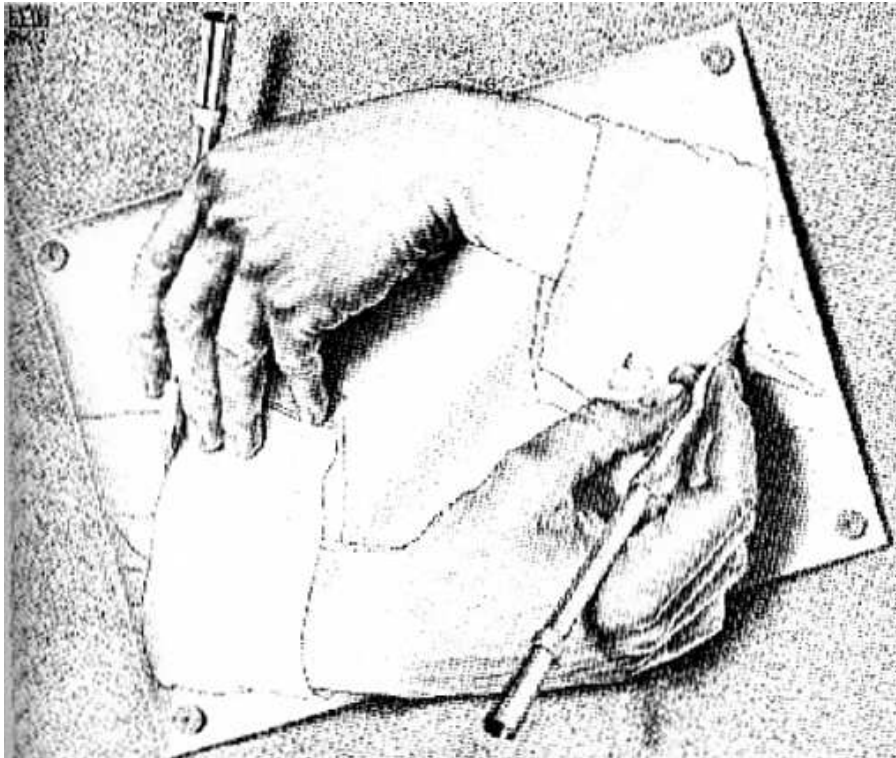
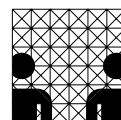


Softwareentwicklung III: Logikprogrammierung

Wolfgang Menzel



Universität Hamburg
Department Informatik
WS 2016/2017



Diese Prüfungsunterlagen sind eine Materialsammlung, die Ihnen dabei helfen soll, sich den Stoff der Vorlesung Logikprogrammierung im Modul Softwareentwicklung III zu erarbeiten. Sie finden hier die wichtigsten Fachbegriffe und Konzepte, die in der Vorlesung eingeführt werden, Verweise auf die einschlägige Literatur, Querverweise zu verwandten Themen in anderen Vorlesungen, sowie die Vorlesungsfolien mit vielen Programmbeispielen.

Dies ist kein Skript zur Vorlesung! Die Veranstaltung behandelt eine vielleicht etwas ungewöhnliche Sicht auf die Grundlagen der Praktischen Informatik, zu der es dennoch zahlreiche gute Lehrbücher gibt. Deshalb sehe ich auch wenig Sinn darin, noch einmal neu aufzuschreiben, was andere schon sehr gut dargestellt haben. Das Studium wenigstens eines Lehrbuches zur Logikprogrammierung ist daher ein unverzichtbarer Bestandteil der Veranstaltung und der Leseausweis der Hamburger Hochschulbibliotheken ist für einen Studienerfolg wohl mindestens so wichtig wie die selbständige Lösen der Übungsaufgaben mit Hilfe eines der gängigen Prolog-Systeme. In der Vorlesung, in den Übungen und beim Durcharbeiten der Literatur lernen Sie Grundbegriffe, Konzepte und Methoden der Logikprogrammierung kennen. In den praktischen Übungen am Rechner werden Sie die Fertigkeiten erwerben, mit diesen Konzepten sicher umzugehen.

Diese Fähigkeit zur sinnvollen Anwendung von programmiersprachlichen Konzepten beim Lösen praktischer Problemstellungen ist sicherlich das wichtigste Lernziel der Veranstaltung. Man erwirbt sie nicht durch das Auswendiglernen von Merksätzen, sondern eher durch den wiederholten und durchaus auch spielerischen Umgang mit konkreten Programmbeispielen. Nutzen Sie dazu Ihr Programmiersystem als Helfer: Es kann Ihnen viele Fragen beantworten, auf die sie in Lehr- und Handbüchern nur schwer eine Antwort finden werden. Sie müssen nur lernen, die richtigen Fragen zu stellen und die Ergebnisse richtig zu interpretieren. Modifizieren Sie systematisch die in der Vorlesung und den Übungen diskutierten Lösungsansätze und beobachten Sie sorgfältig die daraus resultierenden Veränderungen im Programmverhalten. Und vergessen Sie dabei niemals die Frage nach dem "warum" zu stellen.

Für diejenigen unter Ihnen, die bisher noch nie mit Logikprogrammierung in Berührung gekommen sind, wird gerade zu Beginn vieles sehr ungewohnt und vielleicht auch einigermaßen sinnlos erscheinen. Lassen Sie sich überraschen! Und lassen Sie sich nicht durch die auf den ersten Blick fast trivialen Grundprinzipien täuschen: Erst in Ihrem Zusammenspiel entfaltet sich die ganze Schönheit und Mächtigkeit dieses Programmierparadigmas. Doch um bis zu dieser Stufe der Einsicht zu gelangen, ist die Gewöhnung an das Ungewohnte unverzichtbare Voraussetzung. Lassen Sie sich auch auf ein kleines Abenteuer ein. Alles, was Sie dafür mitbringen müssen, ist Neugier und Offenheit für Neues. Einige Ihnen vielleicht liebgewordene Denkschemata aus anderen Programmierparadigmen sollten Sie allerdings lieber erst einmal zu Hause lassen. Als Lohn winkt Ihnen eine ganz andersartige Sicht auf die Programmierung, in der so manche, recht komplexe Problemstellung auf einmal eine ganz einfache und natürliche Lösung findet. In diesem Sinne bietet Ihnen das Paradigma der Logikprogrammierung dann auch einen alternativen *Denkansatz*, der zu ganz neuen Einsichten bei der Problemanalyse und dem Lösungsdesign führen kann. Die Umsetzung in eine konkrete Implementation kann dann letztendlich in jeder beliebigen Programmiersprache Ihrer Wahl erfolgen ...

Die Lehrveranstaltung wird Ihnen nicht alle Details vermitteln können, die Sie für die Entwicklung umfangreicherer Programmsysteme benötigen. Hierfür stehen Ihnen Handbücher und auch die On-line-Hilfen der bereitgestellten Prologsysteme zur Verfügung. Viele dieser Unterlagen sind nur in englischer Sprache verfügbar. Schon deshalb möchte ich Sie nachdrücklich dazu ermuntern, englischsprachige Literatur in der ursprünglichen Fassung zu lesen und nicht auf die Übersetzungen zurückzugreifen. Als Informatiker werden Sie nicht darum herumkommen, über weite Strecken mit englischsprachiger Literatur zu arbeiten.

Die Logikprogrammierung ist ein Gebiet, das mich auch nach über zwanzig Jahren immer wieder fasziniert, in dem auch ich immer wieder Überraschendes entdecken kann und dies trotz der Tatsache, dass sie sich weder in ihren Grundzügen noch in den Details von Syntax und Semantik der einschlägigen Programmiersprachen wesentlich verändert hat. Sie ist damit sicherlich eines der stabilsten Programmierparadigmen, das die Informatik hervorgebracht hat. Ich schätze die Logikprogrammierung besonders, weil sie das Denken in *Alternativen* auf ganz fundamentale und dennoch natürliche Weise fördert und fordert. Nur im Ausnahmefall gibt es in unserem täglichen Leben auf eine Frage eine eindeutige Antwort. Die Logikprogrammierung erhebt genau dies zum Grundprinzip und zwar mit aller Konsequenz: Selbst der Umgang mit unendlich großen Antwortmengen stellt hier nichts sonderlich Außergewöhnliches dar. Aber auch andere Merkmale sind es, die mir immer wieder großen Respekt für ein geradezu einzigartiges Sprachdesign abverlangen: Zu sehen etwa, wie die dynamischen Aspekte eines algorithmischen Ablaufs immer weiter in den Hintergrund treten, bis sie hinter einigen wenigen statischen Unifikationsforderungen vollständig verschwinden. Auch die Möglichkeit, Syntax *und* Semantik der Sprache ganz gezielt und individuell an die jeweiligen Bedürfnisse einer Aufgabenstellung anzupassen, gehört hier sicherlich dazu.

Für mich war die Logikprogrammierung in vielen Fällen erst der Schlüssel zum Verständnis der formalen Grundlagen der Informatik. Ich hoffe, auch Sie werden die Vorzüge eines Programmierparadigmas kennen und schätzen lernen, das das Lösen praktischer Programmieraufgaben auf ganz natürliche Weise mit strikt logischem Denken, sowie dem Nachdenken über die Möglichkeiten und Grenzen formaler Berechnungssysteme verbindet. Wohl nirgendwo kann man auf so direktem Wege erfahren, dass die Beschäftigung mit den formalen Grundlagen keine zusätzliche Hürde auf dem Weg zum gestandenen Informatiker darstellt, sondern dass man sich damit einen Werkzeugkasten von Methoden und Formalismen erarbeitet, der für die tägliche Arbeit außerordentlich hilfreich sein kann.

Sie merken schon: bei diesen Fragen rational zu bleiben, fällt mir nicht ganz leicht! Ich gehe daher auch in die Veranstaltung mit der Hoffnung, einen Teil meiner Begeisterung für die "etwas andere Art der Programmierung" an Sie weitergeben zu können. Vor allem freue ich mich auf einen spannenden Dialog, der auch mich zum Nachdenken bringt über Dinge, die ich bisher vielleicht immer nur als Selbstverständlichkeit hingenommen habe.

Hamburg, im September 2016

Wolfgang Menzel

Inhaltsverzeichnis

1	Allgemeines	5
2	Programmierparadigmen	9
2.1	Verarbeitungsmodelle	9
2.2	Abstraktion	14
2.3	Programmierstile	22
2.4	Logikprogrammierung	23
3	Relationale Datenbanken	25
3.1	Relationen	25
3.2	Fakten	27
3.3	Anfragen	30
3.4	Anfragen mit Variablen	33
3.5	Komplexe Anfragen	41
3.6	Prädikate zweiter Ordnung	45
3.7	Relationale DB-Systeme	45
4	Deduktive Datenbanken	47
4.1	Deduktion	47
4.2	Regeln	48
4.3	Spezielle Relationstypen	54
4.4	Anwendung: Wegplanung	61
5	Rekursive Datenstrukturen	65
5.1	Eingebettete Strukturen	66
5.2	Arithmetik, relational	69
5.3	Operatorstrukturen	78
5.4	Arithmetik, funktional	82
6	Verkettete Listen	96
6.1	Listennotation	96
6.2	Listenunifikation	98
6.3	Listenverarbeitung	100
6.4	Suchen und Sortieren	114
6.5	Memoization	119
6.6	Suchraumverwaltung	122
6.7	Bäume	124
7	Extra- und Metalogische Prädikate	126
7.1	Typen und Typkonversion	127
7.2	Kontrollierte Instanziierung	130
7.3	Suchraummanipulation	133
7.4	Extralogische Prädikate	139

8	Prädikate höherer Ordnung	140
8.1	Prädikatsaufruf	140
8.2	Steuerstrukturen	144
8.3	Resultatsaggregation	149
8.4	Metaprogrammierung	153
9	Funktionale Programmierung	160
9.1	Grundbegriffe	160
9.2	Umgebungen	167
9.3	Funktionale Auswertung	171
9.4	Rekursive Funktionen	179
9.5	Funktionen höherer Ordnung	181
9.6	Ein abschließender Vergleich	186
10	Aktive Datenstrukturen	188
10.1	Offene Listen	189
10.2	Differenzlisten	192
10.3	Definite Clause Grammar	199
11	Fazit und Ausblick	210
11.1	Prädikatenlogische Grundlagen	210
11.2	Logikprogrammierung, wozu?	213
11.3	Erweiterungen	215

1 Allgemeines

SE III: Logikprogrammierung

Wolfgang Menzel

Telefon: 42883-2435

E-Mail: menzel@informatik.uni-hamburg.de

Raum: F-411

Sprechstunde: Di 16 - 17 Uhr

Der Zyklus "Softwareentwicklung"

Vier Grundbausteine

- SE1/SE2: Zustands- und Objektorientierte Programmierung
- SE3: Programmieren mit Funktionen / Relationen
- ein Praktikum (IP11)
 - Softwaretechnik, (Logikprogrammierung,) Bildverarbeitung, Internetwerkzeuge, Roboterprogrammierung, IT-Sicherheit . . .

Lernziele

- Kennenlernen von grundlegenden Konzepten, Methoden und Werkzeugen der Softwareentwicklung in einem alternativen Programmierparadigma
- Erwerben der Fähigkeit zur kritischen Auswahl von Methoden und Werkzeugen
- Erwerben der Fähigkeit zum selbständigen und systematischen Lösen programmiertechnischer Aufgaben
- Vertiefung von formalen Konzepten der Informatik anhand von konkreten Programmierproblemen

Lernformen

- Die Vorlesung:

Konzepte, Begriffe, Zusammenhänge, Beispiele
- Das Selbststudium:

Vertiefung, Literaturstudium, Klärung technischer Details, praktische Programmierübungen, Lerngruppenarbeit
- Die Übung:

Hausaufgaben, Präsenzübungen, Klären von Verständnisproblemen

- Die Rückkopplung:
Anregungen, Kritik

Die Übungen

- Erste Übung: 25./26./27.10.2016
- Ausgabe der Übungsaufgaben im NatsWiki
Department Informatik → Einrichtungen → Arbeitsbereiche → NATS → Courses:
SE3 Logikprogrammierung
- Bearbeitung der Aufgaben überwiegend am Rechner (unbetreut)
- Diskussion der Lösungen in der Übungsgruppe
- Abgabe der Lösungen
 - per Mail bei der/dem Übungsgruppenleiter(in)
 - mit Angabe derjenigen Aufgaben, für die die Bereitschaft zum Vortragen der Lösung besteht
- Bearbeitung in Gruppen (max. 3 Studierende) ist möglich und erwünscht

Die Software

- SWI-Prolog
 - Open-Source-Projekt (Universität Amsterdam)
 - Download: www.swi-prolog.org
 - integriertes Objektsystem
 - ausreichend für die Lehrveranstaltung SE III
- Sicstus-Prolog (Swedish Institute of CS, Kista)
 - semi-professionelles System
 - Studentenlizenzen über Michael König (RZ)
 - zahlreiche spezielle Erweiterungen (Coroutinen, Constraints, Constraint-Handling Rules, ...)
 - empfehlenswert für Prolog-Interessenten mit Ambitionen

Literatur

- Clocksin, W. F. und Mellish, C. S. (2003) Programming in Prolog. 5th edition, Springer-Verlag, Berlin.
- Sterling, L. und Shapiro, E. (1994) The Art of Prolog. Advanced Programming Techniques. 2nd edition, MIT-Press, Cambridge MA.

- Clocksin, W. F. (1997) Clause and Effect. Prolog Programming for the Working Programmer. Springer-Verlag, Berlin.
- Bratko, I. (2012) Prolog - Programming for Artificial Intelligence. 4th edition, Addison-Wesley/Pearson Education, Harlow.

Die hier angegebenen Literaturstellen sind nur eine kleine Auswahl aus der großen Menge an guten Lehrbüchern zur Logikprogrammierung. Recherchieren Sie selbst in der Bibliothek unter solchen Stichwörtern wie "Prolog", "Logikprogrammierung", "logic programming" usw. Sie werden überrascht sein von der Fülle des Angebots.

Vergleichen Sie einige dieser Bücher. Sie wurden mit sehr unterschiedlichen Erwartungen an die individuellen Voraussetzungen ihrer Leser geschrieben. Einige heben die logischen Grundlagen stärker hervor, andere betonen die Anwendungen. Es gibt Unterschiede im Stil und im didaktischen Geschick.

Suchen Sie sich diejenigen Quellen heraus, die Ihren individuellen Vorstellungen und Vorlieben am nächsten kommen. Auch die zielgerichtete Auswahl aus der Vielzahl von Angeboten will erlernt sein. Und wenn Sie dabei ein Buch gefunden haben, das Ihnen ausgesprochen gut gefallen hat, verraten Sie es mir bitte ...

2 Programmierparadigmen

Programmierparadigmen

```
function member(Element,Liste,Laenge);
  declare Element,Laenge,I integer;
  declare Liste(Laenge) array of integer;
  for I=1 to Laenge do;
    if Element=Liste(I) then return TRUE;
  end do;
  return FALSE;
end function;
```

```
(define member (element liste)
  (cond ((null liste) #f)
        ((equal (car liste) element) #t)
        (member element (cdr liste)) ) )
```

```
member(E,[E|_]).
member(E,[_|R]) :- member(E,R).
```

2.1 Verarbeitungsmodelle

Verarbeitungsmodelle

berechnungsuniverselle Verarbeitungsmodelle

- imperatives Modell
- funktionales Modell
- logik-basiertes Modell
- ...

eingeschränkte Verarbeitungsmodelle

- relationale Datenbanken
- Tabellenkalkulation
- ...

Imperative Verarbeitung

- Anweisungsfolgen verändern den internen Zustand des Automaten bzw. veranlassen Ein-/Ausgabeoperationen. → SE I

```
function member(Element,Liste,Laenge);
  declare Element,Laenge,I integer;
  declare Liste(Laenge) array of integer;
  for I=1 to Laenge do;
    if Element=Liste(I) then return TRUE;
  end do;
  return FALSE;
end function;
```

Das imperative Verarbeitungsmodell

Das imperative Verarbeitungsmodell basiert auf der Vorstellung von einer abstrakten Maschine, die in der Lage ist, Befehlssequenzen abzuarbeiten, mit denen der Inhalt eines adressierbaren Speichers verändert werden kann. Dieses Modell kommt damit dem weit verbreiteten Hardwarekonzept der VON NEUMANN-Maschine am nächsten. Wir beschreiben, wie der Speicher für Daten bereitgestellt und organisiert werden soll, sowie den Prozess, der die Daten und damit auch den Zustand der Berechnung verändert. Da Zustandsänderungen bei komplexen Abläufen leicht zu unüberschaubaren Abhängigkeiten führen, wurden für das imperative Modell eine Reihe von Ansätzen zur stärkeren *Disziplinierung* der Programmierung eingeführt. Ein erster Ansatz hierzu war die Idee der *strukturierten Programmierung* mit ihrer Vermeidung von Sprunganweisungen ("GOTO considered harmful"). Weitere Entwicklungsschritte führten über die *modulare Programmierung* zu den *abstrakten Datentypen*, wobei der Fortschritt vor allem in der Möglichkeit zum konsequenten Abstrahieren von Implementationsdetails bestand. Aktuelle Entwicklungen orientieren sich an der Idee der *objektorientierten Programmierung*, bei der die Zustandsänderungen an abstrakte Objekte gekoppelt sind. Zusätzlich werden mit der Einbeziehung von Vererbungsmechanismen hervorragende Mittel zur Strukturierung hochkomplexer Softwaresysteme bereitgestellt.

Funktionale Verarbeitung

- Durch Auswertung funktionaler Ausdrücke werden Werte (Berechnungsergebnisse) ermittelt. → SE III/F

```
(define member (Element Liste)
  (cond ((null Liste) #f)
        ((equal Element (car Liste)) #t)
        (member Element (cdr Liste)) ) )
```

Das funktionale Verarbeitungsmodell

Das funktionale Modell basiert auf der Definition von Funktionen und der Konstruktion von funktionalen Termen. Die zugrundeliegende Maschine muss daher über einen Mechanismus zur Auswertung funktionaler Ausdrücke verfügen. Die Programme sind meist sehr prägnant und kurz, was aber bei unerfahrenen Programmierern nicht unbedingt garantiert, dass die Programme auch verständlich sind. Ein Vorzug dieses Verarbeitungsmodells ist, dass die Semantik der Programme formal definiert werden kann. Das ermöglicht es im Gegensatz zur imperativen Programmierung Korrektheitsbeweise zu führen oder die Programme direkt aus formalen Spezifikationen zu entwickeln.

Logik-basierte Verarbeitung

- Durch logische Deduktion wird die Zugehörigkeit von Wertekombinationen zu einer Relation geprüft. Dabei werden Wertebindungen hergestellt, die sich als Berechnungsergebnisse interpretieren lassen.

→ SE III/L

```
member(Element, [Element|_]).  
member(Element, [_|Restliste]) :-  
    member(Element, Restliste).
```

Das logik-basierte Verarbeitungsmodell

Das logik-basierte Verarbeitungsmodell basiert auf der Vorstellung von einem Automaten, der bei einer gegebenen Menge von Fakten und einer gegebenen Menge von Schlussfolgerungsregeln aufgrund eines strikten Ableitungsmechanismus neue Fakten herleiten und damit Anfragen bearbeiten kann. Typisch ist der relationale Charakter des logik-basierten Verarbeitungsmodells, d.h. die Verarbeitungsrichtung ist nicht von vornherein vorgegeben (wie beim imperativen oder funktionalen Verarbeitungsmodell), sondern hängt von der jeweils zu bearbeitenden Berechnungsaufgabe ab. Als Berechnungsergebnis ergeben sich oftmals ganze Ergebnismengen, deren Elemente in vielen Fällen durch Aufzählung angegeben werden können. Wie das funktionale Modell auch zeichnet sich das logik-basierte Modell durch eine gut formalisierbare Semantik aus. Wegen der großen Nähe zum menschlichen logischen Schließen hat sich dieses Verarbeitungsmodell vor allem im Bereich der Expertensysteme bewährt, die in einem kleinen, überschaubaren Anwendungsbereich (Infektionskrankheiten, technische Diagnose, juristische Teilbereiche, Konfiguration komplexer technischer Systeme usw.) Spezialwissen modellieren und dieses damit einem großen Anwenderkreis zugänglich machen.

Verarbeitungsmodelle

Weitere berechnungsuniverselle Verarbeitungsmodelle:

objektorientiertes Modell

Botschaften werden an Objekte verschickt und verändern deren internen Zustand. Objekte können durch Abstraktion hierarchisch strukturiert werden. Zustandsänderungen werden imperativ oder funktional formuliert.

→ SE I

constraint-basiertes Modell

Die möglichen Werte von Variablen werden durch die Angabe von Bedingungen sukzessive eingeschränkt (Beschränkungserfüllung).

→ GWV

Das objektorientierte Verarbeitungsmodell

Das objektorientierte Verarbeitungsmodell versucht Objekte der Anwendungsdomäne mit den für sie definierten Aktionen im Rechner zu modellieren. Dank der Tatsache, dass dadurch virtuelle Objekte ähnlich manipulierbar werden, wie ihre physisch realen Vorbilder, entstehen Softwareprodukte, die sehr gut verständlich, wartbar und erweiterbar sind. Die erste und nach wie vor prototypische Anwendung war die Entwicklung der Fensterumgebungen grafischer Bedienoberflächen.

Das constraint-basierte Verarbeitungsmodell

Das constraint-basierte Verarbeitungsmodell ist ebenfalls ein relationales Verarbeitungsmodell und hat daher weitgehende Gemeinsamkeiten mit dem logik-basierten Verarbeitungsmodell. Auch hier ist die Verarbeitungsrichtung nicht vorgegeben und kann je nach der zu bearbeitenden Berechnungsaufgabe variieren. Eindeutige Berechnungsergebnisse sind eher die Ausnahme als die Regel, statt dessen ergeben sich typischerweise Ergebnismengen, die durch Bedingungen beschrieben werden können. Wesentliche Unterschiede zum logik-basierten Modell bestehen jedoch hinsichtlich der Steuerung des Berechnungsablaufs. Typische Anwendungen sind Planungs- und Schedulingprobleme z.B. für Stundenpläne oder Sendefrequenztabelle.

Das mystische Verarbeitungsmodell

Das mystische Verarbeitungsmodell ist ein vielfach anzutreffender Zwischenzustand beim Erlernen eines neuen Verarbeitungsmodells, der insbesondere beim unsystematischen Herangehen durchaus typisch ist. Dieses Verarbeitungsmodell sollten Sie möglichst schnell zu überwinden versuchen.

Auch bei diesem Modell sehen wir den Rechner als "black box" mit regeltem E/A-Verhalten, aber im Unterschied zu den anderen Modellen unterstellen wir wunderbare, unerklärliche und quasi magische Fähigkeiten. Unbewusst personifizieren wir den Rechner und kommunizieren mit ihm auch emotional, wie mit einem Lebewesen. Wir verhalten uns so, als hätte der Rechner eigene Absichten und Ziele und wir sind uns vielfach nicht sicher, ob wir ihm auch trauen können. Dies kommt insbesondere bei unerwartetem Verhalten des Rechners zum Ausdruck, wenn er sich aus unserer Sicht fehlerhaft, unkooperativ, widerspenstig oder gar boshaft verhält. Die eine Ursache für ein nicht beherrschbares Verarbeitungsmodell ist schlechte, schlecht dokumentierte oder gar fehlerhafte Software, die alle, die damit arbeiten müssen, zur Verzweiflung treibt. Die Verwendung solcher Erzeugnisse ist riskant und führt zu neuen unsicheren Programmen.

Die zweite, nicht weniger gefährliche Ursache für die Annahme eines mystischen Verarbeitungsmodells ist ein zu großes Vertrauen in die Fähigkeiten der Maschine. Wir lassen uns

durch die Verarbeitungsleistung, die Präzision der Verarbeitung und die Eleganz formaler Notationen beeindrucken, die uns *Scheinobjektivität* suggerieren. Aber auch das ausgefeiltste formale System schützt uns nicht davor, darin großen Unsinn auszudrücken. Wir dürfen uns nicht von der maschinellen Präzision blenden lassen und niemals Resultate ungeprüft übernehmen. Das verpflichtet uns aber auch dazu, Software so zu entwerfen, dass sie validiert und die erzeugten Ergebnisse überprüft werden können.



© 1996 Randy Glasbergen. E-mail: randy@glasbergen.com www.glasbergen.com

Any sufficiently advanced technology is undistinguishable from magic. Arthur C. Clarke

Verarbeitungsmodelle

Eingeschränkte Verarbeitungsmodelle

- Endliche Automaten, Reguläre Grammatiken, Reguläre Ausdrücke

→ FGI I

- Tabellenkalkulationsprogramme

- Markup-Sprachen (HTML, XML, ...)

- Relationale Datenbanken

→ GDB

weitere Verarbeitungsmodelle

→ STOYAN (1988) Programmiermethoden der Künstlichen Intelligenz

2.2 Abstraktion

Abstraktion

- Alle Verarbeitungsmodelle sind Abstraktionen von der zugrundeliegenden Hardware → "Abstrakte Maschine"

elementare Operationen des Prozessors → (komplexe) Operationen der abstrakten Maschine

(virtueller) Speicher → abstraktes Speichermodell

- Befreiung der Programmierung vom VON NEUMANN-Stil

Abstraktionsniveau

- Verarbeitungsmodelle abstrahieren unterschiedlich stark von der zugrundeliegenden Hardwarestruktur

	imperativ	funktional	logik-basiert
elementare Datentypen	–	–	(–)
Anweisungslogik	+	++	+++
Steuerstrukturen	+	++	+++
Speichermodell	–	++	+++
Verarbeitungsrichtung	–	–	++

Während sich die imperative Programmierung in verschiedener Hinsicht sehr stark an den Gegebenheiten der Hardwarearchitektur eines VON NEUMANN-Rechners orientiert, bieten die alternativen Paradigmen der funktionalen und logik-basierten Programmierung ein weitaus höheres Abstraktionsniveau. So entspricht etwa eine Variable in der imperativen Programmierung ziemlich genau einem (konkreten) Speicherplatz im Hardwarespeicher, dessen Inhalt beliebig verändert werden kann. Im funktionalen Modell spielt diese Sicht nur noch eine untergeordnete Rolle. Variable dienen hauptsächlich der Wertübergabe an Funktionen. In einem rein logik-basierten Modell hingegen ist eine solche Wertmanipulation völlig ausgeschlossen. Variable erhalten ihren Wert durch einen speziellen Mechanismus (Unifikation) und diese Bindung kann ebenfalls nur durch einen speziellen Mechanismus (Backtracking) wieder aufgelöst werden. Scheinen die Maschinenbefehle einer konkreten Prozessorarchitektur in der imperativen Programmierung noch sehr stark durch (z.B. Wertzuweisung oder bedingte Anweisungen), ist dieser Zusammenhang in der funktionalen Programmierung bereits weitaus schwächer. In der Logikprogrammierung müssen ähnliche Konstrukte erst nachträglich wieder eingebaut werden, weil die abstrakte Maschine sie gar nicht mehr direkt zur Verfügung stellt. Eine große Ähnlichkeit besteht hingegen zwischen allen drei Paradigmen hinsichtlich der elementaren Datentypen, deren Grundbestand sich in allen Fällen sehr stark am Angebot der zugrundeliegenden Hardware orientiert (Integer, Real, Character).

Abstraktion

Abstraktion ...

- ... ist das Hervorheben bestimmter Merkmale eines Objekts als relevant
- ... ist das Vernachlässigen anderer Merkmale eines Objekts als irrelevant

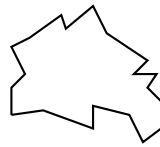
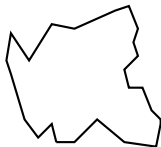
Abstraktion ...

- ... ist ein wesentliches Hilfsmittel zur Reduktion von Komplexität
- ... ist eine zentrale Kulturleistung des Menschen
- ... ist aufgabenbezogen und subjektiv
- ... kann daher eine wesentliche Quelle für Missverständnisse sein

Abstraktion

Abstraktion reduziert Komplexität

Beispiel: Entfernung zwischen zwei Städten

**Abstraktion als Kulturleistung**

Beispiel: Handel

Ware gegen Ware

- Ware gegen Materialwert
- Ware gegen Nominalwert
- Ware gegen Verrechnungseinheit

Abstraktion

Abstraktion als Quelle von Missverständnissen

Beispiel: Maßeinheiten

°C oder °F ?

km oder Meilen?

km/h oder mph?

Welche Meilen?

Abstraktion in der Softwareentwicklung

- | | |
|---------------------------------------|-------------------------|
| Abstraktion vom Einzelfall | → Wiederverwendbarkeit |
| Abstraktion von irrelevanten Aspekten | → Anwendungsbezug |
| Abstraktion von der Hardware | → Komplexitätsreduktion |

Abstraktion durch Generalisierung

- Suche nach regelhaften Zusammenhängen in den Daten
- Wiederverwendbarkeit für große Klassen von Beispielfällen

funktionale Abstraktion

```
x = 1, f(x) = 1
x = 2, f(x) = 4
x = 3, f(x) = 9
```

```
(define square(x) (* x x))
```

relationale Abstraktion

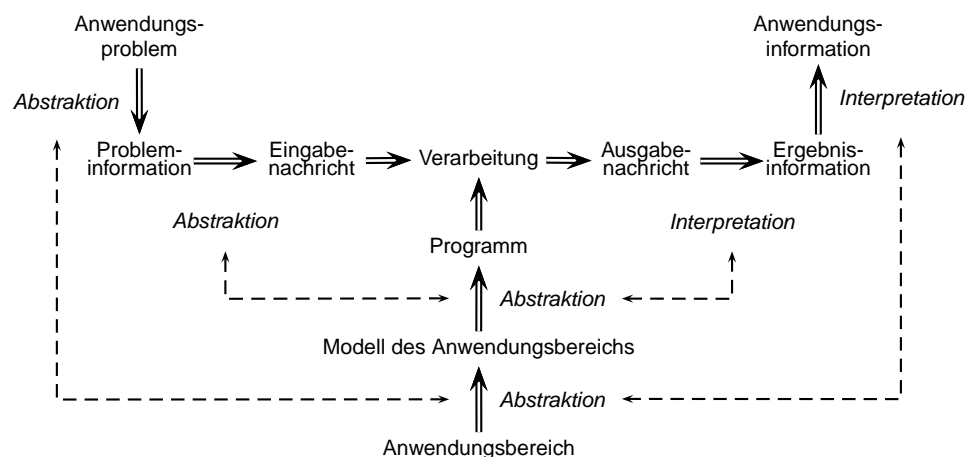
```
vogel(amsel)    kann_fliegen(amsel)
vogel(star)     kann_fliegen(star)
vogel(meise)    kann_fliegen(meise)
```

```
kann_fliegen(X) :- vogel(X).
```

Abstraktion durch Vergrößerung

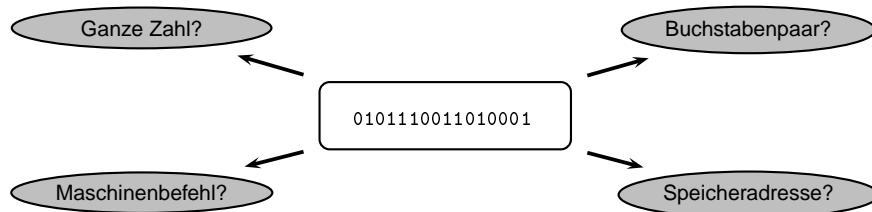
- Weglassen von Sonder- und Ausnahmefällen
- Weglassen von Maßeinheiten
- Weglassen von Ausgabeinformation
- Weglassen von Kommentaren
- ... in Abhängigkeit von der Relevanz für eine Aufgabenstellung
- aber: Relevanzbewertungen sind hochgradig subjektiv!

Abstraktion durch Vergrößerung



Abstraktion durch Verbergen

- Abstraktion als Voraussetzung für universell einsetzbare Hardware:
Beispiel: Binärcodierung



Abstraktion durch Verbergen

- Abstraktion von Möglichkeiten der Hardware
→ Voraussetzung für den verlässlichen Umgang mit ihr
- Beispiel: goto considered harmful
 - DIJKSTRA 1968
Jedes Programm, das goto-Befehle enthält, kann in ein goto-freies Programm umgewandelt werden. → SE I

- Beispiel: Wertmanipulation als Fehlerquelle
 - Berechnungsergebnis ist vom Zustand eines Programms abhängig
 - Zustand ist zum Entwicklungszeitpunkt nicht sichtbar
 - Bedeutung eines Programmes ist von (lokal) nicht sichtbaren Eigenschaften abhängig: Seiteneffekte

Abstraktion durch Verbergen

- Umgang mit Seiteneffekten: Unterschiedliche Lösungsansätze
- Objektorientierung:
 - Zustandsänderungen nur über wohldefinierte Schnittstellen (Methoden)
 - Einschränkung auf lokale Gültigkeitsbereiche (Klassendefinition)
- funktionale Programmierung:
 - Zurückdrängen von Wertmanipulationsmöglichkeiten
 - aber nichtlokale Gültigkeitsbereiche

- Logikprogrammierung:
 - Wertemanipulation nur durch das Verarbeitungsmodell
 - extrem lokale Gültigkeitsbereiche

Abstraktion

- ... ist ein wesentlicher Bestandteil der Softwareentwicklung
- ... erlaubt kompakte und überschaubare Problemlösungen
- ... kann leicht zur Fehlerquelle werden
- ... erfordert *Sorgfalt* bei Entwicklung und Dokumentation
 - insbesondere für die als irrelevant betrachteten Aspekte!
 - erfordert das "Hineinversetzen" in den Adressaten!
 - Dokumentation der Entwicklerintention in vielfältigen Formen: Programmkommentare, Handbücher, Gebrauchsanweisungen, ...

Abstraktion ...

... ist keine Fehlentwicklung, sondern der zentrale Kern der Informatik. Sie ist Voraussetzung

- für die Entwicklung von Anwendungsprogrammen (Auskunftssysteme, Lohnbuchhaltung usw.), da für die Modellierung einer Anwendungsdomäne stets eine Vielzahl von als irrelevant erachteten Details ausgeklammert werden muss.
- für die Entwicklung von universell verwendbaren Softwarewerkzeugen. Produkte, wie Betriebs-, Datenbank-, Textverarbeitungs- aber auch Programmiersysteme, konnten nur entstehen, indem in einem langwierigen Entwicklungs- und Auswahlprozess eine weitgehende Übereinkunft über die relevanten (d.h. verbindenden) Aspekte des jeweiligen Anwendungsbereichs (Betriebsmittelverwaltung, Datenhaltung usw.) hergestellt wurde, während individuelle Besonderheiten spezieller Einsatzfälle unberücksichtigt blieben und der Nutzer des Systems von den Details der gerätetechnischen Umsetzung abgeschirmt wird.
- für die Entwicklung komplexer Systemlösungen. Erst wenn eine Problemlösung auf einem hinreichend abstrakten Niveau beschrieben ist, wird sie überschaubar und verständlich.
- für die arbeitsteilige Entwicklung von Software, da erst durch die Zerlegung in beherrschbare Teilaufgaben, komplexe Informationsverarbeitungsaufgaben umgesetzt werden können. Auch hier spielen Prozesse der Abstraktion eine entscheidende Rolle.

Abstraktion lässt sich in der Informatik also nicht vermeiden, vielmehr muss der souveräne Umgang mit ihr gelernt, geübt und organisiert werden. Abstraktion ist in der Informatik kein notwendiges Übel, sondern ein wichtiges methodisches Handwerkszeug. Es ist daher auch kein Zufall, dass die Informatik mit der Mathematik als der abstraktesten der Wissenschaften sehr viel gemeinsam hat. Rechnen, als wichtige kulturelle Leistung des Menschen, wurde auch erst praktikabel, als man gelernt hatte, von allen anwendungsbezogenen Gesichtspunkten zu

abstrahieren (es ist egal, ob ich Hühner zähle, oder Schritte) und sich ganz auf den relevanten Kern, die quantitativen Zusammenhänge zu konzentrieren.

Der problematische Aspekt der Abstraktion ist dabei immer die Frage *wovon* im Einzelfall abstrahiert wird, d.h. welche Gesichtspunkte — aus welchen Gründen auch immer — *nicht* berücksichtigt wurden, denn hier entstehen die fatalen Missverständnisse beim Einsatz der Informationstechnologie: Wofür ist das Programm *nicht* geeignet?, „Welche Spezial- und Extremfälle werden *nicht* korrekt behandelt?“ usw. Das bedeutet aber, dass sich der Informatiker nicht nur mit der Frage auseinandersetzen muss, „Was mache ich da?“ (Damit befasst er sich ja ohnehin!), sondern in weitaus stärkerem Maße mit Fragen wie, „Was mache ich gerade *nicht*?“, „Was erachte ich als irrelevant?“, ja vor allem „Was habe ich womöglich einfach übersehen?“. Spätestens bei der letzte Frage sollte auffallen, dass die Antwort darauf nur dann gefunden werden kann, wenn man nicht nur die eigene, sondern vor allem die Perspektive der anderen beteiligten Personen (Kollegen, Kunden, Nutzer) angemessen berücksichtigt. Wichtig ist also immer die Frage „Wovon abstrahiere ich?“, „Warum abstrahiere ich davon?“ und nicht zuletzt: „Wie kommuniziere ich dies meinen Partnern?“ All diese Überlegungen müssen geeignet dokumentiert und kommuniziert werden, denn sie finden sich im Endprodukt (dem Programmtext) ja gerade *nicht*!

Relationale Abstraktion

- Logikprogrammierung abstrahiert auch vom Algorithmusbegriff
- Ein Algorithmus beschreibt einen (parametrisierbaren) dynamischen Ablauf
 - Eingabedaten
 - prozeduraler Ablauf
 - Ausgabedaten
- Logikprogramme beschreiben den generellen Zusammenhang zwischen zwei oder mehr Datenstrukturen und lassen prinzipiell beliebige Verarbeitungsrichtungen zu

Relationale Abstraktion

- Kontrast: funktionale Programmierung ist immer richtungsabhängig

```
(define member (Element Liste)
  (cond ((null Liste) #f)
        ((equal Element (car Liste)) #t)
        (member Element (cdr Liste)) ) )
```

- einzig mögliche Abbildung: $\text{Element} \times \text{Liste} \mapsto \{\#t, \#f\}$

```
(member 'a '()) ==> #f
(member 'b '(a b c)) ==> #t
(member 'd '(a b c)) ==> #f
```

Relationale Abstraktion

- Ein Logikprogramm kann üblicherweise unterschiedliche algorithmische Abläufe realisieren:

```
member(Element, [Element|_]).
member(Element, [_|Restliste]) :-
    member(Element, Restliste).
```

- Aufrufvariante 1: $\text{Element} \times \text{Liste} \mapsto \{\text{true}, \text{false}\}$

```
?- member(a, []).
false.
?- member(b, [a,b,c]).
true.
?- member(d, [a,b,c]).
false.
```

Relationale Abstraktion

- Aufrufvariante 2: $\text{Liste} \mapsto \text{Element}$

```
?- member(E, []).
false.
?- member(E, [a,b,c]).
E = a ;
E = b ;
E = c .
```

Relationale Abstraktion

- Aufrufvariante 3: $\text{Element} \mapsto \text{Liste}$

```
?- member(a, L).
L = [a|_] ;
L = [_ , a|_] ;
L = [_ , _ , a|_] ;
L = [_ , _ , _ , a|_] .
```

- Aufrufvariante 4: $\emptyset \mapsto \text{Element} \times \text{Liste}$

Relationale Abstraktion

- Logikprogramme implementieren nicht konkrete Algorithmen sondern generalisierte Algorithmentschemata
- Logikprogramme kodieren nicht das Lösungsverfahren sondern das zur Problemlösung erforderliche (statische) Wissen über die Zusammenhänge zwischen den formalen Parametern

Sagen, wie es ist.

KAY V. LUCK

- Logikprogramme sind im allgemeinen Fall *richtungsunabhängig*.

Relationale Abstraktion

- die algorithmischen Details müssen nicht im Programm ausgedrückt werden sondern sind bereits Bestandteil des Verarbeitungsmodells

algorithm = logic + control

(BOB KOWALSKI)

Programm

abstrakte Maschine

- die Verarbeitungsrichtung wird erst beim Aufruf festgelegt

Hybride Modelle

Verarbeitungsmodelle können miteinander kombiniert werden:

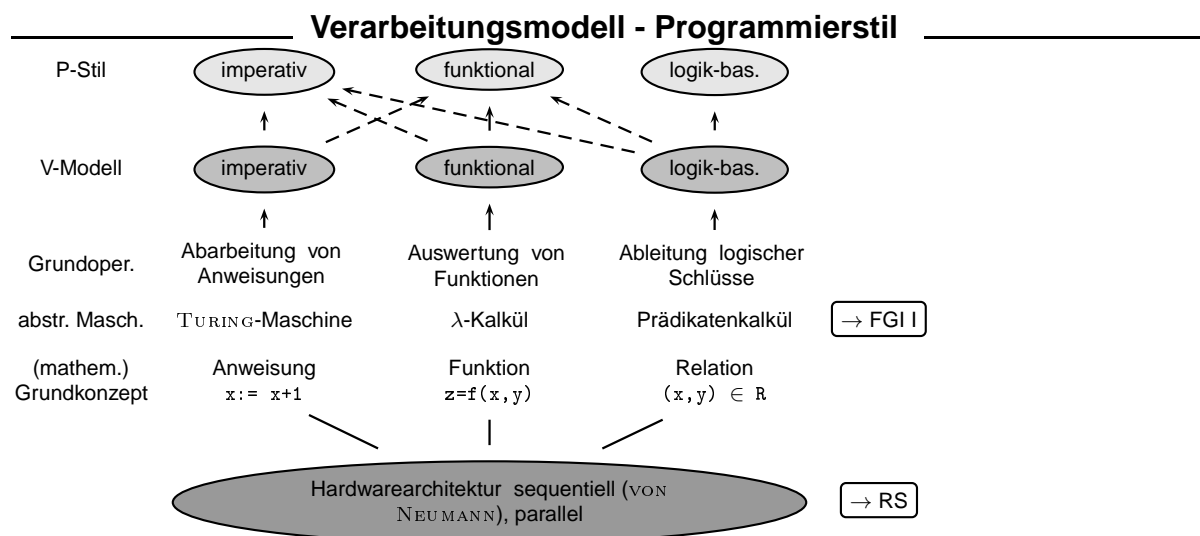
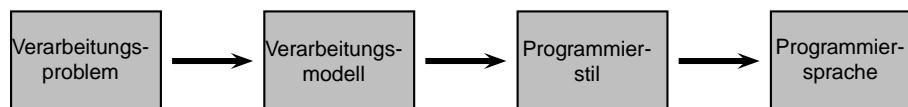
- imperativ + Elemente von funktional (C, Java)
- imperativ + objektorientiert (C++)
- funktional + objektorientiert (Common Lisp)
- logik-basiert + Elemente von funktional (Prolog)
- funktional + relational
- logik-basiert + constraint-basiert (CLP)
- ...

2.3 Programmierstile

Programmierstil

Ein Verarbeitungsmodell begründet einen bestimmten Programmierstil, ist aber nicht auf diesen eingeschränkt.

- Jedes Verarbeitungsproblem legt ein geeignetes Verarbeitungsmodell nahe.
- Verarbeitungsmodelle sind Denkwerkzeuge.
- Ein bestimmtes Verarbeitungsmodell wird durch die einzelnen Programmiersprachen mehr oder weniger gut unterstützt.
- Jede Programmiersprache fördert einen bestimmten Programmierstil besonders.



Welches ist der beste Programmierstil?

Diese Frage lässt sich nicht allgemeingültig beantworten. Je nach Anwendungsproblem bietet sich der eine oder der andere Programmierstil an. Bei der Programmentwicklung ist es die wichtigste Aufgabe, zuerst das Anwendungsproblem zu analysieren und die Grundoperationen festzulegen, mit denen eine Verarbeitungsaufgabe am besten beschrieben werden kann. Dadurch ergibt sich der problemadäquate Programmierstil.

Wenn keine der zur Verfügung stehenden Programmiersprachen diese Grundoperationen bietet, dann ist es oft am günstigsten, sich selbst eine Arbeitsumgebung zu schaffen, in der man den gewünschten Programmierstil anwenden kann, d.h. sich eine *virtuelle Maschine* zu implementieren. Der Fortschritt der Informatik als Disziplin beruht ganz wesentlich auf diesem Herangehen!

Nur sehr selten kommt ein Verarbeitungsmodell in seiner “reinen” Form zum Einsatz. Imperative und logik-basierte Programmiersprachen greifen auf funktionale Auswertungsumgebungen zurück und in funktionalen Sprachen finden sich imperative Elemente. So entstehen *hybride* Verarbeitungsmodelle, wobei die Existenz hybrider Modelle in erster Linie eine Folge des hybriden Charakters realer Problemstellungen ist.

