntaktisch zweideutig: Intransitiv (‘‘rennen’’) oder transitiv (‘‘jemanden hetzen’’). Aber: als transitives Verb ist ‘‘to race’’ sehr selten. Man wählt daher sofort die Lesart

13b) Das Pferd rannte ...

und wenn der Satz dann weitergeht

© Michael Hess 2009

13a) The horse raced past the barn.

sieht man, dass diese Strategie (hier!) korrekt und effizient war.

13 ist ein gutes Beispiel für *lokale* Mehrdeutigkeit.

Die Strategie funktioniert oft auch bei *globaler* Mehrdeutigkeit:

14) I saw that gas can explode

mit den zwei Lesarten

14a) Ich sah, dass Benzin explodieren kann

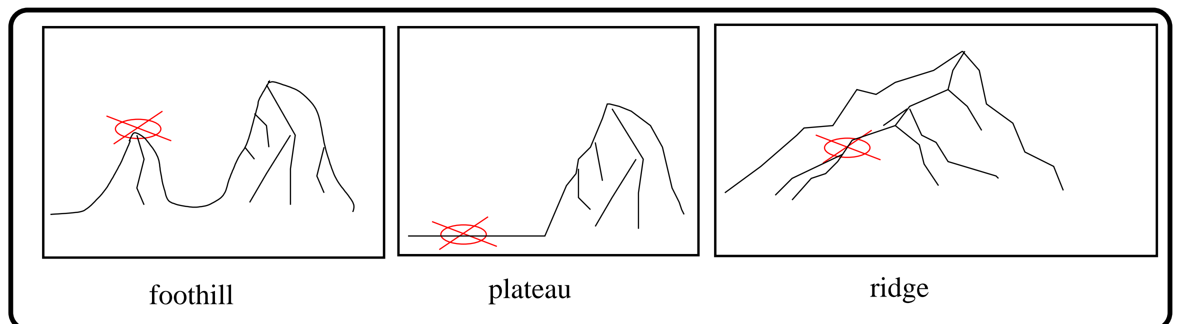
14b) Ich sah diesen Benzinkanister explodieren

14a ist die deutlich bevorzugte Lesart: ‘‘saw that$_{\texttt{SubordKonj}}$’’ ist deutlich wahrscheinlicher als ‘‘saw that$_{\texttt{DemonPron}}$’’.

...aber bleibt manchmal stecken.

Nachteil des Hügelkletterns: Man gerät manchmal in eine lokal gute, aber global schlechte Position und kommt nicht mehr davon weg:

1. Zwischenhügel (‘‘foothill problem’’)
2. Plateau
3. Bergrücken



Computerlinguistisches Beispiel für 1:

‘‘garden path sentences’’

a. ‘‘Holzwegsätze’’ (‘‘garden path sentences’’):

15) The horse raced past the barn fell

Die Lesart von oben ist falsch. Richtig ist

15a) Das Pferd, das an der Scheune vorbeigehetzt wurde, fiel

(Siehe auch oben).

© Michael Hess 2009

Man wählt aber automatisch die erste Lesart, und bleibt dann (fast unrettbar) stecken.

b. In Figur Rowe 1988:202, mit G als Ziel, ergibt sich die Reihenfolge S:12, A:7, E:4. Ist eine Sackgasse.

Computerlinguistisches Beispiel für 2:

16) I saw the man with the telescope run away

Der Satzanfang

16a) I saw the man with the telescope

ist zweideutig: Der Mann hat ein Teleskop, oder ich sehe ihn mittels des Teleskops. Der Rest des Satzes macht die erste Lesart obligatorisch, aber nach

16b) I saw the man

d.h. an der Stelle, wo man die Anschlussart bestimmen sollte, gibts keinen offensichtlichen ‘‘Anstieg’’: Beide Lesarten sind (an dieser Stelle!) gleich gut, und man kommt (*allein* mit der Hügelklettermethode) nicht mehr weiter.

Computerlinguistisches Beispiel für 3: schwer zu finden.

Lösungsmethoden:

drei wichtige Lösungsansätze

1. Zwischenhügel: ‘‘schlechte’’ Schritte unternehmen
2. Plateau: ‘‘irrelevante’’ Schritte unternehmen
3. Bergrücken: ‘‘Makro-Schritte’’ finden

5.4.2 Breitensuche + Evaluationsfunktion = ‘‘Best-First’’-Suche

Die Nachteile des Hill-climbing können vermieden werden, wenn man Breitensuche mit Evaluationsfunktion verwendet.

Verbreitete Modifikationen:

1. nicht nur ungetestete Zustände auf der gleichen Ebene, sondern *irgendwelche* noch nicht getestete Zustände werden ausprobiert
   Daher springt dieses Verfahren häufig sehr im Suchraum herum, z.B. von D/E nach C; Beispiel adaptiert von Rowe 1988:202; siehe Bild unten.

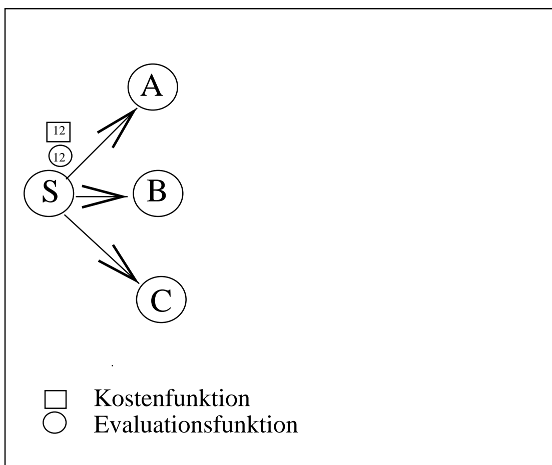
© Michael Hess 2009

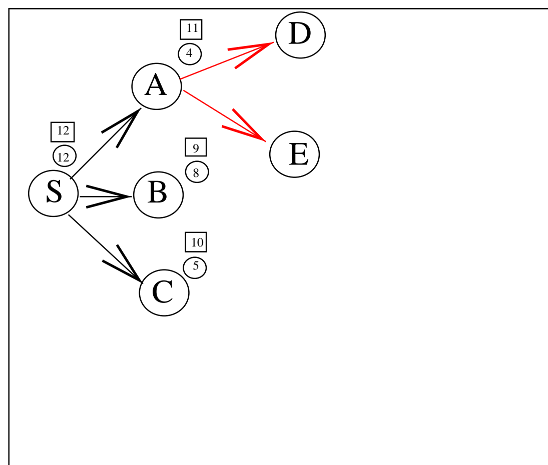
‘‘beam search’’

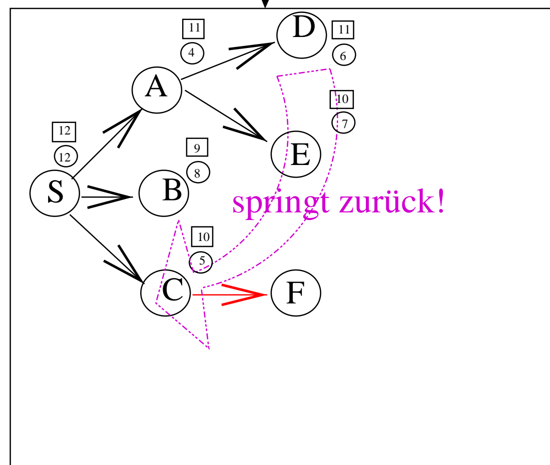
2. ‘‘beam search’’: nur die aussichtsreichsten Zustände kommen überhaupt in die Agenda

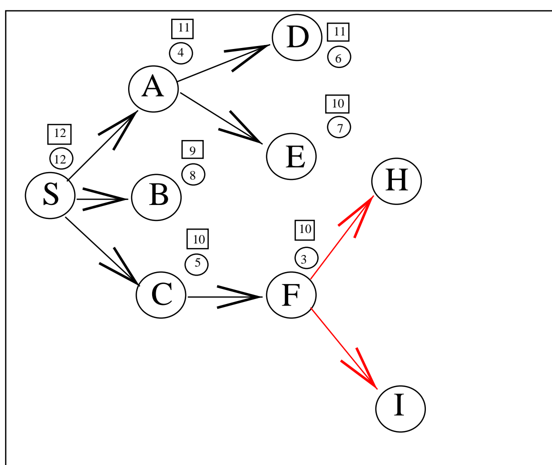
© Michael Hess 2009

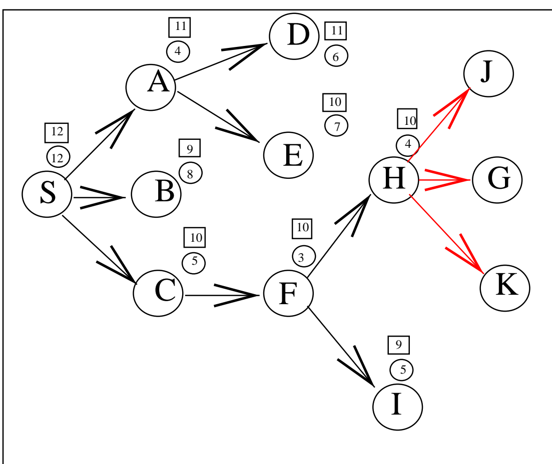
Suchreihenfolge (im Fall 1):

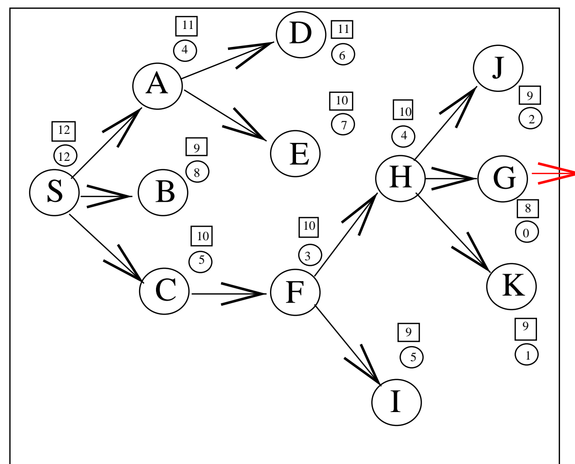












© Michael Hess 2009

```
1. Schritt: S erweitern zu A,B,C (keine echte Wahl)
2. Schritt: Evaluationsfunktionen nachschauen; A hat tiefste;
   A erweitern zu D,E
3. Schritt: Evaluationsfunktionen nachschauen;
   C hat tiefere als D,E,B; C erweitern
4. Schritt: F hat tieferen Wert als D,E,B; F erweitern
5. Schritt: H  hat tieferen Wert als D,E,B; H erweitern
6. Schritt: G hat tieferen Wert als D,E,B,I.
   Da Wert=0, ist Ziel erreicht
```

Woher kommt die Evaluationsfunktion?

Aber: Wie kommt man überhaupt auf die Werte der Evaluationsfunktion? Letztlich immer empirisch. Heisst:

1. Statistik
2. Befragen von Experten
3. Experimentieren

Z.B. aus der Analyse von Corpora

Computerlinguistische Beispiele: Statistiken über syntaktisch mehr oder weniger analysierten (!) Corpora ergeben,

1. wie oft eine Wortform in eine bestimmte Wortkategorie fällt, z.B. wie oft ‘‘can’’ Substantiv, wie oft es Hilfsverb ist (*absolute* Wahrscheinlichkeit)
2. wie oft bestimmte Wortkategorien *aufeinanderfolgen* (*bedingte* Wahrscheinlichkeit)
3. wie oft bestimmte Konstituenten in einer bestimmten Art *kombiniert* werden, z.B. wie oft ‘‘to race’’ transitiv verwendet wird
4. und natürlich Kombinationen hiervon:

Kombination von Werten: schwierig.

   a. absolute Wahrscheinlichkeit, dass ‘‘that’’ Relativpronomen ist (eher hoch, z.B. 0.7) vs. dass es Determinator ist (eher tief, z.B. 0.3)
   b. absolute Wahrscheinlichkeit, dass ‘‘can’’ Substantiv ist (tief, z.B. 0.1) vs. dass es Hilfsverb ist (hoch, z.B. 0.9)
   c. bedingte Wahrscheinlichkeit, dass ‘‘saw that’’ Verb+Relativpronomen ist (recht hoch, z.B. 0.8) vs. dass ‘‘that can’’ Determinator+Substantiv (recht niedrig, z.B. 0.2) ist
   d. ergibt z.B. $0.7+0.9+0.8/3 = 0.8 > 0.2$ (Verrechnung im realen Fall nicht so einfach; siehe unten)

Corpusanalyse ist zur Zeit ein sehr beliebtes Forschungsthema. Aber: Je ‘‘höher’’ man steigt,

© Michael Hess 2009

- desto schwieriger ist es, entsprechendes Rohmaterial für Statistiken zu bekommen
- aber desto leichter wird es offenbar, relativ brauchbare Werte allein durch Herumprobieren und Befragen von Experten (resp. Introspektion) zu erhalten.

5.4.3 Breitensuche + Kostenfunktion = ‘‘Branch-and-bound’’-Suche

Anderes Mass: Kosten des *zurückgelegten* Wegs

‘‘zurückschauen’’ statt ‘‘vorwärtsvermuten’’

- nicht vorher geschätzt, sondern nachher gemessen (zuverlässiger)
- erlaubt (in der Breitensuchversion), den garantiert optimalen Weg zu finden
- aber vielleicht nicht sehr schnell (‘‘schaut’’ eben nur ‘‘rückwärts’’; springt deshalb viel herum)

Beispiele für Kostenfunktion:

1. reine Wegsuche: effektiv zurückgelegte Distanz
2. ‘‘erschwerte’’ Wegsuche z.B. in unbekannter Stadt: effektiv zurückgelegte Distanz, aber um einen bestimmten Faktor vergrössern pro Hindernis (z.B. gekreuzte Strasse, Ampel oder Stopzeichen
3. allgemein: verbrauchte Ressourcen (Zeit, Energie, Material): ‘‘resource bounded inferencing’’
4. Computerlinguistik: Art der aufgebauten Syntaxstruktur. Siehe gleich unten.

Zwei besonders wichtige Beispiele für Kostenfunktionen in der Computerlinguistik sind zwei generelle Prinzipien, um bei den Mehrdeutigkeiten des Präpositionalphrasenanschlusses die wahrscheinlichere Lesart zu ermitteln:

1. Maximierung der Tiefe der Struktur: im Englischen Bevorzugung der ‘‘Rechtsassoziation’’ (‘‘Right Association’’) oder ‘‘Late Closure’’; (ursprünglich Kimball 1973, modifiziert durch Frazier und Fodor in Fodor 1980). Das würde im Fall des Beispiels (wiederholt)

   2) The woman sees the man with the telescope

   bedeuten, dass man ‘‘the man’’ durch die Präpositionalphrase (‘‘with the telescope’’) modifiziert und nicht das Verb.

© Michael Hess 2009

Minimalanschluss

2. Minimierung der Anzahl der nicht-terminalen Knoten in der Struktur: Das Prinzip des ‘‘Minimalanschlusses’’ (‘‘Minimal Attachment’’); Frazier und Fodor in Frazier 1978.

Dem Vernehmen nach gibt es psycholinguistische Evidenz für beide Prinzipien: Beide Prinzipien minimieren die nötige Gedächtnisleistung

- Rechtsassoziation, weil man sich nicht daran erinnern muss, dass da noch eine Stelle ist, an der man die Präpositionalphrase anhängen kann
- Minimalanschluss, weil man einfach weniger Struktur aufbauen (und speichern) muss

Frage: Was tun, wenn verschiedene Kostenfunktionen in Konflikt geraten?

ein auch andernorts bekanntes Problem

- Beispiel aus der Computerlinguistik: Im Beispiel 2 oben zieht das Prinzip der Rechtsassoziation die zweite Lesart vor, das Prinzip des Minimalanschlusses das erste
- Die Situation ist in der *Informatik* wohlbekannt, z.B. der ‘‘space/time trade-off’’ (compilierte Programme laufen z.B. schneller, als uncompilierte, sind aber viel grösser, als diese).

  Daher: Keine *allgemeine* Lösung, sondern Entscheid auf der Basis der verfügbaren Ressourcen.
- Situation in der *Computerlinguistik* etwas anders: Wir müssten das Verhalten des Menschen modellieren, um die Maschine die gleichen Entscheidungen wie der Mensch zu treffen zu lassen.

  Daher: Auch keine *allgemeine* Lösung; es wäre Evidenz aus der Psycholingustik gefragt.

Zusätzliches Problem im konkreten Fall oben:

Minimalanschluss ist sensitiv bezüglich Grammatiktheorie

- Bestimmte Prinzipien sind sehr abhängig von der verwendeten Grammatiktheorie (z.B. Minimalanschluss). Wenn eine etwas simplere Grammatik als die oben verwendete (u.a. ohne ‘‘cnp’’ und ‘‘cvp’’) eingesetzt wird, haben beide Syntaxstrukturen gleich viele Knoten (nämlich 15).:

© Michael Hess 2009

```
sent(np(det(the)
        np(noun(woman)
           []))
     vp(verb(see)
        np(det(the)
           noun(man)
                pp(prep(with)
                np(det(the)
                   noun(telescope)
                   []))))
        []))
```

```
sent(np(det(the)
        np(noun(woman)
           []))
     vp(verb(see)
        np(det(the)
           noun(man)
           [])
        pp(prep(with)
           np(det(the)
              noun(telescope)
              []))
```

- Die Frage ist also, ob ein solches Prinzip überhaupt psycholinguistisch verifizierbar ist, da wir ja nicht wissen, wie ‘‘die’’ Grammatik des Englischen (Deutschen, Französischen, ...) aussieht. Aber wenn man sich einmal auf eine Grammatik festgelegt hat, kann das Prinzip sehr wohl nützlich sein (siehe unten).

5.4.4 Breitensuche + Evaluationsfunktion + Kostenfunktion = ‘‘A*’’-Suche

Alles Gute kombiniert ergibt den A*-Algorithmus:

1. Breitensuche, mit

© Michael Hess 2009

2. Evaluationsfunktion, summiert mit
3. Kostenfunktion

Beispiele:

1. Wegsuchen: Rowe 1988:204
2. Computerlinguistik: Auflösen von Mehrdeutigkeiten .

‘‘LogDoc’’: ein (sehr) experimentelles (und †) Satzretrievalsystem

Konkretes System: ‘‘LogDoc’’

1. gewisse Arten von Strukturen werden bevorzugt, d.h. zwei *Kostenfunktionen* werden minimiert, und zwar die oben erwähnten:
   a. stärker: tief eingebettete Strukturen
   b. weniger stark: Strukturen mit wenig Knoten

   Die empirischen Werte werden als Fakten festgehalten:

```
level_reward(1.25).
node_penalty(2).
```

2. gewisse Kombinationen von Konstituenten werden durch (empirisch ermittelte) *Evaluationsfunktionen* bevorzugt (höhere Werte sind hier ‘‘besser’’):

```
preference(verb_pp,4).
preference(post_adjcts,2).
```

3. wegen der *Kombination* von Kosten- und Evaluationsfunktion wird z.B. nicht einfach die verbale der nominalen Modifikation vorgezogen.
4. Breitensuche ist in der Form eines Chart-Parsers implementiert
5. nur die beste(n) Lesart(en) werden durchgelassen.

$$V_\alpha = \frac{\sum_{i=1}^{i=n} V_i + \mathbf{Pen}}{\mathbf{Rew}} - Spec_\alpha$$

Equation 1. Berechnung der Präferenzwerte

wobei gilt:

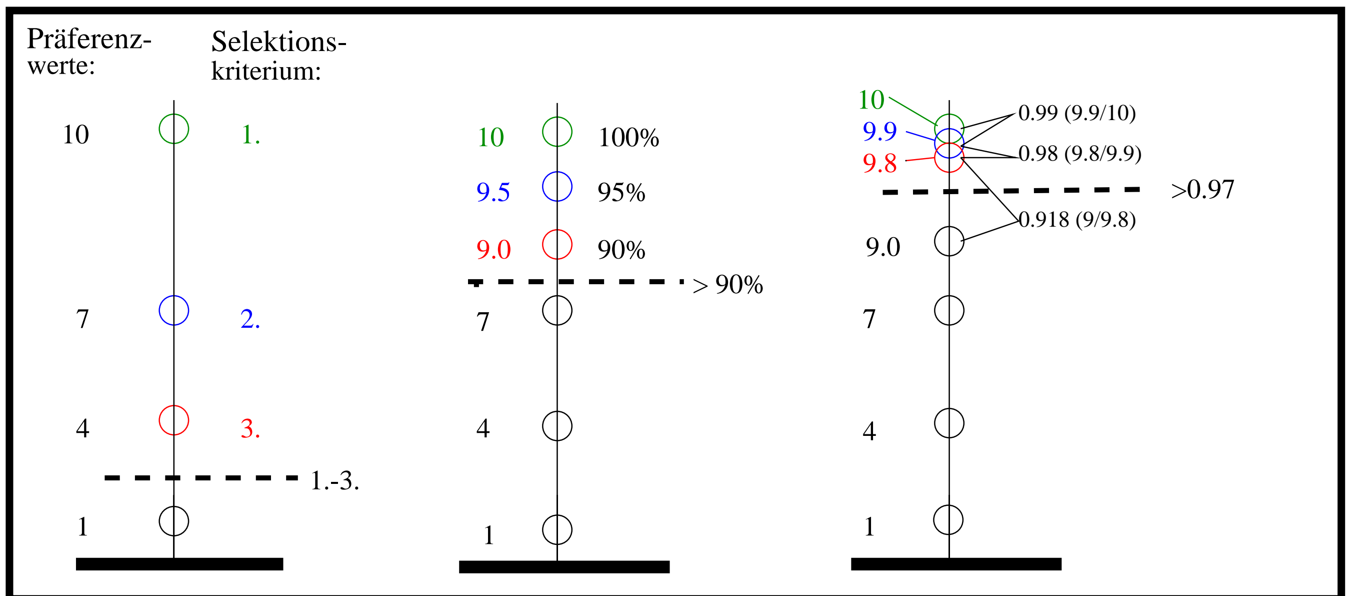
© Michael Hess 2009

- **Pen** ist die ''Bestrafung'' zusätzlicher Knoten
- **Rew** ist die ''Belohnung'' für Einbettung
- $Spec_{\alpha}$ ist der Wert der *spezifischen* Präferenzwerte der Konstituente α
- V_i sind die Präferenzen der Teilstrukturen von α (rekursiv aus deren Teilstrukturen errechnet)

Arten von Grenzwerten

Problem: Wo setzt man den Grenzwert für das Ausfiltern/Auslichten?

- offensichtlich nicht brauchbar: *Rangfolge*
- auch nicht genügend: *Prozentanteil*
- besser: relativer *Abstand*



Annahme: Die konkreten Präferenzwerte werden normalisiert (d.h. auf eine Skala von 1 bis 10 Punkten abgebildet: Der höchste Wert ist immer 10).

© Michael Hess 2009

Man muss v.a. die spezifischen Werte beim Parsen errechnen. Wie?

Z.B. durch explizites Einfügen entsprechender nicht-inputkonsumierender Tests:

Ausschnitt aus der Grammatik:

```
% postmodifying pp-adjuncts:
cnp(V,cnp(V,[],CNp,Pp),Nr)  -->
                cnp(V0,CNp,Nr),
                pp(V1,Pp),
                {preference(post_adjcts,Pf),
                 compute(V0,V1,Pf,V)}.

% modification of verbs by pp:
cvp(V,cvp(V,Verb,Np1,Np2,Nr),Nr) -->
                verb(V0,Verb,2-tr,Nr,Tns),
                opt_np(V1,Np1,_),
                {verb(Vb) = Verb,
                 transitivity(Vb,2-tr,Prep)},
                 opt_pp(V2,pp(V2,prep(Prep),Np2)),
                {preference(verb_pp,Pf),
                 compute(V0,V1,V2,Pf,V)}.

preference(verb_pp,4).
preference(post_adjcts,2).
```

Die technischen Details sind wenig wichtig.

Die Verrechnung geschieht dann anhand der Definition von ‘‘compute’’ und der schon erwähnten Parameter, welche die allgemeinen Prinzipen des Minimalanschlusses und der Rechtsassoziation implementieren):

Ergebnisse: Wenn man die unwahrscheinlichen Lösungen ausfiltert

© Michael Hess 2009

```
1200: logic for natural language analysis

cnp(5.47583
    []
    cnp(2.4
        []
        noun(logic)
        []
        []
        [])
    pp(2.4448
       prep(for)
       np(2.4448
          []
          cnp(1.056
              []
              noun(analysis)
              []
              cnp(-1.68
                  adjp(1
                       []
                       adjective(natural))
                  cnp(2.4
                      []
                      noun(language)
                      []
                      []
                      [])
                  [])
              []))))

Corresponding DB entry is:

object(language,sk-7)/1200.
object(logic,sk-8)/1200.
property(natural,sk-7)/1200.
role(beneficiary,sk-9,sk-8)/1200.
object(analysis,sk-9,sk-10,sk-7)/1200.
```

Ein Satz, wo *keine* Lesarten ausgefiltert werden können, d.h. ein Fall *genuiner* Mehrdeutigkeit, ist das Beispiel 2 von oben .

Die Resultate (0.L.14)

Anderes Beispiel für Best-First-Suche: Interpretation von Nominalkomposita (cf. oben und Hobbs 1988):

“lube-oil alarm”

Man erinnere sich: Man muss unterscheiden zwischen:

1. Erkennungswahrscheinlichkeit (dass dies ein Nominalkompositum ist; hier: *hoch*)
2. Qualität der Erklärung (bei der Verwendung des “Dummy”-Prädikats `nn/2`: *gering*)

Die zwei Werte entsprechen der Kostenfunktion (Kosten der Annahme) und der

© Michael Hess 2009

G

Evaluationsfunktion (Kosten der Erklärung) im A*-Suchverfahren.

Adversative Suchstrategien (0.L.15)

6. Logik-basierte Problemlösung

6.1 Mehr Flexibilität: Meta-Interpretion in logik-basierten Systemen

Rigide Strategien haben Nachteile.

Die bisher besprochenen Strategien waren sehr unflexibel: Die im Beweiser implementierte Strategie (mit oder ohne Verwendung bereichs-spezifischen Wissens) wurde unerbittlich angewendet.

Das ist oft ungünstig:

- Oft ist es sinnvoll, die Abarbeitungsstrategie dynamisch zu ändern (z.B. in Abhängigkeit von verfügbaren Ressourcen) Dies gilt besonders für die Verwendung von Wissen über den Anwendungsbereich, aber in vielen Fällen auch *ohne* solches Wissen.

- Beim *Entwerfen* von Strategien zuerst auf hoher Abstraktionsebene experimentieren, erst dann auf Effizienz achten

Daher:

Der Meta-Interpreter verwendet Meta-Regeln und erzeugt den Kontrollfluss

- Man verwendet statt eines Interpreters einen *Meta-Interpreter*,
- der gemäss *Meta-Regeln* (und Meta-Fakten) vorgeht, welche
- den Kontrollfluss beim Abarbeiten eines Objekt-Programms definieren,
- sodass eine lokale Veränderung der Meta-Regeln die globale Abarbeitungsstrategie verändert.

Meta? Was Meta?

Warum *Meta*-Interpreter und *Meta*-Regeln?

- Ein normaler Interpreter verarbeitet die Ausdrücke eines (Objekt-)Programms (Regeln und Fakten) gemäss einer ‘‘fest verdrahteten’’ Abarbeitungsstrategie.

© Michael Hess 2009

- Ein Meta-Interpreter verarbeitet hingegen die Ausdrücke eines Meta-Programms (also *spezielle* Fakten und Regeln — eben: Meta-Regeln und Meta-Fakten —, welche vom Objekt-Programm ganz unabhängig sind), welche sich *auf die Ausdrücke des (Objekt-)Programms beziehen* (und *daher* ''Meta'').

Zur besseren Verständlichkeit vorab ein

Beispiel: Im Prinzip verwendet man z.B. absteigende *rechtsläufige* Tiefensuche, *aber*

1. Die Auswahl der nächsten zu verwendenden *Regel* des Objekt-Programms kann durch Veränderung von Meta-Regeln und Meta-Fakten beeinflusst werden
2. Die Auswahl des nächsten zu verwendenden *Terms* innerhalb einer gewählten Regel kann in analoger Weise beeinflusst werden

Allgemein gilt:

- Es gibt in der Informatik *generell* einen ''Trade-off'' zwischen Effizienz und Flexibilität, und so auch hier (Figur: Rowe 1988:100). Siehe gleich unten.
- Alle folgenden Überlegungen sind *nicht* auf logik-basierte Systeme beschränkt. Wir werden aber nur Beispiele aus dem Kontext der Logikprogrammierung verwenden.

Im einzelnen gilt:

Effizienzverlust

- **Nachteil** der Meta-Interpretation: *Geringere Effizienz* (Faustegel im Kontext der Logikprogrammierung: ca. Faktor 10)
- **Vorteile** der Meta-Interpretation:

Gewinn an:

Flexibilität

Verständlichkeit

Einflussmöglichkeit

  a. *grössere Flexibilität:* man ändert bloss die Form einer Meta-Regel, und schon kann man eine völlig andere Art der Abarbeitung beobachten
  b. *grössere Verständlichkeit:* das Verfahren wird oft zu besser verständlichen Regelsystemen führen: Der Mensch scheint sehr oft auf dieser Ebene der Abstraktion zu denken.
  c. grössere *Einflussmöglichkeit:* man kann z.B. einen eigenen Tracer bauen, der einem sehr differenziertes Interagieren mit dem Beweisablauf erlaubt

- der Nachteil ist oft behebbar: *wenn* die Sache einmal läuft, compiliert man die Sache in ein ''normales'' Programm (d.h. ein Objekt-Ebene-Programm)
- logikbasierte Sprachen sind für die Meta-Interpretation deshalb besonders geeignet, da Programme und Daten hier formal nicht unterschieden werden (funktionale Sprachen sind auch nicht schlecht)

© Michael Hess 2009

6.1.1 Implementation von Meta-Interpretern in der Logik-Programmierung

Grundsätzlich sind Interpreter (und Compiler) in Prolog sehr einfach zu schreiben. Besonders einfach, wenn die Tiefensuche des System-Interpreters bewahrt wird. Der primitivste Meta-Interpreter:

Vollständiger Code ⇒hier

Primitiver geht's nimmer.

```
prove([]).
prove([G|Gs])    :-     prove_one(G), prove(Gs).

prove_one(G)     :-     G. % ← f. Systemprädikate
prove_one(G)     :-     cl(G,SGs),
                        prove(SGs).
```

Dieser Interpreter setzt voraus, dass die Klauseln des zu interpretierenden Programms *anders* geschrieben sind, als jene des Interpreters selbst. Statt

```
grandfather(X,Z)        :-      father(X,Y),
                                parent(Y,Z).
parent(X,Y)             :-      father(X,Y).
parent(X,Y)             :-      mother(X,Y).

father(peter,john).
father(john,jim).
father(bob,mary).
mother(mary,jill).
```

nunmehr

© Michael Hess 2009

```
cl(grandfather(X,Z),[father(X,Y),parent(Y,Z)]).
cl(parent(X,Y),[father(X,Y)]).
cl(parent(X,Y),[mother(X,Y)]).

cl(father(peter,john),[]).
cl(father(john,jim),[]).
cl(father(bob,mary),[]).
cl(mother(mary,jill),[]).
```

Kann man auch automatisch erzeugen.

Automatische Erzeugung spezifischer Klauseln (0.L.16)

Beispiel aus der Computerlinguistik: *Interpretation* von DCGs (statt der üblichen *Compilation*).