2.4 Logikprogrammierung

Logikprogrammierung

- hohes Abstraktionsniveau
- sehr mächtige Werkzeuge der Informatik als Basiskonstrukte
 - Datenbanken, nichtdeterministische Suche, Mustervergleich, ...
- unmittelbarer Bezug zu zentralen Konzepten der theoretischen Informatik und Mathematik
 - Relation, Funktion, Unifikation, Substitution, Subsumtion, Verband, Unterspezifikation, Transducer, Grammatik, Symmetrie, Transitivität, Ableitung, ...
- kompakte Problemlösungen
- direkter Bezug zur Problemstellung
- schnelle Realisierung von funktionsfähigen Prototypen

Logikprogrammierung

- berechnungsuniverselle Programmiersprache durch Kombination weniger, aber sehr mächtiger Basiskonzepte
 - Unifikation von rekursiven (Daten-)Strukturen
 - Termersetzung und Variablensubstitution
 - nichtdeterministische Suche
- Syntax und Semantik der Sprache können leicht modifiziert werden
 - Anpassung an spezielle Nutzungsbedingungen
 - experimentelles Prototyping für neue Verarbeitungsmodelle

3 Relationale Datenbanken

Relationale Datenbanken

Name	Vorname	Gehalt
Meier	Kerstin	3400
Schulze	Hans	5300
Müller	Annegret	6300
Heimann	Manfred	2500

3.1 Relationen

Relationen

Relation in M

Teilmenge des Kreuzproduktes der Menge M mit sich selbst.

$$R^{(2)} \subseteq M \times M = M^2$$

Kreuzprodukt (Cartesisches Produkt) zweier Mengen A und B

Menge aller geordneten Paare (2-Tupel), die sich aus den Elementen von A und B bilden lassen:

$$A \times B = \{(x, y) | x \in A, y \in B\}$$

Relationen

Domäne einer Relation

Menge M , über der die Relation definiert ist.

Extensionale Spezifikation einer Relation

Aufzählung ihrer Elemente

Beispiele für Relationen

$$A = \{o, O, \bigcirc\}$$

$$A \times A = \{ (o, o), (O, o), (\bigcirc, o), \\ (o, O), (O, O), (\bigcirc, O), \\ (o, \bigcirc), (O, \bigcirc), (\bigcirc, \bigcirc) \}$$

$$\text{"ist gleich groß" (in } A) = \{(o, o), (O, O), (\bigcirc, \bigcirc)\} \subseteq A^2$$

$$\text{"ist kleiner als" (in } A) = \{(o, O), (o, \bigcirc), (O, \bigcirc)\} \subseteq A^2$$

$$\begin{aligned} \text{"ist größer oder gleich" (in } A) = \{ & (o, o), (O, o), (\bigcirc, o), \\ & (o, O), (O, O), (\bigcirc, O) \} \subseteq A^2 \end{aligned}$$

Der Begriff der Relation wird in der Mathematik üblicherweise etwas enger gefasst, als in der Informatik. Wir müssen daher erst einige Erweiterungen vornehmen, ehe wir dann auf die re-chentechnische Umsetzung des Relationenbegriffs eingehen können. Zudem existieren einige

terminologische Abweichungen, die in diesem Zusammenhang klar werden sollten.

Modellierung mit Relationen

Relationen

sind Beziehungen zwischen Objekten, Ereignissen und abstrakten Ideen und Begriffen.

- Relation ist ein formales Konstrukt
- Problem bei der Anwendung auf "reale" Fragestellungen
 - Objekte und Begriffe treten niemals isoliert auf.
 - vollständige Erfassung (Verstehen) eines Begriffs erst durch Berücksichtigung der Gesamtheit seiner Relationen zu anderen Objekten und Begriffen möglich.
- Approximation nötig:
 - *Weitgehende* Erfassung . . .
 - Berücksichtigung der *wesentlichen* Relationen . . .
 - → Abstraktion durch Vergrößerung

Modellierung mit Relationen

- Relationen im strengen Sinne: zweistellige Relationen
 - A "ist kleiner als" B
 - A "passierte zeitlich nach" B
 - A "ist Voraussetzung für" B
- Allgemeiner Fall: n-stellige Relationen ($n \geq 1$)
 - A "liegt zwischen" B "und" C
 - A "transportiert" B "von" C "nach" D
- Spezialfall: einstellige Relationen (Eigenschaften, Prädikate)
 - A "ist groß"
 - A "ist blau"
 - A "ist ein Haus"

Modellierung mit Relationen

Beispiel: zweistellige Relation

$P = \{\text{Anna, Susi, Elli}\}$ [5mm]

$$\begin{aligned} \text{"ist Mutter von" (in } P) &= \{(Susi, Anna), \\ &\quad (Elli, Susi)\} \\ &\subseteq P^2 \end{aligned}$$

Beschränkung auf *eine* Domäne ist zu restriktiv für viele praktische Anwendungen

Erweiterungen

- Erster Schritt: Verallgemeinerung auf unterschiedliche Domänen

$$R^{(2)} \subseteq M \times N, \quad M \neq N$$

$$B = \{\circ, \square, \triangle\}, \quad C = \{\bullet, \blacksquare, \blacktriangle\}$$

$$\text{"ist kongruent"} \text{ (über } B \text{ und } C) = \{(\circ, \bullet), (\square, \blacksquare), (\triangle, \blacktriangle)\} \subseteq B \times C$$

$$P = \{\text{Anna, Susi, Elli}\}, \quad N = \{0, 1, 2, \dots\}$$

$$\begin{aligned} \text{"hat_alter"} \text{ (über } P \text{ und } N) &= \{(\text{Anna}, 12), (\text{Susi}, 35), (\text{Elli}, 63)\} \\ &\subseteq P \times N \end{aligned}$$

Erweiterungen

- Zweiter Schritt: Verallgemeinerung auf n-stellige Relationen

$$R^{(n)} \subseteq \underbrace{A \times \dots \times Z}_{n \text{ Faktoren}}$$

$$P = \{\text{Anna, Susi, Elli}\}, \quad N = \{0, 1, 2, \dots\}$$

$$F = \{\text{ledig, verheiratet, geschieden, verwitwet}\}$$

$$\begin{aligned} \text{"hat_alter_und_familienstand"} \text{ (über } P, N \text{ und } F) \\ &= \{(\text{Anna}, 12, \text{ledig}), \\ &\quad (\text{Susi}, 35, \text{geschieden}), \\ &\quad (\text{Elli}, 63, \text{verheiratet})\} \\ &\subseteq P \times N \times F \end{aligned}$$

3.2 Fakten

Fakten

Relationale Datenbank

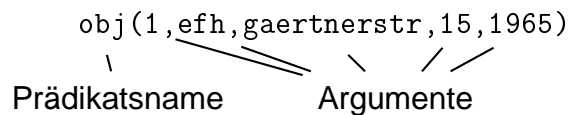
Aufzählung von Faktenwissen über einen zu modellierenden Gegenstandsbereich (extensionale Definition).

id	type	str	no	jahr
1	EFH	Gaertnerstr	15	1965
2	EFH	Bahnhofsstr	27	1943
3	EFH	Bahnhofsstr	29	1955
4	MFH	Bahnhofsstr	28	1991
5	Bahnhof	Bahnhofsstr	30	1901
6	Kaufhaus	Bahnhofsstr	26	1997
7	EFH	Gaertnerstr	17	1982

Fakten

Fakten

elementare Klauseln, die genau eine *Grundstruktur* enthalten

Struktur**Grundstruktur**

Struktur, deren Argumente atomar sind.

Fakten

- Syntax:

Klausel	::=	(Fakt ...) ''
Fakt	::=	Struktur
Struktur	::=	Name ['(' Term {',' Term} ')']
Term	::=	Konstante ...
Konstante	::=	Zahl Name gequoteter_Name
Name	::=	Kleinbuchstabe {alphanumerisches_Symbol}
- Semantik: Jedes Faktum spezifiziert ein Element einer Relation mit dem angegebenen Prädikatsnamen und der Stelligkeit der Struktur.
 - Stelligkeit der Struktur ist Bestandteil der Prädikatsdefinition.
 - Strukturen unterschiedlicher Stelligkeit definieren unterschiedliche Relationen.

Fakten

- Pragmatik: Eindeutige Prädikatsidentifizierung erfordert die Angabe der Stelligkeit:

```
obj/5, liebt/2, teil_von/2 ...
singt/1, singt/2 ...
```

Prädikate

Prädikat (Relation, Prozedur)

Menge von Fakten mit gleichem Namen und gleicher Stelligkeit

```
obj(1,efh,gaertnerstr,15,1965).
obj(2,efh,bahnhofsstr,27,1943).
obj(3,efh,bahnhofsstr,29,1955).
obj(4,mfh,bahnhofsstr,28,1991).
obj(5,bahnhof,bahnhofsstr,30,1901).
obj(6,kaufhaus,bahnhofsstr,26,1997).
obj(7,efh,gaertnerstr,17,1982).
```

Prädikate

Prädikatsschema

Interpretationshilfe für die Argumentstellen eines Prädikats

% obj (Objekt-Nr,Objekttyp,Strasse,Hausnummer,Baujahr)

Dient nur der zwischenmenschlichen Kommunikation

Datenbasis

Menge von Prädikatsdefinitionen.

Wird aus einem File eingelesen (`consult/1`).

Der Relationenbegriff der Datenbankwelt ist gegenüber dem der Logikprogrammierung noch etwas allgemeiner gefasst. Wie auch in der Mathematik wird von der Reihenfolge der Elemente abstrahiert (Die Relation ist eine Menge!) zusätzlich aber auch noch von der Anordnung der Attribute der Relation. Der Zugriff zu den Attributen erfolgt dann über einen Namen für die betreffende Tabellenspalte.

Demgegenüber sind Prädikatsdefinitionen in der Logikprogrammierung geordnet und die Identifizierung der Attribute (Argumente) ist nur über ihre *Position* möglich. Die Wahl der Positionen ist beliebig, muss aber für alle Elemente der Relation gleich sein. Die vereinbarte Zuordnung von Attributen zu Argumentpositionen veranschaulicht das *Prädikatsschema*. Dieses entspricht der Signatur einer Prozedur in der Objektorientierten Programmierung, hat aber anders als diese nur eine informelle, aber keinerlei formale Bedeutung. Insbesondere trifft dies auf alle dort angegebenen Typrestriktionen zu.

Zur Begriffswelt der Mathematik ergeben sich die folgenden terminologischen Entsprechungen:

Mathematik	Logikprogrammierung
Relation	zweistelliges Prädikat
n-stellige Relation	Prädikat
Prädikat	einstelliges Prädikat
Element einer Relation	Fakt

3.3 Anfragen

Anfragen

Anfrage (Ziel, Goal)

elementare Klausel (am Systemprompt eingegeben)

```
?- obj(1,efh,gaertnerstr,15,1965).
```

- Syntax:

Klausel	::=	(Fakt Ziel ...)	''
Ziel	::=	elementares_Ziel	...
elementares_Ziel	::=	Struktur	
- Semantik: Prüfen auf Konsistenz mit den Fakten der Datenbasis
 - Ableitbarkeit, Beweisbarkeit bezüglich einer Axiomenmenge (Datenbasis)
 - abgeschlossene Welt (closed world assumption), vollständige Beschreibung

Semantik

Denotationelle Semantik

Implementationsunabhängige Beschreibung der Bedeutung, wobei das betreffende sprachliche Konstrukt als statisches Objekt betrachtet wird.

Operationale Semantik

Beschreibung der dynamischen Aspekte eines Programms, des durch das Programm erzeugten Verhaltens.

Anfragen

Denotationelle Semantik eines Ziels

Folgt das Ziel (eine logische Aussage) aus der Datenbasis (Axiomenmenge)?

```
?- obj(1,efh,gaertnerstr,15,1965).
true.
?- obj(1,efh,gaertnerstr,15,1966).
false.
```

Negatives Resultat bezieht sich nur auf die gegebene Datenbasis (abgeschlossene Welt)

Anfragen

Operationale Semantik eines Ziels:

Suche nach einem *unifizierbaren* Axiom in der Datenbasis.

- Suche: Systematisches Durchmustern von Entscheidungsalternativen.
- Suchstrategie: Festlegungen über die Reihenfolge bei der Betrachtung der Alternativen.
- Prolog: Durchsuchen der Datenbasis von oben nach unten.

- Negatives Resultat: Erfolgreiche Suche

Unifikation (I)

Unifikation

Überprüft die Verträglichkeit zweier Strukturen

- Semantik: Unifikation von Grundstrukturen ist ein Identitätstest.
- Prolog: Vergleich des Ziels mit den Fakten der Datenbasis
- Vergleichskriterien:
 - identischer Prädikatsname
 - gleiche Stelligkeit
 - paarweise identische Argumente

Unifikation ist die grundlegende Operation des logikorientierten Verarbeitungsmodells. Die hier angegebene Semantik stellt den einfachsten Spezialfall einer Unifikationsoperation dar, die nur zwei unterschiedliche Resultate kennt: Die Unifikation gelingt, bzw. sie scheitert. Dies entspricht in der Logik den beiden Wahrheitswerten: Eine Aussage ist wahr, bzw. sie ist falsch. Dieser primitive Unifikationsbegriff wird in den folgenden Abschnitten schrittweise auf den generellen Fall erweitert, wodurch letztendlich eine sehr leistungsfähige Operation entsteht. Durch die Einbettung der Unifikation in die Suche des Programmiersystems, wird der statische Ableitungsbegriff der Logik auf eine dynamische Berechnungsprozedur abgebildet. Die im jeweiligen Einzelfall durchlaufene Sequenz von Suchschritten kann in Form eines Ablaufprotokolls oder aber mit Hilfe eines Suchbaums veranschaulicht werden.

Anfrageabarbeitung

Beispieldatenbasis 1

```
% liebt(Wer,Wen-oder-was)
% Wer und Wen-oder-was sind Namen, so dass Wen-oder-was von
% Wer geliebt wird
liebt(hans,geld).
liebt(hans,susi).
liebt(susi,buch).
liebt(karl,buch).
```

Exkurs: Programmiermethodik

Kommentare

Jede Prädikatsdefinition sollte zumindest mit

1. dem Prädikatsschema und
2. *Zusicherungen* über die zulässigen Argumentbelegungen

kommentiert werden.

Anfrageabarbeitung

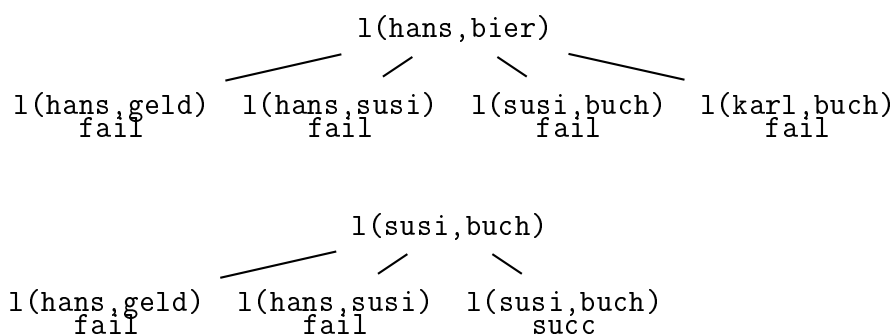
- Operationale Semantik: Veranschaulichung durch ein Ablaufprotokoll (Trace)

```
?- liebt(hans,bier).
    liebt(hans,geld).    fail
    liebt(hans,susi).    fail
    liebt(susi,buch).    fail
    liebt(karl,buch).    fail
false.
```

```
?- liebt(susi,buch).
    liebt(hans,geld).    fail
    liebt(hans,susi).    fail
    liebt(susi,buch).    succ
true.
```

Anfrageabarbeitung

- Operationale Semantik: Darstellung des Suchraumes als Baum
 - Mutterknoten: Anfrage,
 - Tochterknoten: Fakten aus der Datenbasis



Anfrageabarbeitung

Suchreihenfolge

In Prolog erfolgt die Suche in der Datenbasis in der Reihenfolge der Einträge (d.h. von oben nach unten).

Im Suchbaum entspricht der Wurzelknoten der am Systemprompt eingegebenen Anfrage. Seine Tochterknoten (d.h. die unmittelbar untergeordneten Knoten des Baumes) entsprechen den Eintragungen der Datenbasis. Eine Kante steht damit für einen Unifikationsversuch und an den Tochterknoten ist dementsprechend das jeweilige Unifikationsresultat vermerkt (`fail` bzw. `succ`).

Anfrageabarbeitung

Beobachtung

Für die angegebenen Beispiele sind die denotationelle und die operationale Semantik identisch.

Bezugstransparenz

Eine Sprache, bei der die denotationelle und die operationale Semantik zusammenfallen, nennt man referentiell transparent bzw. deklarativ.

Referentielle Transparenz bedeutet insbesondere, dass das Berechnungsergebnis vom Zustand der (abstrakten) Maschine unabhängig ist.

3.4 Anfragen mit Variablen

Anfragen mit Variablen

- universelle Programmiersprachen spezifizieren Berechnungsvorschriften
 - gewünschtes Produkt: Berechnungsergebnisse (Datenobjekte)
 - bisher: Berechnungsergebnisse auf `BOOLE`'sche Werte eingeschränkt (Zugehörigkeit zur Relation)
- Erweiterung:
Ermittlung von Berechnungsergebnissen über Variable:
 - Datenobjekte werden an Bezeichner gebunden
 - Variablenbindung (-belegung, -substitution) wird als Berechnungsergebnis ausgegeben

Anfragen mit Variablen

Variable

Paar aus Bezeichner und gebundenem Objekt (z.B. Name, numerischer Wert)

Variablenbindung

Einem Variablenbezeichner wird ein Datenobjekt eindeutig zugeordnet. Die Variable wird instanziiert.

Instanziierungsvarianten

- nichtinstanzierte Variable: Identität des Objekts noch unbekannt
- instanzierte Variable: Bindung an ein Objekt hergestellt

Variable

- Syntax:

Term	::=	Konstante <i>Variable</i> ...
Variable	::=	benannte_Variable ...
benannte_Variable	::=	Großbuchstabe {alphanumerisches_Symbol}

Gültigkeitsbereich (Sichtbarkeitsbereich)

Der Gültigkeitsbereich einer Variablen ist die Klausel.

- keine Referenz auf einen Speicherplatz!
 - keine Zustandsabhängigkeit!
- keine Blockstruktur, keine "Verschattung"
 - keine Skopusfehler
- interne Umbenennung von Variablen zur Konfliktvermeidung nötig

Der hier verwendete Variablenbegriff ist eng mit dem mathematischen Verständnis von Variablen verwandt. Der Variablenbezeichner dient als Platzhalter, der durch individuelle Objekte der Domäne *substituiert* werden kann. Eine Variable in diesem Sinne stellt keinen (bzw. keinen unmittelbaren) Zusammenhang zu einem konkreten Speicherbereich der zugrundeliegenden Maschine her. Noch deutlicher: Das logische Verarbeitungsmodell (im strengen Sinne) kennt den Begriff eines Speicherbereichs für Daten überhaupt nicht! Vielmehr enthält der Speicher des Prologsystems ausschließlich Klauseln (und zwar Fakten bzw. die später noch hinzukommenden Regeln). Klauseln lassen sich sowohl als Daten als auch als Programme interpretieren. Einen formalen Unterschied zwischen den beiden gibt es nicht. Das Verarbeitungsmodell bezieht sich ausschließlich auf die Suche nach geeigneten Eintragungen in der Datenbasis und die in diesem Zusammenhang hergestellten Variablenbindungen.

Insbesondere das imperative Verarbeitungsmodell basiert vorrangig auf der Idee, Speicherinhalte durch entsprechende Anweisungen zu verändern. Dadurch wird jedoch eine starke Abhängigkeit des Programmverhaltens von der jeweiligen Speicherbelegung begründet, die insbesondere bei komplexen Abläufen eine hohe Fehleranfälligkeit nach sich zieht. Ihr kann

nur durch strikte Vorschriften zur Programmierdisziplin (strukturierte Programmierung, Modularisierung, abstrakte Datentypen, objektorientierte Programmierung) begegnet werden. Aufgrund der kompletten Abschirmung des logikbasierten Verarbeitungsmodells vom zugrundeliegenden Speicher sind derartige Manipulationsmöglichkeiten hier weitgehend ausgeschlossen. Wegen der Beschränkung ihres Gültigkeitsbereiches auf die Klausel sind Variable in der Logikprogrammierung immer lokal und darüberhinaus auch streng gekapselt; ein Zugriff ist allein über die formalen Parameter der Prozedur möglich. Mit anderen Worten: Logikprogramme sind von Natur aus modular und wohlstrukturiert.

Da Variable in Prolog dynamisch typisiert werden, entfällt auch die Notwendigkeit zu ihrer Deklaration.

Anfrage mit Variablen

- Variable in einer Anfrage müssen nicht instantiiert sein

Unterspezifikation

- vollständig spezifiziertes Ziel

```
?- liebt(susi,karl).
```

- partiell unterspezifiziertes Ziel

```
?- liebt(susi,X).
```

- vollständig unterspezifiziertes Ziel

```
?- liebt(X,Y).
```

Anfragen mit Variablen

- Semantik:

Prüfen der Konsistenz mit den Eintragungen der Datenbasis unter Unifikation

- Instanziierung von Variablen durch Unifikation.
- Ermitteln einer oder mehrerer Variablenbindungen, die die Aussage des Ziels wahr machen.

→ Erweiterung des Unifikationsbegriffs erforderlich

Unifikation (II)

Unifikation von variablenhaltigen Strukturen

Ermittlung einer solchen Variablensubstitution (Bindung), die die Gleichheit (Verträglichkeit) der Strukturen (Anfrage und Datenbasiseintrag) herstellt

- Vergleichskriterien:
 - identischer Prädikatsname
 - gleiche Stelligkeit
 - Unifikation der Argumente

Konstante / Konstante	Identitätstest	susi = karl
Variable / Konstante	Instanziierung	X = susi
Konstante / Variable	Instanziierung	karl = Y
Variable / Variable	Koreferenz	X = Y

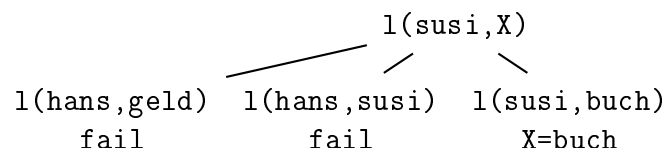
Unifikation (II)

Anfrage	Fakt	Variablensubstitution
liebt(susi,X)	liebt(susi,buch)	X=buch
liebt(X,buch)	liebt(susi,buch)	X=susi
	liebt(karl,buch)	X=karl

Anfragen mit Variablen

- Denotationelle Semantik: Gibt es eine Substitution für den / die Variablenbezeichner in der Anfrage, so dass die Anfrage wahr wird?
- Operationale Semantik: Suche nach einer konsistenten Variablenbindung

```
?- liebt(susi,X).
    liebt(hans,geld).      fail
    liebt(hans,susi).     fail
    liebt(susi,buch).     succ(X=buch)
X=buch
```



Im Suchbaum wird nunmehr anstelle der erfolgreichen Unifikation (succ), die durch die Unifikation erzeugte Variablenbindung vermerkt (z.B. X=buch).

Alternative Variablenbindungen

- Ermittlung alternativer Variablenbindungen: Eingabe von “;” (logisches ODER) im Anschluss an eine Resultatsausgabe

```
?- liebt(susi,X).  
X=buch.
```

```
?- liebt(X,buch).  
X=susi ;  
X=karl.
```

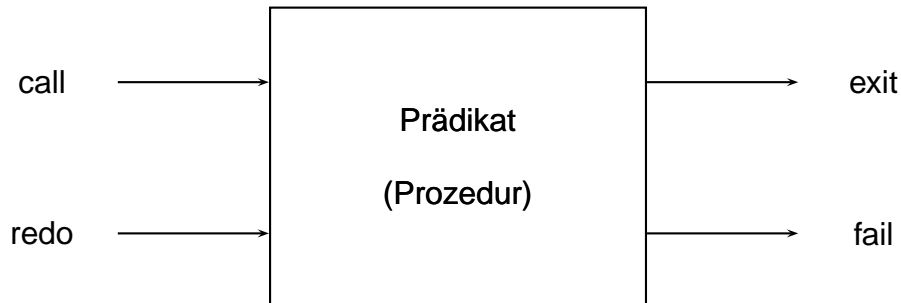
Alternative Variablenbindungen

- Operationale Semantik:
Auffinden alternativer Variablenbindungen durch Backtracking (Zurücksetzen)
 - bei erfolgreicher Unifikation: Zeiger in die Datenbasis setzen
 - bei Aufforderung:
 - * Zurückgehen bis zum Zeiger
 - * ggf. Variablenbindungen rückgängig machen
 - * Suche nach alternativen Fakten in der Datenbasis

Wegen der prinzipiellen Möglichkeit zur Arbeit mit unterspezifizierten Anfrage, in denen bisher noch nicht instanziierte Variable auftreten, können und müssen in jedem Fall *alle* Instanzierungsvarianten und das durch sie ausgelöste Programmverhalten betrachtet werden. Für ein Prädikat mit n Argumentpositionen sind dies 2^n verschiedene Instanzierungsvarianten, für die jeweils wieder mehrere Testfälle zu berücksichtigen sind! Ein Test bzw. eine Programmverifikation ist erst vollständig, wenn die Korrektheit des Programms für alle Instanzierungsvarianten gezeigt werden konnte.

Die aktuell vorliegende Instanzierungsvariante wird im folgenden durch ein Instanzierungsschema verdeutlicht (z.B. `liebt(in,out)`), wobei `in` für eine instanziierte Variable, d.h. eine Eingabeinformation des Programmaufrufs, und `out` für eine (zum Anfragezeitpunkt) uninstantiierte Variable, d.h. eine Ausgabeinformation des Programmaufrufs steht.

Das Vier-Port-Modell



Das Vier-Port-Modell

- ports können als spy-points für `trace/1`, `trace/2`, `debug/0`, `leash/1`, etc. spezifiziert werden

```

?- trace(liebt).
...
?- trace(liebt,[call,fail]).
...
?- leash(+call).
  
```

- `leash/1` erlaubt auch 5. "Port": `unify`

Exkurs: Programmiermethodik

Testen eines Prädikats (1)

Beim systematischen Testen eines Prädikats müssen stets

- positive und negative Beispiele und
- alle Instanziierungsvarianten berücksichtigt werden.
- Empfehlenswert ist die Zusammenstellung einer repräsentativen Testmenge bereits vor und während der Programmerarbeitung.

Prädikatsschemata

Im Prädikatsschema werden stets die zulässigen Instanziierungsvarianten vermerkt:

- + nur Eingabeparameter
- Ausgabeparameter
- ? Ein-/Ausgabeparameter

→ SE I

- Beispiel: `liebt(?wer,?wen-oder-was)`

Beispielanfragen

- Anfrage mit einem partiell spezifizierten Ziel (1)

aktuelle Instanziierung: `liebt(+,-)`

```
?- liebt(susi,X).      #1
    liebt(hans,geld).  fail
    liebt(hans,susi).  fail
    liebt(susi,buch).  succ(X=buch)
X=buch ;
    liebt(karl,buch).  fail
false.
```

$$\begin{array}{c}
 \text{l(susi,X)} \\
 \swarrow \quad \downarrow \quad \searrow \\
 \text{l(hans,geld)} \quad \text{l(hans,susi)} \quad \text{l(susi,buch)} \quad \text{l(karl,buch)} \\
 \text{fail} \quad \text{fail} \quad \text{X=buch} \quad \text{fail}
 \end{array}$$

Beispielanfragen

- Anfrage mit einem partiell spezifizierten Ziel (2)

aktuelle Instanziierung: `liebt(-,+)`

```
?- liebt(X,buch).     #1
    liebt(hans,geld).  fail
    liebt(hans,susi).  fail
    liebt(susi,buch).  succ(X=susi)
X=susi ;
    liebt(karl,buch).  succ(X=karl)
X=karl ;
false.
```

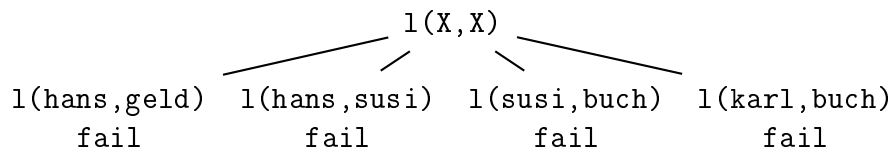
$$\begin{array}{c}
 \text{l(X,buch)} \\
 \swarrow \quad \downarrow \quad \searrow \\
 \text{l(hans,geld)} \quad \text{l(hans,susi)} \quad \text{l(susi,buch)} \quad \text{l(karl,buch)} \\
 \text{fail} \quad \text{fail} \quad \text{X=susi} \quad \text{X=karl}
 \end{array}$$

Beispielanfragen

- Anfrage mit einem partiell spezifizierten Ziel (3)

aktuelle Instanziierung: `liebt(-,-)`

```
?- liebt(X,X).       #1
    liebt(hans,geld). fail
    liebt(hans,susi). fail
    liebt(susi,buch). fail
    liebt(karl,buch). fail
false.
```

Koreferenz

Koreferenz (Sharing)

Durch mehrfache Verwendung einer Variablen innerhalb einer Klausel wird *Koreferenz* gefordert.

Koreferente Variable haben immer den gleichen Wert (sind an das gleiche Datenobjekt gebunden) sobald sie instanziiert werden.

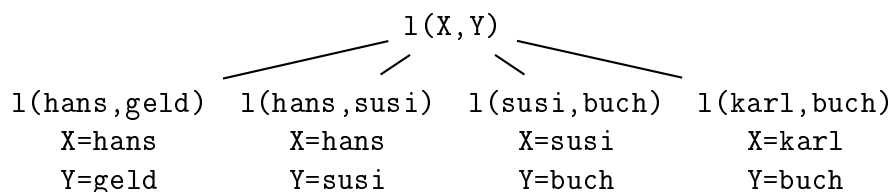
Koreferenz spezifiziert eine Forderung nach Identität auch wenn der Wert der Variablen zum Aufrufszeitpunkt noch gar nicht bekannt ist.

Beispielanfragen

- Anfrage mit einem vollständig unterspezifizierten Ziel aktuelle Instanziierung:
`liebt(-,-)`

```

?- liebt(X,Y).
X=hans,Y=geld ;
X=hans,Y=susi ;
X=susi,Y=buch ;
X=karl,Y=buch.
  
```



Anfragen mit Variablen

Verschieden stark spezifizierte Ziele können unterschiedliches prozedurales Verhalten auslösen:

vollständig spezifiziert	→	Enthaltensein in einer Relation
partiell (unter-)spezifiziert	→	Teilmenge einer Relation
vollständig unterspezifiziert	→	gesamte Relation

Auch bei Einführung von Variablen bleibt die Bezugstransparenz erhalten.

Anfragen mit Variablen

Informationsanreicherung

Relationale Programmierung kann als Prozess der informationellen Anreicherung einer (unterspezifizierten) Anfrage durch die Instanziierung von Variablen interpretiert werden.

```

liebt(susi,buch)  $\Rightarrow$  { liebt(susi,buch) }
liebt(susi,X)     $\Rightarrow$  { liebt(susi,buch) }
liebt(X,buch)     $\Rightarrow$  { liebt(susi,buch), liebt(karl,buch) }
liebt(X,X)        $\Rightarrow$   $\emptyset$ 
liebt(X,Y)        $\Rightarrow$  { liebt(hans,geld), liebt(hans,susi),
                        liebt(susi,buch), liebt(karl,buch) }

```

Zwischenbilanz: Logikprogrammierung

- Die algorithmische Grundstruktur wird im Verarbeitungssystem festgelegt, nicht im Programm.
- Ein Programm beschreibt eine ganze Klasse von Algorithmen (Algorithmenschema).
- Algorithmische Details, insbesondere die Verarbeitungsrichtung, werden erst durch die Anfrage (Prädikatsaufruf) festgelegt.

Die Richtungsunabhängigkeit (Reversibilität) ist ein wesentliches Merkmal der relationalen Programmierung, für das in den anderen Verarbeitungsmodellen kein Äquivalent existiert: Ein Programm realisiert in Abhängigkeit vom Prädikatsaufruf ganz unterschiedliche prozedurale Abläufe.

3.5 Komplexe Anfragen

Komplexe Anfragen

- Konjunktion mehrerer Teilziele

```

?- liebt(X,susi),liebt(X,geld).
X=hans.

```

- Syntax:

<i>Ziel</i>	<i>::=</i>	<i>elementares_Ziel</i> <i>komplexes_Ziel</i>
<i>komplexes_Ziel</i>	<i>::=</i>	<i>elementares_Ziel</i> <i>','</i> <i>Ziel</i>

- Denotationelle Semantik: logisches UND

- Operationale Semantik: konjunktiv verknüpfte Ziele werden sequentiell abgearbeitet. Die Konjunktion war erfolgreich, wenn alle Teilziele erfolgreich waren.

Komplexe Anfragen

- Disjunktion mehrerer Teilziele

```
?- liebt(X,susi);liebt(susi,X).
X=hans ;
X=buch.
```

- Syntax: $\text{komplexes_Ziel} ::= \text{elementares_Ziel} (',' | ';') \text{Ziel}$
- Denotationelle Semantik: logisches ODER
- Operationale Semantik: disjunktiv verknüpfte Ziele werden sequentiell abgearbeitet. Die Disjunktion war erfolgreich, wenn ein Teilziel erfolgreich war.
- Pragmatik: Vermeiden Sie (vorerst) die Verwendung der Disjunktion, weil sie Programme teilweise schwer verständlich macht.

Suche bei konjunktiv verknüpften Anfragen

Tiefensuche

Eine (partiell) erfolgreiche Variablenbindung wird an den noch verbleibenden Teilzielen überprüft.

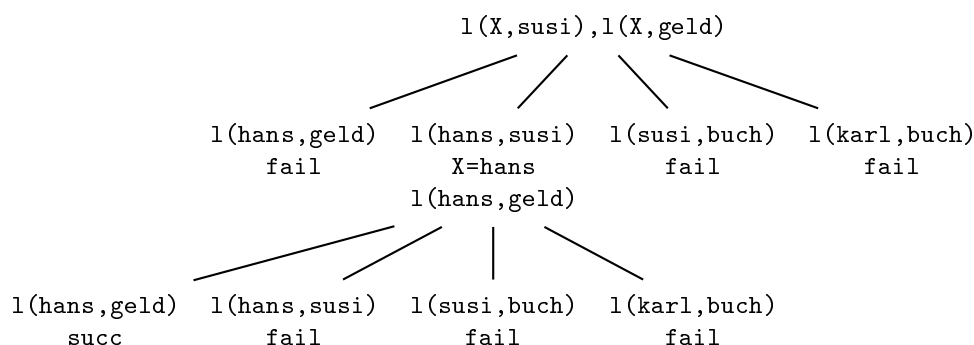
Der Suchbaum wird zuerst in die Tiefe expandiert.

Backtracking:

Wurde ein Teilbaum exhaustiv, aber erfolglos durchsucht, wird zu einem früheren Entscheidungspunkt zurückgegangen.

Beispielanfragen

- konjunktiv verknüpfte Anfrage: $\text{liebt}(-,+), \text{liebt}(-,+)$
- Suchbaum



Durch die Verwendung komplexer Anfragen entsteht ein hierarchisch strukturierter Suchraum: Nach der erfolgreichen Unifikation eines Teilziels mit einer Eintragung der Datenbasis und der dadurch hergestellten Variablenbindung führt die Aktivierung des folgenden Teilziels zu einer erneuten Verzweigung des Suchraums, da das nunmehr aktuelle Teilziel wieder mit allen Einträgen der Datenbasis unifiziert werden muss.

Beispielanfragen

- Trace

```
?- liebt(X,susi),liebt(X,geld).
    ?- liebt(X,susi).                #1
        liebt(hans,geld).            fail
        liebt(hans,susi).            succ(X=hans)
    ?- liebt(hans,geld).              #2
        liebt(hans,geld).            succ(X=hans)
X=hans ;
        liebt(hans,susi).            fail
        liebt(susi,buch).            fail

        liebt(karl,buch).            fail
BT #1
    liebt(susi,buch).                fail
    liebt(karl,buch).                fail
false.
```

Auch bei komplexen Anfragen sind unterschiedliche Instanziierungsvarianten möglich. Überprüfen Sie das Programmverhalten unter geänderten Instanziierungsbedingungen bitte selbst!

Anonyme Variable

- Syntax:

Variable ::=	benannte_Variable
	anonyme_Variable ...
anonyme_Variable ::=	'_'
- Semantik: Eine anonyme Variable kann an jeden Wert gebunden werden. Mehrere anonyme Variable innerhalb einer Klausel werden unabhängig voneinander instanziiert (stellen keine Koreferenz her).
- Pragmatik: Anonyme Variable werden verwendet, wenn die Variablenbindung für die jeweilige Berechnungsaufgabe irrelevant ist.
- Beispiel: Welche Mehrfamilienhäuser gibt es?
aktuelle Instanziierung: obj(,+,,-,-,)

```
?- obj(_,mfh,Str,No,_).
    Str=bahnhofstr, No=28.
```

Anfragen über mehreren Relationen

Beispieldatenbasis 2

```
% obj(?Objektnr,?Objekttyp,?Strassenname,?Hausnr,?Baujahr).
obj(1,efh,gaertnerstr,15,1965).
obj(2,efh,bahnhofsstr,27,1943).
obj(3,efh,bahnhofsstr,29,1955).
obj(4,mfh,bahnhofsstr,28,1991).
obj(5,bahnhof,bahnhofsstr,30,1901).
obj(6,kaufhaus,bahnhofsstr,26,1997).
obj(7,efh,gaertnerstr,17,1982).

% bew(?Vorgangsnr,?Objektnr,?Verkaeuer,?Kaeufer,?Preis,
%      ?Verkaufsdatum).
bew(1,1,mueller,meier,450000,'1997.01.01').
bew(2,3,schulze,schneider,560000,'1988.12.13').
bew(3,3,schneider,mueller,615000,'1996.12.01').
bew(4,5,bund,piepenbrink,3500000,'2001.06.01').
```

Anfragen über mehreren Relationen

- Beispiel: Wer hat welches Haus gekauft?

```
?- obj(X,_,Str,No,_),bew(_,X,_,Kaeufer,_,_).
   X=1, Str=gaertnerstr, No=15, Kaeufer=meier ;
   X=3, Str=bahnhofsstr, No=29, Kaeufer=schneider ;
   X=3, Str=bahnhofsstr, No=29, Kaeufer=mueller ;
   X=5, Str=bahnhofsstr, No=30, Kaeufer=piepenbrink.
```

- Der Bezug zwischen den Tabellen wird durch Koreferenz hergestellt.
- Eine komplexe Anfrage mit Variablen definiert eine neue Relation über den betreffenden Domänen.

3.6 Prädikate zweiter Ordnung

Prädikate zweiter Ordnung

- ausgewählte Prädikate fordern Prädikatsaufrufe (Ziele) als Eingabewerte
- Beispiel: `findall/3`
 - Syntax: `findall(Term,Ziel,Liste)`
 - Semantik: Aufsammeln *aller* Resultate des Prädikatsaufrufs `Ziel` als instanziierte Varianten des Ausdrucks `Term` in einer Ergebnisliste `Liste`
 - Pragmatik: `Term` und `Ziel` haben üblicherweise gemeinsame (uninstanziierte) Variable.
 - Einzige Instanzierungsvarianten: `findall(+Term,+Ziel,?Liste)`

Prädikate zweiter Ordnung

- Beispielaufruf

```
?- findall(B,obj(_,efh,_,_,B),L).  
L = [1965, 1943, 1955, 1982].  
?- findall(adresse(Str,Nr),  
           (obj(_,efh,Str,Nr,B),B>1960),  
           L).  
L = [adresse(gaertnerstr, 15), adresse(gaertnerstr, 17)].
```

- weitere Prädikate 2. Ordnung: `setof/3`, `bagof/3`

3.7 Relationale DB-Systeme

Relationale Datenbank-Systeme

- SQL als Abfragesprache
- Relationenalgebra als Verarbeitungsmodell
 - Selektion: Anfragen
 - Projektion: Anfragen mit anonymen Variablen
 - Join: Mehrtabellenanfragen mit gemeinsamer Variablen
 - aggregierende Operatoren (`count`, `max`, ...): Prädikate zweiter Ordnung

Relationale Datenbankabfragesprachen sind referentiell transparent und richtungsunabhängig.

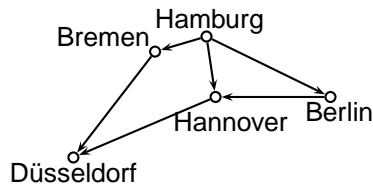
Relationale Datenbank-Systeme

- der relationale Datenbank-Kern von Prolog ...
 - ist äquivalent zur Recherchefunktionalität von SQL
 - ist eine echte Teilmenge von Prolog
 - ist nicht berechnungsuniversell (wie jede DB-Abfragesprache)
- Datenbanksysteme ...
 - ... ermöglichen persistente Haltung großer Datenbestände
 - ... ermöglichen Mehrnutzerbetrieb
 - ... unterstützen Wartung und Aktualisierung der Daten
 - ... sichern die Integrität der Daten

→ GDB

4 Deduktive Datenbanken

Deduktive Datenbanken



4.1 Deduktion

Deduktion

Deduktion

Ableitung von Folgerungen aus einer Axiomenmenge

- Aussagenlogik:
 - Axiome: Atomare Formeln und Implikationen
 - Schlussregel: modus ponens

$$a \wedge (a \rightarrow b) \models b$$

Deduktion

- Prädikatenlogik

Hans ist ein Mann.
Alle Männer sind Verbrecher.

Hans ist ein Verbrecher.

mann(hans)
$\forall x. \text{mann}(x) \rightarrow \text{verbrecher}(x)$

verbrecher(hans)

Deduktion

- Bisher: Spezifikation einer Relation durch Aufzählung ihrer Elemente (Fakten)
→ extensionale Spezifikation

$$\mathcal{R}_i = \{(x_{11}, \dots, x_{m1}), (x_{12}, \dots, x_{m2}), \dots, (x_{1n}, \dots, x_{mn})\}$$

- Ziel: Berechnung der Elemente einer Relation aufgrund von *Regeln* → intensionale Spezifikation

$$\mathcal{R}_i = \{(x_1, \dots, x_n) | \text{Bedingungen}\}$$

- Regeln entsprechen den Implikationen der Prädikatenlogik
- Ableitung neuer Fakten durch Anwendung von Regeln auf bereits bekannte Fakten

Deduktion

- relationale Abstraktion: Aufbau intensionaler Spezifikationen.
 - Beseitigung von Redundanz
 - Spezifikation von Relationen über unendlichen Domänen

4.2 Regeln

Regeln

- Syntax:

Klausel	::=	(Fakt Ziel Regel) '.'
Regel	::=	Kopf ':-' Körper
Kopf	::=	Struktur
Körper	::=	Ziel
- Beispiele

Kopf	Körper
verbrecher(X)	:- mann(X).
mann(X)	:- mensch(X), maennlich(X), erwachsen(X).
- ein Prädikat ist eine Menge von Klauseln mit gleicher Signatur

Die Definition von Relationen mit Hilfe von Regeln basiert auf der Tatsache, dass allein durch die Vorgabe eines komplexen, variablenhaltigen Ziels eine neue Relation über den Domänen dieser Variablen definiert ist (vgl. vorangegangener Abschnitt). Die neue Relation besitzt damit jedoch noch keinen Namen.

Tritt nunmehr ein solches komplexes Ziel als Körper einer Regel auf, so wird der neuen Relation durch den Kopf der Regel auch noch ein Name zugewiesen, der dann in anderen Anfragen verwendet werden kann. Hierauf beruht der Abstraktionsmechanismus der relationalen Komposition: komplexe Prädikatsdefinitionen werden schrittweise aus elementaren zusammengesetzt. Auf diese Weise kann ggf. Komplexität vor dem Nutzer eines Prädikats verborgen werden.

Am Beispiel der Regel `verbrecher(X) :- mann(X).` sehen wir, dass sich dieses Herangehen nicht auf den (allerdings typischen) Fall der komplexen Anfragen beschränkt. Da in diesem Fall allerdings nur die Vergabe eines zusätzlichen Namens für die gleiche Relation erfolgt, ist eine solche Definition für das Ergebnis konkreter Berechnungsprozesse bedeutungslos.

Regeln

- Denotationelle Semantik:
logische Ableitbarkeit mit *modus ponens*
- Operationale Semantik: (Abarbeitung einer Regel)
 - Unifikation des Regelkopfes mit der Anfrage
 - falls erfolgreich:
 - * Umbenennen der Variablen zur Vermeidung von Namenskonflikten
 - * Ersetzen der Anfrage durch Regelkörper
 - * Verwenden der durch die Unifikation des Regelkopfes hergestellten Variablenbindungen
 - * Abarbeiten des Regelkörpers als neue, komplexe Anfrage
- Pragmatik: Ein Regelkopf ist normalerweise keine Grundstruktur. Die Variablen des Regelkopfes können als formale Parameter einer Prozedur betrachtet werden.

Nunmehr wird endgültig deutlich, dass zwischen einer Anfrage, die am Systemprompt des Prolog-Interpreters eingegeben wird, und dem Körper einer Klausel, der bei der Anwendung einer Regel durch Variableninstanziierung zum neuen Teilziel wird, praktisch keinerlei Unterschied besteht. Eine Anfrage ist demnach einfach eine Regel ohne Kopf!

Auf ganz ähnliche Weise kann man auch einen Zusammenhang zwischen Fakten und Regeln herstellen. Auf der einen Seite fordert eine Regelanwendung die erfolgreiche Unifikation des Kopfes mit der Anfrage, ehe dann der Aufruf des neuen Teilziel erfolgen kann. Dabei entsteht das neue Teilziel durch Variableninstanziierung aus dem Körper der Regel. Auf der anderen Seite erfordert auch die Einbeziehung eines Faktums eine erfolgreiche Unifikation, allerdings entfällt hier die Aktivierung eines neuen Teilziels. Offensichtlich stellt also ein Fakt nichts anderes dar, als eine Regel ohne Körper.

Versucht man diesen Gedankengang konsequent fortzusetzen, so kann man feststellen, dass auch das Resultat einer Ableitungskette dem allgemeinen Klauselformat einer Regel entspricht: Unifiziert man eine Anfrage (Regel ohne Kopf) mit einem Faktum (Regel ohne Körper) und aktiviert anschließend den allerdings gar nicht existenten Körper des Fakts als neues Teilziel, so entsteht eine Klausel ohne Kopf (weil eine Anfrage) *und* ohne Körper (weil aus einem Fakt erzeugt). Die Ableitungsprozedur bricht also immer dann erfolgreich ab, wenn irgendwann eine solche "leere Klausel" als Teilziel auftritt.

Somit ist die Regel als universelle Klauselform die einzige "Datenstruktur" auf der der gesamte Ableitungsprozess eines Prologsystems beruht. Durch diese bemerkenswerte Einfachheit entsteht ein Verarbeitungsmodell von großer Eleganz und innerer Schönheit, das ausschließlich auf den drei Elementen Klauseln, Unifikation und Suche basiert.

Für Freunde der Logik: Das hier betrachtete spezielle Regelformat wird in der mathematischen Logik auch als HORN-Klausel bezeichnet. Die leere Klausel entspricht dem generellen Widerspruch und das Ableitungsverfahren (ein Widerspruchsbeweis) wird als Resolution bezeichnet.

Regeln

erweiterte Beispieldatenbasis 1, unverändertes Prädikatsschema

```
% liebt(?Wer,?Wen-oder-was)
% Wer und Wen-oder-was sind Namen, so dass
% Wen-oder-was von Wer geliebt wird
liebt(hans,geld).
liebt(hans,susi).
liebt(susi,buch).
liebt(karl,buch).
liebt(paul,X):-liebt(X,buch).
```

Exkurs: Programmiermethodik

Testen eines Prädikats (2)

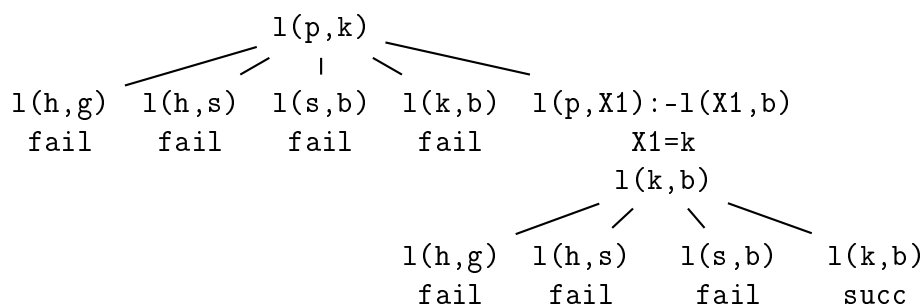
Beim systematischen Testen eines Prädikats müssen stets

- positive und negative Beispiele,
- alle Instanziierungsvarianten und
- alle wesentlichen Fallunterscheidungen berücksichtigt werden.

Beispielaufufe

- Anfrage mit vollständig instanziiertem Ziel: liebt(+,+)

```
?- liebt(paul, karl).
```



→ Test auf Enthaltensein eines Elementes in der Relation

Neben der Abarbeitung komplexer Ziele (siehe vorangegangener Abschnitt) ist also die Anwendung einer Regel ein weiterer Grund, der eine Verzweigung im Suchraum auslösen kann. Nach der erfolgreichen Unifikation des Kopfes mit der Anfrage wird der Körper der Regel (unter Variablensubstitution) zur neuen Anfrage, die mit allen Einträgen der Datenbasis auf Unifizierbarkeit geprüft werden muss.

Da sich dieser Vorgang beliebig oft wiederholen kann, stellt er eine ernstzunehmende Quelle für Effizienzprobleme dar. Zur Verdeutlichung des Problems empfehle ich Ihnen, das Wachstum des Suchraums bei konstantem Verzweigungsfaktor (z.B. $v = 3$) für relativ geringe Suchraumtiefen (z.B. bis zu einer Tiefe $t = 5$) zu untersuchen. Sie sollten dabei aber beachten,

dass dieses Wachstum keine inhärente Eigenschaft des verwendeten Verarbeitungmodells ist, sondern immer aus den jeweils zu bearbeitenden Problemen resultiert. Diese auch als nicht-deterministische Probleme bezeichnete Klasse von Suchaufgaben betrachten wir im Zusammenhang mit den praktischen Anwendungsbeispielen der Logikprogrammierung näher.

Beispielaufufe

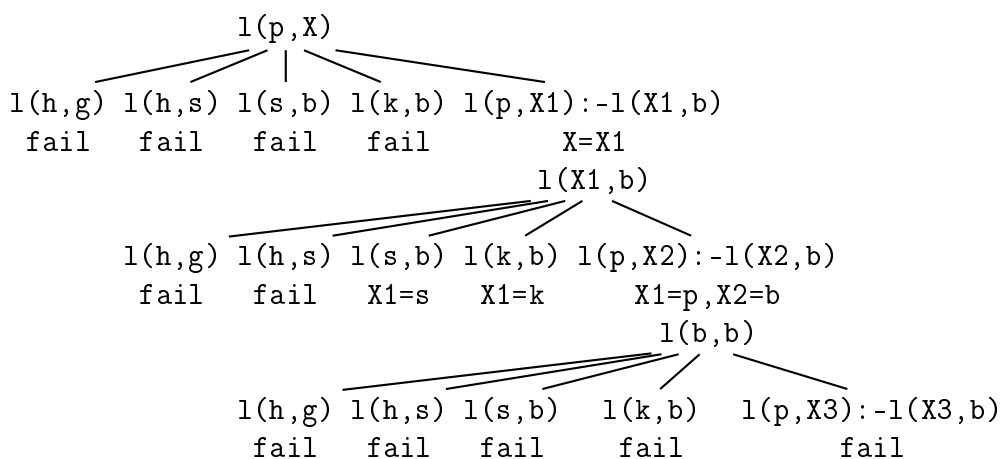
- Anfrage mit vollständig instanziiertem Ziel liebt(+,+)

```
?- liebt(paul, karl).           #1
    liebt(hans, geld).         fail
    liebt(hans, susi).         fail
    liebt(susi, buch).         fail
    liebt(karl, buch).         fail
    liebt(paul, X1):-liebt(X1, buch). succ(X1=karl)
?-  liebt(karl, buch).         #2
    liebt(hans, geld).         fail
    liebt(hans, susi).         fail
    liebt(susi, buch).         fail
    liebt(karl, buch).         succ
true.
```

Beispielaufufe

- Anfrage mit partiell instanziiertem Ziel (1): liebt(+,-)

```
?- liebt(paul, X).
```



Beispielaufufe

- Anfrage mit partiell instanziiertem Ziel (1) liebt(+,-)

```

?- liebt(paul,X).                #1
   liebt(hans,geld).             fail
   ...
   liebt(paul,X1):-liebt(X1,buch). succ(X=X1)
?- liebt(X1,buch).               #2
   liebt(hans,geld).             fail
   liebt(hans,susi).             fail
   liebt(susi,buch).             succ(X=X1=susi)
X=susi ;
   liebt(karl,buch).             succ(X=X1=karl)
X=karl ;
   liebt(paul,X2):-liebt(X2,buch). succ(X1=paul,
                                     X2=buch)
?- liebt(buch,buch).             #3
   liebt(hans,geld).             fail
   ...
   liebt(paul,X3):-liebt(X3,buch). fail
BT #2
BT #1
false.

```

Beispielaufufe

- Anfrage mit partiell instanziiertem Ziel (2) liebt(-,+)

```

?- liebt(X,buch).                #1
   liebt(hans,geld).             fail
   liebt(hans,susi).             fail
   liebt(susi,buch).             succ(X=susi)
X=susi ;
   liebt(karl,buch).             succ(X=karl)
X=karl ;
   liebt(paul,X1):-liebt(X1,buch). succ(X=paul,X1=buch)
?- liebt(buch,buch).             #2
   liebt(hans,geld).             fail
   liebt(hans,susi).             fail
   liebt(susi,buch).             fail
   liebt(karl,buch).             fail
   liebt(paul,X2):-liebt(X2,buch). fail
BT #1
false

```

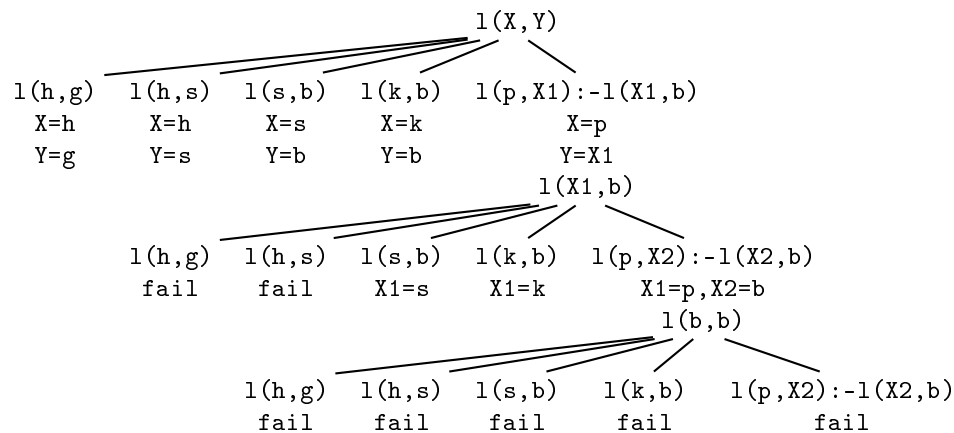
→ Aufzählung aller Elemente der Relation, die der Anfragebedingung genügen

Beispielaufufe

- Anfrage mit vollständig uninstantiiertem Ziel

liebt(-,-)

?- liebt(X,Y).



→ vollständige Aufzählung der Relation

Beispielanfragen

- Anfrage mit vollständig uninstantiiertem Ziel

```

?- liebt(X,Y).                                #1
    liebt(hans,geld).                          succ(X=hans,Y=geld)
X=hans,Y=geld ;
...
    liebt(paul,X1):-liebt(X1,buch).            succ(X=paul,Y=X1)
?- liebt(X1,buch).                            #2
    liebt(hans,geld).                          fail
    liebt(hans,susi).                         fail
    liebt(susi,buch).                         succ(X=paul,Y=X1=susi)
X=paul,Y=susi ;
    liebt(karl,buch).                         succ(X=paul,Y=X1=karl)
X=paul,Y=karl ;
    liebt(paul,X2):-liebt(X2,buch).            succ(X=paul,Y=X1=paul,X2=buch)
?- liebt(buch,buch).                          #3
...
BT #2
BT #1
false.

```

4.3 Spezielle Relationstypen

Spezielle Relationstypen

- Symmetrische Relationen
- Transitive Relationen
- 1:m Relationen
- Reflexive Relationen

Im Folgenden untersuchen wir die Besonderheiten extensional und intensional definierter Relationen im Hinblick auf eine Reihe von Eigenschaften, wie sie auch in der Mathematik zur Charakterisierung von Relationen benutzt werden. Ziel dieses Vorgehen ist es, das Denken in Relationen (im Gegensatz zum Denken in prozeduralen Abläufen) zu fördern. Am Beginn der Entwicklung eines Programms im logikbasierten Verarbeitungsmodell sollte immer die Frage stehen: *Was für eine Art von Relation möchte ich spezifizieren und welche Eigenschaften soll sie haben?* Aus der Antwort darauf leiten sich dann bestimmte Einsichten hinsichtlich der möglicherweise zu erwartenden Implementierungsprobleme und der jeweils in Frage kommenden Lösungsansätze ab.

Hierbei wird deutlich werden, dass der analytische Apparat der Mathematik ganz unmittelbare Hinweise für die praktische Programmentwicklung bereitstellen kann.

Symmetrische Relationen

Symmetrie

Eine zweistellige Relation p ist symmetrisch, wenn

$$p(X, Y) \leftrightarrow p(Y, X)$$

allgemeingültig ist.

Symmetrische Relationen

- Beispiel: Zwillingsrelation

```
% zwilling_von(?Pers1,?Pers2)
% Pers1 und Pers2 sind Namen, so dass Pers1 und Pers2
% Zwillinge sind
```

```
zwilling_von(paul,anne).
zwilling_von(hans,nina).
```

```
?- zwilling_von(paul,anne).
true.
?- zwilling_von(anne,paul).
false.
```

Eine extensional definierte, zweistellige Relation ist standardmäßig unsymmetrisch.

Symmetrische Relationen

- Herstellung der Symmetrieeigenschaft erfordert zusätzlichen Aufwand
- 3 Möglichkeiten

1. Angabe der Inversen

```
zwilling_von(paul,anne).  
zwilling_von(anne,paul).  
zwilling_von(hans,nina).  
zwilling_von(nina,hans).
```

→ Redundanz in der Datenbasis

Symmetrische Relationen

- Herstellung der Symmetrieeigenschaft (Forts.)

2. Symmetriedefinition über ein Hilfsprädikat

```
% zwilling_von(?Pers1,?Pers2)  
% Pers1 und Pers2 sind Namen, so dass Pers1  
% und Pers2 Zwillinge sind  
zwilling_von(X,Y):-z_v(X,Y).  
zwilling_von(X,Y):-z_v(Y,X).  
z_v(paul,anne).  
z_v(hans,nina).
```

→ faktenbasierte Modellierung

3. intensionale Definition

```
zwilling_von(X,Y) :- mutter_von(M,X), mutter_von(M,Y),  
                    hat_geburtstag(X,D), hat_geburtstag(Y,D).
```

→ wissensbasierte (symmetrische) Modellierung

Symmetrische Relationen

- Hilfsprädikat: ist es wirklich erforderlich?

```
zwilling_von(paul,anne).
zwilling_von(X,Y):-zwilling_von(Y,X).
```

```
?- zwilling_von(paul,anne).
true.
```

```
?- zwilling_von(anne,paul).
true.
```

Symmetrische Relationen

- unterspezifizierter Aufruf: `zwilling_von(-,-)`

```
?- zwilling_von(X,Y).
X=paul, Y=anne ;
X=anne, Y=paul;
X=paul, Y=anne ;
X=anne, Y=paul .
```

- Differenz zwischen denotationeller und operationaler Semantik:
 - denotationell: Relation enthält zwei Elemente
 - operational: Suche erzeugt (zyklisch) beliebig viele Variablenbindungen

Symmetrische Relationen

- Aufruf mit einem unbekannten Objekt: `zwilling_von(+,+)`

```
?- zwilling_von(paul,karl).
...

```

- Differenz zwischen denotationeller und operationaler Semantik:
 - denotationell: Ziel ist in der Relation nicht enthalten (Resultat: no)
 - operational: Suche terminiert nicht
- Verdacht: *rekursive* Spezifikation ruft Terminierungsprobleme hervor!
- Ausweg: Vermeiden *unbeschränkter* Rekursion durch Hilfsprädikate

Rekursion (1)

Rekursion

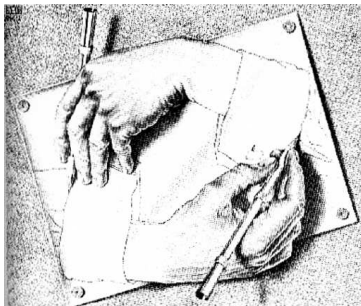
Selbstbezüglichkeit einer formalen Struktur:

- zwei Formen
 - Eine Prädikatsdefinition wird auf sich selbst zurückgeführt:

```

zwillling(X,Y) :-
    zwillling(Y,X).

```



- Ein Datentyp wird durch sich selbst definiert (Struktur, Liste).

Wie Sie im weiteren Verlauf der Vorlesung noch sehen werden, ist Rekursion eine sehr wichtige Steuerstruktur, insbesondere in der funktionalen und logik-basierten Programmierung. Da Rekursion aber offensichtlich auch Terminierungsprobleme hervorrufen kann, ist bei ihrer Verwendung erhöhte Wachsamkeit geboten. Rekursive Programme, die zu einem zyklischen Prädikatsaufruf mit *völlig unveränderten* Teilzielen führen, sind unbedingt zu vermeiden, da sie zwangsläufig Terminierungsprobleme hervorrufen. Genau dieser Fall ist in der ESCHER-Grafik sehr schön illustriert. Die typische Lösung für derartige Fälle ist die Verwendung von Hilfsprädikaten, die den unmittelbaren rekursiven Bezug aufbrechen.

Der Grund dafür, dass das Problem in der Mathematik in dieser Form nicht auftritt, ist die rein deklarative (statische) Betrachtungsweise der Logik: Die Anfrage ist Element der Relation oder nicht. Mehr noch, die Elemente einer Menge sind eindeutig bestimmt, d.h. ein und dasselbe Element kann nicht zweimal Element der gleichen Menge sein. Diese Identität muss bei einem prozeduralen (dynamischen) Berechnungsverfahren aber erst einmal nachgewiesen werden. Rekursive Regeln, die unveränderte Teilziele aktivieren, erzeugen demgegenüber jedoch beliebig viele formal gleiche Elemente, ohne dass allerdings deren Identität dem System bekannt wäre. Durch zusätzlichen Programmieraufwand kann in Spezialfällen die Ableitung identischer Teilziele überwacht und unterbunden werden. Eine allgemeine Lösung dieses Problems für beliebige Programme existiert jedoch nicht.

Transitive Relationen

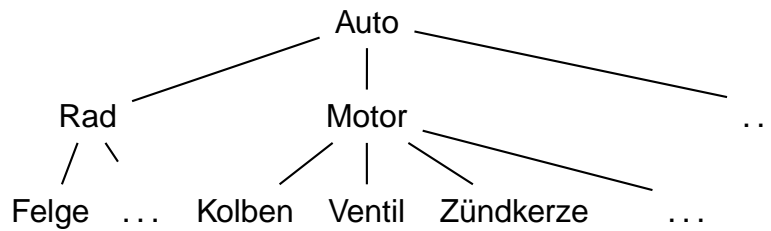
Transitivität

Eine zweistellige Relation p ist transitiv, wenn

$$p(X, Y) \wedge p(Y, Z) \rightarrow p(X, Z)$$

allgemeingültig ist.

- Verwendung z. B. zur terminologischen Wissensrepräsentation



Dies ist bereits die zweite Verwendung von Baumstrukturen zur Modellierung von bestimmten Gegebenheiten eines Gegenstandsbereichs. Ging es im ersten Fall um die Veranschaulichung der Abfolge von Suchschritten bei der Abarbeitung von Logikprogrammen, so handelt es sich hier um die Abbildung der konzeptuellen Struktur eines bestimmten Anwendungsgebiets. Die Modellierung erfolgt dabei durch Fakten in der Datenbasis und eine Regel, die die Transitivitätsbeziehung herstellt.

Transitive Relationen

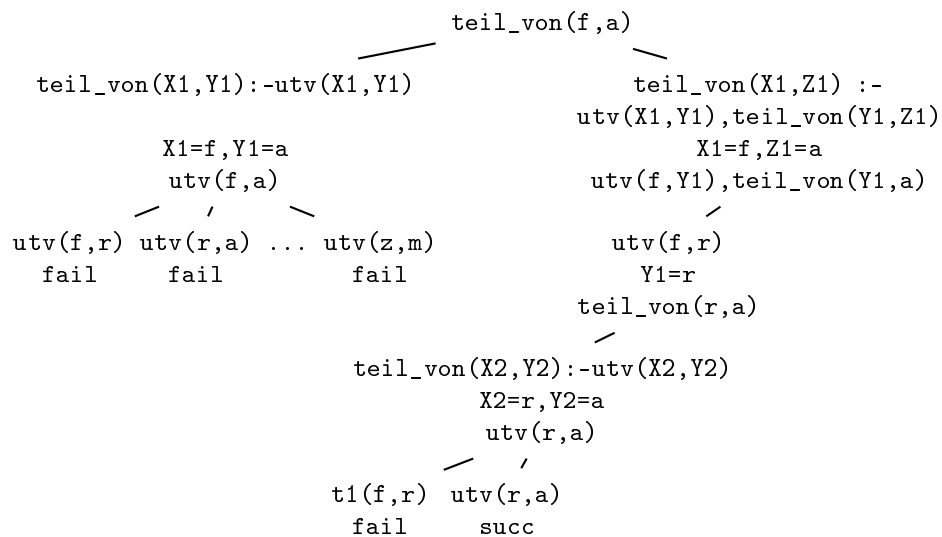
```

% teil_von(?Teil,?Komplexes-Teil)
% Teil und Komplexes-Teil sind Namen, so dass Teil
% Bestandteil von Komplexes-Teil ist
teil_von(X,Y):-utv(X,Y).
teil_von(X,Z):-utv(X,Y),teil_von(Y,Z).
utv(felge,rad).
utv(rad,auto).
utv(motor,auto).
utv(kolben,motor).
utv(ventil,motor).
utv(zuendkerze,motor).
  
```

Transitive Relationen

- vollständig spezifizierte Anfrage: Relationstest `teil_von(+,+)`

```
?- teil_von(felge,auto).
```



Transitive Relationen

- sonstige Anfragen:

```
?- teil_von(bremse,auto).
   false.
?- teil_von(felge,X).
   X=rad ;
   X=auto.
?- teil_von(X,rad).
   X=felge.
?- teil_von(X,Y).
   X=felge, Y=rad ;
   ...
   X=zuendkerze, Y=auto.
```

1:m-Relationen

Funktionale Abhängigkeit

Eine zweistellige Relation $p(X, Y)$ heißt

- vom Typ 1:m, wenn $p(X, Y) \wedge p(Z, Y) \rightarrow X = Z$ allgemeingültig ist.
- vom Typ m:1, wenn $p(X, Y) \wedge p(X, Z) \rightarrow Y = Z$ allgemeingültig ist.

→ “funktionale” Spezifikation einer Argumentposition: Ist $p(X, Y)$ eine Relation vom Typ 1:m, dann gilt $X = f(Y)$.

→ 1:m- und m:1-Relationen sind Spezialfälle einer m:n-Relation

1:m-Relationen

- Beispiele für 1:m-Relationen:

- `mutter_von(Mutter, Tochter)`
- `alter_von(Jahre, Person)`

- Beispiele für m:1-Relationen:

- `utv(Obj1, Obj2)`
- `liegt_in(Stadt, Land)`

1:m-Relationen

Prädikatsdefinitionen sind im allgemeinen Fall vom Typ m:n.

- auch bei eindeutigen Relationen wird auf Anfrage immer nach alternativen Funktionswerten gesucht
 - unnötiger Suchaufwand
 - u.U. Terminierungsprobleme bei rekursiven Definitionen
 - möglicher Ausweg:
Abbruch der Suche nach dem ersten Resultat $\rightarrow !/0$ (cut)

Reflexive Relationen

Reflexivität

Eine zweistellige Relation p ist reflexiv, wenn

$$p(X, X)$$

allgemeingültig ist.

- Beispiel: Gleichheit, Unifizierbarkeit

$$X = Y \Leftrightarrow Y = X$$

Intensionale Relationsdefinitionen, die sich (indirekt) auf eine Gleichheitsrelation (z.B. Unifizierbarkeit) beziehen, sind standardmäßig reflexiv.

Reflexive Relationen

```
% schwester_von(?Schwester,?Person)
schwester_von(X,Y):-kind_von(X,Z),kind_von(Y,Z),weiblich(X).

% kind_von(?Kind,?Elternteil)
kind_von(susi,anne).
kind_von(jane,anne).
kind_von(karl,anne).

weiblich(susi).
weiblich(jane).
weiblich(anne).
maennlich(karl).

?- schwester_von(S,susi).
S=susi ;
S=jane.
```

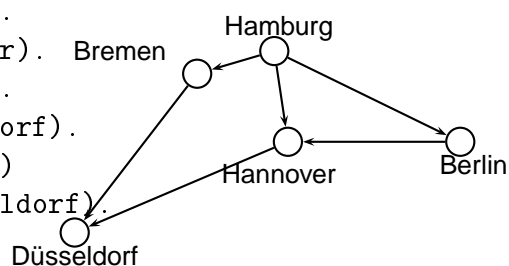
4.4 Anwendung: Wegplanung

Anwendungsbeispiel: Wegplanung

- Basis: Erreichbarkeitsrelation

```
% con(?Ort1,?Ort2)
% Ort1 und Ort2 sind Namen, so dass Ort2
% von Ort1 aus erreichbar ist
```

```
con(hamburg,berlin).
con(hamburg,hannover).
con(hamburg,bremen).
con(bremen,duesseldorf).
con(berlin,hannover)
con(hannover,duesseldorf).
```



Anwendungsbeispiel: Wegplanung

- Symmetrieeigenschaft: Eine extensional definierte, zweistellige Relation ist standardmäßig unsymmetrisch.

```
?- con(hamburg,bremen).
    true.
?- con(bremen,hamburg).
    false.
```

Anwendungsbeispiel: Wegplanung

- Symmetrie durch intensionale Spezifikation mit Hilfsprädikat

```
% con_sym(?Ort1,?Ort2)
% Ort1 und Ort2 sind Namen, so dass Ort1
% und Ort2 wechselseitig erreichbar sind
con_sym(X,Y):-con(X,Y).
con_sym(X,Y):-con(Y,X).
con(hamburg,berlin).
con(hamburg,hannover).
...

?- con_sym(hamburg,hannover).
    true.
?- con_sym(hannover,hamburg).
    true.
```

Anwendungsbeispiel: Wegplanung

- Transitivität durch intensionale Spezifikation mit Hilfsprädikat

```
% con_trans(?Ort1,?Ort2)
% Ort1 und Ort2 sind Namen, so dass zwischen Ort1 und Ort2
% eine (unsymmetrische) transitive Erreichbarkeit besteht
con_trans(X,Y):-con(X,Y).
con_trans(X,Y):-con(X,Z),con_trans(Z,Y).
con(hamburg,berlin).
con(hamburg,hannover).
...
```

Anwendungsbeispiel: Wegplanung

```
?- con_trans(hamburg,duesseldorf).
    true ;
    true ;
    true ;
    false.
?- con_trans(hamburg,X).
    X=berlin ;
    X=hannover ;
    X=bremen ;
    X=hannover ;
    X=duesseldorf ;
    X=duesseldorf ;
    X=duesseldorf ;
    false.
```

Anwendungsbeispiel: Wegplanung

- Kopplung von Transitivität und Symmetrie ???

```
% con_sym_trans(Ort1,Ort2)
% Ort1 und Ort2 sind Namen, so dass zwischen Ort1 und Ort2
% eine symmetrische und transitive Erreichbarkeit besteht
con_sym_trans(X,Y):-con(X,Y).
con_sym_trans(X,Y):-con(Y,X).
con_sym_trans(X,Y):-con(X,Z),con_sym_trans(Z,Y).
con_sym_trans(X,Y):-con(Z,X),con_sym_trans(Z,Y).
...

?- con_sym_trans(hannover,bremen).
    true ;
    true ;
    true ;
    ...
?- con_sym_trans(duesseldorf,berlin).
    ...
```

Auch die Ermittlung der Verbindung zwischen zwei Orten in einem Wegenetz ist ein nichtdeterministisches Suchproblem. Die Suchraumgröße ergibt sich hier in Abhängigkeit von der Anzahl der im Wegenetz eingetragenen Verbindungen. Untersuchen Sie selbst anhand der kompletten Implementierung (symmetrisches und transitives Verbundenheitsprädikat) Aufwandsfragen und mögliche Terminierungsprobleme!

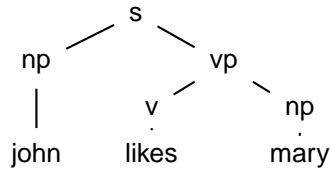
Fazit: Deduktive Datenbanken

- Berechnung einer Relation statt Aufzählung
- Vermeiden von Redundanz
- Spezifikation von Relationen über unendlichen Domänen
- transitive Hülle einer Relation ist berechenbar
- aber: Terminierungsprobleme bei rekursiven Definitionen
- Effizienzprobleme bei persistenter Datenhaltung
- eingeschränktes Modell ist Bestandteil von SQL-99

[→ GDB](#)

5 Rekursive Datenstrukturen

Rekursive Datenstrukturen



Mit der Möglichkeit zur rekursiven Definition von Prädikaten ist bereits der wichtigste Schritt auf dem Weg zu einer berechnungsuniversellen Programmiersprache getan. Es fehlen allerdings noch geeignete Datenstrukturen, über denen die Rekursion operieren kann. Diese werden in diesem Abschnitt in Form der rekursiv eingebetteten Terme eingeführt.

Der Abschnitt wird eingeleitet mit einem Überblick über die bisher definierten Syntaxelemente der Sprache Prolog, um die noch zu ergänzenden Konstruktionen einordnen zu können. Ein solches noch nicht behandeltes Konstrukt ist der Bereich der *Operatorausdrücke*, die ein sehr leistungsfähiges Hilfsmittel zur Konstruktion eigener syntaktischer Varianten für bestehende Prädikate darstellen. Sie ermöglichen damit eine einfache Anpassung der Sprachsyntax an die Bedürfnisse unterschiedlicher Nutzergruppen. Weitere noch erforderliche Erweiterungen der Syntax betreffen die Arbeit mit *arithmetischen Ausdrücken* (ein Beispiel für die Anwendung von Operatoren), sowie *Listen* als wichtigste rekursive Datenstruktur. Diese Konstrukte werden in gesonderten Abschnitten behandelt.

Überblick Prolog-Syntax

Datenbasis	::=	Klausel {Klausel}
Prozedur	::=	Klausel {Klausel}
Ziel	::=	elementares_Ziel komplexes_Ziel
elementares_Ziel	::=	Struktur
komplexes_Ziel	::=	elementares_Ziel (';' ',') Ziel
Klausel	::=	(Fakt Ziel Regel) '.'
Fakt	::=	Struktur
Regel	::=	Kopf ':-' Körper
Kopf	::=	Struktur
Körper	::=	Ziel
Struktur	::=	Name['(' Term{',' Term}')'] Operatorausdruck
Term	::=	Konstante Variable Struktur Liste
Konstante	::=	Zahl Name gequoteter_Name
Name	::=	Kleinbuchstabe {alphanumerisches_Symbol}
Variable	::=	benannte_Variable unbenannte_Variable
benannte_Variable	::=	Großbuchstabe {alphanumerisches_Symbol}
unbenannte_Variable	::=	'_'

5.1 Eingebettete Strukturen

Eingebettete Strukturen

- Strukturen können rekursiv eingebettet sein
- Prolog-Datenbanken müssen nicht in 1. Normalform sein (NF²)

```
angebot(produkt('tm-416'),
        kategorie(radio),
        status(am_lager),
        preis(euro(100))).
```

- wichtiger Spezialfall: rekursive Selbsteinbettung einer Struktur

```
s(s(s(0)))

dp(a,dp(b,dp(c,nil)))
```

Eingebettete Strukturen

- z.B. Verwendung zum typsicheren Programmieren

```
angebot(produkt(radio),euro(100)).

?- anbot(Produkt,euro(Preis)), Preis < 200.
Produkt=produkt(radio),Preis=100.
```

- Datenabstraktion: Zusammenfassung von elementaren zu komplexen Strukturen

```
adresse(plz(22527),
        strasse('Vogt-Köln-Straße'),
        ort('Hamburg'))
```

Eingebettete Strukturen

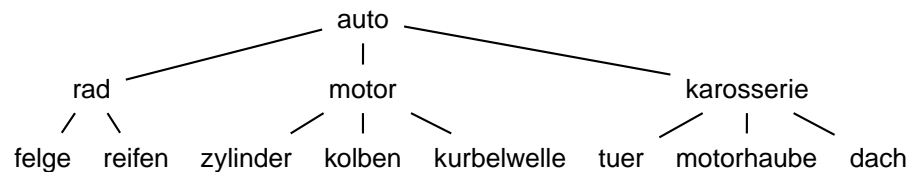
- rekursiv eingebettete Strukturen entsprechen einer Baumstruktur

Strukturen	⇔	Verzweigungen des Baumes (Knoten)
Prädikatsnamen	⇔	Knotenmarkierungen
äußerster Prädikatsname	⇔	Wurzelknoten des Baumes
Argumente	⇔	Kanten
Atome als Argument	⇔	Blattknoten

Baumstrukturen

- Teil-von-Hierarchien

```
auto(rad(felge,reifen),
      motor(zylinder,kolben,kurbelwelle),
      karosserie(tuer,motorhaube,dach)).
```



Teil-von-Hierarchien sind bereits im vorangegangenen Abschnitt als Beispiel für die Arbeit mit Baumstrukturen aufgetreten. Hier liegt nun eine alternative Repräsentation als eingebettete Struktur vor. Beide Implementationsvarianten sind gleichermaßen universell: Sie erlauben die Kodierung allgemeiner Graphstrukturen, von denen Bäume nur ein Spezialfall sind. Allerdings erfordert die Repräsentation von allgemeinen Graphen mit Hilfe von Termstrukturen den Rückgriff auf Koreferenz durch die Verwendung identischer Variablen.

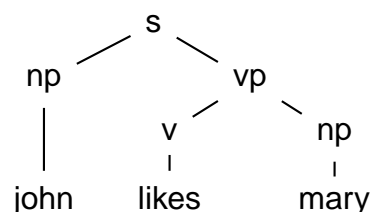
Als weitere Beispiele für die Verwendung von Bäumen in der Informatik werden hier Syntaxdiagramme für natürliche Sprache bzw. für Programmiersprachenkonstrukte vorgestellt. Sie dienen beide als Grundlage für eine semantische Interpretation der betreffenden sprachlichen Ausdrücke.

Die Hinzunahme rekursiv eingebetteter Strukturen zur Syntax der Programmiersprache erfordert schließlich eine erneute Erweiterung des Unifikationsbegriffs, der nunmehr aber in seiner endgültigen Form vorliegt.

Baumstrukturen

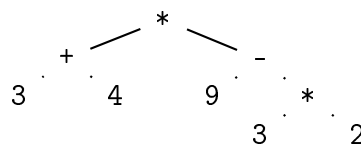
- Syntaxbäume in der natürlichen Sprache

```
s(np(john),
  vp(v(likes),
     np(mary))).
```



Baumstrukturen

- Syntaxbäume für Programmiersprachen
 - Arithmetischer Ausdruck: $(3 + 4) * (9 - 3 * 2)$
 - in Präfixnotation:
 - $*(+(3,4),$
 - $- (9,$
 - $*(3,2)))$



Unifikation III

Unifikation von komplexen Strukturen

Ermitteln einer solchen Variablensubstitution, die die Gleichheit von *zwei Strukturen* herstellt.

- Vergleichskriterien:
 - identischer Prädikatsname
 - gleiche Stelligkeit
 - *rekursive* Unifikation der Argumente

Struktur / Struktur	Unifikation	$a(X, c) = a(b, Y)$
Variable / Struktur	Instanziierung	$X = a(b, c)$
Struktur / Variable	Instanziierung	$d(e, f) = Y$
Variable / Variable	Koreferenz	$X = Y$

Unifikation III

- Beispiele für die Unifikation von Strukturen

Struktur ₁	Struktur ₂	Variablensubstitution
$a(X, c)$	$a(b, Y)$	$X=b, Y=c$
$a(X, X)$	$a(d(Y), d(b))$	$X=d(b), Y=b$
$a(X, a(X))$	$a(Y, Y)$	$X=Y=a(X)$

- Problemfall: Koreferenz über mehrere Rekursionsebenen hinweg
→ Aufbau unendlicher durch Unifikation endlicher Strukturen.

?- $a(X)=X$.

$X = a(X)$.

?- $a(X)=X, \text{write}(X)$.

$a(**)$

$X = a(X)$

?- $a(X)=X, X=a(A), X=a(a(B)), X=a(a(a(C)))$.

$X = A, A = B, B = C, C = a(C)$.

Unifikation III

- Pragmatik: Unifikation ist gleichzeitig
 - a) Testoperator:
Sind die zu unifizierenden Strukturen miteinander verträglich?
 - b) Accessor (Selektor, Observer):
Aus welchen Bestandteilen setzt sich eine komplexe Struktur zusammen?
 - c) Konstruktor:
Setze eine komplexe Struktur aus gegebenen Bausteinen zusammen!

5.2 Arithmetik, relational

Arithmetik, relational

- Axiomatisierung der Arithmetik für natürliche Zahlen
- Giuseppe PEANO, italienischer Mathematiker, 1858-1932

natürliche Zahl

Anfangselement: 0 ist eine natürliche Zahl.

Nachfolgerbeziehung: Jede natürliche Zahl n besitzt einen unmittelbaren Nachfolger $s(n)$, der ebenfalls eine natürliche Zahl ist.

Domänenabschluss: Nur die so gebildeten Objekte sind natürliche Zahlen.

→ rekursive Definition!

Die Beschäftigung mit der PEANO-Arithmetik bietet uns nicht nur einen guten Einblick in die Technik der Axiomatisierung, sondern gleichzeitig ein schönes und sehr einfaches Beispiel für die Definition rekursiver Prädikate über rekursiven Datenstrukturen. Eine für praktische Zwecke weitaus besser geeignete Behandlung der Arithmetik folgt dann in der zweiten Hälfte dieses Abschnitts.

Arithmetik, relational

- Daten: PEANO-Terme
- Syntax: $\text{PEANO-Term} ::= '0' \mid 's' (\text{PEANO-Term})$
- Semantik:

Term	arithmetische Interpretation
0	0
s(0)	1
s(s(0))	2
s(s(s(0)))	3
...	...

- PEANO-Terme sind kein Prolog-Konstrukt, sondern eine spezielle Form von Prolog-Strukturen

Arithmetik, relational

- Programme: Prädikate über PEANO-Termen
- z.B. Typtest für PEANO-Zahlen: `peano/1`

```
% peano(+Term)
% Term ist ein Peano-Term
peano(0).                % Rekursionsabschluss
peano(s(X)) :- peano(X). % Rekursionsschritt
```

→ Rekursive Definition

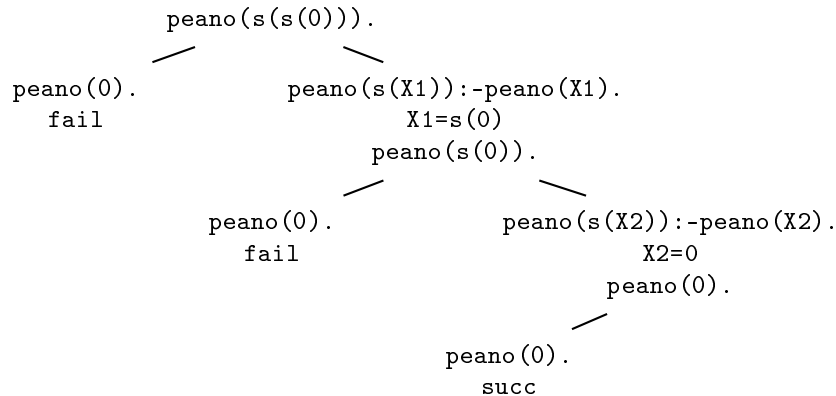
Jede rekursive Definition besteht aus den beiden Elementen Rekursionsabschluss und Rekursionsschritt, wobei der Rekursionsabschluss für die Terminierung und der Rekursionsschritt für den Berechnungsfortschritt verantwortlich ist.

In unserem speziellen Fall besteht der Berechnungsfortschritt darin, dass bei jedem Aufruf von `peano/1` eine Ebene der Peano-Struktur abgebaut wird. Die Berechnung endet erfolgreich, wenn auf diese Weise das Startelement 0 gefunden werden kann (Rekursionsabschluss).

Bei unterspezifiziertem Aufruf werden sukzessiv tiefer geschachtelte unterspezifizierte Terme (mit uninstantiierten Variablen) aufgebaut, die unter Verwendung des Rekursionsabschlusses dann vollständig instantiiert werden. Auf diese Weise werden (entsprechend der Definition der natürlichen Zahlen) unendlich viele unterschiedliche Ergebnisterme erzeugt.

Beispielfragen

- vollständig instantiierter Aufruf: `peano(+)`



Beispielfragen

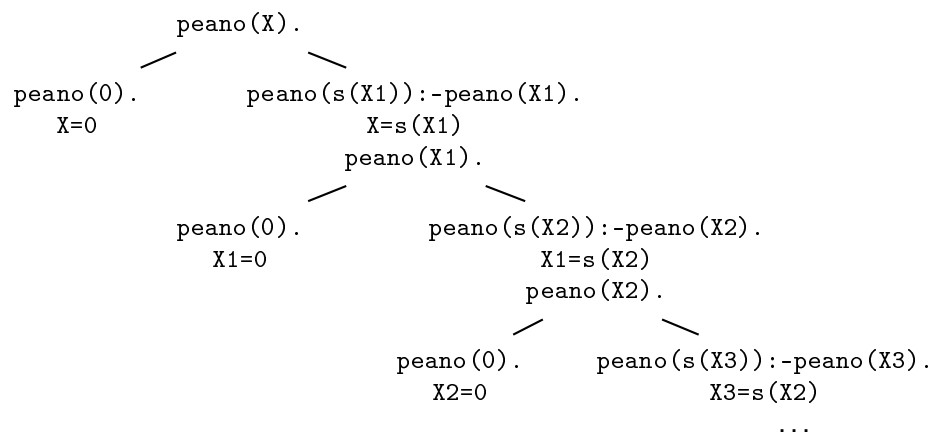
- vollständig instanzierter Aufruf

?- peano(s(s(0))).	#1
peano(0).	fail
peano(s(X1)) :- peano(X1).	succ(X1=s(0))
?- peano(s(0)).	#2
peano(0).	fail
peano(s(X2)) :- peano(X2).	succ(X2=0)
?- peano(0).	#3
peano(0).	succ
true.	
?- peano(susi).	#1
peano(0).	fail
peano(s(X1)) :- peano(X1).	fail
false.	

- Konsumtion einer rekursiven Struktur auf der Argumentposition beim rekursiven Abstieg

Beispielfragen

- unerspezifizierter Aufruf: `peano(-)`



- Prädikatsschema kann verallgemeinert werden: `peano(?Term)`

Beispielanfragen

- Unterspezifizierter Aufruf

<code>?- peano(X).</code>	<code>#1</code>
<code> peano(0).</code>	<code>succ(X=0)</code>
<code>X=0 ;</code>	
<code> peano(s(X1)):-peano(X1).</code>	<code>succ(X=s(X1))</code>
<code> ?- peano(X1).</code>	<code>#2</code>
<code> peano(0).</code>	<code>succ(X1=0)</code>
<code>X=s(0) ;</code>	
<code> peano(s(X2)):-peano(X2).</code>	<code>succ(X1=s(X2))</code>
<code> ?- peano(X2).</code>	<code>#3</code>
<code> peano(0).</code>	<code>succ(X2=0)</code>
<code>X=s(s(0)) ;</code>	
<code>...</code>	

- Konstruktion einer rekursiven Struktur auf der Argumentposition beim rekursiven Abstieg

Rekursion (2)



BAUER/GOOS, 1982

- Rekursion (2): selbstbezügliche Definition eines Prädikats
 - unter *Konsumtion einer Datenstruktur* auf einer Argumentposition und
 - mit einer Terminierungsbedingung (Rekursionsabschluss) für diese Struktur
- wichtiger Spezialfall: Endrekursion

Die Konsumtion einer Datenstruktur beim rekursiven Abstieg ist eine wichtige Voraussetzung für die Definition terminierungssicherer Prädikate. Allerdings kann die Konsumtion für sich allein genommen noch nicht das Anhalten der Berechnung garantieren. Dies wird erst durch einen geeigneten Rekursionsabschluss erreicht. Zur Konsumtion kommen nicht nur rekursive Datenstrukturen in Frage, sondern auch numerische Werte, insbesondere die natürlichen Zahlen. Dies ist keine allzu überraschende Feststellung mehr, nachdem wir gerade die sehr enge konzeptuelle Verwandtschaft zwischen diesen beiden Datentypen feststellen konnten.

Endrekursion

Endrekursion

- Berechnung erfolgt nur beim rekursiven Abstieg.
- Endergebnis ist am Rekursionsabschluss bereits vollständig bekannt
- Zwischenergebnisse müssen nicht mehr auf dem Stack der abstrakten Maschine aufbewahrt werden
- Compilation in effizienten (weil iterativen) Code möglich
- Endrekursion liegt vor, wenn der rekursive Aufruf das letzte Teilziel im Rekursionsschritt ist.

Arithmetik, relational

- Vergleich von PEANO-Zahlen: `lt(Peano_Zahl1, Peano_Zahl2)` (less than)
- gewünschtes Verhalten:

	Ziel	Resultat		Ziel	Resultat
1	<code>lt(0,s(0))</code>	<code>true.</code>	4	<code>lt(0,0)</code>	<code>false.</code>
2	<code>lt(0,s(s(0)))</code>	<code>true.</code>	5	<code>lt(s(0),0)</code>	<code>false.</code>
3	<code>lt(s(0),s(s(0)))</code>	<code>true.</code>	6	<code>lt(s(s(0)),s(0))</code>	<code>false.</code>

- rekursive Definition

```
% lt(?Term1,?Term2)
% Term1 und Term2 sind Peano-Terme, so dass Term1
% kleiner als Term2
lt(0,s(_)).
lt(s(X),s(Y)):-lt(X,Y).
```

Diese Definition kann man sich leicht veranschaulichen, wenn man sich zwei Behälter vorstellt, die beide Kugeln enthalten. Um zu ermitteln, in welchem Behälter sich weniger Kugeln befinden, entnehmen wir in jedem Schritt jeweils eine Kugel aus jedem Behälter. Derjenige Behälter, in dem sich zuerst keine Kugeln mehr befinden, enthielt auch ursprünglich weniger Kugeln als der andere.

Beispielfragen

- vollständig spezifizierte Aufrufe: Konsumtion von rekursiven Datenstrukturen auf beiden Argumentpositionen
- unterspezifizierte Aufrufe

```

?- lt(X,s(s(0))).          #1
   lt(0,s(_)).             succ(X=0)
X=0 ;
   lt(s(X1),s(Y1)):-lt(X1,Y1). succ(X=s(X1),Y1=s(0))
?- lt(X1,s(0)).            #2
   lt(0,s(_)).             succ(X1=0)
X=s(0) ;
   lt(s(X2),s(Y2)):-lt(X2,Y2). succ(X1=s(X2),Y2=0)
?- lt(X2,0).               #3
   lt(0,s(_)).             fail
   lt(s(X3),s(Y3)):-lt(X3,Y3). fail
BT #2
BT #1
false.

```

Beispielfragen

- **unterspezifizierte Aufrufe:**

```

?- lt(s(s(0)),X).          #1
   lt(0,s(_)).             fail
   lt(s(X1),s(Y1)):-lt(X1,Y1). succ(X1=s(0),X=s(Y1))
?- lt(s(0),Y1).            #2
   lt(0,s(_)).             fail
   lt(s(X2),s(Y2)):-lt(X2,Y2). succ(X2=0,Y1=s(Y2))
?- lt(0,Y2).               #3
   lt(0,s(_)).             succ(Y2=s(_))
X=s(s(s(_))) ;
   lt(s(X3),s(Y3)):-lt(X3,Y3). fail
BT #2
BT #1
false.

```

unterspezifiziertes Resultat

generische Beschreibung einer unendlich großen Resultatsmenge Das Ziel wird aufgrund der vorhandenen Information soweit wie möglich instanziiert.