Ausschnitt:

Interpretation (statt Compilation) von DCGs

```
parse(NT,P0,P)              :- (NT --> Body),
                               parse(Body,P0,P).

parse((Body1,Body2),P0,P) :- parse(Body1,P0,P1),
                               parse(Body2,P1,P).
parse([],P,P).

parse([Word|Rest],P0,P)    :-  connects(Word,P0,P1),
                                parse(Rest,P1,P).

parse({Goals},P,P)          :-  call(Goals).

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

Der vollständige Code ist ⇒hier

metazirkulärer Interpreter

Wenn man *dieselbe* Form für Objekt- und Meta-Klauseln verwenden will, muss man einen *metazirkulären* Interpreter schreiben; (in einem anständigen Prolog kann man das).

© Michael Hess 2009

```
prove((G,Gs))    :-      prove_one(G), prove(Gs).
prove(G)         :-      prove_one(G).

prove_one(true)  :-      !.
prove_one(G)     :-      clause(G,SGs), %% NEU!
                         prove(SGs).
prove_one(G)     :-      G.   %% Anmerkung!
```

Der vollständige Code ist ⇒hier

Der Meta-Call muss jetzt *nach* dem Aufruf von ''clause'' eingefügt werden, da sonst einfach der System-Interpreter für alles verwendet wird. Siehe unten!

Sich selbst interpretierende Interpreter (0.L.17)

So nur für ''reines Prolog''...

Aber: Dieser meta-zirkuläre Interpreter ist *sehr* primitiv, und er funktioniert nur für ''reines Prolog'' (Ross 1989:249 sqq):

1. jedes Ziel, das durch eine Programmklausel bewiesen werden kann, wird (potentiell) *zweimal* bewiesen
2. Disjunktionen werden nicht erfasst
3. Kontrollstrukturen (Cut) ebenfalls nicht erfasst

...ohne Disjunktionen.

Daher werden wir im folgenden wieder *nicht*-zirkuläre Interpreter verwenden.

6.1.2 Mögliche Kriterien für die Formulierung von Meta-Regeln

Mögliche Kriterien:

1. Bevorzugung jener *Terme* in der RHS einer Regel, welche
   a. am schwierigsten zu beweisen sind (allgemein: in der Ordnung zunehmender Wahrscheinlichkeit einer Lösung)
   b. am wenigsten ungebundene Variablen besitzen (Sonderfall: *keine* ungebundenen Variablen mehr; sinnvoll bei 'Negation')

vielleicht die drei sinnvollsten Kriterien

© Michael Hess 2009

c. zusammen mit anderen Termen gemeinsame Variablen besitzen (d.h. Bilden von Gruppen von Termen), z.B. zwecks ‘‘intelligentem’’ Backtracken (Beispiel unten)

2. Bevorzugung jener *Regeln*,
   a. welche im Laufe des Beweises am häufigsten erfolgreich waren
   b. welche die wenigsten Bedingungen (RHS-Terme) haben; Intuition: dürfte schnell abgearbeitet werden können; Beispiel unten
   c. explizit als besonders relevant markiert sind (bei Expertensystemen: welche wurden vom erfahrensten Experten beigesteuert, oder: welche wurden bei der Anwendung als zielführend erkannt; ...); Grenzfall zu bereichsspezifischem Wissen

etwas weniger offensichtlich

6.1.2.1 Beweisreihenfolge von Teilzielen: Schwierigste Teilziele zuerst beweisen

Beispiel zu Fall 1a: Zu unterscheiden sind (cf. Rowe 1988:311)

1. Prädikate ohne Variablen
2. Prädikate ohne lokale Variablen
3. Prädikate mit lokalen Variablen

Prädikate *ohne* Variablen:

```
q       :-      a, b, c.
```

Wahrscheinlichkeit des Gelingens

Annahme: (absolute) Wahrscheinlichkeit des Gelingens ist

```
a = 0.9
b = 0.5
c = 0.8

q = 0.9*0.5*0.8 = 0.36
```

d.h.: Beweis von `q` misslingt öfter als nicht. Misslingen merkt man mit folgender Wahrscheinlichkeit:

© Michael Hess 2009

```
                FAIL
                 ↓         SUCCESS  FAIL
nach a allein:  0.1            ↓    ↓
nach a und b:   0.45  (näml. 0.9 * 0.5)

total:          0.55
```

Bei anderer Reihenfolge der RHS-Terme:

```
q :- b, c, a.
```

Misslingen merkt man mit folgender Wahrscheinlichkeit

```
                FAIL
                 ↓         SUCCESS    FAIL
nach b allein:  0.5            ↓    ↓
nach b und c:   0.1  (näml. 0.5 * 0.2)

total:          0.6
```

früher Missrfolg ist besser als später

Also findet man den Misserfolg im zweiten Fall im Durchschnitt (etwas) früher.

Problem: Wie ermittelt man die Wahrscheinlichkeit des Gelingens?

Prädikate ohne *lokale* Variablen:

```
q(X)    :-      a(X), b(X), c(X).
```

Die Variable `X` ist nicht lokal, weil sie ‘‘von oben’’ geliefert wird. Dasselbe Prinzip wie oben ist anwendbar, sofern wir nichts Spezifischeres über die Prädikate wissen.

Prädikate *mit* lokalen Variablen:

```
q       :-      a(X), b(X).
```

Zwei Unterschiede zu oben:

1. Allgemeine *Wahrscheinlichkeiten* sind hier wenig aussichtsreich.

© Michael Hess 2009

2. Stattdessen sind die *Kardinalitäten* der Extension der Prädikate entscheidend (man erinnere sich an das Beispiel der Rolls-besitzenden Amerikaner in der Vorlesung ECL2 ⇒hier).

6.1.2.2 Beweisreihenfolge von Teilzielen: Verzögern der Evaluation von Termen mit ungebundenen Variablen

ein wichtiger Spezialfall:

ungebundene Variablen in ''negierten'' Termen

Beispiel zu Fall 1b: die Definition von ''Junggeselle'' ist im folgenden (mit Absicht) ungünstig definiert:

```
bachelor(X)             :-  not(married(X)), male(X).

married(jim).
married(mary).

male(john).
male(jim).

female(mary).
female(jill).
```

''Closed World Assumption''

Wir wissen, gemäss der ''Closed World Assumption'' (CWA), dass es mindestens einen Junggesellen gibt (John).

Die ''Closed World Assumption'' ist die Annahme, dass alles, was wir nicht positiv als wahr kennen, als falsch betrachtet werden kann. Darauf beruht die Prolog-typische ''negation by failure'' (siehe dazu Vorlesung ''Semantikanalyseverfahren'' ⇒ hier.)

Die Frage

```
?- bachelor(john).
```

wird mit 'yes' beantwortet werden, und

© Michael Hess 2009

```
?- bachelor(jim).
```

mit ‘no’. **Aber:**

```
?- bachelor(X).
```

Prolog ist nicht korrekt!

(für ‘‘Gibt es einen Junggesellen?’’) ergibt ebenfalls ‘no’. *Warum*?

Grund: Das Teilziel

```
not(married(X))
```

wird misslingen.

Lösung: Negierte Terme solange nicht auswerten, wie sie ungebundene Variablen enthalten (ggf. reklamieren).

Besonders bemerkenswert: Eine entsprechende Meta-Regel testet nicht den (statischen) Text der Objekt-Regel, sondern den (dynamischen) Bindungszustand ihrer Variablen. Kann von Abarbeitungsstrategien ohne Meta-Regeln *prinzipiell* nicht geleistet werden. Wird später wichtig werden!

6.1.2.3 Beweisreihenfolge von Teilzielen: Abhängigkeitsgesteuertes Backtracken

Chronologisches Backtracking ist (bekanntlich) dumm.

Beispiel zu Fall 1c: Rein chronologisches Backtracking ist recht stupid. Bei einer Anfrage

```
?-     a(X), b(Y), c(X), e(X,Y), d(X), f(Y), g(Y).
```

muss der Interpreter bei einem Fehlschlag von ‘g(Y)’ *fünf* Schritte zurücksetzen. Durch Beachtung der Abhängigkeiten durch Variablen lässt sich eine bessere Reihenfolge erreichen:

```
?-     a(X), c(X), d(X), e(X,Y), b(Y), f(Y), g(Y).
```

Bei einem Fehlschlag von ‘g(Y)’ muss er nur noch *drei* Schritte zurück.

© Michael Hess 2009

dynamische Bevorzugung gewisser Terme

Implementation: Dynamische Bevorzugung gewisser *RHS-Terme* (jener ohne ungebundene Variablen):

```
prove([]).
prove([true])   :-  !.
prove([G|Gs])   :-      prove_one(G), prove(Gs).

prove_one(true) :-  !.

prove_one(G)    :-  clause(G,SGs),
                    rearrange(SGs,Rearranged), % ← |
                    prove(Rearranged).
prove_one(G)    :-  G.
```

Das Prädikat ‘‘rearrange’’ (|) müsste man nun entsprechend definieren (mit var(X) usw.).

In manchen Fällen muss man die Terme nicht umordnen; es reicht, das Backtracken in gewisse Terme hinein zu verhindern.

‘‘Abschotten’’ von Termen gegen das Backtracking

Beispiel aus der Computerlinguistik: Die Frage

17) Welche Ozeane grenzen an ein afrikanisches und an ein asiatisches Land

könnte übersetzt werden als

```
?- ocean(O),
   borders(O,C1), african(C1), country(C1),
   borders(O,C2), asian(C2), country(C2),

   write(O), nl,
   fail.
```

Realistischere Implementation (0.L.18)

Der Zustand nach Finden der ersten Antwort möge sein

© Michael Hess 2009

```
ocean(indian_ocean),
borders(indian_ocean,sudan), african(sudan), country(sudan),
borders(indian_ocean,india), asian(india), country(india),

write(indian_ocean), nl,
fail.
```

worauf das System backtracken wird: Die Bindung ‘‘india’’ der Variablen ‘C2’ wird gelöst, und die folgende Anfrage wird gestartet:

```
ocean(indian_ocean),
borders(indian_ocean,sudan), african(sudan), country(sudan),
borders(indian_ocean,C2), asian(C2), country(C2).
```

die uns *interessierende* Variablenbindung (die für *O*) wird unverändert sein (‘‘indian_ocean’’).

Was wir möchten, ist, dass das System direkt zum Prädikat ‘‘ocean(O)’’ zurückspringt und einen neuen Wert für *diese* Variable findet und dann die folgenden Bedingungen testet. Wie kann man ein solches Verhalten erreichen?

Wir gehen aus von der Beobachtung, dass in dem Moment, da die Variable ‘‘O’’ gebunden ist, die zwei Gruppen von Prädikaten

```
  borders(indian_ocean,C1), african(C1), country(C1),
```

resp.

```
  borders(indian_ocean,C2), asian(C2), country(C2).
```

voneinander *unabhängig* geworden sind: Sie haben mit keinem anderen Term ungebundene Variablen gemeinsam, sie bilden deshalb zwei in sich geschlossene Gruppen.

Deshalb kann man diese zwei Gruppen von Prädikaten als separate Probleme behandeln, die je nur eine Lösung liefern sollten. Sie müssen also irgendwie vor dem Backtracken von aussen geschützt werden. Das wird hier durch ‘{...}’ dargestellt:

© Michael Hess 2009

```
?- ocean(O),
   { borders(O,C1), african(C1), country(C1) },
   { borders(O,C2), asian(C2), country(C2) }.
```

Implementation:

einfachst zu implementieren via *cut*

1. Entweder Erzeugen einer modifizierten Anfrage, plus Definition

   ```
   {T} :-  T, !.
   ```

   (und entsprechender Operator-Deklaration für ‘‘{’’ und ‘‘}’’)

2. oder die modifizierte Abarbeitung der (unveränderten) Anfrage *direkt* durch den Interpreter (durch ‘‘nichtchronologisches’’, ‘‘intelligentes’’ oder ‘‘dependency-based backtracking’’). Zu letzterem: Rowe 1988:319.

Ad 1:

oft vordefiniert als ‘once’

1. Nunmehr wird beim Backtracken, wie gewünscht, direkt zum ‘‘ocean(O)’’ zurückgesprungen, zur einzigen Stelle, wo potentiell neue Resultate generiert werden.
2. In manchen Prologs ist ‘‘{...}’’ als ‘‘once’’ oder ‘‘one’’ vordefiniert.

6.1.2.4 Verwendungsreihenfolge von Regeln: Leichteste Regel zuerst verwenden

Regeln anders als Terme behandeln!

Bei der Reihenfolge, in der *Regeln* verwendet werden, sieht es anders aus, als bei RHS-Termen: Die *leichteste* Regel muss zuerst verwendet werden. Grund: In

```
r       :-      a.
r       :-      b.
r       :-      c.
```

sind die Regeln ja *disjunktiv* abzuarbeiten. Wenn man eine erfolgreich verwendet hat, ist man fertig. Welche Regel ist die leichteste?

Etwas künstliches **Beispiel:** Bevorzugung von Regeln mit kurzen Klausel-Rümpfen
Idee klarerweise: die sind am schnellsten durch; stimmt eben nur nicht immer

© Michael Hess 2009

```
prove([]).
prove([true])    :-  !.
prove([G|Gs])    :-  prove_one(G), prove(Gs).

prove_one(true) :-  !.

prove_one(G)     :-  bagof((G,List),
                            SGs^(clause(G,SGs),
                                 conjtolist(SGs,List)),
                            Clauses),
                     rearrange(Clauses,Sorted), % 1
                     member((G,Cl),Sorted),     % 2
                     prove(Cl).
prove_one(G)     :-  G.
```

Das komplette Beispiel (0.L.19)

Das Beispiel ist natürlich ziemlich stupid (die längere Regel wäre wohl effizienter, da sie einen Unifikationschritt vermeidet). Abgesehen davon aber: Test t2 zeigt (beim Tracen), dass nunmehr Regeln mit kürzerem Rumpf bevorzugt abgearbeitet werden: 1 ordnet sie um, und 2 nimmt sie in der neuen Reihenfolge von der Liste.

Andere mögliche Meta-Regeln (0.L.20)

Abschliessend: Meta-Interpretation erlaubt

1. Gewinn an Flexibilität
2. Gewinn an Übersichtlichkeit
3. Verlust an Effizienz
4. besonders einfache Implementation in logikbasierten Sprachen

© Michael Hess 2009

6.2 Mehr Effizienz: Speicherung von Zwischenresultaten

Beispiel: Über der Grammatik

```
vp              -->     v_ditr, np, pp(to).
vp              -->     v_ditr, np, np.
```

müsste im Satz

Sinnlose Doppelarbeit

18) Alfred gave *every student who passed the test* a book

die Nominalphrase ‘‘every student who passed the test’’ *zwei* Mal analysiert werden. Das ist sinnlose Arbeit.

‘‘Ausfaktorieren’’ ist eine Idee, aber...

Idee: Gemeinsames Präfix der Regelrümpfe ausfaktorieren.

Mögliche Lösung: Grammatik umformulieren

```
vp              -->     v_ditr, np, ditr_compl.

ditr_compl       -->     pp(to).
ditr_compl       -->     np.
```

... Tabular Parsing ist besser.

Sehr unschön. `ditr_compl` (oder was auch immer) ist keine linguistisch motivierte Konstituente. **Besser** ist ‘‘tabular parsing’’ (zu allem Folgenden: Pereira 1987:185):

‘‘Well-Formed Substring Table’’ (WFST)

- Einmal erreichte Zwischenresultate (Lemmata) werden in einer speziellen Datenstruktur, einer *‘‘Well-Formed Substring Table’’* (WFST), eingetragen, und
- nie mehr neu errechnet, sondern nachgeschaut;
- heisst auch ‘‘Memoisieren’’.

‘‘Memoisieren’’

ganz schlechte Implementation

Simplistische Lösung 1:

© Michael Hess 2009

```
np(np(Det,Adjp,N),S0,S) :-  det(Det,S0,S1),
                            adjp(Adjp,S1,S2),
                            n(N,S2,S),
                            asserta(np(np(Det,Adjp,N),S0,S)).
np(np(pn(Pn),S0,S)      :-  pn(Pn),
                            asserta(np(pn(Pn),S0,S)).
<etc.>
```

resp. die entsprechende compilierte Lösung ist kontraproduktiv: macht alles *ineffizienter*. (und zwar, wenn man von irgendwoher backtrackt).

auf andere Art schlechte Implementation

Simplistische Lösung 2:

```
np(np(Det,Adjp,N),S0,S) :-  det(Det,S0,S1),
                            adjp(Adjp,S1,S2),
                            n(N,S2,S),
                            asserta(np(np(Det,Adjp,N),S0,S) :- !).
np(np(pn(Pn),S0,S)      :-  pn(Pn),
                            asserta(np(pn(Pn),S0,S) :- !).
<etc.>
```

Der Cut blockiert alle anderen Analysen! (wenn z.B. eine Eigennamen-Np erforderlich ist, findet man die nicht mehr).

Daher: Man kann Lemmata erst brauchen, wenn *alle* Analysen (einer gegebenen Phrase) errechnet worden sind.

gute Implementation

Die richtige Lösung

```
:- op(1200,xfx,--->).

parse(NT,S0,S)              :-  nonterminal(NT),
                                find_phrase(NT,S0,S).
parse((Body1,Body2),S0,S)   :-  parse(Body1,S0,S1),
                                parse(Body2,S1,S).
parse([],P,P).

parse([Word|Rest],S0,S)     :-  connects(Word,S0,S1),
                                parse(Rest,S1,S).
parse({Goals},S,S)          :-  call(Goals).

find_phrase(NT,S0,S)        :-  complete(NT,S0), !, % ← !!
```

© Michael Hess 2009

```
                                   known_phrase(NT,S0,S).
find_phrase(NT,S0,S)          :-   (NT ---> Body),
                                   parse(Body,S0,S1),
                                   assert(known_phrase(NT,S0,S1)),
                                   S1 = S.
find_phrase(NT,S0,S)          :-   assert(complete(NT,S0)),
                                   fail.
nonterminal(LHS)              :-   not not(LHS ---> RHS).

connects(Word,[Word|Rest],Rest).

test(Goal)                    :-   parse(s,Goal,[]).
```

Das gesamte Programm ist ⇒hier.

mit Einschränkungen

Aber: Funktioniert so nur für *atomare* Nichtterminale. Cf. Pereira 1987:188 ff. Sinnvoll z.B. in der Morphologieanalyse (siehe unten).

Weiterentwicklung zum Chart-Parser

Weiterentwicklung dieses Verfahrens: *Chart-Parser:*

1. *alle* Analysen *aller* Teilstücke werden (quasi-)simultan errechnet und
2. in einer gemeinsamen *Datenstruktur*, dem Chart, eingetragen (d.h. *alle* Analysen eines Teilstücks sind dort simultan repräsentiert)
3. das Verfahren kann auf- oder absteigend eingesetzt werden (aber aufsteigend ist üblicher)
4. weiterer Vorteil: man erhält *alle* Analysen eines syntaktisch mehrdeutigen Satzes schon im ersten Anlauf

Zum Chart-Parser siehe Genaueres ⇒hier.

6.3 Mehr Effizienz: Techniken der Compilation in logik-basierten Systemen

Der Begriff ‘‘Compilation’’ wird in der Informatik meist verstanden als ‘‘Übersetzung in Maschinensprache’’. Hier allgemeiner verwendet: ‘‘Übersetzung in eine

© Michael Hess 2009

verschiedene Arten von ‘‘Compilation’’

weniger allgemeine Repräsentation’’ (d.h. in eine maschinen*nähere* Repräsentation).

Vier Fälle:

‘‘Entfalten’’ ‘‘Regelexpansion’

‘‘partielle Evaluation’’

‘‘sekundäre Compilation’’

‘‘source to source program transformation’’

‘‘normale’’ Compilation

1. Übersetzung von Metainterpreter relativ zu einem gegebenen *Objektprogramm* in ein direkt ausgeführtes Programm (‘‘Entfalten’’ [‘‘unfolding’’], ‘‘Regelexpansion’’, ‘‘rule collapsing’’)
2. Vorberechnen eines Programms für bestimmte *Input-Werte* (‘‘partielle Evaluation’’)
3. Übersetzung von nicht-deterministischen regelbasierten Systemen in deterministische Systeme (ein Fall von ‘‘sekundärer Compilation’’ oder ‘‘source to source program transformation’’; wie beim Beispiel ‘‘v_ditr’’ oben S. 98). Sekundäre Compilation ist ein Typ von *Optimierung*.
4. klassische Compilation im Sinne der Informatik

Terminologie nicht gefestigt

Aber:

- Unterscheidung zwischen 1 und 2 (und auch 3) nicht sehr hart (auch terminologisch nicht)
- In 1 bis 3 bleibt man in der gleichen Programmiersprache (z.B. Prolog), aber kommt der Maschine dennoch näher
- Die vier Typen von Compilation kann man eine nach der andern anwenden
- 4 werden wir nicht betrachten (gehört in die Kern-Informatik)

Fälle 1 bis 3: Man versucht Programme so zu transformieren, dass sie *nicht* mehr erfordern:

1. Simulation einer anderen Abarbeitungsstrategie als der vom Systeminterpreter direkt verwendeten
2. on-line Verzögern von Variablenbindungen
3. Mehrfachverwenden von Variablen (als Input- und Output-Kanal)
4. Backtracking

Ziele

Statt dessen: Programm so transformieren, dass

1. das Programm direkt ausgeführt werden kann
2. Terme in Klauseln so angeordnet sind, dass sie ‘‘in situ’’ bewiesen werden können

© Michael Hess 2009

3. pro Verwendung einer Variablenbelegung ein eigenes Prädikat existiert
4. das Programm deterministisch ist

6.3.1 Entfalten von Programmen

Jene Meta-Regeln, welche auf die *statische* Struktur der Objekt-Regeln Bezug nahmen, können wegcompiliert werden, *wenn die Menge der Objekt-Regeln unveränderlich ist* (allgemeiner: wenn das *Objekt-Programm* unveränderlich ist).
Das Prinzip: Aus den Klauselm

```
H        :-  A, B, C.
U        :-  V, W, X.
```

wo B und U unifizierbar sind mit **mgu** θ, ergibt Entfaltung

```
(H       :-  A, V, W, X, C) θ
```

‘‘Hilfsprädikate’’

relativ zum Prädikat ‘B’, einem sog. ‘‘Hilfsprädikat’’ (A,B,C,...,X sind alles Meta-Variablen).

Beachte:

- man entfernt einige jener Prädikate, welche man vorher evtl. aus Gründen der Übersichtlichkeit als sog. ‘‘Zwischenprädikate’’ eingeführt hatte.
- letzlich führt man bloss einige Schritte des Beweises früher, d.h. *off-line* aus (und ein für alle Mal)
- das resultierende Programm ist daher zwar weniger übersichtlich, aber *gleich allgemein* wie das urspüngliche Programm
- das resultierende Programm *ersetzt* daher das ursprüngliche Programm

kein Verlust an Allgemeinheit

```
relevant(T,D,X)      :-  tf_idf(T,D,F), threshold(X,F).
threshold(X,F)       :-  F > X.
tf_idf(T,D,F)        :-  term_frequency(T,D,TF),
                         inverse_document_frequency(D,T,DF),
                         F is DF / TF.
```

entfaltet (relativ zu den Hilfsprädikaten ‘tf_idf/3’ und ‘threshold/2’):

© Michael Hess 2009

```
relevant(T,D,X)      :-  term_frequency(T,D,TF),
                         inverse_document_frequency(D,T,DF),
                         F is DF / TF, F > X.
```

Besonders wichtig: Fälle, wo man bis auf die Fakten hinunter expandieren kann:

```
relevant(T,D,X)      :-  tf_idf(T,D,F), threshold(X,F).
threshold(X,F)       :-  F > X.
tf_idf('file system',d2234,0.0122).  %  bekannter Wert für Term 'file system'
                                        in Dokument Nr. D2234
```

ergibt (relativ zu den selben Hilfsprädikaten) in einem ersten Schritt:

```
relevant(T,D,X)      :-  tf_idf(T,D,F), F > X.
tf_idf('file system',d2234,0.0122).
```

und in einem zweiten Schritt:

```
relevant(T,d2234,X) :-  true, 0.0122 > X.
```

resp. einfach

```
relevant('file system',d2234,X)        :-  0.0122 > X.
```

Wichtig ist der Fall, wo ein Ziel *mehrere* Definitionen hat, d.h. wenn mehrere Klausel-Rümpfe ‘‘einzukoppeln’’ sind

```
relevant(T,D,X)      :-  tf_idf(T,D,F), threshold(X,F).
threshold(X,F)       :-  F > X.

tf_idf('file system',d2234,0.0122).
tf_idf('operating system',d2234,0.00142).
tf_idf(file,d2234,0.153).
```

ergibt sich

```
relevant('file system',d2234,X)          :-  0.0122 > X.
relevant('operating system',d2234,X)     :-  0.00142 > X.
relevant(file,d2234,X)                   :-  0.153 > X.
```

Wie macht man das automatisch?

© Michael Hess 2009

Der vollständige Code ist ⇒hier ; (Siehe auch Pereira 1987, Kraan 1989).

```
%%%%%%%%%%%%%%% Meta-Compiler %%%%%%%%%%%%%%%%%%%%%%%%%%%


compile :-
                program_clause(Clause),
                unfold(Clause, CompiledClause),
                assert(CompiledClause),
                write(CompiledClause), nl,
                fail.
compile.


unfold((Head :- Body), (Head :- ExpandedBody))
            :-  !,
                unfold(Body, ExpandedBody).
unfold((Literal,Rest),Expansion)
            :-  !,
                unfold(Literal,ExpandedLiteral),
                unfold(Rest,ExpandedRest),
                conjoin(ExpandedLiteral,ExpandedRest,Expansion).
unfold(Literal,Expansion)
            :-  aux_literal(Literal),
                match_exists(Literal),
                !, clause(Literal,Body),
                unfold(Body,Expansion).
unfold(Literal,Literal).

match_exists(Literal)
            :-  verify(clause(Literal,Body)).


verify(T)   :-  not(not(T)).

conjoin((A,B),C,ABC)    % just like 'append'
            :-  !,
                conjoin(B,C,BC),
                conjoin(A,BC,ABC).
conjoin(true,A,A)   :-  !.
conjoin(A,true,A)   :-  !.
conjoin(A,B,(A,B)).


%%%% Objekt-Programm mit Deklaration der Programm-Klausel %%%%%

program_clause(( relevant(T,D,X) :- tf_idf(T,D,F), threshold(X,F) )).

threshold(X,F)  :-  X > F.

relevant(T,D,X) :-  term_frequency(T,D,TF),
                    inverse_document_frequency(D,T,DF),
                    F is DF / TF, F > X.

%%%%%%%%%%%%%%% Deklaration Hilfsprädikate %%%%%%%%%%%%%
```

© Michael Hess 2009

```
aux_literal(tf_idf(_,_,_)).
aux_literal(threshold(_,_)).
```

Nunmehr sind *drei* Sorten von Klauseln zu unterscheiden: (da ein komplettes Programm verarbeitet wird)

''Programmklauseln''

''Hilfsklauseln''

alle übrigen Klauseln

1. ''*Programmklauseln*'', die entfaltet werden, jedoch selbst nicht gegen ein Literal resolviert werden dürfen; d.h. sie *bleiben* als (modifizierte) Klauseln *bestehen*
2. ''*Hilfsklauseln*'', die entfaltet werden, und die auch selbst gegen ein Literal resolviert werden dürfen; d.h. sie *werden* komplett *wegcompiliert*
3. alle übrigen Klauseln, welche weder entfaltet werden noch selbst gegen ein Literal resolviert werden dürfen ; d.h. sie bleiben *unverändert*

Entscheidung, welche Klauseln welche Rolle spielen sollen, ist *nicht* trivial! Offenbar ist es noch nicht bekannt, wie man eine vernünftige Aufteilung *automatisch* erreichen kann.

Testlauf von `unfold.pl:`

```
| ?- compile.
relevant(_551,_552,_553) :- term_frequency(_551,_552,_725),
inverse_document_frequency(_552,_551,_718),
_546 is _718/_725, _546>_553
yes
```

Bemerkung: Etwas unschön ist, dass `unfold.pl` das Objekt-Programm nicht aus einem separaten File einliest. Dann müsste man nämlich nicht im Programm selbst die Programmklauseln entsprechend modifizieren, sondern könnte sie separat notieren und das Objekt-Programm unverändert lassen.

nur in reinem Prolog!

Beachte: Da beim Entfalten ein *Teil* der Resolutionen schon zur Compilations-Zeit vorgenommen wird, ändert sich u.U. die Reihenfolge der Resolutionsschritte. Deshalb funktioniert es nur, wenn diese Reihenfolge für die Korrektheit des Programms keine Rolle spielt, also im Bereich des reinen Prolog.

Entfalten metazirkulärer Interpreter

Da metazirkuläre Interpreter definitionsgemäss die eigene Sprache interpretieren, kann man die Methode des Entfaltens auch auf sie anwenden.

Einfachstes der Beispiele zur Meta-Interpretation: Beeinflussung der Reihenfolge der Regeln (z.B.: innerhalb der selben ''Prozedur'' jene mit den wenigsten RHS-Termen zuerst).

© Michael Hess 2009

Anstatt im Objekt-Programm die Klauseln

```
greatgrandfather(W,Z)   :-      father(W,X),
                                parent(X,Y),
                                parent(Y,Z).
greatgrandfather(X,Z)   :-      grandfather(X,Y),
                                parent(Y,Z).
```

via Meta-Interpreter ‘‘fliegend’’ umordnen zu lassen, transformiert man das Objekt-Programm *selbst* permanent.

Etwas schwieriger als oben (nicht nur isolierte Klauseln des Objektprogramms werden verarbeitet). Deshalb hier nur skizziert.

Der zentrale Teil des Meta-Interpreters war:

```
prove_one(G)    :-  bagof((G,List),
                           G^SGs^(clause(G,SGs),
                                conjtolist(SGs,List)),
                           Clauses),
                        rearrange(Clauses,Sorted),
                        member((X,Cl),Sorted),
                        prove(Cl), X=G.
```

Ausschnitt aus dem Trace des Meta-Interpreters:

```
prove_one(greatgrandfather(_142,_143))
    :-  prove([grandfather(_547,_553),parent(_553,_548)]).

<backtracking>

prove_one(greatgrandfather(_142,_143))
    :-  prove([father(_569,_575),parent(_575,_580),
                parent(_580,_570)]).
```

Daraus ist abzuleiten als Zielprogramm des Entfaltens:

© Michael Hess 2009

```
greatgrandfather(_142,_143)
    :-  grandfather(_547,_553),parent(_553,_548).
greatgrandfather(_142,_143)
    :-  father(_569,_575),parent(_575,_580),
                parent(_580,_570).
```

d.h. dasselbe wie oben im Objekt-Programm, bloss eben in anderer Reihenfolge.

Beachte:

- Ist rein *statische* Analyse und Transformation des Programms.
- Kann relativ leicht automatisiert werden; aber Achtung: rekursive Regeln kann man nicht ungezielt entfalten

Probleme bei rekursiven Regeln

- Das Gegenteil (das ‘‘Falten’’ von Programmen) ist schwieriger zu automatisieren (und schwieriger zu motivieren)

Schlagwortartig:

```
  (allgemeiner) Meta-Interpreter
+ (allgemeines) Objekt-Programm
  _________________________________   Entfaltung

= (allgemeines) Objekt-Programm
```

ein reales Beispiel für Entfaltung

Realistischeres (und bekanntes) Beispiel für Entfaltung: (realistischer, da ganzes Programm)

- Entfaltung eines
    1. (allgemeinen) Top-Down-Parsers, der
    2. eine (spezifische) DCG interpretiert

Spezialiserung eines Top-Down-Parsers

- das Resultat: ein *direkt* lauffähiges Prolog-Programm, d.h. die Spezialiserung des TD-Parsers *relativ zur gegebenen Grammatik.*

© Michael Hess 2009

Die Grammatik (im Prinzip DCG, aber ---> statt -->):

```
sent(sent(Np,Vp))                    --->    np(Np,Number),
                                             vp(Vp,Number).

np(np(Number,Det,Adjp,N),Number)--->    det(Det,Number),
                                             adjp(Adjp),
                                             n(N,Number).

vp(vp(Number,V,Np),Number)           --->    v(V,tr,Number),
                                             np(Np,_).
vp(vp(Number,V),Number)              --->    v(V,intr,Number).

adjp([])                             --->    [].
adjp(adjp(Adj,Adjp))                 --->    adj(Adj),
                                             adjp(Adjp).

det(det(the),_)                      --->    [the].
det(det(a),sing)                     --->    [a].

n(n(dog),sing)                       --->    [dog].

n(n(man),plur)                       --->    [men].

v(v(eat),tr,sing)                    --->    [eats].
v(v(eat),tr,plur)                    --->    [eat].

adj(brown)                           --->    [brown].
```

Der Top-Down-Interpreter für DCGs

© Michael Hess 2009

```
% parse(NT,P0,P)                :-  (NT ---> Body), % s.u.
%                                   parse(Body,P0,P).

parse((Body1,Body2),P0,P)       :-  parse(Body1,P0,P1),
                                    parse(Body2,P1,P).

parse([],P,P).

parse([Word|Rest],P0,P)         :-  connects(Word,P0,P1),
                                    parse(Rest,P1,P).

parse({Goals},P,P)              :-  call(Goals).