Das Berechnungsergebnis für diesen Aufruf enthält noch eine uninstanziierte Variable, die natürlich wiederum durch beliebige Terme substituiert werden kann. Damit steht der Ergebnisausdruck für eine Menge von Einzellösungen, deren Kardinalität in diesem Fall sogar unendlich groß ist. Beschränkt man sich bei der Substitution wieder auf *PEANO*-Terme, so kann das Resultat als "Alle natürlichen Zahlen größer oder gleich 3" interpretiert werden.

Wir sehen, dass nicht nur Prädikatsaufrufe sondern auch Datenstrukturen Variable enthalten können und also partiell unterspezifiziert sind. Damit verfügen wir über die einzigartige Möglichkeit, mit Beschreibungen zu operieren, deren Details im Einzelnen (noch) gar nicht bekannt sind oder — etwa wegen der unendlichen Kardinalität — prinzipiell nicht bekannt sein können. Bei Bedarf lassen sich derartige Terme dann im Verlauf der weiteren Berechnung durch Variableninstanzierung mit zusätzlicher Information anreichern.

Indirekte Rekursion

- Test auf gerade/ungerade PEANO-Zahlen: `even(Peano_Zahl)`, `odd(Peano_Zahl)`
- gewünschtes Verhalten:

	Ziel	Resultat		Ziel	Resultat
1	<code>even(0)</code>	<code>true.</code>	4	<code>odd(0)</code>	<code>false.</code>
2	<code>even(s(0))</code>	<code>false.</code>	5	<code>odd(s(0))</code>	<code>true.</code>
3	<code>even(s(s(0)))</code>	<code>true.</code>	6	<code>odd(s(s(0)))</code>	<code>false.</code>
4	<code>even(s(s(s(0))))</code>	<code>false.</code>	6	<code>odd(s(s(s(0))))</code>	<code>true.</code>

- rekursive Definitionen

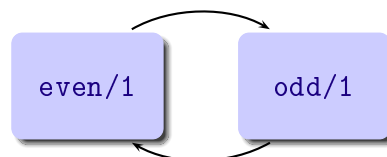
```
% even(?Term), odd(?Term)
% Term ist ein geradzahliger/ungeradzahliger Peano-Term
even(0).
even(s(P)) :- odd(P).

odd(s(P)) :- even(P).
```

Indirekte Rekursion

indirekte Rekursion

Der rekursive Aufruf erfolgt nicht unmittelbar in der Klausel, die das Prädikat definiert, sondern in einem zweiten Prädikat, das von Ersterem aufgerufen wird.



Indirekte Rekursion kann auch über mehr als eine Zwischenstufe erfolgen

Arithmetik, relational

- Addition von PEANO-Zahlen: `add(Summand1, Summand2, Summe)`
- gewünschtes Verhalten:

1	<code>add(0,0,0)</code>	<code>true.</code>	5	<code>add(0,s(0),0)</code>	<code>false.</code>
2	<code>add(0,s(0),s(0))</code>	<code>true.</code>	6	<code>add(s(0),s(0),s(0))</code>	<code>false.</code>
3	<code>add(s(0),0,s(0))</code>	<code>true.</code>			
4	<code>add(s(s(0)),s(0),s(s(s(0))))</code>	<code>true.</code>			

- rekursive Definition:

```

% add(?Summand1,?Summand2,?Summe)
% Summand1, Summand2 und Summe sind Peano-Terme,
% so dass gilt: Summand1 + Summand2 = Summe
add(0,X,X).                               % Rekursionsabschluss
add(s(X),Y,s(R)):-add(X,Y,R).            % Rekursionsschritt

```

Hier werden auf dem ersten Argument und dem dritten Argument systematisch PEANO-Strukturen konsumiert, bis diese vollständig verbraucht sind. Man kann sich die Berechnung auch als eine Waage vorstellen, bei der sich auf einer Waagschale zwei Behälter mit Kugeln befinden und ein dritter Behälter auf der anderen Waagschale. Die Waage befindet sich im Gleichgewicht, wenn die Summe der Kugeln in den ersten beiden Behältern gleich der Anzahl der Kugeln im dritten Behälter ist. Das Gleichgewicht bleibt erhalten, wenn auf beiden Seiten der Waage gleich viele Kugeln entnommen werden. Ungewohnt ist vielleicht die dreistellige (relationale) Behandlung der Addition, wenn man von der Mathematik her vor allem den funktionalen Umgang gewöhnt ist:

$$summe = f(summand_1, summand_2)$$

Da Relationen jedoch keinen "Funktionswert" besitzen, ist die dritte Argumentstelle für die Verwaltung des Berechnungsergebnisses (der Summe) notwendig. Allerdings werden wir gleich sehen, dass bei der relationalen Behandlung die Summe bei Bedarf natürlich auch die Eingabeinformation sein kann und nicht unbedingt immer das Berechnungsergebnis darstellen muss.

Beispielanfragen

- unterspezifizierte Aufrufe: `add(+,+, -)` z.B. `2+1=Summe`

```

?- add(s(s(0)),s(0),Sum).                #1
   add(0,X1,X1).                          fail
   add(s(X1),Y1,s(R1)):-add(X1,Y1,R1).    succ(X1=s(0),Y1=s(0),Sum=s(R1))
?- add(s(0),s(0),R1).                    #2
   add(0,X2,X2).                          fail
   add(s(X2),Y2,s(R2)):-add(X2,Y2,R2).    succ(X2=0,Y2=s(0),R1=s(R2))
?- add(0,s(0),R2).                        #3
   add(0,X3,X3).                          succ(X3=s(0)=R2)
Sum=s(s(s(0))) ;
   add(s(X3),Y3,s(R3)):-add(X3,Y3,R3).    fail
BT #2
BT #1
false.

```

Beispielanfragen

- **unterspezifizierte Aufrufe:** `add(+, -, +)` z.B. `2+Smd=3`

```
?- add(s(s(0)), Smd, s(s(s(0)))) .      #1
add(0, X1, X1) .                        fail
add(s(X1), Y1, s(R1)) :- add(X1, Y1, R1) . succ(X1=s(0), Smd=Y1,
                                           R1=s(s(0)))

?- add(s(0), Y1, s(s(0))) .              #2
add(0, X2, X2) .                        fail
add(s(X2), Y2, s(R2)) :- add(X2, Y2, R2) . succ(X2=0, Y1=Y2, R2=s(0))
?- add(0, Y2, s(0)) .                    #3
add(0, X3, X3) .                        succ(Y2=X3=s(0))

Smd=s(0) ;
add(s(X3), Y3, s(R3)) :- add(X3, Y3, R3) . fail
BT #2
BT #1
false.
```

Beispielanfragen

- **weitere unterspezifizierte Aufrufe:**

```
?- add(Smd, s(0), s(s(s(0)))) .      % Smd + 1 = 3
Smd=s(s(0)) .

?- add(Smd1, Smd2, s(s(0))) .        % Smd1 + Smd2 = 3
Smd1=0, Smd2=s(s(0)) ;
Smd1=s(0), Smd2=s(0) ;
Smd1=s(s(0)), Smd2=0.
```

Exkurs: Programmiermethodik

Testen eines Prädikats (3): Regressionstests

1. Speichern aller Testfälle als Fakten
2. Ableiten der Prädikatsdefinition aus den Testbeispielen
3. teilautomatisierte Überprüfung mit einfacher Testroutine
4. Testen der unterspezifizierten Aufrufe
5. sukzessives Erweitern der Testdatenbank

- **positive und negative Testfälle, z.B.**

```
test_daten(add(0,0,0)) .                % positiv
test_daten(add(0,s(0),s(0))) .          % positiv
test_daten(add(s(0),0,s(0))) .          % positiv
test_daten(add(s(s(0)),s(0),s(s(s(0))))) . % positiv
test_daten(add(0,s(0),0)) .              % negativ
test_daten(add(s(0),s(0),s(0))) .        % negativ
```

Exkurs: Programmiermethodik

- teilautomatisierte Überprüfung mit einfacher Testroutine, z.B.

```
% Testroutine
% ruft Testbeispiele der Form test_daten(call) auf

test :- test_daten(X), write(X), diagnose(X), fail.

diagnose(X) :- X, write_ln(' yes').
diagnose(X) :- not(X), write_ln(' no').

?- test.
add(0, 0, 0) yes
add(0, s(0), s(0)) yes
add(s(0), 0, s(0)) yes
add(s(s(0)), s(0), s(s(s(0)))) yes
add(0, s(0), 0) no
add(s(0), s(0), s(0)) no
false.
```

Achtung: Hier wird nur eine einzige Instanziierungsvariante getestet! Für unterspezifizierte Aufrufe ist ein teilautomatisierter Test nur in Ausnahmefällen geeignet. Versuchen Sie sich klar zu machen, warum das so ist!

Auf der anderen Seite zeigt das kleine Programm sehr schön, wie man sich dank der formalen Äquivalenz von Programmen und Daten im logikbasierten Paradigma mit sehr einfachen Mitteln recht leistungsfähige Programmierwerkzeuge schaffen kann. Modifizieren Sie sich das Programm nach Ihren individuellen Wünschen. Schreiben Sie sich Ihr privates "Punit"!

5.3 Operatorstrukturen

Operatorstrukturen

Operatoren

- erlauben es dem *Nutzer*, die Syntax der Sprache zu modifizieren:
 - übersichtliche Schreibweise
 - Nähe zu etablierten Notationen (z.B. Mathematik)

- sind syntaktische Varianten für ein- oder zweistellige Strukturen:

$$A \text{ op } B \Leftrightarrow \text{op}(A, B)$$

- haben keine Semantik und definieren keine eigenständigen Prädikate!

Operatoren

- Syntax: $\text{Operatorausdruck} ::= \text{Präfixoperator} \mid \text{Infixoperator} \mid \text{Postfixoperator}$
 $\text{Präfixoperator} ::= \text{Operator Operand}$
 $\text{Infixoperator} ::= \text{Operand Operator Operand}$
 $\text{Postfixoperator} ::= \text{Operand Operator}$
- Operortyp:

Präfixoperator	Infixoperator	Postfixoperator
fx	yfx	xf
lg 10	3 + 4	4!

Operatoren

- Operatorpräzedenz

→ SE I

- numerische Angabe zur Disambiguierung von Operatorausdrücken mit mehr als einem Operator
 $a + b * c \equiv a + (b * c)$
- Wertebereich ist implementationsabhängig (1 ... 1200)
- geringere Werte binden stärker:

+	-	*	/
500	500	400	400

Operatoren

- symmetrische Operatoren (nicht-assoziativ): $\text{xfx} \text{ xf fxf}$
- keine Operatoreinbettung möglich
- $1 < 2 < 3$ ist syntaktisch unzulässig
- Beispiel: `liebt/2`
- Verwendung als Klauselkopf

```
:-op(300,xfx,liebt).
hans liebt geld.
liebt(hans,susi).
```

```
?- listing.
    hans liebt geld.
    hans liebt susi.
    true.
```

Operatoren

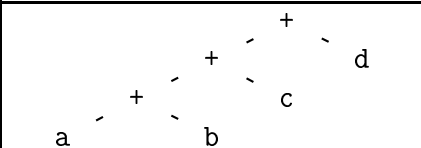
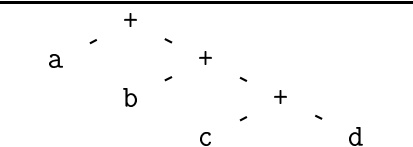
- Verwendung als Klauselkörper

```
?- X liebt susi.
   X=hans.
```

```
?- liebt(X,geld).
   X=hans.
```

Spezielle Operatoren

- asymmetrische Operatoren erlauben rekursive Einbettung
- Beispiel: $a + b + c + d$

links-assoziativ	rechts-assoziativ
yfx yf	xfy fy
'+'('+'('+'(a,b),c),d)	'+'(a,'+'(b,'+'(c,d))))
	

Spezielle Operatoren

	Präzedenz	Typ	Operator
Klauselstruktur	1200	xfx	:-
	1100	xfy	;
	1000	xfy	,
Direktiven	1200	fx	:- ?-
Negation	900	fy	\+, not
Unifikation	700	xfx	= \=
strukturelle Identität	700	xfx	== \==
strukturelle Gleichheit	700	xfx	=@= \=@=
numerischer Vergleich	700	xfx	> < >= <= =\= =:=
struktureller Vergleich	700	xfx	@> @< @>= @<=
arithmetische Funktionen	500	yfx	+ -
	400	yfx	* / //
	300	xfx	mod
	200	xfy	^

Spezielle Operatoren

- Klauselsyntax ist Operatorausdruck

```
grossvater_von(G,E) :-
    vater_von(G,V),
    vater_von(V,E).
```

ist vollständig äquivalent zu

```
' :- ' (grossvater_von(G,E),
        ', ' (vater_von(G,V), vater_von(V,E))).
```

- Klammerstruktur von komplexen Zielen (Konjunktion und Disjunktion) bzw. komplexen arithmetischen Funktionen wird durch mehrstufige Präferenzen festgelegt

Spezielle Operatoren

- struktureller Vergleich: Standardordnung
 - Variable @< Atome (@< Zahl) @< Struktur
 - Atome: lexikalische Sortierung nach Zeichencode
 - Zahlen: Sortierung nach numerischem Wert (ohne Typunterscheidung)
 - Strukturen: Sortierung nach Funktor, Stelligkeit und Argumenten
- Beispiele

A @< a	true	a(b) @< b(c)	true
aaa @< abc	true	a(b) @< a(b,c)	true
a @< a(b)	true	a(b,c) @< a(b,d)	true

- wichtiger Spezialfall: lexikalischer Vergleich

Operatoren

- Strukturen als einzig verfügbare Basisrepräsentation der Logikprogrammierung
- Operatoren sind nur syntaktische Notationsvarianten.
- Programme und Daten sind syntaktisch nicht unterscheidbar
- Prädikatsdefinitionen und Prädikatsaufrufe können dynamisch erzeugt werden
- wechselseitige Umwandlung zwischen Daten und Programmen ist möglich
 - Aufruf von Daten als Programm: `call(+Struktur)`
 - Interpretation von Programmklauseln als Daten: `clause(?Head,?Body)`

5.4 Arithmetik, funktional

Arithmetik, funktional

- Operatorausdrücke und Strukturen stehen nur für sich selbst und haben im relationalen Verarbeitungsmodell keinen Wert
- arithmetische Relationen beschreiben Beziehung zwischen Operanden *und* gewünschtem Berechnungsergebnis

arithmetische Operatorstruktur	arithmetisches Prädikat
$+(2, 3)$	<code>add(2, 3, R) .</code>
$*(4, 6)$	<code>mult(4, 6, R) .</code>

- Arithmetische Ausdrücke müssen erst in eine Auswertungsumgebung gestellt werden
→ lokales funktionales Verarbeitungsmodell im relationalen Paradigma

Zwar ist es prinzipiell möglich, auf der Basis der PEANO-Axiome das Gebäude der Arithmetik logisch zu rekonstruieren, jedoch ist dies keine geeignete Grundlage für eine effiziente Behandlung arithmetischer Probleme. Dies gilt umso mehr, als die zugrundeliegende Hardware, arithmetische Operationen in extrem effizienter Weise unterstützt. Nur hat diese "Implementierung in Silizium" einen kleinen Nachteil: Sie ist funktional, damit einseitig gerichtet und fügt sich also nur schwer in den Rahmen der richtungsunabhängigen relationalen Programmierung ein. Aus diesem Grunde muss eine spezielle Auswertungsumgebung für funktionale Ausdrücke bereitgestellt werden.

Arithmetische Auswertungsumgebung

arithmetische Auswertungsumgebung

spezieller Infixoperator, der den (arithmetischen) Wert von funktionalen Ausdrücken ermittelt.

- Syntax: Auswertungsumgebung ::= Zahl 'is' Arithmetischer_Ausdruck
- Semantik: Der Wert des Ausdrucks auf der rechten Seite wird mit dem Ausdruck auf der linken Seite unifiziert.

Arithmetische Auswertungsumgebung

- Auswertung greift auf die numerischen Operationen der Basismaschine zurück
- Konsequenzen
 - linke Seite sollte keine Struktur, nur Variable oder numerische Konstante sein
 - die rechte Seite muss sich zu einem arithmetischen Wert auswerten lassen

- die rechte Seite darf Variable nur dann enthalten, wenn diese bereits instanziiert sind
- arithmetische Ausdrücke und Ordnungsprädikate sind nicht mehr richtungsunabhängig

Arithmetische Auswertungsumgebung

- `is/2` schlägt fehl, wenn
 - eine der Typrestriktionen für die Argumente verletzt ist
 - * z.B. linke Seite ist ein komplexer Term
 - linke Seite und Wert der rechten Seite nicht miteinander unifizieren
 - * z.B. Ungleichheit der numerischen Werte auf der rechten und der linken Seite
- `is/2` bricht mit Fehler ab, wenn
 - sich die rechte Seite nicht auswerten lässt, weil sie
 - * kein arithmetischer Ausdruck (function) ist
 - * uninstantiierte Variable enthält
- `is/2` ist mehr als die Ergibtanweisung imperativer Sprachen (`:=`)

Arithmetische Auswertungsumgebung

- Beispiele

<code>3 is 2 * 5 - 7</code>	<code>succ</code>
<code>X is 2 * 3 + 1</code>	<code>succ(X=7)</code>
<code>2 + 1 is 2 * 5 - 7</code>	<code>fail</code>
<code>X is 5, X is 2 * 6</code>	<code>fail</code>
<code>X is 1, X is X + 1</code>	<code>fail</code>
<code>X is 1, X is X * 1</code>	<code>succ(X=1)</code>
<code>X is Y + 1</code>	<code>ERROR</code>
<code>X is a</code>	<code>ERROR</code>

Weitere Auswertungsumgebungen

- arithmetische Vergleichsoperatoren: `:=` `<` `>`
- funktionale Auswertung auf beiden Argumentpositionen

```
?- 10 / 2 := 4 + 1.
true.
?- 10 - 2 := 4 * 1.
false.
```

- können nicht zur Variableninstanziierung verwendet werden

```
?- X ::= 2 * 3 - 1.
```

```
ERROR: Arguments are not sufficiently instantiated
```

Arithmetik

- hybride Sprache
 - Funktionale Auswertung in einer relationalen Umgebung
 - Restriktionen spiegeln die nichtdeklarative Semantik der imperativen Hardwarearchitektur wieder
- Ursache: Behandlung der Arithmetik mit außerlogischen Mitteln

Fakultät

- Induktive Definition (funktional)

$$n! = \begin{cases} 1 & \text{für } n = 0 \\ (n-1)! * n & \text{sonst} \end{cases}$$

Induktionsanfang
Induktionsschritt

- Übertragung in eine relationale Definition

$$\text{fak}(n, r) \leftarrow \begin{cases} r = 1 & \text{für } n = 0 \\ \text{fak}(n-1, r_1), r = r_1 * n & \text{sonst} \end{cases}$$

Fakultät

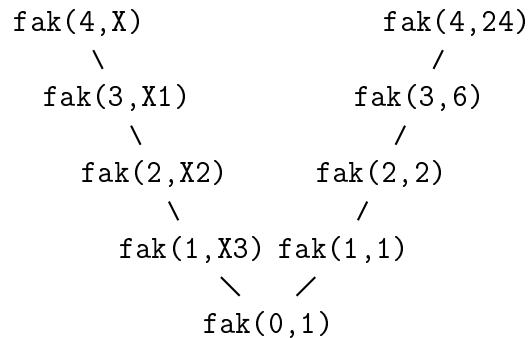
- Übertragung in ein relationales Programm

$$\text{fak}(n, r) \leftarrow \begin{cases} r = 1 & \text{für } n = 0 \\ \text{fak}(n-1, r_1), r = r_1 * n & \text{sonst} \end{cases}$$

```
% fak(+NatZahl,?Resultat)
% NatZahl und Resultat sind natuerliche Zahlen, so dass
% Resultat = fakultaet(NatZahl)
fak(0,1).                % Rekursionsabschluss
fak(N,FakN) :-           % Rekursionsschritt
    N > 0,                % nur fuer natuerliche Zahlen
    N1 is N - 1,          % Konsumtion des Eingabearguments
    fak(N1,FakN1),        % rekursiver Aufruf
    FakN is FakN1 * N.    % Aufbau des Resultats
```

Fakultät

- Rekursionsschema



- eigentliche Berechnung erfolgt nur beim rekursiven Aufstieg

→ Induktion, Spezialfall einer schlichten Rekursion

Ausgangspunkt einer induktiven Berechnung ist der Induktionsanfang. Dieser wird beim rekursiven Abstieg durch sukzessive Konsumtion des ersten Arguments gesucht und ist gefunden, wenn der Rekursionsabschluss erreicht wurde. Die eigentliche Berechnung der Fakultät erfolgt dann erst beim rekursiven Aufstieg (letzte Zeile der zweiten Klausel).

Eine Rekursion, bei der die Berechnungen ausschliesslich beim rekursiven Aufstieg vorgenommen werden, entspricht damit vollständig einer induktiven Schlusskette, wie sie etwa dem Beweisverfahren der vollständigen Induktion zugrunde liegt. Der Induktionsanfang in der induktiven Definition ist dann dem Rekursionsabschluss des Prädikats äquivalent.

Obwohl wir das Prädikat im Prädikatsschema als funktional gerichtet deklariert haben (das erste Argument muss instanziiert sein), kann es aber dennoch unspezifiziert aufgerufen werden. Überprüfen Sie das Verhalten in diesen Fällen. Im Abschnitt 7.2 werden wir die Mittel kennenlernen, die Zusicherungen des Prädikatsschemas wirklich umzusetzen.

Schlichte Rekursion

Schlichte Rekursion

Berechnung nur auf einem Ast der Rekursion

- maximal ein rekursiver Aufruf pro Klausel
- zwei wichtige Fälle
 1. aufsteigende Rekursion, Induktion
 2. Endrekursion, absteigende Rekursion, repetitive Rekursion, tail recursion

Schlichte Rekursion

aufsteigende Rekursion (Induktion)

- rekursiver Abstieg erfolgt nur, um den Induktionsanfang/Rekursionsabschluß zu finden
- alle wesentlichen Berechnungen erfolgen beim rekursiven Aufstieg
- z.B. Fakultät
- erlaubt oftmals richtungsunabhängige aber ineffiziente Implementierungen für arithmetische Probleme als generate-and-test-Verfahren.

Schlichte Rekursion

Endrekursion (absteigende Rekursion)

- die gesamte Berechnung erfolgt beim rekursiven Abstieg
- das Ergebnis liegt am Rekursionsabschluss vor
- der rekursive Aufstieg dient nur der Übermittlung des Ergebnisses
- z.B. wenn Induktionsanfang / Rekursionsabschluss vor Berechnung noch unbekannt ist
- effiziente iterative Implementierung (ohne Verwendung des Stacks) ist möglich

Augmentation

Umwandlung in Endrekursion

Schlicht rekursive Berechnungsvorschriften, die aufsteigend sind (d.h. alle induktiven Programme), können immer in absteigende (endrekursive) Verfahren umgewandelt werden, wenn ein weiteres Argument zur Übergabe von Berechnungsergebnissen bereitgestellt wird (Augmentation).

Akkumulator

Zusätzliche Argumentposition, auf der das Berechnungsergebnis sukzessiv aufgebaut wird.

Fakultät

- Erinnerung: aufsteigend rekursive (induktive) Rekursion

$$\text{fak}(n, r) \leftarrow \begin{cases} r = 1 & \text{für } n = 0 \\ \text{fak}(n-1, r_1), r = r_1 * n & \text{sonst} \end{cases}$$

```
% fak(+NatZahl,?Resultat)
% NatZahl und Resultat sind natuerliche Zahlen, so dass
% Resultat = fakultaet(NatZahl)
fak(0,1).                % Rekursionsabschluss
fak(N,FakN) :-           % Rekursionsschritt
    N > 0,                % nur fuer natuerliche Zahlen
    N1 is N - 1,          % Konsumtion des Eingabearguments
    fak(N1,FakN1),        % rekursiver Aufruf
    FakN is FakN1 * N.    % Aufbau des Resultats
```

Augmentation

- absteigend rekursive (endrekursive) Definition:

$$\text{fak}(n, r) \leftarrow \text{fak1}(n, 1, r)$$

$$\text{fak1}(n, a, r) \leftarrow \begin{cases} r = a & \text{für } n = 0 \\ \text{fak1}(n - 1, n * a, r) & \text{sonst} \end{cases}$$

```
% fak_er(+NatZahl,?Resultat)
% NatZahl und Resultat sind natuerliche Zahlen, so dass
% Resultat = fakultaet(NatZahl) [ueber Endrekursion]

fak_er(N,FakN) :- fak1(N,1,FakN).
fak1(0,X,X).        % Rekursionsabschluss
fak1(N,A,R) :-      % Rekursionsschritt
    N > 0,           % nur fuer nat. Zahlen
    N1 is N - 1,     % Konsumtion der Eingabe
    A1 is N * A,     % Akkumulation des Resultats
    fak1(N1,A1,R).   % rekursiver Aufruf
```

Augmentation

Vergleich

% fak(+NatZahl,?Resultat)	fak_er(+NatZahl,?Resultat)
% aufsteigend rekursiv	endrekursiv
	fak_er(N,FakN):-fak1(N,1,FakN).
fak(0,1).	fak1(0,X,X).
fak(N,FakN) :-	fak1(N,A,R) :-
N > 0,	N > 0,
N1 is N - 1,	N1 is N - 1,
fak(N1,FakN1),	A1 is N * A,
FakN is FakN1 * N.	fak1(N1,A1,R).

Augmentation

- Rekursionsschemata

```
fak(4,X)      fak(4,24) fak1(4,1,X)      fak1(4,1,24)
  \   X is X1*4   /           \   X = X1   /
fak(3,X1)     fak(3,6)  fak1(3,4,X1)     fak1(3,4,24)
  \  X1 is X2*3  /           \  X1 = X2   /
fak(2,X2)     fak(2,2)  fak1(2,12,X2)    fak1(2,12,24)
  \ X2 is X3*2 /           \  X2 = X3   /
fak(1,X3) fak(1,1)      fak1(1,24,X3) fak1(1,24,24)
  (X3 is X4*1)           (X3 = X4)
  fak(0,1)              fak1(0,24,24)
```

Im rekursiven Aufstieg finden keine Berechnungen mehr statt.
Er kann entfallen!

Weitere Beispiele

- Umwandlung von PEANO-Zahlen
- größter gemeinsamer Teiler
- Russische Bauernmultiplikation

Umwandlung von PEANO-Zahlen

```
% int2peano(+nicht_negative_Integerzahl,?Peanozahl)
```

```
int2peano(0,0).      % Rekursionsabschluss
int2peano(N,s(P)):-
  N > 0,
  N1 is N - 1,      % Konsumtion
  int2peano(N1,P).  % Rekursionsschritt
```

- Konsumtion der Integer-Zahl

Umwandlung von PEANO-Zahlen

- Rekursionsschema

```
i2p(4,X)      i2p(4,s(s(s(s(0))))))
  \   X = s(X1)   /
i2p(3,X1)     i2p(3,s(s(s(0))))
  \  X1 = s(X2)  /
i2p(2,X2)     i2p(2,s(s(0)))
  \  X2 = s(X3)  /
i2p(1,X3)     i2p(1,s(0))
  (X3 = s(X4))
  i2p(0,0)
```

int2peano/2 ist endrekursiv: Beim rekursiven Aufstieg finden keine Berechnungen mehr statt. Er kann entfallen!

Umwandlung von PEANO-Zahlen

- induktive Implementierung
- Ziel: Richtungsunabhängigkeit wieder herstellen
- Ansatz: generate-and-test

```
% int2peano(?Integer,?Peano) <-- Wunsch!
int2peano(0,0).      % Rekursionsabschluss
int2peano(N,s(P)) :-
    int2peano(N1,P), % Rekursionsschritt
    N is N1 + 1.     % Erhöhen des Zählers
```

Umwandlung von PEANO-Zahlen

- Terminierungsproblem

```
?- int2peano(N,s(s(0))).
   N = 2 ;
   No
?- int2peano(2,P).
   P = s(s(0)) ;
   ...
```

- hier (teilweise) Korrektur durch cut/0 möglich (aber eindeutiges Ergebnis auch bei vollständiger Unterspezifikation)

Umwandlung von PEANO-Zahlen

- Rekursionsschemata

```
i2p(X,s(s(s(0))))i2p(3,s(s(s(0))))
  \   X is X1 + 1   /
i2p(X1,s(s(0)))i2p(2,s(s(0)))
  \   X1 is X2 + 1   /
i2p(X2,s(0)) i2p(1,s(0))
  \   X2 is X3 + 1   /
  i2p(0,0)
      i2p(3,X)          i2p(3,s(s(s(0))))
        \   3 is N1 + 1, X = s(X1)   /
i2p(N1,X1)          i2p(2,s(s(0)))
        \N1 is N2 + 1, X1 = s(X2)/
i2p(N2,X2)          i2p(1,s(0))
        \N2 is N3 + 1, X2 = s(X3)/
        i2p(0,0)
```

Größter gemeinsamer Teiler

- rekursive Definition (funktional)

$$\text{ggt}(x, y) = \begin{cases} x & \text{wenn } x = y \\ \text{ggt}(y, x) & \text{wenn } x < y \\ \text{ggt}(x - y, y) & \text{sonst} \end{cases}$$

- Rekursionsabschluss liefert gewünschtes Berechnungsergebnis
- induktive Berechnung nicht möglich

- Übertragung in eine relationale Definition (Augmentation)

$$\text{ggt}(x, y, g) = \begin{cases} g = x & \text{wenn } x = y \\ \text{ggt}(y, x, g) & \text{wenn } x < y \\ x_1 = x - y, \text{ggt}(x_1, y, g) & \text{sonst} \end{cases}$$

Größter gemeinsamer Teiler

- Übertragung in ein relationales Programm

```
% ggt(+NatZahl1,+NatZahl2,-GGT)
% alle drei Argumente sind natuerliche Zahlen, mit GGT ist
% der groesste gemeinsame Teiler von NatZahl1 und NatZahl2
ggt(X,X,X).                % Rekursionsabschluss
ggt(X,Y,Z) :- X<Y, ggt(Y,X,Z). % Ordnen der Argumente
ggt(X,Y,Z) :- X>Y,          % Rekursionsschritt
               X1 is X-Y,
               ggt(X1,Y,Z).
```

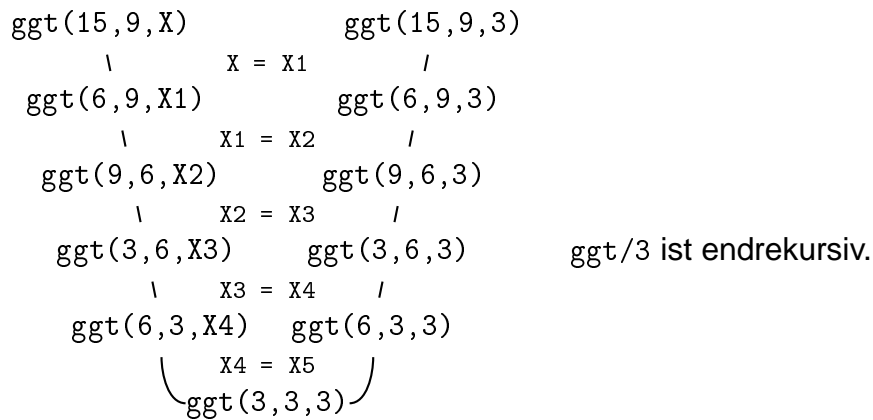
```
?- ggt(15,9,X).
   X=3.
```

```
?- ggt(2,3,X).
   X=1.
```

Die Definition des größten gemeinsamen Teilers hat bereits eine rekursive Grundstruktur, so dass die Übertragung in ein Logikprogramm besonders einfach ausfällt. Allerdings ist bei der Implementation noch der Übergang von der funktionalen Spezifikation in der Definition zur relationalen Spezifikation im Logikprogramm erforderlich: Es muss eine zusätzliche Argumentposition für das Berechnungsergebnis zur Verfügung gestellt werden. Die entsprechende Variable wird beim Rekursionsabschluss durch Koreferenz mit der ersten Argumentposition gebunden.

Größter gemeinsamer Teiler

- Rekursionsschema



Russische Bauernmultiplikation

- rekursive Definition (funktional)

$$\text{rbm}(f_1, f_2) = \begin{cases} f_1 & \text{falls } f_2 = 1 \\ f_1 + \text{rbm}(f_1, f_2 - 1) & \text{falls ungeradzahlig}(f_2) \\ \text{rbm}(2f_1, f_2/2) & \text{sonst} \end{cases}$$

- Berechnung erfolgt sowohl beim rekursiven Abstieg (Fall 3), als auch beim rekursiven Aufstieg (Fall 2)
- schlichte Rekursion ist nicht möglich

- Übertragung in eine relationale Definition (Augmentation)

$$\text{rbm}(f_1, f_2, p) =$$

$$\begin{cases} p = f_1 & \text{falls } f_2 = 1 \\ f_{21} = f_2 - 1, \text{rbm}(f_1, f_{21}, p), p = p_1 + f_1 & \text{falls ungeradzahlig}(f_2) \\ f_{11} = f_1 * 2, f_{21} = f_2/2, \text{rbm}(f_{11}, f_{21}, p) & \text{sonst} \end{cases}$$

Russische Bauernmultiplikation

- Übertragung in ein relationales Programm

```
% rbm(+Faktor1,+Faktor2,-Produkt)
% die drei Argumente sind nat. Zahlen groesser Null
% mit Faktor1 * Faktor2 = Produkt
rbm(F1,1,F1).                % Rekursionsabschluss
rbm(F1,F2,R) :- F2>1, odd(F2), % Rekursionsschritt
                F21 is F2-1,    % fuer unger. Faktor
                rbm(F1,F21,R1),
                R is R1+F1.
rbm(F1,F2,R) :- F2>1, even(F2), % Rekursionsschritt
                F11 is F1*2,    % fuer geraden Faktor
                F21 is F2/2,
                rbm(F11,F21,R).
```

Russische Bauernmultiplikation

- Hilfsprädikate

```
% odd(+NatZahl)
% NatZahl ist eine ungerade natuerliche Zahl
odd(X) :- X>0, 1 is X mod 2.
```

```
% even(+NatZahl)
% NatZahl ist eine gerade natuerliche Zahl
even(X) :- X>0, 0 is X mod 2.
```

?- odd(1).	?- even(2).
true.	true.
?- odd(2).	?- even(1).
false.	false.
?- odd(-1).	?- even(0).
false.	false.
	?- even(-2).
	false.

Das Grundschema der russischen Bauernmultiplikation ist dem Vorgehen bei der PEANO-Axiomatisierung sehr ähnlich. Ein Faktor wird durch sukzessives Halbieren konsumiert und

das Gesamtergebnis durch entsprechende Verdopplung des anderen Faktors konstant gehalten. Da die Division natürlich nur für gerade Zahlen ohne Rest möglich ist, müssen die entsprechenden Fehlbeträge beim “Abrunden” zu dem Gesamtergebnis hinzuaddiert werden. Die Definition ist in ihrer Struktur bereits rekursiv, so dass die Übertragung in ein Logikprogramm keine Probleme bereitet.

Russische Bauernmultiplikation

- Testläufe

?- rbm(4,7,R).	?- rbm(1,0,R).
R=28.	false.
?- rbm(24,157,R).	?- rbm(-5,4,R).
R=3768.	R=-20.
?- rbm(1,1,R).	?- rbm(4,-5,R).
R=1.	false.
?- rbm(0,1,R).	?- rbm(X,5,20).
R=0.	Instantiation Error
	?- rbm(5,X,20).
	Instantiation Error

Russisches Bauernmultiplikation

- Rekursionsschema

rbm(4,7,X)	rbm(4,7,28)	
\	X is X1 + 4	/
rbm(4,6,X1)	rbm(4,6,24)	
\	X1 = X2	/
rbm(8,3,X2)	rbm(8,3,24)	
\	X2 = X3 + 8	/
rbm(8,2,X3)	rbm(8,2,16)	
\	X3 = X4	/
rbm(16,1,16)		rbm/3 ist nicht endrekursiv.

Exkurs: Programmiermethodik

Testen eines Prädikats (4)

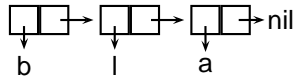
Beim systematischen Testen eines Prädikats müssen stets

- positive und negative Beispiele,
- alle Instanziierungsvarianten,
- alle wesentlichen Fallunterscheidungen
- wesentliche Verletzungen der Zusicherungen und
- wichtige Spezial- und Grenzfälle

berücksichtigt werden.

6 Verkettete Listen

Listenverarbeitung



Listen

- verkettete Listen sind die wichtigste rekursive Datenstruktur
- flexibel einsetzbar
 - Container für beliebige Datentypen (im Gegensatz zu Collections)
 - nicht längenbegrenzt
- sehr gute Grundlage für rekursive Prädikatsdefinitionen
 - spezielle Syntax zur bequemen Handhabung
- Vorgehen:
 - zuerst Beschränkung auf lineare Listen
 - später Erweiterung auf verzweigende Listen (Bäume)

6.1 Listennotation

Listennotation

- Iterative Elementaufzählung: `[a]` `[a, b]` `[a, b, c]`
- leere Liste: `[]`
- Rekursive Spezifikation:
 - Anfangselement: die leere Liste ist eine Liste
 - Nachfolgerbeziehung: ein *geordnetes Paar* aus einem Element und einer Liste ist eine Liste
 - Domänenabschluss: Nur die so gebildeten Objekte sind Listen

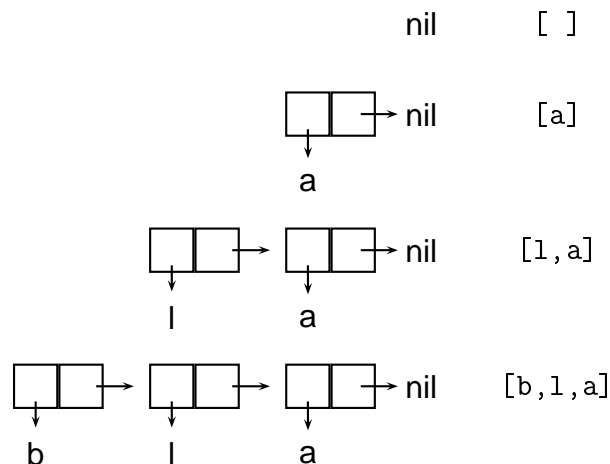
- Restlistenseparator/Listenkonstruktor:

`[Listenkopf | Restliste]`

- Gemischte Darstellung: `[Kopf1, Kopf2, Kopf3 | Restliste]`

Listennotation

- Implementation durch zweistellige Struktur



- vollständige Repräsentation der Zellenstruktur $[b|[1|[a|[]]]]$ ist synonym zu $[b, 1, a]$
- Eingebettete Listen: Listenelemente können Listen sein $[b, [b, [b, 1, a], 1, a], 1, a]$

Verkettete Listen sind intern als rekursiv eingebettete Terme unter Verwendung des Prädikats `./2` implementiert, wobei das erste Argument des Prädikats das erste Listenelement beschreibt und das zweite Argument die Restliste. Beachten Sie dabei vor allem die Asymmetrie des Restlistenseparators: Während der Operand nach dem Separator in einer echten Liste stets wieder eine Liste (oder der Listenendemarker) sein *muss*, unterliegt der Operand vor dem Separator keinerlei derartigen Einschränkungen. Anders herum: Wendet man den Separator auf eine lineare Liste von Atomen an, ist der erste Operand immer ein Atom, während der zweite eine Liste von Atomen (oder der Listenendemarker) ist.

Alternative Listenimplementationen, wie Sie sie vielleicht in Java kennengelernt haben, werden durch Prolog normalerweise nicht unterstützt. Sie würden auch nicht gut zu dem grundsätzlich rekursiven Charakter von Logikprogrammen passen.

Die beiden Notationsvarianten unter Verwendung der eckigen Klammern sind nur "syntaktischer Zucker", um einen bequemen Umgang mit Listen zu ermöglichen. Aus diesem Grunde ist für die Einführung der Listen auch keine Erweiterung der Unifikationssemantik erforderlich. Die im folgenden diskutierte Listenunifikation ist demnach nur eine Paraphrasierung der Unifikation rekursiver Terme im Lichte der speziellen Listennotation.

Zugleich ändert sich auch nichts an der Architektur des Logikprogrammiersystems: Terme sind die einzig verfügbare Datenstruktur und die Unifikation von Termen ist die einzige Operation über Termen.

Wegen dieser engen Kopplung an die Unifikation entfällt in der Logikprogrammierung auch eine gefürchtete Fehlerquelle im Umgang mit Pointern. Die für die Listenverarbeitung benötigten Zeigerstrukturen sind für den Programmierer weder sichtbar, noch änderbar. Hierin unterscheidet sich die Logikprogrammierung auch fundamental von den beiden anderen im Zyklus

”Softwareentwicklung” behandelten Programmierparadigmen. So sind in Java Objektreferenzen zwar sichtbar, können aber nicht manipuliert werden. In Scheme hingegen ist ein lesender bzw. schreibender Zugriff auf Pointer unbeschränkt möglich. Ein vollständiger Verzicht würde in einigen Fällen (z.B. Erweiterung einer Liste ”nach rechts”) erheblichen Kopieraufwand mit sich bringen. Wie dieses Problem durch die Unifikation variablenhaltiger Datenstrukturen gelöst werden kann, werden Sie im Abschnitt 10.1 sehen. Damit ist Prolog eine Sprache, die massiv von Pointerstrukturen Gebrauch macht, in der das Pointerkonzept im abstrakten Verarbeitungsmodell aber überhaupt nicht vorkommt, und somit vor dem Programmierer vollständig verborgen ist.

6.2 Listenunifikation

Listenunifikation

- Semantik (iterativ): Zwei Listen unifizieren, wenn ihre Elemente paarweise unifizieren. Die Unifikation schlägt fehl, wenn eine paarweise Zuordnung nicht möglich ist oder wenigstens ein Elementpaar nicht unifiziert.

?- [a,b,c]=[a,b,c].
true.

?- [a,X,c]=[Y,b,c].
X=b, Y=a.

?- [a,X,c]=[X,b,c].
false.

?- [a,b,c]=[a,b].
false.

Listenunifikation

- Semantik (rekursiv): Zwei Listen unifizieren, wenn die Listenköpfe und die Restlisten jeweils miteinander unifizieren. Die Unifikation schlägt fehl, wenn der Unifikationspartner keine Liste ist oder aber der Listenkopf bzw. die Restliste nicht unifizieren.

?- [a,b,c,d]=[X|Y].
X=a, Y=[b,c,d].

?- [a,b,c,d]=[X,Y|_].
X=a, Y=b.

Am Beispiel der Listenunifikation lässt sich sehr schön der vielseitige Charakter der Unifikation als Testoperation, Accessor und Konstruktor studieren, wobei im folgenden nur die beiden letzteren Verwendungen diskutiert werden.

Listenunifikation

- Elementzugriff durch Unifikation (Dekomposition einer Liste)

```
?- [susi,hans,nina,paul]=[X|_].
   X=susi.
```

```
?- [susi,hans,nina,paul]=[_,_ ,X|_].
   X=nina.
```

Listenunifikation

- Konstruktion von Listen durch Unifikation (Komposition)

```
?- L=[susi,hans,nina,paul],EL=[karl|L].
   L = [susi,hans,nina,paul],
   EL = [karl,susi,hans,nina,paul].
```

```
?- [X,Y|R]=[susi,hans,nina,paul],EL=[X,Y,karl|R].
   X = susi,Y=hans,R=[nina,paul],
   EL = [susi,hans,karl,nina,paul].
```

Listenunifikation

- Unifikation rekursiv eingebetteter Listen

```
?- [[a,b]|X]=[X,a,b].
   X=[a,b].
```

```
?- [O,[P,Q],[a,O]]=[b,Q],[b,[c,P]],[a,[b,[R,b]]]
   O=[b,[c,b]],P=b,Q=[c,b],R=c.
```

6.3 Listenverarbeitung

Beispiel 1: Erstes Element einer Liste

- `first(?Element, ?Liste)`
- gewünschtes Verhalten:

1	<code>first(a,[a]).</code>	<code>true.</code>	4	<code>first(a,[]).</code>	<code>false.</code>
2	<code>first(a,[a,b]).</code>	<code>true.</code>	5	<code>first(b,[a,b]).</code>	<code>false.</code>
3	<code>first(a,[a,b,c]).</code>	<code>true.</code>			

- rekursive Definition:

```
% first(?Element,?Liste)
% Element ist ein beliebiger Term und Liste ist eine
% Liste, so dass Element das erste Element der Liste ist
first(E,[E|_]).           % Der Kopf ist das
                           % erste Element einer Liste
```

Beispiel 1: Erstes Element einer Liste

- partiell unterspezifizierte Anfragen

```
?- first(X,[a,b,c]).           % first(-,+)
   X=a.

?- first(a,X).                 % first(+,-)
   X=[a|X1].
```

Beispiel 1: Erstes Element einer Liste

- partiell unterspezifizierte Anfragen

```
?- first(X,X).                 % first(-,-)
   X = [X|_G202].

?- first(X,X), write(X).
   [**|_G202]
   X = [X|_G202].

?- first(X,X), X = [A|_], X = [[B|_|_|],
   X = [[[C|_|_|]|_|_|].
   X = A, A = B, B = C, C = [C|_G202].
```

- vollständig unterspezifizierte Anfrage

```
?- first(X,Y).                                % first(-,-)
   X=X1, Y=[X1|Y1].
```

Das letzte Beispiel unter den partiell spezifizierten Aufrufen

```
?- first(X,X).
```

ist besonders interessant. Hier wird offenbar eine unendlich tief geschachtelte Liste erzeugt. Bei einigen Systemimplementationen führt dies auch zu einer nicht mehr unterbrechbaren Programmschleife. Wenn ich Ihre Neugier geweckt habe, dann versuchen Sie doch einmal nachzuvollziehen, was hier genau passiert. Vergleichen Sie dazu das Beispiel zur Unifikation im Abschnitt 5.1.

Eigentlich müsste ein System zur Logikprogrammierung die hier gegebene Konfiguration erkennen können und angemessen darauf reagieren. Der entsprechende Test (meist *occur check* genannt), ist jedoch nicht ganz billig und daher in den meisten Systemen nicht vorgesehen.

Im Falle des vollständig unterspezifizierten Aufrufs sehen wir wieder ein Beispiel für ein partiell unterspezifiziertes Berechnungsergebnis: “Als Wert von Y kommen alle Listen in Frage, die X als erstes Element enthalten.” Beachten Sie, dass auch diese Ergebnismenge unendlich groß ist.

Im weiteren Verlauf dieses Kapitels betrachten wir eine Vielzahl von Prädikaten zur Listenverarbeitung mit allmählich wachsendem Schwierigkeitsgrad. Dabei sollte das immer wiederkehrende Schema rekursiver Programme deutlich werden, bei dem eine (oder mehrere) Klauseln für den Rekursionsschritt das Berechnungsproblem schrittweise auf den Rekursionsabschluss zurückführen, der ebenfalls durch eine oder mehrere Klauseln abgedeckt wird. Die Listenverarbeitung ist nicht nur wegen ihrer zahlreichen Anwendungsmöglichkeiten von Bedeutung, sondern ist auch derjenige Bereich der Programmierung, in dem sich die Richtungsunabhängigkeit von Logikprogrammen besonders überzeugend zeigt.

Beispiel 2: Element einer Liste

- `in_list(?Element,?Liste)`

- gewünschtes Verhalten:

1	<code>in_list(a,[a]).</code>	<code>true.</code>	5	<code>in_list(a,[]).</code>	<code>false.</code>
2	<code>in_list(a,[a,b]).</code>	<code>true.</code>	6	<code>in_list(d,[a,b,c]).</code>	<code>false.</code>
3	<code>in_list(b,[a,b,c]).</code>	<code>true.</code>			
4	<code>in_list(c,[a,b,c]).</code>	<code>true.</code>			

- rekursive Definition:

```
% in_list(?Element,?Liste)
% Element ist ein beliebiger Term und Liste eine Liste,
% so dass Element ein beliebiges Element der Liste ist
```

```

in_list(E,[E|_]).      % Der Kopf ist Element der Liste
in_list(E,[_|Rest]):- % Ein Element der Restliste
    in_list(E,Rest).   % ist auch Element der Liste

```

Das Prädikat `in_list/2` konsumiert eine Liste auf der zweiten Argumentsposition, bis der Rekursionsabschluss die Gleichheit zwischen der ersten Argumentsposition und dem ersten Listenelement feststellt.

Beispiel 2: Element einer Liste

- partiell unterspezifizierte Anfragen:

```

?- in_list(X,[a,b,c]).      ?- in_list(a,X).
    X=a ;                  X=[a|X1] ;
    X=b ;                  X=[X2,a|X3] ;
    X=c .                  X=[X2,X4,a|X5] .

```

- vollständig unterspezifizierte Anfrage:

```

?- in_list(X,Y).           % in_list(-,-)
    X=X1, Y=[X1|Y1] ;
    X=X1, Y=[X2,X1|Y2] ;
    X=X1, Y=[X2,X3,X1|Y3] .

```

(oftmals) eingebautes Prädikat `member/2`

Beispiel 2: Element einer Liste

- Rekursionsschema: die Variablen werden auf jeder Rekursionsebene durch den Rekursionsabschluss instanziiert

```

il(X,[a,b,c])  il(a,[a,b,c])
      \      X = X1      /
il(X1,[b,c])  il(b,[b,c])
      \      X1 = X2     /
il(X2,[c])   il(c,[c])

                                     il(a,X)           il(a,[a|_])
                                     \      X = [_|X1]      /
                                     il(a,X1)           il(a,[_,a|_])
                                     \      X1 = [_|X2]      /
                                     il(a,X2)           il(a,[_,_,a|_,_])
                                     \      X2 = [_|X3]      /

```

Beispiel 3: Länge einer Liste

- `mylength(+Liste,-Laenge)`
- gewünschtes Verhalten:

1	<code>mylength([],0).</code>	<code>true.</code>	4	<code>mylength([a],0).</code>	<code>false.</code>
2	<code>mylength([a],1).</code>	<code>true.</code>	5	<code>mylength([a,b],1).</code>	<code>false.</code>
3	<code>mylength([a,b],2).</code>	<code>true.</code>			

- rekursive Definition:

```
% mylength(+Liste,-NatZahl)
% Liste ist eine Liste und NatZahl eine natuerliche Zahl
% mit NatZahl gleich der Anzahl der Elemente von Liste

mylength([ ],0).           % die leere Liste hat Laenge 0
mylength([_|Rest],N):-     % die Laenge einer Liste ist
    mylength(Rest,N1),      % die Laenge der Restliste
    N is N1+1.             % erhoeht um 1
```

Das Prädikat `mylength/2` konsumiert auf der ersten Argumentposition eine Liste, bis der Rekursionabschluss dort die leere Liste registriert. Die Ermittlung der Länge erfolgt beim rekursiven Aufstieg durch "Zählen" der Rekursionsschritte ausgehend vom Anfangswert 0 (definiert durch den Rekursionsabschluss).

Versuchen Sie herauszufinden, warum im partiell unterspezifizierten Aufruf

```
?- mylength(X,3).
```

ein Terminierungsproblem entsteht! Lässt sich die Prädikatsdefinition terminierungssicher gestalten? Sehen Sie Parallelen zur Umwandlung von `PEANO-Zahlen`?

Beispiel 3: Länge einer Liste

- partiell unterspezifizierte Anfragen:

```
?- mylength([a,b,c],X).           % mylength(+,-)
    X=3.

?- mylength(X,3).                 % mylength(-,+)
    X=[X1,X2,X3] ;
    . . .
```

- vollständig unterspezifizierte Anfrage:


```
?- mylength(X,Y).                % mylength(-,-)
   Y=[ ], X=0 ;
   Y=[Y1], X=1 ;
   Y=[Y1,Y2], X=2 ;
   Y=[Y1,Y2,Y3], X=3 .
```

Beispiel 3: Länge einer Liste

- Rekursionsschema

```
l([a,b,c],X)      l([a,b,c],3)
  \  X is X1 + 1  /
l([b,c],X1)      l([b,c],2)
  \  X1 is X2 + 1 /
l([c],X2)        l([c],1)
  \  X2 is X3 + 1 /
  l([ ],0)

l(X,3)           l([_,_,_],3)
  \  X = [_|X1], 3 is N1 + 1 /
l(X1,N1)         l([_,_],2)
  \  X1 = [_|X2], N1 is N2 + 1 /
l(X2,N2)         l([_],1)
  \  X2 = [_|X3], N2 is N3 + 1 /
  l([ ],0)
```

Beispiel 4: Verketteten zweier Listen

- `app(?Liste1,?Liste2,?Gesamtliste)`
- gewünschtes Verhalten:

1	<code>app([],[],[]).</code>	yes
2	<code>app([],[a],[a]).</code>	yes
3	<code>app([],[a,b],[a,b]).</code>	yes
4	<code>app([a],[b,c],[a,b,c]).</code>	yes
5	<code>app([a,b],[c,d],[a,b,c,d]).</code>	yes
6	<code>app([a,b,c],[d,e],[a,b,c,d,e]).</code>	yes
7	<code>app([],[a],[]).</code>	no
8	<code>app([a,b],[c,d],[a,d]).</code>	no

4a	<code>app([a []],[b,c],[a [b,c]]).</code>	yes
5a	<code>app([a [b]], [c,d],[a [b,c,d]]).</code>	yes
6a	<code>app([a [b,c]], [d,e],[a [b,c,d,e]]).</code>	yes

Beispiel 4: Verketteten zweier Listen

- rekursive Definition:

```
% app(?Liste1,?Liste2,?Resultat)
% Liste1, Liste2 und Resultat sind Listen,
% so dass Resultat die Verkettung von Liste1
% und Liste2 ist

app([ ],L,L).           % [ ] ist neutrales Element
app([K|R1],L2,[K|VL]):- % Verk. von L2 mit der Restliste
    app(R1,L2,VL).      % ergibt die verkettete Restliste
```

Ein strukturell sehr ähnliches Prädikat ist Ihnen in diesem Kapitel bereits begegnet. Welches ist es?

`app/3` konsumiert auf der ersten Argumentposition eine Liste, wobei das jeweils abgetrennte erste Listenelement beim rekursiven Aufstieg auf dem dritten Argument an den Anfang der Ergebnisliste gehängt wird. Der rekursive Abstieg ist beendet, wenn der Rekursionsabschluss auf der ersten Argumentposition eine leere Liste vorfindet. Die Konstruktion der verknüpften Ergebnisliste erfolgt auf der dritten Argumentposition. Diese wird am Rekursionsabschluss initialisiert, indem die Liste von der zweiten Argumentposition (der "hintere" Teil der Liste) unverändert übernommen wird. Beim Rekursionsschritt braucht diese daher auch nur unverändert "durchgereicht" zu werden.

Das Ergebnis der Verkettung ist beim rekursiven Abstieg bereits "bekannt", aber noch (partiell) unterspezifiziert: es enthält am Listeneende noch ein uninstantiierte Variable. Diese Listenstruktur wird nun schrittweise mit zusätzlicher Information angereichert, bis sie dann am Rekursionsabschluss in vollständig instantzierter Form vorliegt.

Ich empfehle Ihnen dringend, diesen Prozess der Listenmanipulation für verschiedene Instantzierungsvarianten im Detail nachzuvollziehen. Das `append/3`-Prädikat gilt gewöhnlich als Prüfstein für das Verständnis der komplexen Abläufe bei der Abarbeitung nichttrivialer Anfragen. Bei der Analyse des dabei hergestellten Netzes aus Variablenbindungen erhält man gleichzeitig einen guten Eindruck von der Mächtigkeit und Schönheit der (im strengen Sinne) rein deklarativen Unifikationsoperation.

Beispiel 4: Verketteten zweier Listen

- ein unterspezifiziertes Argument

```
?- app([a,b],[c,d,e],X).           % app(+,+, -)
    X=[a,b,c,d,e].

?- app(X,[c,d,e],[a,b,c,d,e]).      % app(-,+, +)
    X=[a,b].

?- app([a,b],X,[a,b,c,d,e]).        % app(+,-, +)
    X=[c,d,e].
```

Beispiel 4: Verketteten zweier Listen

- Rekursionsschema

$$\begin{array}{c}
 \text{app}([a,b,c],[d,e],X) \quad \text{app}([a,b,c],[d,e],[a,b,c,d,e]) \\
 \quad \backslash \quad \quad \quad X = [a|X1] \quad \quad \quad / \\
 \text{app}([b,c],[d,e],X1) \quad \text{app}([b,c],[d,e],[b,c,d,e]) \\
 \quad \backslash \quad \quad \quad X1 = [b|X2] \quad \quad \quad / \\
 \text{app}([c],[d,e],X2) \quad \text{app}([c],[d,e],[c,d,e]) \\
 \quad \quad \quad \backslash \quad \quad \quad X2 = [c|X3] \quad \quad \quad / \\
 \quad \quad \quad \text{app}([], [d,e], [d,e])
 \end{array}$$

$$\begin{array}{c}
 \text{app}(X,[d,e],[a,b,c,d,e]) \quad \text{app}([a,b,c],[d,e],[a,b,c,d,e]) \\
 \quad \backslash \quad \quad \quad X = [a|X1] \quad \quad \quad / \\
 \text{app}(X1,[d,e],[b,c,d,e]) \quad \text{app}([b,c],[d,e],[b,c,d,e]) \\
 \quad \backslash \quad \quad \quad X1 = [b|X2] \quad \quad \quad / \\
 \text{app}(X2,[d,e],[c,d,e]) \quad \text{app}([c],[d,e],[c,d,e]) \\
 \quad \quad \quad \backslash \quad \quad \quad X2 = [c|X3] \quad \quad \quad / \\
 \quad \quad \quad \text{app}([], [d,e], [d,e])
 \end{array}$$

Beispiel 4: Verketteten zweier Listen

- Rekursionsschema

$$\begin{array}{c}
 \text{app}([a,b,c],X,[a,b,c,d,e]) \quad \text{app}([a,b,c],[d,e],[a,b,c,d,e]) \\
 \quad \backslash \quad \quad \quad X = X1 \quad \quad \quad / \\
 \text{app}([b,c],X1,[b,c,d,e]) \quad \text{app}([b,c],[d,e],[b,c,d,e]) \\
 \quad \backslash \quad \quad \quad X1 = X2 \quad \quad \quad / \\
 \text{app}([c],X2,[c,d,e]) \quad \text{app}([c],[d,e],[c,d,e]) \\
 \quad \quad \quad \backslash \quad \quad \quad X2 = X3 \quad \quad \quad / \\
 \quad \quad \quad \text{app}([], [d,e], [d,e])
 \end{array}$$

Beispiel 4: Verketteten zweier Listen

- zwei un spezifizierte Argumente (1):

```
?- app(X,Y,[a,b,c,d,e]).                % app(-,-,+)
   X=[ ],Y=[a,b,c,d,e] ;
   X=[a],Y=[b,c,d,e] ;
   X=[a,b],Y=[c,d,e] ;
   X=[a,b,c],Y=[d,e] ;
   X=[a,b,c,d],Y=[e] ;
   X=[a,b,c,d,e],Y=[ ] ;
no

?- app(X,[c,d,e],Y).                    % app(-,+, -)
   X=[ ], Y=[c,d,e] ;
   X=[X1], Y=[X1,c,d,e] ;
   X=[X1,X2], Y=[X1,X2,c,d,e] ;
   X=[X1,X2,X3], Y=[X1,X2,X3,c,d,e]
yes
```

Beispiel 4: Verketteten zweier Listen

- Rekursionsschema

```
app(X,Y,[a,b,c,d,e])  app([a,b,c],[d,e],[a,b,c,d,e])
      \      X = [a|X1], Y = Y1      /
app(X1,Y1,[b,c,d,e])  app([b,c],[d,e],[b,c,d,e])
      \      X1 = [b|X2], Y1 = Y2      /
app(X2,Y2,[c,d,e])    app([c],[d,e],[c,d,e])
      \      X2 = [c|X3], Y2 = Y3      /
      \app([ ],[d,e],[d,e])      /

app(X,[d,e],Y)        app([_,_,_],[d,e],[_,_,_,d,e])
      \      X = [_|X1], Y = [_|Y1]      /
app(X1,[d,e],Y1)      app([_,_],[d,e],[_,_,d,e])
      \      X1 = [_|X2], Y1 = [_|Y2]      /
app(X2,[d,e],Y2)      app([_],[d,e],[_,d,e])
      \      X2 = [_|X3], Y2 = [_|Y3]      /
      \app([ ],[d,e],[d,e])      /
```

Beispiel 4: Verketteten zweier Listen

- vollständig unterspezifizierter Aufruf

```
?- app(X,Y,Z).                % app(-,-,-)
   X=[ ], Y=Y1, Z=Y1 ;
   X=[X1], Y=Y1, Z=[X1|Y1] ;
   X=[X1,X2], Y=Y1, Z=[X1,X2|Y1] ;
   X=[X1,X2,X3], Y=Y1, Z=[X1,X2,X3|Y1]
yes
```

(oftmals) eingebautes Prädikat `append(?Liste1,?Liste2,?Resultat)`

Beispiel 5a: Suffix einer Liste

- rekursive Definition

```
% suffix(?Suffix,?Liste)
suffix(L,L).
suffix(S,[_|R]) :- suffix(S,R).
```

- partiell unterspezifizierte Anfragen

```
?- suffix(S,[a,b,c]).        % suffix(-,+)
   S = [a,b,c] ;
   S = [b,c] ;
   S = [c] ;
   S = [].

?- suffix([a,b],L).          % suffix(+,-)
   L = [a,b] ;
   L = [_G298,a,b] ;
   L = [_G298,_G301,a, b] .
```

Beispiel 5b: Präfix einer Liste

- rekursive Definition

```
% prefix(?Prefix,?Liste)
prefix([],_).
prefix([E|P],[E|R]) :- prefix(P,R).
```

- unterspezifizierte Anfragen

```
?- prefix(P,[a,b,c]).           % prefix(-,+)
P = [] ;
P = [a] ;
P = [a,b] ;
P = [a,b,c].

?- prefix([a,b],L).             % prefix(+,-)
L = [a,b|_G302].
```

Beispiel 5c: Teilliste einer Liste

```
% sublist(?Subliste,?Liste)
sublist(S,L) :- prefix(S,L).
sublist(S,[_|R]) :- sublist(S,R).
```

```
?- sublist(S,[a,b,c]).          % sublist(-,+)
S = [] ;
S = [a] ;
S = [a, b] ;
S = [a, b, c] ;
S = [] ;
S = [b] ;
S = [b, c] ;
S = [] ;
S = [c] ;
S = [].
```

Querbeziehungen in der Listenverarbeitung

- Basisprädikate zur Listenverarbeitung können wechselseitig auseinander definiert werden

```

member(E,L) :- sublist([E],L).
sublist(Sub,L) :- prefix(Pre,L), suffix(Sub,Pre).
sublist(Sub,L) :- suffix(Suf,L), prefix(Sub,Suf).
prefix(P,L) :- append(P,_,L).
suffix(S,L) :- append(_,S,L).
sublist(Sub,L) :- append(_,Suf,L), append(Sub,_,Suf).
sublist(Sub,L) :- append(Pre,_,L), append(_,Sub,Pre).
sublist(Sub,L) :- prefix(Pre,L), suffix(Suf,L),
    append(Pre,Sub,L1), append(L1,Suf,L).
...

```

Beispiel 6: Umdrehen einer Liste

- `reverse(?Liste1,?Liste2)`
- gewünschtes Verhalten:

1	<code>reverse([],[])</code>	yes
2	<code>reverse([a],[a])</code>	yes
3	<code>reverse([a,b],[b,a])</code>	yes
4	<code>reverse([a,b,c],[c,b,a])</code>	yes

- Direkte, längenunabhängige Implementierung?

Beispiel 6: Umdrehen einer Liste

- Repräsentation von Teilergebnissen:

1	<code>rev1([a,b,c],[])</code>	yes
2	<code>rev1([b,c],[a])</code>	yes
3	<code>rev1([c],[b,a])</code>	yes
4	<code>rev1([],???)</code>	yes

- Augmentation: zusätzliches Argument für Endergebnis:

1	<code>rev2([a,b,c],[],_)</code>	yes
2	<code>rev2([b,c],[a],_)</code>	yes
3	<code>rev2([c],[b,a],_)</code>	yes
4	<code>rev2([],[c,b,a],[c,b,a])</code>	yes

Beispiel 6: Umdrehen einer Liste

- Rekursive Prädikatsdefinition:

```
rev2([ ],L,L).
rev2([H|T],A,R):-rev2(T,[H|A],R).
```

- Einkleiden zur Unterdrückung des Hilfsarguments

```
% reverse(?Liste1,?Liste2)
% Liste1 und Liste2 sind Listen, so dass sie jeweils
% die Elemente der anderen Liste in umgekehrter
% Reihenfolge enthalten

reverse(L,R):-
    rev2(L,[ ],R).
```

Beispiel 6: Umdrehen einer Liste

- Rekursionsschema

```
rev2([a,b,c],[ ],X)      rev2([a,b,c],[ ],[c,b,a])
      \                X = X1                /
rev2([b,c],[a],X1)      rev2([b,c],[a],[c,b,a])
      \                X1 = X2                /
rev2([c],[b,a],X2)      rev2([c],[b,a],[c,b,a])
      \                X2 = X3                /
      ( rev2([ ],[c,b,a],[c,b,a]) )
```

Die hier angegebene Implementation des Prädikats `reverse/2` ist bereits im Hinblick auf effiziente Abarbeitung optimiert. Sie verwendet ein Hilfprädikat `rev/3`, das ein zusätzliches Argument (initialisiert mit der leeren Liste) einführt, auf dem die Listenelemente in umgekehrter Reihenfolge “eingesammelt” werden. Überlegen Sie sich eine Implementation des `reverse/2` Prädikats, die ohne die Verwendung eines Hilfsprädikats auskommt. Hinweis: Verwenden Sie das bereits definierte `append/3` Prädikat.

Beispiele 7a: Löschen von Elementen

- alle Vorkommen des Elements

```
% delete_element(?Element,?Liste,?RedListe)
delete_element(_,[_],[_]).
delete_element(E,[E|R],RL) :-
    delete_element(E,R,RL).
delete_element(E,[X|R],[X|RL]) :-
    X\=E, delete_element(E,R,RL).

% sinnvolle Verwendung:
?- delete_element(a,[a,f,d,w,a,g,t,s,a],L).
   L = [f, d, w, g, t, s].
```

→ delete/3 (deprecated)

Beispiel 7b: Löschen eines Elements

- Entfernen nur eines Vorkommens des gegebenen Elementes

```
% select_element(?Element,?Liste,?RedListe)
select_element(E,[E|R],R).
select_element(E,[X|R],[X|RL]) :-
    select_element(E,R,RL).

% sinnvolle Verwendungen
?- select_element(a,[a,f,d,w,a,g,t,s,a],L).
   L = [f, d, w, a, g, t, s, a] ;
   L = [a, f, d, w, g, t, s, a] ;
   L = [a, f, d, w, a, g, t, s].

?- select_element(E,[a,b,c],L).
   E = a, L = [b, c] ;
   E = b, L = [a, c] ;
   E = c, L = [a, b].
```

→ select/3

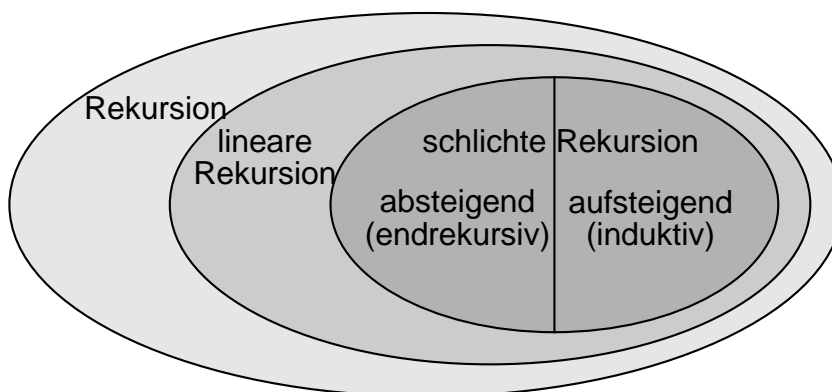
Beispiel 8: Einebnen einer Liste

- Erzeugen einer linearen Liste aus einer rekursiv geschachtelten Liste

```
% flatten(+geschachtelte Liste,?lineare_Liste)
flatten([], []).
flatten([K|R], [K|RF]) :-
    atomic(K),
    flatten(R, RF).
flatten([K|R], F) :-
    is_list(K),
    flatten(K, KF),
    flatten(R, RF),
    append(KF, RF, F).
```

- verzweigende Rekursion: mehrere rekursive Aufrufe in einer Klausel

Überblick Rekursion



Programme mit verzweigender Rekursion haben im allgemeinen Fall einen hohen Rechenzeitbedarf, der z.B. bei unbeschränkter Traversierung eines Baumes mit konstantem Verzweigungsfaktor exponentiell mit der Tiefe des Baumes wächst. Leider besteht bei verzweigender Rekursion auch die Möglichkeit zur Umwandlung in ein endrekursives Programm nur noch in wenigen Spezialfällen, in denen eine Beschreibung des Problemzustandes durch endlich viele Parameter möglich ist. Dies ist z.B. bei der Berechnung der FIBONACCI-Zahlen der Fall, nicht jedoch bei der Verarbeitung von verzweigend rekursiven Datenstrukturen.

6.4 Suchen und Sortieren

Sortierverfahren haben Sie schon im ersten Modul zur Softwareentwicklung kennengelernt. Wir greifen hier einige davon wieder auf, nicht zuletzt, weil sie in vielen Fällen auf eine sehr elegante rekursive Programmlösung führen. Nicht eingeschlossen sind natürlich Sortierverfahren, die ausschließlich auf Arrays arbeiten. Vergleichen Sie die angegebenen Verfahren im Hinblick, auf die aus SE I bekannten Kriterien: Komplexität, Speicherbedarf und Stabilität.

Anwendungsbereich 1: Suchen und Sortieren

- Suchen in einer sortierten Liste
- Sortieren durch Einfügen in eine sortierte Liste
- Sortieren durch Auswahl des minimalen Elements
- Sortieren durch rekursives Zerlegen einer Liste (Quicksort)
- Sortieren durch Aufbau eines (sortierten) Baumes

Suche in einer sortierten Liste

```
% member_sort(+Element,+SortierteListe)
member_sort(E,[E|_]).
member_sort(E,[X|R]) :-
    E @> X,
    member_sort(E,R).
```

Sortieren durch Permutation

```
% permutation(+Liste,?Permutierte Liste)
permutation([],[]).
permutation(L,[E|R]) :- select_element(E,L,L1),
    permutation(L1,R).
```

```
?- permutation([a,b,c],L).
    L = [a, b, c] ;
    L = [a, c, b] ;
    L = [b, a, c] ;
    L = [b, c, a] ;
    L = [c, a, b] ;
    L = [c, b, a].
```

Sortieren durch Einfügen

- Einfügen in eine sortierte Liste

```
% sort_e(+Liste,?SortierteListe)
sort_e([ ],[ ]).
sort_e([E|L],SL) :-
    sort_e(L,SL1),
    insert_l(E,SL1,SL).
```

Sortieren durch Einfügen

- Einfügen eines Elements

```
% insert_l(+Element,+SortierteListe,?ErwListe)
insert_l(E,[ ],[E]).
insert_l(E,[X|R],[X|RL]) :-
    E@>X,
    insert_l(E,R,RL).
insert_l(E,[X|R],[E,X|R]) :-
    E@=<X.
```

Sortieren durch Auswahl des minimalen Elements

- Entferne das kleinste Element und setze es an den Listenanfang

```
% sort_a(+Liste,?SortierteListe)
sort_a([E],[E]).
sort_a(L,[M|R]):-
    select_minimum(M,L,Rest),
    sort_a(Rest,R).

% select_minimum(?Minimum,+Liste,?Restliste)
select_minimum(E,L,R) :-
    minimum(E,L),
    select_element(E,L,R).
```

Sortieren durch Auswahl des minimalen Elements

- Minimales Element einer Liste

```
% minimum(?MinimalesElement,+Liste)
minimum(M,[M]).
minimum(M,[X|R]) :-
    minimum(M,R),
    M@<X.
minimum(X,[X|R]) :-
    minimum(M,R),
    M@>=X.
```

Quicksort

- Sortieren durch rekursive Zerlegung in Teilprobleme

```
% sort_q(+Liste,?SortierteListe)
sort_q([],[]).
sort_q([E|R],SL) :-
    split(R,E,Vorn,Hinten),
    sort_q(Vorn,VS),
    sort_q(Hinten,HS),
    append(VS,[E|HS],SL).
```

Quicksort

- Zerlegen einer Liste

```
%split(+Liste,+MittleresElement,
%      ?VordereElemente,?HintereElemente)
split([ ],_,[ ],[ ]).
split([E|R],M,[E|VL],HL) :-
    E@=<M, split(R,M,VL,HL).
split([E|R],M,VL,[E|HL]) :-
    E@>M, split(R,M,VL,HL).

?- split([c,a,e,b],d,V,H).
   V = [c, a, b], H = [e].
```

Sortieren durch Konstruktion eines Baumes

- Umwandeln einer Liste in einen sortierten Baum und Rückumwandeln in eine (sortierte) Liste

```
% sort_t(+Liste,?SortierteListe)
sort_t(L,SL) :-
    list2tree(L,B),
    tree2list(B,SL).
```

Sortieren durch Konstruktion eines Baumes

- Umwandeln der Liste in einen Baum

```
% list2tree(+Liste,?Baum)
list2tree([ ],end).
list2tree([E|R],t(E,VB,HB)) :-
    split(R,E,VL,HL),
    list2tree(VL,VB),
    list2tree(HL,HB).

?- list2tree([d,c,a,e,b],B).
   B = t(d, t(c, t(a, end, t(b, end, end)),
          end), t(e, end, end)).
```

Sortieren durch Konstruktion eines Baumes

- Umwandeln der Liste beim rekursiven Aufstieg

```
% list2tree(+Liste,?Baum)
list2tree([E],t(E,end,end)).
list2tree([E|R],t(S,VB,HB)) :-
    list2tree(R,t(S1,VB1,HB1)),
    intree(E,t(S1,VB1,HB1),t(S,VB,HB)).

?- list2tree([d,c,a,e,b],B).
   B = t(b, t(a, end, end), t(e, t(c, end,
        t(d, end, end)), end)).
```

Sortieren durch Konstruktion eines Baumes

- Einfügen eines Elementes in einen (sortierten) Baum

```
% intree(+Element,+BaumAlt,?BaumNeu)
intree(E,end,t(E,end,end)).
intree(E,t(S,VB,HB),t(S,VBN,HB)) :-
    E@=<S, intree(E,VB,VBN).
intree(E,t(S,VB,HB),t(S,VB,HBN)) :-
    E@>S, intree(E,HB,HBN).
```

Sortieren durch Konstruktion eines Baumes

- Beispielaufruf

```
?- intree(f,t(d, t(c, t(a, end, t(b, end, end)),
    end), t(e, end, end)),T).
   T = t(d, t(c, t(a, end, t(b, end, end)),
    end), t(e, end, t(f, end, end))).
```

Sortieren durch Konstruktion eines Baumes

- Umwandeln des Baumes in eine Liste

```
% tree2list(+Baum,?Liste)
tree2list(end,[ ]).
tree2list(t(E,VB,HB),L) :-
    tree2list(VB,VL),
    tree2list(HB,HL),
    append(VL,[E|HL],L).
```

Sortieren durch Konstruktion eines Baumes

```
?- tree2list(t(d, t(c, t(a, end, t(b, end, end)),
    end), t(e, end, t(f, end, end)),L).
   L = [a, b, c, d, e, f].
```

6.5 Memoization

Anwendungsbereich 2: Memoization

- Memoization: Buchführung über die bisher besuchten Suchzustände
 - Ausgabe der Zustandsfolge
 - * z.B. Endlicher Automat
 - Überwachung von Zyklen
 - * z.B. Suche in einem Labyrinth

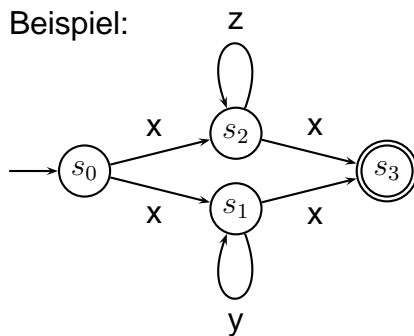
Anwendungsbeispiel: Endlicher Automat

- Repräsentation eines Endlichen Automaten:

- Startzustand: $a(\text{zustand})$
- Zielzustand: $e(\text{zustand})$
- Zustandsübergang: $t(\text{zustand-alt}, \text{symbol}, \text{zustand-neu})$

- Beispiel:

Regulärer Ausdruck: $x(y^*|z^*)x$



Anwendungsbeispiel: Endlicher Automat

- Repräsentation des Beispielautomaten:

$a(s_0).$	$t(s_0, x, s_1).$
	$t(s_0, x, s_2).$
$e(s_3).$	$t(s_1, x, s_3).$
	$t(s_2, x, s_3).$
	$t(s_1, y, s_1).$
	$t(s_2, z, s_2).$

- Repräsentation der Zeichenfolge als Liste: $[x, z, z, z, x]$

Anwendungsbeispiel: Endlicher Automat

```
% generate(?Wort)
% Wort ist eine Liste von Symbolen, so dass
% Wort Element der durch den endlichen
% Automaten definierten Sprache ist

generate(Wort):-
    a(Start),
    gen(Wort,Start,Ziel),
    e(Ziel).
gen([ ],Start,Start).
gen([Kopf|Rest],Start,Ziel):-
    t(Start,Kopf,Zwischenzustand),
    gen(Rest,Zwischenzustand,Ziel).
```

Anwendungsbeispiel: Endlicher Automat

```
?- generate([x,y,x]).
true.
?- generate([x,y]).
false.
?- generate([y,z]).
false.
?- generate(X).
X=[x,x] ;
X=[x,y,x] ;
X=[x,y,y,x] ;
X=[x,y,y,y,x] .
```

Anwendungsbeispiel: Zyklenvermeidung im Labyrinth

- Repräsentation eines Labyrinths

```
% weg(Ort_a, Ort_b)
c(a,b).
c(b,c).
c(a,d).
c(b,d).
c(c,d).
weg(A,B) :- c(A,B).
weg(A,B) :- c(B,A).
weg(A,C) :- c(A,B), weg(B,C).
weg(A,C) :- c(B,A), weg(B,C).
```

Anwendungsbeispiel: Zyklenvermeidung im Labyrinth

```
?- weg(d,X).
   X = a ;
   X = b ;
   X = c ;
   X = b ;
   X = d ;
   ...
   X = a ;
   X = b ;
   X = c ;
   X = b ;
   X = d ;
   ...
```

Anwendungsbeispiel: Zyklenvermeidung im Labyrinth

- Protokollierung der (virtuell) zurückgelegten Wegstrecke in einem Akkumulator (3. Argument)
- Initialzustand ist der Ausgangsort
- Abbruch bei erneutem Auftreten eines bereits besuchten Ortes

Anwendungsbeispiel: Zyklenvermeidung im Labyrinth

```
% weg_oz(?Start,?Ziel)
% zyklensfreie Verbindung
weg_oz(A,B) :- w_oz(A,B,[A]).
w_oz(A,B,W) :-
    c(A,B), \+ member(B,W).
w_oz(A,B,W) :-
    c(B,A), \+ member(B,W).
w_oz(A,C,W) :-
    c(A,B), \+ member(B,W), w_oz(B,C,[B|W]).
w_oz(A,C,W) :-
    c(B,A), \+ member(B,W), w_oz(B,C,[B|W]).
```

Anwendungsbeispiel: Zyklenvermeidung im Labyrinth

```
?- weg_oz(d,X).
   X = a ;
   X = b ;
   X = c ;
   X = b ;
   X = c ;
```

```

X = c ;
X = a ;
X = b ;
X = a.

```

Anwendungsbeispiel: Zyklenvermeidung im Labyrinth

- Ausgabe der zurückgelegten Wegstrecke

```

% weg_oz(?Start,?Ziel,?BishWeg,?GesWeg)
% zyklenfreie Verbindung
weg_oz(A,B,G) :- w_oz(A,B,[A],R), reverse(R,G).
w_oz(A,B,W,[B|W]) :-
    c(A,B), \+ member(B,W).
w_oz(A,B,W,[B|W]) :-
    c(B,A), \+ member(B,W).
w_oz(A,C,W,GW) :-
    c(A,B), \+ member(B,W), w_oz(B,C,[B|W],GW).
w_oz(A,C,W,GW) :-
    c(B,A), \+ member(B,W), w_oz(B,C,[B|W],GW).

```

Anwendungsbeispiel: Zyklenvermeidung im Labyrinth

```

?- weg_oz(d,X,G).
X = a, G = [d, a] ;
X = b, G = [d, b] ;
X = c, G = [d, c] ;
X = b, G = [d, a, b] ;
X = c, G = [d, a, b, c] ;
X = c, G = [d, b, c] ;
X = a, G = [d, b, a] ;
X = b, G = [d, c, b] ;
X = a, G = [d, c, b, a].

```

6.6 Suchraumverwaltung

Anwendungsbereich 3: Suchraumverwaltung

- Prolog realisiert standardmäßig eine Tiefe-zuerst-Suche
- übernimmt das Programm selbst die Verwaltung der Suchraumzustände, können auch alternative Suchstrategien realisiert werden

Anwendungsbereich 3: Suchraumverwaltung

- Beispiel: endlicher Automat
- Verwalten einer Agenda auf der ersten Argumentstelle
 - Zustand des Automaten
 - Bisher erzeugtes Wort (rückwärts)

z.B. [s1, [y, y, x]]

- Abarbeiten der Agenda vom Listenanfang
- Hinzufügen neuer Agendaelemente
 - am Listenanfang → Kellerspeicher (Stack)
 - am Listenende → Warteschlange (Queue)

Anwendungsbereich 3: Suchraumverwaltung

- Erzeugen eines Wortes

```
% generate(?Wort)
generate(Word) :-
    findall([A, [ ]], a(A), Agenda),
    gen(Agenda, WordR),
    reverse(WordR, Word).
```

Anwendungsbereich 3: Suchraumverwaltung

- Abarbeiten der Agenda

```
% gen(+Agenda, ?BisherigesWort)
gen([[End, PWord] | _], PWord) :- e(End).
gen([[Z, PWord] | AgendaR], Word) :-
    findall([ZNext, [E | PWord]],
            t(Z, E, ZNext),
            NewItems),
    % append(NewItems, AgendaR, AgendaNew), % stack
    append(AgendaR, NewItems, AgendaNew), % queue
    gen(AgendaNew, Word).
```

Bäume

- Test: Ist Wort (gegeben als Liste) im Trie enthalten?

```
% word(?Word,?Trie).
word([],[_|_]).
word([C|RW],[[C|RT]|_]) :- word(RW,RT).
word(W,[_|Alt]) :- word(W,Alt).

?- trie(T),word([a,l,s,o],T).      % word(+,+)
   T = [...].

?- trie(T),word([a,b,e,r],T).      % word(+,+)
   false.
```

Bäume

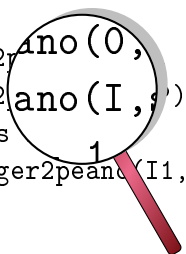
```
?- trie(T),word(W,T).              % word(-,+)
   T = [...]
   W = [a, b, e, n, d] ;
   T = [...]
   W = [a, f, f, e] ;
   T = [...]
   W = [a, l, l, e] ;
   ...
   T = [...]
   W = [b, a, u, e, r].

?- word([a,l,s,o],T).               % word(+,-)
   T = [[a, [l, [s, [o, []|...]|_G259]|_G253]|_G247]|_G241] ;
   T = [[a, [l, [s, [o, _G264|...]|_G259]|_G253]|_G247]|_G241] ;
   true.
```

7 Extra- und Metalogische Prädikate

In diesem Kapitel behandeln wir eine Reihe von Prädikaten, die dazu dienen, den Prozess der Programmabarbeitung im logikbasierten Paradigma zu beobachten oder zu steuern (metalogische Prädikate) oder aber die keinerlei Bezug zur logischen Ableitung der Berechnungsergebnisse haben. Zu letzteren gehören etwa der Bereich der Ein-/Ausgabe (ein reiner Nebeneffekt der Programmabarbeitung!), Veränderungen an der Axiomenmenge (Einfügen oder Löschen von Klauseln), die Arbeit mit globalen Variablen (ja, selbst so etwas gibt es in Prolog!) und das Interface zum Betriebssystem, von denen wir hier aber nur einen Überblick über die Möglichkeiten zur Datenbankmanipulation und zur Verwendung globaler Variablen geben werden.

Extra- und Metalogische Prädikate



```
integer2peano(0,
integer2peano(I, S)) :-
    I1 is I-1,
    integer2peano(I1, P).
```

Extra- und Metalogische Prädikate

Metalogische Prädikate

- Typinspektion
- Typkonversion
- Typkonstruktion
- Kontrollierte Instantiierung
- Suchraummanipulation
- (Trace und Debug)

Extralogische Prädikate

- (Input/Output)
- Datenbankmanipulation
- globale Variable
- (Interface zum Betriebssystem)

7.1 Typen und Typkonversion

Atomare Basistypen

integer/1	Ganze Zahlen
float/1	Gleitkommazahlen
atom/1	Atome (Namen)
string/1	SWI-String

Abgeleitete atomare Typen

```
number(X) :- integer(X).
number(X) :- float(X).
atomic(X) :- string(X).
atomic(X) :- atom(X).
atomic(X) :- number(X).
```

Typüberprüfung

- Beispiel: Terminierungssicheres ggt/3

```
% ggt(+NatZahl1,+NatZahl2,?GGT)
ggt(X,Y,Z) :- integer(X), integer(Y),
    X>0, Y>0, ggt1(X,Y,Z).
ggt1(X,X,X).
ggt1(X,Y,Z) :- X<Y, ggt1(Y,X,Z).
ggt1(X,Y,Z) :-
    X>Y, X1 is X-Y, ggt1(X1,Y,Z).

?- ggt(1.5,2.1,X).
false.
```

Zeichenketten

- Datentyp string
 - im originalen Edinburgh-Prolog als Liste von ASCII-Werten
 - Konvertierung direkt beim Einlesen read/1
 - logisch "sauber"
 - aber ineffizient [5mm]
- daher in ISO-Norm: separater Datentyp String

```
?- X="string".
X = "string".

?- X="string", string(X).
X = "string".
```

Zeichenketten

- ab SWI-Prolog 7.0: Strings als Listen von ASCII-Codes mit Backquotes

```
?- X = "string", string(X).
X = "string".
```

```
?- X = 'string', string(X).
false.
```

```
?- X = 'string', is_list(X).
X = [115, 116, 114, 105, 110, 103].
```

Zeichenketten

- drei Möglichkeiten zur Repräsentation von Text-Daten [3mm]
 - als Atom: 'Atom' wenn als Identifier verwendet [3mm]
 - als Listen von ASCII-codes: 'a+b' wenn der Text strukturell analysiert werden soll, z.B. Programmiersprachenausdrücke [3mm]
 - als Zeichenketten (String): "String" wenn weder Identifier noch strukturelle Analyse notwendig

Strukturen

- Strukturtypen

compound(+Term)	compound(X) :- \+ atomic(X)
is_list(+List)	oberste Ebene ist gepunktetes Paar
proper_list(+List)	rekursiver Nachweis
is_set(+List)	wie proper_list, aber keine Duplikate

Typkonversion

- arithmetische Funktionen

```
float
integer
```

Typkonversion

- Atom - Zeichenkette - Struktur

```
atom_codes(?Atom,?List_of_char_codes)
name(?AtomOrInteger,?List_of_char_codes)
atom_char(?Atom,?ASCII_Value)
int_to_atom(+Integer,+Base,-Atom)
term_to_atom(?Term,?Atom)
string_to_atom(?String,?Atom)
string_codes(?String,?List_of_char_codes)
    äquivalent zu: string_to_list/2
...
```

Typkonversion

- Strings

```
?- string_to_list("ABC",L), atom_codes(A,L).
L = [65, 66, 67],
A = 'ABC'.

?- term_to_atom(p(1,2),A), atom_codes(A,L),
   string_to_atom(S,A).
A = 'p(1,2)',
L = [112, 40, 49, 44, 50, 41],
S = "p(1,2)".
```

Typspezifische Accessoren

- Atome

```
atom_length(+Atom,-Length)
string_length(+String,-Length)
substring(+String,+Start,+Len,-Substr)
```

- Listen

```
last/2
member/2
nth0/3, nth1/3
length/2
...
```

Typkonstruktion

- Atome

```
atom_chars(?Atom,?ListOfChars)
number_chars(?Number,?ListOfDigits)
concat(?Atom1,?Atom2,?Atom3)
concat(+ListOfAtoms,-Atom)
```

Typkonstruktion

- Listen

```
% ?Term =.. ?List

?- a(b,c) =.. X.
   X = [a,b,c].

?- X =.. [a,b,c].
   X = a(b,c).
```

7.2 Kontrollierte Instanziierung

Variableninspektion

```
% var(+Term)
?- var(X).
   X = _G123.

?- var(a).
   false.
```

Variableninspektion

```
% nonvar(+Term)
?- nonvar(X).
   false.

?- nonvar(a).
   true.

?- nonvar(a(X)).
   X = _G215.
```

Variableninspektion

```
% ground(+Term)
?- ground(X).
   false.

?- ground(a(X)).
   false.

?- ground(a(b)).
   true.
```

Variableninspektion

- Beispiel: (partiell) relationale Addition
 - wenigstens zwei Argumente müssen instanziiert sein

```
% add(?Zahl1, ?Zahl2, ?Sum)

add(X,Y,Z) :-
    nonvar(X), nonvar(Y), Z is X + Y.
add(X,Y,Z) :-
    nonvar(X), nonvar(Z), Y is Z - X.
add(X,Y,Z) :-
    nonvar(Y), nonvar(Z), X is Z - Y.
```

Variableninspektion

- Beispiel: (partiell) relationale Addition

```
?- add(3,4,X).
   X=7.

?- add(1,X,4).
   X=3.

?- add(X,Y,7).
   false.
```

Instanziierungsfreier Vergleich

- Vergleich variablenhaltiger Terme ohne Instanziierung der freien Variablen
 - strukturelle Identität
 - strukturelle Gleichheit

strukturelle Identität

- gleicher Typ, gleicher Funktor, gleiche Stelligkeit, gleiche Argumente
- Identität von Variablen: gleicher Name oder vorherige Unifikation

a == a	true	a(X) == a(X)	true
a == A	false	a(X) == a(Y)	false
a(b) == a(b)	true	X = Y, a(X) == a(Y)	true

Instanziierungsfreier Vergleich

strukturelle Gleichheit

- Identitätstest, aber Variable sind auch dann gleich, wenn sich bei konsistenter Umbenennung gleiche Koreferenzbeziehungen ergeben

a =@= A	false	x(A,A) =@= x(B,C)	false
A =@= B	true	x(A,A) =@= x(B,B)	true
		x(A,B) =@= x(C,D)	true

- Strukturelle Identität
 - ⊂ Strukturelle Gleichheit
 - ⊂ Unifizierbarkeit

Unifikationsfreie Strukturerzeugung

- `free_variables(+Term,-ListOfFreeVariables)`: Erzeugt eine Liste aller freien Variablen in Term
- `copy_term(+In,-Out)`: Erzeugt einen strukturgleichen Term

Erzeugen uninstanzierter Terme

```
% functor(?Term,?Functor,?Arity)
?- functor(a(b),X,Y).
   X = a, Y = 1.

?- functor(X,a,2).
   X = a(_G123, _G125).
```

Instantiieren von einzelnen Argumenten

```
% arg(?ArgNo,?Term,?Value)
?- arg(1,a(b,c),X).
   X = b.

?- functor(X,a,2),arg(1,X,b),arg(2,X,c).
   X = a(b, c).
```

7.3 Suchraummanipulation

Suchraummanipulation

- Abschneiden unerwünschter Suchpfade
- z.B. Erzwingen eines eindeutigen Berechnungsergebnisses
- spezielles nullstelliges Prädikat $!/0$ (cut)
- Semantik:
 - legt die Suche auf diejenige Auswahl fest, die seit der Unifikation des Klauselkopfes entstanden ist, in der der cut auftritt.
 - ist immer erfolgreich.