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

Definition von Programmklauseln und Hilfsklauseln: wichtige Entscheidung

(Eine mögliche) Definition von Programmklauseln und Hilfsklauseln:

```
program_clause(( parse(NT,P0,P) :-  (NT ---> Body),  % s.o.
                                    parse(Body,P0,P) )).

aux_literal(parse(_,_,_)).
aux_literal((_ ---> _)).
```

erste Entfaltung

Der entfaltete TD-Parser+Grammatik: (nur Variablen etc. z.T. editiert)

© Michael Hess 2009

```
parse(sent(sent(Np,Vp)),P0,P)
            :-  parse(np(Np,N),P0,P1),
                parse(vp(Vp,N),P1,P).

parse(np(np(Num,Det,Adjp,N),Num),P0,P)
            :-  parse(det(Det,Num),P0,P1),
                parse(adjp(Adjp),P1,P2),
                parse(n(N,Num),P2,P).

parse(vp(vp(Det,_243,_244),Det),P0,P)
            :-  parse(v(_243,tr,Det),P0,P1),
                parse(np(_244,_233),P1,P).
parse(vp(vp(_240,_237),_240),P0,P)
            :-  parse(v(_237,intr,_240),P0,P).

parse(adjp([]),P0,P0):-true.
parse(adjp(adjp(_239,_240)),P0,P)
            :-  parse(adj(_239),P0,P1),
                parse(adjp(_240),P1,P).

parse(det(det(the),_237),P0,P):-connects(the,P0,P).
parse(det(det(a),sing),P0,P):-connects(a,P0,P).
parse(n(n(dog),sing),P0,P):-connects(dog,P0,P).
parse(n(n(man),plur),P0,P):-connects(men,P0,P).
parse(v(v(eat),tr,sing),P0,P):-connects(eats,P0,P).
parse(v(v(eat),tr,plur),P0,P):-connects(eat,P0,P).
parse(adj(brown),P0,P):-connects(brown,P0,P).
```

So ist's nur ± die normale Expansion einer DCG.

Aber mit einer *weiteren* Hilfsklausel

```
aux_literal(connects(_,_,_)).
```

zweite Entfaltung

erhält man immerhin

© Michael Hess 2009

```
parse(sent(sent(Np,Vp)),P0,P)

<... wie oben ...>

            :-  parse(v(_237,intr,_240),P0,P).

parse(det(det(the),_237),[the|P],P):-true.
parse(det(det(a),sing),[a|P],P):-true.
parse(n(n(dog),sing),[dog|P],P):-true.
parse(n(n(man),plur),[men|P],P):-true.
parse(v(v(eat),tr,sing),[eats|P],P):-true.
parse(v(v(eat),tr,plur),[eat|P],P):-true.
parse(adj(brown),[brown|P],P):-true.
```

Es ist (etwas) einfacher geworden.

Beachte:

- `connects` ist wegcompiliert worden.

Ein umfangreicheres Beispiel (0.L.21)

Bemerkung: Der Effiziengewinn durch Entfaltung ist schwer vorauszusehen. Cf. Kraan 1989: Nur ein Left-Corner Parser wurde (15%) schneller, ein TD-Parser wurde *langsamer*.

Entfaltung ist nicht immer gut!

Mögliche Gründe:

1. Grössere Anzahl von Klauseln
2. andere Reihenfolge von Klauseln

© Michael Hess 2009

6.3.2 Partielle Evaluation

Vorausbeweisen mit *bestimmten* Input-Werten:

```
  (allgemeines) Objekt-Programm
+ (spezielle) Input-Werte
________________________________ PARTIELLE EVALUATION
= (spezielles) Objekt-Programm
```

"permanentes Memoisieren"

Beispiel: Lemmatisierung

Letzlich: *Speichern von Zwischenresultaten*, wie beim schon erwähnten "Memoisieren", nur dass man sie dort (durch `assert`) temporär in den Speicher schreibt, während man sie hier *permanent* ins Programm übernimmt. Beispiel 1: Lemmatisierung in der Morphologieanalyse (cf. Gal 1991:91)

```
% specific rules for verb present singulars

% y -> ies except after a vowel
verb_singular(pres,Sing,Plur)
        :- suffix(Sing,[C,y],not(vowel(C)),[C,i,e,s],Plur).

% s,x,z -> es
verb_singular(pres,Sing,Plur)
        :- suffix(Sing,[C],member(C,[o,s,x,z]),[C,e,s],Plur).

% ch,sh -> es
verb_singular(pres,Sing,Plur)
        :- suffix(Sing,[C,h],member(C,[c,s]),[C,h,e,s],Plur).

% default case
verb_singular(pres,Form,Root)
        :- suffix(Form,[],true,[s],Root).
```

mit der folgenden Definition von `suffix`

© Michael Hess 2009

```
suffix(Sing,Astring,Condition,Bstring,Plur) :-  % version for analysis
    not(var(Plur)),
    convert(Plur,Plurlist),
    append(Base,Bstring,Plurlist),
    call(Condition),
    append(Base,Astring,Singlist),
    convert(Sing,Singlist),
    asserta((suffix(Sing,Astring,Condition,Bstring,Plur):- !)), !.
```

Grösserer Ausschnitt des Programms (0.L.22)

memoisierte Werte

Führt zu Ergänzungen des *allgemeinen* Programms durch *spezielle* Werte:

```
suffix(tax, [x], member(x,[o,s,x,z]), [x,e,s], taxes) :- !.
suffix(publish, [s,h], member(s,[c,s]), [s,h,e,s], publishes) :- !.
suffix(try, [r,y], not vowel(r), [r,i,e,s], tries) :- !.
suffix(teach, [c,h], member(c,[c,s]), [c,h,e,s], teaches) :- !.

suffix(taxe, [], true, [s], taxes) :- !.
suffix(publishe, [], true, [s], publishes) :- !.
```

Bemerkungen:

1. Die Cuts sind hier OK: Es sind keine weiteren Analysen möglich.
2. Auch Analysen, die nicht zum Erfolg führen, werden lemmatisiert (z.B. ‘‘publishe’’ und ‘‘taxe’’ sind keine Infinitivformen von Verben) - aber die ‘‘Memoisierung’’ spart dennoch Zeit.

Und: Keine Probleme mit Subsumption, da Identität geprüft wird.

Beachte: Falsch wäre es, Lemmas z.B. der folgenden Art zu assertieren:

```
word(verb(try,1-tr,3+sing,pres),tries) :-  !.
```

Hier wären weitere Analysen möglich (zumindest: Substantiv-Plural); der ‘‘complete’’-Test wäre hier erforderlich.

Beispiel: ‘‘membership’’

Beispiel 2: Listenoperationen (aus der Quintus Prolog Library):

© Michael Hess 2009

```
member(X, [X|_]    ).  %% keine Cuts nötig!
member(X, [_,X|_]  ).
member(X, [_,_,X|_]).
member(X, [_,_,_|L]) :-  %% (fast) allgemeine Version
        member(X, L).

/*  The original definition was
        member(X, [X|_]).
        member(X, [_|L]) :- member(X, L).
    We have unrolled that, to save 10-20%
    of the time.  %% der Grund
*/
```

Die ‘‘speziellen Input-Werte’’ sind hier die *Anzahl* von Elementen vor dem getesteten Element. Die anonymen Variablen kann man sich vorstellen als Verallgemeinerung konkreter Werte.

Hier sind keine Cuts erforderlich, weil die vierte Klausel nur *fast* allgemein ist: Sie schliesst die speziellen Fälle aus. Elegant, aber nicht immer zu machen.

Hintergrundinformation (0.L.23)

Zusammenfassend:

- ein partiell evaluiertes Programm ist ein *spezielles* Programm für bestimmte Input-Werte
- zwei mögliche Verwendungen: man weiss, dass
  1. nur bestimmte Input-Werte vorkommen werden: das resultierende Programm *ersetzt* das ursprüngliche Programm
  2. bestimmte Input-Werte besonders häufig vorkommen werden: das resultierende Programm *ergänzt* das ursprüngliche Programm
- d.h.: partielle Evaluation ist Verallgemeinerung von ‘‘Memoization’’

© Michael Hess 2009

6.3.3 Sekundäre Compilation

''Entfaltete'' und partiell evaluierte Programme sind z.T. immer noch nicht-deterministisch. Beispiel oben: Wenn die erste greatgrandfather-Klausel misslingt, muss zurückgesetzt werden, und die zweite Klausel muss probiert werden. Hierzu: Rowe 1988:110, 151.

Ziel: deterministische Prozedur

''directed acyclic graph'' (''dag'')

''Sekundäre Compilation'' hat zum Ziel eine rein deterministische Entscheidungsprozedur:

- in graphentheoretischer Darstellung: einen ''directed acyclic graph'' (''dag''; in einfachen Fällen: Entscheidungs*baum*),den man mit *Garantie* auf Erfolg *in einem Zug* abschreiten kann
- in automatentheoretischer Interpretation: einen deterministischen endlichen Automaten ohne Schleifen

Beurteilung:

- **Vorteil:** Schnell! ''No matching, no binding, no backtracking''
- **Nachteile:**
    1. schwer zu entwerfen: jede Frage muss die Antworten auf frühere Fragen implizit berücksichtigen
    2. Aus dem selben Grund: Modifikation und Fehlersuche schwierig
    3. Fragen werden u.U. mehrfach gestellt, da kein echter Speicher (zum Cachen von Antworten) zur Verfügung steht

Aber: Nachteile nur relevant, wenn man ''von Hand'' arbeitet. *Nicht* relevant, wenn man Entscheidungsstrukturen automatisch errechnen lässt. Wie?

Zwei Stufen:

1. Regelexpansion (wie gehabt)
2. Partitionierung der Regelmenge

Beispiel: (Rowe 1988:111)

```
r   :-  a, d, not(e).
s   :-  not(a), not(c), q.
t   :-  not(a), p.
u   :-  a, d, e.
u   :-  a, q.
v   :-  not(a), not(b), c.
```

© Michael Hess 2009

```
p   :-  b, c.
p   :-  not(c), d.
q   :-  not(d).
```

Erster Schritt: Expansion ergibt

```
r   :-  a, d, not(e).
s   :-  not(a), not(c), not(d).
t   :-  not(a), b, c.
t   :-  not(a), not(c), d.
u   :-  a, d, e.
u   :-  a, not(d).
v   :-  not(a), not(b), c.
```

Zwischenprädikate `p` und `q` sind verschwunden.

Zweiter Schritt: Datenbank d.h.: Programm partitionieren.

1. *Frage:* Welches Prädikat kommt in möglichst vielen RHS vor?

   Antwort: `a`.

2. zwei Sub-Datenbanken bilden, alle Vorkommen von ‘a’ und ‘not(a)’ entfernen

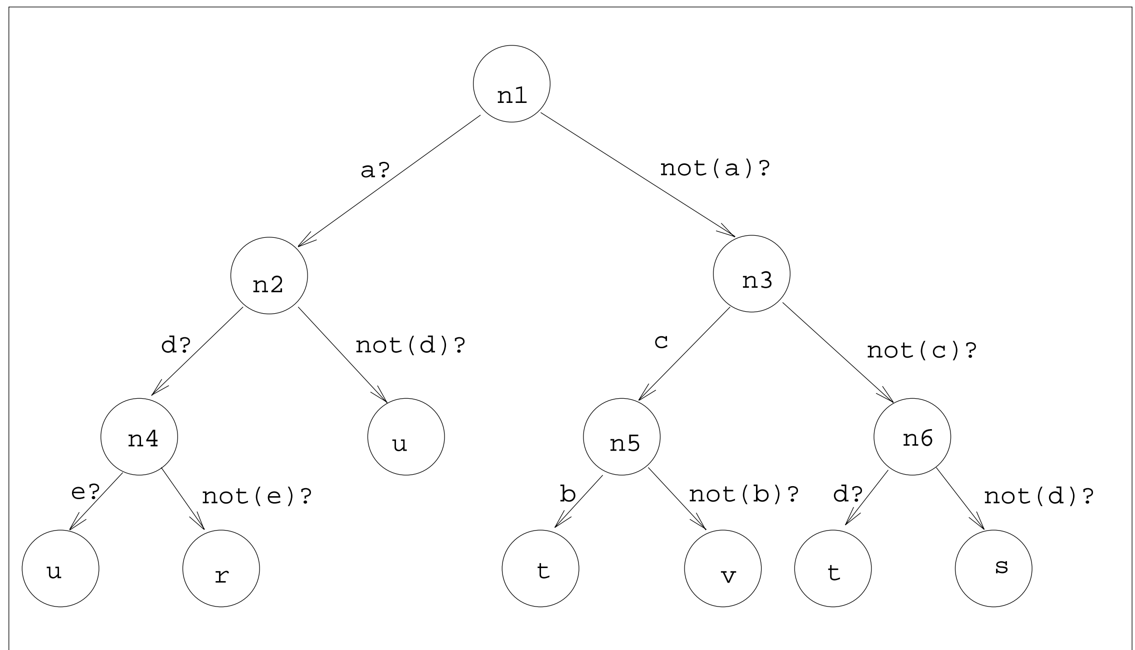
```
r   :-  d, not(e).      %% Sub-Datenbank für 'a'
u   :-  d, e.
u   :-  not(d).

s   :-  not(c), not(d). %% Sub-Datenbank für 'not(a)'
t   :-  b, c.
t   :-  not(c), d.
v   :-  not(b), c.
```

Prozedur rekursiv wiederholen:

© Michael Hess 2009

```
r    :-  not(e).    % Sub-Datenbank für 'a, d'
u    :-  e.

u.                  % Sub-Datenbank für 'a, not(d)'

t    :-  b.         % Sub-Datenbank für 'not(a), c'
v    :-  not(b).

s    :-  not(d).    % Sub-Datenbank für 'not(a), not(c)'
t    :-  d.
```

Ergibt Baum (Illustration: Rowe 1988:154)



In Prolog :

```
successor(n1,n2)        :-  askif(y,a).
successor(n1,n3)        :-  askif(n,a).
successor(n2,n4)        :-  askif(y,d).
<etc.>
```

plus einige Interface-Prädikate.

© Michael Hess 2009

Umfangreicherer Programmausschnitt (0.L.24)

Testlauf:

```
| ?- go.
Is this correct: a
|: n.
Is this correct: c
|: y.
Is this correct: b
|: y.

The answer is: t
```

erneut: nur für reines Prolog

Beachte: Da die Reihenfolge in der RHS-Termen nicht beachtet wurde, geht das im Prinzip nur für *reines* Prolog.

6.4 Mehr Effizienz: Parallele Abarbeitung in logik-basierten Systemen

Parallele Abarbeitung von Programmen ist

Aufspalten des Kontrollflusses

- ein grundsätzlich anderer Ansatz zur Effizienzerhöhung: *Aufspalten* des Kontrollflusses (statt Umlenken)
- letztlich nur bei entsprechender *Hardware* sinnvoll (aber sequentiell simulierbar)
- im Kontext der Logik-Programmierung besonders *attraktiv*

A propos spezielle Hardware: Es müssen nicht unbedingt Multi-Prozessor-Maschinen sein; man kann auch die einzelnen Prozesse (via Netz) auch verschiedene (u.U. ürtlich weit verstreute) Computer verteilen.

drei Sorten von Parallelität

In der Logik-Programmierung gibt es drei Sorten von Parallelität:

1. Oder-Parallelismus
2. Und-Parallelismus

© Michael Hess 2009

3. Unifikations-Parallelismus

Illustriert am folgenden Beispiel:

```
a(X,Y)      :-      b(X), c(X), d(Y,Z), e(Y).
a(X,Y)      :-      e(Y), f(X,Y), not(c(X)).

b(2).
c(2).
d(3,4).
e(5).
f(1,5).
```

Angenommenes Ziel:

```
?-  a(A,B).
```

Antwort:

```
A = 1
B = 5
```

Im einzelnen:

1. Oder-Parallelismus: Die zwei Regeln können gleichzeitig abgearbeitet werden (Bedingung: unifizierbare Regelköpfe)
2. Und-Parallelismus: Die Gruppen ‘b(X), c(X)’ und ‘d(Y,Z), e(Y)’ der ersten Regel können gleichzeitig bewiesen werden.

   Aber: Wenn es gemeinsame Variablen gibt, wie in ‘e(Y), f(X,Y), not(c(X))’, dann entstehen u.U. Probleme (Variablenbindungskollisionen).
3. Unifikations-Parallelismus: beim Unifizieren von ‘d(Y,Z)’ mit ‘d(3,4)’ (etc.) können die Variablen parallel gebunden werden

   Aber: Wenn es gemeinsame Variablen innerhalb eines Terms gibt (z.B. ‘q(X,X)’) dann entstehen u.U. erneut Variablenbindungskollisionen

Beachte:

© Michael Hess 2009

am einfachsten: Oder-Parallelismu

- Einziger wirklich unproblematischer Fall: Oder-Parallelismus. Deshalb auch als einziger relativ oft realisiert.
- Situation ist immer noch besser als in imperativen Sprachen! Dort muss man spezielle Compiler haben, welche die Programme parallelisieren.

Extremfall: Konversion in einen Und-Oder-Nicht-Graph. Jede Regel des Programms funktioniert als (oder *ist*) ein Hardware-Element:

1. Konjunktionen in Regeln werden zu ''und''-Elementen
2. Regeln mit gleichen Köpfen werden zu ''oder''-Elementen
3. Negationen werden zu Invertern

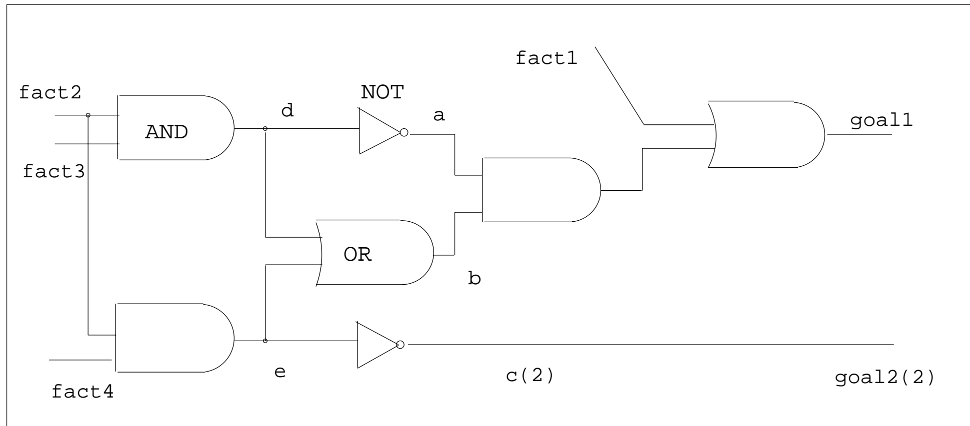
Fakten ('fact1' bis 'fact4') sind ''inputs'', Ziele ('goal1' und 'goal2') sind ''outputs''. Alle Elemente errechnen Ausgangswerte, sobald was reinkommt.

Die Regeln von oben (hier wiederholt)

```
goal1       :-      fact1.              % R1
goal1       :-      a, b.               % R2
goal2(X)    :-      c(X).               % R3
a           :-      not(d).             % R4
b           :-      d.                  % R5
b           :-      e.                  % R6
c(2)        :-      not(e).             % R7
d           :-      fact2, fact3.       % R8
e           :-      fact2, fact4.       % R9
```

werden so zu einem Schaltkreis (Rowe 1988:114):

© Michael Hess 2009



Man beachte: Hier gibt es keine zeitliche Abfolge zwischen dem Errechnen einzelner Werte: völlig parallel.

7. Planung

7.1 Unterschiede zwischen Suche und Planung

Beispiele für Planungsaufgaben:

1. Wie transportiert man bestimmte Güter mit einer gegebenen Anzahl von Fahrzeugen mit verschiedenen Eigenschaften an verschiedene Orte, wobei gewisse Restriktionen der Strecken beachtet werden müssen?
2. Wie produziert man Güter in einer Fabrik, wobei eine Reihe von Fabrikationsschritten bekannt sind, bestimmte Ressourcen in bestimmten Mengen zur Verfügung stehen, bestimmte Zeiteinschränkungen beachtet werden müssen usw.
3. Abfolge von Handlungen eines *Roboters* zur Erreichung vorgegebener Ziele in einer komplexen und sich verändernden Umgebung
4. Abfolge von Handlungen eines *Informationssystems* zur Erreichung vorgegebener Ziele
5. Generierung eines natürlichsprachlichen Texts

© Michael Hess 2009

Derartige Problemstellungen wurden z.T. schon lange vor der ''Erfindung'' der Künstlichen Intelligenz in den Disziplinen des Operation Research und z.T. der sog. Systemforschung theoretisch untersucht. In der Künstlichen Intelligenz werden diese Ansätze in einen weiteren Rahmen integriert und vor allem einer automatischen Lösung zugeführt.

Definition ''Planung'': Problemlösung

1. in Situationen, wo *Handlungen* nicht (oder nur schwer) *rückgängig* zu machen sind
2. wobei die *Welt* sich *verändern* kann
3. in *unvollständig beschriebenen* (oder beschreibbaren) *Situationen*
4. durch die Lösung von *Teilproblemen*
5. unter Berücksichtigung von *Zielinteraktionen* zwischen Teilproblemen
6. unter Berücksichtigung der *Verwendung* von *konkurrierenden Ressourcen*
7. wobei u.U. *mehrere Ziele* gleichzeitig erfüllt werden müssen
8. wobei oft *verschiedene Typen von Zielen* unterschieden werden müssen

Im einzelnen:

1. Die bisherigen Problemlösungssituationen nahmen immer an, man könne alles rückgängig machen
2. Die Welt, in der ein Planer agieren muss, wird sich in der Regel verändern, entweder durch die Handlungen des Agenten selbst, oder unabhängig von ihnen;
3. Die vollständige Beschreibung auch einfacher Situationen der *realen* Welt ist dagegen prohibitiv aufwendig. Schon im (sehr einfachen) Fall eines Roboters in einer Wohnung müssten beschrieben werden
    a. alle ''relevanten'' Objekte im Raum
    b. ihre absoluten Positionen
    c. oder ihre relativen Positionen

4. Dementsprechend möchte man das Gesamtproblem in eine Anzahl von Teilproblemen auflösen können, von denen jedes einen Aspekt der Gesamtsituation behandelt.

© Michael Hess 2009

5. Dabei kann es aber geschehen, dass die Lösung eines nachfolgenden Teilproblems ein schon erreichtes Teilziel wieder rückgängig macht. Wenn man mehrere Ziele gleichzeitig erreichen will, muss man derartige *Zielinteraktionen* berücksichtigen.

   Derartige Zielinteraktionen müssen entweder

   a. von vornherein *verhindert* oder
   b. in ihrem Effekt *korrigiert*

   werden.

6. Beim Lösen von Teilproblemen kann man auch Ressourcen aufbrauchen (Zeit, Energie, Material), welche für die Lösung nachfolgender Teilprobleme fehlen werden.
   a. mehrere Sorten von Ressourcen berücksichtigen
   b. die eine Sorte von Ressourcen gegen die andere abwägen:
      i. Speicherplatz vs. Rechenzeit (in der Informatik)
      ii. Kosten vs. Zeit (bei vielen Konstruktionsvorhaben)
7. Nicht nur (vom System generierte) Teilziele können in Konflikt geraten, sondern auch die vom Benutzer vorgegebenen.
8. Es gibt z.B. *temporäre* und *permanente* Ziele

7.2 Grundlegende Planungsverfahren

7.2.1 Die Klötzchenwelt als Modell

Zur Diskussion dieser Fragen muss man natürlich dennoch eine vereinfachte Welt betrachten. Der typische Ausschnitt der Welt, den man dazu betrachtet, ist eine Spielzeugwelt aus Klötzchen, Pyramiden etc., die ein Roboter mit sehr beschränkten Möglichkeiten (zum Beispiel mit einem einzigen Arm) umschichten soll, um bestimmte Zielkonfigurationen zu erreichen.

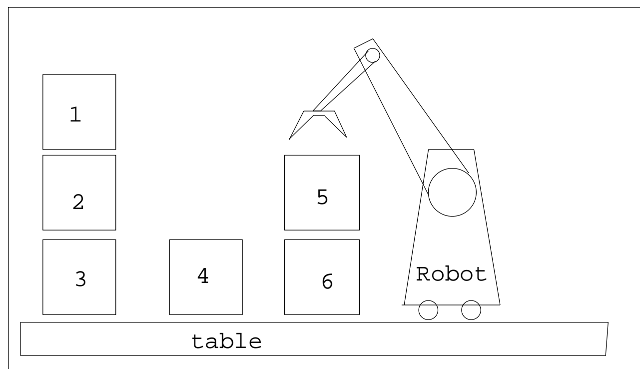
Zuerst muss man den Ausgangszustand (welcher Block wo liegt etc.) sowie die herrschenden physikalischen uws. Gesetze in einer Weise darstellen, welche das Erkennun und Verändern von *Teil*situationen erlaubt. Man kann dazu erneut die Logik als Repräsentationssprache verwenden. Dann wird man zum Beispiel die

© Michael Hess 2009

Ausgangssituation als eine Menge von Partikularaussagen darstellen:

```
block(block1).
block(block2).
block(block3).
block(block4).
block(block5).
block(block6).

robot(robot).

on(block1,block2).
on(block2,block3).
on(block3,table).
on(block4,table).
on(block5,block6).
on(block6,table).

on(robot,table).
arm_empty(robot).

clear(block1).
clear(block4).
clear(block5).
```

also graphisch:



Beachte: was ‘‘table’’ bedeutet, ist nicht gesagt worden, und muss auch nicht gesagt werden.

Die wohl einfachste Fragestellung ist: Wie kann man einen bestimmten Klotz auf

© Michael Hess 2009

einen andern Klotz legen? Wir müssen nun *nicht* die gesamte Zielsituation beschreiben, sondern nur den relevanten Ausschnitt davon, z.B.

```
on(block1,block6).
```

Auch die Operatoren können über Ausschnitten der Gesamtsituation definiert werden. Um das obige Ziel zu erreichen, brauchen wir z.B. die Position anderer Objekte (z.B. von Klotz 4 oben) überhaupt nicht zu beachten. So könnte die einfachste Version eines Operators zur Erreichung des obigen Ziels so aussehen:

```
achieve(on(X,Y), stack(R,X,Y)).
```

Hier nimmt man (in HCL-Art) an, alle Variablen seien universell quantifiziert. Im folgenden werden alle Regeln in HCL ausgedrückt werden, sodass diese Konvention beibehalten werden kann.

Aber das genügt nicht: Um etwas auf etwas anderes stellen zu können, müssen eine Reihe von Bedingungen erfüllt sein:

1. das Zielobjekt muss frei sein
2. der Agent muss das bewegte Objekt halten
3. der Agent muss beim Zielobjekt stehen

Daher: Wenn diese Bedingungen nicht erfüllt sind, muss man sie zuerst wahr machen, und dazu muss man Operatoren einsetzen, zu deren Einsatz ebenfalls Bedingungen erfüllt sein müssen etc.:

```
achieve(on(X,Y),stack(R,X,Y),[holding(R,X),at(R,Y),clear(Y)]).

achieve(holding(R,X),pickup(R,X),[at(R,X),arm_empty(R),clear(X)]).
achieve(at(R,X),walk_to(R,X),[]).
achieve(clear(X),unstack(R,Y,X),[on(Y,X),at(R,Y),arm_empty(R)]).

achieve(arm_empty(R),put_down(R,X),[holding(R,X)]).
```

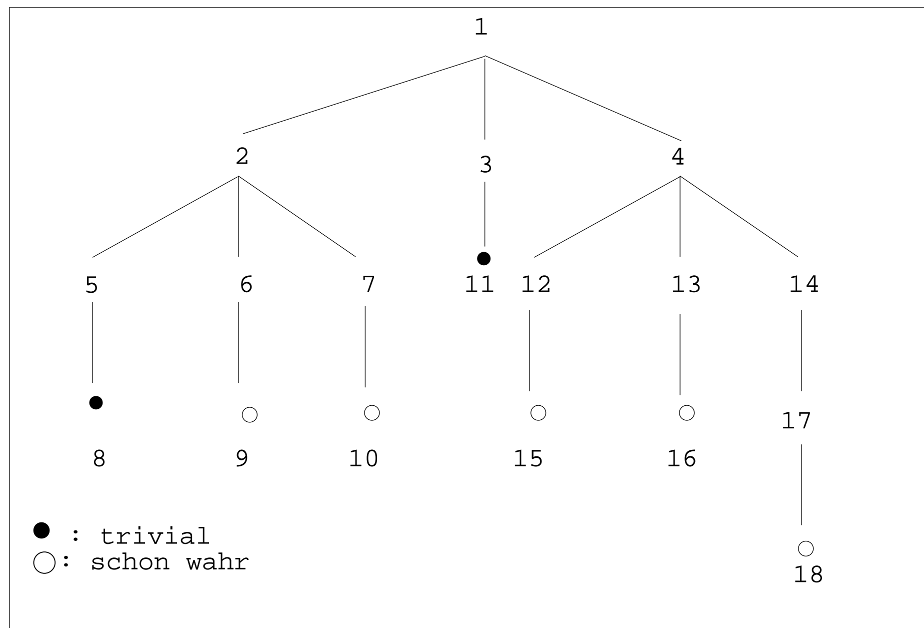
Die Situation legt ein absteigendes Theorembeweisvorgehen nahe:

© Michael Hess 2009

```
ZIEL:    on(block1,block6).

(ANGESTREBTES) VORGEHEN:

achieve(on(block1,block6), stack(robot,block1,block6),                      (1)
                                [holding(robot,block1),                     (2)
                                 at(robot,block6),                          (3)
                                 clear(block6)]).                           (4)

        achieve(holding(robot,block1), pickup(robot,block1),
                                            [at(robot,block1),              (5)
                                             arm_empty(robot),              (6)
                                             clear(block1)]).               (7)

                achieve(at(robot,block1), walk_to(robot,block1),[]). TRIVIAL
                arm_empty(robot).        SCHON WAHR                         (9)
                clear(block1).           SCHON WAHR                         (10)
        achieve(at(robot,block6), walk_to(robot,block6),[]). TRIVIAL        (11)
        achieve(clear(block6), unstack(robot,block5,block6),
                                         [on(block5,block6),                (12)
                                         at(robot,block5),                  (13)
                                         arm_empty(robot)]).                (14)
                on(block5,block6)        SCHON WAHR                         (15)
                at(robot,block5)         SCHON WAHR                         (16)
                achieve(arm_empty(robot),put_down(robot,block1),
                                         [holding(robot,block1)]).          (17)
                        holding(robot,block1)   SCHON WAHR                  (18)
```

Nach dem Ausarbeiten des Planes würde die geeignete Verkettung der mittleren Terme aller Operatoren als Handlungsvorschrift interpretiert:

© Michael Hess 2009



Rekursives Absteigen und Ausführen der Handlungen beim Verlassen eines Knotens ergibt:

```
[walk_to(robot,block1), pickup(robot,block1),
 walk_to(robot,block6), put_down(robot,block1),
 unstack(robot,block5,block6), stack(robot,block1,block6)].
```

Aber da sind einige Dinge noch nicht geregelt:

1. **Hauptproblem:** der Plan ist *falsch*. Die letzte Handlung `stack(robot,block1,block6)` kann nicht ausgeführt werden, da der Roboter den Klotz 1 zu dem Zeitpunkt nicht mehr hält.

   Grund: Als der Roboter Klotz 1 ablegte, um Klotz 5 abzuräumen (14), machte er ein schon erreichtes Zwischenziel, `holding(robot,block1)`, das zum Realisieren von `stack(robot,block1,block6)` erforderlich ist, rückgängig.

   Lösung: z.B. Einschieben von `pickup(robot,block1)` vor dem letzten Schritt.

2. wie beschreibt man den neuen Zustand der Welt nach Anwendung eines Operators?

© Michael Hess 2009

3. was tut man, wenn eine Handlung mehrere Veränderungen der Welt bewirkt (Ziel und Nebeneffekte)?

4. wie unterscheidet man (wahrzumachende) *Voraussetzungen* und (zu testende) *Bedingungen*?

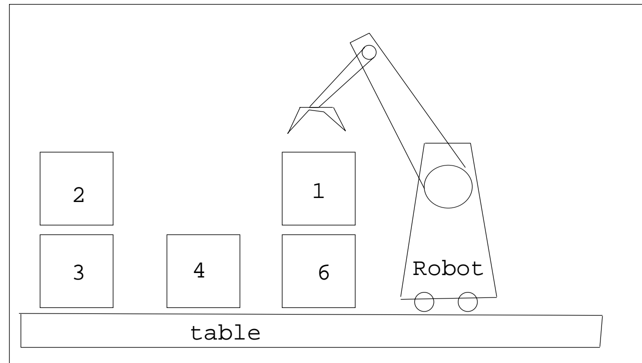
7.2.2 Die Beschreibung von partiellen Situationen und das ‘Frame’-Axiom

Scheinbar einfachste Methode zur Beschreibung der neuentstandenen Situation: Alte Zustandsbeschreibung löschen, neue assertieren.

```
block(block1).
block(block2).
block(block3).
block(block4).
block(block5).
block(block6).

robot(robot).

on(block1,block6).
on(block2,block3).
on(block3,table).
on(block4,table).
on(block6,table).

on(robot,table).
arm_empty(robot).

clear(block1).
clear(block2).
clear(block4).
```

also:

© Michael Hess 2009



Nachteil: unnötig aufwendig.

Grund: Die allermeisten Dinge in der Welt werden durch eine gegebene Aktion *nicht* verändert, sollten daher auch nicht gelöscht und unverändert assertiert werden. Beispiele: alle Aussagen über Sorten (dass ‘‘block1’’ ein Klotz ist etc.; das liesse sich über Partitionierung der Datenbank machen, aber: auch die Position von Klotz 4 bleibt unverändert).

Das sog. *Frame-Problem*. Name: ‘‘Frames’’ sind die Hintergrundsbilder in manuell erstellten Trickfilmen. Problem bekannt seit 1968 - Steinzeit also

Grundidee einer Lösung: Was nicht explizit als verändert bezeichnet worden ist, blieb unverändert. Man beachte: Ist eine Variante der CWA, aber in der speziellen Sprache der Planung (nicht Prolog), muss daher vom Planungsprogramm (dem Interpreter dieser Sprache) auch explizit berücksichtigt werden. Hat die üblichen Nachteile der CWA, v.a. Nicht-Monotonie der Schlussfolgerungen.

Konkrete Implementation des Frame-Axioms: Entweder den Jetzt-Zustand anhand der Operatoren aus der Ausgangssituation errechnen, oder die Ausgangssituation aus dem Jetztzustand errechnen.

Wir wählen dei erste Methode: ein Zustand wird immer dann als vorliegend betrachtet,

1. wenn man im Ausgangszustand ist, und wenn der Zustand explizit als gegeben ausgewiesen ist, oder
2. wenn die letzte ausgeführte Aktion diesen Zustand erzeugt hat, oder
3. wenn er zu einem früheren Zeitpunkt vorlag und nicht durch eine der nachfolgenden Aktionen explizit *zerstört* worden ist.

© Michael Hess 2009

Alle durchgeführten Aktionen werden z.B. in einer Liste eingetragen, und anhand dieser ‘‘Geschichte’’ kann der jeweilige Zustand der Welt, nach Bedarf, hochgerechnet werden. Das heisst, dass man wirklich nur die unumgängliche Arbeit leistet (Implementation unten).

Aus (ergänzt um fehlende Aktion)

```
[walk_to(robot,block1), pickup(robot,block1),
 walk_to(robot,block6), put_down(robot,block1),
 unstack(robot,block5,block6),
 pickup(robot,block1), stack(robot,block1,block6)].
```

und der Ausgangssituation von oben, d.h.

```
<...>
on(block4,table).
<...>
```

kann man erschliessen:

1. Klotz 4 ist auf dem Tisch
2. Klotz 1 ist auf Klotz 6
3. der Roboter steht bei Klotz 6

Weitere Anwendung der CWA: Bedingungen fürs Aufschichten, d.h.

1. Zielklotz frei
2. zu bewegender Klotz ebenfalls frei

bisher im Ausgangszustand explizit aufgelistet:

```
clear(block1).
clear(block4).
clear(block5).
```

Einfacher durch die eine Regel

```
∀ X (block(X) ∧ ¬ (∃ Y: (block(Y)
        ∧ on(Y,X))) → clear(X))
```

‘‘ein jeder Klotz, so dass es keinen Klotz gibt, der auf ihm steht, ist frei’’.

© Michael Hess 2009

Stärker als Frame-Axiom, welches voraussetzte, dass eine Relation zumindest im Ausgangszustand genannt sein musste, um zu bestehen.

Partielle Beschreibungen der Realität sind nun möglich: Genannt wird, was relevant ist, der Rest gilt als nicht vorliegend.

7.2.3 ‘ADD’- und ‘DELETE’-Listen zur Beschreibung des Effekts von Operatoren

Offene Fragen:

1. wo steht, was wodurch rückgängig gemacht wird?
2. **Und:** Obwohl das Ziel des Operators ‘stack’

   ```
   achieve(on(X,Y),stack(R,X,Y),[holding(R,X),at(R,Y),clear(Y)]).
   ```

   darin besteht, Objekt *X* auf Objekt *Y* zu bringen, ist das nicht das einzige, was durch Anwendung des Operators erreicht wird: Nebeneffekte

Daher: sog. STRIPS-Regeln. ‘‘STRIPS’’: Name des Programms, in dem das implementiert wurde [1971]

```
stack(R,X,Y)
ADD:            on(X,Y)
                at(R,Y)

CONDITIONS:     holding(R,X)
                at(R,Y)
                clear(Y)

DEL:            clear(Y)
                holding(R,X)
```

Beachte: Gewisse der Bedingungen erscheinen in der ADD-List, andere in der DEL-List. Frage: denkbar, dass eine Bedingung weder in der ADD- noch in der DEL-Liste auftaucht?

Vorteile:

© Michael Hess 2009

1. man kann Nebeneffekte angeben
2. man kann Nebeneffekte zur Planung ausnützen
3. und natürlich die Zerstörung von Zuständen überhaupt erkennen

Aber:

1. eigentliches Ziel und Nebeneffekt nicht unterschieden
2. Bedingungen und Voraussetzungen immer noch nicht unterschieden:

```
unstack(R,Y,X)
ADD:            clear(X)

CONDITIONS:     on(Y,X)      ← Bedingung
                at(R,Y)      ← Voraussetzung
                arm_empty(R)← Voraussetzung

DEL:            on(Y,X)
```

7.2.4 Zielinteraktionen und ihre Behandlung

Wenn man mehrere Ziele gleichzeitig erreichen will, muss man *Zielinteraktionen* berücksichtigen.

Beispiel oben: zu haltenden Klotz weglegen, um was anderes zu tun. Lösung dort (Klotz weglegen und wieder aufheben) war nicht clever (doppelte Arbeit).

Besser, die Reihenfolge der Einzelziele zu vertauschen, um sie beide zu realisieren: Ansatz der sog. *linearen Planungsprogramme*.

Aber in den meisten Fällen ist das nicht so einfach. Am besten zu sehen anhand von ''Warplan'' (von D.H.D. Warren, 1974 [!]; cf. [/home/ludwig6/hess/prolog/plans/warplan0.pl]

© Michael Hess 2009

```
/* Warplan (Ausschnitte); die originale Version
nach Warren 1974, aber mit moderner Syntax */

plans(C,T,impossible)   :- \+consistent(C,true), !.
plans(C,T,T1)           :- plan(C,true,T,T1).

plan([],P,T,T).
plan([X|C],P,T,T2)      :- solve(X,P,T,P1,T1),
                           plan(C,P1,T1,T2).

solve(X,P,T,P,T)        :- always(X).    %%%%%%% 1
solve(X,P,T,P1,T)       :- holds(X,T),   %%%%%%% 2
                           include(X,P,P1).
solve(X,P,T,[X|P],T1)   :- add(X,U),     %%%%%%% 3
                           achieve(X,U,P,T,T1).


include(X,Y,Y)          :- X==Y.
include(X,Y,[X|Y]).
```

Die Grundstruktur des Programms ist einfach:

1. `plans(Ziele,Start,Result)` testet, ob das Ziel inkonsistent ist, sonst ruft es `plan(Ziele,Protected,Aktionen,NeueAktionen)` auf;

2. ‘‘plan’’ löst *mehrere* Probleme, `[X|C]`, ‘‘solve’’ ein *einziges:* (`X`). Ein Ziel ist erreicht,
   a. wenn es eine ‘‘ewige Wahrheit’’ ist (**1**)
   b. wenn es zum gegenwärtigen Zeitpunkt schon realisiert ist (**2**)
   c. oder wenn alle seine Voraussetzungen realisiert worden sind und wenn anschliessend eine zielführende Aktion durchgeführt worden ist. : **3**

3. `include(State,OldProtected,NewProtected)` fügt den erreichten Zustand zur Liste der geschützten Ziele

© Michael Hess 2009

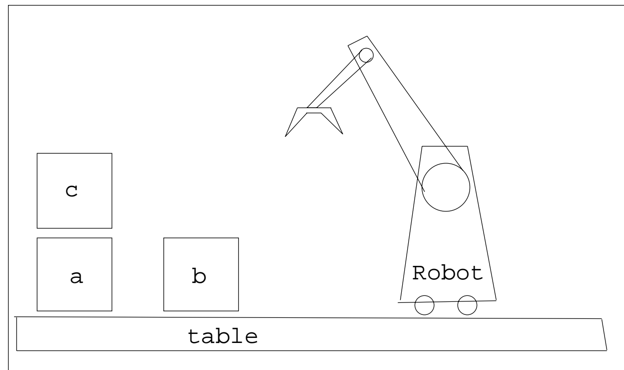
G

```
add(on(U,W), move(U,V,W)).
add(clear(V), move(U,V,W)).

holds(X,T)              :- given(T,X).    %%%%%%% 4
holds(X,(T,V))          :- add(X,V).      %%%%%%% 5
holds(X,(T,V))          :- !, preserved(X,V), %% 6
                           holds(X,T),
                           preserved(X,V).
```

Ein Ziel ist erreicht (‘‘holds’’), wenn

1. es im gegenwärtigen Zustand (T) gegeben ist (**4**)
2. soeben realisiert worden ist (**5**)
3. früher realisiert worden ist und nachher nie zerstört wurde (**6**).

Wie wird nun aber das Problem der Zielinteraktionen gelöst? Wenn die Ausgangssituation so aussieht:



und das Mehrfach-Ziel

```
on(c,b), on(b,a)
```

ist, so wird ein lineares Programm die Ziele zuerst in der angegebenen Reihenfolge zu lösen versuchen. Es wird c auf b legen, aber um nachher b auf a legen zu können, muss dieses erreichte Ziel wieder rückgängig gemacht werden. Um diese unnötige Arbeit einzusparen, wird jedes erreichte Ziel *geschützt* und in einer besonderen Liste mitgeführt. Wenn ein geschütztes Ziel durch eine Aktion verletzt würde, wird die

© Michael Hess 2009

Reihenfolge der Ziele vertauscht.

Wenn aber das Ziel

```
on(a,b), on(b,c)
```

ist, so wird ein Vertauschen der zwei Ziele *nicht* zu einem Erfolg führen. In jedem Fall müsste ein geschütztes Ziel rückgängig gemacht werden. Die Lösung besteht natürlich in der Sequenz

```
move c from a to floor
move b from floor to c
move a from floor to b
```

Man sieht, dass die Handlungsschritte *verflochten* sind: Der erste und dritte führen zur Realisierung des Ziels ‘‘on(a,b)’’, und der zweite realisiert das Ziel ‘‘on(b,c)’’. Wie kann man ein deartiges Verflechten der Einzelhandlungen erreichen?

Lösung von Warplan: zwei verschiedene Arten von Handlungsplanung, `achieve(Ziel,Handlung,Protected,Aktionen,NeueAktionen)`

- durch Erweiterung (**7**)
- durch Einsetzung (**8**)

Normalfall: neue Aktionen ‘‘vorne angesetzt’’ (**7**): (Voraussetzungen von Aktionen: unter ‘‘can’’ aufgelistet)

```
can(move(U,V,floor), [on(U,V), \+(V==floor), clear(U)]).
can(move(U,V,W), [clear(W), on(U,V), \+(U==W), clear(U)]).

achieve(X,U,P,T,(T1,U)) :- preserves(U,P),      %%%% 7
                           can(U,C),
                           consistent(C,P),
                           plan(C,P,T,T1),
                           preserves(U,P).
```

Bei drohender Verletzung eines geschützten Ziels: Aktion irgendwo früher, mitten in die schon geplante Sequenz von Aktionen, eingesetzt

© Michael Hess 2009

```
achieve(X,U,P,(T,V),(T1,V))                          %%%% 8
                       :- preserved(X,V),             %%%% 9
                          retrace(P,V,P1),            %%% 10
                          achieve(X,U,P1,T,T1),
                          preserved(X,V).
```

Zu beachten:

1. das *Ziel* darf durch die ''alte'' Aktion *V*, die *nach* der eingefügten Aktion zu stehen kommt, nicht zerstört werden: `preserved(X,V)` (**9**):
2. eingefügte Aktion darf die *Voraussetzungen* der ''alten'' Aktionen nicht zerstören: `retrace` (**10**):

```
retrace(P,V,P2)        :- can(V,C),
                          retrace1(P,V,C,P1),
                          append(C,P1,P2).            %% 11

retrace1([X|P],V,C,P1) :- add(Y,V),                  %% 12
                          X==Y,
                          !, retrace1(P,V,C,P1).
retrace1([X|P],V,C,P1) :- member(Y,C),               %% 13
                          X==Y,
                          !, retrace1(P,V,C,P1).
retrace1([X|P],V,C,[X|P1])                            %% 14
                       :- retrace1(P,V,C,P1).
retrace1(true,V,C,true).
```

Daher: Voraussetzungen der ''alten'' Aktion *V* werden in die Liste der geschützten Ziele der *einzuschiebenden* Aktion *P* eingefügt (in `retrace1`; **11**), wobei nur solche Ziele eingefügt werden, die

1. nicht schon dort sind (**12**)
2. nicht in der Menge der Voraussetzungen der letzten schon geplanten Aktion V sind (**13**)

Die übrigen Prädikate:

```
del(on(U,Z), move(U,Z,W)).
del(clear(W), move(U,V,W)).
```

© Michael Hess 2009

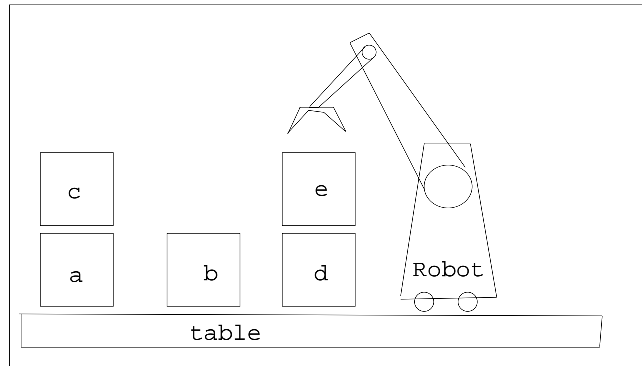
```
imposs([on(X,Y), clear(Y)]).
imposs([on(X,Y), on(X,Z), \\+(Y==Z)]).
imposs([on(X,X)]).

given(start, on(a,floor)).
given(start, on(b,floor)).
given(start, on(c,a)).
given(start, on(d,floor)).
given(start, on(e,d)).
given(start, clear(b)).
given(start, clear(c)).
given(start, clear(e)).

preserves(_,true).
preserves(U,[X|C])      :- preserved(X,U),
                           preserves(U,C).

preserved(X,V)          :- numbervars((X,V),0,N),
                           del(X,V), !, fail.
preserved(_,_).


test(N)         :- testvalue(N,V), plans(V,start,T1),
                   nl, write(T1).

testvalue(1,[on(c,b)]).
testvalue(2,[on(a,b)]).
testvalue(3,[on(a,e),on(c,a)]).
testvalue(4,[on(c,b),on(b,e)]).
testvalue(5,[on(a,b),on(b,c),on(c,d),on(d,e)]).
testvalue(6,[on(c,b),on(b,a)]).
```

Aber: Frage 6 kann nicht gelöst werden, owbohl erforderliche Sequenz von Aktionen ganz einfach:

```
move c from a to floor
move b from floor to a
move c from floor to b
```

Ausgangslage: (wiederholt)

© Michael Hess 2009



Grund für Fehlschlag:

- erstes Ziel (‘‘on(c,b)’’) wird durch Aktion ‘‘move(c,a,b)’’ erreicht. Voraussetzungen für diese Aktion

  ```
  clear(b), on(c,a), \+(c==b), clear(c)
  ```

  werden geschützt.
- Ziel ‘‘on(b,a)’’ muss vorher eingeschoben werden
- Dieses Ziel setzt aber ‘‘clear(a)’’ voraus, was mit ‘‘on(c,a)’’ kollidiert
- ‘‘on(c,a)’’ ist aber nicht eine *zu erfüllende Voraussetzung* für das Verschieben von c auf b.
- ‘‘on(c,X)’’ ist vielmehr ein *Test*, um zu ermitteln, worauf c zur Zeit liegt - es muss nicht wahr *gemacht* werden!

Deshalb unsinnig zu verlangen, dass der in einer früheren Phase der Planung zufällig ermittelte Wert ‘‘X=a’’ geschützt wird.

Lösung:

- für jede Aktion zu unterscheiden
  1. (aktiv zu realisierende) *Unterziele* (‘‘subgoals’’) und
  2. (nur zu testende resp. zu ermittelnde) *Voraussetzungen* (Ausgangswerte, Vorbedingungen, ‘‘conditions’’)

© Michael Hess 2009

```
subgoals(move(U,V,floor), [clear(U)]). % modifiziert
subgoals(move(U,V,W), [clear(W), clear(U)]). % modifiziert

conditions(move(U,V,floor), [on(U,V), +(V==floor)]).  % neu
conditions(move(U,V,W), [on(U,V), +(U==W)]).          % neu
```

Ausgangswerte werden *nicht* geschützt.

Komplikation: Ziel ist

```
[on(c,b),on(b,a)]
```

erster Versuch:

```
on(c,b)                       on(b,a)
  ↓                            ↓
move(c,a,b),                  move(b,floor,a)

  clear(c) -> clear(c)          clear(a) -> ...
  clear(b) -> \+clear(b)        clear(b) -> ... KONFLIKT
```

zweiter Versuch (Aktion ‘‘ move(b,floor,a)’’ **eingeschoben**):

```
on(b,a)                          on(c,b)
  ↓                               ↓
move(b,floor,a),                 move(c,a,b)

  clear(b) -> clear(b)             clear(b)
  clear(a) -> \+clear(a)           clear(c)
    ↓
    move(c,a,floor)

      clear(c) -> clear(c)
```

nunmehr:

```
start, move(c,a,floor), move(b,floor,a), move(c,a,b)
               1                2               3
```

Ist unsinnig: Wenn man c von a auf b stellen will (3), ist c schon lange nicht mehr

© Michael Hess 2009

auf a, sondern auf dem Boden, wohin man a gestellt hatte, um a freizumachen (1)

Grund: Werte der *Bedingungen* von schon geplanten Aktionen durch nachträglich eingeschobenen Aktion verändert

Daher: aktuelle Werte für Variablen in Bedingungen am Schluss des Zieleinfügens (neu) errechnen.

Nunmehr: korrekte Sequenz:

```
start, move(c,a,floor), move(b,floor,a), move(c,floor,b)
```

Zudem: nicht nur Aktionen festhalten, sondern auch Ziele:

```
Goal:Action
```

Vorteil: Vorliegen eines Tatbestandes direkt aus der ‘‘Geschichte’’ abzulesen

```
holds(X,(T,X:V))        :- !.
```

Kern des modifizierten Programms:

```
achieve(X,U,P,T,(T1,X:U))       % modifiziert X:U
        :-  preserves(U,P),
            conditions(U,C),% neu
            hold(C,T),      % neu; Bedingungen
                            % werden nur getestet
            subgoals(U,S),  % modifiziert
            consistent(S,P),
            plan(S,P,T,T1), % modifiziert; nur Unter-
                            % ziele werden erzeugt
            preserves(U,P).
```

© Michael Hess 2009

```
achieve(X,U,P,(T,Y:V),(T1,Y:V1))
        :-  preserved(X,V),
            retrace(P,V,P1),
            achieve(X,U,P1,T,T1),
            preserved(X,V),
            add(Y,V1),         % neu; ermittelt Beding-
            conditions(V1,C), % ungen der alten Aktion
            hold(C,T1).        % Y und errechnet sie
```

Weitere sinnvolle Differenzierungen:

1. ‘‘Ziele’’ vs. ‘‘Nebenwirkungen’’: nicht alle Effekte sind sinnvollerweise als Ziele zu behandeln
2. ‘‘Handlungen’’ vs. ‘‘Ereignisse’’: viele Ereignisse sind nicht unter der Kontrolle des Agenten

Fortgeschrittene Planungsverfahren (0.L.25)

7.3 Computerlinguistische Anwendungen von Planungsmethoden

Anwendungen von Planungsmethoden in der Computerlinguistik v.a. in Diskursanalyse und -generierung.

Diskurs = Dialog ∪ Text

Analyse:

Bestimmte Typen von Diskurs sind oft nur zu verstehen, wenn man die Intentionen der Handelnden kennt: Diskurs, der von Handlungen autonomer Agenten handelt: Geschichten, Dialoge.

Ziele sind oft nicht offensichtlich, und dann muss man sie u.U. inferieren aufgrund des sichtbaren Verhaltens: Planung *rückwärts*.

Beachte Sonderproblem: Auflösung von anaphorischen Referenzen und Ellipsen (*natürlich* auch in Texten, aber in Dialogen verbreiteter und ruchloser).

Generierung: Texte verfolgen Ziele. Unterscheidung zwischen *Diskursziel* (durch

© Michael Hess 2009

den Diskurs zu erreichen: Information gewinnen etc.) und *Bereichsziel* (z.B. mit der Information etwas tun).

7.3.1 Anwendungen von Planungsmethoden bei der Textanalyse

Im folgenden oft nach Allen 1987, Kap. 14.

Beispiel 1:

```
19) Text: Hans wollte sein neues Fahrrad
          zusammensetzen. Er suchte überall nach
          einem Schraubenzieher.
   Frage: Wozu wollte Hans den Schraubenzieher
          verwenden?
   Antwort: Um sein Fahrrad zusammensetzen.

   Frage: Wo befindet sich Hans zur Zeit?
   Antwort: Vermutlich nicht in der Nähe seines
            Fahrrads.
```

Scheint offensichtlich zu sein, ist es aber nicht. Erfordert

1. Erkennen des Ziels (der beschriebenen Handlung)
2. Wissen um die einzelnen Schritte zielführender Handlungs-Pläne, hier:
   a. um ein komplexes Ding zusammenzusetzen, benutzt man oft Werkzeuge
   b. um ein Werkzeug benutzen zu können, muss man es zur Verfügung haben
   c. wenn man ein Objekt zur Verfügung haben will, muss man wissen, wo es ist, und es dann in seine Verfügungsgewalt bringen
   d. wenn man von etwas wissen will, wo es ist, muss man es suchen
   e. wenn man etwas suchen will, muss man sich von seinem gegenwärtigen Standort wegbegeben

Beachte:

© Michael Hess 2009

Klingt alles höchst trivial, aber ist es nicht: Aufbau und Strukturierung von ‘‘common sense knowledge’’ ist etwas vom Anspruchsvollsten in der Künstlichen Intelligenz

Man kann das selbe Ziel oft auf verschiedenen Wegen erreichen. Da muss man beim Textverstehen *alle* möglichen Pläne bedenken.

Die Bezeichnungen der Objekte werden immer allgemeiner: ‘‘komplexes Ding’’, ‘‘Werkzeug’’, ‘‘Objekt’’, ‘‘etwas’’. Widerspiegelt ein wichtiges Problem bei der Planung (und v.a. beim induktiven Lernen erfolgreicher Pläne): Grad der Allgemeinheit.

Beim Verstehen *aller* Sorten von Texten ist viel Wissen erforderlich, aber spezifisch *Planungs*wissen ist erforderlich bei Texten, die von Handlungen autonomer Agenten handeln.

Beispiel 2: (Smith 1991:387)

> Louise wanted to read ‘‘Pride and Prejudice’’ or ‘‘Wuthering Heights’’. She went to the library. The computer gave her the call numbers. Louise went to the shelves, found ‘‘Pride and Prejudice’’, checked it out, and brought it home

wuther (/ w∧.../): ‘‘a gust of wind’’

‘‘call number’’: Signatur, unter der man das Buch ausleiht.

Was alles *nicht* gesagt wird:

1. dass sie in die Bibliothek ging, *um* einen der Titel auszuleihen
2. dass der Computer ihr die Signatur von *‘‘Pride and Prejudice’’ und von ‘‘Wuthering Heights’’* gab
3. *wie* sie den Computer dazu brachte, ihr die Signaturen zu geben
4. dass dieser Computer sich *in der Bibliothek* befindet
5. dass sie diese Signaturen benötigte, um das Buch auszuleihen
6. dass sie keines dieser Bücher besitzt
7. dass sich ‘‘Pride and Prejudice’’ nachher bei ihr zu Hause befindet

Arten von Wissen, womit das erschlossen werden muss:

- Sprachwissen

© Michael Hess 2009

- Weltwissen
- Planungswissen

Bei stereotypen Handlungsabläufen genügt oft ein ‘‘Skript’’:

1. Bibliotheks-Besuchs-Skript
2. Restaurant-Besuchs-Skript
3. Vorlesungs-Besuchs-Skript
4. Party-Teilnahme-Skript
5. etc. etc.

War zu Schanks Zeiten der grosse Renner. Aber:

Genügt kaum bei selteneren Arten von Ereignis, wie ‘‘Ersatz gestohlener Güter’’:

20) Jack and Sue went to a hardware store to buy a new lawnmower.

20a) Their old one had been stolen.

20b) Sue had seen the men who took it

20c) and had chased them down the street,

20d) but they’d driven away in a truck.

20e) After looking in ***the store***, ***they*** realized that ***they*** couldn’t afford a new ***one***.

Denkbare Fragen zu dieser Geschichte:

1. besassen Jack und Sue einen Rasenmäher, als sie ins Geschäft gingen?
2. wo ist der Rasenmäher?
3. *wozu* ist Sue den Dieben nachgejagt?
4. warum hat sie die Diebe nicht eingeholt?
5. *wer* realisierte, nachdem sie *wo* gewesen waren, dass sie sich *was* nicht leisten konnten? .

© Michael Hess 2009

G

7.3.1.1 Anwendungen von Planungsmethoden bei der Dialoganalyse

Bedeutung in der Computerlinguistik: Auskunftssysteme!

Beispiel 3 (Dialog):

```
21) Benutzer: Wer hält die Vorlesung
              ''Informatik 2'' im Sommersemester 1994?
    System:   Müller
    Benutzer: Welche Scheine muss ich beibringen?
    System:   ''Informatik 1'' und
              ''Hardwarepraktikum''
    Benutzer: Muss man sich in jedem Fall
              voranmelden?
```

Beispiel 21 erfordert zu erkennen

1. Benutzer will offenbar die ‘‘Vorlesung Informatik 2’’ besuchen
2. Benutzer will wissen, was alles die Bedingungen sind, an ‘‘Vorlesung Informatik 2’’ teilzunehmen
3. die einzelnen Schritte des verwendeten Dialog-Plans

Also: Bei Dialogen kann Planinferenz in zweierlei Beziehungen eingesetzt werden:

1. was will der Sprechende *in der realen Welt* erreichen, und wie?
2. was will der Sprechende *im Dialog* erreichen, und wie?
3. das Dialogziel ist ein Teilziel im ‘‘task specific plan’’, aber es ist u.U. nur aus letzterem zu erschliessen

Beachte:

Annahme: Nur der Benutzer hat (zu erkennende) Ziele; das System hat nur das Ziel, dem Benutzer bestmöglich zu helfen. Diskussionen mit mehreren Teilnehmern sind natürlich auch denkbar. Oder: Planung eines gemeinsamen Unternehmens mit Hilfe mehrere autonomer Agenten (die müssen miteinander kommunizieren).

Verwandtes Konzept: Partnermodellierung

© Michael Hess 2009

Beispiel für Rolle der Dialogplanung: (XCALIBUR; Carbonell, z.B. IJCAI 1983, ACL 1983)

REQUEST-ROLE-CORROBORATION

```
Wenn:   Das System hat eine Frage gestellt, die eine
        Bestätigung für einen vorgeschlagenen Wert
        für eine Kasusrolle verlangt

Dann:   wird eine der folgenden Reaktionen erwartet:

 1.         ein für die Frage passendes Muster für
            Bestätigung oder Zurückweisung

 2.         ein anderer als der vorgeschlagene Wert,
            der aber semantisch zulässig sein muss

 3.         ein Vergleichs- oder Bewertungsausdruck für
            den vorgeschlagenen Wert

 4.         eine Rückfrage über die Kasusrolle oder
            Restriktionen für zulässige Werte
```

Beispiel:

```
Benutzer: Add a line printer with graphics capabilities.
System:   Is 150 lines per minute acceptable?

Benutzer 1: No, 320 is better

Benutzer 2: Other options for the speed?

Benutzer 3: Too slow, try 300 or faster
```

Wichtigster Punkt: Anwendung der Planungsmethoden auf *Sprechakte* durch

1. Axiomatiserung von Sprechakten , aber
2. unter Benützung von Modaloperatoren für Wissen, Glauben, Intentionen etc.

© Michael Hess 2009

Beispiele:

‘‘A informs B of P’’ (nach Rich 1991:422):

```
INFORM(A,B,P)
    Bedingungen:    BELIEVE(A,P)
                    TRUST(B,A)
    Unterziele      KNOW(A,LOCATION(B))
    Effekte:        BELIEVE(B,P)
                    BELIEVE(B,BELIEVE(A,P))
```

1. A glaubt selbst P , und B vertraut A
2. A weiss, wo B ist
3. B glaubt P

‘‘A asks B about P’’:

```
ASK(A,B,P)
    Bedingungen:    KNOW(B,P)
                    TRUST(A,B)
    Unterziele      KNOW(A,LOCATION(B))
                    WILLING-TO-PERFORM(B,INFORM(B,A,P))
    Effekte:        KNOW(A,P)
```

Natürlich muss man da auf unendliche Schlaufen achten (B fragen, wo er ist, um ihn etwas fragen zu können).

Kenntnis von Dialog-Plänen kann auch dazu dienen, indirekte Sprechakte zu erkennen:

Verwendung von *konversationellen Postulaten:* (± nach Grice)

1. *Aufrichtigkeitspostulat:* Damit eine Aufforderung von A an B, R zu tun, *aufrichtig* ist, muss
   a. A *wollen*, dass B R tut
   b. A annehmen, dass B R tun *kann*
   c. A annehmen, dass B *bereit ist*, R zu tun
   d. B *nicht ohnehin* R tun wollte

   Wenn A B nach dem Erfülltsein einer dieser Bedingungen *fragt*, heisst das in

© Michael Hess 2009

der Regel, dass A B indirekt um R *ersucht*.

Beispiel:

18) Wolltest du nicht die Tür öffnen?

2. *Begründungspostulat:* Damit eine Aufforderung von A an B, R zu tun, *vernünftig* ist, muss
   a. A einen Grund dafür haben zu *wollen*, dass B R tut
   b. A einen Grund dafür haben anzunehmen, dass B R tun *kann*
   c. A einen Grund dafür haben anzunehmen, dass B *bereit ist*, R zu tun
   d. A einen Grund dafür haben anzunehmen, dass B *nicht ohnehin* R tun wollte

   Wenn B A nach dem Erfülltsein einer dieser Bedingungen *fragt*, heisst das manchmal, dass B die *Berechtigung* von Ás Wunsch in Frage stellt.

   Beispiel:

   18) Kannst du die Tür öffnen?

   18a) Warum willst du sie offen haben?

3. *Angebrachtheitspostulat:* Damit eine Aussage angebracht ist, muss sie
   a. genügend Information geben
   b. die Überzeugung des Sprechers korrekt wiedergeben
   c. konzis sein
   d. eindeutig sein

   Aus dem Nicht-Sagen kann man meist ein Nicht-Wissen ableiten:

   23) A: Wer hat das Rennen gewonnen?

   23a) B: Jemand mit langen dunklen Haaren

   23b) A: Ich glaubte, du kennst alle Reiter

© Michael Hess 2009

7.3.1.2 Planung zur Auflösung von Ellipsen und anaphorischen Referenzen

V.a. in Dialogen wird sehr viel weggelassen. Vergleiche (Frage an einen Bahnbeamten):

24) Wo kommt der Bonner Zug an?

25) Wo fährt der Bonner Zug ab?

Was soll ‘‘Bonner Zug’’ heissen? ‘‘Zug *aus* Bonn’’ oder ‘‘Zug *nach* Bonn’’? Ist zu erschliessen: In 24 sehr wahrscheinlich ‘‘Zug *aus* Bonn’’, in 25 ‘‘Zug *nach* Bonn’’.