Suchraummanipulation

- schneidet alle Suchpfade ab,
 - die sich aus alternativen Klauseln für das betreffende Prädikat ergeben, bzw.
 - die sich aufgrund alternativer Wertebindungen der bereits abgearbeiteten Teilziele im Klauselkörper ergeben.
- verhindert ein Backtracking in Teilziele, die links vom cut stehen
- macht alle Klauseln, die in der Datenbank auf den cut folgen unsichtbar

Suchraummanipulation

$a(X) :- b(X,Y), c(Y).$

$a(X) :- d(X).$

$b(a,b).$

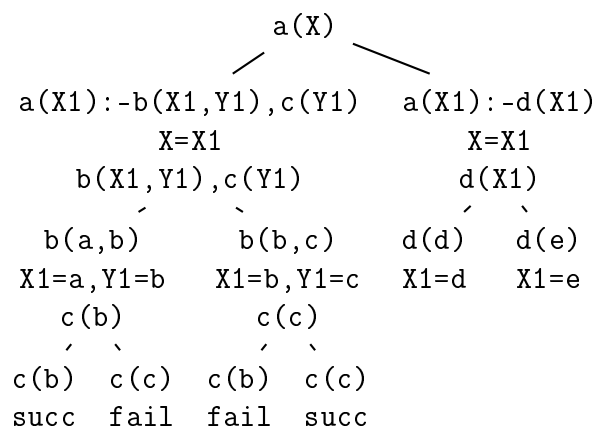
$b(b,c).$

$c(b).$

$c(c).$

$d(d).$

$d(e).$



Suchraummanipulation

a(X) :- b(X,Y), !, c(Y).

a(X) :- d(X).

b(a,b).

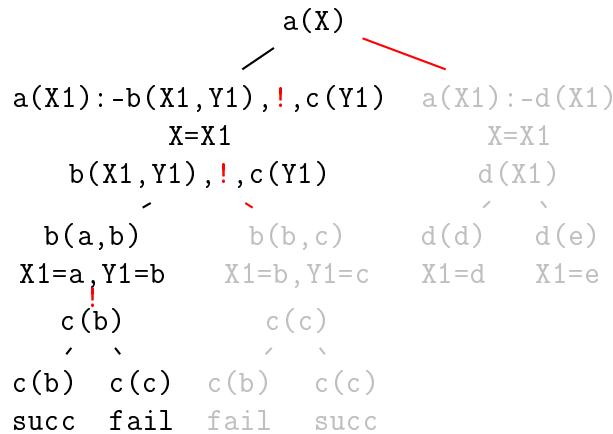
b(b,c).

c(b).

c(c).

d(d).

d(e).



Suchraummanipulation

Pragmatik:

- green cut: deterministische Programmierung in eindeutigen Kontexten
 - Effizienzsteigerung bei unveränderter Prädikatssemantik
- red cut: eine mehrdeutige Relation wird funktional
 - immer mit Änderung der Programmsemantik verbunden

Beispiele: green cut

- Verhindern unnötiger Klauselaufrufe bei disjunkten Fallunterscheidungen

```
% maximum (+X,+Y,?Max)
% Max ist Max von X und Y
maximum(X,Y,X) :- X>Y.
maximum(X,Y,Y) :- Y>=X.
```

- falls erste Klausel erfolgreich, scheitert zweite Klausel zwangsläufig

```
% maximum (+X,+Y,?Max)
% Max ist Max von X und Y
maximum(X,Y,X) :- X>Y, !.
maximum(X,Y,Y) :- Y>=X.
```

Beispiele: green cut

- kann Bedingung in zweiter Klausel ganz entfallen?

```
% maximum1 (+X,+Y,?Max)
% Max ist Max von X und Y
maximum1(X,Y,X) :- X>Y, !.
maximum1(X,Y,Y).
```

- ~~Verlust einer Instanziierungsvariante!~~

```
?- maximum1(3,2,M).
M = 3.
?- maximum1(3,2,3).
true.
?- maximum1(3,2,2).
true.
```

- → red cut
- korrigiertes Prädikatsschema: `maximum1(+X,+Y,-Max)`
- vgl. if-then-else

Das hier angegebene Beispiel zeigt sehr schön, dass die unüberlegte Verwendung des cut erhebliche Konsequenzen für die Korrektheit eines Programms in allen seinen Instanziierungsvarianten haben kann. Der cut sollte daher auch nur in solchen Situationen verwendet werden, in denen sich seine Auswirkungen vollständig überschauen lassen, z.B. wenn durch das Prädikat ein eindeutiger Berechnungszusammenhang spezifiziert werden soll. Im Abschnitt 8.2 werden Sie anhand des Konditionals `if ... then ... else` noch Möglichkeiten kennenlernen, den cut unter ganz kontrollierten Bedingungen einzusetzen.

Beispiele: red cut

- Verhindern unnötiger Klauselaufrufe bei garantiert eindeutigem Ergebnis

```
% ist_mutter(+name) Instanziierungsabhängig!
% name ist Mutter
ist_mutter(Mutter) :-
    elternteil_von(Mutter,_),
    weiblich(Mutter), !.
```

- cut verhindert aussichtslose Suche nach anderen Müttern

- aber: relationaler Charakter geht verloren
Aufzählen der Mütter ist nicht mehr möglich

Beispiele: red cut

- Verlust von alternativen Berechnungsergebnissen

```
% mutter_von(?name1,?name2)
mutter_von(Mutter,Kind) :-
    elternteil_von(Mutter,Kind),
    weiblich(Mutter), !.
```

- terminiert nach dem ersten Suchergebnis

```
elternteil_von(susi,hans).
elternteil_von(susi,anna).
```

```
?- mutter_von(susi,X).
    X = hans.
```

Anwendung: Defaultschließen

- Normalfall wird durch Ausnahmen außer Kraft gesetzt
- Normalfall: Tiere haben 6 Beine

```
hat_beine(vogel,2).           % Ausnahme
hat_beine(spinne,8).          % Ausnahme
hat_beine(lurch,4).           % Ausnahme
hat_beine(saeugetier,4).       % Ausnahme
hat_beine(_,6).               % Normalfall
```

Anwendung: Defaultschließen

- reine Faktensammlung ergibt unerwünschte Mehrfachresultate

```
?- hat_beine(spinne,Beine).
    Beine = 8 ;
    Beine = 6.
```

```
?- hat_beine(Tier,4).
    Tier = Lurch ;
    Tier = Saeugetier.
```

Anwendung: Defaultschließen

- Unterdrücken der Mehrfachresultate durch Cut:

```

hat_beine(vogel,2) :- !.           % Ausnahme
hat_beine(spinne,8) :- !.         % Ausnahme
hat_beine(lurch,4) :- !.          % Ausnahme
hat_beine(primate,2) :- !.        % Ausnahme
hat_beine(saeugetier,4) :- !.     % Ausnahme
hat_beine(_,6).                   % Normalfall

```

```

?- hat_beine(spinne,Beine).
   Beine = 8.

```

```

?- hat_beine(Tier,4).
   Tier = lurch.

```

- aber: funktional in beiden Argumenten
- eine Instanziierungsvariante geht verloren
- Reihenfolge der Klauseln ist signifikant

Defaultschließen

- Defaultschließen ist ein wichtiger Spezialfall des nichtmonotonen Schließens

Monotonie

Durch die Hinzunahme neuer Axiome wächst die Zahl der ableitbaren Theoreme monoton

$$A_1 \subset A_2 \wedge A_1 \models T_1 \wedge A_2 \models T_2 \rightarrow T_1 \subseteq T_2$$

- bei der Erweiterung einer Theorie gehen keine Schlussfolgerungen verloren

Nichtmonotonie

Durch die Hinzunahme neuer Axiome verringert sich die Zahl der ableitbaren Theoreme

- bei der Erweiterung einer Theorie können Schlussfolgerungen verloren gehen

Monotonie

```
hat_beine(_,6).
```

```
?- hat_beine(regenwurm,B).  
   B = 6.
```

```
hat_beine(regenwurm,0).  
hat_beine(_,6).
```

```
?- hat_beine(regenwurm,B).  
   B = 0 ;  
   B = 6.
```

Nichtmonotonie

```
hat_beine(_,6).
```

```
?- hat_beine(regenwurm,B).  
   B = 6.
```

```
hat_beine(regenwurm,0) :- !.  
hat_beine(_,6).
```

```
?- hat_beine(regenwurm,B).  
   B = 0.
```

7.4 Extralogische Prädikate

Datenbankmanipulation

- Eintragen von Klauseln in die Datenbank

<code>consult(+File)</code>	Einlesen aus File
<code>assert(+Term)</code>	Eintragen einer Klausel ...
<code>assert(+Term, -Reference)</code>	
<code>asserta, assertz</code>	... am Anfang / ... am Ende

- Entfernen von Klauseln aus der Datenbank

```
abolish(+Functor, +Arity)
retract(+Head)
retractall(+Head)
```

- Suche von Klauseln in der Datenbank

```
clause(?Head, ?Body)
nth_clause(?Predicate, ?Index, ?Reference)
```

Globale Variable

- globale Variable

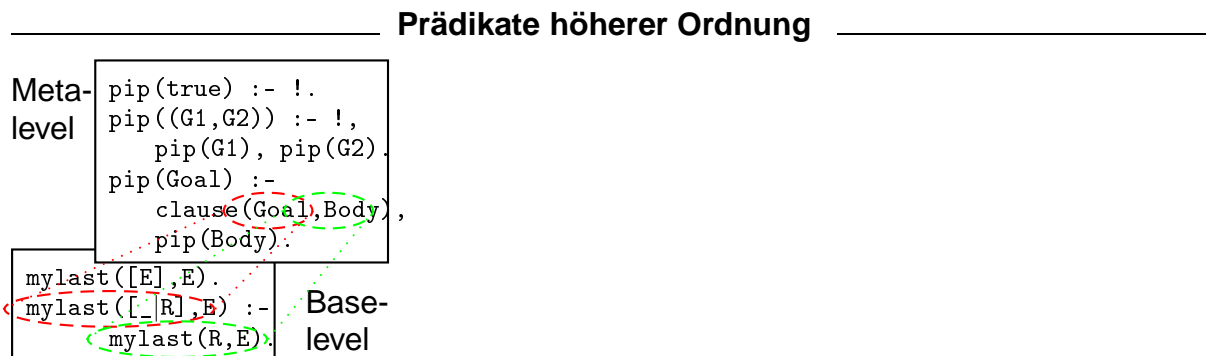
```
% flag(+Key, -OldInteger, +NewInteger)}
?- flag(mycounter, N, N+1)
   N = 0.

?- flag(mycounter, N, N+1).
   N = 1.

?- flag(mycounter, N, N+1).
   N = 2.
```

- auch für nichtnumerische Atome

8 Prädikate höherer Ordnung



Während sich die im letzten Kapitel behandelten metalogischen Prädikate immer noch im Rahmen des Prädikatenkalküls erster Stufe bewegten, überschreiten wir nunmehr eine weitere Grenze. Wir beginnen mit der Betrachtung von Prädikaten höherer Ordnung, d.h. solchen, die selbst wieder Prädikatsdefinitionen oder Prädikatsaufrufe verarbeiten oder erzeugen. Den Abschluss unserer Exkursion in die Welt der Logikprogrammierung bildet dann die Metaprogrammierung, die uns die Möglichkeit bieten soll, nunmehr auch noch das letzte unberührt gebliebene Reservat auf den Kopf zu stellen: die Semantik unseres Programmiersystems selbst.

Prädikate höherer Ordnung

- ... erlauben Prädikatsaufrufe als Argumentbelegungen *und aktivieren diese als (Teil-)ziel*
 - Prädikatsaufruf
 - Negation
 - Steuerstrukturen
 - Ermittlung von Resultatsaggregationen
 - Konsistenzforderungen
 - ...
- ... können Prädikatsdefinitionen erzeugen und in die Datenbasis eintragen (über `assert/1`)
 - aber: Prädikatsdefinition erfolgt immer klauselweise!

8.1 Prädikatsaufruf

Prädikatsaufruf

- `call(+Struktur)`
- Semantik: startet die Suche mit einem (möglicherweise unterspezifizierten) Ziel (Struktur)

- Pragmatik: sinnvoll insbesondere:
 - im Zusammenhang mit dynamisch erzeugten Prädikaten und Prädikatsaufrufen
 - zur verzögerten Aktivierung von Zielen
 - Metaprogrammierung

Prädikatsaufruf

- Metacall: ist ein Teilziel im Körper einer Klausel eine Variable, so wird ihr Wert als Prädikatsaufruf interpretiert.

```
mycall(P) :- P.
```

```
?- mycall(length([a,b,c],X)).
X = 3.
```

- Beispiel: Definition der Disjunktion

```
% +X ; +Y:
% Disjunktion von zwei Zielen X und Y
X ; Y :- X.
X ; Y :- Y.
```

Das Prädikat `mycall/1` ist der einfachste (weil triviale) Metainterpreter, den man sich vorstellen kann: Er nimmt ein Ziel `P` und aktiviert es unter der originalen Semantik des Prolog-Systems. Im Abschnitt 8.4 werden Sie sehen, wie bei dieser Gelegenheit die Semantik des Interpreters ganz gezielt verändert werden kann.

Prädikatsaufruf

- `once(+Struktur)`
- Aufruf eines Prädikats ohne Backtracking

```
% once(+Ziel)
% Ruft Ziel ohne Backtracking
once(Goal) :- Goal, !.
```

- oftmals in eindeutige Versionen nichtdeterministischer Prädikate eingebaut, z.B.

```
memberchk(Elem,Liste) :-
    once(member(Elem,Liste)).
```

Negation

- einstelliges Prädikat `\+/1` bzw. `not/1`
`\+/1` auch als Präfixoperator vordefiniert
- `\+` Ziel ist wahr, wenn der Aufruf von `Ziel` scheitert

```
% \+(+Ziel)
% Prüft, ob Ziel scheitert
'\+'(Goal) :- Goal, !, fail.
'\+'(_).
```

```
?- \+(1=2).
true.
```

```
?- \+(1=1).
false.
```

Negation

- `\+/1` instanziiert sein Argument nicht!
 - Frage nach den Bedingungen, die ein Ziel scheitern lassen, ist nicht möglich.
 - doppelte Negation ändert das Systemverhalten.

```
?- last(X, [a,b,c]).  
X = c.
```

```
?- \+ last(X, [a,b,c]).  
false.
```

```
?- \+ \+ last(X, [a,b,c]).  
X = _G281.
```

Anhand dieser Beispielaufufe sehen wir, dass `\+/1` kein vollständiger Ersatz für die logische Negation sein kann: Es erlaubt uns nicht zu ermitteln, unter welchen Bedingungen, d.h. für welche Variablenbelegungen ein Ziel scheitert. Im zweiten Beispielaufuf wären das etwa die Listenelemente `a` und `b`, die uns das Backtracking doch eigentlich aufzählen sollte. Warum es das nicht tut, erkennen wir an der Definition von `\+/1`: Falls `\+/1` erfolgreich ist (2. Klausel) schaut es den Wert seines Arguments überhaupt nicht mehr an ...

Negation

- Die Negation prüft nur das Scheitern eines Ziels (negation as failure).
- Voraussetzung: Annahme einer abgeschlossenen Welt (closed world assumption)
- Negation as failure ist der einfachste Fall einer nichtmonotonen Logik (vgl. auch `cut/0`)

Negation

- Monotonie: Hinzunahme von zusätzlichen Axiomen führt nicht zum Verlust von Theoremen:

<code>a(X) :- b(X).</code>	<code>a(X) :- b(X).</code>
<code>b(a).</code>	<code>b(a).</code>
	<code>b(b).</code>
<code>?- a(a).</code>	<code>?- a(a).</code>
<code> true.</code>	<code> true.</code>
<code>?- a(b).</code>	<code>?- a(b).</code>
<code> false.</code>	<code> true.</code>

Negation

- Nichtmonotonie: Hinzunahme von Axiomen führt zum Verlust von Theoremen

<code>a(X) :- \+ b(X).</code>	<code>a(X) :- \+ b(X).</code>
<code>b(a).</code>	<code>b(a).</code>
	<code>b(b).</code>
<code>?- a(a).</code>	<code>?- a(a).</code>
<code> false.</code>	<code> false.</code>
<code>?- a(b).</code>	<code>?- a(b).</code>
<code> true.</code>	<code> false.</code>

8.2 Steuerstrukturen

Wenn man von der imperativen Programmierung her zur Logikprogrammierung kommt, vermisst man wohl einen Bereich am allermeisten: die Steuerstrukturen der strukturierten Programmierung Verzweigung (`if ... then ... else; case ...; usw.`) und Zyklus (`while ... do ...; for ... do ...; usw.`). Natürlich gibt es sie auch in Prolog, oder besser, sie können mit den Mitteln der Logikprogrammierung leicht rekonstruiert werden. Hierbei werden wir erkennen, dass zumindest für Letztere ein `cut` unverzichtbar ist, so wie wir das bereits bei der Negation gesehen haben. Mit anderen Worten: auch Verzweigungsstrukturen sind aus der Perspektive der Logikprogrammierung nichtmonoton. Wir wenden dieses Erkenntnis ins Positive: Steuerstrukturen können oftmals verwendet werden, ein `cut`, das für die Entwicklung überschaubarer und damit sicherer Programme nicht ganz unproblematisch ist, hinter einem Programmiersprachenkonstrukt mit intuitiv plausibler Semantik zu verbergen.

Steuerstrukturen

- Grundlegende Steuerstrukturen
 - Guarded Clauses
 - Rekursion
 - Cut
- zusätzliche Konstrukte
 - Konditionale
 - Iterationszyklen

Konditionale

- if-then

```
% +Condition -> +Action
If -> Then :- If, !, Then.
```

Achtung: wenn If scheitert, scheitert das gesamte Konditional!

- if-then-else:

```
% +Condition -> +Then_Act ; Else_Act
If -> Then ; _ :- If, !, Then.
_ -> _ ; Else :- !, Else.
```

Konditionale

- Soft-Cut: `*->` If-Ziel bleibt backtrackbar

```
a(1).          ?- a(X) -> Y=a ; Y=b.
a(2).          X=1, Y=a.
```

```
?- a(4) -> Y=a ; Y=b.
Y=b.
```

```
?- a(X) *-> Y=a ; Y=b.
X=1, Y=a ;
X=2, Y=a.
```

```
?- a(4) *-> Y=a ; Y=b.
Y=b.
```

Failure-gesteuerte Zyklen

- über eine Folge natürlicher Zahlen (between/3)

```
% tab(+Integer)
tab(N) :-
    between(1,N,_),
    put(' '),
    fail.
```

- über eine Kollektion

```
% write_items(+List).
write_items(List) :-
    member(X,List),
    write_ln(X),
    fail.
```

Failure-gesteuerte Zyklen

- über der Datenbank

```
loop :-
    mitarbeiter(Vorname,_,_,_,_),
    write_ln(Vorname),
    fail.

?- loop.
susi
hans
...
karl
false.
```

Failure-gesteuerte Zyklen

- Failure-gesteuerte Zyklen scheitern immer
 - ggf. muss alternative Klausel bereitgestellt werden

```
% write_items(+List).
write_items(List) :-
    member(X,List),
    write_ln(X),
    fail.
write_items(_) :- true.
```

Iteration

- **Iterator:** forall(+Cond,+Goal)
- Iteration über eine Folge natürlicher Zahlen

```
% tab(+Integer)
tab(N) :-
    forall(between(1,N,_),
        put(' ')).
```

- Äquivalent zur for-Schleife
- Iteration über eine Kollektion

```
% write_items(+List).
write_items(List) :-
    forall(member(X,List),
        write_ln(X)).
```

Iteration

- Überprüfung von Konsistenzforderungen: Erfüllen alle Resultate des Aufrufs von Ziel, die Bedingung?

```
?- forall(mitarbeiter(Name,Gehalt),Gehalt>0).
    true.
```

```
?- forall(member(E,[1,3,-2,1,2]),E>0).
    false.
```

Iteration

- forall/2 instanziiert keine Variablen
- Implementation durch doppelte Negation [3mm]

```
forall(Cond, Goal) :-
    \+ (Cond, \+ Goal).
```

- Falls eine Variablenbindung "nach außen" gegeben werden soll: foreach/2

Iteration

- foreach(+Cond,+Goal) ruft eine Konjunktion aller Instanzierungsvarianten von Goal auf, die sich aus dem Aufruf von Cond ergeben

```
?- foreach(between(1,3,X),member(X,Y)).
Y = [1, 2, 3|_G549] ;
Y = [1, 2, _G548, 3|_G552] ;
Y = [1, 2, _G548, _G551, 3|_G555] ;
Y = [1, 2, _G548, _G551, _G554, 3|_G558] .
```

- zum Vergleich

```
?- forall(between(1,3,X),member(X,Y)).
true.
```

Unbeschränkte Iteration

- Anwendung für Interaktionszyklen
Abbruch bei Eingabe von vereinbartem Endekennzeichen

```
loop :-
    repeat,
    read(X),
    echo(X), !.

echo(X) :- last_input(X), !.    % Abbruch
echo (X) :- write_ln(X), fail.

last_input(end).
```

Iteration

- Iteration erlaubt keinen Bezug auf vorangegangene Berechnungsergebnisse
- sinnvoll nur ...
 - ... falls Berechnungsergebnis irrelevant
 - * Überprüfung von Konsistenzbedingungen
 - ... im Zusammenhang mit Nebeneffekten
 - * Input/Output
 - * Manipulation der Datenbasis
 - * Verwendung globaler Variablen

8.3 Resultatsaggregation

Resultatsaggregation

- Aufsammeln aller Lösungen für ein Ziel: `findall(+Term,+Ziel,-Liste)`
- Semantik: sammelt alle Instanziierungsvarianten, die beim Aufruf von `Ziel` sukzessive für den Term `Term` erzeugt werden, in der Liste `Liste`.
- Pragmatik: `Term` sollte Variable enthalten, die mit Variablen in `Ziel` koreferenzieren

Resultatsaggregation

- die Datenbank

```
% ma(Vorname,Name,Abteilung,Position,Gehalt)
ma(susi,sorglos,verwaltung,sekretaerin,40000).
ma(hans,im_glueck,verwaltung,manager,900000).
ma(anne,pingelig,rechenzentrum,operator,50000).
ma(paul,kraft,montage,wartung,70000).
ma(karl,wunderlich,versand,fahrer,55000).
```

Resultatsaggregation

```
?- findall(name(Vorname,Name),
    ma(Vorname,Name,_,_,_),
    Namen).
Name = _G123,
Vorname = _G234,
Namen =
    [name(susi,sorglos), name(hans,im_glueck),
     name(anne,pingelig), name(paul,kraft),
     name(karl,wunderlich)].
```

Resultatsaggregation

```
?- findall(Name,
    ma(_,Name,verwaltung,_,_),Namen).
    Name = _G123, Namen = [sorglos,im_glueck].

?- findall(Abteilung,
    (ma(_,_,Abteilung,_,Gehalt),Gehalt>50000),
    Abteilungen).
    Abteilung = _G123,
    Gehalt = _G234,
    Abteilungen = [verwaltung, montage, versand].
```

Resultatsaggregation

```
?- findall(Gehalt,
    ma(_,_,_,_,Gehalt),Gehaelter),
    min-list(Minimum,Gehaelter).
    Gehalt = _G123,
    Gehaelter = [40000,900000,50000,70000,55000],
    Minimum = 55000.

?- findall(Name,
    ma(_,Name,verwaltung,_,_),
    Namen), length(Anzahl,Namen).
    Name = _G123, Namen = [sorglos, im_glueck],
    Anzahl = 2.
```

Resultatsaggregation

- Aufsammeln aller Lösungen gruppiert nach identischen Bindungen für freie Variable: `bagof(+Term,+Ziel,-Liste)`

```
?- bagof(m(V,N,P,G),ma(V,N,A,P,G),L).
    N = _G123, ... , A = verwaltung,
    L = [m(susi, sorglos, sekretaerin, 40000),
        m(hans, im_glueck, manager, 900000)] ;
    N = _G123, A = rechenzentrum,
    L = [m(anne, pingelig, operator, 50000)] ;
    N = _G123, ... , A = montage,
    L = [m(paul, kraft, wartung, 70000)] ;
    N = _G123, ... , A = versand,
    L = [m(karl, wunderbar, fahrer, 55000)].
```

Resultatsaggregation

- Ignorieren irrelevanter Variablenbelegungen:

– neuer Infixoperator: +Variable \wedge +Ziel

```
?- bagof(V, N^P^G^ma(V,N,A,P,G), L).
   N = _G123, ..., A = verwaltung,
   L = [susi,hans] ;
   N = _G123, ..., A = montage,
   L = [paul] ;
   N = _G123, ..., A = rechenzentrum,
   L = [anne] ;
   N = _G123, ..., A = versand,
   L = [karl].
```

Resultatsaggregation

- Ermitteln von Lösungsmengen: setof(+Term,+Ziel,-Liste)
- Semantik: Wie bagof/3, aber Ergebnislisten sind lexikalisch sortiert und duplikatenfrei

```
?- setof(V, N^P^G^ma(V,N,A,P,G), L).
   N = _G123, ..., A = verwaltung,
   L = [hans,susi] ;
   N = _G123, ..., A = montage, L = [paul] ;
   N = _G123, ..., A = rechenzentrum,
   L = [anne] ;
   N = _G123, ..., A = versand, L = [karl].
```

Resultatsaggregation

- verallgemeinerte Resultatsaggregation: [2mm] aggregate_all(+Template,+Goal,?Result)
[3mm]
- Beispiele für Templates [2mm]

count	zählt die Anzahl der Lösungen
sum(Expr)	ermittelt die Summe aller Lösungen für Expr
min(Expr)	ermittelt das Minimum aller Lösungen für Expr
max(Expr)	ermittelt das Maximum aller Lösungen für Expr
bag(X)	berechnet eine Liste aller Lösungen für x
set(X)	berechnet eine geordnete Menge aller Lösungen für x

Resultatsaggregation

```
?- aggregate_all(count,member(X,[1,2,3,2,1]),L).
L = 5.
?- aggregate_all(min(X),member(X,[1,2,3,2,1]),L).
L = 1.
?- aggregate_all(max(X),member(X,[1,2,3,2,1]),L).
L = 3.
?- aggregate_all(sum(X),member(X,[1,2,3,2,1]),L).
L = 9.
?- aggregate_all(set(X),member(X,[1,2,3,2,1]),L).
L = [1, 2, 3].
?- aggregate_all(bag(X),member(X,[1,2,3,2,1]),L).
L = [1, 2, 3, 2, 1].
```

Resultatsaggregation

- Variante `aggregate_all/4` erlaubt die Angabe eines Diskriminators an der zweiten Argumentposition, wodurch Mehrfachbindungen für eine Variable berücksichtigt werden

```
a(1,a).
a(2,a).
a(1,b).
a(2,b).
a(3,b).

?- aggregate_all(count,a(_,_),Anz).
Anz = 5.
?- aggregate_all(count,X,a(X,_),Anz).
Anz = 3.
?- aggregate_all(count,Y,a(_,Y),Anz).
Anz = 2.
```

Resultatsaggregation

- Verhindert, dass bei m:n-Relationen Ergebnisse verloren gehen

```
?- aggregate_all(sum(X),a(X,Y),R).
R = 9.
?- aggregate_all(sum(X),X,a(X,Y),R).
R = 6.
?- aggregate_all(sum(X),Y,a(X,Y),R).
R = 9.
```

8.4 Metaprogrammierung

Metaprogrammierung

- Metaprogrammierung: Prolog-Programme verarbeiten andere Prolog-Programme als Daten
- Anwendungen:
 1. Analyse von Programmen
 - Pretty-Printer
 2. Transformation von Programmen
 - Programmoptimierung (z.B. Tailrekursion, delayed deduction)
 - Programmcompilierung
 - Übersetzen einer DCG → Differenzlistennotation
 3. Simulation der Programmabarbeitung: Meta-Interpreter

Metaprogrammierung

- Meta-Interpreter: Interpreter für eine Programmiersprache, der in der Programmiersprache selbst geschrieben ist
- einfachster Meta-Interpreter (`call/1`)

```
rufe(Goal):-Goal.
```

Prolog in Prolog

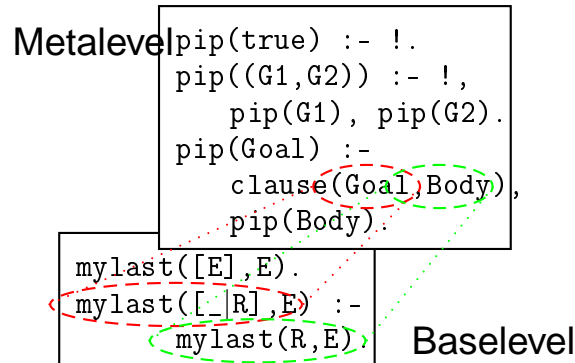
- Meta-Interpreter zur Simulation des Berechnungsmodells von Prolog
- Grundstruktur tritt in allen Anwendungen wieder auf
- eingeschränkte Syntax: Körper ist Konjunktion von positiven Literalen

Prolog in Prolog

```
% pip(+Ziel)
pip(true) :- !.
pip((G1,G2)) :-
    !,
    pip(G1),          % Zerlegung einer Konjunktion
    pip(G2).          % von Teilzielen
pip(Goal) :-          % Aktivierung eines Teilziels
    clause(Goal,Body),
    pip(Body).
```

```
?- pip(mylast(L,a)).
L = [a] ;
L = [X1, a] ;
L = [X2, X1, a] ;
...
```

Prolog in Prolog



Prolog in Prolog

- Anwendungen:
 - eigene Trace- und Debug-Werkzeuge
 - Interpreter mit veränderter operationaler bzw. denotationeller Semantik → modifizierte Inferenzregel
- Standardinferenzregel:
 - Ableitbarkeit eines Ziels
 - * aus den in der Datenbasis abgespeicherten Klauseln und
 - * mit Hilfe der SLD-Inferenzstrategie: Top Down - Tiefe zuerst - Links-Rechts

Prolog in Prolog

- alternative Inferenzstrategien: Modifikation der operationalen Semantik
 - bottom-up Suche (z.B. Datalog, Parsing)
 - Breite-zuerst-Verarbeitung
 - parallele Bearbeitung von Teilzielen
- eingeschränkte Resolutionsverfahren: z.B. Zyklenüberwachung
- inferentiell erweiterte Klauselmenge: z.B. durch Vererbung (Modifikation der denotationellen Semantik)

Ablaufprotokollierung

- visible Prolog: Trace der Programmabarbeitung
 - Ausgabe der aktivierten Teilziele
 - Visualisierung der rekursiven Einbettung

Ablaufprotokollierung

```
% vip(+Ziel)
vip(G) :- vi('',G). % Init. der Einrücktiefe

vi(_,true) :- !.
vi(Indent,(G1,G2)) :-
    !,
    vi(Indent,G1), % Zerlegung eines Teilziels
    vi(Indent,G2).
vi(Indent,G1) :-
    clause(G1,Body), % Suche einer relev. Klausel
    write(Indent), % Ausgabe der Einrückung
    write(' ?- '),
    write(G1),nl, % Ausg. des aktiven Teilziels
    atom_concat(Indent,' ',NewIndent), % Einrück.
    vi(NewIndent,Body). % rekursiver Aufruf
```

Ablaufprotokollierung

```
?- vip(mylast([a,b,c],E)).
?- mylast([a, b, c],_G198)
?- mylast([b, c]_G198)
?- mylast([c],c)
E = c ;
?- mylast([c]_G198)
false.
```

Ablaufprotokollierung

```
?- vip(mylast(X,a)).
?- mylast([a], a)
X = [a] ;
?- mylast([_G255|_G256], a)
?- mylast([a], a)
X = [_G255, a] ;
?- mylast([_G274|_G275], a)
?- mylast([a], a)
X = [_G255, _G274, a] ;
?- mylast([_G293|_G294], a)
?- mylast([a], a)
X = [_G255, _G274, _G293, a] .
```

Ablaufprotokollierung

- Erweiterung:
 - Ausgabe der zum Vergleich herangezogenen Klauseln und des Unifikationsresultats
 - Protokollieren auch der erfolglosen Klauselaufufe
- Trennen:
 - Suche nach einer relevanten Klausel
 - Unifikation mit dem Klauselkopf

Ablaufprotokollierung

```

vi2(Indent,G1) :-
    % Erzeugen eines uninstantiierten Teilziels
    G1=..[Pred|Args1],                % functor(G1,F,N)
    length(Args1,Length),
    length(Args2,Length),
    G2=..[Pred|Args2],                % functor(G2,F,N)
    % Suche nach einer passenden Klausel
    find_clause(Indent,G2,Body,Pred),
    write(Indent), write(' ?- '), write(G1),
    % Ausgabe: Klauselkopf
    write('   clause: '),
    write(G2),
    % Matching des Klauselkopfes
    (G1 = G2 ->
        write('   S'), nl ;
        write('   F'), nl, fail),
    concat(Indent,' ', NewIndent),
    vi2(NewIndent,Body).

```

Ablaufprotokollierung

- Finden einer relevanten Klausel

```

% find_clause(+Indent,+Query,-Body,
%   +PredicateNameOfQuery)
find_clause(_,Q,Body,_) :-
    % Suche einer passenden Klausel
    clause(Q,Body).
find_clause(Indent,_,_,Pred) :-
    % Klauselmenge vollstaendig abgearbeitet
    write(Indent),
    write(': End of definition: '),
    write(Pred), nl, fail.

```

Ablaufprotokollierung

```
?- vip2(mylast(E,[a,b,c])).
?- mylast([a, b, c],_G204)   clause: mylast([_G321],_G321)   F
?- mylast([a, b, c],_G204)   clause: mylast([_G345|_G346],_G321)   S
?- mylast([b, c],_G204)     clause: mylast([_G361],_G361)   F
?- mylast([b, c],_G204)     clause: mylast([_G385|_G386],_G361)   S
?- mylast([c],_G204)       clause: mylast([_G401],_G401)   S
E = c ;
?- mylast([c],_G204)       clause: mylast([_G425|_G426],_G401)   S
?- mylast([],_G204)       clause: mylast([_G441],_G441)   F
?- mylast([],_G204)       clause: mylast([_G465|_G466],_G441)   F
: End of definition: mylast
: End of definition: mylast
: End of definition: mylast
: End of definition: mylast
false.
```

Zyklenüberwachung

- Memoization:
 - Verwalten einer Abarbeitungsgeschichte (History): Liste aller bisher aktivierten Teilziele
 - Überprüfen, ob aktuelles Teilziel bereits in der History registriert wurde

Zyklenüberwachung

```
% ldi(+Goal)
% Zyklenerkennender Interpreter
ldi(Goal) :- ld([ ],Goal). % leere History
ld(_,true) :- !.
    % true ist immer wahr
ld(History,(Subgoal1,Subgoal2)) :- !,
    % Zerlegung einer Konjunktion
    ld(History,Subgoal1),
    ld(History,Subgoal2).
ld(History,Subgoal) :-
    % Bearbeitung eines einfachen Teilziels
    not(unifiable(Subgoal,History,3)),
    % keine Unifizierbarkeit mit drei
    % aufeinanderfolgenden Elementen der History
    clause(Subgoal,Body), % relevante Klausel
    ld([Subgoal|History],Body). % rekursiver Aufruf
    % mit erweiterter History und Klauselkörper
```

Zyklenüberwachung

- Hilfsprädikat: Unifizierbarkeit mit einem Element der Ableitungsgeschichte

```
% unifiable(+Goal,+History,+N)
% Goal ist mit N aufeinanderfolgenden
% Elementen der History unifizierbar
unifiable(_,_,0) :- !.
    % Abbruch: maximale Anzahl erreicht
unifiable(Goal,[Goal|Hrest],N) :-
    % rekurs. Abstieg mit Kons. der History
    N1 is N - 1, !,      % und der max. Anzahl
    unifiable(Goal,Hrest,N1).
```

Zyklenüberwachung

- Datenbasis

```
isa(a,b).
isa(b,c).
isa(X,Z):-isa(X,Y),isa(Y,Z).
```

- Testaufrufe

<pre>?- isa(a,X). X = b ; X = c ; ...</pre>	<pre>?- ldi(isa(a,X)). X = b ; X = c .</pre>
---	--

Zyklenüberwachung

- Modifizierte Inferenzregel:
- Ableitbarkeit eines Ziels
 - aus den in der Datenbasis abgespeicherten Klauseln und
 - mit Hilfe der SLD-Inferenzstrategie, falls dabei nicht mehr als n-mal in direkter Folge jeweils miteinander unifizierbare Teilziele aktiviert werden
- Lösung deckt nicht alle Fälle ab
 - nur Zyklen, die durch direkte Rekursion entstehen
- Lösung ist zu rigide:
 - Zyklen zur Erzeugung von Nebeneffekten werden auch unterbunden

Zyklenüberwachung

- Beispiel für indirekte Rekursion:

```
isa(a,b).  
isa(b,c).  
isa(X,Y):-is1(X,Y).  
is1(X,Y):-isa(X,Z),isa(Z,Y).
```

```
?- ldi(isa(a,X)).  
X = b ;  
X = c ;  
. . .
```


9 Funktionale Programmierung

Funktionale Programmierung

$$(\lambda(x).x + 2)$$

$$((\lambda(x).x^2)$$

$$3)$$

$$\implies 11$$

9.1 Grundbegriffe

Funktionale Programmierung

- Funktion: eindeutige Abbildung aus einer Definitionsmenge in eine Zielmenge

$$y = f(x) \qquad x \mapsto y$$

- Applikation: Auswertung funktionaler Ausdrücke
 - Anwendung der Funktionsdefinition auf ein Wertetupel
 - Ermittlung des Funktionswerts für die gegebenen Argumentbelegungen
- Funktionale Programmierung: Berechnung durch Ermittlung von Funktionswerten

Mit der funktionalen Auswertung ist ein Verarbeitungsmodell gegeben, das wegen seiner zentralen Bedeutung für die Lösung numerischer Aufgabenstellungen unverzichtbarer Bestandteil aller universellen Programmiersprachen ist. Genau in dieser Rolle haben wir ja bereits die funktionale Auswertungsumgebung als hybrides Element für arithmetische Berechnungen im Rahmen der Logikprogrammierung kennengelernt. In diesem Abschnitt werden wir nunmehr einen Eindruck davon bekommen, wie die funktionale Programmierung auch als leistungsfähiges Werkzeug zur symbolischen Informationsverarbeitung eingesetzt werden kann. Viele der Implementationstechniken zur rekursiven Verarbeitung rekursiver Datenstrukturen, die Sie in der Logikprogrammierung kennengelernt haben, lassen sich ohne Schwierigkeiten in äquivalente funktionale Lösungen umsetzen. Verloren geht dabei natürlich die Richtungsunabhängigkeit, da ja der Begriff der Funktion gerade auf eine (gerichtete) Abbildung zurückgeführt wird und daher inhärent asymmetrisch ist.

Der gemeinsame Vorläufer aller funktionalen Programmiersprachen ist die Sprache Lisp, die in Form von Common Lisp als *de facto*-Standard auch heute noch eine wichtige Rolle spielt. Wir werden die funktionale Programmierung am Beispiel der Sprache Scheme betrachten, die sich wie auch die anderen Mitglieder der Lisp-Sprachenfamilie durch eine fast triviale Syntax auszeichnet. Vor allem aber besitzt sie dank der durchgängigen Verwendung einiger weniger Grundprinzipien eine besonders einfache semantische Struktur und ist daher für die Zwecke der Lehre besonders gut geeignet. Wir werden uns im Folgenden mit der Semantik der Sprache vertraut machen, indem wir die hervorragende Eignung von Prolog zur Definition von Metainterpretern nutzen und schrittweise den Kern eines Scheme-Interpreters in Prolog implementieren.

Scheme

- quasi-Standard: Revised Report on the Algorithmic Language Scheme (KELSEY ET AL. 1998)
- möglichst einfach gehalten
- viele Einzelheiten unspezifiziert → zahlreiche Implementationsunterschiede
- Systeme im Informatik-Rechenzentrum
 - MIT-Scheme
 - Dr. Scheme

Scheme

- (leicht) vereinfachte Syntax

Ausdruck	:=	Konstante Name s-Ausdruck
Konstante	:=	Zahl Wahrheitswert
Wahrheitswert	:=	#t #f
Variable	:=	Name
s-Ausdruck	:=	Liste gepunktetes Paar
Liste	:=	'()' '(' Elemente ')'
Elemente	:=	Element Elemente
Element	:=	Ausdruck
gepunktetes Paar	:=	'(' Element '.' Element ')'
- keine Makros
- keine reservierten Symbole für `quote` (und `backquote`)

Das Verarbeitungsmodell von Scheme kennt nur wenige Typen von Ausdrücken, wobei jeder Ausdruck einen Wert besitzt (und sei es der spezielle Wert `#undefined`). Ein Scheme-Interpreter durchläuft dementsprechend stets einen elementaren Interaktionszyklus, in dem er — ähnlich dem Prolog-System — syntaktisch korrekte Ausdrücke vom Nutzer entgegennimmt und deren Wert ermittelt. Mit Hilfe von Ausdrücken

können so Standardfunktionen des Systems, z.B. + für die Addition von Zahlenwerten aufgerufen werden.

Durch die Eingabe von geeigneten Ausdrücken kann der Nutzer (unter Rückgriff auf die Standardfunktionen) auch eigene Funktionen definieren und diese mit bestimmten Parametern aufrufen, um so aufgabenspezifische Abstraktionen für die jeweilige Problemstellung zu schaffen bzw. Berechnungsprozesse anzustoßen.

Scheme

- Wertesemantik
- Konstanten haben sich selbst als Wert

```
2 ==> 2
#t ==> #t
```

- Variablen haben den Wert, der Ihnen (z.B. mit `define`) zugewiesen wurde

```
x ==> error: unbound variable
(define x 2) ==> #undefined
x ==> 2
```

- der Wert von s-Ausdrücken muss *berechnet* werden (z.B. durch funktionale Applikation)

Die Auswertung eines funktionalen Ausdrucks erfolgt immer in einer *Umgebung*. Eine Umgebung bestimmt die in ihr gültigen Variablen und deren Wertebindungen. Daher muss das Prädikat zur Auswertung dreistellig angelegt werden: Es nimmt einen Ausdruck, sowie die Umgebung in der er ausgewertet werden soll, entgegen und gibt den Wert des Ausdrucks auf der dritten Argumentstelle zurück.

Scheme in Prolog (1): Auswertung

- zentrales Auswertungsprädikat `eval/3`
- wird sukzessive definiert

eval/3 (1): Auswertung einer Konstanten

```
%%% eval(Expression, Environment, Value)
% self-evaluating expressions
eval(Exp,_,Exp) :- number(Exp), !.
eval('#t',_, '#t') :- !.
eval('#f',_, '#f') :- !.
```

```
?- eval(2,_,X).
   X = 2.
?- eval('#t',_,X).
   X = '#t'.
```

Obwohl Prolog mit den Operatoren über weitreichende Möglichkeiten zur nutzerge- steuerten Manipulation der Syntax von Termen verfügt, ist es mit der Lisp-Syntax überfordert: Angeblich wurden aufgrund eines Programmierfehlers in einer sehr frü- hen Lisp-Version Kommas nicht mit ausgedruckt, da das System ihren internen Zei- chenkode auf ein undruckbares Zeichen abgebildet hatte. Dadurch wurde das Kom- ma zu einem "Unzeichen": Die Syntax wurde dahingehend modifiziert, dass auch das Leerzeichen als Separator für Listenelemente fungieren kann und damit entfiel jede Notwendigkeit zur Nutzung des Kommas.

Leider wurde durch diesen Trick das Leerzeichen aber semantisch überladen: Neben seiner üblichen Funktion als Füllelement zwischen Listenelementen dient es auch als Listenseparator, zwei funktionale Rollen, die in Prolog sauber getrennt sind. Daher ist es unmöglich, einem Leerzeichen in Scheme automatisch eine der beiden Bedeu- tungen in Prolog zuzuordnen. Insbesondere kann nicht jedes Leerzeichen als Listen- elementseparator interpretiert werden. Hierfür müsste ein separater Konverter imple- mentiert werden, der jeden Ausdruck erst in eine Prolog-konforme Syntax umwandelt. Da ein solcher Konverter aber wenig zum Verständnis der Semantik eines Scheme- Programmes beiträgt, wurde hier der einfachere Weg beschritten: Es wird eine Prolog- basierte Syntax für Scheme-Ausdrücke verwendet, in der das Komma explizit zur Se- parierung der Listenelemente eingesetzt wird.

Scheme in Prolog (2): Notation

- Notation

- Lisp-Notation ist historischer Unfall
- Leerzeichen ist überladen:
 - * Separation von Listenelementen (→ Komma)
 - * Füllmaterial zwischen Syntaxelementen (→ Leerzeichen)

→ Repräsentation von s-Ausdrücken als Prolog-Listen

Scheme	2	name	Name	(a b c)	(a . b)
P-Scheme	2	name	name	[a,b,c]	[a b]

Variable

- Ein Name identifiziert eine Variable (Symbol), deren Wert ein (abstraktes) Objekt ist.
- Benennung als einfachste Form der Abstraktion: Man sieht einem Wert nicht mehr an, wie er entstanden ist.

```
(define <name> <konstante>)
```

```
(define <name> <s-expression>)
```

- `define` ist eine *special form expression*
- Wert des `define`-Aufrufs ist undefiniert

Variable

- die funktionale Programmierung unterstützt im Prinzip keine Wertmanipulation ...
- ... lässt aber verschiedene Hintertüren offen: `define`, `set!`
- beide entsprechen der Wertzuweisung in den imperativen Sprachen
- Namen dürfen nicht doppelt belegt werden
- betrifft insbesondere
 - Standardfunktionen: `sqrt`, `sin`, `append`, ...
 - Syntaktische Schlüsselwörter: `and`, `begin`, `case`, `cond`, `define`, `delay`, `do`, `else`, `if`, `lambda`, `let`, `let*`, `letrec`, `or`, `quasiquote`, `quote`, `set!`, ...
 - Konstanten: `#t` (true), `#f` (false)
- Interpunktionszeichen: `'(, ')`, `'''` und `'''` können nicht Teil eines Namens sein

Das Variablenkonzept der funktionalen Programmierung ähnelt dem der Logikprogrammierung: Variable erhalten ihren Wert im Prozess des Prozeduraufrufs und *sollten* ihn danach nicht mehr ändern, um ein seiteneffektfreies Programmieren und damit den deklarativen Charakter der Sprache zu garantieren. Allerdings werden in Scheme aber auch diejenigen Mittel zur Verfügung gestellt, die für solche Manipulationen erforderlich sind: Während `define` sowohl zur Zuweisung eines Initialwerts als auch zum Ändern einer Wertbindung verwendet werden kann, steht `set!` nur für Letzteres zur Verfügung (ist dafür aber weniger eingeschränkt im Hinblick auf die Positionen, an denen es verwendet werden kann). Mit diesen Konstrukten kann der deklarative Charakter der Sprache vollständig ausgehebelt werden.