Oder: (atemloser Reisender mit Koffer)

26) Der Zug nach Mainz?!

‘‘Wo fährt ... ab?’’ ist zu erschliessen.

Wie? Intentionen eines typischen Bahnhofsbenutzers (**beachte:** *typisch*, *Intention*). Und: Was muss man tun, um die Intention zu realisieren? Planwissen. Also erneut: Erschliessen des Ziels, Erschliessen des verwendeten Plans.

Typen von Ellipsen: (intra-sententielle Ellipsen)

1. syntaktische Ellipsen
   a. Rückwärtsellipsen:

      27) Hans suchte ____ und fand schliesslich seinen Schlüssel

   b. Vorwärtsellipsen:

      28) Hans ass einen Apfel, Peter ___ eine Birne, und Anna ___ eine Orange

   Grundsätzliche Rekonstruktionsprinzipien: Die für die Analyse von Fernabhängigkeiten benutzten Methoden

2. pragmatische Ellipsen
   a. situative Ellipsen:

      29) Frisch gestrichen!

      30) Donnerstag Ruhetag

© Michael Hess 2009

31) Wolfshund frisst Baby!

b. kotextuelle Ellipsen:

i. Wiederholungsellipsen:

32) Gibt es eine direkte Bahnverbindung von Koblenz nach Passau?

32a) Ja, aber nur einmal die Woche

32b) ___ Nur einmal die Woche?!

Rekonstruktionsprinzip: Speichern der (±) *Oberflächenstruktur* der letzten Eingabe; Kopieren eines kohärenten Teils davon

ii. Ersetzungsellipsen:

33) Fährt ein Bus vom Bahnhof nach Waldesch?

33a) Ja, vormittags und nachmittags.

33b) Und ___ Sonntags?

Rekonstruktionsprinzip: Speichern der *Syntaxstruktur* der letzten Eingabe(n); Kopieren eines kohärenten Teils davon (oder mehrerer Teile)

iii. Expansionsellipse:

32) Gibt es eine direkte Bahnverbindung von Koblenz nach Passau?

32c) Nein

32d) Aber ___ von München ___ ?

32e) Ja

32f) Und ___ von *hier* nach München?

Rekonstruktionsprinzip: Speichern der *semantischen Struktur* der letzten Eingabe(n); Kopieren eines Teils davon (oder mehrerer Teile)

Hier immer nur Ellipsen in den Fragen des Benutzers; dass System kann (und sollte, um akzeptabel zu sein) ebenfalls Ellipse benutzen.

© Michael Hess 2009

7.3.2 Planung bei der Generierung von Sprache

Generierung von Sprache lange Zeit in der Forschung über Verarbeitung natürlicher Sprachen eher vernachlässigt. Beginnt sich derzeit als eigenständiges Gebiet zu etablieren. Der Antrieb kommt aus einer Reihe von Quellen. Zum einen können es theoretische Motive sein:

Neu erwachtes Interesse:

- Generierung als psycholinguistisch fundierte Simulation menschlicher Sprachprozesse
- Generierung als Technik zum Austesten linguistischer Grammatikformalismen und Pendant zu entsprechenden Parsern
- Generierung als Gebiet zum Erproben neuer informatischer Paradigmen

Daneben gibt es auch ‘‘Praxisbedürfnisse’’:

- das neuerliche Interesse an maschineller Übersetzung
- das Bedürfnis nach natürlichsprachlichen Erklärungs- und Hilfe-Komponenten, z.B. für Datenbank- oder Expertensysteme

Notwendigkeit *sinnvoller* Generierung für Erklärungs- und Hilfe-Komponenten ist evident.

Beispiele:

1. alle Arten von *Auskunftssystemen* mit natürlichsprachlichem Output

```
awk: syntax error near line 1
awk: illegal statement near line 1
```

2. Texte ab Wissensbasis generieren; Beispiel: TechDoc (FAW Ulm)
   a. immer der aktuellste Stand
   b. multilingual
   c. später auch: Input direkt vom CAD-System
   d. später auch: Simulation in der Wissensbasis, und Generieren ab Simulation

© Michael Hess 2009

3. Beschreibung von Szenen
4. Beschreibung von Situationen
5. Beschreiben von Inferenzketten
6. Verfassen von Zusammenfassungen von Texten
7. Verfassen von Geschichten

Grundprobleme:

1. aus den ggf. vielfältigen Elementen einer Situation ist eine Spezifikation des Inhalts der zu generierenden Äusserung bzw. des zu generierenden Texts zu extrahieren
2. aus der ‘‘message’’ ist eine linguistische Spezifikation zu gewinnen, aus der sich die endgültige sprachliche Form der Äusserung bestimmt.

Also:

1. *was* man sagt: *Inhalt*
2. *wie* man es formuliert: *Form*

Recht oft nicht scharf zu trennen.

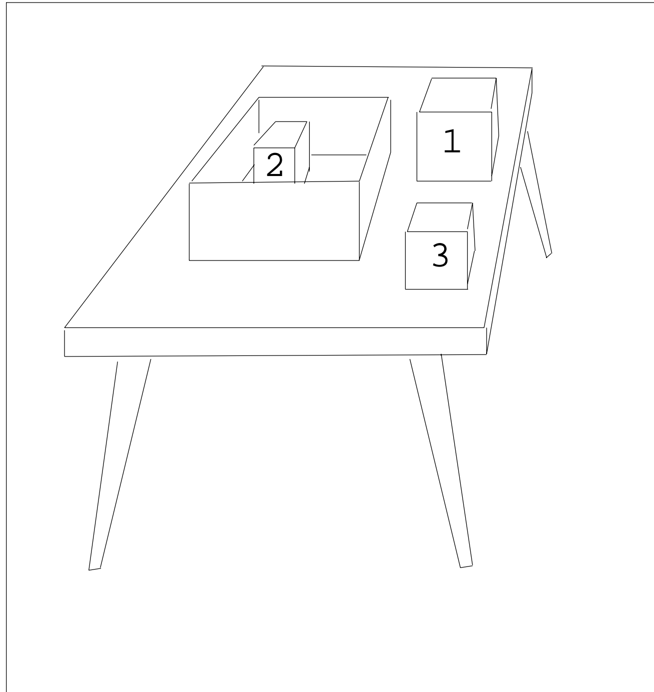
7.3.2.1 Das Auswahlproblem

Andere Bezeichnungen:

1. strategische Komponente
2. konzeptuelle Spezifizierung
3. ‘‘what-to-say’’-Komponente

Einfachster Fall:

© Michael Hess 2009



Intern repräsentiert als:

```
block(block1).
block(block2).
block(block3).
box(box1).
table(table1).

on(block1,table1).
on(box1,table1).
on(box3,table1).

in(block2,box1).
```

Wie referiert man auf Klotz 1? Auf die Kiste? Auf Klotz 2?

Beachte: Die Klötze tragen keine sichtbaren Nummern!

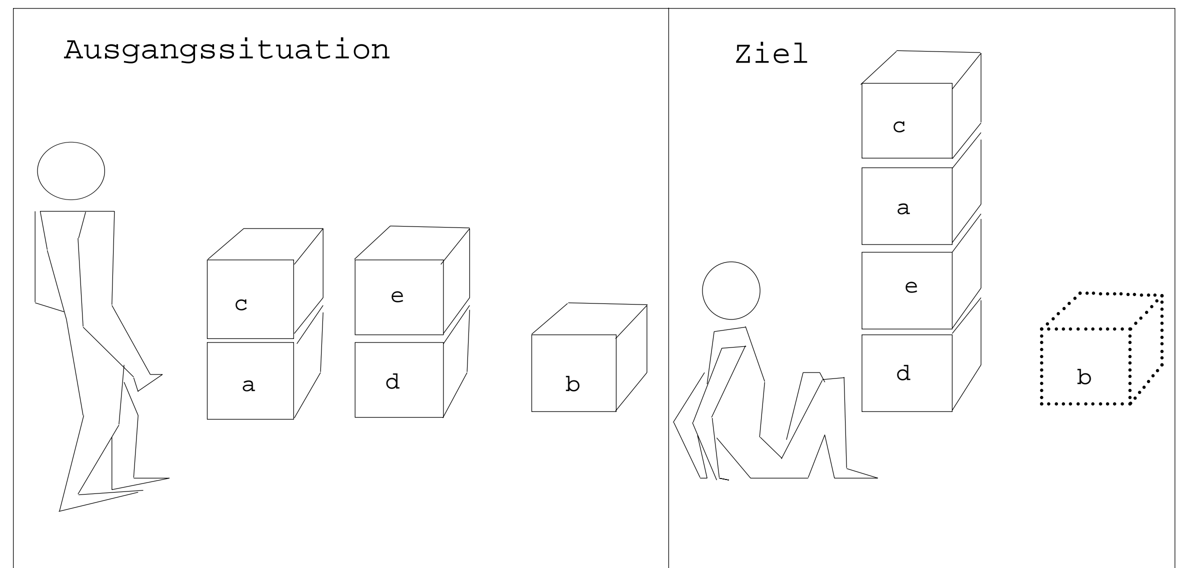
1. ‘‘the block on the table’’ ist nicht eindeutig
2. ‘the block in the box on the table’’ ist redundant

Das ist eines der wichtigsten Darstellungsprobleme! Siehe unten.

© Michael Hess 2009

Schwieriger: Beschreiben einer Problemlösung (z.B. durch einen Roboter):

```
Ausgangssituation:

on(a,floor).
on(b,floor).
on(c,a).
on(d,floor).
on(e,d).
clear(b).
clear(c).
clear(e).

block(a).
block(b).
block(c).
block(d).
block(e).

Ziel:

on(a,e), on(c,a).
```



Die Problemlösung:

© Michael Hess 2009

```
The sequence of moves is:
 ((start,move(c,a,floor)),move(a,floor,e)),move(c,floor,a)
```

Probleme:

1. erneut: wie auf die Objekte referieren?
2. nur die beobachtbaren Handlungen beschreiben?
3. nur die erfolgreichen Handlungen beschreiben?

ad 1:

```
``Der Klotz ganz rechts auf dem Boden''
``Der Klotz, der auf dem am nächsten beim Menschen auf dem
             Boden liegenden Klotz liegt''
``Der Klotz, den er zuletzt auf den Boden gelegt hat''
```

Indexikalische Beschreibungen (relativ zum Sprecher, zum Angesprochenen, zum Beschriebenen?), relative Beschreibungen (relativ zu andern Objekten), Bezugnahme auf Handlungen.

ad 2: Einfach die exakte Handlungsabfolge auf Englisch formulieren?

```
He moved c from a to the floor,
then he moved a from the floor onto e,
then he moved c from the floor onto a
```

Korrekt, aber:

- redundant
- überspezifiziert
- unmotiviert: *weshalb* räumt man c von a weg?

Erforderlich oft: Angabe von *Zielen:*

In order to get access to a, remove c

Oder: (Frage an ein Unix-Beratunsgsystem)

© Michael Hess 2009

```
Wie vertausche ich die Namen zweier Dateien?

mv <filename1> <filename.tmp>
mv <filename2> <filename1>
mv <filename.tmp> <filename1>
```

Eine Begründung wäre sehr hilfreich, weil der Benutzer sonst nie versteht, was er tut (und nichts wirklich lernen kann).

Dazu erforderlich: *Erschliessen* des vom Handelnden ausgeführten Plans.

Probleme dabei:

1. *was* vom Plan soll man alles berichten:
   a. ‘‘um *die zwei Klötze* `c` und `a` auf Klotz `e` zu bringen...’’
   b. ‘‘da er *aber* nur einen einzigen Klotz aufs Mal heben *kann*...’’

2. *wie* erschliesst man den Plan?

7.3.2.2 Das Darstellungsproblem

Hat man sich für den Inhalt entschieden: Wie sage ich es?

Das *wie* kann sehr wichtig sein: Schilderung eines Verkehrsunfalls gegenüber seiner Freundin resp. gegenüber dem Richter

Andere Bezeichnungen:

1. taktische Komponente
2. sprachliche Realisierung
3. ‘‘how-to-say’’-Komponente

Manche Systeme gehen davon aus, der erste Schritt (Auswahl) sei schon getan: ‘‘taktische Generatoren’’:

Man hat schon eine (z.B. logische) Repräsentation eines Sachverhalts und will das sprachlich ausdrücken.

© Michael Hess 2009

Beispiel für taktische Generierung (Semsyn):

Englische Eingabe:

i have a pain in the throat

Semantische Struktur als Schnittstelle zum SEMSYN-Generator:

```
(*HAVE-A-SYMPTOM :MOOD DEC

   :AGENT (*PATIENT :HUMAN + :PRO 1 :NUMBER SG :PERSON 1)
   :TIME PRESENT

   :SYMPTOM (*PAIN :LOCATION (*BODY-PART :NAME *THROAT)))
```

Generierungsergebnis:

Ich habe Schmerzen im Rachen.

Beispiel für vollständige Generierung (StockReporter):

Ausgangspunkt: Die Dow-Jones-Werte eines Börsentages für eine Firma (Microsoft):

```
04/10/96        103     101.25  101.625 32444   -74     5485
04/09/96        104     101.5   101.625 41839   -33     5560
04/08/96        103.875 101.875 103.75  46096   -88     5594
04/05/96        Holiday
04/04/96        104.875 103.5   104.375 18101   -6      5682
```

Ergebnis:

Stock Report for 4/4/1996

Microsoft avoided the downwards trend of the Dow Jones average today. Confined trading by all investors occurred today. After shooting to a high of $104.87, its highest price so far for the month of April, Microsoft stock eased to finish at an enormous $104.37. The Dow closed after trading at a weak 5682, down 6 points.

© Michael Hess 2009

Quelle: http://www.mri.mq.edu.au/stockreporter/ (Language Technology Group at MacquarieUniversity's Microsoft Research Institute).

Taktische Generatoren:

1. BABEL (1975: Goldman): Erwartet als Eingabe CD-Graphen, um Zusammenfassungen für vorverarbeitete Texte zu generieren
2. PHRED (= PHRasal English Dictionary, 1984: Jacobs): Wird vor allem im Rahmen des UNIX Consultant (UC) eingesetzt. PHRED generiertHilfeinformation in Form englischer oder spanischer Einzelsätze.
3. MUMBLE (1983: McDonald): KI-Programme sollten ihre Resultate in englischer Sprache präsentieren können. Beispiel: Isolierte Formeln und komplette Beweise des Prädikatenkalküls als englische Texte wiedergeben.
4. SEMSYN (1986: Rösner): Generierung von Deutsch aus semantischen Strukturen. Eingabe: erweiterte Kasusrahmen-Notationen. Im Rahmen semantikbasierter maschineller Übersetzung, aber auch als Komponente von Textgeneratoren eingesetzt.

Vollständige Generatoren:

1. SEMTEX (1986: Rösner): Generierung von Zeitungsmeldungen.

   Ausgangspunkt: Daten der Statistik der Nürnberger Bundesanstalt für Arbeit.

   Ergebnis: Texte wie der folgende:

   > Geringfügige Reduzierung der Arbeitslosenzahl
   > NÜRNBERG/BONN (cpa)
   >
   > Die Zahl der Arbeitslosen in der Bundesrepublik Deutschland hat sich während des Oktober nur sehr wenig verringert. Sie ist von 2151600 auf 2148800 zurückgegangen. Die Arbeitslosen- quote hatte Ende Oktober einen Wert von 8.6 Prozent. Sie hatte am Ende des Verg1eichzeitraumes des Vorjahrs ebenfalls bei 8.6 Prozent gelegen.

Gesichtspunkte:

1. Linearisierung komplexer Zusammenhänge:
   a. mehrere einfache Sätze vs. ein komplexer Satz

© Michael Hess 2009

b. explizite vs. implizite Darstellung temporaler, kausaler etc. Beziehungen:

34) Das Auto ist kaputt. Peter hat es gegen eine Mauer gefahren

34a) Das Auto ist kaputt weil Peter es gegen eine Mauer gefahren hat

2. syntaktische Form der Sätze:

a. aktiv vs. passiv:

```
Benutzer:   Wo sind die Baupläne für die
            Laugenpumpe?

System:     Weg. Müller hat sie vernichtet.

System:     Weg. Sie sind vernichtet worden.
```

b. Herstellung der Textkohärenz, z.B. durch Ellipsen- und Anapherngenerierung:

```
Benutzer: Welche Schiffe der britischen Home Fleet
          haben zur Zeit Kurs auf Portsmouth?

System:   Die HMS ''Invincible'' und die
          Begleitschiffe der ''Invincible''

System:   Die HMS ''Invincible'' und ihre
          Begleitschiffe

System:   Alle _____.
```

c. Fokusverwaltung: neutrale Form vs. Topikalisierung:

```
Benutzer: Welche Papiere wurden vernichtet?

System:   Es waren nur die unwichtigen
          Papiere, welche vernichtet wurden.
```

3. Wortwahl:

© Michael Hess 2009

a. kaufen vs. verkaufen

b. schreiben vs. kritzeln

4. Generierung referierender Ausdrücke

a. anaphorische Begriffe: Pronomina, definite Nominalphrasen (siehe oben)

b. Nominalphrasengenerierung

i. Individuenkonstanten:

```
Benutzer: Welcher Flugzeugträger liegt
          zur Zeit in Portsmouth?

System:   Die HMS ``Invincible''

System:   sk00214
```

ii. indefinite Nominalphrasen vs. definite Nominalphrasen

```
Benutzer: Welche Schiffe haben zur Zeit
          Kurs auf Portsmouth?

System:   Ein Flugzeugträger der
          amerikanischen Flotte

System:   Der Flugzeugträger der
          britischen Flotte
```

iii. Nominalphrase mit: Relativsatz, Präpositionalphrase, adjektivischer o.a. Modifikation, Komposition etc. etc.

35) Die Baupläne, welche die Laugenpumpe beschreiben

35a) Die Baupläne für die Laugenpumpe

35b) Die Baupläne der Laugenpumpe

35c) Die Laugenpumpenbaupläne

iv. abgeleitete Mengenbegriffe:

© Michael Hess 2009

```
Benutzer: Welche Schiffe haben zur Zeit
          Kurs auf Portsmouth?

System:   Die  HMS ''Invincible'', die HMS
          ''Hood'', die HMS ''Essex'', die
          HMS ''Prince Edward'',  etc. etc.

System:   Ein Flugzeugträger, zwei Schlacht-
          schiffe und fünf Zerstörer

System:   alle Schiffe der britischen
          Home Fleet

System:   acht Schiffe

System:   einige Schiffe
```

5. Diskursstruktur: Organisation des Texts, meta-textuelle Referenzen (siehe unten)

Allgemein:

1. was ist *selbstverständlich*?
2. was kann man aus *allgemeinem* Weltwissen ableiten?
3. was weiss der *konkrete* Benutzer (vermutlich)?
4. was kann der Benutzer aus dem (situativen) *Kotext* ableiten?
5. was kann der Benutzer aus dem (sprachlichen) *Kontext* ableiten?
6. wie *detailliert* muss der generierte Output sein?

Beispiel: SEMTEX.

1. Textplan in Form einer Liste semantischer Repräsentationen erstellen.
2. den durch bereits generierte Sätze gegebenen Kontext benutzen
   - um Wortwiederholungen zu vermeiden (vg1. Satz l: ‘‘sich verringern’’, Satz 2: ‘‘zurückgehen’’ für das semantische Symbol DECREASE, Satz 3: ‘‘einen Wert haben von’’, Satz 4: ‘‘liegen bei’’ für HAVE-VALUE),

© Michael Hess 2009

- um redundante Information bewusst elidieren zu können (z.B. keine Wiederholung der Zeitangabe in Satz 2),
- um über Pronominalisierung und andere Arten der Referenz zu entscheiden.

Bestimmung der grammatischen Zeit Generierung geeigneter Benennungen für Zeitangaben

© Michael Hess 2009

8. Zitierte Literatur

[Allen 1987] Allen, J., *Natural Language Understanding,* Benjamin/Cummings, Menlo Park etc., 1987.

[Alshawi 1992] Alshawi, Hiyan (ed.), *The Core Language Engine,* ACL-MIT Press Series in Natural Language Processing, MIT Press, Cambridge, Mass/London, 1992.

[Bratko 1989] Bratko, Ivan, Mozetic, Igor, and Lavrac, Nada, *KARDIO: a study in deep and qualitative knowledge for expert systems,* MIT Press, Cambridge, Mass, 1989.

[Bratko 1986] Bratko, I., *Prolog Programming for Artificial Intelligence,* International Computer Series, Addison Wesley, Wokingham etc., 1986.

[Carroll 1992] Carroll, John and Briscoe, Ted, ''Probabilistic Normalisation and Unpacking of Packed Parse Forests for Unification-based Grammars,'' in: *AAAI Fall Symposium on Probabilistic Approaches to Natural Language*, 1992.

[Charniak 1985] Charniak, E. and McDermott, D., *Introduction to Artificial Intelligence,* Addison-Wesley, Reading, MA. etc., 1985.

[Colmerauer 1987] Colmerauer, A., *Les bases de Prolog III*, 1987.

[Copestake 1999] Copestake, Ann, Flickinger, Dan, Sag, Ivan A., and Pollard, Carl, ''Minimal Recursion Semantics: An introduction,'' http://www-csli.stanford.edu/~aac/papers/newmrs.ps, 1999.

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

[Dörre 1997] Dörre, Jochen, ''Efficient Construction of Underspecified Semantics under Massive Ambiguity,'' *in: Proceedings of the ACL, 35th Annual Meeting*, pp. 386-393, 1997.

[Edwards 1967] Edwards, Paul ed. in chief, *The encyclopedia of philosophy,* Macmillan & Free Press, New York, 1967.

© Michael Hess 2009

[Fodor 1980] Fodor, J.D. and Frazier, L., ‘‘Is the human sentence parsing mechanism an ATN?,’’ *Cognition*, vol. 8, no. 4, pp. 417-459, December 1980.

[Frazier 1978] Frazier, L. and Fodor, J.D., ‘‘The sausage machine: A new two-stage parsing model,’’ *Cognition*, vol. 6, no. 4, pp. 291-325, December 1978.

[Frost 1986] Frost, R., *Introduction to Knowledge Base Systems,* Collins, London, 1986.

[Gal 1991] Gal, Annie, Lapalme, Guy, Saint-Dizier, Patrick, and Somers, Harry, *Prolog for Natural Language Processing,* Wiley, Chichester etc., 1991.

[Genesereth 1987] Genesereth, M.R. and Nilsson, N.J., *Logical Foundations of Artificial Intelligence,* Morgan Kaufmann, Los Altos, CA, 1987.

[Georgeff 1987] Georgeff, M.P., ‘‘Planning,’’ in: *Annual Review of Computer Science, Vol. II*, ed. Traub, J.F. et al, pp. 359-400, 1987.

[Hobbs 1988] Hobbs, Jerry R. and Stickel, Mark, ‘‘Interpretation as Abduction,’’ in: *Proceedings of the 26th Conference of the Association for Computational Linguistics*, pp. 95-103, Buffalo, June 1988.

[Hobbs 1993] Hobbs, Jerry R., Stickel, Mark E., Appelt, Douglas E., and Martin, Paul, ‘‘Interpretation as Abduction.,’’ *Artificial Intelligence*, vol. 63, no. 1-2, pp. 69-142, 1993.

[Jackson 1985] Jackson, P., *Introduction to Expert Systems,* International Computer Science Series, Addison-Wesley, Wokingham etc., 1985.

[Josephson 1994] Josephson, J. R. and Josephson, S. G. (Eds.), *Abductive Inference: Computation, Philosophy, Technology,* Cambridge University press., New York, 1994.

[Jourdan 1988] Jourdan, J., ‘‘Quelques Applications de Prolog III,’’ Groupe Intelligence Artificelle, Faculté de Luminy Marseille Mémorie de D.E.A. , October 1988.

[Kay 1999] Kay, Martin, ‘‘Chart translation,’’ *in: Proceedings of MT Summit VII. ‘‘MT in the Great Translation Era’’*, pp. 9-14, Kent Ridge Digital Labs, Singapore, 1999.

[Kimball 1973] Kimball, J., ‘‘Seven principles of surface structure parsing in natural language,’’ *Cognition*, vol. 2, no. 1, pp. 15-47, 1973.

© Michael Hess 2009

[Koller 1999] Koller, Alexander and Niehren, Joachim, *Scope Underspecification and Processing*, Lecture Notes, ESSLLI '99, Utrecht, September 1999.

[Kraan 1989] Kraan, I., *Das Parsen erweiterter Definite Clause Grammars*, August 1989. Semesterarbeit, Institut für Informatik der Universität Zürich

[Lehner 1990] Lehner, Ch., ''Constraint Logic Programming for Natural Language Analysis I,'' Universität München, Centrum für Informations- und Sprachverarbeitung CIS-Bericht 90-25 , 1990.

[Norvig 1990] Norvig, Peter and Wilensky, Robert, ''A Critical Evaluation of Commensurable Abduction Models for Semantic Interpretation,'' in: *Coling-90; Proceedings of the 13th International Conference on Computational Linguistics*, vol. 2, pp. 225-230, Helsinki, 1990.

[Pereira 1980] Pereira, Fernando C. N. and Warren, David H. D., ''Definite Clause Grammars for Language Analysis - A Survey of the Formalism and a Comparison with Augmented Transition Networks,'' *Artificial Intelligence*, no. 13, pp. 231-278, 1980.

[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.

[Pinkal 1999] Pinkal, Manfred, *On Semantic Underspecification,* ITK, Tilburg University, Tilburg, 1999.

[Rapaport 1987] Rapaport, W.J., ''Logic,'' in: *Stuart C. Shapiro, ed., Encyclopedia of Artificial Intelligence*, vol. 1, John Wiley, 1987.

[Rich 1991] Rich, E. and Knight, K., *Artifcial intelligence, 2n (ed.),* McGraw-Hill, New York, 1991.

[Ross 1989] Ross, Peter, *Advanced Prolog: Techniques and Examples,* Addison-Wesley, Wokingham, England, 1989.

[Rowe 1988] Rowe, Neil C., *Artificial Intelligence through Prolog,* Prentice Hall, New Jersey, 1988.

[Russell 1995] Russell, Stuart J. and Norvig, Peter, *Artificial Intelligence: A Modern Approach,* Prentice Hall, Englewood Cliffs, N.J., 1995.

© Michael Hess 2009

[Shapiro 1987] Shapiro, Stuart C., *Encyclopedia of Artificial Intelligence,* John Wiley, 1987.

[Smith 1991] Smith, George W., *Computers and Human Language,* Oxford University Press, New York/Oxford, 1991.

[Warren 1974] Warren, D.H.D., ''Warplan: A System for Generating Plans,'' Dept. of Computational Logic, University of Edinburgh School of Artificial Intelligence Memo 76 , June 1974.

[Wos 1984] Wos, Larry, Overbeek, Ross, Lusk, Ewing, and Boyle, Jim, *Automated Reasoning, Introduction and Application,* Prentice Hall, 1984.

© Michael Hess 2009

9. Inhaltsverzeichnis

1. Beschreibung der Veranstaltung 2

2. Ressourcen 3
2.1 Literatur 3
2.2 Informationen auf dem Internet 4

3. Was ist Künstliche Intelligenz? 4
3.1 Einige Typen von KI-Systemen 5
3.2 ''Alte'' und ''Neue'' Künstliche Intelligenz 6
3.3 Komponenten der Intelligenz 8
3.3.1 Informationsgewinnung 8
3.3.2 Informationsverdichtung 9
3.3.3 Informationstransformation 9

4. Inferenzverfahren 10
4.1 Deduktion 10
4.1.1 Grundsätzliches 10
4.1.2 Ein allgemeines Anwendungsbeispiel: Automatisches Theorembeweisen 13
4.1.3 Ein computerlinguistisches Anwendungsbeispiel: ''Parsing as deduction'' 14
4.2 Abduktion 15
4.2.1 Grundsätzliches 15
4.2.2 Ein allgemeines Anwendungsbeispiel: Expertensysteme 18
4.2.2.1 Was sind Expertensysteme? 19
4.2.2.2 Problemlösung in ''seichten'' Expertensystemen 20
4.2.2.3 Problemlösung in ''tiefen'' Expertensystemen 25
4.2.2.4 Verhältnis zwischen ''seichten'' und ''tiefen'' Expertensystemen 30
4.2.3 Ein computerlinguistisches Anwendungsbeispiel: Interpretation durch Abduktion 31
4.2.3.1 Referenzauflösung durch Abduktion 31
4.2.3.2 Kompositaauflösung durch Abduktion 32
4.2.3.3 Desambiguierung struktureller Mehrdeutigkeiten durch Abduktion 35
4.3 Induktion 36
4.4 Deduktion, Abduktion und Induktion im Vergleich 37

5. Problemlösungsstrategien 38
5.1 Kontrolle von Inferenzprozessen 38

5.2 ‘‘Suche’’ als Abstraktion von Problemlösungsverfahren 40
5.3 Allgemeine Suchstrategien 46
5.3.1 Einige reine Strategien 47
5.3.1.1 Absteigende rechtsläufige Tiefensuche 47
5.3.1.2 Aufsteigende rechtsläufige Tiefensuche 51
5.3.1.3 Aufsteigende rechtsläufige Breitensuche 54
5.3.1.4 Absteigende rechtsläufige Breitensuche 56
5.3.2 Einige gemischte Strategien 56
5.3.2.1 ‘‘Left-corner’’-Analyse 56
5.3.2.2 ‘‘Iterative Deepening’’ 58
5.3.3 Strategien ohne Backtracking 59
5.3.3.1 Verwendung kanonischer Lösungen 60
5.3.3.2 Gepackte Strukturen 62
5.3.3.3 ‘‘Constraint’’-Sprachen 65
5.4 Bereichsspezifische Suchstrategien 71
5.4.1 Tiefensuche + Evaluationsfunktion = ‘‘Hill-Climbing’’ 72
5.4.2 Breitensuche + Evaluationsfunktion = ‘‘Best-First’’-Suche 74
5.4.3 Breitensuche + Kostenfunktion = ‘‘Branch-and-bound’’-Suche 78
5.4.4 Breitensuche + Evaluationsfunktion + Kostenfunktion = ‘‘A*’’-Suche 80

6. Logik-basierte Problemlösung 85
6.1 Mehr Flexibilität: Meta-Interpretion in logik-basierten Systemen 85
6.1.1 Implementation von Meta-Interpretern in der Logik-Programmierung 87
6.1.2 Mögliche Kriterien für die Formulierung von Meta-Regeln 89
6.1.2.1 Beweisreihenfolge von Teilzielen: Schwierigste Teilziele zuerst beweisen 90
6.1.2.2 Beweisreihenfolge von Teilzielen: Verzögern der Evaluation von Termen mit ungebundenen Variablen 92
6.1.2.3 Beweisreihenfolge von Teilzielen: Abhängigkeitsgesteuertes Backtracken 93
6.1.2.4 Verwendungsreihenfolge von Regeln: Leichteste Regel zuerst verwenden 96
6.2 Mehr Effizienz: Speicherung von Zwischenresultaten 98
6.3 Mehr Effizienz: Techniken der Compilation in logik-basierten Systemen 100
6.3.1 Entfalten von Programmen 102
6.3.2 Partielle Evaluation 112
6.3.3 Sekundäre Compilation 115
6.4 Mehr Effizienz: Parallele Abarbeitung in logik-basierten Systemen 118

7. Planung 121
7.1 Unterschiede zwischen Suche und Planung 121
7.2 Grundlegende Planungsverfahren 123

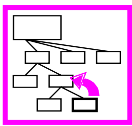
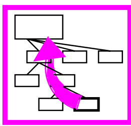
7.2.1 Die Klötzchenwelt als Modell 123
7.2.2 Die Beschreibung von partiellen Situationen und das 'Frame'-Axiom 128
7.2.3 'ADD'- und 'DELETE'-Listen zur Beschreibung des Effekts von Operatoren 131
7.2.4 Zielinteraktionen und ihre Behandlung 132
7.3 Computerlinguistische Anwendungen von Planungsmethoden 141
7.3.1 Anwendungen von Planungsmethoden bei der Textanalyse 142
7.3.1.1 Anwendungen von Planungsmethoden bei der Dialoganalyse 145
7.3.1.2 Planung zur Auflösung von Ellipsen und anaphorischen Referenzen 149
7.3.2 Planung bei der Generierung von Sprache 151
7.3.2.1 Das Auswahlproblem 152
7.3.2.2 Das Darstellungsproblem 156

8. Zitierte Literatur 163

9. Inhaltsverzeichnis 167

© Michael Hess 2009

© Michael Hess 2009

Zeitschriften:

1. Artificial Intelligence
2. Künstliche Intelligenz
3. Computational Intelligence
4. Journal of Logic Programming
5. Data & Knowledge Engineering
6. Angewandte Informatik
7. International Journal of Intelligent Systems

Konferenzberichte:

1. Proceedings of the *Nth* International Joint Conference on Artificial Intelligence (IJCAI-*N*)
2. Proceedings of the *Nth* European Conference on Artificial Intelligence (ECAI-*N*)
3. Proceedings of the *Nth* Conference of the American Association for Artificial Intelligence (AAAI-*N*)
4. Proceedings of the *Nth* IEEE conference on Artificial Intelligence
5. Proceedings of the *Nth* Biennial Conference of the Canadian Society for Computational Studies of Intelligence
6. Proceedings of the *Nth* Canadian Conference on Artificial Intelligence
7. Proceedings of the International Conference on Fifth Generation Computer Systems
8. Expert Systems & their Applications: *Nth* International Workshop

Ende des Unterdokuments

© Michael Hess 2009

Zwei weitere gemischte Strategien, die wir nur erwähnen, sind:

0.0.0.1 ‘‘Middle-Out’’

Bei der Erkennung gesprochener Sprache ist es oft nicht möglich, eine eindeutige Segmentierung des Signals durchzuführen, bevor man mit der Erkennung von Wortformen (und Sätzen) anfängt. In diesem Fall ist es sinnvoll, bei einem erkannten Teil (irgendwo in der Mitte) anzufangen und sich dann nach beiden Seiten vorzuarbeiten.

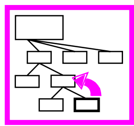
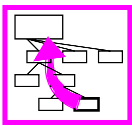
0.0.0.2 ‘‘Means-Ends Analysis’’

Prinzip: Zuerst die Hauptteile eines Problems lösen und dann zurückgehen und die Resultate (ggf. nach Anbringen kleinerer Modifikationen an den Teillösungen) zur Gesamtlösung kombinieren. Da diese Technik vor allem im Bereich der *Planung* verwendet wird, wird sie dort näher beschrieben.

Hierzu: Rowe 1988, Rich 1991:94.

Ende des Unterdokuments

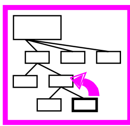
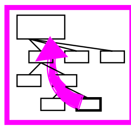
© Michael Hess 2009

Eine Darstellung, welche den Begriff ‘‘packed forest’’ noch verständlicher macht, verwendet Kästchen, um ‘‘oder’’-Knoten darzustellen[1] (anderes Beispiel!):

1. aus Dörre 1997

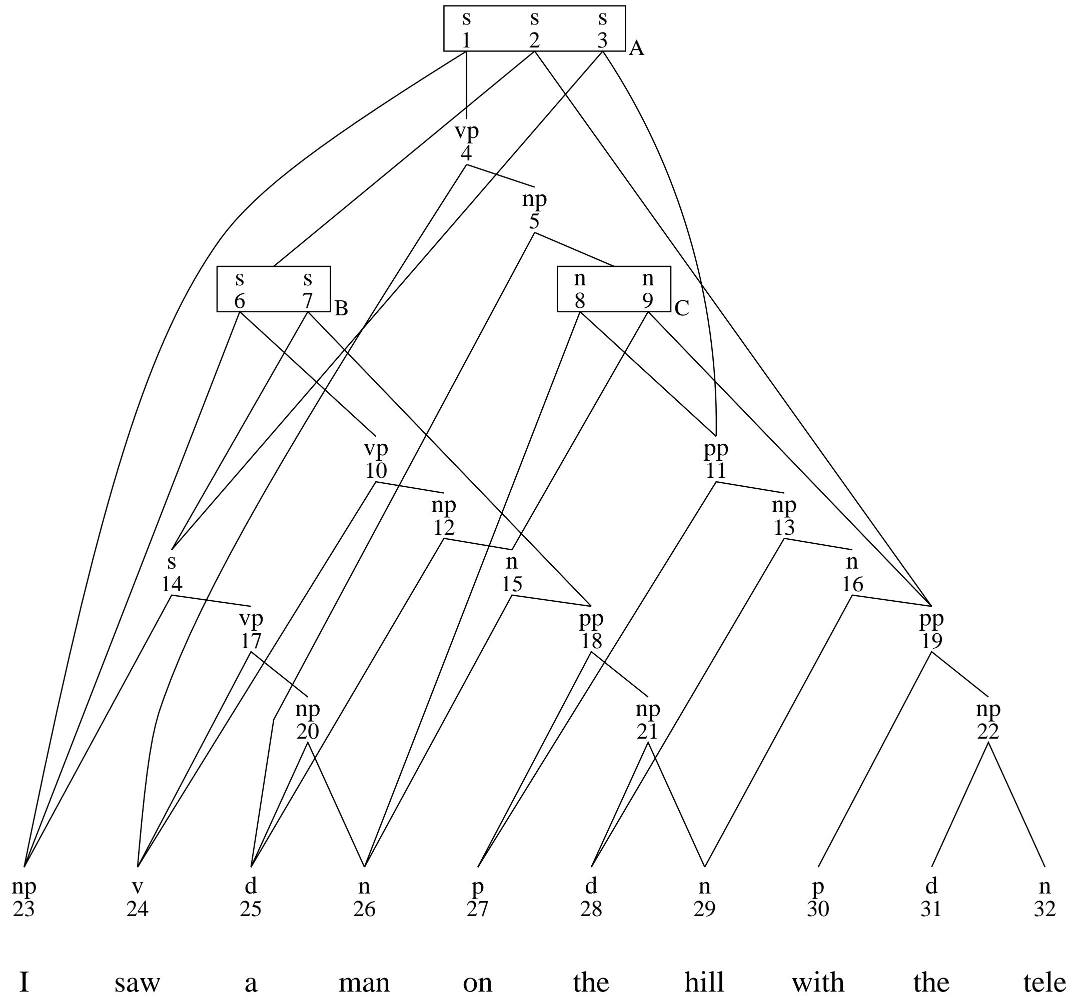
© Michael Hess 2009

Gepackte Strukturen

[Warum ‘‘packed forest’’?]

(0.L.11)



Ende des Unterdokuments

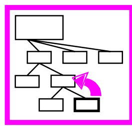
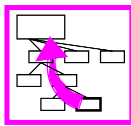
© Michael Hess 2009

Die zugrundliegende Grammatik sieht so aus:

```
rule([np,vp],s).
rule([vt,np],vp).
rule([vt,np,pp],vp).
rule([vi],vp).
rule([vi,pp],vp).
rule([p,np],pp).
rule([pn],np).
rule([n,n],np).
rule([det,n1],np).
rule([n1],np).
rule([n],n1).
rule([n,pp],n1).
```

Ende des Unterdokuments

© Michael Hess 2009

Implementation

Eine bekannte Methode, derartige Strukturen programmiertechnisch darzustellen, verwendet statt einer einzigen Syntaxstruktur nun deren zwei. Sie unterscheidet terminologisch zwischen

Konstituenten-struktur

1. *Konstituentenstruktur*, d.h. der Gesamtstruktur
2. *Analysestruktur*, d.h. der internen Struktur eines solchen Segments; oben: Inhalt der Kästchen

Dazu siehe Alshawi 1992:141-144

Analysestruktur

Folgende ‘‘Etikettierung’’ wird im folgenden verwendet:

```
 Fall leaves fall and spring leaves spring
0    1      2    3   4      5      6      7
```

Nun kann man die syntaktischen Kategorien von Ketten repräsentieren als *Konstituentenstrukturen* der Form

```
con(<von>, <Konstituententyp>, <bis>)

con(0,S,7)

con(0,S,3)
con(3,Cj,4)
con(4,S,7)
```

Beachte: Nur eine einzige Konstituentenangabe für ‘‘S(0-3)’’ und ‘‘S(4-7)’’ - die (je zweifach mehrdeutige) innere Struktur wird *hier* nicht repräsentiert.

Die inneren Strukturen von Konstituenten stellt man durch *Analysestrukturen* dar, von der Form:

```
ana(<Regelbezeichnung>,<Mutter>,[<Tochter_1>,<Tochter_2>...<Tochter_n>])
```

wo die Terme `Mutter` und `Tochter_n` von der Form `con(<von>,<Konstituententyp>,<bis>)` sind. (Die Regelbezeichnungen spielen im vorliegenden Kontext keine Rolle).

© Michael Hess 2009

Also: Immer nur die Mutter und alle ihre Töchter (also nur eine einzige Ebene tief).

Das ergibt dann insgesamt

```
ana(s_s_s,con(0,S,7),[con(0,S,3),con(3,Cj,4),con(4,S,7)])

ana(s_np_vp,con(0,S,3),[con(0,Np,2),con(2,Vp,3)])
ana(s_np_vp,con(0,S,3),[con(0,Np,1),con(1,Vp,3)])

ana(s_np_vp,con(4,S,7),[con(4,Np,6),con(6,Vp,7)])
ana(s_np_vp,con(4,S,7),[con(4,Np,5),con(5,Vp,7)])

<etc.>
```

Das ‘‘<etc.>’’ bezieht sich auf die Struktur von Np, Vp etc.

Man beachte:

- Die allen vier Syntaxstrukturen gemeinsame Grundstruktur wird nur *ein* Mal aufgeführt: erste Zeile.
- Die je zwei Strukturen der zwei Teilsätze werden nur je *ein* Mal aufgeführt.
- ‘‘Ausmultipliziert’’ wird *nicht!*

Dass die gleiche Information, nur kompakter, durch die passiven Kanten eines Chart-Parsers am Ende des Analysevorgangs repräsentiert wird, sieht man an folgendem Beispiel (beachte: anderer Beispiel-Satz):

```
Sentence:
 <0> peter <1> saw <2> the <3> woman <4> with <5> the <6> telescope <7>

The Result is:
----------------------
edge(np,[pn],[],0,1)
edge(n1,[n],[],3,4)
edge(np,[n1,det],[],2,4)
edge(vp,[np,vt],[],1,4)
edge(n1,[n],[],6,7)
edge(np,[n1,det],[],5,7)
```

© Michael Hess 2009

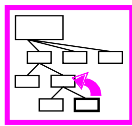
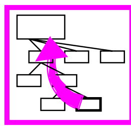
```
edge(pp,[np,p],[],4,7)
edge(n1,[pp,n],[],3,7)
edge(np,[n1,det],[],2,7)
edge(s,[vp,np],[],0,7)       % 1
edge(vp,[np,vt],[],1,7)      % 2
edge(vp,[pp,np,vt],[],1,7)   % 3
```

Die Gesamtstruktur des Satz wird nur ein *einziges* Mal genannt (%1), und die (zweifache) Ambiguität der Teilstruktur von Position 1 bis 7 wird in den Kanten %2 und %3 zum Ausdruck gebracht.

Die Grammatik (0.L.12.L.1)

Ende des Unterdokuments

© Michael Hess 2009

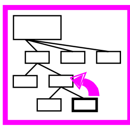
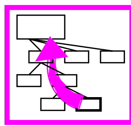
‘‘Conventional computing paradigms, such as functional, imperative, and object-oriented programming, deal primarily with full information notions of objects and data-types. Constraint programming (CP) is based on the ability to represent and manipulate partial information about objects, i.e., constraints such as equalities and inequalities. CP augments conventional notions of state, state-change, and control with notions of monotonic accumulation of partial information about objects of interest, and with operations involving constraints such as consistency and entailment.

Early studies, in the 60’s and 70’s, introduced and made use of CP in graphics and in artificial intelligence. In the 80’s, considerable progress was achieved with the emergence of constraint logic programming and of concurrent constraint programming. CP has been applied with some success to operations research scheduling problems, hardware verification, user-interface design, decision-support systems, and simulation and diagnosis in model-based reasoning. Currently, CP is contributing exciting new directions in research areas such as: artificial intelligence, computational linguistics, concurrent and distributed computing, database systems, graphical interfaces, operations research and combinatorial optimization, programming language design and implementation, symbolic computing algorithms and systems.’’

(from: Announcement of PPCP93 --- Call For Papers First Workshop on Principles and Practice of Constraint Programming April 28--30, 1993 Newport, Rhode Island, U.S.A.)

Ende des Unterdokuments

© Michael Hess 2009

Breitensuche + Evaluationsfunktion + Kostenfunktion = ‘‘A*’’-Suche

[Die Resultate]

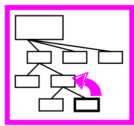
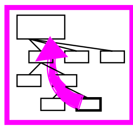
(0.L.14)


```
1302: the woman sees the man with a telescope

0 parses were pruned
Pruning with increment 0.897
1 constituent was recognized.
Parse 1 of 2:

sent(10.6464
     np(3.52
        det(def)
        cnp(2.4
            []
            noun(woman)
            []
            []
            []))
     vp(7.78801
        cvp(7.73502
            verb(see)
            np(6.66879
               det(def)
               cnp(6.33599
                   []
                   cnp(2.4
                       []
                       noun(man)
                       []
                       []
                       [])
                   pp(3.52
                      prep(with)
                      np(3.52
                         det(indef)
                         cnp(2.4
                             []
                             noun(telescope)
                             []
                             []
                             []))))))
            []
            3+sing)
        []
        []
        3+sing))
```

© Michael Hess 2009

```
Parse 2 of 2:

sent(11.287
     np(3.52
        det(def)
        cnp(2.4
            []
            noun(woman)
            []
            []
            []))
     vp(8.58878
        cvp(5.21599
            verb(see)
            np(3.52
               det(def)
               cnp(2.4
                   []
                   noun(man)
                   []
                   []
                   []))
            []
            3+sing)
        pp(3.52
           prep(with)
           np(3.52
              det(indef)
              cnp(2.4
                  []
                  noun(telescope)
                  []
                  []
                  [])))
        []
        3+sing))
```

© Michael Hess 2009

Ende des Unterdokuments

© Michael Hess 2009

‘‘Adversativ’’: in feindlicher Umgebung.

Sonderfall von Problemlösung:

- *mehrere Partien* (normalerweise zwei), abwechselnd ziehend
- Partien arbeiten für ihre jeweils *eigenen Interessen*
- Interessen *nie ganz kongruent*
- aber es liegen *keineswegs immer Nullsummen-Bedingungen* vor

Relevanz von Spielen

1. als theoretisches Instrument für die Künstliche Intelligenz:
   a. sehr klar umgrenzter Problembereich
   b. sehr klar definierte Ziele
   c. erheblicher Erfahrungsschatz menschlicher Spieler
   d. deshalb geeignet, das Verhältnis zwischen Suche und Wissen zu untersuchen

2. für konkrete Anwendungen (Spieltheorie)
   a. militärische (strategische und taktische) Anwendungen
   b. Wirtschaftswissenschaft
   c. Politikwissenschaft (Staaten und andere politische Entitäten als Agenten)
   d. oft wird das Aggregat aller Kontrahenten sinnvollerweise als Gegner behandelt, v.a. wenn es wegen Interaktionen u. dgl. nur schwer in einzelne Faktoren aufzuspalten ist ; pessimistische Annahme
   e. spieltheoretische Begründung der Logik
   f. *Computerlinguistik:* Dialogführung.

Wie kann man das mit den bisher besprochenen Problemlösungsmethoden vergleichen? Indem man es ebenfalls als Zustandsraumsuche darstellt, aber

- der Zustandsraum ist‘‘geschichtet’’, wo jeder Spieler nur jedes zweite Mal die Freiheit hat, einen Operator zu wählen

© Michael Hess 2009

- dann kann man prinzipiell alle oben eingeführten Optimierungs-Methoden anwenden (Kosten- und Evaluationsfunktion etc.); Evaluationsfunktionen sind z.B. (je nach Spiel)
  - i. Verhältnis der Anzahl von übrigbleibenden Figuren beider Partien
  - ii. Freiheitsgrade der eigenen Figuren
  - iii. Kontrolle zentraler Regionen (Mitte des Felds)
- bloss eben abwechslungsweise aus verschiedener Sicht, was kombinierte Strategien ergibt (z.B. Minimax)

Vereinfachende Annahmen für die folgenden Überlegungen:

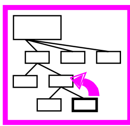
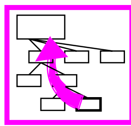
1. zwei Spieler
2. Nullsummenspiel
3. vollständige Information (gegnerischer Zug ist bekannt, wenn man zieht; vergleiche dagegen ‘‘Stein-Schere-Papier’’-Spiel)

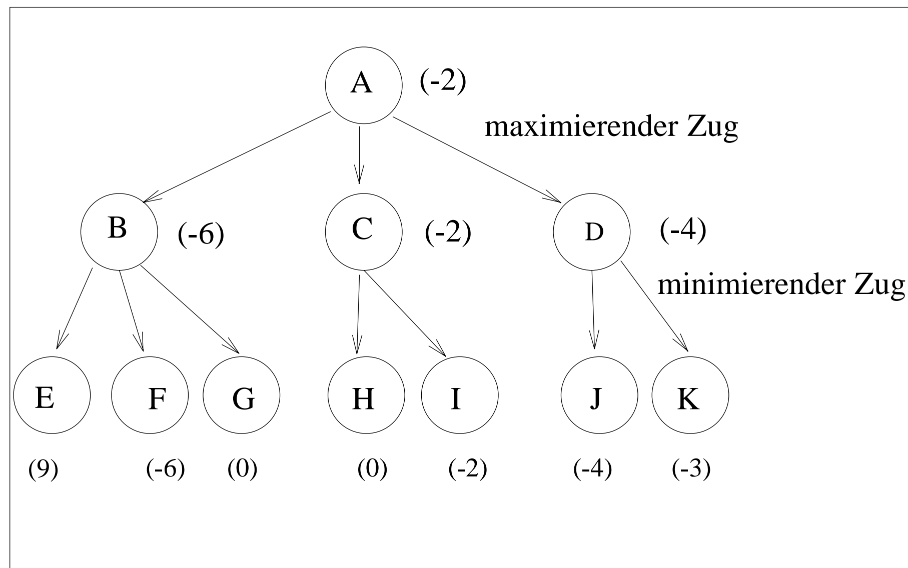
Wegen des gegnerischen Verhaltens ist eine reine Breitensuche (z.B. A*) ungeeignet (Kosten- und Evaluationsfunktion können nicht als bekannt vorausgesetzt werden).

Daher:

- Tiefensuche, um die möglichen Reaktionen des Gegners einzubeziehen
- aber als ‘‘iterative deepening’’ implementiert:
  - i. so tief, wie es die Zeit eben zulässt
  - ii. dann den besten Zug durchführen (Minimum: zwei Ebenen).

© Michael Hess 2009



Illustrationen: Rich 1991:311

0.0.1 Minimax-Strategie

In Fig. 12.2 wäre die erste, naive, Annahme, dass die Werte sind wonach B der beste Zug wäre (A → B → E). Ist aber falsch, weil der Gegner bei *seinem* ersten Zug F wählen würde (Gewinn 6 für ihn), was für uns viel schlechter ist (*Verlust* 6 für uns), als A → C → H (Gewinn 0 für uns).

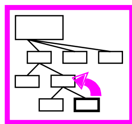
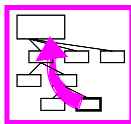
Daher: Prinzip

1. in den ‘‘gegnerischen’’ Schichten des Suchbaums immer das *Minimum* aller möglichen Werte wählen
2. die entsprechenden Werte nach oben propagieren
3. in ‘‘unseren’’ Schichten davon das *Maximum* wählen: *‘‘Minimax’’*

Also: Fig. 12.3.

Ein reales Beispiel:[2] Japaner vs. Amerikaner

© Michael Hess 2009

1. Japaner wollen nach Neu-Guinea vorrücken. Zur Auswahl stehen die Route nördlich von Neubritannien, und die südlich davon.
2. Die Allierten wollen das verhindern, indem sie die Japaner bombardieren.
3. An so vielen Tagen können die Allierten die Japaner bombardieren:

```
                           Wahl der Japaner
                        nördlich       südlich
                        ------------------------
Wahl der  nördlich       2 Tage         2 Tage
Allierten südlich        1 Tag          3 Tage
```

4. Sollen die Allierten optimistischerweise hoffen, die Japaner wählten die südliche Route, oder sich pessimistischerweise auf die Nordvariante einstellen?
5. Antwort: Beide Gegner wählen Strategie ‘‘nördlich’’. So war es auch in der Realität.
6. Grund: Minimax

```
Wahl der       Wahl der Japaner:   Zeilen-
Allierten: nördlich     südlich    Minimum:
        ------------------------------------
nördlich    2 Tage         2 Tage     2
südlich     1 Tag          3 Tage     1
                                    --------
Kolonnen-Maximum                       2
```

2. p 30 in; M.D. Davis, Spieltheorie für Nichtmathematiker, Oldenbourg, München/Wien, Scientia Nova, 1972

© Michael Hess 2009

0.0.2 Verfeinerungen der Minimax-Strategie

1. Warten auf Stabilisierung: (cf. Rich 1991:319; Fig. 12.7, 12.8, 12.9)
2. sekundäre Suche
3. ‘‘Buchzüge’’ (‘‘case based reasoning’’)

0.0.3 Beschränkungen

Die bisher besprochenen Strategien setzen eine extrem einfache (und seltene) Situation voraus:

1. ein vollständig rationaler Gegner
2. nur eine einzige Spiel-Ebene (elementare Züge).

MEHR: Shapiro 1987, Charniak 1985:281.

Zu computerlinguistischen Anwendungen:

‘‘Traditionally, Dialogue Games have been used to assist with the ’deciding what utterance to maké problem

A different approach has been taken by philosophers studying Dialogue Games as a means to study argumentation. ’’

http://cbl.leeds.ac.uk/rachel/eaied.html resp.
/home/ludwig6/hess/classes/mki/eaied.html

http://www.cbl.leeds.ac.uk/˜euroaied/papers/Burton/ resp.
/home/ludwig6/hess/classes/mki/clarissa.html

Ende des Unterdokuments

© Michael Hess 2009

Um diese spezifischen Klauseln automatisch aus ‘‘normalen’’ Klauseln zu erzeugen, verwendet man am besten das Systemprädikat `expand`:

```
expand(Clause, cl(Head,Body)) :-
        (   Clause = (Head:-Body)
        ->  true
        ;   Head = Clause, Body = true
        ).
```

NB: ‘‘term_expansion’’, z.B. so

```
term_expansion(Head,cl(Head,true)).
term_expansion((Head:-Body),cl(Head,Body)).
```

geht hier nicht, da nachher versucht würde, *alles* zu expandieren (inkl. end_of_file!):

```
| ?- listing.
{consulting /tmp_mnt/home/ludwig6/hess/texts/classes/zurich/mki/t.pl...}
{EXISTENCE ERROR: get(_327): attempt to read past end of stream}
{EXISTENCE ERROR: get(_667): attempt to read past end of stream}
<etc.>
```

Ende des Unterdokuments

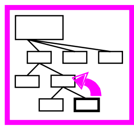
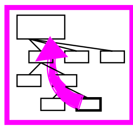
© Michael Hess 2009

Da der Interpreter zirkulär ist, kann er sich selbst interpretieren! Führt das nicht zum Disaster? Nicht immer:

```
trace, prove(prove(grandfather(_135,_136))).

{The debugger will first creep -- showing everything (trace)}
   1  1  Call: prove(prove(grandfather(_87,_109))) ?
   2  2  Call: prove_one(prove(grandfather(_87,_109))) ?
   3  3  Call: clause(prove(grandfather(_87,_109)),_374) ?
{ERROR: clause(prove(grandfather(_87,_109)),_374) - not a dynamic predicate}
   3  3  Fail: clause(prove(grandfather(_87,_109)),_374)
   3  3  Call: call(prove(grandfather(_87,_109))) ?
   4  4  Call: prove(grandfather(_87,_109)) ?
   5  5  Call: prove_one(grandfather(_87,_109)) ?
   6  6  Call: clause(grandfather(_87,_109),_660) ?
   6  6  Exit: clause(grandfather(_87,_109),(father(_87,_766),parent(_766,_10
   7  6  Call: prove((father(_87,_766),parent(_766,_109))) ?
   8  7  Call: prove_one(father(_87,_766)) ? l

_135 = peter,
_136 = jim ?

yes
```

Ende des Unterdokuments

© Michael Hess 2009

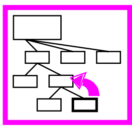
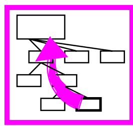
Eine realistischere Implementation würde Meta-Level-Prädikate verwenden:

```
?- setof(O, C1^C2^(ocean(O),
            borders(O,C1), african(C1), country(C1),
            borders(O,C2), asian(C2), country(C2)),
         Os).
```

Genauso geschieht es in Chat-80, von wo dieses Beispiel stammt.

Ende des Unterdokuments

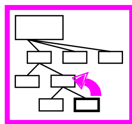
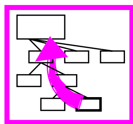
© Michael Hess 2009

Der vollständige Code ist
⇒hier
und hier:

```
prove([]).
prove([true])   :-  !.
prove([G|Gs])   :-  prove_one(G), prove(Gs).

prove_one(true) :-  !.

prove_one(G)    :-  bagof((G,List),
                           SGs^(clause(G,SGs),
                                conjtolist(SGs,List)),
                           Clauses),
                    rearrange(Clauses,Sorted), % 1
                    member((G,Cl),Sorted),     % 2
                    prove(Cl).
prove_one(G)    :-  G.

%%%%%%%%%%%%%%%%%%% Umordnung der RHS-Terme %%%%%%%%%%%%%%%%%%%%%%%%%
%               (parametrisierbarer 'insert-sort')


rearrange([X|Xs],Ys)            :-      rearrange(Xs,Zs),
                                        insert(X,Zs,Ys).
rearrange([],[]).

insert(X,[],[X]).
insert(X,[Y|Ys],[Y|Zs])         :-      prefer(Y,X),
                                        insert(X,Ys,Zs).
insert(X,[Y|Ys],[X,Y|Ys])       :-      prefer(X,Y).

%%%%%%%%%%%%%%%%%%%%%%% Hilfspraedikat %%%%%%%%%%%%%%%%%%%%%%

conjtolist((Term,Terms), [Term|List_of_terms]) :- !,
```

© Michael Hess 2009

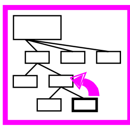
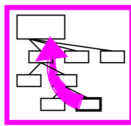
Verwendungsreihenfolge von Regeln: Leichteste Regel zuerst verwenden
[Das komplette Beispiel]

(0.L.19)


```
        conjtolist(Terms,List_of_terms).
conjtolist(Term,[Term]).


%%%%%%%%%%%%%%%%%%%%%%% Meta-Regeln %%%%%%%%%%%%%%%%%%%%%%%%%%

prefer((_,T1),(_,T2))          :-      length(T1,L1),
                                       length(T2,L2),
                                       L1 =< L2.
prefer((_,T2),(_,T1))          :-      length(T1,L1),
                                       length(T2,L2),
                                       L1 > L2.

%%%%%%%%%%%%%%%%%%%%%%%% Datenbasis %%%%%%%%%%%%%%%%%%%%%%%%%%

grandfather(X,Z)        :-      father(X,Y),
                                parent(Y,Z).
greatgrandfather(W,Z)   :-      father(W,X),
                                parent(X,Y),
                                parent(Y,Z).
greatgrandfather(X,Z)   :-      grandfather(X,Y),
                                parent(Y,Z).

parent(X,Y)             :-      father(X,Y).
parent(X,Y)             :-      mother(X,Y).

father(peter,john).
father(john,jim).
father(jim,bob).
father(bob,mary).
mother(jill,mary).
```

Ende des Unterdokuments

© Michael Hess 2009

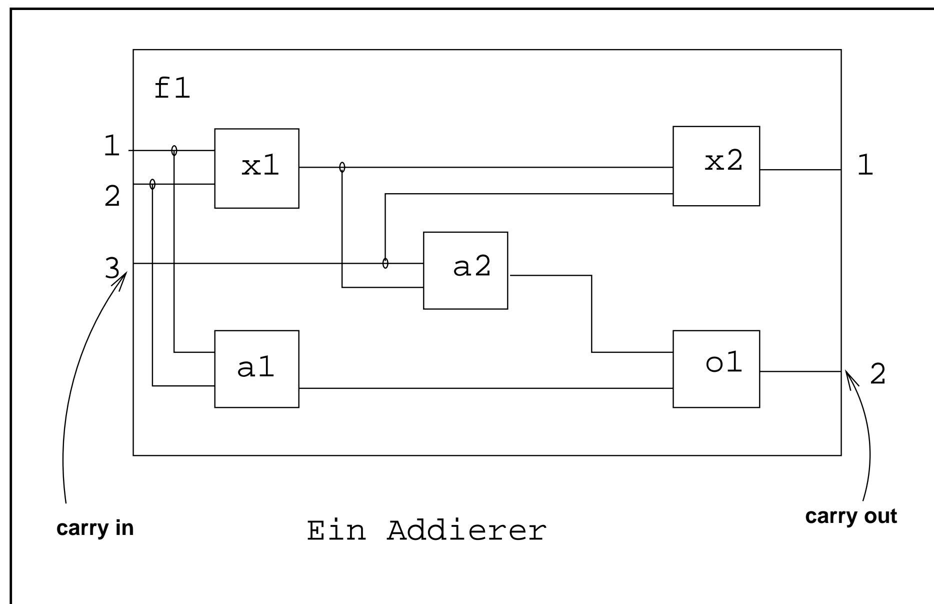
 

0.0.0.1 Einfache Simulation mit logik-basierten Systemen

Beispiel zur einfachen *Simulation:* Ein Addierer (‘‘Full adder’’) für zwei einstellige Binärzahlen:

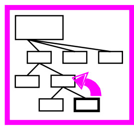
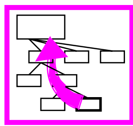
```
0 + 0 = 0 0
0 + 1 = 1 0
1 + 0 = 1 0
1 + 1 = 0 1
```

Illustration: Genesereth 1987:29 ‘A full adder’



Der Text des folgenden Programms ist ⇒hier.

© Michael Hess 2009

```
/* A Full Adder; after Clocksin */

/* Struktur (und Spannungsübertragung) */

adder(I1,I2,I3,O1,O2)    :-
       xorg(I1,I2,T1), xorg(T1,I3,O1), andg(I3,T1,T2),
       andg(I1,I2,T3), org(T2,T3,O2).

/*  Funktionsweise der Bauteile */
/*  And-Gate  */

andg(1,1,1).
andg(0,X,0).
andg(X,0,0).

/*  Or-Gate  */

org(1,X,1).
org(X,1,1).
org(0,0,0).

/*  Xor-gate  */

xorg(X,X,0).
xorg(X,Y,1)     :-      not(X=Y).
```

‘‘xor’’: *exklusives* ‘‘oder’’ (‘‘entweder - oder, aber nicht beide’’).

Mit einem Aufruf

```
?-      adder(V1,V2,V3,O1,O2).
```

(mit Werten ‘1’ oder ‘0’ für die Variablen V1 bis V3) kann man 2 Ein-Bit-Binärzahlen addieren.

V1 und V2: Eingangswerte; V3: eingehender Übertrag; O1: Ausgangswert; O2: ausgehender Übertrag

© Michael Hess 2009

Beispiele für Simulation: (Resultat errechnen; hier: Korrektheit des Schaltkreises testen)

```
Computing the output values from the input values.
The input values are:
Line 1 = 0
Line 2 = 0
Line 3 = 0
The output values are:
Line 1 = 0
Line 2 = 0

Computing the output values from the input values.
The input values are:
Line 1 = 1
Line 2 = 0
Line 3 = 0
The output values are:
Line 1 = 1
Line 2 = 0

Computing the output values from the input values.
The input values are:
Line 1 = 0
Line 2 = 1
Line 3 = 1
The output values are:
Line 1 = 0
Line 2 = 1
```

© Michael Hess 2009

0.0.0.2 Fehlerdiagnose mit logikbasierten Systemen

Beispiel zur *Fehlerdiagnose:* Hardware-Sortierer für (z.B.) Dezimalzahlen

Diese Anwendung ist einfacher als der Addierer oben; vor allem ist die Korrektheit der Resultate leichter zu sehen.