Verglichen damit ist das Variablenkonzept der Logikprogrammierung weitaus weniger zugänglich für derartige Manipulationen. Lokale Variable können ausschließlich durch Unifikation gebunden werden und eine Änderung der Variablenbindung ist nur durch das Backtracking möglich. Damit sind alle Wertebindungen vollständig vor dem Zugriff des Programmiers geschützt. Die einzigen Möglichkeiten zur Zustandsänderung eines Programms führen über die Datenbasis des Systems, wo neue Klauseln eingefügt (`assert/1`) bzw. gelöscht werden können (`retract/1`). Darüberhinaus können über das Prädikat `flag/3` auch globale Variable verwaltet werden. In jedem Fall sind derartige seiteneffektbehaftete Mechanismen strikt von der normalen Variablenbehandlung des Systems getrennt.

s-Ausdrücke

Funktionsaufrufe (in Präfix-Notation)

`(⟨Funktionsname⟩ . ⟨Liste von Argumenten⟩)`

`(sqrt (+ (expt x 2) (expt y 2)))`

`[sqrt, [+ , [expt, x, 2], [expt, y, 2]]]`

lambda-ausdrücke

`(lambda ⟨Liste von Argumenten⟩ . ⟨Liste von s-Ausdrücken⟩)`

`(lambda (x y) (sqrt (+ (expt x 2) (expt y 2))))`

`[lambda, [x, y], [sqrt, [+ , [expt, x, 2], [expt, y, 2]]]]`

s-Ausdrücke

special form expressions

special form expressions definieren eine spezielle Auswertungsreihenfolge

- `define` ist eine *special form expression*

Unter den s-Ausdrücken unterscheiden wir zwischen normalen Funktionsaufrufen, die ihre Elemente in einer allgemeingültigen Reihenfolge auswerten, die allerdings modifizierbar und leider auch systemspezifisch ist. Darüberhinaus existieren *special form expressions*, die von dieser Reihenfolge abweichen, indem sie bestimmte Argumentpositionen ganz von der Auswertung ausschließen oder deren Auswertung erst unter bestimmten Bedingungen veranlassen.

Funktionen

- Funktionsdefinition als Abstraktion
- Funktionen müssen keinen Namen haben

Name wird "nur" für Dokumentation, Wiederverwendung und den rekursiven Aufruf benötigt

- lambda-Ausdrücke sind anonyme (unbenannte) Funktionen

Anonyme Funktionen haben in der Logikprogrammierung kein Äquivalent. Jede Prädikatsdefinition muss in der Datenbasis über ihren Prädikatsnamen zugreifbar sein. Das erschwert die Nutzung von dynamisch erzeugten Prädikaten erheblich, da sie vor ihrem Aufruf erst in die Datenbasis eingetragen werden müssen, wobei immer mit der Gefahr von (globalen) Namenskonflikten gerechnet werden muss. Diese Probleme werden bei der Verwendung von anonymen Funktionen vollständig vermieden.

Funktionen

Verwendung von lambda-Ausdrücken zur Definition einer benannten Funktion

```
(define pythagoras (lambda (x y)
  (sqrt (+ (expt x 2) (expt y 2)))))

[define, pythagoras, [lambda, [x, y],
  [sqrt, [+ , [expt, x, 2], [expt, y, 2]]]]]
```

Kurzversion ohne lambda

```
(define (pythagoras x y)
  (sqrt (+ (expt x 2) (expt y 2))))

[define, [pythagoras, x, y],
  [sqrt, [+ , [expt, x, 2], [expt, y, 2]]]]
```

Zuweisung eines lambda-Ausdrucks zu einem Namen

Funktionsauswertung

eval: Auswerten *aller* Listenelemente

apply: Anwenden des Evaluationsergebnisses für das erste Listenelement auf die Evaluationsergebnisse der Restliste

- Funktionsdefinition:

```
(define (square x) (* x x))
```

- Funktionsanwendung:

```

      ( square      (+ 3 1 2) )
      (lambda (x)      6
        (* x x) )
              36

```

9.2 Umgebungen

Umgebungen

- Umgebungen sind Gültigkeitsbereiche für Variable
- jeder Funktionsaufruf eröffnet eine neue Umgebung und bindet die aktuellen an die formalen Parameter
- Funktionsaufrufe sind hierarchisch eingebettet → Umgebungen sind rekursiv verschachtelt
- Sichtbarkeit von Variablen: bottom-up Suche nach einer Variablenbindung

Umgebungen

- lokale Umgebungen können zusätzliche Variable einführen, die nicht formale Parameter sind
- $(\text{let } (\langle \text{Name-Wert-Paar}_1 \rangle \dots \langle \text{Name-Wert-Paar}_n \rangle) \langle \text{Funktionsaufruf} \rangle)$
- falls ein Ausdruck in Name-Wert-Paar_i auf einen Namen in einem Name-Wert-Paar_j mit $j < i$ Bezug nimmt, muss statt `let` `let*` verwendet werden.

Umgebungen

Verschattung

```
(define x 3)
(define y 4)

(define (pythagoras x y)
  (let*
    ((square (lambda (z) (* z z)))
     (x2 (square x))
     (y2 (square y)) )
    (sqrt (+ x2 y2)) ) )

(pythagoras (+ x 3) (+ y 2))
```

Umgebungen

Verschattung

```
[define, x, 3]
[define, y, 4]

[define, [pythagoras, x, y],
 [let_star,
  [[square, [lambda, [z], [* , z, z]]]
   [x2, [square, x]],
   [y2, [square, y]] ],
  [sqrt, [+ , x2, y2]] ] ]

[pythagoras, [+ , x, 3], [+ , y, 2]]
```

Umgebungen

```
(define x 3)

(define y 4)

(define (pythagoras x y)
  (let*
    ((square (lambda (z)
                (* z z)) )
     (x2 (square x))
     (y2 (square y)) )
    (sqrt (+ x2 y2)) )

  (pythagoras
    (+ x 3)
    (+ y 2))
```

Globale Umgebung:

Standard-Namen, pythagoras, x, y

Umgebung von pythagoras:

x, y

Umgebung von let*:

square, x2, y2

Verschattung und die damit verbundene Gefahr von Programmierfehlern ist ein gemeinsames Problem aller blockorientierten Programmiersprachen. Die Logikprogrammierung kennt dieses Problem nicht, da sie mit der Beschränkung des Gültigkeitsbereiches auf die Klausel und der Umbenennung der Variablen beim Klauselaufufruf ein extrem simples Skopusmodell verwendet.

Scheme in Prolog (3): Umgebungen

- Umgebungen implementiert als Fakten in der Datenbasis

- Variablenbindungen

```
% env(Environment, Variable, Value).
env(e11,x,6).
env(e11,y,6).
env(e11,square,[lambda,[x],[*], x, x])).
...
```

- rekursive Einbettung von Umgebungen

```
% env(Sub_environment,Super_environment).
env(e12,e11).
env(e11,global).
...
```

- `global` wird als oberste Umgebung vereinbart

Die Umgebungen sind der weitaus komplizierteste Teil unseres Scheme-Interpreters. Sie müssen beim Funktionsaufruf eingerichtet und beim Verlassen einer Funktion wieder aufgegeben werden. Darüberhinaus müssen Prädikate für das Initialisieren, Verändern und Abfragen von Variablenwerten in einer Umgebung bereitgestellt werden.

Scheme in Prolog (3): Umgebungen

- die Prädikate `env/2` und `env/3` müssen dynamisch veränderbar und initial leer sein

```
:- dynamic(env/2).
:- dynamic(env/3).
:- retractall(env(_, _)).
:- retractall(env(_, _, _)).
```

Scheme in Prolog (3): Umgebungen

Abfragen eines Variablenwerts

```
%%% get_value(Var,Env,Val)
% retrieve a variable value from the hierarchy of
% environments
get_value(Var,Env,Val) :-
    (env(Env,Var,Val) ->          % retrieve the value
     true ;                      % from the given environment
     ((Env\=global,env(Env,Senv)) -> % look up the variable
      get_value(Var,Senv,Val) ;    % one environment
      (write('no value for variable '), % higher up
       write(Var),
       write(' in environment '),
       writeln(Env)))).
```

Scheme in Prolog (3): Umgebungen

Eröffnen einer neuen Umgebung als Unterumgebung zu einer existierenden Umgebung

```
%%% extend_env(Variables,Values,Environment,NewEnvironment)
% extend the given environment with a new subenvironment
extend_env(Vars,Vals,Env,NewEnv) :-
    gensym(e,NewEnv),          % generate a name
    assert(env(NewEnv,Env)),    % insert in hierarchy
    bind_variables(Vars,Vals,NewEnv).
```

Hinzufügen von Variablenbindungen zu einer Umgebung

```
%%% bind_variables(Variables,Values,Environment)
% insert variable bindings into a new environment
bind_variables([],[],_).
bind_variables([Var|Vars],[Val|Vals],Env) :-
    assert(env(Env,Var,Val)),
    bind_variables(Vars,Vals,Env).
```

Scheme in Prolog (3): Umgebungen

Definieren eines Variablenwerts (define)

```
%%% define_variable(Variable,Value,Environment)
% define a new variable in a given environment
define_variable(Var,Val,Env) :-
    (env(Env,Var,_) -> retract(env(Env,Var,_)) ; true),
    assert(env(Env,Var,Val)).
```

Ändern eines Variablenwerts (set!)

```
%%% set_variable(Variable,Value,Environment)
% change the value of an already defined variable
set_variable(Var,Val,Env) :-
    env(Env,Var,_) ->
        (retract(env(Env,Var,_)), assert(env(Env,Var,Val))) ;
        (write('unknown variable '),writeln(Var)).
```

Scheme in Prolog (4): Auswertung

eval/3 (2): Auswertung einer Variablen

```
% variables
eval(Exp,Env,Val) :-
    atom(Exp), !,
    get_value(Exp,Env,Val).

?- define_variable(x,5,global).
   true.
?- eval(x,global,X).
   X = 5.
```

9.3 Funktionale Auswertung

Die funktionale Auswertung ist der Normalfall der Berechnung eines Wertes für einen s-Ausdruck in Scheme. Dabei werden stets *alle* Elemente des s-Ausdrucks ausgewertet und anschließend der Wert des ersten Arguments (der ein Lambda-Ausdruck sein muss) auf die Werte der restlichen Argumente angewendet. Grundsätzlich können Funktionen auch eine beliebige Anzahl von Argumenten haben (z.B. +).

Bei der Auswertung von special form expressions wird von dieser Standardauswertung abgewichen, wobei die genaue Behandlung der Argumente in jedem Einzelfall gesondert vereinbart und entsprechend im Interpreter umgesetzt werden muss.

Funktionale Applikation

- Funktionsaufruf: s-Ausdruck

(Funktor, Argument₁, ..., Argument_n)

- funktionale Applikation

1. eval: Auswertung der Listenelemente

- erstes Element
→ Lambda-Ausdruck
- Auswertung der restlichen Elemente
→ aktuelle Parameter

2. apply: Anwendung des Lambda-Ausdrucks auf die Werte

Special form expressions

- *special form expressions*:

Abweichung von der Standard-Auswertungsreihenfolge

- müssen in jedem Einzelfall gesondert implementiert werden
- Grundsatz: der Funktor von *special form expressions* wird nie ausgewertet

Special form expressions

- Fall 1: Argumente werden nicht ausgewertet

(quote <expression>)

- Fall 2: Argumente werden nur teilweise ausgewertet

(set! <variable> <expression>)

(define <name> <expression>)

nur <expression> wird ausgewertet

Special form expressions

- Fall 3: Argumente werden bedingt ausgewertet

```
(if <condition> <then> <else>)
```

<condition> wird immer ausgewertet und in Abhängigkeit vom Resultat entweder der <then>- oder der <else>-Zweig

```
(and <expr1> <expr2> ...)
```

```
(or <expr1> <expr2> ...)
```

Auswertung der Argumente wird abgebrochen, sobald der Funktionswert ermittelt wurde

- Fall 4: Ausdruck wird transformiert

cond wird in geschachtelten if-Ausdruck umgewandelt

Für die *special form expressions* gibt es kein direktes Äquivalent in der Logikprogrammierung. Beispielsweise sind die Konditionale bereits in der Klauselstruktur festgelegt (oder aber werden durch den redundanten Operator `->` kodiert) und die Variablenbindung wird durch die Unifikation übernommen. Die Notwendigkeit für ein `quote` entfällt; zum einen, wegen der syntaktischen Unterscheidung zwischen Variablen und alphanumerischen Konstanten und zum anderen, weil Terme standardmäßig nicht ausgewertet werden, solange dies nicht ausdrücklich durch eine (arithmetische) Auswertungsumgebung gefordert wird. Schließlich existieren in Prolog auch noch spezielle Vergleichsoperatoren (`=@=`, `\=@=`), die die Unifikation und damit den normalen Wertebindungsmechanismus außer Kraft setzen können.

Scheme in Prolog (5): special form expressions

Fall 1: Argumente werden nicht ausgewertet

```
% quoted expressions
eval([quote,Val],_,Val) :- !.
```

```
?- eval([quote, x],global,X).
X = x.
```

Scheme in Prolog (5): special form expressions

Fall 2: Argumente werden teilweise ausgewertet

zweites Listenelement ist ein Atom

```
% normal definitions
eval([define,Var,Valexpr],Env,ok) :- atom(Var), !,
    eval(Valexpr,Env,Val),define_variable(Var,Val,Env).
```

zweites Listenelement ist eine Liste

```
% shorthand for procedure definitions
eval([define,[Var|Parms]|Valexpr],Env,ok) :- !,
    define_variable(Var,[closure,Parms,Valexpr,Env],Env).
```

Scheme in Prolog (5): special form expressions

```
?- eval([define, x, 5],global,X).
    X = ok.
?- eval(x,global,X).
    X = 5.
?- eval([define, y, [+ , 1, 2]],global,X).
    X = ok.
?- eval(y,global,X).
    X = 3.
?- eval([define, [square, x], [* , x, x]],global,X).
    X = ok.
?- eval(square,global,X).
    X = [closure, [x], [* , x, x], global].
```

Scheme in Prolog (5): special form expressions

Fall 2: Argumente werden teilweise ausgewertet (Forts.)

```
% assignments
eval([setbang,Var,Valexpr],Env,ok) :- !,
    eval(Valexpr,Env,Val),set_variable(Var,Val,Env).

?- eval([define, x, 5],global,X).
   X = ok.
?- eval([setbang, x, 6],global,X).
   X = ok.
?- eval(x,global,X).
   X = 6.
```

Scheme in Prolog (5): special form expressions

Fall 3: Argumente werden bedingt ausgewertet

```
% conditionals without alternative
eval([if,Pred,Cons],Env,Val) :- !,
    eval(Pred,Env,Boolean),
    (true(Boolean) ->          % evaluiert Pred zu TRUE?
     eval(Cons,Env,Val)).

true('#t').

% conditionals with alternative
eval([if,Pred,Cons,Alter],Env,Val) :- !,
    eval(Pred,Env,Boolean),
    (true(Boolean) ->
     eval(Cons,Env,Val) ;
     eval(Alter,Env,Val)).
```

Scheme in Prolog (5): special form expressions

```
?- eval([define, x, 5],global,X).
   X = ok.
?- eval([if, [eq, x, 5], [quote, yes], [quote, no]],global,X).
   X = yes.
```

Scheme in Prolog (6): primitive Funktionen

Auswertung primitiver Funktionen

```
% function application for primitives
eval([Optor|Opnds],Env,Val) :-
    eval(Optor,Env,Optorval),
    primitive_proc(Optorval),!,
    list_of_values(Opnds,Env,Opndsval),
    apply_primitive(Optorval,Opndsval,Val),
    (flag(verbose,on,on) ->
        (write('into value '),writeln(Val)) ; true).
```

- Der Operator muss ausgewertet werden,
 - um auf eine Funktionsdefinition für ein Atom zuzugreifen, oder
 - um eine Funktionsdefinition dynamisch berechnen zu können.

Neben den special form expressions existieren in jedem Scheme-System auch eine Vielzahl vordefinierter (primitiver) Funktionen, etwa zur Arithmetik und zur Listenbehandlung. Für diese existiert keine Definition als Lambda-Ausdruck. Vielmehr muss der Interpreter vordefinierte Funktionen als solche erkennen und eine Semantik in Form eines ausführbaren Prolog-Prädikats bereitstellen.

Scheme in Prolog (6): primitive Funktionen

Auswertung primitiver Funktionen

```
%%% apply_primitive(Procedure,Arguments,Value)
% connects a primitive to an underlying Prolog-predicate
apply_primitive(+,Args,Value) :- sumlist(Args,Value).
apply_primitive(*,Args,Value) :- multlist(Args,Value).
apply_primitive(eq,[X,Y],Value) :-
    X==Y -> Value='#t' ; Value='#f'.
apply_primitive(car,[[Value|_]],Value).
apply_primitive(cdr,[_|Value]],Value).
apply_primitive(cons,[X,Y],[X|Y]).
apply_primitive(pair,[X],Value) :-
    X=[_|_] -> Value='#t' ; Value='#f'.
apply_primitive(null,[X],Value) :-
    X=[] -> Value='#t' ; Value='#f'.
apply_primitive(atom,[X],Value) :-
    atomic(X) -> Value='#t' ; Value='#f'.
```

Closures

- Funktionen sind "Bürger erster Klasse"
- Sie können als Wert einer Variablen zugewiesen werden ...

```
(define pythagoras (lambda (x y)
  (sqrt (+ (expt x 2) (expt y 2)))))
```

```
[define, pythagoras, [lambda, [x, y],
  [sqrt, [+ , [expt, x, 2], [expt, y, 2]]]]]
```

- ... als Wert an einer Argumentstelle übergeben werden ...

```
(apply pythagoras (3 4))
```

- ... oder als Resultat einer Berechnung auftreten

```
((list (quote lambda) (quote (x y))
  (quote (sqrt ...)))
  3 4)
```

Die Möglichkeit, Funktionen an andere Funktionen zu übergeben bzw. als Wert eines Funktionsaufrufs zu ermitteln, ist nicht nur wesentlich für die in der funktionalen Programmierung sehr stark verbreitete Nutzung von Funktionen höherer Ordnung. Sie stellt auch eine wichtige Grundlage für die Realisierung objektorientierter Konzepte dar, da man mit ihnen sehr gut leistungsfähige Mechanismen zum Nachrichtenaustausch zwischen Objekten implementieren kann. Historisch gesehen wurden daher auch viele der frühen objektorientierten Sprachen zuerst als Erweiterung des funktionalen Paradigmas erprobt und eingesetzt.

Wegen der universellen Verwendbarkeit von Funktionen muss in Scheme strikt zwischen dem Definitionszeitpunkt und dem Ausführungszeitpunkt einer Funktion unterschieden werden. Dies ist immer dann wesentlich, wenn eine Funktion freie Variable benutzt, d.h. solche Variable, die nicht als Argumente beim Aufruf der Funktion gebunden werden, sondern solche, die bereits durch die Umgebung, in der die Funktion definiert wurde, bereitgestellt werden. Dann muss die Umgebung, in der die Definition erfolgte, auch zum Aufrufzeitpunkt zur Verfügung stehen, weil sonst Differenzen in der Definitions- und Ausführungsumgebung zu einem unterschiedlichen Programmverhalten führen können. Um solche Unterschiede zu vermeiden, wird die Funktionsdefinition zusammen mit ihrer Definitionsumgebung in einer *Closure* zusammengefasst, damit sie auch beim Aufruf noch in der ursprünglichen Definitionsumgebung ausgewertet werden kann.

Frühe Versionen von Lisp-Systemen stellten keine Closures bereit, so dass der Programmierer durch Hygiene-Regeln ("Freie Variable verwendet man nicht!") dafür sorgen musste, dass entsprechende Probleme nicht auftraten.

Closures

- Funktionsdefinitionen können freie und gebundene Variable enthalten
- Problemfall: freie Variable

Wertebindung zum Zeitpunkt der Funktionsdefinition muss nicht der Wertebelegung zum Zeitpunkt der Funktionsauswertung sein

- Lösung: Closure
 - jede Funktionsdefinition geschieht in einer Umgebung
 - das Paar aus Funktionsdefinition und seiner Definitionsumgebung heisst Closure

Scheme in Prolog (7): Closures

Erzeugen einer Closure

```
% lambda expressions
eval([lambda,Parms|Body],Env,[closure,Parms,Body,Env]) :- !.

% shorthand for procedure definitions (repeated here)
eval([define,[Var|Parms]|Valexpr],Env,ok) :- !,
    define_variable(Var,[closure,Parms,Valexpr,Env],Env).
```

Scheme in Prolog (8): Nutzerdefinierte Funktionen

Auswerten einer Funktion bzw. einer Closure

```
% procedures
eval([Optor|Opnds],Env,Val) :-
    eval(Optor,Env,[closure,Pparms,Body,Penv]),
    % retrieve the definition
    list_of_values(Opnds,Env,Opndsval),
    % evaluate the arguments
    extend_env(Pparms,Opndsval,Penv,Newenv),
    % create a new sub-environment
    eval_sequence(Body,Newenv,Val),
    % evaluate the procedure in the new environment
    retractall(env(Newenv,_,_)),
    retractall(env(Newenv,_)).
    % housekeeping: remove the used environment
```

Scheme in Prolog (8): Nutzerdefinierte Funktionen

Auswerten einer Funktion (Hilfsprädikate)

```
%%% list_of_values(expressions,environment,list_of_values)
% evaluate a list of expressions into a list of values
list_of_values([],_,[]).
list_of_values([Exp|Expns],Env,[Val|Vals]) :-
    eval(Exp,Env,Val),list_of_values(Expns,Env,Vals).

%%% eval_sequence(Expressions,Environment,Value)
% evaluate a sequence of expressions into a single value
eval_sequence([Exp],Env,Val) :- !, eval(Exp,Env,Val).
eval_sequence([Exp|Expns],Env,Val) :-
    eval(Exp,Env,_),eval_sequence(Expns,Env,Val).
```

9.4 Rekursive Funktionen

Im Folgenden betrachten wir eine Reihe von Beispielen aus den Bereich der Listenprogrammierung. Zum besserern Vergleich der beiden Paradigmen sind dabei immer eine funktionale und eine logikbasierte Definition mit vergleichbarer Funktionalität gegenübergestellt.

Rekursive Funktionen

Typstest Liste

```
(define (proper-list? liste)
  (if (equal? liste '()) #t
      (if (pair? liste) (proper-list? (cdr liste)) #f) ) )

proper_list([]).
proper_list([_|Rest]) :- proper_list(Rest).
```

Rekursive Funktionen

Element einer Liste

```
(define (member elem liste)
  (if (null? liste) #f
      (if (equal? elem (car liste)) #t
          (member elem (cdr liste))) ) )

(define (member elem liste)
  (if (null? liste) #f
      (if (equal? elem (car liste)) (cdr liste)
          (member elem (cdr liste))) ) )

member(Elem,[Elem|_]).
member(Elem,[_|Rest]) :- member(Elem,Rest).
```

Rekursive Funktionen

Letztes Element einer Liste

```
(define (last liste)
  (if (null? (cdr liste)) (car liste) (last (cdr liste)) ) )

last(Elem,[Elem]).
last(Elem,[_|Rest]) :- last(Elem,Rest).
```

Länge einer Liste

```
(define (length liste)
  (if (null? liste) 0 (+ 1 (length (cdr liste))) ) )

length([],0).
length([_|Rest],Laenge) :-
  length(Rest,RLaenge),
  Laenge is RLaenge + 1.
```

Rekursive Funktionen

Verketten zweier Listen

```
(define (append liste1 liste2)
  (if (null? liste1)
      liste2
      (cons (car liste1)
            (append (cdr liste1)
                    liste2)) ) )

append([],Liste,Liste).
append([Elem|Rest],Liste,[Elem|NeueListe]) :-
  append(Rest,Liste,NeueListe).
```

Rekursive Funktionen

Umdrehen einer Liste

```
(define (reverse liste)
  (define (rev liste acc)
    (if (null? liste)
        acc
        (rev (cdr liste)
              (cons (car liste) acc))) )
  (rev liste '()) )
```

```
reverse(Liste,RListe) :- rev(Liste,[],RListe).
rev([],RListe,RListe).
rev([Elem|Rest],Acc,RListe) :- rev(Rest,[Elem|Acc],RListe).
```

9.5 Funktionen höherer Ordnung

Funktionen höherer Ordnung

- Funktionen, die Funktionen als Argumente fordern:
map, filter, reduce, ...
- Funktionen, die Funktionen als Argumente nehmen *und* Funktionen als Funktionswert zurückgeben:
curry, rcurry, compose, conjoin, ...

Im Gegensatz zur Logikprogrammierung werden im funktionalen Paradigma Prozeduren höherer Ordnung in weitaus stärkerem Maße als Ansatz zur systematischen Komposition komplexer Programme genutzt. Dies hängt sicherlich damit zusammen, dass die garantierte Eindeutigkeit des Funktionswerts in der funktionalen Programmierung für ein solches Vorgehen sehr förderlich ist, während analoge Konstrukte in der Logikprogrammierung vor allem zum Aufsammeln oder Durchmustern alternativer Berechnungsergebnisse dienen (vgl. `findall/3`, `bagof/3`, `setof/3`, `forall/2` usw.). So ist z.B. die Funktion höherer Ordnung `map` ein Iterator über Listen, der alle Elemente aufgrund einer gegebenen funktionalen Berechnungsvorschrift transformiert und die Transformationsergebnisse in einer Ergebnisliste sammelt. In der Logikprogrammierung wäre es dann aber dem Programmierer überlassen, dafür Sorge zu tragen, dass das Transformationsprädikat genau zweistellig *und* funktional ist.

Funktionen höherer Ordnung

Transformieren aller Elemente einer Liste: `map`

```
(define (map* function liste)
  (if (null? liste)
      '()
      (cons (function (car liste) )
            (map* function (cdr liste))) ) )

(map* (lambda (x) (* 2 x) ) '(1 2 3) ) ==> (2 4 6)

(define (zero? x) (if (eq? x 0) #t #f) )
(map* zero? '(1 0 -1) ) ==> (#f #t #f)
```

Funktionen höherer Ordnung

Selektieren von Elementen einer Liste: `filter`

```
(define (filter* test liste)
  (if (null? liste)
      '()
      (if (test (car liste))
          (cons (car liste) (filter* test (cdr liste) ) )
          (filter* test (cdr liste) ) ) ) )

(filter* zero? '(1 0 -1)) ==> (0)
```

Funktionen höherer Ordnung

Erweitertes `map` mit beliebig vielen Argumenten

```
(define (map** function . liste-von-listen)
  (if (null? (filter* null? liste-von-listen))
      (cons (apply function (map* car liste-von-listen))
            (apply map**
                    function
                    (map* cdr liste-von-listen) ) )
      '( ) ) )

(map + '(1 2 3) '(2 3 4) '(3 4 5)) ==> (6 9 12)
```

Funktionen höherer Ordnung

Falten: reduce

Listenelemente paarweise mit einem Operator verknüpfen und auf einen Resultatswert reduzieren

```
(define (reduce function liste anfangswert)
  (if (null? liste)
      anfangswert
      (reduce function
                (cdr liste)
                (function anfangswert (car liste))) ) )
```

```
(reduce + (1 2 3 4 5) 0) ==> 15
```

```
(reduce + (map (lambda (x) 1) '(a b c)) 0) ==> 3
```

Funktionen höherer Ordnung

Funktionskomposition: linkes Argument eines partiellen Funktionsaufrufs: curry

```
(define (curry function argument)
  (lambda (neues-argument)
    (function argument neues-argument) ) )
```

Funktionskomposition: rechtes Argument eines partiellen Funktionsaufrufs: rcurry

```
(define (rcurry function argument)
  (lambda (neues-argument)
    (function neues-argument argument) ) )
```

```
((curry < 0) 1) ==> #t
```

```
((rcurry < 0) 1) ==> #f
```

Funktionen höherer Ordnung

- typische Verwendungsweisen von curry/rcurry

(curry + 1)	$f(x) = x + 1$	Inkrement
(curry / 1)	$f(x) = \frac{1}{x}$	Reziprokwert
(rcurry / 2)	$f(x) = \frac{x}{2}$	Halbieren
(curry * 2)	$f(x) = 2x$	Verdoppeln
(rcurry expt 2)	$f(x) = x^2$	Quadrieren
(curry - 0)	$f(x) = -x$	Vorzeichenumkehr
(curry = 0)	$f(x) = \begin{cases} \#t & \text{für } x = 0 \\ \#f & \text{sonst} \end{cases}$	Test gleich Null

- typischer Verwendungskontext

```
(map (rcurry > 0) '(1 0 -1)) ==> (#t #f #f)
```

- auch Erweiterung auf beliebig viele Argumente

Funktionen höherer Ordnung

Sukzessive Funktionsanwendung: `compose`

```
(define (compose func1 func2)
  (lambda (x) (func1 (func2 x))))
```

```
((compose car cdr) '(a b c)) ==> b
```

- weitere Funktionen

- `conjoin`: wahr wenn alle Prädikate wahr
- `disjoin`: wahr wenn mindestens ein Prädikat wahr
- `always`: erzeugt Funktion mit konstantem Wert

```
(reduce + (map (always 1) '(a b c) ) 0) ==> 3
```

Funktionen höherer Ordnung

- Beispielanwendungen für Funktionen höherer Ordnung

```
(define (mean werte)
  (/ (reduce + werte 0)
     (reduce + (map (lambda (x) 1) werte)) ))
```

```
(define (variance werte)
  (let (m (mean werte))
    (mean (map square (map (rcurry - m) werte)) ) ) )
```

```
(define (variance werte)
  (let (m (mean werte))
    (mean ((compose (curry map square)
                     (curry map (rcurry - m)))
           werte)) ) )
```

Die Verwendung von Funktionen höherer Ordnung bietet einen sehr systematischen Ansatz zur Konstruktion komplexer Softwarelösungen in Analogie zu einem Baukastensystem. Für die Kombinationsmöglichkeiten der verschiedenen Elementarbausteine existieren sogar eine Reihe von "Rechenregeln", mit deren Hilfe semantik-erhaltende Äquivalenztransformationen vorgenommen werden können, z.B.

```
(filter p (filter q xs)) ≡ (filter q (filter p xs))
```

```
(curry map (compose f g)) ≡ (compose (curry map f) (curry map g))
```


9.6 Ein abschließender Vergleich

Mit dem logik-basierten und dem funktionalen Verarbeitungsmodell haben Sie zwei Programmierparadigmen kennengelernt, die eine Reihe von weitreichenden Gemeinsamkeiten aufweisen, aber in einigen Bereichen auch deutlich unterschiedliche Akzente setzen. Beide Modelle sind prinzipiell sehr offen für Erweiterungen, mit denen dann die spezifischen Möglichkeiten um zusätzliche Aspekte alternativer Verarbeitungsmodelle ergänzt werden können: Die funktionale Programmierung kann durch relationale Elemente (z.B. Suche nach Alternativwerten) ergänzt werden, so dass dann auch ein Interpreter für eine Hornklausellogik oder gar ein ausgebauter Theorembeweiser für noch leistungsfähigere Logikkalküle bereitgestellt werden kann. Umgekehrt kann die Auswertungsumgebung der Logikprogrammierung auf generelle funktionale Ausdrücke erweitert werden. Eine sinnvolle Erweiterung in dieser Richtung sind dann Constraint-Systeme, die die Logikprogrammierung von der strikten Anwendungsreihenfolgen von Klauseln und Teilzielen befreien und auch für numerische Probleme noch unterspezifizierte Resultate ermitteln können. Ein Vergleich der spezifischen Unterschiede zwischen den beiden Paradigmen ist daher nur dann sinnvoll, wenn man sich auf eine "Kernausrüstung" bezieht, so wie sie etwa hier in der Veranstaltung behandelt wurde.

Logik-basiert und funktional: Ein Vergleich

logik-basiert und funktional

- leistungsfähige Abstraktionskonzepte
- flexible Sprachen zum funktionalen Prototyping
- schwache Typisierung, dynamische Datenstrukturen
- Interpretative Abarbeitung
- Erweiterbarkeit
- Metaprogrammierung

Logik-basiert und funktional: Ein Vergleich

logik-basiert

- leistungsfähige build-in-Konzepte (relationale DB, Unifikation, Suche)
- Richtungsunabhängigkeit
- extrem einfacher Variablenskopus
- intuitiv plausible Listenverarbeitung
- flexible Syntax

funktional

- systematische Softwarekomposition mit Funktionen höherer Ordnung
- gute Passfähigkeit zur objektorientierten Programmierung
- vollständige Integration der Arithmetik in das Verarbeitungsmodell

Logik-basiert und funktional: Ein Vergleich

- verschiedene Versuche zur Synthese
- z.B. Mercury (Universität Melbourne)
- Deklaration von Verarbeitungsmodi
- Compilation in ausführbaren Code für die verschiedenen Aufrufvarianten
- Reihenfolge der Abarbeitung von Teilzielen prinzipiell beliebig
- sequentielle Abhängigkeiten zwischen Teilzielen werden separat verwaltet (analog zur Differenzlistentechnik)

10 Aktive Datenstrukturen

Aktive Datenstrukturen

append($X \setminus Y, Y \setminus Z, X \setminus Z$).

Aktive Datenstrukturen

dynamische Datenstrukturen

können sich bei Bedarf an unterschiedliche quantitative Anforderungen anpassen

- Listen, im Gegensatz zu Arrays

→ SE I

aktive Datenstrukturen

enthalten noch variable (instanziierbare) Elemente

- unbekannte Elemente, im Gegensatz zur Änderung von Elementen
- → unvollständige Datenstrukturen
- → offene Datenstrukturen

Wir erinnern uns: Einzelne Fakten, die Variable enthalten, haben bereits eine prozedurale Semantik. Z.B. instanziiert das Prädikat $p(X, X)$, wenn es partiell unterspezifiziert aufgerufen wird, die nicht instanziierte Argumentposition mit dem Wert des anderen, instanziierten Arguments. Daten sind in der Logikprogrammierung nicht mehr rein passiv, sondern haben, solange sie noch uninstantiierte Variable enthalten, einen aktiven Charakter. Dies kann man sich zu Nutze machen, um Daten durch Unifikation mit zusätzlicher Information anzureichern, z.B. einen derzeit noch unbekannten Listenrest näher zu spezifizieren. Auf diese Weise kann man u.U. die Datenstruktur modifizieren, ohne kostspielige Kopieroperationen durchführen zu müssen. Wir betrachten dies für den Fall der Warteschlange, die ja ein Einfügen (bzw. Entfernen) der Elemente am Listenende erfordert.

10.1 Offene Listen

Abgeschlossene Listen

- Listenzugriff erfolgt immer vom Kopf aus
- → Modifikationen am Listenende erfordern vollständiges Kopieren der Liste
- Beispiel Stack: Zugriff nur vom Kopf aus

```
% stack(?Element,?AlterStack,?NeuerStack)
```

```
stack(E,Stack,[E|Stack]).
```

Abgeschlossene Listen

- drei Stackoperationen als unterschiedliche Instanziierungsvarianten

– push:

```
?- stack(a,[ ],S).  
S = [a].
```

```
?- stack(b,[a],S).  
S = [b,a].
```

Abgeschlossene Listen

- Stackoperationen (2)

– pop:

```
?- stack(E,S,[b,a]).  
E = b, S = [a].
```

```
?- stack(E,S,[a]).  
E = a, S = [ ].
```

– non-empty:

```
?- stack(_,[a]).  
true.
```

```
?- stack(_,[ ]).  
false.
```

Abgeschlossene Listen

- Beispiel Warteschlange: Zugriff auch am Listenende
 - **enqueue**: Einfügen am Listenende
 - **dequeue**: Wegstreichen am Listenanfang
- Einfügen und Wegstreichen sind keine Umkehroperationen mehr
 - nicht mehr in einem Prädikat realisierbar

Abgeschlossene Listen

- **enqueue**

```
% enqueue(?Element,?AlteQueue,?NeueQueue)
enqueue(E,[ ],[E]).
enqueue(E,[H|AR],[H|NR]) :- enqueue(E,AR,NR).
```

```
?- enqueue(a,[ ],Q).
   Q = [a].
```

```
?- enqueue(b,[a],Q).
   Q = [a,b].
```

Abgeschlossene Listen

- **dequeue**

```
% dequeue(?Element,?AlteQueue,?NeueQueue)
dequeue(E,[E|R],R).
```

```
?- dequeue(E,[a,b],Q).
   E = a, Q = [b].
```

```
?- dequeue(E,[a],Q).
   E = a, Q = [ ].
```

- Problem: Kopieren beim Einfügen
 - Uninstanciierter Aufruf eines Listenarguments ist automatisch Listenkonstruktor
 - Konsumtion von Freispeicherzellen

Offene Listen

- alternative Implementation als offene Liste
 - Listenrest ist eine uninanzierte Variable
 - erlaubt das Hinzuunifizieren einer neuen Restliste
- Listenendedetektion durch Test auf leere Restliste:
 - leere Liste benötigt Sonderbehandlung
 - Listenelemente dürfen keine uninanzierten Variablen mehr sein
- Repräsentation der Warteschlange

[]	[_ _]
[a]	[a _]
[a,b]	[a,b _]
[a,b,c]	[a,b,c _]
...	...

Offene Listen

- enqueue:

```
% enqu_ol(+Element,+Queue)
enqu_ol(E,[Y|X]) :-          % Einfuegen in die
    var(Y),var(X),Y=E.       % leere Liste
enqu_ol(E,[Y|X]) :-          % Einfuegen einer
    nonvar(Y),var(X),X=[E|_] % neuen Restliste
enqu_ol(E,[_|T]) :-          % Suchen der
    nonvar(T),enqu_ol(E,T).  % offenen Restl.

?- X=[_|_],enqu_ol(a,X),
    enqu_ol(b,X),enqu_ol(c,X).
X = [a, b, c|X1].
```

Offene Listen

- dequeue:

```
% dequ_ol(-Element,+AlteQueue,-NeueQueue)
dequ_ol(_,[Y|X],_) :-        % Entfernen aus
    var(Y),var(X),fail.       % der leeren Liste
dequ_ol(E,[E|X],[_|X]) :-    % Entfernen aus
    nonvar(E),var(X).         % der Einerliste
dequ_ol(E,[E|T],T) :-        % Entfernen sonst
    nonvar(T).
```

Offene Listen

- dequeue:

```
?- dequ_ol(E, [b, c | _], N).
    E = b, N = [c | X1].
```

```
?- dequ_ol(E, [c | _], N).
    E = c, N = [X1 | X2].
```

```
?- dequ_ol(E, [_ | _], N).
    false.
```

- Nachteil: beim Einfügen nach wie vor Suche nach Listenende erforderlich
→ Differenzlisten

10.2 Differenzlisten

Differenzlisten

Listendifferenz

jede Liste kann als Differenz zweier Listen dargestellt werden.

```
?- op(650, xfx, \).
```

- z.B. Liste $[a, b, c]$ ist äquivalent zu:

```
[a, b, c] \ [ ]
[a, b, c, d] \ [d]
[a, b, c, d, e] \ [d, e]
[a, b, c, d, e, f] \ [d, e, f]
```

- generalisiert zu

```
[a, b, c | X] \ X
```

Differenzlisten sind Paare aus zwei verschiedenen Informationen: Einem Verweis auf einen Listenanfang (und damit indirekt auch auf die möglicherweise folgenden Listenelemente), sowie ein Verweis auf die Elemente der Liste, die auf das letzte signifikante Element noch folgen könnten und somit auf das Ende der Differenzliste selbst. Diese beiden Verweise sind — sofern sie bereits instanziiert sind — tatsächlich Zeiger auf gepunktete Paare im Listenspeicher. Aber auch hier gilt: Wir arbeiten mit Zeigerstrukturen, können die Zeiger aber weder direkt beobachten, noch manipulieren!

Ist der zweite Verweis noch uninstantiiert und koreferiert er mit der noch variablen Restliste in der Differenzliste selbst ($[a, b | X] \ X$), so kann die Liste "nach rechts" erweitert werden. Dazu wird die Restliste und der koreferente zweite Verweis (das bisherige Listenende) mit

einem neuen gepunkteten Paar aus einem neuen Listenelement und einer neuen uninstantiierten Restliste instanziiert: $X = [c|Y]$. An der ursprünglichen Differenzliste ändert sich dadurch (auch physisch im Speicher) überhaupt nichts, denn der nunmehr instanziierte zweite Verweis zeigt ja immer noch auf das ursprüngliche Listenende, d.h. die "Differenz" bleibt unverändert: $[a, b, c|Y] \setminus [c|Y]$. Ist man jedoch an der erweiterten Differenzliste interessiert, so steht diese jetzt ebenfalls zur Verfügung, nur wählt man hierfür einfach eine andere Kombination von Verweisen: $[a, b, c|Z] \setminus Z$. Das bedeutet, die Erweiterung der Differenzliste "nach rechts" verbraucht weder unnötig Freispeicherplatz, noch ist sie destruktiv (Es werden an keiner Stelle Pointer "umgebogen")! Für eine solche Lösung gibt es keinerlei Äquivalent in einem der anderen Programmierparadigmen.

Differenzlisten

- **enqueue:**

```
% enqueue_dl(+Element,+AlteQueue,?NeueQueue)
enqueue_dl(E,[ ]\ [ ],[E|X]\X).
    % Einfuegen in leere Differenzliste
enqueue_dl(E,Left\[E|Right],Left\[Right]).
    % Einfuegen sonst

% Testdaten:
[ ]\ [ ]           % leere D-Liste
[a|X]\X           % einelementige D-Liste
[a,b|X]\X         % zweielementige D-Liste
```

Differenzlisten

- **enqueue:**

```
?- Q=[ ]\ [ ], enqueue_dl(a,Q,N).
   Q = [ ]\ [ ], N = [a|X1]\X1.

?- Q=[a|X]\X, enqueue_dl(b,Q,N).
   Q = [a, b|X2]\[b|X2], N = [a, b|X2]\X2.

?- Q=[a, b|X]\X, enqueue_dl(c,Q,N).
   Q = [a, b, c|X3]\[c|X3],
   N = [a, b, c|X3]\X3.
```

Differenzlisten

- **dequeue:**

```
% dequeue(?Element,+AlteQueue,?NeueQueue).
dequeue_dl(E,[E|Left]\Right,Left\Right).
```

```
?- Q=[]\[], dequeue_dl(E,Q,N).
    false.
```

```
?- Q=[a|X]\X, dequeue_dl(E,Q,N).
    Q = [a|X1]\X1, E = a, N = X1\X1.
```

```
?- Q=[a, b|X]\X, dequeue_dl(E,Q,N).
    Q = [a, b|X1]\X1, E = a, N = [b|X1]\X1.
```

→ Rechtserweiterung ohne Kopieren und ohne Suche nach dem rechten Listende

Differenzlisten

- Problemfall:

```
?- Q=X\X, dequeue_dl(E,Q,N).
    Q = [E|X1]\[E|X1], X = [E|X1], N = X1\[E|X1].
```

- Liste mit negativer Länge!

Differenzlisten

→ Entfernen muss blockiert werden, falls Warteschlange leer

```
dequeue2(E,Left\Right,LeftNew\Right) :-
    Left \= [] = Right,
    Left = [E|LeftNew].
```

```
?- Q=X\X, dequeue2(E,Q,N).
    false.
```

```
?- Q=[a|X]\X, dequeue2(E,Q,N).
    Q = [a|X]\X, E = a, N = X\X.
```

```
?- Q=[a,b|X]\X, dequeue2(E,Q,N).
    Q = [a, b|X]\X, E = a, N = [b|X]\X.
```

```
?- Q=[a,b|X]\[a,b|X], dequeue2(E,Q,N).
    false.
```

Differenzlisten

- Verwendung zur Listenverkettung ohne Dekomposition und Kopieren: `append_d1/3`

```
% append_d1(?DListe1,?DListe2,?DListe3)
append_d1(X\Y,Y\Z,X\Z).
```

$$\begin{array}{lcl}
 X: & \underbrace{a \ b \ c}_{X \setminus Y} & d \ e \ f \ g \ h \ i \ j \ k \\
 Y: & & \underbrace{d \ e \ f}_{Y \setminus Z} \ g \ h \ i \ j \ k \\
 Z: & \underbrace{a \ b \ c \ d \ e \ f}_{X \setminus Z} & g \ h \ i \ j \ k
 \end{array}$$

- `append_d1/3` ist nur noch ein Fakt!
- drei Unifikationsforderungen über den beteiligten Differenzlisten
- Wo ist die prozedurale Semantik geblieben?
 - konstanter Zeitbedarf?
 - kein Kopieren → ökonomische Freispeicherverwendung

Hier ist nun tatsächlich der Punkt erreicht, wo sich die klassische prozedurale Semantik eines (nichttrivialen) Algorithmus vollständig in die deklarative Semantik von drei Unifikationsforderungen aufgelöst hat. Da auch hier kein Kopieren mehr erforderlich ist, entsteht keinerlei "Datenmüll". Es sieht zudem so aus, als ob ein Algorithmus, dessen Berechnungsaufwand bisher ja doch linear von der Länge der ersten Liste abhängig war, nunmehr in konstanter Zeit abgearbeitet werden kann. Wo wird hier noch "gerechnet"? Ist das noch Informatik, oder schon Magie?

Differenzlisten

- Testaufrufe:

```
?- L1=[a|X]\X, L2=[a, b|Y]\Y, append_d1(L1,LX,L2).
   L1 = [a, b|X1]\[b|X1], L2 = [a, b|X1]\X1,
   LX = [b|X1]\X1.
```

Differenzlisten

- Testaufrufe:

```
?- L1=[a|X]\X, L2=[a, b|Y]\Y, append_dl(L1,L2,LX).
    L1 = [a, a, b|X1]\[a, b|X1],
    L2 = [a, b|X1]\X1,
    LX = [a, a, b|X1]\X1.

?- L2=[a, b|Y]\Y, append_dl(LX,L2,L2).
    L2 = [a, b|X2]\X2, LX = [a, b|X2]\X3,
    LY = X3\X2.
```

Das letzte Aufrufbeispiel ist wegen seines hochgradig unterspezifizierten Resultats interessant. Erinnern Sie sich: die Instanziierungsvariante `append(-,-,+)` zerlegt eine Liste in alle Kombinationen von Teillisten. Im Falle von `append/3` wurden sie auf Anfrage sukzessiv aufgezählt. Nunmehr sind all diese Varianten (in unserem Falle wegen des einfachen Beispiels insgesamt nur vier) in dem Ergebnis implizit enthalten! Was müssen Sie tun, um sie "sichtbar" zu machen?