Der Text des folgenden Programms ist ⇒hier.

```
/* Hardware-Sortierer */
/* Struktur */

sorter(I1,I2,I3,I4,O1,O2,O3,O4) :-
        two_sorter(1,I1,I2,T1,T2),
        two_sorter(4,I3,I4,T3,T4),
        two_sorter(2,T1,T3,O1,T5),
        two_sorter(5,T2,T4,T6,O4),
        two_sorter(3,T5,T6,O2,O3).

/*  Übergangsverhalten des intakten Zweier-Sortierers  */

two_sorter(N,A,B,A,B)   :-      working(N), A=<B.
two_sorter(N,A,B,B,A)   :-      working(N), B < A.

/* Übergangsverhalten des defekten Zweier-Sortierers */

two_sorter(N,A,B,A,B)   :-      defect(inversion),
                                broken(N), B =< A.
two_sorter(N,A,B,B,A)   :-      defect(inversion),
                                broken(N), A < B.

two_sorter(N,A,B,0,0)   :-      defect(rupture), broken(N).

two_sorter(N,A,B,A,B)   :-      defect(shorted), broken(N).

two_sorter(N,A,B,X,Y)   :-      defect(random), broken(N),
                                random(9,X), random(9,Y).
```

© Michael Hess 2009

```
                                 %% natuerlich ganz hoffnungslos
```

Einschränkungen: Im ganzen Schaltkreis kann gleichzeitig nur ein einziger Typ von Fehler vorkommen. ‘‘defect(N,Type)’’ wäre besser, aber aufwendiger zu implementieren.

Testlauf (intakter Sortierer):

```
| ?- test(1).
Input values:  4 1 3 5 Output values: 1 3 4 5
```

kaputter Sortierer: (test**1** heisst: Schaden setzen, Normalschaden: ‘‘rupture’’, d.h. Output auf beiden Leitungen 0)

```
| ?- test1(1).

Which element should be broken?
|: 3.
3 is broken
Input values:  4 1 3 5.   Output values: 1 0 0 5
```

Lokalisieren eines Fehlers (test**2** heisst: Schaden lokalisieren):

```
| ?- test2(1).

Which element should be broken?
|: 3.
Element 3 already broken
The broken element(s) now: 3
Type of defect: rupture
Input values:  4 1 3 5.   Output values now: 1 0 0 5
```

Die Angaben ↑ dienen nur zur Information des Benutzers; das System weiss das nicht.

Von jetzt an Diagnose (System geht von intaktem System aus und zerstört ein Element nach dem andern):

```
Assuming now:
```

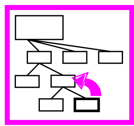
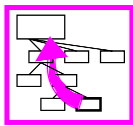
© Michael Hess 2009

```
1: working
2: working
3: working
4: working
5: broken


Assuming now:
1: working
2: working
3: working
4: broken
5: working

<...>

Assuming now:
1: working
2: working
3: broken
4: working
5: working


Broken element(s) in reality:  3
Diagnosis:  3
```

Andere Typen von Schaden können gesetzt werden, z.B. ‘‘inversion’’ (Werte kommen verkehrt raus):

```
| ?- inversion.

yes
| ?- test1(1).
Which element should be broken?
|: 3.
3 is broken
Input values:  4 1 3 5.   Output values: 1 4 3 5
```

© Michael Hess 2009

```
| ?- test2(1).
Which element should be broken?
|: 3.
Element 3 already broken
The broken element(s) now: 3
Type of defect: inversion
Input values:  4 1 3 5.   Output values now: 1 4 3 5

Assuming now:
1: working
2: working
3: working
4: working
5: broken


Assuming now:
1: working
2: working
3: working
4: broken
5: working

<...>

Assuming now:
1: working
2: working
3: broken
4: working
5: working


Broken element(s) in reality:  3
Diagnosis:  3
```

Zu beachten:

© Michael Hess 2009

- Das ist ein *sehr* einfaches Programm! Am aufwendigsten ist der (oben nicht gezeigte) Code fürs gezielte Kaputtmachen (und ‘‘Reparieren’’) einzelner Elemente (nämlich das Modifizieren des Programms)
- Die Annahme, dass man den *Typ* des Fehlers kennt, ist unrealistisch. Aber natürlich könnte man das Programm so erweitern, dass es einen Typ von Fehler nach dem andern durchprobiert. Programmiertechnisch nicht sehr aufwendig, aber die Effizienz wird weiter sinken.
- Jedes Element sollte eine andere Sorte von Schaden haben dürfen

Bei Berücksichtigung all dieser Dinge wird ‘‘analysis by synthesis’’ extrem ineffizient.

Daher: Weiterführende Fragestellungen:

1. Wie könnte man aus dem Output *direkt* auf den Input schliessen? (eben *Abduktion*; z.B. aus Hypothese über defektive Struktur und beobachtetem Output auf den Input schliessen, dann mir realem Input vergleichen)
2. Wie könnte man aus den Beobachtungen des I/O-Verhaltens *Regeln* für ein seichtes Expertensystem erschliessen? (META-DENDRAL versuchte das. Sehr schwierig: ‘‘*Induktion*’’ (s.u.))

Dazu folgende Beispiele: Aufgrund

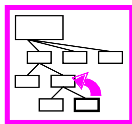
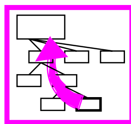
```
The broken element(s) now: 1
Type of defect: rupture
Input values:  4 1 3 5.   Output values now: 0 0 3 5
Input values:  9 0 3 5.   Output values now: 0 0 3 5
Input values:  1 2 3 4.   Output values now: 0 0 3 4
Input values:  8 3 5 7.   Output values now: 0 0 5 7
Input values:  3 7 1 0.   Output values now: 0 0 0 1
Input values:  1 8 4 6.   Output values now: 0 0 4 6
```

könnte man die *erste Vermutung* haben : Wann immer

1. erste zwei Output-Werte 0
2. letzte zwei Output-Werte unverändert

dann ist Element 1 kaputt.

© Michael Hess 2009

Fast richtig, **aber:** Zeile 5 widerlegt die Vermutung.

Zweite Vermutung : Wann immer

1. erste zwei Output-Werte 0 (wie zuvor)
2. letzte zwei Output-Werte die *sortierten* Input-Werte sind

dann ist Element **1** kaputt.

Also könnte man schreiben:

```
defekt(1)       :-      output(1,0),
                        output(2,0),
                        output(3,M),
                        output(4,N),
                        input(3,A),
                        input(4,B),
                        sort_all([A,B],[M,N]). % keeps duplicates
```

(die Prädikate `output` und `input` müssten noch entsprechend definiert werden).

Für Element 4: dasselbe (ist symmetrisch zu Element 1 positioniert)

```
The broken element(s) now: 4
Type of defect: rupture
Input values:  4 1 3 5.   Output values now: 0 0 1 4
Input values:  9 0 3 5.   Output values now: 0 0 0 9
Input values:  1 2 3 4.   Output values now: 0 0 1 2
Input values:  8 3 5 7.   Output values now: 0 0 3 8
Input values:  3 7 1 0.   Output values now: 0 0 3 7
Input values:  1 8 4 6.   Output values now: 0 0 1 8
```

Das ist leider nicht so einfach für die anderen Elemente:

```
The broken element(s) now: 2
Type of defect: rupture
Input values:  4 1 3 5.   Output values now: 0 0 4 5
Input values:  9 0 3 5.   Output values now: 0 0 5 9
```

© Michael Hess 2009

```
Input values:  1 2 3 4.   Output values now: 0 0 2 4
Input values:  8 3 5 7.   Output values now: 0 0 7 8
Input values:  3 7 1 0.   Output values now: 0 0 1 7
Input values:  1 8 4 6.   Output values now: 0 0 6 8
```

Eventuell: zwei Mal 0 in Output-Werten 1 und 2, Output-Werte 3 und 4 geordnet aber gerade *nicht* aus den Input-Werten 1 und 2 bestehend:

```
defekt(4)       :-      output(1,0),
                        output(2,0),
                        output(3,M),
                        output(4,N),
                        M < N,
                        \+ ((  input(1,A),
                                input(2,B),
                                sort_all([A,B],[M,N]) )).
```

Sehr fraglich.

Noch undurchsichtiger im folgenden:

```
The broken element(s) now: 3
Type of defect: rupture
Input values:  4 1 3 5.   Output values now: 1 0 0 5
Input values:  9 0 3 5.   Output values now: 0 0 0 9

Wrong Diagnosis:               4
Broken element(s) in reality:  3

Input values:  1 2 3 4.   Output values now: 1 0 0 4
Input values:  8 3 5 7.   Output values now: 3 0 0 8
Input values:  3 7 1 0.   Output values now: 0 0 0 7
Input values:  1 8 4 6.   Output values now: 1 0 0 8


The broken element(s) now: 5
Type of defect: rupture
Input values:  4 1 3 5.   Output values now: 1 0 3 0
Input values:  9 0 3 5.   Output values now: 0 0 3 0
```

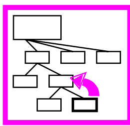
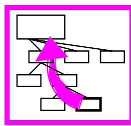
© Michael Hess 2009

```
Input values:  1 2 3 4.   Output values now: 1 0 3 0
Input values:  8 3 5 7.   Output values now: 3 0 5 0
Input values:  3 7 1 0.   Output values now: 0 0 3 0
Input values:  1 8 4 6.   Output values now: 1 0 4 0
```

Mit anderen Worten: Es ist sehr schwierig, ‘‘von Hand’’ Regeln aus den Daten zu erschliessen. Das sollte man von einem System errechnen lassen können, statt ± blind zu raten. Dazu braucht man Verfahren der *Induktion*. Siehe dazu unten.

Ende des Unterdokuments

© Michael Hess 2009

Verwendungsreihenfolge von Regeln: Leichteste Regel zuerst verwenden

[Andere mögliche Meta-Regeln]

(0.L.20)

Andere mögliche, ± intelligente, Meta-Regeln sind (z.T. Mischungen aus Term- und Regelbevorzugung)

1. Jene Regeln werden bevorzugt, deren RHS sich *direkt* durch Fakten beweisen lassen (ohne Regelanwendung): Der Interpreter muss drei Schritte ‘‘voraus-denken’’

```
a(X)    :-  b(X), c(X).
a(X)    :-  f(X), g(X).

b(1).
c(1).
f(X)    :-  h(X), i(X,Y), k(Y,X).
g(X)    :-  l(X), m(Y), X is Y+2.
```

   Frage: wann wird diese Optimierung via Meta-Interpreter teurer als die direkte Interpretation?

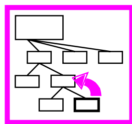
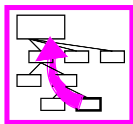
2. Gewisse Terme werden vom Programmierer explizit als erst später zu beweisen ausgezeichnet: Direkte Beeinflussung des Kontrollflusses im Programmtext

3. Gewisse Terme werden in Tiefensuche bewiesen , andere in Breitensuche

---

Ende des Unterdokuments

© Michael Hess 2009

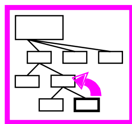
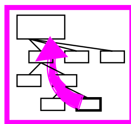
Deutlicher wird der Nutzen des Entfaltens bei einem komplizierteren Interpreter.

Der *Left-Corner-Parser* von oben (und dieselbe Grammatik):

```
%%%%%%%% Objekt-Programm: Left-Corner Iinterpreter  %%%%%%%

parse({Term})                --> {Term}.
%parse(Phrase)               --> leaf(SubPhrase),
%                               {link(SubPhrase,Phrase)},
%                                lc(SubPhrase,Phrase).

leaf(Cat)                    --> [Word], {Cat ---> [Word]}.
leaf(Phrase)                 --> {Phrase ---> []}.

lc(Phrase,Phrase)            --> [].
lc(SubPhrase,SuperPhrase)  --> {(Phrase ---> [SubPhrase|Rest]),
                                link(Phrase, SuperPhrase)},
                                parse_rest(Rest),
                                lc(Phrase,SuperPhrase).

parse_rest([])               --> [].
parse_rest([Phrase|Phrases])--> parse(Phrase),
                                parse_rest(Phrases).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

link(np(_,_), sent(_)).
link(det(_,_), np(_,_)).
link(det(_,_), sent(_)).
link(v(_,_,_), vp(_,_)).
link(adj(_),adjp(_)).
link(X,X).
```

mit der Programmklausel

© Michael Hess 2009

```
program_clause(( parse(Phrase,P0,P) :-
                        leaf(SubPhrase,P0,P1),
                        link(SubPhrase,Phrase),
                        lc(SubPhrase,Phrase,P1,P) )).
```

und mit den Hilfsprädikaten

```
aux_literal(leaf(_,_,_)).
aux_literal(parse_rest(_,_,_)).

aux_literal((_ ---> _)).
```

ergeben vorerst einmal (nur Variablen etc. editiert)

```
parse(SP,P0,P)  :-  'C'(P0,the,P1),
                    link(det(det(the),N),SP),
                    lc(det(det(the),N),SP,P1,P).
parse(SP,P0,P)  :-  'C'(P0,a,P1),
                    link(det(det(a),sing),SP),
                    lc(det(det(a),sing),SP,P1,P).
parse(SP,P0,P)  :-  'C'(P0,dog,P1),
                    link(n(n(dog),sing),SP),
                    lc(n(n(dog),sing),SP,P1,P).
<etc.>
```

Nun noch *zusätzlich* definiert

```
aux_literal(link(_,_)).
```

und es wird interessanter:

© Michael Hess 2009

```
parse(sent(A),P0,P) :-  'C'(P0,the,P1),
                        lc(det(det(the),N),sent(A),P1,P).
parse(sent(A),P0,P) :-  'C'(P0,a,P1),
                        lc(det(det(a),sing),sent(A),P1,P).

parse(np(A,B),P0,P) :-  'C'(P0,a,P1),
                        lc(det(det(a),sing),np(A,B),P1,P).
parse(np(A,B),P0,P) :-  'C'(P0,the,P1),
                        lc(det(det(the),N),np(A,B),P1,P).

parse(vp(B,N),P0,P) :-  'C'(P0,eats,P1),
                        lc(v(v(eat),tr,sing),vp(B,N),P1,P).
parse(vp(B,N),P0,P) :-  'C'(P0,eat,P1),
                        lc(v(v(eat),tr,plur),vp(B,N),P1,P).

parse(v(v(eat),tr,sing),P0,P)
                    :-  'C'(P0,eats,P1),
                        lc(v(v(eat),tr,sing),v(v(eat),tr,sing),P1,P).
<etc.>
```

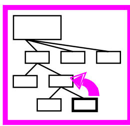
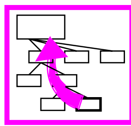
Also: Obwohl man den Beweis von `parse(sent(A),P0,P)` führt, wird zuerst *auf der Wortebene* getestet, ob der Input einen Satz beginnen kann `'C'(P0,the,P1)` etc.

Wenn man nun auch noch

```
lc(SubPhrase,SuperPhrase)   --> {(Phrase ---> [SubPhrase|Rest]),
                                 link(Phrase, SuperPhrase)},
                                 parse_rest(Rest),
                                 lc(Phrase,SuperPhrase).
```

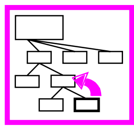
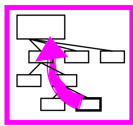
zur Programmklausel macht, ergibt sich (zusätzlich zum Obigen, aber an Stelle der neuen Programmklausel)

© Michael Hess 2009

```
lc(the,np(Np,_360),P0,P)     :-  X=P0,
                                 lc(det(det(the),_254),np(Np,_360),X,P).
lc(the,sent(Np),P0,P)        :-  X=P0,
                                 lc(det(det(the),_254),sent(Np),X,P).
lc(the,det(det(the),_254),P0,P)
                             :-  X=P0,
                                 lc(det(det(the),_254),det(det(the),_254),X,P)


lc(a,np(Np,_360),P0,P)       :-  X=P0,
                                 lc(det(det(a),sing),np(Np,_360),X,P).
lc(a,sent(Np),P0,P)          :-  X=P0,
                                 lc(det(det(a),sing),sent(Np),X,P).
lc(a,det(det(a),sing),P0,P)  :-  X=P0,
                                 lc(det(det(a),sing),det(det(a),sing),X,P).


lc(dog,n(n(dog),sing),P0,P)  :-  X=P0,
                                 lc(n(n(dog),sing),n(n(dog),sing),X,P).
lc(men,n(n(man),plur),P0,P)  :-  X=P0,
                                 lc(n(n(man),plur),n(n(man),plur),X,P).


lc(eats,vp(_360,_361),P0,P)  :-  X=P0,
                                 lc(v(v(eat),tr,sing),vp(_360,_361),X,P).
lc(eats,v(v(eat),tr,sing),P0,P) :-  X=P0,
                                 lc(v(v(eat),tr,sing),v(v(eat),tr,sing),X,P).
lc(eat,vp(_360,_361),P0,P)   :-  X=P0,
                                 lc(v(v(eat),tr,plur),vp(_360,_361),X,P).
lc(eat,v(v(eat),tr,plur),P0,P)  :-  X=P0,
                                 lc(v(v(eat),tr,plur),v(v(eat),tr,plur),X,P).
```

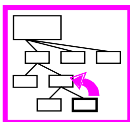
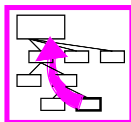
© Michael Hess 2009

```
lc(brown,adjp(_356),P0,P)    :-  X=P0,
                                 lc(adj(brown),adjp(_356),X,P).
lc(brown,adj(brown),P0,P)    :-  X=P0,
                                 lc(adj(brown),adj(brown),X,P).
```

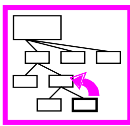
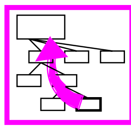
Ende des Unterdokuments

© Michael Hess 2009

```
% specific rules for verb present singulars

% y->ies except after a vowel
verb_singular(pres,Sing,Plur) :- suffix(Sing,[C,y],not(vowel(C)),[C,i,e,s],Pl

% s,x,z -> es
verb_singular(pres,Sing,Plur) :- suffix(Sing,[C],member(C,[o,s,x,z]),[C,e,s],

% ch,sh -> es
verb_singular(pres,Sing,Plur) :- suffix(Sing,[C,h],member(C,[c,s]),[C,h,e,s],

% default case
verb_singular(pres,Form,Root) :- suffix(Form,[],true,[s],Root).

verb_past(Form,Root) :- suffix(Form,[C],not(vowel(C)),[C,e,d],Root).
verb_past(Form,Root) :- suffix(Form,[],true,[d],Root).

verb_past_part(Form,Root) :- suffix(Form,[C],not(vowel(C)),[C,e,d],Root).
verb_past_part(Form,Root) :- suffix(Form,[],true,[d],Root).

% verb_gerund(Form,Root) :- suffix(Form,[C],vowel(C),[i,n,g],Root).
verb_gerund(Form,Root) :- suffix(Form,[V,C],short_vowel(V,C),[V,C,C,i,n,g],Ro
verb_gerund(Form,Root) :- suffix(Form,[e],true,[i,n,g],Root).
verb_gerund(Form,Root) :- suffix(Form,[],true,[i,n,g],Root).

vowel(V)                  :- member(V,[a,e,i,o,u]).
short_vowel(V,C)          :- vowel(V), shortening_consonant(C).
shortening_consonant(C) :- member(C,[p,t,b,d,g,m,n]).



%%%%%%%%%%%%%%%% NOUNS %%%%%%%%%%%%%%%%%%%%%

word_form(noun(Freq,noun(Root),sing),Root,Freq)         :-      noun_subcat(R
word_form(noun(Freq,noun(Root),plur),Form,Freq)         :-      noun_plural(R
                                                        noun_subcat(Root,A,B,

noun_plural(man,men) :-!.
noun_plural(woman,women) :-!.
noun_plural(child,children) :-!.
```

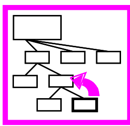
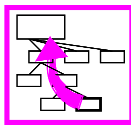
© Michael Hess 2009

```
noun_plural(mouse,mice) :-!.
noun_plural(sheep,sheep) :-!.
noun_plural(fish,fish).       % no cut because ‘fishes’ also ok
% specific rules for noun plurals

% y->ies except after a vowel
noun_plural(Sing,Plur) :- suffix(Sing,[C,y],not(vowel(C)),[C,i,e,s],Plur).

% s,x,z -> es
noun_plural(Sing,Plur) :- suffix(Sing,[C],member(C,[o,s,x,z]),[C,e,s],Plur).

% ch,sh -> es
noun_plural(Sing,Plur) :- suffix(Sing,[C,h],member(C,[c,s]),[C,h,e,s],Plur).

% default case
noun_plural(Sing,Plur) :- suffix(Sing,[],true,[s],Plur).

suffix(Sing,Astring,Condition,Bstring,Plur) :-  % version for analysis
                     not(var(Plur)),
                     convert(Plur,Plurlist),
                     append(Base,Bstring,Plurlist),
                     call(Condition),
                     append(Base,Astring,Singlist),
                     convert(Sing,Singlist),
                     asserta((suffix(Sing,Astring,Condition,Bstring,Plur):- !

convert(X,X1) :-        %explode atom -> string
         var(X1),!,name(X,Codes),name1(Codes,X1).

convert(X,X1) :-        %implode string -> atom
         var(X),!,name1(Codes,X1),name(X,Codes).

name1([],[]).
name1([X|Xs],[X1|X1s]) :- name(X1,[X]), name1(Xs,X1s).
```

Ende des Unterdokuments

© Michael Hess 2009

```
From sics.se!sicstus-users-request Sat Feb 12 20:32:30 1994
Return-Path: <sicstus-users-request@sics.se>
Received: by mailhost.uni-koblenz.de (Smail3.1.28.1)
                 id m0pVQ4s-0001iSC; Sat, 12 Feb 94 20:32 MET
Received: from sics.se by mailhost.uni-koblenz.de with smtp
                 (Smail3.1.28.1) id m0pVQ4o-0001kQC; Sat, 12 Feb 94 20:32 MET
Received: by sics.se (5.65+bind 1.7+ida 1.4.2/SICS-1.4)
                 id AA02688; Sat, 12 Feb 94 20:23:49 +0100
Received: from dist.dist.unige.it by sics.se (5.65+bind 1.7+ida 1.4.2/SICS-1.4)
                 with SMTP id AA02682; Sat, 12 Feb 94 20:23:47 +0100
Received: from tscisi12.cisi.unige.it by dist.dist.unige.it with SMTP
                 (5.61++/IDA-1.2.8) id AA25720; Sat, 12 Feb 94 20:23:07 +0100
Date: Sat, 12 Feb 94 20:23:07 +0100
Message-Id: <9402121923.AA25720@dist.dist.unige.it>
X-Sender: minollo@dist.unige.it
Mime-Version: 1.0
Content-Type: text/plain; charset="us-ascii"
To: wahba@ametista.imag.fr (ayman wahba), sicstus-users@sics.se
From: minollo@dist.unige.it (Carlo Innocenti)
Subject: Re: concatenate
Cc: vivi@itd.ge.cnr.it, sarti@itd.ge.cnr.it
X-Mailer: <PC Eudora Version 1.4b22>
Status: OR

At 10:12 AM 11/2/94 GMT, ayman wahba wrote:
>In my program, I use the concatenation of two strings a lot of times. I use
the >following:
>
>/*-------------------------------------------------------------------*/
>/* cat(string1,string2,string3)                                      */
>/* (i,i,o): concatenates 'string1' and 'string2'                     */
>/*-------------------------------------------------------------------*/
>cat(X,Y,Z):-
>   name(X,Xlist),
>   name(Y,Ylist),
>   append(Xlist,Ylist,Zlist),
>   name(Z,Zlist),!.
>
>append([],X,X).
>append([H|T],X,[H|T1]):-
```

© Michael Hess 2009

```
>   append(T,X,T1),!.
>
>
>The problem is that 'append of two lists' takes a lot of time if the first
list >is long, which greatly affects the performance of my program.

This is a frequent problem when working with Prolog and lists; it is due to
the asimmetry in Prolog list handling between heads and tails.
I think you have two main ways to optimize your code:
one is to write a foreign C predicate which uses the C standard library for
strings.
The other is to make your code optimize itself, "learning" about its task;
if you are concatenating a lot of strings, may be that often the first
string (X in your code) has the same length for many cat/3 calls; if this is
the case, the following code should solve your problems; every time it
founds a string X whose length has never been found before, it uses the
standard append process and it generates a new Prolog clause which is able
to immediately solve the concatenation of future strings X of the same
length of the given one.

cat(X,Y,Z):-
                 name(X,Xlist),
                 name(Y,Ylist),
                 cat_list(Xlist,Ylist,Zlist),
                 name(Z,Zlist),!.

%cat_list/3 makes the true concatenation; at the beginning the only clause
%is the following
cat_list(X,Y,Z):-
                 my_append(X,Y,Z,NextX,NextY,NextZ),
                 asserta((cat_list(NextX,NextY,NextZ):-!)),!.

%my_append/6 builds the arguments for the new cat_list/3 clauses which are
%asserted during the program execution
my_append([],X,X,[],NextY,NextY).
my_append([H|T],X,[H|T1],[AnElem|XRest],NextY,[AnElem|ZRest]):-
                 my_append(T,X,T1,XRest,NextY,ZRest),!.
```

© Michael Hess 2009

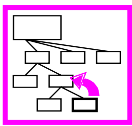
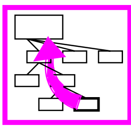
```
For example, the first time you ask
        ?-cat(i_am_,minollo,Z).
the time spent by the algorithm above is almost the same of the one you are
already using; but a new clause is asserted in the database,
        cat_list([A,B,C,D,E],F,[A,B,C,D,E|F]):-!.
if you ask again the same query, or a query where the first string is 5
characters long, the newly asserted query will solve the problem without
calling the my_append rule with a dramatic speed up of the algorithm.
Obviously, the number of cat_list/3 clauses will grow every time an X with a
length not yet examined is used.

I hope this helps.


Minollo
```



```
=========================================================
Minollo is
  Carlo Innocenti                  Phone: +39 10 353 2988
  c/o DIST - University of Genova
  Via all'Opera Pia, 11a
  16145 Genova - Italy
=========================================================
```


Ende des Unterdokuments

© *Michael Hess 2009*

```
successor(n1,n2)        :-  askif(y,a).
successor(n1,n3)        :-  askif(n,a).
successor(n2,n4)        :-  askif(y,d).
successor(n4,u)         :-  askif(y,e).
successor(n4,r)         :-  askif(n,e).
successor(n2,u)         :-  askif(n,d).
successor(n3,n5)        :-  askif(y,c).
successor(n5,t)         :-  askif(y,b).
successor(n5,v)         :-  askif(n,b).
successor(n3,n6)        :-  askif(n,c).
successor(n6,t)         :-  askif(y,d).
successor(n6,s)         :-  askif(n,d).
```

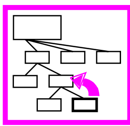
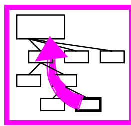
Interface-Prädikate:

```
diagnosis(D,Start)      :-  successor(Start,X),
                            diagnosis(D,X).
diagnosis(Node,Node).

askif(n,T).
askif(y,T):-  write('Is this correct: '), write(T), nl,
              read(y).

go         :-  diagnosis(D,n1), nl,
               write('The answer is: '), write(D), nl.
```

Vollständiger Code ⇒hier

Ende des Unterdokuments

© *Michael Hess 2009*

0.1 Fortgeschrittene Planungsverfahren

Übersicht:

1. Ausführungsüberwachung
2. Planung bei unvollständiger Information
3. abstrahierende und hierarchische Planer (ABSTRIPS)
4. nicht-lineare Planung (NOAH)
5. Planung unter zeitlichen Einschränkungen
6. parallele Aktionen
7. andere Typen von Zielen als einfache Zustände

0.1.1 Ausführungsüberwachung

Ausführen von Plänen erfordert Ausführungsüberwachung (‘‘execution monitoring’’). Planung und Ausführungsüberwachung sind oft strikt getrennt; bei Misserfolg eines einzelnen Schritts muss

1. entweder im fertigen Plan ein Schritt wiederholt werden
2. oder der fertige Plan modifiziert werden

Unrealistischer Idealfall. Daher:

© Michael Hess 2009

0.1.2 Planung bei unvollständiger Information

Oft kann man für die Planung erforderliche Kenntnisse der Welt erst *während* der Planausführung bekommen.. Beispiel: zum Flughafen fahren - welche Strassen sind von Staus betroffen? Daher:

1. Mögliche Antwort: Konditionale Pläne.
2. Aber: Führt oft zur kombinatorischen Explosion des Suchraums. Daher auch:
    a. Hierarchische Planung (siehe unten) und/oder
    b. Verschränken von Planung und Planausführung
3. Oft erfordert das Ermitteln von Wissen über die Situation selbst Planung.

0.1.3 Abstrahierende und hierarchische Planung

Nachteil der bisher beschriebenen Verfahren:

- detaillierte Ausarbeitung von u.U. sowieso unpassenden Planungsschritten
- Beispiel: *Wie kommme ich am schnellsten von New York nach Versailles?*
    1. zur U-Bahnstation gehen
    2. U-Bahn-Fahrkarte zum Hafen kaufen
    3. U-Bahn zum Hafen nehmen
    4. Büro der Cunard Line aufsuchen
    5. sich nach Reisen mit Passagierschiffen erkundigen - gibt’s nicht mehr
    6. alles obige rückgängig machen, etc.

Besser Breitensuche:

© Michael Hess 2009

1. zuerst eine Grobplanung ausarbeiten:
   a. Reisemittel finden: Schiff, Flugzeug, Eisenbahn, Auto?. Lösung: Flugzeug.
   b. dann die einzelnen Schritte eine Stufe tiefer planen:
      i. Bus zum Flughafen NY benutzen
      ii. Flugzeug nach Paris benutzen
      iii. von Paris nach Versailles Nahverkehrsmittel benutzen

   c. dann Busfahrt planen
      i. zur Busstation gehen
      ii. Bus-Fahrkarte zum Flughafen kaufen
      iii. Bus zum Flughafen nehmen

   d. dann Flug planen
      i. passenden Flug NY-Paris ermitteln
      ii. Ticket kaufen
      iii. Flugzeug besteigen

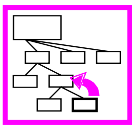
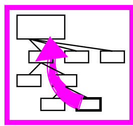
   e. <usw.>

Zu beachten:

- *Grundidee:* die Teilpläne müssen unabhängig voneinander ausgearbeitet werden können
- *erforderlich:* Interaktionen zwischen Detailschritten müssen auf den höheren Ebenen repräsentiert werden
- *daher:* hierarchische Planung nur sinnvoll, wenn ein grösserer Teil der Detailschritte keine Interaktionen über den Teilplan hinaus erzeugen

Beispiel: wenn der Schritt

© Michael Hess 2009

Ticket kaufen

nicht im Flughafen selbst realisiert werden könnte (Ticket-Verkauf nur im Stadt-Büro),

- so müsste dies als Vorbedingung für den Teilplan

  Flug nach Paris

  eine Ebene höher repräsentiert sein
- heisst nicht, dass man hier *nicht* hierarchisch planen kann: Vorbedingung von Teilplan 2 muss aber (auf der höheren Ebene) nach vorn gezogen werden

Siehe hierzu Georgeff 1987:378.

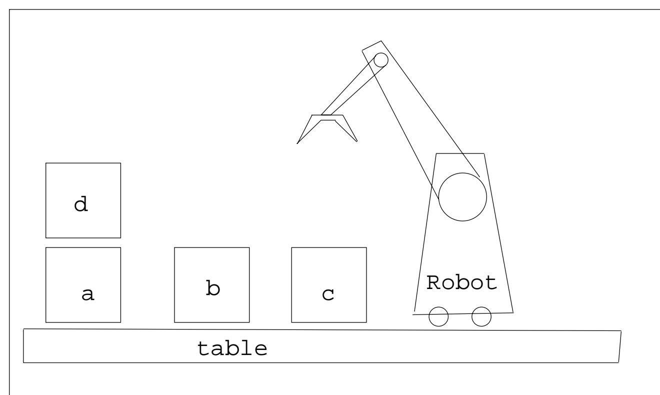
Frühe Realisierung: ABSTRIPS: (Shapiro 1987:582)

1. Vorbedingungen in Operatoren erhalten ‘‘criticality value’’ (Priorität)
2. zuerst alle Vorbedingungen mit höchster Priorität erfüllen
3. dann per Breitensuche weiter

Ergo: Ist Planung mit Best-First Strategie; ‘‘criticality value’’ ist die Evaluationsfunktion.

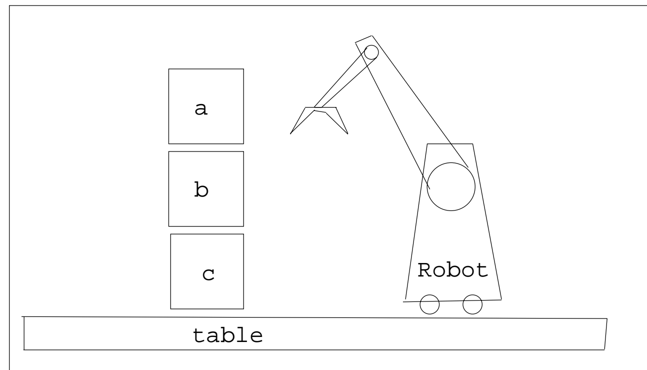
Beispiel:

- Ausgangslage in Klötzchenwelt:



© *Michael Hess 2009*

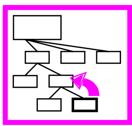
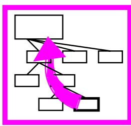
- Ziel:



- Operatoren:

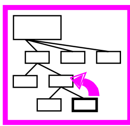
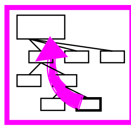
```
stack(R,X,Y)
ADD:            on(X,Y)
                at(R,Y)

CONDITIONS:     2: holding(R,X)
                2: at(R,Y)
                1: clear(Y)

TESTS:          -

DEL:            clear(Y)
```

© Michael Hess 2009

```
go_to(R,Y)
ADD:            at(R,Y)

CONDITIONS:     -

TESTS:          at(R,X)

DEL:            at(R,X)


unstack(R,X,Y)
ADD:            clear(Y)
                at(R,Y)

CONDITIONS:     3: holding(R,X)
                2: at(R,Y)
                1: clear(X)

TESTS:          on(X,Y)

DEL:            on(X,Y)


pickup(R,X)
ADD:            holding(R,X)

CONDITIONS:     3: handempty(R)
                2: clear(X)
                1: at(R,X)

TESTS:          -

DEL:            handempty(R)
```

© *Michael Hess 2009*

- Grobplan (Priorität 1) demnach:

```
[start,             stack(r,b,c),             stack(r,a,b)]
```

  Reihenfolge schon korrekt (wegen Bedingung ‘‘clear(X)’’ bezügl. Zielklotz)

- danach (Priorität 2):

```
[start,             stack(r,b,c),             stack(r,a,b)]

       pickup(r,b),              pickup(r,a),
       goto(r,c)                 goto(r,b)
```

- usw.

0.1.4 Nicht-lineare Planung

Oft ist es kontraproduktiv, sofort eine fixe Abfolge von Operatoranwendungen zu wählen und eventuelle Zielinteraktionen nachher zu reparieren. Statt dessen: Anwendung von Methoden der ‘‘Constraint Satisfaction’’

1. einzelne *Planelemente* (Teilsequenzen) *ungeordnet* belassen
2. erst, wenn ein Konflikt auftaucht, Teilordnung wählen
3. **oder:** sogar *Objekte* nicht mehr explizit spezifizieren, sondern inkrementell beschreiben
4. Beispiele: NOAH (Sacerdoti), NONLIN (Tate), SIPE (Wilkins) u.a.

© *Michael Hess 2009*

0.1.5 Planung unter zeitlichen Einschränkungen

Die beste Planung ist wertlos, wenn sie zu lange dauert. Daher:

1. Gewichtung von Zielen, und vordringliche Bearbeitung wichtiger Ziele
2. Gewichtung von Ereignissen (z.B. Gefährlichkeit, Dringlichkeit und vordringliche Beachtung wichtiger Ereignisse
3. oft kombiniert mit:
   a. hierarchische Planung
   b. Verschränken von Planung und Planausführung
   c. nicht-lineare Planung

   in allen Fällen wird ein Teil der Planungsarbeit auf später verschoben

Systeme mit derartigen Fähigkeiten: oft ‘‘reaktive Systeme’’ genannt

0.1.6 Planung mit parallelen Aktionen

Häufigste Fälle

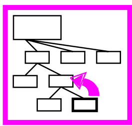
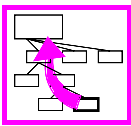
1. Mehrhändigkeit
2. mehrere unabhängige Agenten
3. ‘‘anstossbare’’ Prozesse

Zu beachten (wie üblich): Sequenzierung, Benutzen von Nebeneffekten, Zielinteraktionen

Zusätzlich erforderlich:

1. Vermeiden gegenseitiger Behinderung
2. Kommunikation zwecks Koordination

© *Michael Hess 2009*

3. Berücksichtigen des unterschiedlichen Wissensstands verschiedener Agenten

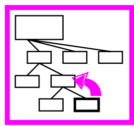
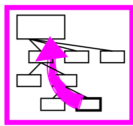
0.1.7 Andere Typen von Zielen

Nicht nur *Zustände* können Ziele sein, sondern auch

1. *Handlungen* selbst (oder: Transformationen) können Ziele sein
2. bestimmte *Sequenzen* von *Zuständen*;
3. bestimmte *Sequenzen* von *Handlungen*
4. nicht nur das *Erreichen* von Zuständen kann ein Ziel sein, sondern auch
   a. das *Aufrechterhalten* bestimmter Zustände
   b. das *Vermeiden* bestimmter Zustände
5. *formale Eigenschaften* des Plans selbst (z.B. Eleganz, Kürze, Einfachheit, etc.)

Ende des Unterdokuments

© *Michael Hess 2009*

Bei Tests ist zu berücksichtigen:

- *Sensitivität:* Wie viele ''false negatives'' werden gemeldet (resp. wie viele echte Fälle werden übersehen).

  Entspricht im IR der Ausbeute (recall).

- *Spezifizität:* Wie viele ''false positives'' werden gemeldet (resp. wie viele ''Pseudo-Fälle'' [spurious cases]) werden gemeldet.

  Entspricht im IR der Präzision.

Hierzu ein interessantes **Beispiel:** (die Zahlenwerte sind in etwa realistisch, aber nicht effektive Resultate von Untersuchungen)[3]

- *Prävalenz:* Die Wahrscheinlichkeit, dass eine symptomfreie Frau im Alter zwischen 40 und 50 Jahren, die an einer Mammographie-Reihenuntersuchung teilnimmt, Brustkrebs hat, beträgt 0.6%.

  Oder: $p(K) = 0.006$

- *Sensitivität:* Wenn eine der untersuchten Frauen Brustkrebs hat, dann beträgt die Wahrscheinlichkeit, dass der Mammographie-Befund positiv ausfällt, 94%. D.h. 6% falsch Negative.

  Oder: $p(M+ \mid K) = 0.94$

- *Spezifizität:* Wenn bei einer Frau aus dieser Gruppe kein Brustkrebs vorliegt, dann beträgt die Wahrscheinlichkeit, dass ein Mammographie-Befund dennoch positiv ausfällt, 7%. D.h. 7% falsch Positive

  Oder: $p(M+ \mid \neg K) = 0.07$

Was ist die Wahrscheinlichkeit, dass eine 50jährige Frau mit *positivem* Mammographie-Befund Krebs hat?

---

3. Siehe auch: ⇒hier.

© *Michael Hess 2009*

Die meisten Leute (inkl. Allgemeinmediziner!) schätzen die Wahrscheinlichkeit auf 70% bis 80%.

Tatsächlich beträgt die aus der gegebenen Information ermittelte Wahrscheinlichkeit, Krebs zu haben für die 50jährige Frau mit *positivem* Mammographie-Befund lediglich 7,5 Prozent.

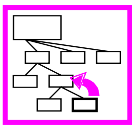
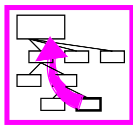
Woher kommen diese grotesken Fehlbeurteilungen? Die Berechnung (u.a. nach Bayes) ist

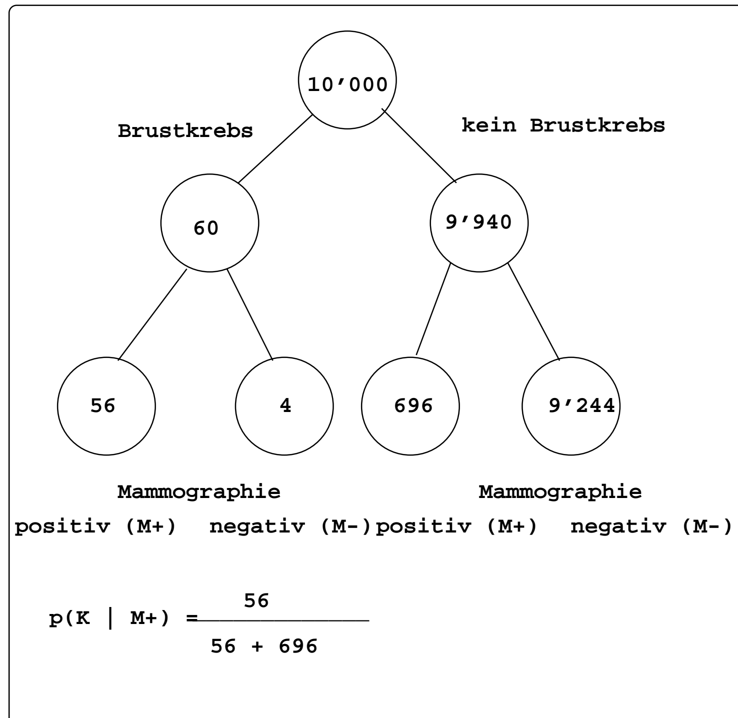
$$p(K \mid M+) = \frac{0.94 \times 0.006}{0.006 \times 0.94 + 0.994 \times 0.07} = \frac{0.00564}{0.07522} = 0.07498$$

Das ist schwer zu vestehen! Doch es gibt einen Trick, die oben erwähnten Angaben so zu ‘‘übersetzen"’’ dass jedem das Ergebnis unmittelbar einleuchtet. Man ersetzt einfach die Prozentangaben durch natürliche Häufigkeiten (Zahlen z.T. gerundet):

1. Von 10’000 asymptomatischen Frauen haben 60 Brustkrebs.
2. Von diesen 60 weisen 56 einen positiven Mammographiebefund auf.
3. Aber auch von den verbleibenden 9’940 gesunden Frauen zeigen 696 eine positive Mammographie.

© *Michael Hess 2009*

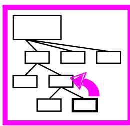
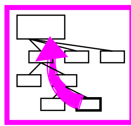


Das ergibt:

$$p(K|M+) = \frac{56}{56+696} = 0.074468$$

Ende des Unterdokuments

© Michael Hess 2009

Exkurs: Was ist unter ‘‘unreinem’’ Prolog zu verstehen?

Es ist Prolog mit Konstrukten,

1. welche selbst Einfluss auf das Verhalten des Interpreters nehmen
2. welche auf den Zustand des Interpreters während des Beweises Bezug nehmen
3. welche den Zustand des Programms selbst verändern (Selbstmodifikation)

Ad 1: Der Cut (‘!’) steuert das Verhalten des Interpreters ganz direkt

Ad 2: ‘var(V)’, ‘integer(I)’ etc. sprechen über den *gegenwärtigen* Bindungszustand der Variablen. Reihenfolge der Terme entscheidend

```
?- var(X), X=a, nonvar(X).
```

gelingt, jede andere Reihenfolge nicht!

Die Meta-Variable (für ‘call(X)’) erlaubt, Daten zu Programmen zu machen.

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

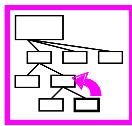
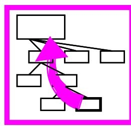
Ist im Grunde ein Programm-*Schema:* Steht für beliebig viele unterschiedliche Klauseln (d.h. jeder Bindungszustand von ‘‘E’’ ergibt ein potentiell neues Programm).

Daher: ‘‘not(X)’’ ist *keine Negation im logischen Sinn*, sondern Nicht-Beweisbarkeit (‘‘negation by failure’’). Das ist ein *meta-logisches* Konzept (‘‘im gegenwärtigen Zustand des Axiomsystems nicht beweisbar’’). Allerhand Folgen: ‘‘Closed World Assumption’’, Nicht-Monotonie der Schlussfolgerungen

ad 3: ‘assert(E)’ und ‘retract(E)’ modifizieren das Programm ‘‘fliegend’’. Man kann Fakten und Regeln einfügen oder entfernen. Ist natürlich weit ausserhalb der Logik erster Stufe (die spricht über *gegebene*, und unveränderliche, Axiomensysteme). Resultiert natürlich ebenfalls in Nicht-Monotonie

Ende des Unterdokuments

© *Michael Hess 2009*

Man kann Beweisbäume leicht automatisch generieren lassen, allerdings nicht in einer besonders lesefreundlichen Form. Man muss dazu nur einen kleinen Meta-Interpreter (siehe dazu weiter unten) schreiben, der immer *alle* Lösungen zu finden versucht und der dabei alle Erfolge *und* alle Misserfolge, die während eines Beweises ermittelt werden, protokolliert.

Für das Ziel `grandfather(peter,Z)` (das Beispiel in der Illustration oben) ergibt das (gänzlich unediert):

```
    trying to prove grandfather(peter,_112)
        trying to prove father(peter,_241)
        successfully proved predicate father(peter,john)
        trying to prove parent(john,_112)
            trying to prove father(john,_112)
            successfully proved predicate father(john,jim)
        successfully proved predicate parent(john,jim)
    successfully proved predicate grandfather(peter,jim)

OR
            failure to prove predicate father(john,_112)
            trying to prove mother(john,_112)
            failure to prove predicate mother(john,_112)
        failure to prove predicate parent(john,_112)
        failure to prove predicate father(peter,_241)
    failure to prove predicate grandfather(peter,_112)

no
```

und für das Ziel `grandfather(X,Y)` (mehrere Lösungen)

```
    trying to prove grandfather(_111,_112)
        trying to prove father(_111,_241)
        successfully proved predicate father(peter,john)
        trying to prove parent(john,_112)
            trying to prove father(john,_112)
            successfully proved predicate father(john,jim)
        successfully proved predicate parent(john,jim)
    successfully proved predicate grandfather(peter,jim)

OR
```

© Michael Hess 2009

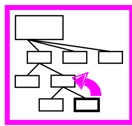
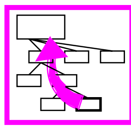
```
            failure to prove predicate father(john,_112)
            trying to prove mother(john,_112)
            failure to prove predicate mother(john,_112)
        failure to prove predicate parent(john,_112)
        successfully proved predicate father(john,jim)
        trying to prove parent(jim,_112)
            trying to prove father(jim,_112)
            failure to prove predicate father(jim,_112)
            trying to prove mother(jim,_112)
            failure to prove predicate mother(jim,_112)
        failure to prove predicate parent(jim,_112)
        successfully proved predicate father(bob,mary)
        trying to prove parent(mary,_112)
            trying to prove father(mary,_112)
            failure to prove predicate father(mary,_112)
            trying to prove mother(mary,_112)
            successfully proved predicate mother(mary,jill)
        successfully proved predicate parent(mary,jill)
    successfully proved predicate grandfather(bob,jill)

OR
            failure to prove predicate mother(mary,_112)
        failure to prove predicate parent(mary,_112)
        failure to prove predicate father(_111,_241)
    failure to prove predicate grandfather(_111,_112)

no
```

und schliesslich für das Ziel `grandfather(mary,X)` (gar keine Lösung)

```
    trying to prove grandfather(mary,_112)
        trying to prove father(mary,_241)
        failure to prove predicate father(mary,_241)
    failure to prove predicate grandfather(mary,_112)

no
```

Daraus kann man die Suchbäume ableiten.

© *Michael Hess 2009*

Der vollständige Code ist ⇒hier.

Zu beachten ist dabei, dass nicht alle Fehlschläge ‘‘echt’’ sind - da wir (im Prädikat `pr(G)`) einen künstlichen Fehlschlag auslösen, um alle Beweise zu finden, ergeben sich auch Meldungen der Art `failure to prove predicate ...`, welche nicht echte Fehlschläge sind (jene direkt nach dem `OR`).

Andere (sauberere) Lösungen sind auch denkbar, abder sie werden rasch aufwendiger (Hint: `setof` verwenden plus ein Argument, um den Beweisbaum inkl. Meldungen intern zu akkumulieren, statt laufend auf den Schirm zu schreiben).

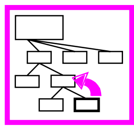
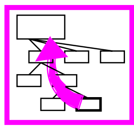
Ende des Unterdokuments

© *Michael Hess 2009*

Der Nachweis dafür, dass genau das in Prolog passiert, ist der Trace:

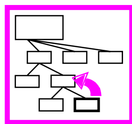
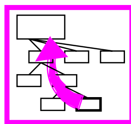
```
| ?- p.
p.
   1  1  Call: p ?

   2  2  Call: a ?

   3  3  Call: d(_466) ?

   4  4  Call: h ?

   5  5  Call: b ?

   5  5  Exit: b ?

   4  4  Exit: h ?

   6  4  Call: i(_466) ?

   7  5  Call: y(_466) ?

   7  5  Fail: y(_466) ?

   7  5  Call: g(_466) ?

   7  5  Exit: g(2) ?

   6  4  Exit: i(2) ?

   3  3  Exit: d(2) ?

   2  2  Exit: a ?

   1  1  Exit: p ?


yes
```

© *Michael Hess 2009*

Ende des Unterdokuments

© *Michael Hess 2009*

Absteigende rechtsläufige Tiefensuche

[Ein Versuch, das graphisch darzustellen]

(0.L.6)

Ein Versuch, das graphisch darzustellen:

```
Beweis von ``?- p. '' (ohne Ausbreitung der Variablenbindungen):

|                                |
 -------------------------------
```

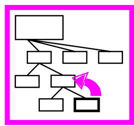
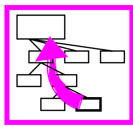
Ziele "pusht" man in den Speicher...

```
push p (Theorem):

|          p.                    |
 -------------------------------
```

..."pop" oberstes Element, "expand" es durch seine "Definition(en)", und behandle die als Ziel(e).

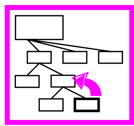
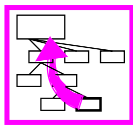
```
pop, expand, push:

|   a.                           |
|   b,                           |
|   c(X),                        |
|   d(X).                        |
 -------------------------------

alle Expansionen von p! Man beachte: ``.'' vs. ``,''
```

© *Michael Hess 2009*

```
pop Top-Element (:a), expandiere es, pushe das
Resultat:

|   d(X).                        |
|   b,                           |
|   c(X),                        |
|   d(X).                        |
 ------------------------------

``Bodensatz'' bleibt!


|   h,                           |
|   i(X).                        |
|   k,                           |
|   l.                           |
|   b,                           |
|   c(X),                        |
|   d(X).                        |
 ------------------------------
alle (beide) Expansionen von d/1!


|   b,                           |
|   i(X).                        |
|   x,                           |
|   i(X).                        |
|   k,                           |
|   l.                           |
|   b,                           |
|   c(X),                        |
|   d(X).                        |
 ------------------------------

alle (beide) Expansionen von h/0.
```

© Michael Hess 2009

Absteigende rechtsläufige Tiefensuche
[Ein Versuch, das graphisch darzustellen]

(0.L.6)

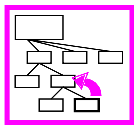
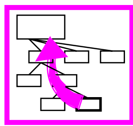

```
|    i(X).                       |
|    x,                          |
|    i(X).                       |
|    k,                          |
|    l.                          |
|    b,                          |
|    c(X),                       |
|    d(X).                       |
 -------------------------------

'b' ist Faktum: Erfolg


|    y(X).                       |
|    g(X).                       |
|    x,                          |
|    i(X).                       |
|    k,                          |
|    l.                          |
|    b,                          |
|    c(X),                       |
|    d(X).                       |
 -------------------------------
```

© Michael Hess 2009

```
y(X) kann nicht reduziert werden: Fehlschlag.

Daher:
pop y(X);
pop (neues) Top-Element (g(X)), expandiere es, pushe das Resultat:

|   g(X).                         |
|   x,                            |
|   i(X).                         |
|   k,                            |
|   l.                            |
|   b,                            |
|   c(X),                         |
|   d(X).                         |
 -------------------------------

g(X) wird zu g(2) expandiert.
X → 2.


pop Top-Element (TRUE).

|   TRUE.                         |
|   x,                            |
|   i(X).                         |
|   k,                            |
|   l.                            |
|   b,                            |
|   c(X),                         |
|   d(X).                         |
 -------------------------------

Keine (aktive) Ziele mehr: ERFOLG!
```

Wenn man das explizit als Suche im Zustandsraumraum implementieren will, kann man den Beweisablauf genau verfolgen (Programm bei Rowe 1988:229)

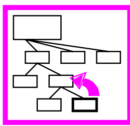
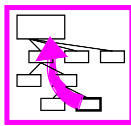
© *Michael Hess 2009*

Der Nachweis (0.L.6.L.1)

Ende des Unterdokuments

© *Michael Hess 2009*

Aufsteigende rechtsläufige Breitensuche
[Konkrete Implementation]

(0.L.7)

Implementation eines allgemeinen Breitensuchprogramms (mit Daten aus einem Expertensystembeispiel; aus Rowe 1988:232, und davon abgeleitet: ‘bottomup.pl’; spezielle Programme, wie Chart-Parser, können effizienter sein):

Kern des (bereichsunabhängigen) Programms:

```
breadthsearch(Start,Ans)        :-
    cleandatabase,                               % 1
    asserta(agenda(Start,[Start])),
    agenda(State,Oldstates),                     % 2
    find_successors(State,Oldstates,Newstate),
    goalreached(Newstate),
    agenda(Newstate,Ans),
    retract(agenda(Newstate,Ans)),
    asserta(oldagenda(Newstate,Ans)).

find_successors(State,Oldstates,Newstate)       :-
    successor(State,Newstate),
    \+(State = Newstate),                         % 3
    \+agenda(Newstate,S),                         % 4
    \+oldagenda(Newstate,S),                      % 5
    assertz(agenda(Newstate,[Newstate|Oldstates])). %6

find_successors(State,Oldstates,Newstate)       :-
    retract(agenda(State,Oldstates)),
    asserta(oldagenda(State,Oldstates)),
    fail.
```

1: Entfernen von Material von früheren Sessionen
2: findet wiederholt Zustände, zu denen Nachfolger gefunden werden müssen
3: neuer Zustand ist nicht: gegenwärtiger Zustand
4: neuer Zustand ist nicht: auf der Agenda
5: neuer Zustand ist nicht: erschöpfter Zustand
6: Breitensuche! (durch assert**z** erreicht)

Bereichsspezifische Definition von Nachfolgezuständen; hier: Klötzchenwelt.

`'on(X,Y,Z)'` heisst: *X* ist auf *Y*, und zudem auf dem ‘peg’ *Z* aufgespiesst

`'successor(S,[on(X,surface,none)|S2])'` heisst: der Nachfolgezustand von S ist `'on(X,surface,none)'`

© *Michael Hess 2009*

Zustandsbeschreibungen sind komplette Weltbeschreibungen in Listenform (in Variable *S*)

```
% Bewege X auf den Boden (ohne es irgendwo aufzuspiessen)
successor(S,[on(X,surface,none)|S2])    :-      % 1
                        member(on(X,Y,B),S),
                        \+(B=none),
                        cleartop(X,S),          % 2
                        delete(on(X,Y,B),S,S2). % 3


% Bewege X auf ein anderes Objekt (Z),das auf Dorn B2 ist:
successor(S,[on(X,Z,B2)|S2])    :-
                        member(on(X,Y,B),S),
                        cleartop(X,S),          % 2
                        member(on(Z,W,B2),S),
                        \+(B2=none),           % 4
                        cleartop(Z,S),          % 5
                        \+(X=Z),
                        delete(on(X,Y,B),S,S2).

% Bewege X auf Dorn B2, direkt auf den Boden:
successor(S,[on(X,surface,B2)|S2])      :-
                        member(on(X,Y,B),S),
                        cleartop(X,S),
                        member(bolt(B2),S),     % 6
                        \+member(on(Z,W,B2),S),
                        delete(on(X,Y,B),S,S2).


cleartop(Part,State)
                :- \+member(on(X,Part,B),State).


%% Description of goal state:

goalreached(S)  :-  member(on(a,d,bolt2),S).
```

© Michael Hess 2009

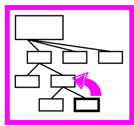
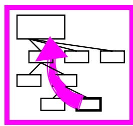
```
also:
          |            |
          |            a
          |            d
-----------------------------------
          1            2

%% Description of the world:

start_state([on(a,b,bolt1),
        on(b,c,bolt1),
        on(c,surface,bolt1),
        on(d,e,bolt2),
        on(e,f,bolt2),
        on(f,surface,bolt2),
        bolt(bolt1),
        bolt(bolt2)]).

also:

          a            d
          b            e
          c            f
-----------------------------------
          1            2
```

1: ‘‘bewege X von (wo immer es ist) auf den Boden (ohne es irgendwo aufzuspiessen)’’
2: X kann nur bewegt werden, wenn es z.Z. leer ist
3: nicht hier definiert; entfernt Eintrag von der Liste
4: Z darf nicht unaufgespiesst sein (d.h.: auf den Boden zu liegen kommen)
5: auch das Zielobjekt muss leer sein
6: muss auf einen Stab zu liegen kommen

Beachte: Beschreibung von Nachfolgerzuständen (natürlich) bereichsspezifisch, aber *kein* Einfluss des Bereichswissens auf die Abarbeitungsstrategie (also immer noch bereichsunspezifische Suchstrategie).

Über der gleichen Beschreibung der Welt wie oben kann man auch (aufsteigende)

© *Michael Hess 2009*

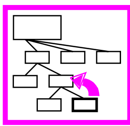
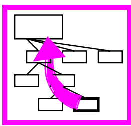
Tiefensuche implementieren:

```
depthsearch(Start,Ans)  :-
   depthsearch(Start,[Start],Ans).

depthsearch(State,Statelist,Statelist)  :-
   goalreached(State).
depthsearch(State,Statelist,Ans)        :-
   successor(State,Newstate),
   \+member(Newstate,Statelist),
   depthsearch(Newstate,[Newstate|Statelist],Ans).
```

Ist viel einfacher, weil man den in Prolog eingebauten Tiefensuch-Mechanismus direkt verwenden kann.

Ende des Unterdokuments

© *Michael Hess 2009*

Kann man durch ‘term_expansion’ automatisch tun lassen:

```
term_expansion((Head ---> Body),(Head ==> Expanded))
        :-      expand_body(Body,Expanded).

expand_body([Body],[Body])  :-  !.
expand_body([],[])          :-  !.
expand_body(Body,Expanded)  :-
                        not(islist(Body)),
                        conjtolist(Body,Expanded).

conjtolist((Term,Terms), [Term|List_of_terms]) :- !,
        conjtolist(Terms,List_of_terms).
conjtolist(Term,[Term]).

islist([]) :- !.
islist([_|_]).
```

Ende des Unterdokuments

© *Michael Hess 2009*

Der Satz

A brown dog eats the men

würde über obiger Grammatik so analysiert:

1. Der Aufruf ist

```
parse(sent(S),[a,brown,dog,eats,the,men],[])
```

   d.h. man geht davon aus, dass es ein Satz ist

2. nun beginnt man mit dem ersten Wort, also ‘‘a’’. Davon weiss man, dass es ein ‘‘Blatt’’ des Baums ist.

```
5   4   Call: leaf(_964,[a,brown,dog,eats,the,men],_966) ?

6   5   Call: 'C'([a,brown,dog,eats,the,men],_1143,_966) ? s
6   5   Exit: 'C'([a,brown,dog,eats,the,men],a,[brown,dog,eats,the,men])
7   5   Call: _964==>[a] ?

7   5   Exit: det(det(a),sing)==>[a]
```

3. anhand des Prädikats ‘link’ testet man, ob diese Kategorie tatsächlich die linke Ecke eines Satzes sein kann (2). Dazu verwendet man die aus der spezifischen Grammatik abgeleitete Liste

```
link(np(_,_), sent(_)).
link(det(_,_), np(_,_)).
link(det(_,_), sent(_)).
link(v(_,_,_), vp(_,_)).
link(adj(_),adjp(_)).
link(X,X).
```

   ‘‘top-down-Filtern’’ der von ‘leaf’ generierten Hypothesen.

© *Michael Hess 2009*

```
8   4   Call: link(det(det(a),sing),sent(S)) ?
                                    ^^^^^^^
8   4   Exit: link(det(det(a),sing),sent(S))
```

4. man versucht nunmehr anhand des Prädikats 'lc' zu ermitteln, wovon dieses Blatt *innerhalb der übergeordneten Konstituente* die linke Ecke sein könnte (3). 'det' ist nach der Regel

```
np(np(A,B,C,D),A)    ==>   [det(B,A),adjp(C),n(D,A)].
```

die linke Ecke einer 'np'.

```
 9   4   Call: lc(det(det(a),sing),sent(S),[brown,dog,eats,the,men],[]) ?

10   5   Call: _2410==>[det(det(a),sing)|_2408] ?

10   5   Exit: np(np(sing,det(a),AP,N),sing)
                 ==>[det(det(a),sing),adjp(AP),n(N,sing)]
```

5. und gemäss 'link' ist sie es innerhalb der Konstituente ''sent''

```
11   5   Call: link(np(np(sing,det(a),AP,N),sing),sent(S)) ?

11   5   Exit: link(np(np(sing,det(a),AP,N),sing),sent(S))
```

6. Nunmehr stellt man in 'parse_rest' *ab*steigend die Hypothese auf, dass als nächstes eine 'adjp' folgt, dann eine 'n'

```
13   5   Call: parse_rest([adjp(AP),n(N,sing)],[brown,dog,eats,the,men],U)

13   6   Call: parse(adjp(AP),[brown,dog,eats,the,men],V) ?
```

© *Michael Hess 2009*

7. dabei wird jede einzelne dieser Konstituenten durch ‘‘parse’’ in der Definition von ‘parse_rest’ wieder *auf*steigend bewiesen, wobei man aber wieder bei der Wortform, also ‘‘brown’’, beginnt, und via die Regel

   ```
   adj(brown)==>[brown].
   ```

   aufsteigend ‘‘adj’’ zu beweisen versucht

   ```
   14   7   Call: leaf(_3485,[brown,dog,eats,the,men],_3487) ? s

   14   7   Exit: leaf(adj(brown),[brown,dog,eats,the,men],[dog,eats,the,men]
   ```

   Der Link-Test ist hier schon sehr eingeschränkt:

   ```
   17   7   Call: link(adj(brown),adjp(AP)) ? s

   17   7   Exit: link(adj(brown),adjp(AP))
   ```

   Beweis für ‘‘linke Ecke’’ mit der schon durch ‘‘adjp’’ instantiierten Ergebnis-Variablen

   ```
   18   7   Call: lc(adj(brown),adjp(AP),[dog,eats,the,men],V) ? s

   18   7   Exit: lc(adj(brown),adjp(adjp(brown,[])),[dog,eats,the,men],[dog,
   13   6   Exit: parse(adjp(adjp(brown,[])),[brown,dog,eats,the,men],[dog,ea
   ```

8. die nächste Wortform, ‘‘dog’’, wird aufsteigend analysiert:

   ```
   33   6   Call: parse_rest([n(N,sing)],[dog,eats,the,men],U) ?
   ```

9. rekursiv versucht man nun, zu ermitteln, ob die gesamte von unten her aufgebaute Konstituente (‘‘np’’) die Ecke einer umfassenderen Konstituente (‘‘sent’’) darstellt, deren Einzelteile (‘‘vp’’) man dann einzeln verifiziert, etc.

© *Michael Hess 2009*

```
43   5   Call: lc(np(np(sing,det(a),adjp(brown,[]),n(dog)),sing),sent(S),
                       [eats,the,men],[]) ?

44   6   Call: _8067==>[np(np(sing,det(a),adjp(brown,[]),n(dog)),sing)|_80

44   6   Exit: sent(sent(np(sing,det(a),adjp(brown,[]),n(dog)),VP))
    ==>[np(np(sing,det(a),adjp(brown,[]),n(dog)),sing),vp(VP,sing)]
<...>
46   6   Call: parse_rest([vp(VP,sing)],[eats,the,men],_8057) ?
```

Der letzte Schritt ist konzeptionell vielleicht etwas verwirrend, deshalb einige Erläuterungen: Es ist ganz normal in der Logik-Programmierung, einen schon von anderswo (und hier eben von *oben*) instantiierten Wert zu *verifizieren*; (und hier eben aufsteigend). Die Verifikation bleibt aber aufsteigend. Anhand des Beispiels der Verbalphrase:

- zwar ist der Wert `vp` schon von oben vorgegeben, aber die *Bestätigung* kann von unten geschehen
- der Wert ist zudem erst *teilweise*, durch `vp(`**`VP`**`,sing)`, vor-instantiiert

---

Ende des Unterdokuments

© Michael Hess 2009

© ***Michael Hess 2009***