Differenzlisten

- Beispiel: Beseitigen rekursiver Listeneinbettungen: `flatten/2`

```
% flatten mit einfachen Listen (modifizierte Version)
flatten([],[]).
flatten(X,[X]) :- atom(X), X\=[ ].
flatten([K|R],F) :-
    flatten(K,KF), flatten(R,RF),
    append(KF,RF,F).

% flatten(+Liste,?DListe) mit Differenzlisten
flatten(Xs,Ys) :- flatten_dl(Xs,Ys\[ ]).
flatten_dl([ ],Xs\Xs).
flatten_dl(X,[X|Xs]\Xs) :- atom(X), X\=[ ].
flatten_dl([X|Xs],Ys\Zs) :-
    flatten_dl(X,As\Bs), flatten_dl(Xs,Cs\Ds),
    append_dl(As\Bs,Cs\Ds,Ys\Zs).
```

Entfalten

Entfalten (Unfolding)

Vereinfachen einer Differenzlisten-Klausel durch Integrieren der "statischen" `append_dl/3`-Definition

```
flatten_dl([X|Xs],Ys\Zs) :-
    flatten_dl(X,As\Bs),
    flatten_dl(Xs,Cs\Ds),
    append_dl(As\Bs,Cs\Ds,Ys\Zs).
%      | |   | |   | |
% append_dl( X\Y,  Y\Z , X\Z)
%
flatten_dl([X|Xs],Ys\Zs) :-
    flatten_dl(X,Ys\Y1s),
    flatten_dl(Xs,Y1s\Zs).
```

Die Differenzlisten-Implementierung von `append_dl/3` ist nicht nur verblüffend einfach, Sie bietet auch die Grundlage zur Vereinfachung anderer Differenzlisten-Prädikate. Auf diese Weise erhalten wir selbst für das "naive" `reverse/2` eine elegante und effiziente Implementierung, die aber bei genauerem Hinsehen erstaunlich viele Gemeinsamkeiten mit unserem Akkumulator-basierten Prädikat hat ...

Entfalten

- Beispiel: Umdrehen einer Liste: `reverse/2`

```
% naives reverse
n_reverse([],[]).
n_reverse([X|Xs],R) :-
    n_reverse(Xs,RXs),
    append(RXs,[X],R).

% reverse(+Liste1,?Liste2)
reverse(Xs,Ys) :- reverse_dl(Xs,Ys\[_]).
reverse_dl([_],Xs\Xs).
reverse_dl([X|Xs],Ys\Zs) :-
    reverse_dl(Xs,As\Bs),
    append_dl(As\Bs,[X|Cs]\Cs,Ys\Zs).
```

Entfalten

- Vereinfachen durch Entfalten

```
reverse_dl([ ],Xs\Xs).
reverse_dl([X|Xs],Ys\Zs) :-
    reverse_dl(Xs,As\Bs),
    append_dl(As\Bs,[X|Cs]\Cs,Ys\Zs).
%           | |       |   |   | |
%  append_dl( X\Y,      Y \ Z , X\Z)
%
reverse_dl([X|Xs],Ys\Zs) :-
    reverse_dl(Xs,Ys\[X|Zs]).
```

Entfalten

- Vergleich mit endrekursivem reverse/2

reverse(Xs,Ys) :- reverse_dl(Xs,Ys\[]).	reverse(Xs,Ys) :- reverse(Xs,[],Ys).
reverse_dl([],Xs\Xs).	reverse([],Xs,Xs).
reverse_dl([X Xs],Ys\Zs) :- reverse_dl(Xs,Ys\[X Zs]).	reverse([X Xs],Ys,Zs) :- reverse(Xs,[X Ys],Zs).

→ Analogie zur Akkumulatorverwendung!

Entfalten

- Sortieren einer Liste: quicksort/2

```
% quicksort(+Liste1,?Liste2)
quicksort([X|Xs],Ys) :-
    split(Xs,X,Vorn,Hinten),
    quicksort(Vorn,Vs),
    quicksort(Hinten,Hs),
    append(Vs,[X|Hs],Ys).
quicksort([ ],[ ]).
```

Entfalten

- Implementation mit Differenzlisten:

```
quicksort(Xs,Ys) :- quicksort_dl(Xs,Ys\[ ]).
quicksort_dl([X|Xs],Ys\Zs) :-
    split(Xs,X,Vorn,Hinten),
    quicksort_dl(Vorn,Ys\[X|Y1s]),
    quicksort_dl(Hinten,Y1s\Zs).
quicksort_dl([ ],Xs\Xs).
```

Differenzlisten

- Hauptanwendungsgebiet der Differenzlistentechnik
 - Definite Clause Grammar
- Anwendung der Idee der Differenzlisten
 - Mercury: lineare Ordnung der Teilziele einer Klausel
 - Suchraumverwaltung in Theorembeweisern

→ GWV

10.3 Definite Clause Grammar

Die Klauseln eines Logikprogramms haben sehr große Strukturähnlichkeit zu den Regeln einer kontextfreien Grammatik: einem Symbol auf der linken Regelseite steht eine Folge von Symbolen auf der rechten Regelseite gegenüber. Dass diese Ähnlichkeit nicht nur oberflächlich ist, wird deutlich, wenn wir die Semantik einer kontextfreien Ersetzungsregel als Prolog-Programm spezifizieren. Mit der Differenzlistentechnik haben wir dann auch eine effiziente Implementationsbasis, so dass ersten Experimenten zum Compilerbau für eine selbst definierte Programmiersprache eigentlich nichts mehr im Wege stehen sollte ...

Kontextfreie Grammatiken

- Kompositionalität: komplexe Konstituenten entstehen durch *Verkettung* elementarer Konstituenten
 - Regeln: **s** → **np vp**
 - Übersetzung in ein Logikprogramm:


```
s(X) :- np(Y), vp(Z), append(Y,Z,X).
```
 - präterminale Regeln (Wörterbuch): **n** → **frau**
 - Übersetzung in ein Logikprogramm


```
n([frau]).
```

Kontextfreie Grammatiken

• Beispielgrammatik:

```
s(X) :- np(Y),vp(Z),append(Y,Z,X).
np(X) :- d(Y),n(Z),append(Y,Z,X).
vp(X) :- v(X).
```

```
d([die]).
n([frau]).
v([liest]).
```

Kontextfreie Grammatiken

• Generierung:

```
?- s(X).
s(X1):-np(Y1),vp(Z1),append(Y1,Z1,X1).
?- np(Y1),vp(Z1),append(Y1,Z1,X1).
np(X2):-d(Y2),n(Z2),append(Y2,Z2,X2).
?- d(Y2),n(Z2),append(Y2,Z2,X2).
d([die]).
?- n(Z2),append([die],Z2,X2).
n([frau]).
?- append([die],[frau],X2).
?- vp(Z1),append(Y1,Z1,X1).
vp(X3):-v(X3).
?- v(Y3).
v([liest]).
?- append([die,frau],[liest],X1).
X=[die,frau,liest] ;
BT #4
...
BT #1
false.
```

```
#1
succ(X=X1)
#2
succ(X2=Y1)
#3
succ(Y2=[die])
#3a
succ(Z2=[frau])
succ(X2=[die,frau])
#2a
succ(X3=Z1)
#4
succ(X3=[liest])
succ(X1=[die,frau,liest])
```

Kontextfreie Grammatiken

• Erkennung:

```
?- s([die,frau,liest]).
s(X1):-np(Y1),vp(Z1),append(Y1,Z1,X1).
?- np(Y1),vp(Z1),append(Y1,Z1,[die,frau,liest]).
np(X2):-d(Y2),n(Z2),append(Y2,Z2,X2).
?- d(Y2),n(Z2),append(Y2,Z2,X2).
d([die]).
?- n(Z2),append([die],Z2,X2).
n([frau]).
?- append([die],[frau],X2).
?- vp(Z1),append(Y1,Z1,[die,frau,liest]).
vp(X3):-v(X3).
?- v(X3).
v([liest]).
?- append([die,frau],[liest],[die,frau,liest]).
true.
```

```
#1
succ(X1=[die,frau,liest])
#2
succ(X2=Y1)
#3
succ(Y2=[die])
#3a
succ(Z2=[frau])
succ(X2=[die,frau])
#2a
succ(X3=Z1)
#4
succ(X3=[liest])
succ
```

Kontextfreie Grammatiken

- Analysestrategie:
 - top down, Tiefe zuerst, links-rechts
- blinde Suche:
 - Generieren aller durch die Grammatik lizenzierten Sätze und *anschließender* Vergleich mit dem Eingabesatz
 - generate-and-test ohne Bewertung von Zwischenergebnissen
 - keine Steuerung der Verarbeitung durch die Eingabedaten

Kontextfreie Grammatiken

- Mehrdeutigkeit in der Grammatik

```
s(X) :- np(Y),vp(Z),append(Y,Z,X).  
np(X) :- d(Y),n(Z),append(Y,Z,X).  
vp(X) :- v(X).
```

```
d([der]).  
d([die]).  
n([mann]).  
n([frau]).  
v([lacht]).  
v([liest]).
```

Kontextfreie Grammatiken

```

?- s([die,frau,liest]).
  ?- np(Y1),vp(Z1),append(Y1,Z1,[die,frau,liest]).
    ?- d(Y2),n(Z2),append(Y2,Z2,X2).                Y2=[der]
      ?- n(Z2),append(Y2,Z2,X2).                    Z2=[mann]
        ?- append([der],[mann],X2).                  X2=[der,mann]
      ?- vp(Z1),append(Y1,Z1,[die,frau,liest]).
        ?- v(X3).                                    X3=[lacht]
          ?- append([der,mann],[lacht],[die,frau,liest]). fail
          BT: v(X3)                                   X3=[liest]
          ?- append([der,mann],[liest],[die,frau,liest]). fail
          BT: v(X3)
        BT: vp(Z1)
        BT: n(Z2)                                    Z2=[frau]
          ?- append([der],[frau],X2).                  X2=[der,frau]

```

Kontextfreie Grammatiken

```

?- vp(Z1),append(Y1,Z1,[die,frau,liest]).
  ?- v(X3).                                           X3=[lacht]
    ?- append([der,frau],[lacht],[die,frau,liest]). fail
    BT: v(X3)                                         X3=[liest]
    ?- append([der,frau],[liest],[die,frau,liest]). fail
    BT: v(X3)
  BT: vp(Z1)
  BT: n(Z2)
  BT: d(Y2)                                           Y2=[die]
  ?- n(Z2),append(Y2,Z2,X2).                         Z2=[mann]
    ?- append([die],[mann],X2).                      X2=[die,mann]

```

Kontextfreie Grammatiken

```

?- vp(Z1),append(Y1,Z1,[die,frau,liest]).
  ?- v(X3).                                           X3=[lacht]
    ?- append([die,mann],[lacht],[die,frau,liest]). fail
    BT: v(X3)                                         X3=[liest]
    ?- append([die,mann],[liest],[die,frau,liest]). fail
    BT: v(X3)
  BT: vp(Z1)
  BT: n(Z2)                                           Z2=[frau]
    ?- append([die],[frau],X2).                      X2=[die,frau]
  ?- vp(Z1),append(Y1,Z1,[die,frau,liest]).
    ?- v(X3).                                         X3=[lacht]
      ?- append([die,frau],[lacht],[die,frau,liest]). fail
      BT: v(X3)                                       X3=[liest]
      ?- append([die,frau],[liest],[die,frau,liest]). succ
true.

```

Kontextfreie Grammatiken

- Terminierungsprobleme:
 - Aufruf von $np(X)$, $vp(X)$ usw. mit uninstantiierten Variablen
 - keine Terminierung bei rekursiven Regeln
- $np(X) :- np(Y), pp(Z), append(Y, Z, X).$
 $np(X) :- adj(Y), np(Z), append(Y, Z, X).$

CFG mit Differenzlisten

- Idee: Anwendung von Differenzlisten
 - zur effizienten Verkettung der Teillisten
 - zur Steuerung der Analyse durch Eingabesatz
- Konstituentenprädikate über Differenzlisten

$s/1$	$s(?Liste1\?Liste2)$
$np/1$	$np(?Liste1\?Liste2)$
...	

CFG mit Differenzlisten

- Interpretation:
 - $Liste1$: zu analysierende Liste, für die die Top-down-Hypothese s , np , ... überprüft werden soll
 - $Liste2$: Restliste, die von der aktuellen Konstituente s , np , ... nicht überdeckt wird

CFG mit Differenzlisten

- Regeln:
- $s(B\backslash E) :- np(B\backslash M), vp(M\backslash E).$
 $np(B\backslash E) :- d(B\backslash M), n(M\backslash E).$
 $vp(B\backslash E) :- v(B\backslash E).$
- Präterminale Regeln (Lexikon):

$d([der|R]\backslash R).$
 $d([die|R]\backslash R).$
 $n([mann|R]\backslash R).$
 $n([frau|R]\backslash R).$
 $v([lacht|R]\backslash R).$
 $v([liest|R]\backslash R).$

CFG mit Differenzlisten

- Erkennung:

```

?- s([die,frau,liest]\[ ]).          #1
  s(B1\E1):-np(B1\M1),vp(M1\E1).    succ(B1=[die,frau,liest],
                                     E1=[ ])

?- np([die,frau,liest]\M1),vp(M1\[ ]). #2
  np(B2\E2):-d(B2\M2),n(M2\E2).    succ(B2=[die,frau,liest],
                                     E2=M1)

?- d([die,frau,liest]\M2),n(M2\E2). #3
  d([der|R1]\R1).                  fail
  d([die|R1]\R1).                  succ(R1=[frau,liest]=M2)
?- n([frau,liest]\E2).             #3a
  n([mann|R2]\R2).                 fail
  n([frau|R2]\R2).                 succ(R2=[liest]=E2)
?- vp([liest]\E1).                 #2a
  vp(B3\E3):-v(B3\E3).             succ(B3=[liest],E3=E1)
?- v([liest]\E3).                  #4
  v([lacht|R3],R3).                fail
  v([liest|R3],R3).                succ(R3=[ ]=E3)

true.

```

CFG mit Differenzlisten

- gesteuerte Suche:

- Prädikatsaufrufe sind immer mit dem aktuellen Input instanziiert
- keine freie Generierung von Terminalsymbolketten
- Suche beschränkt sich auf die Konstituenten, die auf den vorgegebenen Listenanfang passen

CFG mit Differenzlisten

- Terminierungsprobleme nur noch bei linksrekursiven Regeln

- *das Haus hinter der Straße am Dorfteich*
- *das Haus hinter der Straße mit dem roten Dach*

```
np(B\E) :- np(B\M),pp(M\E).
```

```

pp([mit,dem,roten,dach|R]\R).
pp([hinter,der,strasse|R]\R).
pp([am,bach|R]\R).

```

```
np([das,haus|R]\R).
```

CFG mit Differenzlisten

- Erkennung

```
?- np([das,haus,am,bach]\[ ]).
    np(B1\E1):-np(B1\M1),pp(M1\E1).
                                succ(B1=[das,haus,am,bach],E1=[ ])
?- np([das,haus,am,bach]\M1),pp(M1\[ ]).
    np(B2\E2):-np(B2\M2),pp(M2\E2).
                                succ(B2=[das,haus,am,bach],E2=M1)
?- np([das,haus,am,bach]\M2),pp(M2\E2).
    np(B3\E3):-np(B3\M3),pp(M3\E3).
                                succ(B3=[das,haus,am,bach],E3=M2)
. . .
```

Definite Clause Grammar

- Operatornotation:

```
-->/2          +Struktur --> +Ziel
```

- syntaktischer Sonderstatus: Operator wird beim Einlesen sofort in die Differenzlistennotation transformiert

externe DCG-Notation	interne Prolog-Darstellung
s --> np, vp.	s(B,E):-np(B,M),vp(M,E).
v --> [liest].	v([liest E],E).

Definite Clause Grammar

- Grammatik in DCG-Notation

```
s(S) :- s(S,[ ]).
```

```
s --> np, vp.
np --> d, n.
vp --> v.
```

```
d --> [die].
n --> [frau].
v --> [liest].
```

Definite Clause Grammar

- Syntax:

DCG-Regel ::= Struktur [Terminal] ' --> ' Produktion
 Produktion ::= Struktur | Terminal |
 Produktion (',' | ';') Produktion |
 '{' Ziel '}' | '!'
 Terminal ::= Liste

- Zusätzliche Ausdrucksmöglichkeiten:

- beliebige Terme als Nichtterminalsymbole → Augmentation
- beliebige Listen als Terminalsymbole
- Listen von Termen als Terminalsymbole
- Anreicherung der Grammatikregeln mit zusätzlichen Bedingungen
- Steuerelemente in Grammatikregeln
- rechtsseitig kontextsensitive Regeln

Tatsächlich haben wir mit den DCG-Regeln einen weitaus mächtigeren Formalismus geschaffen, als ihn uns die Regeln einer kontextfreien Grammatik bieten. Wir können etwa

- durch Augmentierung zusätzliche Bedingungen an die Verträglichkeit von Nichtterminalsymbolen stellen,
- durch die Verwendung eingebetteter Strukturen (vgl. PEANO-Zahlen!) Grammatiken mit unendlich vielen Nichtterminalsymbolen schreiben (wodurch wir allerdings die Definitionsbasis der generativen Grammatiken und damit auch den Bereich der kontextfreien Sprachen verlassen!),
- durch die konstruktive Verwendung der Unifikation beliebige Strukturbeschreibungen erzeugen, die mit der zugrundeliegenden Regelstruktur keinerlei Gemeinsamkeiten mehr aufweisen müssen und vieles Andere mehr ...

Erweiterungen

- Augmentation (1): Zusätzliche Bedingungen an die Verträglichkeit von (Nicht-) Terminalsymbolen durch Koreferenz
 - Typverträglichkeit
 - Kongruenz, Rektion, ...
- Augmentation (2): Erzeugung von Strukturbeschreibungen auf einem zusätzlichen Argument
- einfachster Fall: Strukturbeschreibung reflektiert die Regelstruktur

```
s(S,SB) :- s(SB,S,[ ]).
```

```
s(s(Snp,Svp)) --> np(Snp), vp(Svp).
np(np(Sd,Sn)) --> d(Sd), n(Sn).
vp(vp(Sv,Snp)) --> v(Sv), np(Snp).
vp(vp(Sv)) --> v(Sv).
```

Implementation von DCG

- Augmentierung auch im Lexikon möglich

```
d(d(die)) --> [die].
d(d(das)) --> [das].
n(n(frau)) --> [frau].
n(n(buch)) --> [buch].
v(v(liest)) --> [liest].
```

- Testaufruf

```
?- s([die,frau,liest,das,buch],SB).
    SB = s(np(d(die), n(frau)),
           vp(v(liest), np(d(das), n(buch)))).
```

Parsing

- Erzeugung der Strukturbeschreibungen kann auch unabhängig von der Regelstruktur sein
 - linksrekursive Regeln, die eine rechtsrekursive Struktur erzeugen
 - Vermeiden von Terminierungsproblemen
 - Integration von Syntaxanalyse und semantischer Interpretation bzw. Codegenerierung
 - Integration von Syntaxanalyse und strukturellem Transfer

Definite Clause Grammar

- Verwendung des Regelkopfseparators $-->$ auch zur Verkettung sequentieller Prädikatsaufrufe
- "Durchschleifen" von Zwischenergebnissen

$a --> b, c, d.$

entspricht

$a(A, B) :- b(A, X), c(X, Y), d(Y, B).$

Grammatiken höherer Ordnung

- Spezifikation genereller Transducer
 - String-to-String
 - String-to-Tree
 - Tree-to-String
 - Tree-to-Tree
- Indizierte Grammatiken: Variable können beliebige Strukturen übermitteln
 - Verwendung von Stacks
 - erzeugt verschiedene Instanzen eines Nichtterminalsymbols
 - Grammatiken mit unendlich vielen Nichtterminalsymbolen
 - akzeptiert/generiert Sprachen höherer Ordnung
 - * "kontextfreie" Regeln \rightarrow kontextsensitive (genauer: indizierbare) Sprachen
 - * "reguläre" Regeln \rightarrow kontextfreie Sprachen

Grammatiken höherer Ordnung

- Extremfall Metamorphosis Grammar: linke Regelseite kann beliebige Sequenz aus Nichtterminal- und Terminalsymbolen sein
 - \rightarrow volle Turingmächtigkeit

Grammatikmodellierung

- Beispiel: kontextfreie Sprache mit "regulären" Regeln

- Sprache: $a^i b^j$ mit $j \geq i$
- kontextfreie Grammatik

s --> s1.	s
s --> s, b.	s b
s1 --> a, b.	s1 b
s1 --> a, s1, b.	a s1 b b
a --> [a].	a a b b b
b --> [b].	

Grammatikmodellierung

- Beispiel: kontextfreie Sprache mit "regulären" Regeln
 - "indizierte" reguläre Grammatik

string(X) --> bs(X).	a string(s(s(0)))
string(X) --> a, string(s(X)).	a a string(s(s(s(0))))
bs(s(X)) --> b, bs(X).	a a bs(s(s(s(0))))
bs(X) --> b, bs(X)	a a b bs(s(s(0)))
bs(0) --> [].	a a b b bs(s(0))
	a a b b b bs(0)
	a a b b b b bs(0)
	a a b b b b b

- konsequente links-rechts-Verarbeitung!

11 Fazit und Ausblick

11.1 Prädikatenlogische Grundlagen

Horn-Logik

- Prolog-Klauseln stellen die einfachste Form prädikatenlogischer Formeln dar
- Klauseln: allquantifizierte Disjunktion von Literalen ohne freie Variable
- Skopus der Quantoren ist die gesamte Klausel

$$A_1 \vee \dots \vee A_n \vee \neg B_1 \vee \dots \vee \neg B_m$$

$$(A_1 \vee \dots \vee A_n) \vee \neg(B_1 \wedge \dots \wedge B_m)$$

$$(A_1 \vee \dots \vee A_n) \leftarrow (B_1 \wedge \dots \wedge B_m)$$

- Horn-Klauseln (definite Klauseln): Klausel mit höchstens einem positiven Literal

$$A \vee \neg B_1 \vee \dots \vee \neg B_m$$

$$A \leftarrow (B_1 \wedge \dots \wedge B_m)$$

- keine Existenzquantoren
können nur über Konstanten simuliert werden (Skolemisierung)

Spezielle Horn-Klauseln

a) mit einem positiven Literal

a1) mit wenigstens einem negativen Literal: Prolog-Regel

$$A \vee \neg B_1 \vee \dots \vee \neg B_m \Leftrightarrow A \leftarrow (B_1 \wedge \dots \wedge B_m)$$

a2) ohne negative Literale: Prolog-Fakt

$$A \Leftrightarrow A \vee \mathbf{F} \Leftrightarrow A \leftarrow \mathbf{T}$$

b) ohne positives Literal

b1) mit wenigstens einem negativen Literal: Ziel

$$\mathbf{F} \vee \neg B_1 \vee \dots \vee \neg B_m \Leftrightarrow \mathbf{F} \leftarrow (B_1 \wedge \dots \wedge B_m)$$

b2) ohne negative Literale: leere Klausel (\square)

$$\mathbf{F} \vee \mathbf{F} \Leftrightarrow \mathbf{T} \leftarrow \mathbf{F}$$

Resolution

- Deduktion: Ableiten von Theoremen aus gegebenen Axiomen

- Übertragung auf die Programmierung: Unter welchen Bedingungen (mit welchen Variablenbelegungen) kann ein Theorem G (Ziel) aus den gegebenen Axiomen \mathcal{M} (Regeln und Fakten) abgeleitet werden? \rightarrow inverse Zielstellung

Resolution

- Beweisidee:

$$G = P_1 \wedge \dots \wedge P_n$$

$$\neg(\neg P_1 \vee \dots \vee \neg P_n \vee \mathbf{F})$$

$$\neg(P_1 \wedge \dots \wedge P_n \rightarrow \mathbf{F})$$

$$\neg(G \rightarrow \mathbf{F})$$

- Widerlegungsbeweis: $ag(G) \leftrightarrow \neg ef(\neg G)$
- Beweistechnik: $\mathcal{M} \cup \{\neg G\} \models \mathbf{F}$

– Ableiten der leeren Klausel \square

Resolution

- Resolution: allgemeines Deduktionsverfahren für Klausel-Normalform
- Resolutionsschritt: Zusammenfassen zweier Klauseln, so daß sich positive und negierte Literale gegenseitig aufheben

a) Spezialfall: Widerlegung durch Faktenidentifizierung

$$\frac{\neg P}{P} \quad \frac{\mathbf{F} \leftarrow P}{P \leftarrow \mathbf{T}}$$

$$\square \quad \mathbf{F} \leftarrow \mathbf{T}$$

b) Resolutionsregel

$$\frac{\begin{array}{l} A_1 \vee \dots \vee A_n \leftarrow B_1 \wedge \dots \wedge P \dots \wedge B_m \\ C_1 \vee \dots \vee P \dots \vee C_k \leftarrow D_1 \wedge \dots \wedge D_l \end{array}}{A_1 \vee \dots \vee A_n \vee C_1 \vee \dots \vee C_k \vee \leftarrow B_1 \wedge \dots \wedge B_m \wedge D_1 \wedge \dots \wedge D_l}$$

Resolution

- Resolutionsregel für Horn-Klauseln

$$\frac{\begin{array}{l} A \leftarrow B_1 \wedge \dots \wedge P \dots \wedge B_m \\ P \leftarrow D_1 \wedge \dots \wedge D_l \end{array}}{A \leftarrow B_1 \wedge \dots \wedge B_m \wedge D_1 \wedge \dots \wedge D_l}$$

- das Resultat eines Resolutionsschrittes ist wieder eine Horn-Klausel

Resolution

- Auswahl des zu resolvierenden Literals:
 - L-Resolution (linear resolution): es wird immer die im vorangehenden Resolutionsschritt ermittelte Resolvente weiterverarbeitet
 - SL-Resolution (selective linear resolution): mit einer Auswahlfunktion wird immer eine resolvierbare Klausel ausgesucht
 - SLD-Resolution (selective linear resolution for definite clauses): SL-Resolution für Horn-Klauseln

Resolution

- Kopplung von Resolution und Unifikation

$$\frac{\left. \begin{array}{l} A \leftarrow B_1 \wedge \dots \wedge P_1 \wedge \dots \wedge B_m \\ P_2 \leftarrow D_1 \wedge \dots \wedge D_l \end{array} \right\} \text{ mit } P_2 = \sigma(P_1)}{\sigma(A) \leftarrow \sigma(B_1) \wedge \dots \wedge \sigma(B_m) \wedge \sigma(D_1) \wedge \dots \wedge \sigma(D_l)}$$

Negation

- closed world assumption (CWA): Metaregel zur Inferenz negierter Grundatome

$$\neg(\mathcal{M} \models \mathcal{G}) \rightarrow \mathcal{M} \models_{CWA} \neg \mathcal{G}$$

- negation as finite failure (NFF): Metaregel, die die Negation eines Grundatoms auf das Scheitern seiner Ableitbarkeit in endlich vielen Schritten zurückführt

$$\neg G \left\{ \begin{array}{ll} \text{fail} & \mathcal{M} \models_{SLD} \mathcal{G} \\ \text{success} & \neg(\mathcal{M} \models_{SLD} \mathcal{G}) \end{array} \right.$$

- keine Variablenbindung in negierten Ausdrücken möglich
- "Floundering" für Atome mit nichtinstanzierten Variablen
- SLDNF-Resolution: SLD-Resolution mit NFF-Regel

11.2 Logikprogrammierung, wozu?

Logikprogrammierung, wozu?

- Verarbeitungsmodell, Programmierstil und Programmiersprache bedingen einander
- jedoch keine strikten Abhängigkeiten:
 Problem →
 Verarbeitungsmodell →
 Programmierstil →
 Programmiersprache
- Verarbeitungsmodell ist primär, Programmiersprache ist sekundär
- Welches Verarbeitungsmodell ist angemessen für ein Problem?

Verarbeitungsmodell

- Klauselabarbeitung durch Resolution
 - Nichtdeterminismus
 - Ersetzungsregeln
 - Rekursion
- Unifikation
 - Pattern-Matching für Termstrukturen
 - Listenverarbeitung
- Eignung für komplex strukturierte Symbolverarbeitungsprobleme
- Verarbeitungsmodell ist sehr flexibel
 - veränderte Resolutionsstrategien
 - erweiterte Unifikation

Programmierstil

- relationale Programmierung
 - Ergebnismengen statt Einzelergebnisse
 - Richtungsunabhängigkeit der Berechnung
 - Betonung rekursiver Problemlösungen
 - im Kernbereich deklarativ: referentielle Transparenz ist gewahrt

- Zurückdrängen prozeduraler Aspekte: höherer Abstraktionsgrad
- nichtdeklarative Komponenten in isolierten Bereichen (Arithmetik, Suchraummanipulation, extralogische Prädikate)

Programmierstil

- relationale Programmierung als Ideal?
 - simples Skopusmodell
 - Seiteneffekte als Fehlerquellen zurückgedrängt
- aber:
 - nichtrelationaler Charakter der Arithmetik
 - hoher Abstraktionsgrad erschwert Verständlichkeit
 - relativ schlechte Kompatibilität mit objektorientierter Programmierung

Programmierstil

- Prolog erlaubt unterschiedliche Programmierstile:
- funktionale Programmierung:
 - funktionale Auswertung in der Arithmetik
 - eindeutige Berechnungsergebnisse: `cut`
 - Simulation der Funktionskomposition durch Sequenzen von Prädikatsaufrufen
- zustandsorientierte (imperative) Programmierung:
 - Klauseln als Speicherzellen: `assert/retract`
 - nichtmonotone Steuerkonstrukte: `cut`
- objektorientierte Programmierung:
 - Informationskapselung: über das Modulsystem
 - Vererbung: im Bereich der statischen Wissensrepräsentation
 - schwache Kopplung mit einem Objektsystem: XPCE

Programmiersprache

- Verarbeitungsmodell der Logikprogrammierung enthält im Kern bereits sehr leistungsfähige Basiskomponenten (Suche, Unifikation)
- flexible Syntax
- flexible Semantik
- gut geeignet für Experimente mit neuen Verarbeitungsmodellen → rapid prototyping für Funktionsprototypen

11.3 Erweiterungen

Erweiterungen

- Integration funktionaler Programmierung
- Strikt deklarative Logikprogrammierung
- Constraint-Lösen
- Deduktive Datenbanken
- Graphunifikation
- Induktive Logikprogrammierung

Integration funktionaler Programmierung

- Problem: funktionale Auswertung zur Zeit nur in spezieller Umgebung
 - extralogisches Konstrukt → keine relationale Auswertung möglich
- Ziel: vollständige Integration in die Unifikation
- prinzipieller Lösungsansatz: erweiterte Unifikation für zwei Atome A und B
 - Disagreement-Menge D : enthält die nicht unifizierbaren Bestandteile von A und B , wenn diese als Funktionensymbol definiert sind

$$p(+ (2, x)) \sqcup p(+ (3, y)) \Rightarrow D = + (2, x), + (3, y)$$

- funktionale Evaluation der Elemente der Disagreement-Menge

Strikt richtungsunabhängige Logikprogrammierung

- Lösungsidee: Klauseln werden nicht mehr interpretativ abgearbeitet
 - Deklaration von Verarbeitungsmodi (Instanziierungsvarianten) und Determinismus-Modi
 - *Compilation* in ausführbaren Code für die verschiedenen Aufrufvarianten
 - sequentielle Abhängigkeiten werden separat verwaltet
 - → Mercury (University of Melbourne)
- Deklaration der Verarbeitungsmodi

```
:- pred append(list(T), list(T), list(T)).
:- mode append(in, in, out) is det.
:- mode append(in, out, in) is semidet.
:- mode append(out, in, in) is semidet.
:- mode append(out, out, in) is multi.
```

Strikt richtungsunabhängige Logikprogrammierung

- Problem: Reihenfolgeabhängigkeit bei seiteneffektbehafteten Teilzielen (z.B. Input/Output)
- Lösungsansatz: explizite Verwaltung von Berechnungszuständen (state of the world)
 - Hinzufügen von zwei Zustandsparametern zu jedem IO-Prädikat
 - explizite Modellierung von Sequenzbeziehungen
 - analog zur Differenzlistentechnik

Constraint-Lösen

- Problem: Instanziierung von Teilzielen über großen Faktenmengen führt zu ineffizientem Code: trial-and-error
- Ziel: möglichst frühzeitige Bedingungsüberprüfung schon bei der Variablenbindung und nicht erst bei der Auswertung des Prozedurkörpers

```
?- X in 1 .. 5, Y in 3 .. 7, X?>Y,
    indomain(X).
X = 4, Y = 3 ;
X = 5, Y = 3 ;
X = 5, Y = 4.
```

Constraint-Lösen

- Lösungsansatz: Constraint-Satisfaction
- effiziente Lösungen bekannt, aber nur für Teildomänen
 - Gleitkommazahlen , Rationalzahlen: Intervallarithmetik, lineare Gleichungen/Ungleichungen
 - Ganze Zahlen: Intervalle, Folgen, Aufzählungen, Gleichungen, Ungleichungen, negierte Gleichungen
 - Spezialfälle: endliche symbolische Domänen, Boolesche Domänen

Constraint-Lösen

- 1. Schritt: Erweiterte relationale Programmierung
 - auch für numerische Aufgaben

```
peano(0,0).  
peano(X,s(Y)):-  
    X?>0,  
    peano(X-1,Y).
```

- 2. Schritt: Constraint Logic Programming
 - Constraint-Gleichungen sind Resultat der Unifikation
 - Gleichungen sind a priori nicht unbedingt vollständig bekannt
 - werden erst über mehrere Unifikationen hinweg angereichert

Deduktive Datenbanken

- Problem 1: Prädikatenkalkül ist nur semi-entscheidbar
→ Terminierungsprobleme
- Ziel: Terminierungssicherheit auch für Datenbanken mit Inferenzfähigkeit
- Lösungsansatz:
 - faktengesteuerte Inferenztechniken (OLDT-Resolution, magic sets)
 - Vereinfachung: keine komplexen Terme an Argumentpositionen (keine Funktionensymbole)
- DATALOG:
 - keine Rekursion über Datenstrukturen (nur über der Datenbasis)
 - vereinfachte Unifikation
 - effiziente Indexierung

Deduktive Datenbanken

- Problem 2: unvollständige Behandlung der Negation
- Ziel: Variablenbindungen für negierte Anfragen

```
amount(i_134,2487.28).
amount(i_147,34.00).
amount(i_165,1835.90).
recipient(i_134,meier).
recipient(i_147,schulze).
recipient(i_165,mueller).
paid(i_147).
```

```
?- not(paid(Inv)), amount(Inv,Amount),
    recipient(Inv,To).
```

Deduktive Datenbanken

- Lösungsansatz: Sortenbeschränkung für die Argumente

```
invoice(i_134).
invoice(i_147).
invoice(i_165).
```

```
?- invoice(Inv), not(paid(Inv)),
    amount(Inv,Amount), recipient(Inv,To).
```

```
I=i_134, Amount=2487.28, To=meier ;
I=i_165, Amount=1835.90, To=mueller.
```

Deduktive Datenbanken

- a) direkter Einbau von Sortenbeschränkungen in die Unifikation (sortierte Logik)
- b) explizite Einschränkung der zulässigen Sorten an den Argumentstellen

```
amount(X::invoice,Y::amount_of_money)
recipient(X::invoice,Y::company)
paid(X::invoice)
```

- Problem 3: Integritätsconstraints (Gleichungen und Ungleichungen)

Graphunifikation

- Problem: Stelligkeit ist Bestandteil der Prädikatsdefinition
→ unübersichtliche Programme bei stark unterspezifizierter Modellierung (z.B. Grammatiken für natürliche Sprache)
- Ziel: Beschränkung auf die Angabe der jeweils relevanten Information
→ Merkmalstrukturen sind "seitlich erweiterbar"

$$\left[\begin{array}{cc} a & \boxed{1} \\ b & \boxed{1} \end{array} \left[\begin{array}{cc} c & \boxed{2} \\ d & \boxed{2} \end{array} \right] \right] \quad \left[\begin{array}{cc} a & \boxed{1} \end{array} \left[\begin{array}{cc} b & \boxed{1} \end{array} \right] \right]$$

Graphunifikation

- Subsumtion: A subsumiert B gdw. jeder Pfad aus A auch in B enthalten ist
- Unifikation: der Unifikator von A und B ist die allgemeinste Merkmalstruktur, die sowohl von A als auch von B subsumiert wird

$$\left[\begin{array}{cc} \text{cat} & n \\ \text{agr} & \left[\begin{array}{cc} \text{cas} & \text{nom} \end{array} \right] \end{array} \right] \sqcup \left[\begin{array}{cc} \text{cat} & n \\ \text{agr} & \left[\begin{array}{cc} \text{gend} & \text{fem} \\ \text{num} & \text{sg} \end{array} \right] \end{array} \right] = \left[\begin{array}{cc} \text{cat} & n \\ \text{agr} & \left[\begin{array}{cc} \text{cas} & \text{nom} \\ \text{gend} & \text{fem} \\ \text{num} & \text{sg} \end{array} \right] \end{array} \right]$$

Graphunifikation

- Implementation von DAGs als offene Listen

```
:- op(500, xfy, :).
```

```
fs1([cat : n,
     agr : [cas : nom | _]
     | _]).
fs2([cat : n,
     agr : [gend : fem,
            num : sg | _]
     | _]).
```

Graphunifikation

- Implementation der Unifikation

```
unify(Dag,Dag) :- !.
    % identische DAGs unifizieren

unify(Path:Value|Dags1,Dag) :-
    % Dag1:Path:Value unifiziert
    pathval(Dag,Path,Value,RemDags),
    % mit Dag2:Pfad:Value
    unify(Dags1,RemDags).
    % Rest-DAGs unifizieren
```

Graphunifikation

```

pathval(Dag1,Feature:Path,Value,Dags) :- !,
    % Pfad ist Sequenz aus Merkmalen
    pathval(Dag1, Feature, Dag2, Dags),
    % Zerlegung des Pfades: Teil 1
    pathval(Dag2, Path, Value, _).
    % Zerlegung des Pfades: Teil 2

pathval([Feature:Value1|Dags], Feature, Value2, Dags) :- !,
    % relevantes Merkmal
    unify(Value1,Value2).
    % gefunden und unifiziert

pathval([Dag|Dags1],Feature,Value,[Dag|Dags2]) :-
    % Rekursiver Abbau des Graphen
    pathval(Dags1,Feature,Value,Dags2).

```

Graphunifikation

```

?- fs1(F1),fs2(F2),unify(F1,F2).
F1 = [cat:n, agr:[cas:nom, gend:fem, num:sg|_|_|_]
F2 = [cat:n, agr:[gend:fem, num:sg, cas:nom|_|_|_]

```

- Effizienzverbesserung durch Typisierung

Induktive Logikprogrammierung

- Logikprogramme werden aus Beispieldaten und Hintergrundwissen generalisiert
- Kombination mit Techniken des Maschinellen Lernens
- Beispieldaten, Hintergrundwissen und Lernresultate werden als Logikprogramme beschrieben
- Anwendungen
 - Fehlerdiagnoseregeln
 - Struktur-Wirkungs-Regeln für Medikamente
 - Regeln zur Vorhersage der Sekundärstruktur von Proteinen

Index

- PEANO-Arithmetik, 69
- PEANO-Term, 70
- +
 - /0, 133
- aggregate_all/3, 151
- aggregate_all/4, 152
- app/3, 104
- append/3, 104
 - mit Differenzlisten, 195
- bagof/3, 150, 151
- call/1, 140
- fak/2, 84
- findall/3, 149
- first/2, 100
- flatten/2, 113
 - mit Differenzlisten, 196
- forall/2, 147
- foreach/2, 148
- ggt/3, 90
- in_list/2, 101
- length/2, 103
- member/2, 102
- once/1, 142
- prefix/2, 109
- quicksort/2, 116
 - mit Differenzlisten, 198
- reverse/2, 110
 - mit Differenzlisten, 197
- sublist/2, 109
- suffix/2, 108
- Ablaufprotokoll, 32
- Ableitung, 47
- Accessor, 69
- aktive Datenstrukturen, 188
- Algorithmenschema, 41
- alternative Variablenbindungen, 37
- Anfrage, 30
- Anfrage mit Variablen, 33
- Anfragen
 - komplexe, 41
- Anfragen über mehrere Relationen, 44
- anonyme Variable, 43
- Arithmetik, 69
- Aussagenlogik, 47
- Auswertung
 - funktionale, 172
- Auswertungsumgebung, arithmetische, 82
- Automat
 - endlicher, 119
- Axiome, 47
- Bäume als Listen, 124
- Backtracking, 37, 42
- Baum, 32
 - sortierter, 118
- Baumstruktur, 65
- benannte Funktion, 166
- Bezugstransparenz, 33, 40
- Bindung einer Variablen, 34
- closed world assumption, 212
- Closure, 177
- Constraint-Lösen, 216
- cut, 133
- Datalog, 217
- Datenbank
 - relationale, 27
- Datenbanken
 - deduktive, 217
- Datenbankmanipulation, 139
- Datenbasis, 29
- Datenstruktur
 - rekursive, 65
- Datenstrukturen
 - aktive, 188
 - dynamische, 188
 - unterspezifizierte, 74
- Deduktion, 47
- Deduktive Datenbanken, 217
- Defaultschließen, 136
- Definite Clause Grammar, 205
- Deklarativität, 33
- Differenzlisten, 192
- Disjunktion, 42, 141
- Domäne einer Relation, 25

- dynamische Datenstrukturen, 188
- Element einer Liste, 101
 - erstes, 100
- Endlicher Automat, 119
- Endrekursion, 73
- Entfalten, 197
- Environment, 167
- extensionale Spezifikation einer Relation, 25
- failure-gesteuerte Zyklen, 146
- Fakten, 27
- Fakultät, 84
- Funktion, 166
 - benannte, 166
 - primitive, 176
 - rekursive, 179
- funktional spezifizierte Relation, 60
- funktionale Applikation, 172
- funktionale Auswertung, 172
- Funktionale Programmierung, 160
- Funktionen höherer Ordnung, 181
- Funktionsauswertung, 166
- Gültigkeitsbereich, 34
- globale Variable, 139
- Goal, 30
- größter gemeinsamer Teiler, 90
- Grammatiken
 - kontextfreie, 199
- Graphunifikation, 218
- green cut, 134
- Grundstruktur, 28
- Horn-Klausel, 210
- if-then-else, 145
- indirekte Rekursion, 75
- Induktive Logikprogrammierung, 220
- Infixoperator, 79
- Informationsanreicherung, 41
- Instanziierung, 34
- Instanziierungsvarianten, 34
- Integration funktionaler Programmierung, 215
- intensionale Spezifikation einer Relation, 47
- Iteration, 147
- Join, 45
- Kellerspeicher, 123, 189
- Klausel, 28, 210
- Kommentare, 32
- komplexe Anfragen, 41
- Konditional, 145
- Konjunktion, 41
- Konstruktor, 69
- kontextfreie Grammatiken, 199
- Koreferenz, 40, 44
- Kreuzprodukt, 25
- Länge einer Liste, 103
- Löschen von Listenelementen, 112
- Lösungsalternativen, 37
- Lambda-Ausdruck, 166
- Liste
 - Element einer, 101
 - Länge einer, 103
- Listen, 96
 - offene, 189
 - Verketten zweier, 104
 - verkettete, 96
- Listenunifikation, 98
- Logikprogrammierung, 41
- Mehrtabellenanfragen, 44
- Memoization, 119
- Metacall, 141
- Metainterpreter, 153
- Metaprogrammierung, 153
- modus ponens, 47, 49
- monotones Schließen, 137, 144
- Negation, 142
- nichtmonotones Schließen, 137, 144
- offene Listen, 189
- Operator, 78
- Postfixoperator, 79
- Prädikat, 29
- Prädikate höherer Ordnung, 140
- Prädikate zweiter Ordnung, 45
- Prädikatenlogik, 47

- Prädikatsschema, 29, 32
- Präfix einer Liste, 109
- Präfixoperator, 79
- primitive Funktion, 176
- Programmierstil, 22, 213
- Programmierung, relationale, 41
- Projektion, 45
- Prozedur, 29

- Queue, 123, 190

- red cut, 134
- referentielle Transparenz, 33, 40
- reflexive Relation, 60
- Regel, 48
 - Abarbeitung einer, 49
- Regelkörper, 48
- Regelkopf, 48
- Rekursion, 57
 - indirekte, 75
 - verzweigende, 113
- Rekursionsabschluss, 70
- Rekursionsschritt, 70
- rekursive Datenstruktur, 65
- rekursive Definition eines Prädikats, 70
- rekursive Funktion, 179
- Relation, 25
 - funktional spezifizierte, 60
 - reflexive, 60
 - symmetrische, 54
 - transitive, 57
- relationale Datenbank, 27
- Relationenalgebra, 45
- Resolution, 210
- Restlistenseparator, 96
- Resultatsaggregation, 149
- Reversibilität, 41
- Richtungsunabhängigkeit, 41

- s-Ausdrücke, 165
- Scheme, 161
- Schließen
 - monotones, 144
 - nichtmonotones, 144
- Schlussregel, 47
- Selbstbezüglichkeit, 57
- Selektion, 45

- Semantik
 - denotationelle, 30
 - operationale, 30
- Sichtbarkeitsbereich, 34
- soft cut, 145
- Sortieren, 114
- sortierter Baum, 118
- special form expression, 172
- special form expressions, 165
- SQL, 45
- Stack, 123, 189
- Stelligkeit einer Relation, 28
- Steuerstrukturen, 145
- Strikt richtungsunabhängige Logikprogrammierung, 215
- String, 127
- Struktur, 28
- strukturelle Gleichheit, 132
- strukturelle Identität, 131
- Suche, 30, 32
- Suche bei konjunktiven Anfragen, 42
- Suchraum, 32
- Suchraummanipulation, 133
- Suchraumverwaltung, 122
- Suchstrategie, 42
- Suchstrategien, 122
- Suffix einer Liste, 108
- symmetrische Relation, 54
- Syntaxbaum, 67

- Teilliste einer Liste, 109
- Teilziele, 41
- Test eines Prädikats, 38, 50, 77, 78, 94
- Testoperator, 69
- Tiefensuche, 42
- Trace, 32
- transitive Relation, 57
- Transparenz
 - referentielle, 40
- Trie, 124
- Typkonversion, 128

- Umdrehen einer Liste, 110
- Umgebung, 167
- Unfolding, 197
- Unifikation, 31, 68
- Unterspezifikation, 35, 40

unterspezifizierte Datenstrukturen, 74

Variable, 33

 anonyme, 43

 globale, 139

Variablenbindung, 34

Variablenbindungen

 alternative, 37

Variableninspektion, 130

Verarbeitungsmodell, 213

 constraint-basiertes, 12

 funktionales, 10

 imperatives, 10

 logik-basiertes, 11

 objektorientiertes, 12

Verarbeitungsrichtung, 41

Verketteten zweier Listen, 104

verkettete Listen, 96

Verschattung, 168

verzweigende Rekursion, 113

Warteschlange, 123, 190

Wertesemantik, 162

Widerlegungsbeweis, 211

Zeichenketten, 127

Ziel, 30

Zurücksetzen, 37

Zusicherungen, 32

Zyklen

 failure-gesteuert, 146

Zyklenvermeidung, 120